AccountMerge in C++

```cpp
#include <bits/stdc++.h>
using namespace std;
//User function Template for C++
class DisjointSet {
    vector<int> rank, parent, size;
public:
    DisjointSet(int n) {
        rank.resize(n + 1, 0);
        parent.resize(n + 1);
        size.resize(n + 1);
        for (int i = 0; i <= n; i++) {
            parent[i] = i;
            size[i] = 1;
        }
    }

    int findUPar(int node) {
        if (node == parent[node])
            return node;
        return parent[node] = findUPar(parent[node]);
    }

    void unionByRank(int u, int v) {
        int ulp_u = findUPar(u);
        int ulp_v = findUPar(v);
        if (ulp_u == ulp_v) return;
        if (rank[ulp_u] < rank[ulp_v]) {
            parent[ulp_u] = ulp_v;
        }
        else if (rank[ulp_v] < rank[ulp_u]) {
            parent[ulp_v] = ulp_u;
        }
        else {
            parent[ulp_v] = ulp_u;
            rank[ulp_u]++;
        }
    }

    void unionBySize(int u, int v) {
        int ulp_u = findUPar(u);
        int ulp_v = findUPar(v);
        if (ulp_u == ulp_v) return;
        if (size[ulp_u] < size[ulp_v]) {
            parent[ulp_u] = ulp_v;
            size[ulp_v] += size[ulp_u];
        }
        else {
            parent[ulp_v] = ulp_u;
            size[ulp_u] += size[ulp_v];
        }
    }
};

class Solution {
public:
    vector<vector<string>>
accountsMerge(vector<vector<string>> &details) {
        int n = details.size();
        DisjointSet ds(n);
        sort(details.begin(), details.end());
        unordered_map<string, int> mapMailNode;
```

Input

```
{
    {"John", "j1@com", "j2@com",
"j3@com"},
    {"John", "j4@com"},
    {"Raj", "r1@com", "r2@com"},
    {"John", "j1@com", "j5@com"},
    {"Raj", "r2@com", "r3@com"},
    {"Mary", "m1@com"}
}
```

Let's assume these are indexed from `0` to `5`.

🧩 Step 1: Mapping Emails to Accounts with Union

We initialize a map `mail → nodeIndex`.
As we traverse, if we see a repeated email, we perform **unionBySize** between the current index and the one in the map.

Index	Account Name	Emails	Action
0	John	j1, j2, j3	Add all emails to map ⟶ `j1 → 0`, `j2 → 0`, `j3 → 0`
1	John	j4	`j4 → 1`
2	Raj	r1, r2	`r1 → 2`, `r2 → 2`
3	John	j1 (seen), j5	Union(3, 0) since `j1 → 0` ⟶ 3 belongs to same group as 0
4	Raj	r2 (seen), r3	Union(4, 2) since `r2 → 2` ⟶ 4 belongs to same group as 2
5	Mary	m1	`m1 → 5`

📌 After unions:

- Group 0 includes index `0` and `3` (due to shared `j1`)
- Group 2 includes index `2` and `4` (due to shared `r2`)

🔁 Step 2: Group Emails Based on Ultimate Parent (Union-Find)

We iterate over the map and collect emails in

```
    for (int i = 0; i < n; i++) {
        for (int j = 1; j < details[i].size(); j++) {
            string mail = details[i][j];
            if (mapMailNode.find(mail) ==
mapMailNode.end()) {
                mapMailNode[mail] = i;
            }
            else {
                ds.unionBySize(i, mapMailNode[mail]);
            }
        }
    }

    vector<string> mergedMail[n];
    for (auto it : mapMailNode) {
        string mail = it.first;
        int node = ds.findUPar(it.second);
        mergedMail[node].push_back(mail);
    }

    vector<vector<string>> ans;

    for (int i = 0; i < n; i++) {
        if (mergedMail[i].size() == 0) continue;
        sort(mergedMail[i].begin(), mergedMail[i].end());
        vector<string> temp;
        temp.push_back(details[i][0]);
        for (auto it : mergedMail[i]) {
            temp.push_back(it);
        }
        ans.push_back(temp);
    }
    sort(ans.begin(), ans.end());
    return ans;
  }
};


int main() {

  vector<vector<string>> accounts = {{"John", "j1@com",
"j2@com", "j3@com"},
    {"John", "j4@com"},
    {"Raj", "r1@com", "r2@com"},
    {"John", "j1@com", "j5@com"},
    {"Raj", "r2@com", "r3@com"},
    {"Mary", "m1@com"}
  };


  Solution obj;
  vector<vector<string>> ans =
obj.accountsMerge(accounts);
  for (auto acc : ans) {
    cout << acc[0] << ":";
    int size = acc.size();
    for (int i = 1; i < size; i++) {
      cout << acc[i] << " ";
    }
    cout << endl;
  }
  return 0;
```

