Graph - 2

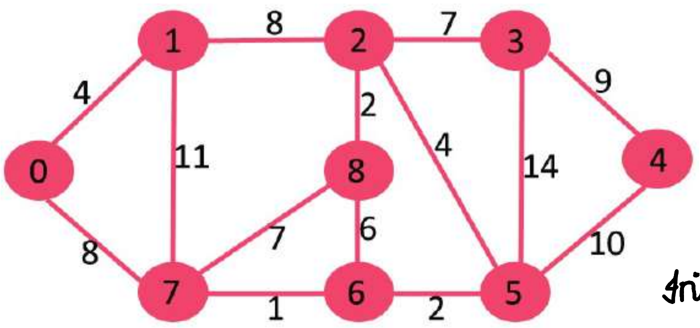
- Karun Karthik

content

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)

src = 1

cost array = min cost from src to every other vertex

Initially cost =

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

vis = { }

→ As it is weighted graph, we'll use priority queue (PQ) instead of normal queue. & 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.

✓

(1,0)

vis = {1}

cost[1] = 0

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

~~1,0~~ (0,4) (7,11) (2,8)

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

⇒ ~~1,0~~ (0,4) (7,11) (2,8)

vis = {1,0}

Cost[0] = 4

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

~~1,0~~ ~~0,4~~ (7,11) (2,8) (7,12)

→ lowest cost is 8 ∴ pop & push its neighbours

⇒ ~~1,0~~ ~~0,4~~ (7,11) (2,8) (7,12)

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

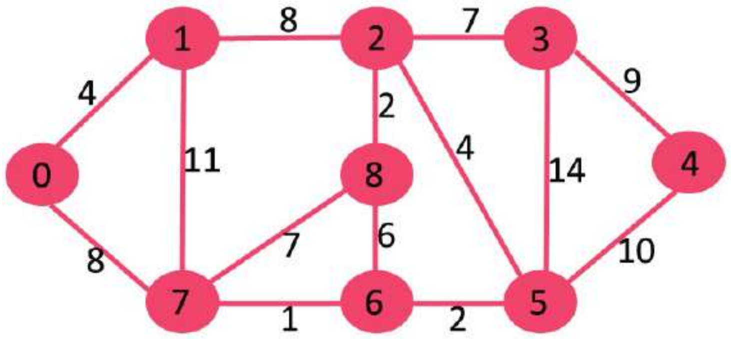
∴ push

vis = {1,0,2}

cost[2] = 8

~~1,0~~ ~~0,4~~ (7,11) ~~2,8~~ (7,12) (8,10), (3,15), (5,12)

→ lowest cost is 10 ∴ pop & push its neighbours



⇒

~~(1,0)~~ ~~(0,4)~~ (7,11) ~~(2,8)~~ (7,12) (8,10), (3,15), (5,12)

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

∴ push

vis = {1,0,2,8}

cost[8] = 10

~~(1,0)~~ ~~(0,4)~~ (7,11) ~~(2,8)~~ (7,12) ~~(8,10)~~, (3,15), (5,12), (7,17), (6,16)

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

⇒

~~(1,0)~~ ~~(0,4)~~ (7,11) ~~(2,8)~~ (7,12) ~~(8,10)~~, (3,15), (5,12), (7,17), (6,16)

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

& (6,12) ∴ push

vis = {1,0,2,8,7}

Cost[7] = 11

~~(1,0)~~ ~~(0,4)~~ ~~(7,11)~~ ~~(2,8)~~ (7,12) ~~(8,10)~~, (3,15), (5,12), (7,17), (6,16), (6,12) ≃ (7,12) (3,15), (5,12), (7,17), (6,16), (6,12)

↳ lowest cost = 12

∴ Anything among 5,6 can be selected & pop & push its neighbours

Not 7, because its already visited & cost is < 12.

⇒

(3,15), (5,12), (7,17), (6,16), (6,12)

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

vis = {1,0,2,8,7,5}

cost[5] = 12

(3,15), ~~(5,12)~~, (7,17), (6,16), (6,12), (4,22), (3,26), (6,14)

→ lowest cost = 12 ∴ pop & push its neighbours

⇒ (3,15), (7,17), (6,16), (6,12), (4,22), (3,26), (6,14)

⇒ neighbours of 6 = 5,7,8 are visited.

∴ no push

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

cost[6] = 12

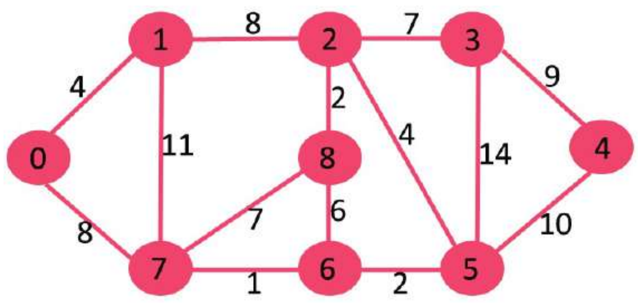
(3,15), (7,17), (6,16), ~~(6,12)~~, (4,22), (3,26), (6,14)

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

(3,15), (7,17), (6,16), (4,22), (3,26), ~~(6,14)~~

∴ Next lowest is 15, ∴ pop & push it's neighbours

https://www.linkedin.com/in/karun-karthik



⇒ (3,15), (7,17), (6,16), (4,22), (3,26) ⇒ neighbours of 3 = 2,5 (visited) (4,24) ∴ push

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

cost[3] = 15

(~~3,15~~), (7,17), (6,16), (4,22), (3,26), (4,24) → next lowest cost = 16 but 6 is already visited ∴ pop

(7,17), (~~6,16~~), (4,22), (3,26), (4,24)

→ next lowest cost = 17 but 7 is already visited ∴ pop

⇒ (4,22), (3,26), (4,24) → next lowest cost = 22 ∴ pop & push its neighbours

⇒ (4,22), (3,26), (4,24) ⇒ neighbours of 4 = 3,5 (visited) ∴ no push

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

cost [4] = 22

(~~4,22~~), (3,26), (4,24) → next lowest cost = 24 but 4 is already visited ∴ pop

(3,26), (~~4,24~~) → next lowest cost = 26 but 3 is already visited ∴ pop ⇒ (~~3,26~~) ⇒ ∴ empty PQ.

Answer ⇒ cost =

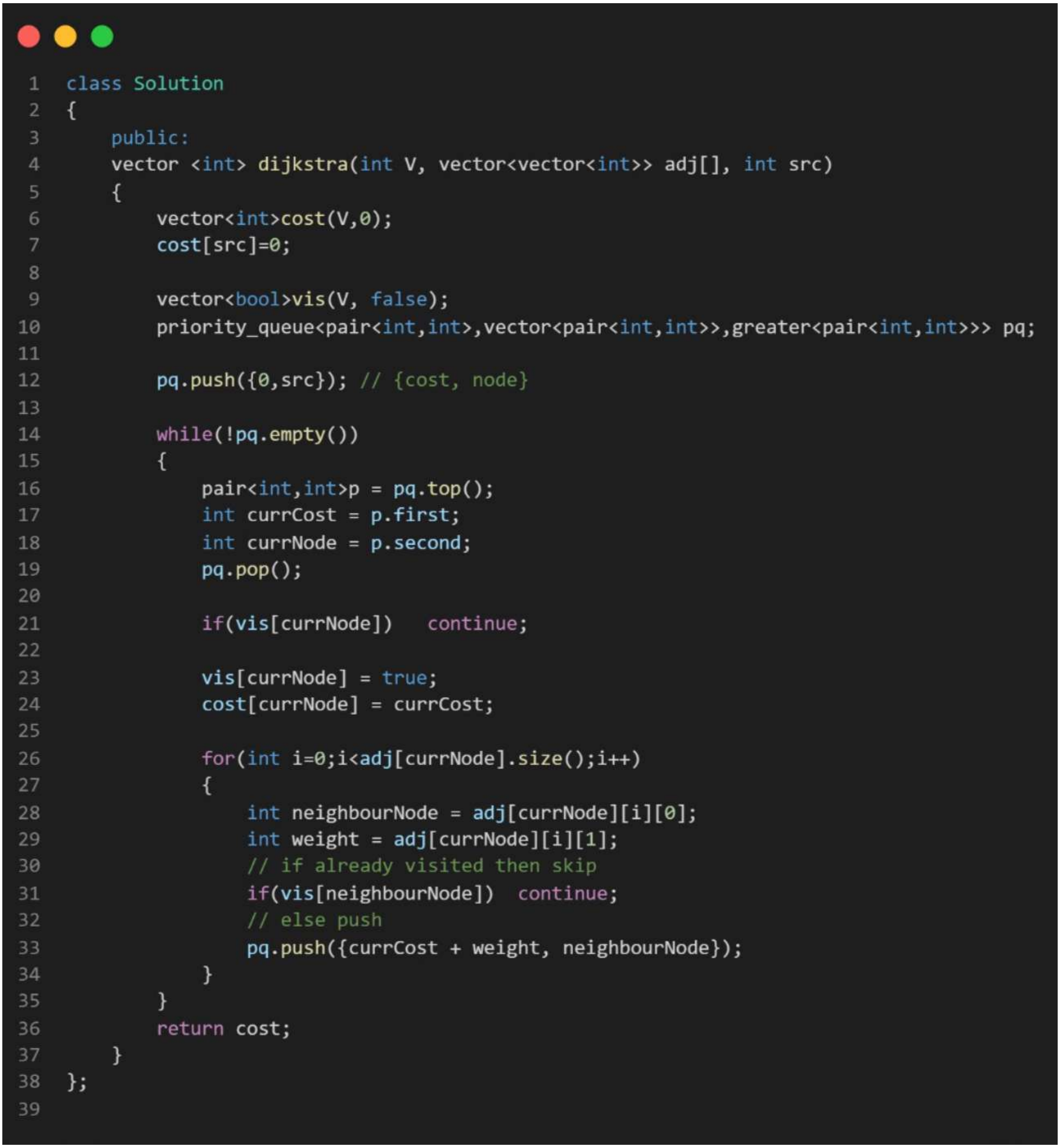
0	1	2	3	4	5	6	7	8
4	0	8	15	22	12	12	11	10

Dijkshras = BFS + PQ

Tc → O(V + E log V)