the list `mergedMail[parent]`.

Example:

- `j1 → 0` ⟶ `findUPar(0) = 0`
- `j5 → 3` ⟶ `findUPar(3) = 0` (after union)
- `r3 → 4` ⟶ `findUPar(4) = 2`

So we get:

Parent Index	Emails
0	j1, j2, j3, j5
1	j4
2	r1, r2, r3
5	m1

🧱 Step 3: Construct Final Answer

We loop over `mergedMail[]`, and for each non-empty vector:

- Sort the emails
- Use the **name from the original account at that index**

Group	Name	Sorted Emails
0	John	j1, j2, j3, j5
1	John	j4
2	Raj	r1, r2, r3
5	Mary	m1

✔ Final Output

```
John:j1@com j2@com j3@com j5@com
John:j4@com
Mary:m1@com
Raj:r1@com r2@com r3@com
```

✔ DSU Table View (Final Parents)

Let's print `findUPar(i)` for `i = 0 to 5`

Index	Account Name	Parent (after unions)
0	John	0
1	John	1
2	Raj	2
3	John	0

```
}
```

Index	Account Name	Parent (after unions)
4	Raj	2
5	Mary	5

Output:-

John:j1@com j2@com j3@com j5@com
John:j4@com
Mary:m1@com
Raj:r1@com r2@com r3@com

Articulation Point in C++

```cpp
#include <bits/stdc++.h>
using namespace std;

//User function Template for C++

class Solution {
private:
   int timer = 1;
   void dfs(int node, int parent, vector<int> &vis, int
tin[], int low[],
             vector<int> &mark, vector<int>adj[]) {
       vis[node] = 1;
       tin[node] = low[node] = timer;
       timer++;
       int child = 0;
       for (auto it : adj[node]) {
          if (it == parent) continue;
          if (!vis[it]) {
             dfs(it, node, vis, tin, low, mark, adj);
             low[node] = min(low[node], low[it]);
             if (low[it] >= tin[node] && parent != -1) {
                mark[node] = 1;
             }
             child++;
          }
          else {
             low[node] = min(low[node], tin[it]);
          }
       }
       if (child > 1 && parent == -1) {
          mark[node] = 1;
       }
   }
public:
   vector<int> articulationPoints(int n, vector<int>adj[])
{
       vector<int> vis(n, 0);
       int tin[n];
       int low[n];
       vector<int> mark(n, 0);
       for (int i = 0; i < n; i++) {
          if (!vis[i]) {
             dfs(i, -1, vis, tin, low, mark, adj);
          }
       }
       vector<int> ans;
       for (int i = 0; i < n; i++) {
          if (mark[i] == 1) {
             ans.push_back(i);
          }
       }
       if (ans.size() == 0) return {-1};
       return ans;
   }
};
int main() {

   int n = 5;
   vector<vector<int>> edges = {
       {0, 1}, {1, 4},
       {2, 4}, {2, 3}, {3, 4}
```

Graph Overview

Given edges:

```
0 - 1
    |
    4
   / \
  2 - 3
```

Adjacency List:

Node	Neighbors
0	1
1	0, 4
2	4, 3
3	2, 4
4	1, 2, 3

🔍 Variables Recap

- tin[node]: Time of first visit
- low[node]: Lowest reachable discovery time
- A node is an **articulation point** if:
  - Not root and low[child] >= tin[node]
  - Root and has ≥ 2 children

🧠 DFS Trace Table

Step	Node	Parent	tin	low	Action & Reasoning
1	0	-1	1	1	Start DFS from 0
2	1	0	2	2	Visit from 0
3	4	1	3	3	Visit from 1
4	2	4	4	4	Visit from 4
5	3	2	5	5	Visit from 2
6	4	3	-	3	Back edge to 4
7	2	4	-	3	low[2] = min(4, 3)
8	4	1	-	3	low[4] = min(3, 3)
9	1	0	-	2	low[1] = min(2, 3)
10	0	-1	-	1	Done

🔎 Articulation Point Analysis

We now check for articulation conditions.

- **Node 1**:

<table>
<tr>
<td>
<pre><code>   };

    vector<int> adj[n];
    for (auto it : edges) {
        int u = it[0], v = it[1];
        adj[u].push_back(v);
        adj[v].push_back(u);
    }
    Solution obj;
    vector<int> nodes = obj.articulationPoints(n, adj);
    for (auto node : nodes) {
        cout << node << " ";
    }
    cout << endl;
    return 0;
}</code></pre>
</td>
<td>
low[4] = 3 >= tin[1] = 2 → ✅ articulation point
Node 4:
low[2] = 3 >= tin[4] = 3
low[3] = 5 >= tin[4] = 3 → ✅ articulation point
Node 0:
Root with only 1 child → ❌ not articulation point
✅ Final Result
Articulation Points: 1 4
</td>
</tr>
<tr>
<td colspan="2">Output:-
1 4</td>
</tr>
</table>

Bellman-Ford in C++

```cpp
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
	/*  Function to implement Bellman Ford
	*   edges: vector of vectors which represents the graph
	*   S: source vertex to start traversing graph with
	*   V: number of vertices
	*/
	vector<int> bellman_ford(int V, vector<vector<int>>& edges, int S) {
		vector<int> dist(V, 1e8);
		dist[S] = 0;
		for (int i = 0; i < V - 1; i++) {
			for (auto it : edges) {
				int u = it[0];
				int v = it[1];
				int wt = it[2];
				if (dist[u] != 1e8 && dist[u] + wt < dist[v]) {
					dist[v] = dist[u] + wt;
				}
			}
		}
		// Nth relaxation to check negative cycle
		for (auto it : edges) {
			int u = it[0];
			int v = it[1];
			int wt = it[2];
			if (dist[u] != 1e8 && dist[u] + wt < dist[v]) {
				return { -1};
			}
		}


		return dist;
	}
};


int main() {

	int V = 6;
	vector<vector<int>> edges(7, vector<int>(3));
	edges[0] = {3, 2, 6};
	edges[1] = {5, 3, 1};
	edges[2] = {0, 1, 5};
	edges[3] = {1, 5, -3};
	edges[4] = {1, 2, -2};
	edges[5] = {3, 4, -2};
	edges[6] = {2, 4, 3};

	int S = 0;
	Solution obj;
	vector<int> dist = obj.bellman_ford(V, edges, S);
	for (auto d : dist) {
		cout << d << " ";
```

Initialization

Vertex	dist
0	0
1	∞
2	∞
3	∞
4	∞
5	∞

📊 After each iteration of relaxation (V-1 = 5 times):

We'll update dist[] step by step, showing changes caused by each edge.

🔁 Iteration 1:

Process edges:

1. 0→1 (5) → dist[1] = 5
2. 1→2 (-2) → dist[2] = 3
3. 1→5 (-3) → dist[5] = 2
4. 5→3 (1) → dist[3] = 3
5. 3→4 (-2) → dist[4] = 1
6. 2→4 (3) → already dist[4] = 1 so not updated
7. Other edges don't apply yet.

Result:

dist = [0, 5, 3, 3, 1, 2]

🔁 Iteration 2 to 5:

Now that distances are optimal and no further relaxation improves any values, **no changes happen**.