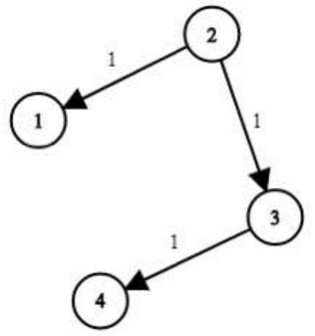
Sc → O(V)

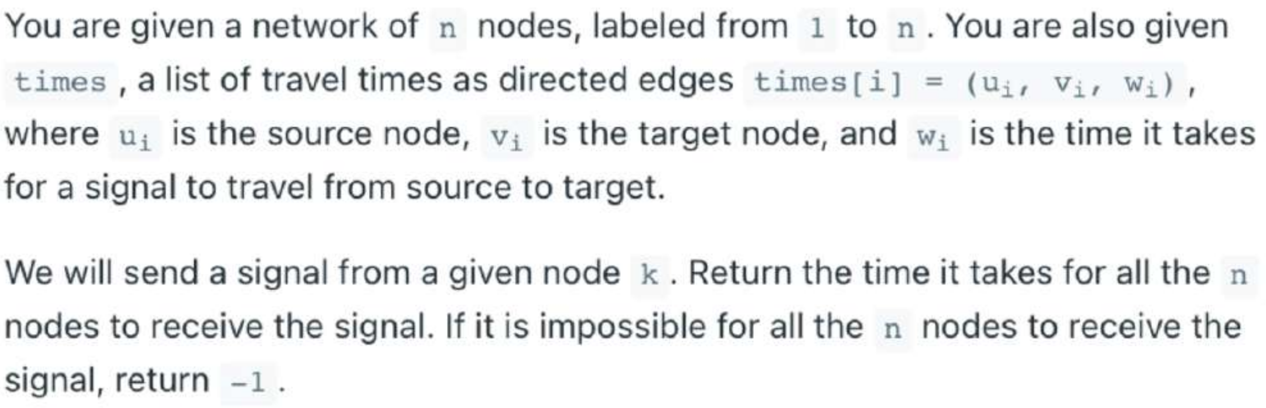
Code →



```cpp
class Solution
{
    public:
    vector <int> dijkstra(int V, vector<vector<int>> adj[], int src)
    {
        vector<int>cost(V,0);
        cost[src]=0;

        vector<bool>vis(V, false);
        priority_queue<pair<int,int>,vector<pair<int,int>>,greater<pair<int,int>>> pq;

        pq.push({0,src}); // {cost, node}

        while(!pq.empty())
        {
            pair<int,int>p = pq.top();
            int currCost = p.first;
            int currNode = p.second;
            pq.pop();

            if(vis[currNode])   continue;

            vis[currNode] = true;
            cost[currNode] = currCost;

            for(int i=0;i<adj[currNode].size();i++)
            {
                int neighbourNode = adj[currNode][i][0];
                int weight = adj[currNode][i][1];
                // if already visited then skip
                if(vis[neighbourNode])  continue;
                // else push
                pq.push({currCost + weight, neighbourNode});
            }
        }
        return cost;
    }
};

```

(12) Netwoork 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] = (ui, vi, wi)`, where `ui` is the source node, `vi` is the target node, and `wi` 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`.

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

vis = {2}
cost[2] = 0

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

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

⇒ (1,1) (3,1)   no new neighbours ∴ pop

vis = {2, 1}
cost[1] = 1

~~(1,1)~~ (3,1)

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

⇒ (3,1)   neighbour = (4,2) ∴ push

vis = {2,1,3}
cost[3] = 1

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

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

⇒ (4,2)   no new neighbours ∴ pop

vis = {2,1,3,4}
cost[4] = 2

~~(4,2)~~

→ pq is empty.

Tc → O(V + E log V)
Sc → O(V)

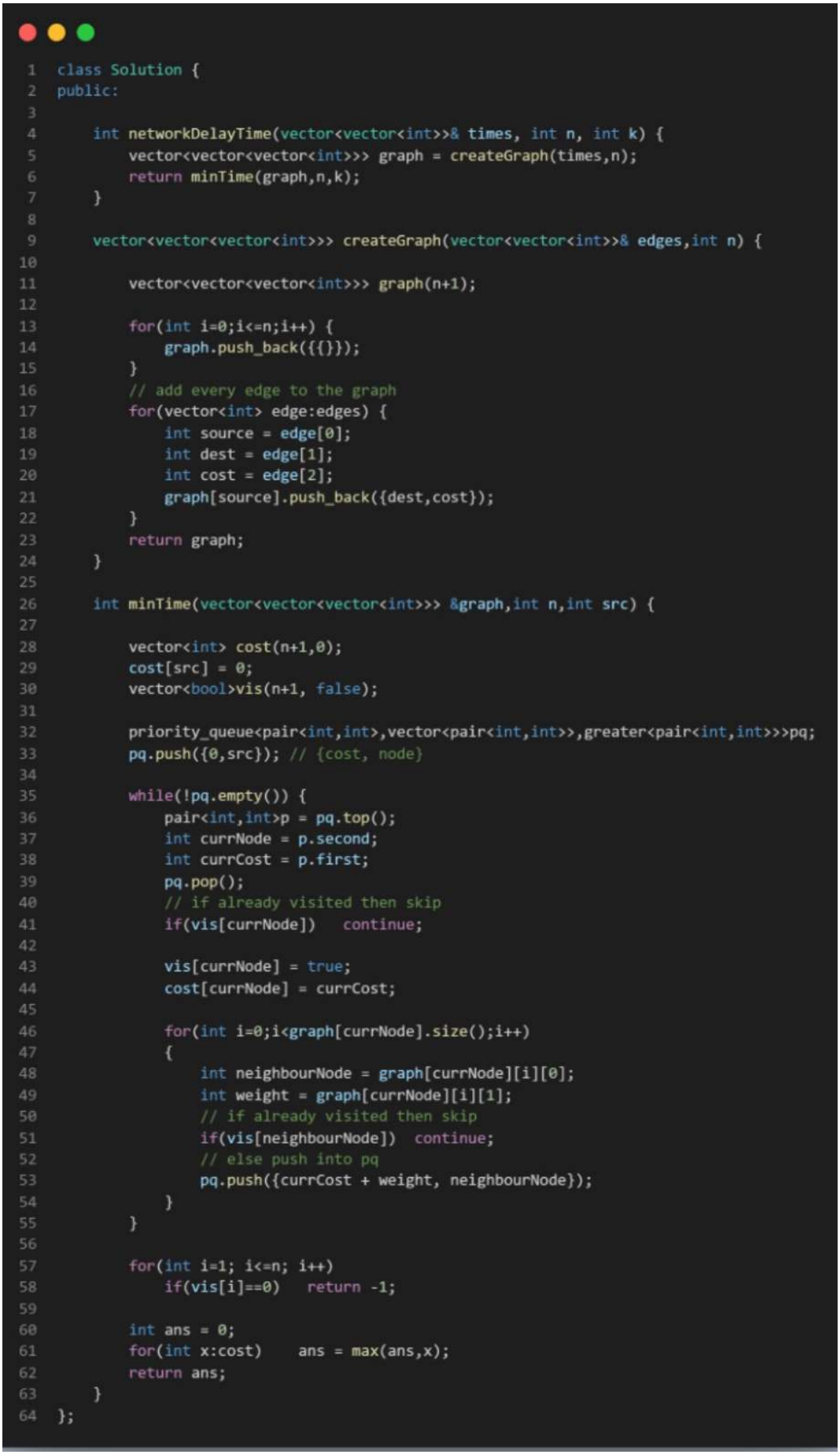
∴ cost =

0	1	0	1	2
0	1	2	3	4

→ check if all nodes are in visited, else return -1.

→ Return max value in cost as

Code →



```
class Solution {
public:

    int networkDelayTime(vector<vector<int>>& times, int n, int k) {
        vector<vector<vector<int>>> graph = createGraph(times,n);
        return minTime(graph,n,k);
    }

    vector<vector<vector<int>>> createGraph(vector<vector<int>>& edges,int n) {

        vector<vector<vector<int>>> graph(n+1);

        for(int i=0;i<=n;i++) {
            graph.push_back({{}});
        }
        // add every edge to the graph
        for(vector<int> edge:edges) {
            int source = edge[0];
            int dest = edge[1];
            int cost = edge[2];
            graph[source].push_back({dest,cost});
        }
        return graph;
    }

    int minTime(vector<vector<vector<int>>> &graph,int n,int src) {

        vector<int> cost(n+1,0);
        cost[src] = 0;
        vector<bool>vis(n+1, false);

        priority_queue<pair<int,int>,vector<pair<int,int>>,greater<pair<int,int>>>pq;
        pq.push({0,src}); // {cost, node}

        while(!pq.empty()) {
            pair<int,int>p = pq.top();
            int currNode = p.second;
            int currCost = p.first;
            pq.pop();
            // if already visited then skip
            if(vis[currNode])   continue;

            vis[currNode] = true;
            cost[currNode] = currCost;

            for(int i=0;i<graph[currNode].size();i++)
            {
                int neighbourNode = graph[currNode][i][0];
                int weight = graph[currNode][i][1];
                // if already visited then skip
                if(vis[neighbourNode])  continue;
                // else push into pq
                pq.push({currCost + weight, neighbourNode});
            }
        }

        for(int i=1; i<=n; i++)
            if(vis[i]==0)   return -1;

        int ans = 0;
        for(int x:cost)    ans = max(ans,x);
        return ans;
    }
};
```

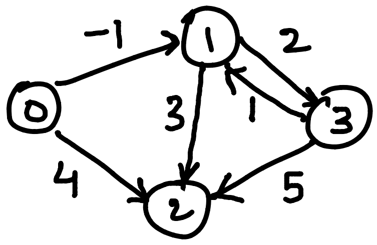
(13) Bellman Ford Algorithm → useful when weights < 0 (Dijkstra fails)

↳ dp algo

→ useful when finding negative weight cycle.

[src, dest, wt]

Eg n = 4 edges = [[0, 1, -1], [0, 2, 4], [1, 2, 3], [1, 3, 2], [3, 1, 1], [3, 2, 5]]



initially dist

0	1	2	3
inf	inf	inf	inf

⇒ dist[src] = 0 &

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

$$dist[dest] = \min(dist[src] + weight,\ dist[dest])$$

⇒ finally relax one more time &

if dist[dest] > dist[src] + wt ⇒ -ve weight cycle present

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

0	1	2	3
0	inf	inf	inf

→ for edge [0,1,-1], dist[1] = min(0+(-1), inf) = -1

[0,2,4], dist[2] = min(0+4, inf) = 4

[1,2,3], dist[2] = min(-1+3, 4) = 2

[1,3,2], dist[3] = min(-1+2, inf) = 1

[3,1,1], dist[1] = min(1+1, -1) = -1

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

∴ dist =

0	1	2	3
0	-1	2	1

→ 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	3
0	-1	2	1

TC → O(V*E)

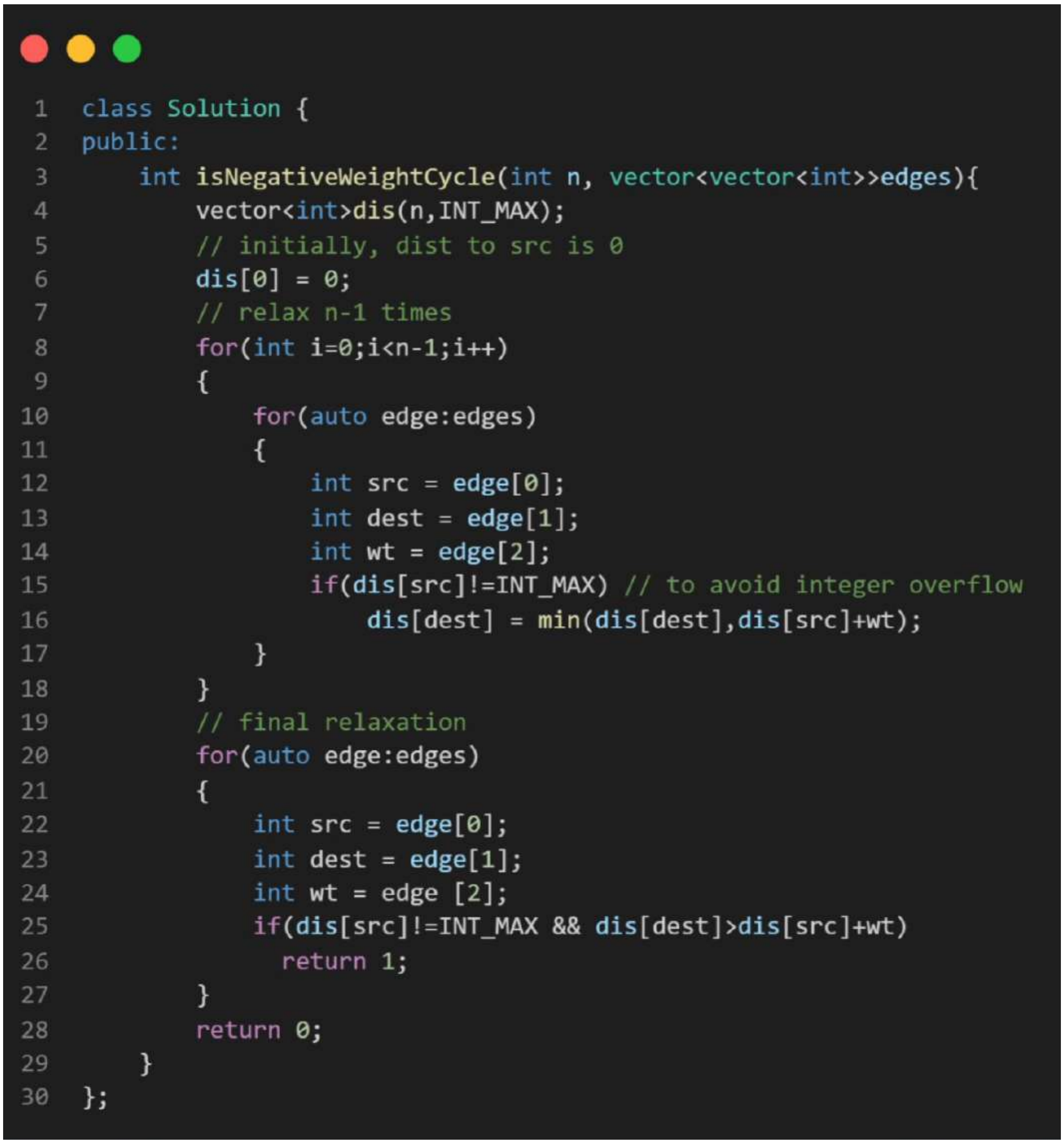
SC → O(V)

(14) Negative weight Cycle → Bellman Ford Algorithm.

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

TC → O(V*E)
SC → O(V)

code →



```
class Solution {
public:
    int isNegativeWeightCycle(int n, vector<vector<int>>edges){
        vector<int>dis(n,INT_MAX);
        // initially, dist to src is 0
        dis[0] = 0;
        // relax n-1 times
        for(int i=0;i<n-1;i++)
        {
            for(auto edge:edges)
            {
                int src = edge[0];
                int dest = edge[1];
                int wt = edge[2];
                if(dis[src]!=INT_MAX) // to avoid integer overflow
                    dis[dest] = min(dis[dest],dis[src]+wt);
            }
        }
        // final relaxation
        for(auto edge:edges)
        {
            int src = edge[0];
            int dest = edge[1];
            int wt = edge [2];
            if(dis[src]!=INT_MAX && dis[dest]>dis[src]+wt)
              return 1;
        }
        return 0;
    }
};
```

15 Floyd Warshall Algorithm

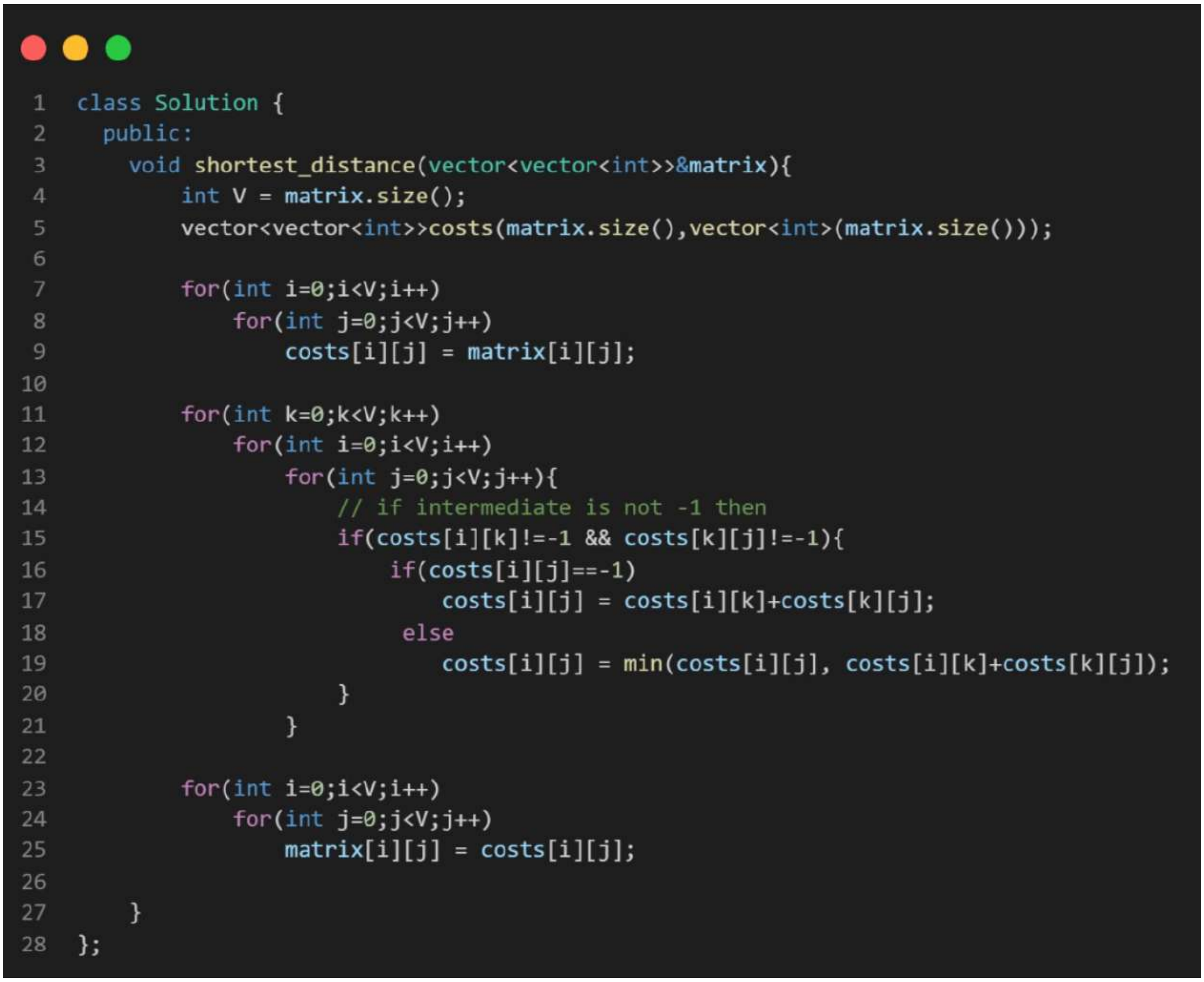
→ All source shortest path & -ve edges allowed.

→ Since its all source shortest path we need to run the loop for all nodes, considering it as intermediatery vertex.