✅ Final dist[] after Bellman-Ford

Vertex	Final dist
0	0
1	5
2	3
3	3
4	1
5	2

`` } cout << endl; return 0; } ```	✅ **Correct Output:** 0 5 3 3 1 2

Output:-
0 5 3 3 1 2

Bipartite in Depth First Search in C++

```cpp
#include<bits/stdc++.h>
using namespace std;

class Solution {
private:
    bool dfs(int node, int col, int color[], vector<int> adj[]) {
        color[node] = col;

        // traverse adjacent nodes
        for(auto it : adj[node]) {
            // if uncoloured
            if(color[it] == -1) {
                if(dfs(it, !col, color, adj) == false) return false;
            }
            // if previously coloured and have the same colour
            else if(color[it] == col) {
                return false;
            }
        }

        return true;
    }
public:
    bool isBipartite(int V, vector<int>adj[]){
        int color[V];
        for(int i = 0;i<V;i++) color[i] = -1;

        // for connected components
        for(int i = 0;i<V;i++) {
            if(color[i] == -1) {
                if(dfs(i, 0, color, adj) == false)
                    return false;
            }
        }
        return true;
    }
};

void addEdge(vector <int> adj[], int u, int v) {
    adj[u].push_back(v);
    adj[v].push_back(u);
}

int main(){

    // V = 4, E = 4
    vector<int>adj[4];

    addEdge(adj, 0, 2);
    addEdge(adj, 0, 3);
    addEdge(adj, 2, 3);
    addEdge(adj, 3, 1);

    Solution obj;
    bool ans = obj.isBipartite(4, adj);
    if(ans)cout << "1\n";
    else cout << "0\n";
```

Graph Construction (4 vertices, 4 edges):

```
addEdge(adj, 0, 2);  // 0 - 2
addEdge(adj, 0, 3);  // 0 - 3
addEdge(adj, 2, 3);  // 2 - 3
addEdge(adj, 3, 1);  // 3 - 1
```

🔁 Adjacency List:

Vertex	Neighbors
0	2, 3
1	3
2	0, 3
3	0, 2, 1

🎯 DFS Coloring Attempt:

- Initialize all colors as -1.
- Try to color graph with **two colors**: 0 and 1.

🧠 Dry Run Table

Node Visited	Action	Color Assigned	Stack/Call Stack	Conflict?
0	Start DFS	0	dfs(0, 0)	No
2	Visit from 0	1	dfs(2, 1)	No
3	Visit from 2	0	dfs(3, 0)	No
0	Already colored	0	Check if conflict with 0	✅ Match
1	Visit from 3	1	dfs(1, 1)	No
3	Already colored	0	Check if conflict with 1	✅ Match
2	Already colored	1	Check if conflict with 3 (expect 1, found 0)	❌ **Conflict!**

At this point, DFS at node 3 sees that its neighbor 2 is also colored 1, and this **violates the bipartite condition**, because both are expected to have **opposite** colors.

❌ Final Result:

0

`    return 0;` `}`	
Output:- 0	

DFS Cycle undirected in C++

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
  private:
    bool dfs(int node, int parent, int
vis[], vector<int> adj[]) {
        vis[node] = 1;
        // visit adjacent nodes
        for(auto adjacentNode: adj[node])
{
            // unvisited adjacent node
            if(!vis[adjacentNode]) {
               if(dfs(adjacentNode, node,
vis, adj) == true)
                    return true;
            }
            // visited node but not a parent
node
            else if(adjacentNode != parent)
return true;
        }
        return false;
    }
  public:
    // Function to detect cycle in an
undirected graph.
    bool isCycle(int V, vector<int> adj[])
{
       int vis[V] = {0};
       // for graph with connected
components
       for(int i = 0;i<V;i++) {
           if(!vis[i]) {
              if(dfs(i, -1, vis, adj) == true)
return true;
           }
       }
       return false;
    }
};

int main() {

    // V = 4, E = 2
    vector<int> adj[4] = {{}, {2}, {1, 3},
{2}};
    Solution obj;
    bool ans = obj.isCycle(4, adj);
    if (ans)
        cout << "1\n";
    else
        cout << "0\n";
    return 0;
}
```

Graph Input (V = 4):