→ $cost[i][j] = min(cost[i][j], cost[i][k] + cost[k][j])$

TC → $O(N^3)$ SC → $O(N^2)$

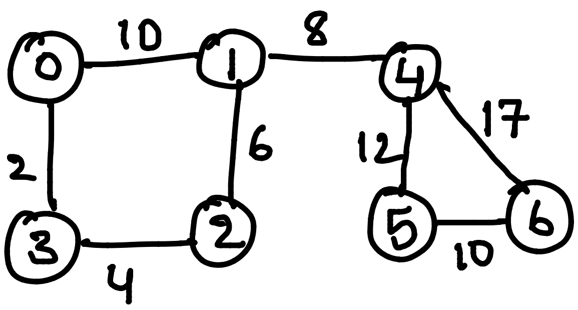
Code →



```
class Solution {
  public:
    void shortest_distance(vector<vector<int>>&matrix){
        int V = matrix.size();
        vector<vector<int>>costs(matrix.size(),vector<int>(matrix.size()));

        for(int i=0;i<V;i++)
            for(int j=0;j<V;j++)
                costs[i][j] = matrix[i][j];

        for(int k=0;k<V;k++)
            for(int i=0;i<V;i++)
                for(int j=0;j<V;j++){
                    // if intermediate is not -1 then
                    if(costs[i][k]!=-1 && costs[k][j]!=-1){
                        if(costs[i][j]==-1)
                            costs[i][j] = costs[i][k]+costs[k][j];
                         else
                            costs[i][j] = min(costs[i][j], costs[i][k]+costs[k][j]);
                    }
                }

        for(int i=0;i<V;i++)
            for(int j=0;j<V;j++)
                matrix[i][j] = costs[i][j];

    }
};
```

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

Eg



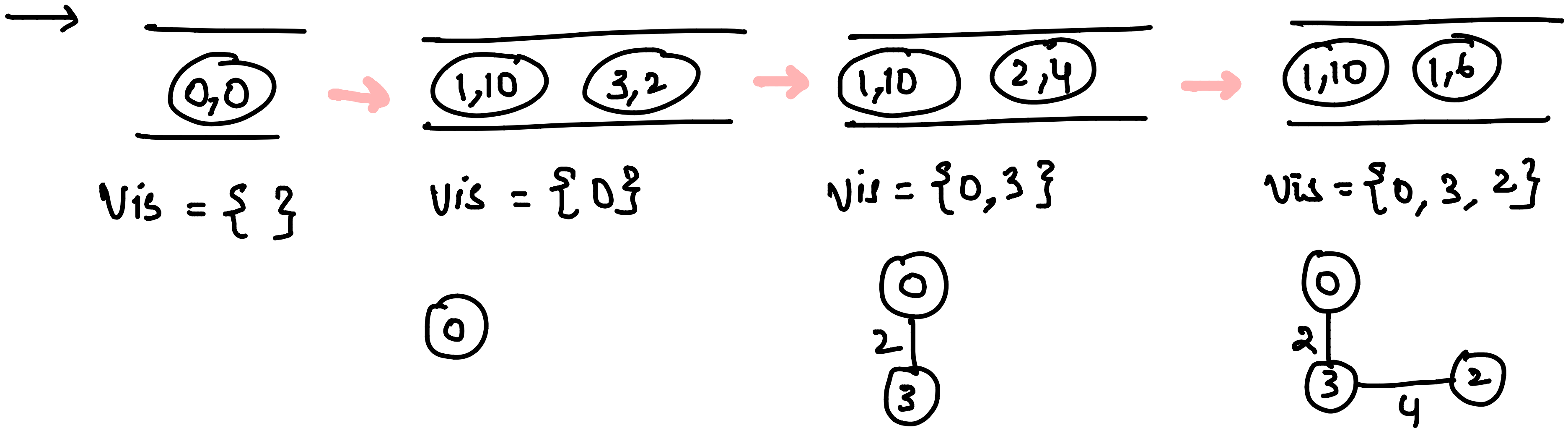
vis = { }

pq

node, weight

↑ returns node with lowest cost/weight.

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

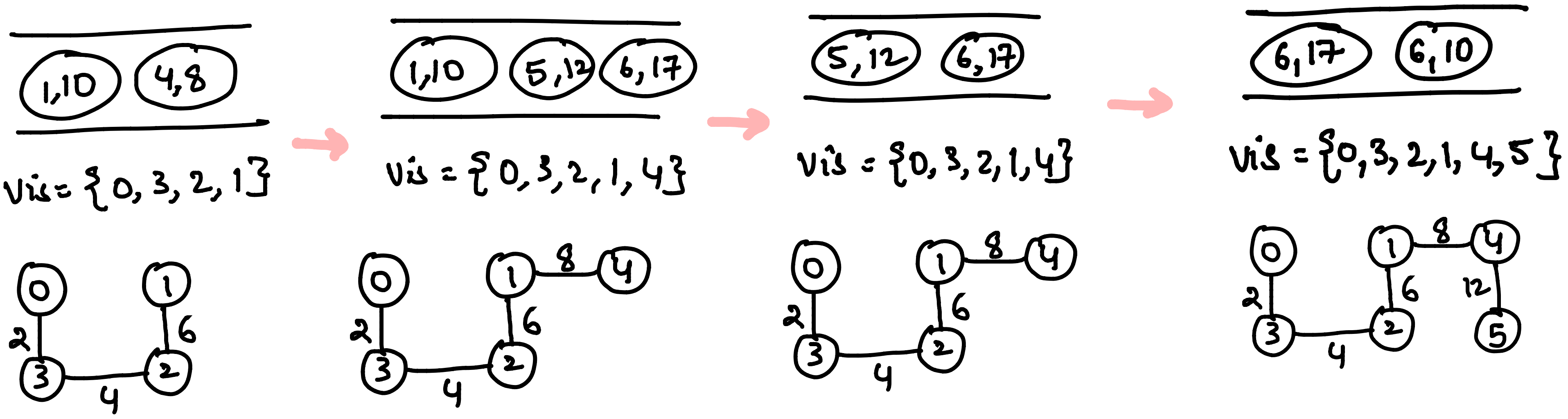


Vis = { }

Vis = {0}

Vis = {0, 3}

Vis = {0, 3, 2}

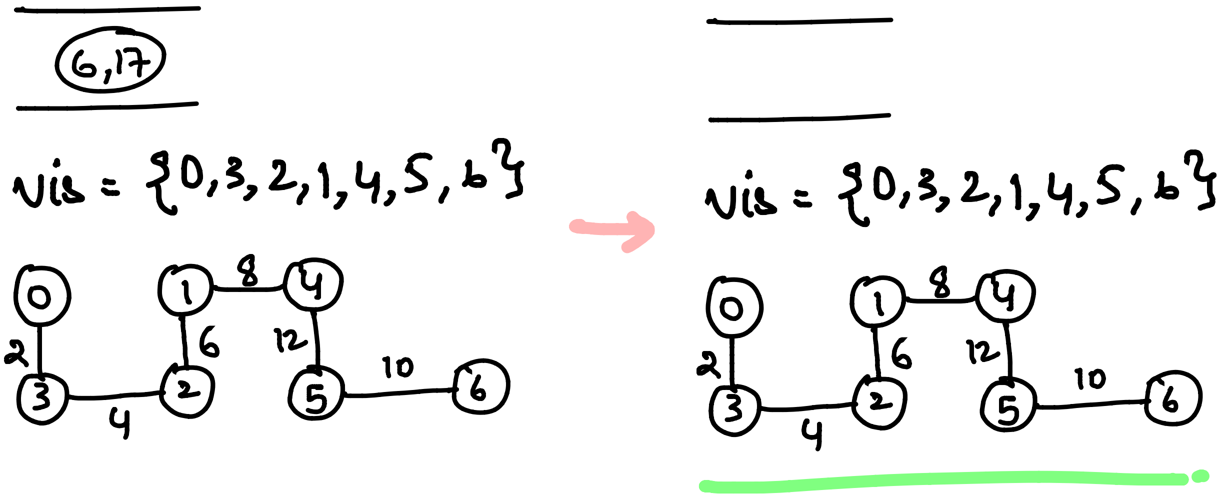


Vis = {0, 3, 2, 1}

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

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

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



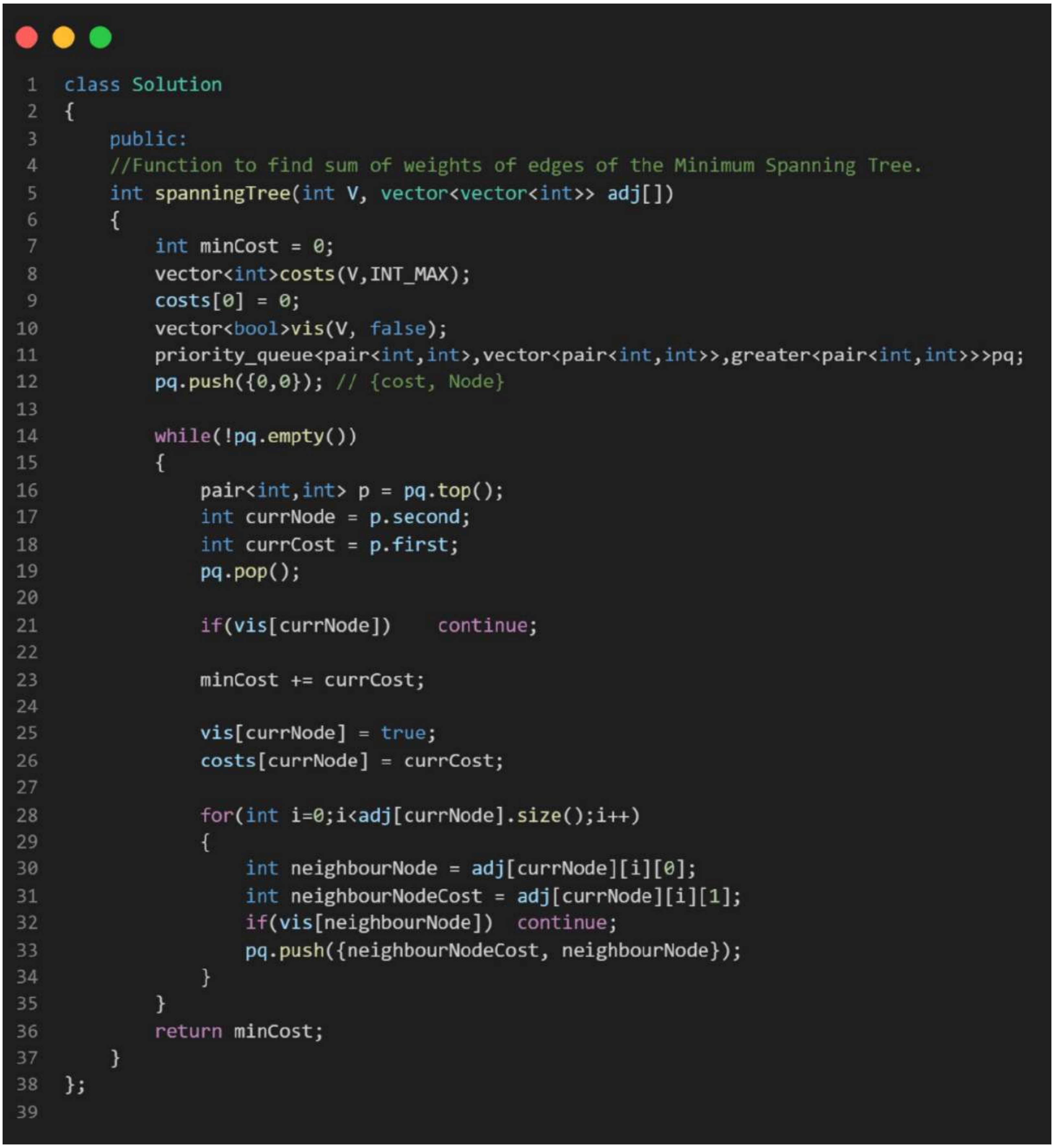
Vis = {0, 3, 2, 1, 4, 5, 6}

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

TC → $O(V + E\log V)$

SC → $O(V)$

Code →



```
class Solution
{
    public:
    //Function to find sum of weights of edges of the Minimum Spanning Tree.
    int spanningTree(int V, vector<vector<int>> adj[])
    {
        int minCost = 0;
        vector<int>costs(V,INT_MAX);
        costs[0] = 0;
        vector<bool>vis(V, false);
        priority_queue<pair<int,int>,vector<pair<int,int>>,greater<pair<int,int>>>pq;
        pq.push({0,0}); // {cost, Node}

        while(!pq.empty())
        {
            pair<int,int> p = pq.top();
            int currNode = p.second;
            int currCost = p.first;
            pq.pop();

            if(vis[currNode])    continue;

            minCost += currCost;

            vis[currNode] = true;
            costs[currNode] = currCost;

            for(int i=0;i<adj[currNode].size();i++)
            {
                int neighbourNode = adj[currNode][i][0];
                int neighbourNodeCost = adj[currNode][i][1];
                if(vis[neighbourNode])  continue;
                pq.push({neighbourNodeCost, neighbourNode});
            }
        }
        return minCost;
    }
};

```

(17) Min cost to connect all points

→ Create graph with each node containing Wt & Node value

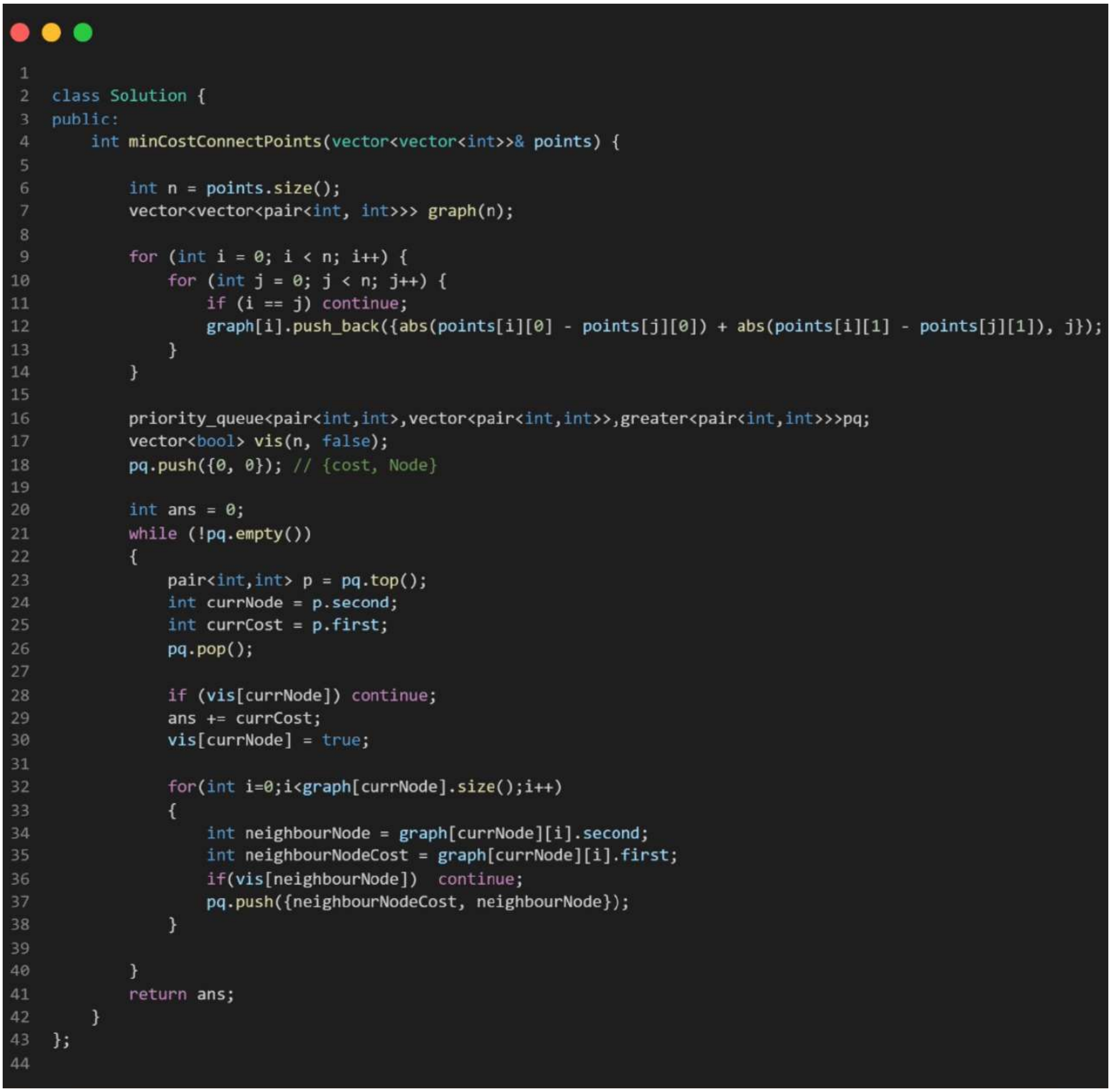
Wt = $abs(X_i - X) + abs(Y_i - Y)$

→ Perform Prims algo.

Tc → $O(V + E\log V)$

SC → $O(V)$

code →

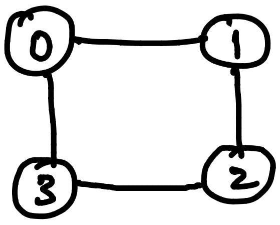


```cpp

class Solution {
public:
    int minCostConnectPoints(vector<vector<int>>& points) {

        int n = points.size();
        vector<vector<pair<int, int>>> graph(n);

        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                if (i == j) continue;
                graph[i].push_back({abs(points[i][0] - points[j][0]) + abs(points[i][1] - points[j][1]), j});
            }
        }

        priority_queue<pair<int,int>,vector<pair<int,int>>,greater<pair<int,int>>>pq;
        vector<bool> vis(n, false);
        pq.push({0, 0}); // {cost, Node}

        int ans = 0;
        while (!pq.empty())
        {
            pair<int,int> p = pq.top();
            int currNode = p.second;
            int currCost = p.first;
            pq.pop();

            if (vis[currNode]) continue;
            ans += currCost;
            vis[currNode] = true;

            for(int i=0;i<graph[currNode].size();i++)
            {
                int neighbourNode = graph[currNode][i].second;
                int neighbourNodeCost = graph[currNode][i].first;
                if(vis[neighbourNode])  continue;
                pq.push({neighbourNodeCost, neighbourNode});
            }

        }
        return ans;
    }
};

```

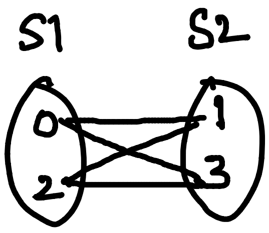
(18) Is graph Bipartite

Bipartite graph is undirected graph, such that all vertices can be divided into 2 sets, S1 & S2 and no two vertices present in same set share an edge.

Eg n = 4



then



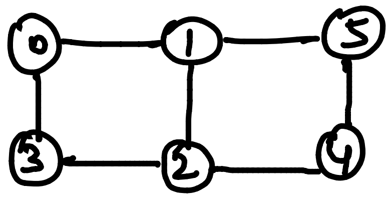
∴ the graph is bipartite.