```
vector<int> adj[4] = {
    {},      // Node 0: No edges
    {2},     // Node 1: Connected to 2
    {1, 3},  // Node 2: Connected to 1 and 3
    {2}      // Node 3: Connected to 2
};
```

So the actual edges are:

- 1 - 2
- 2 - 3

This graph is a simple path, not a cycle.

🔁 Dry Run Table (DFS traversal):

Step	Current Node	Parent	vis[] Status	Adjacent Nodes	Action	Cycle Detected?
1	0	-1	[1, 0, 0, 0]	{}	No adj nodes	No
2	1	-1	[1, 1, 0, 0]	{2}	DFS to 2	No
3	2	1	[1, 1, 1, 0]	{1, 3}	1 is parent, DFS to 3	No
4	3	2	[1, 1, 1, 1]	{2}	2 is parent, backtrack	No

🔚 No cycle detected

The code correctly determines that no adjacent node points back to a **previously visited node that's not its parent**, so there is **no cycle**.

📄 Output:

0

Output:-
0

Depth First Search in C++

```cpp
#include <iostream>
#include <vector>

using namespace std;

class DFSDirected {
public:
    static vector<int> dfs(int s, vector<bool>& vis,
vector<vector<int>>& adj, vector<int>& ls) {
        vis[s] = true;
        ls.push_back(s);
        for (int it : adj[s]) {
            if (!vis[it]) {
                dfs(it, vis, adj, ls);
            }
        }
        return ls;
    }
};

int main() {
    int V = 5;
    vector<bool> vis(V + 1, false);
    vector<int> ls;
    vector<vector<int>> adj(V + 1);

    adj[1].push_back(3);
    adj[1].push_back(2);
    adj[3].push_back(4);
    adj[4].push_back(5);

    vector<vector<int>> res;
    for (int i = 1; i <= V; i++) {
        if (!vis[i]) {
            vector<int> ls;
            res.push_back(DFSDirected::dfs(i, vis, adj, ls));
        }
    }

    for (const auto& component : res) {
        for (int node : component) {
            cout << node << " ";
        }
        cout << endl;
    }

    return 0;
}
```

Graph Construction:

```
int V = 5;
adj[1].push_back(3);   // 1 → 3
adj[1].push_back(2);   // 1 → 2
adj[3].push_back(4);   // 3 → 4
adj[4].push_back(5);   // 4 → 5
```

So the graph looks like:

```
1 → 2
↓
3 → 4 → 5
```

🔁 DFS Traversal (starting from unvisited nodes)

Looping over i = 1 to 5:

i	vis[i]	DFS Starts?	DFS Order (Component)
1	false	Yes	1 → 3 → 4 → 5, then 2 →
2	true	No	Already visited from 1
3	true	No	Already visited from 1
4	true	No	Already visited from 1
5	true	No	Already visited from 1

Note: 2 is visited after 1, since it's a neighbor of 1 and called later in the loop.

So only **one DFS call** is needed, and it covers **all reachable nodes from 1**.

📤 DFS Order (Component):

- From node 1: 1 → 3 → 4 → 5, and then the loop in DFS continues with 2.

So final traversal list:

1 3 4 5 2

🧾 Output:

1 3 4 5 2

Output:-
1 3 4 5 2

Dijkstra in C++

```cpp
#include <bits/stdc++.h>
using namespace std;

class Solution
{
public:
   // Function to find the shortest distance of all the vertices
   // from the source vertex S.
   vector<int> dijkstra(int V, vector<vector<int>> adj[], int S)
   {

       // Create a priority queue for storing the nodes as a pair {dist,node}
       // where dist is the distance from source to the node.
       priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>> pq;

       // Initialising distTo list with a large number to
       // indicate the nodes are unvisited initially.
       // This list contains distance from source to the nodes.
       vector<int> distTo(V, INT_MAX);