⇒ for graph to be bipartite,

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

→ we generally denote sets by coloring it, color = 0, 1.
↓ ↓
S1 S2

Eg n = 6



vis = { } S1 = { } S2 = { }

initially color

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

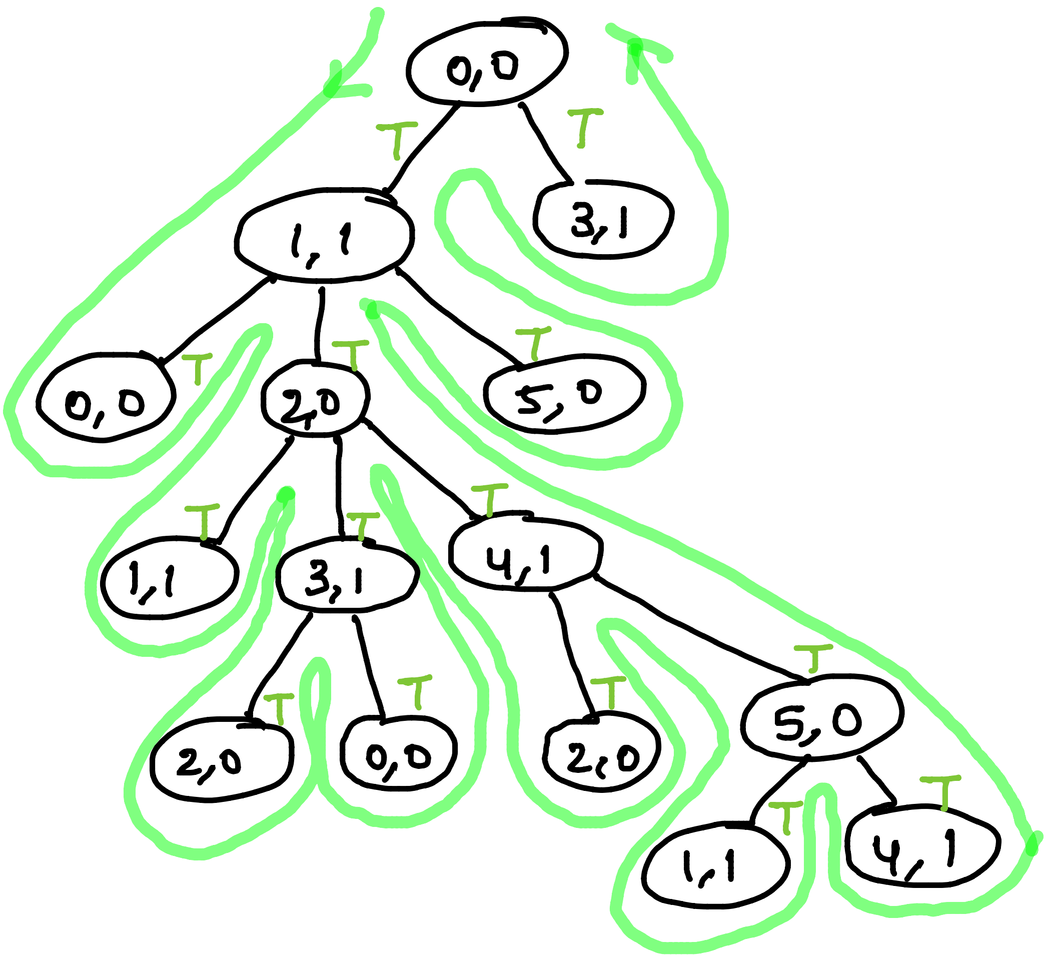
→ at each vertex, check if its visited or not.

→ if visited then check if its present in the intended set or not.

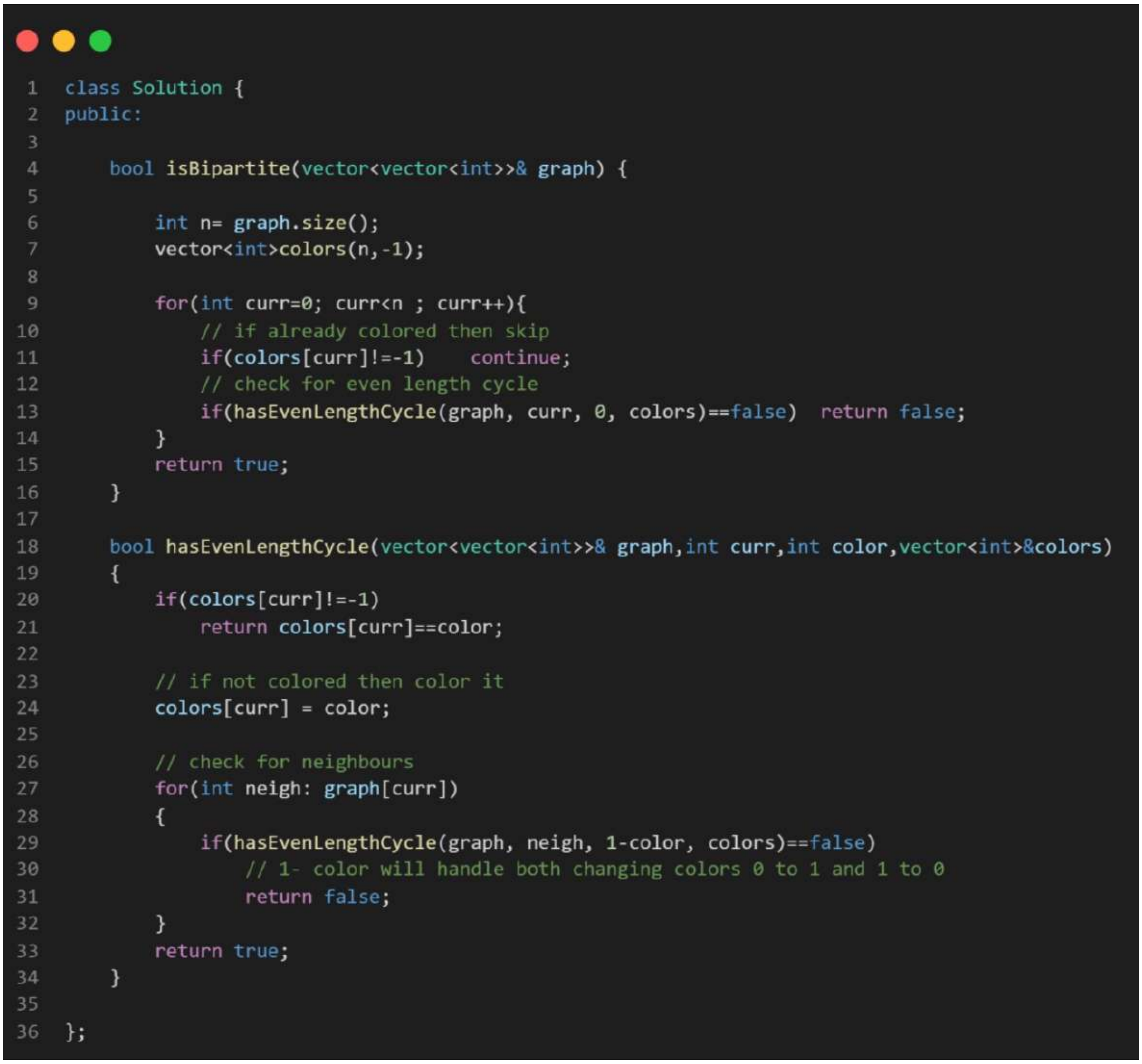
→ if yes then return true, else false

→ return AND of all the boolean values.

⇒



Code →



```
class Solution {
public:

    bool isBipartite(vector<vector<int>>& graph) {

        int n= graph.size();
        vector<int>colors(n,-1);

        for(int curr=0; curr<n ; curr++){
            // if already colored then skip
            if(colors[curr]!=-1)    continue;
            // check for even length cycle
            if(hasEvenLengthCycle(graph, curr, 0, colors)==false)  return false;
        }
        return true;
    }

    bool hasEvenLengthCycle(vector<vector<int>>& graph,int curr,int color,vector<int>&colors)
    {
        if(colors[curr]!=-1)
            return colors[curr]==color;

        // if not colored then color it
        colors[curr] = color;

        // check for neighbours
        for(int neigh: graph[curr])
        {
            if(hasEvenLengthCycle(graph, neigh, 1-color, colors)==false)
                // 1- color will handle both changing colors 0 to 1 and 1 to 0
                return false;
        }
        return true;
    }

};
```

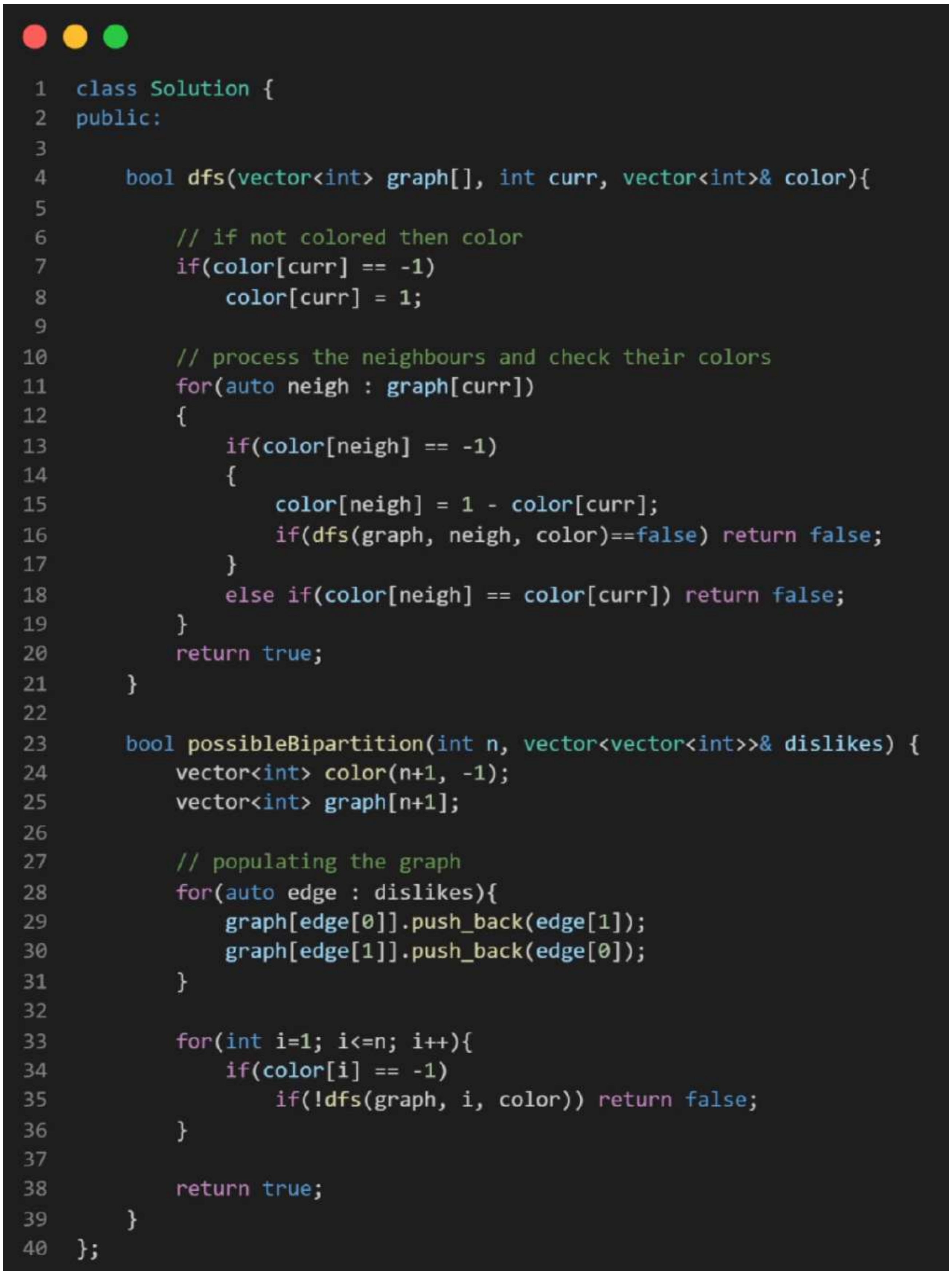
(19) Possible Bipartition →

→ create a graph using dislikes array.

→ use previous problem's approach to solve it.

code →

TC → O(V+E)   SC → O(V+E)



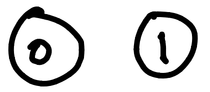
```cpp
class Solution {
public:

    bool dfs(vector<int> graph[], int curr, vector<int>& color){

        // if not colored then color
        if(color[curr] == -1)
            color[curr] = 1;

        // process the neighbours and check their colors
        for(auto neigh : graph[curr])
        {
            if(color[neigh] == -1)
            {
                color[neigh] = 1 - color[curr];
                if(dfs(graph, neigh, color)==false) return false;
            }
            else if(color[neigh] == color[curr]) return false;
        }
        return true;
    }

    bool possibleBipartition(int n, vector<vector<int>>& dislikes) {
        vector<int> color(n+1, -1);
        vector<int> graph[n+1];

        // populating the graph
        for(auto edge : dislikes){
            graph[edge[0]].push_back(edge[1]);
            graph[edge[1]].push_back(edge[0]);
        }

        for(int i=1; i<=n; i++){
            if(color[i] == -1)
                if(!dfs(graph, i, color)) return false;
        }

        return true;
    }
};
```

(20) Disjoint set → UNION & FIND./getParent

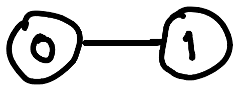
↳ helps in finding parent of component

↳ helps in UNION of components/vertices.

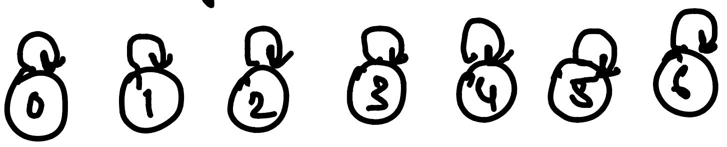
Eg



=> UNION(0,1) →



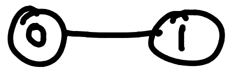
Eg n= 7 initially every component is parent of itself



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

now getParent(2) = 2, getParent(3) = 3.

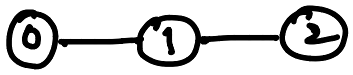
& if UNION(0,1) =>



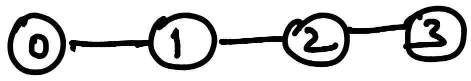
& parent[1] = 0

now getParent(1) = 0

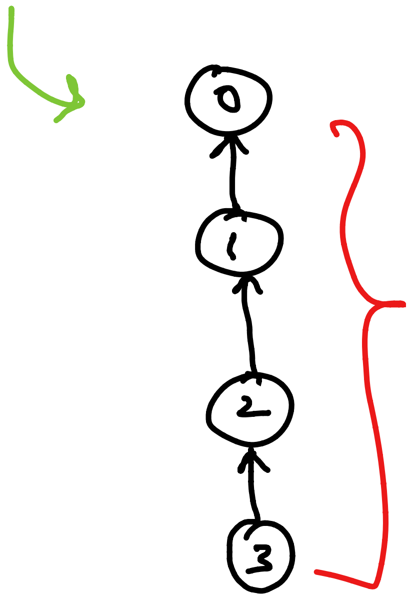
& UNION(1,2) =>



UNION(2,3) =>

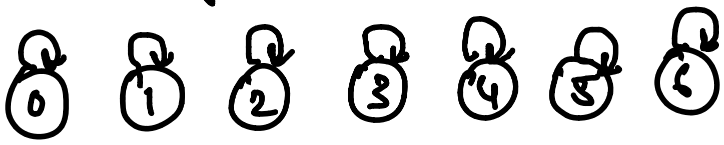


& getParent(3) = 0



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



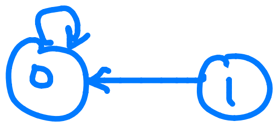
parent =	0	1	2	3	4	5	6
	0	1	2	3	4	5	6

rank =	0	0	0	0	0	0	0
	0	1	2	3	4	5	6

⟹ UNION (0,1) ⟹ then find(0) & find(1) & 0 != 1 ∴ diff components.

as they are diff components find rank & rank[0] = rank[1] = 0

∴ 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

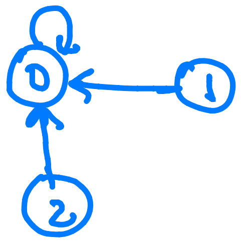
⟹ UNION (1,2) ⟹ parent(1) = 0 & parent(2) = 2

now rank[0] = 1 & rank[2] = 0

as rank[0] > rank[2] ,

vertex 0 should be the parent

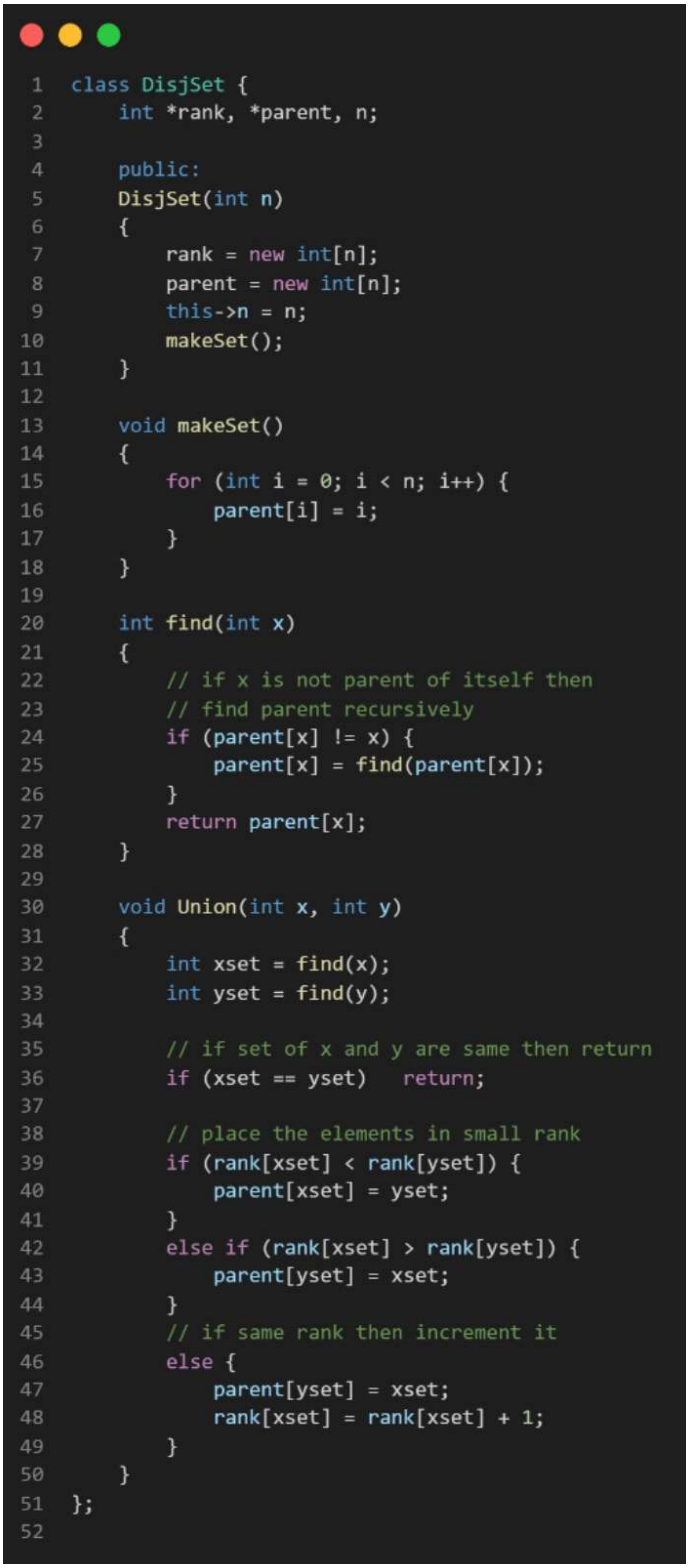
& donot update rank if they are unequal.

⟹ 

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 →



```cpp
class DisjSet {
    int *rank, *parent, n;

    public:
    DisjSet(int n)
    {
        rank = new int[n];
        parent = new int[n];
        this->n = n;
        makeSet();
    }

    void makeSet()
    {
        for (int i = 0; i < n; i++) {
            parent[i] = i;
        }
    }

    int find(int x)
    {
        // if x is not parent of itself then
        // find parent recursively
        if (parent[x] != x) {
            parent[x] = find(parent[x]);
        }
        return parent[x];
    }

    void Union(int x, int y)
    {
        int xset = find(x);
        int yset = find(y);

        // if set of x and y are same then return
        if (xset == yset)   return;

        // place the elements in small rank
        if (rank[xset] < rank[yset]) {
            parent[xset] = yset;
        }
        else if (rank[xset] > rank[yset]) {
            parent[yset] = xset;
        }
        // if same rank then increment it
        else {
            parent[yset] = xset;
            rank[xset] = rank[xset] + 1;
        }
    }
};
```

21 Kruskal's Algorithm →

→ This is used to find minimum spanning tree.