       // Source initialised with dist=0.
       distTo[S] = 0;
       pq.push({0, S});

       // Now, pop the minimum distance node first from the min-heap
       // and traverse for all its adjacent nodes.
       while (!pq.empty())
       {
           int node = pq.top().second;
           int dis = pq.top().first;
           pq.pop();

           // Check for all adjacent nodes of the popped out
           // element whether the prev dist is larger than current or not.
           for (auto it : adj[node])
           {
               int v = it[0];
               int w = it[1];
               if (dis + w < distTo[v])
               {
                   distTo[v] = dis + w;

                   // If current distance is smaller,
                   // push it into the queue.
                   pq.push({dis + w, v});
               }
           }
       }
       // Return the list containing shortest distances
```

Graph Setup

Given:

- **Vertices (V):** 3
- **Source (S):** 2
- **Adjacency list (adj):**

adj[0] = {{1, 1}, {2, 6}};
adj[1] = {{2, 3}, {0, 1}};
adj[2] = {{1, 3}, {0, 6}};

This translates to:

From	To	Weight
0	1	1
0	2	6
1	2	3
1	0	1
2	1	3
2	0	6

🔁 Dijkstra's Algorithm

Start from source 2, initialize:

distTo = [∞, ∞, 0]
pq = [(0, 2)]

Now iterate:

Step	Node	Pop (dist,node)	Neighbors	Update Distances	pq After
1	2	(0, 2)	(1,3), (0,6)	dist[1] = 3, dist[0] = 6	(3,1), (6,0)
2	1	(3, 1)	(2,3), (0,1)	dist[0] = min(6, 4) = 4	(4,0), (6,0)
3	0	(4, 0)	(1,1), (2,6)	dist[1] already 3 < 5 → skip	(6,0)
4	0	(6, 0)	-	Already visited with smaller	—

🧾 Final Distance Array:

res = [4, 3, 0]

Means:

Vertex	Shortest Distance from Source (2)
0	4

```
        // from source to all the nodes.
        return distTo;
    }
};

int main()
{
    // Driver code.
    int V = 3, E = 3, S = 2;
    vector<vector<int>> adj[V];
    vector<vector<int>> edges;
    vector<int> v1{1, 1}, v2{2, 6}, v3{2, 3}, v4{0, 1},
v5{1, 3}, v6{0, 6};
    int i = 0;
    adj[0].push_back(v1);
    adj[0].push_back(v2);
    adj[1].push_back(v3);
    adj[1].push_back(v4);
    adj[2].push_back(v5);
    adj[2].push_back(v6);

    Solution obj;
    vector<int> res = obj.dijkstra(V, adj, S);

    for (int i = 0; i < V; i++)
    {
        cout << res[i] << " ";
    }
    cout << endl;
    return 0;
}
```

Vertex	Shortest Distance from Source (2)
1	3
2	0 (source itself)

🖨 **Output:**

4 3 0

Output:-
4 3 0

Disjoint Set in C++

```cpp
#include <bits/stdc++.h>
using namespace std;

vector<int> parent, rankVec; // Renamed rank to
rankVec

void makeSet(int n) {
    parent.resize(n + 1);
    rankVec.resize(n + 1, 0); // Use rankVec here
    for (int i = 0; i <= n; i++) {
        parent[i] = i;
    }
}

int findUPar(int node) {
    if (node == parent[node])
        return node;
    return parent[node] = findUPar(parent[node]);
}

void unionByRank(int u, int v) {
    int ulp_u = findUPar(u); // ultimate parent of u
    int ulp_v = findUPar(v); // ultimate parent of v
    if (ulp_u == ulp_v) return; // already in the same set

    // Union by rank
    if (rankVec[ulp_u] < rankVec[ulp_v]) { // Use
rankVec here
        parent[ulp_u] = ulp_v;
    }
    else if (rankVec[ulp_u] > rankVec[ulp_v]) { // Use
rankVec here
        parent[ulp_v] = ulp_u;
    }
    else {
        parent[ulp_v] = ulp_u;
        rankVec[ulp_u]++; // Use rankVec here
    }
}

int main() {
    int n = 7; // Number of elements
    makeSet(n);

    unionByRank(1, 2);
    unionByRank(2, 3);
    unionByRank(4, 5);
    unionByRank(6, 7);
    unionByRank(5, 6);