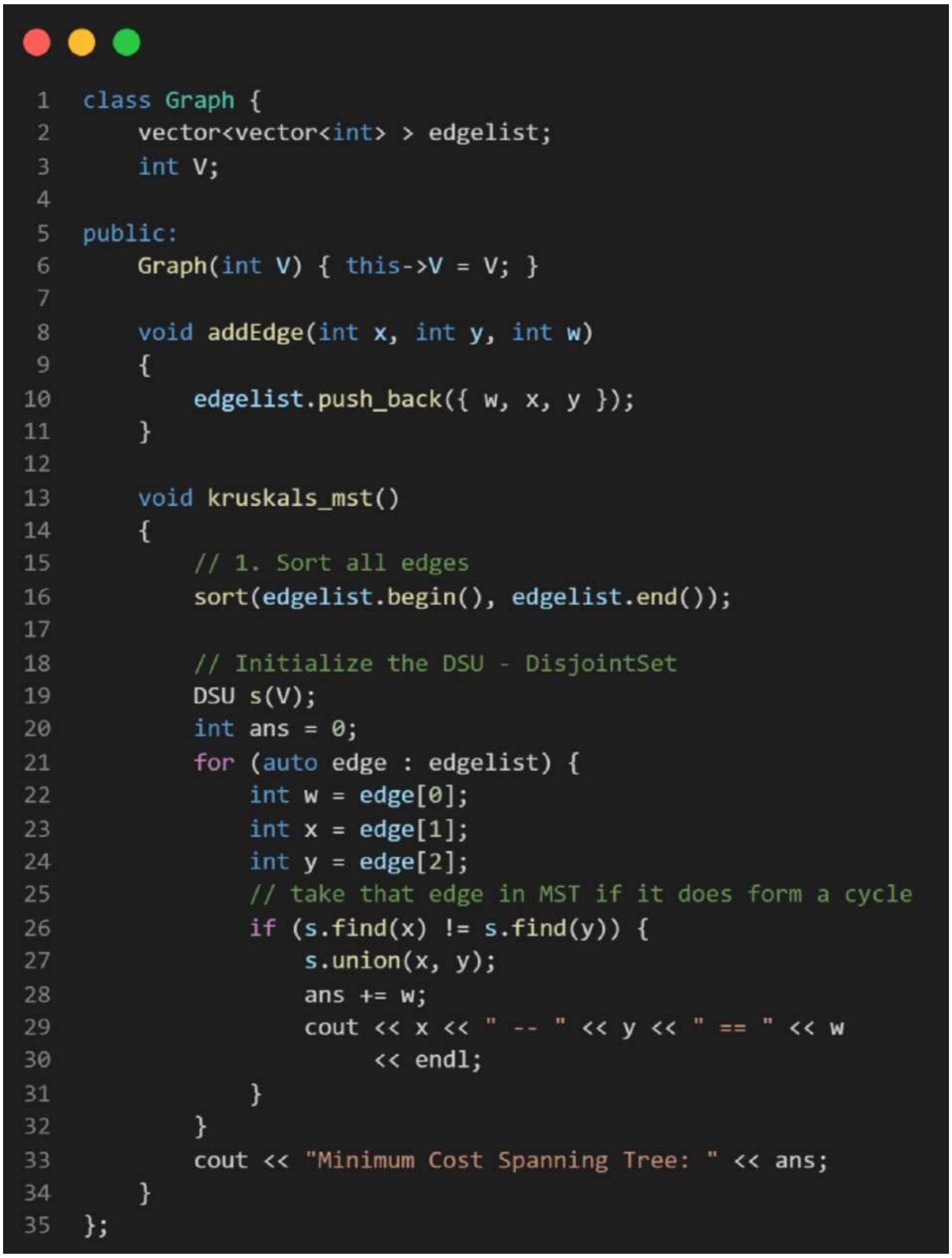
→ can be implemented using Disjoint set.

→ 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.

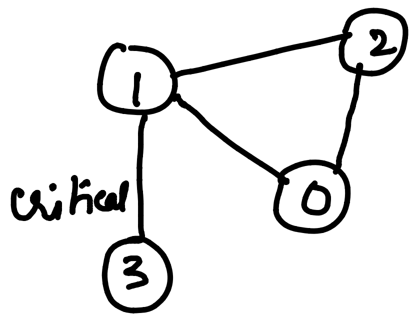
code →



```cpp
class Graph {
    vector<vector<int> > edgelist;
    int V;

public:
    Graph(int V) { this->V = V; }

    void addEdge(int x, int y, int w)
    {
        edgelist.push_back({ w, x, y });
    }

    void kruskals_mst()
    {
        // 1. Sort all edges
        sort(edgelist.begin(), edgelist.end());

        // Initialize the DSU - DisjointSet
        DSU s(V);
        int ans = 0;
        for (auto edge : edgelist) {
            int w = edge[0];
            int x = edge[1];
            int y = edge[2];
            // take that edge in MST if it does form a cycle
            if (s.find(x) != s.find(y)) {
                s.union(x, y);
                ans += w;
                cout << x << " -- " << y << " == " << w
                     << endl;
            }
        }
        cout << "Minimum Cost Spanning Tree: " << ans;
    }
};
```

(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.

distime: discovery time for vertex

mintime: min time for vertex to be discovered.

→ perform dfs from one node

→ if neighbour == parent then continue

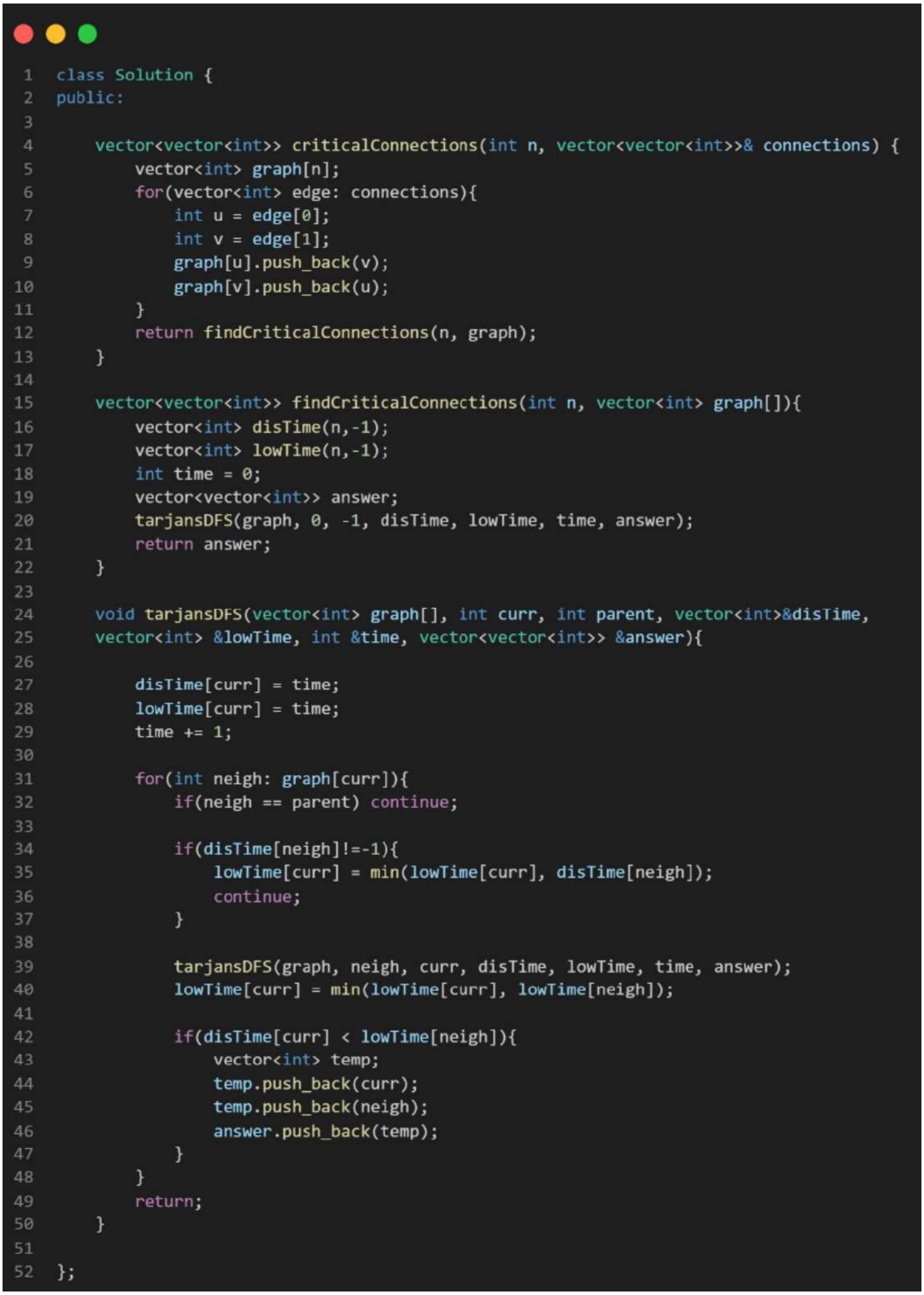
→ else if neighbour is already visited then

mintime[curr] = min(mintime[curr], distime[neigh])

→ while returning mintime[curr] = min(mintime[curr], mintime[neigh])
& at any point if distime[curr] < mintime[neigh]

This indicates critical connection

code →



```
class Solution {
public:

    vector<vector<int>> criticalConnections(int n, vector<vector<int>>& connections) {
        vector<int> graph[n];
        for(vector<int> edge: connections){
            int u = edge[0];
            int v = edge[1];
            graph[u].push_back(v);
            graph[v].push_back(u);
        }
        return findCriticalConnections(n, graph);
    }

    vector<vector<int>> findCriticalConnections(int n, vector<int> graph[]){
        vector<int> disTime(n,-1);
        vector<int> lowTime(n,-1);
        int time = 0;
        vector<vector<int>> answer;
        tarjansDFS(graph, 0, -1, disTime, lowTime, time, answer);
        return answer;
    }

    void tarjansDFS(vector<int> graph[], int curr, int parent, vector<int>&disTime,
    vector<int> &lowTime, int &time, vector<vector<int>> &answer){

        disTime[curr] = time;
        lowTime[curr] = time;
        time += 1;

        for(int neigh: graph[curr]){
            if(neigh == parent) continue;

            if(disTime[neigh]!=-1){
                lowTime[curr] = min(lowTime[curr], disTime[neigh]);
                continue;
            }

            tarjansDFS(graph, neigh, curr, disTime, lowTime, time, answer);
            lowTime[curr] = min(lowTime[curr], lowTime[neigh]);

            if(disTime[curr] < lowTime[neigh]){
                vector<int> temp;
                temp.push_back(curr);
                temp.push_back(neigh);
                answer.push_back(temp);
            }
        }
        return;
    }

};
```

Find the rest on

https://linktr.ee/KarunKarthik

Follow Karun Karthik For More Amazing Content !