    // Check if 3 and 7 are in the same set
    if (findUPar(3) == findUPar(7)) {
        cout << "Same\n";
    } else {
        cout << "Not same\n";
    }

    unionByRank(3, 7);

    // Check again if 3 and 7 are in the same set
    if (findUPar(3) == findUPar(7)) {
```

Initial Setup

You're working with n = 7, i.e., elements from 1 to 7.

makeSet(n):

- parent[i] = i for all i ∈ [0, 7]
- rankVec[i] = 0 initially

✅ Union Operations

Step	Operation	Resulting Union	Parent Array	Rank Array (rankVec)
1	union(1, 2)	1 becomes parent of 2	[0, 1, 1, 3, 4, 5, 6, 7]	[0, 1, 0, 0, 0, 0, 0, 0]
2	union(2, 3)	1 becomes parent of 3 (via 2)	[0, 1, 1, 1, 4, 5, 6, 7]	[0, 1, 0, 0, 0, 0, 0, 0]
3	union(4, 5)	4 becomes parent of 5	[0, 1, 1, 1, 4, 4, 6, 7]	[0, 1, 0, 0, 1, 0, 0, 0]
4	union(6, 7)	6 becomes parent of 7	[0, 1, 1, 1, 4, 4, 6, 6]	[0, 1, 0, 0, 1, 0, 1, 0]
5	union(5, 6)	4 becomes parent of 6 (via 5)	[0, 1, 1, 1, 4, 4, 4, 6]	[0, 1, 0, 0, 2, 0, 1, 0]

❓ First Check: findUPar(3) vs findUPar(7)

- findUPar(3) → follows to 1
- findUPar(7) → 7 → 6 → 4
- So: **1 != 4** → Output: **Not same**

🔁 union(3, 7)

- Ultimate parents: 1 and 4
- Both have rank 2 → tie, choose one (say 1) as parent, and increment rank

Result	Updated Parent Array	Updated Rank Array
1 becomes parent of 4	[0, 1, 1, 1, 1, 4, 4, 6]	[0, 3, 0, 0, 2, 0, 1, 0]

❓ Second Check: findUPar(3) vs findUPar(7)

<table>
<tr>
<td>

```
        cout << "Same\n";
    } else {
        cout << "Not same\n";
    }

    return 0;
}
```

</td>
<td>

- findUPar(3) → 1
- findUPar(7) → 7 → 6 → 4 → 1
- So: **1 == 1** → Output: **Same**

✅ **Final Output**

Not same
Same

</td>
</tr>
<tr>
<td colspan="2">

Output:-
Not same
Same

</td>
</tr>
</table>

Find eventual safe state in C++

```
#include <bits/stdc++.h>
using namespace std;
class Solution {
private:
   bool dfsCheck(int node, vector<int> adj[], int vis[],
 int pathVis[],
       int check[]) {
       vis[node] = 1;
       pathVis[node] = 1;
       check[node] = 0;
       // traverse for adjacent nodes
       for (auto it : adj[node]) {
           // when the node is not visited
           if (!vis[it]) {
           if (dfsCheck(it, adj, vis, pathVis, check) == true) {
                   check[node] = 0;
                   return true;
               }

           }
           // if the node has been previously visited
           // but it has to be visited on the same path
           else if (pathVis[it]) {
               check[node] = 0;
               return true;
           }
       }
       check[node] = 1;
       pathVis[node] = 0;
       return false;
   }
public:
   vector<int> eventualSafeNodes(int V, vector<int>
adj[]) {
       int vis[V] = {0};
       int pathVis[V] = {0};
       int check[V] = {0};
       vector<int> safeNodes;
       for (int i = 0; i < V; i++) {
           if (!vis[i]) {
               dfsCheck(i, adj, vis, pathVis, check);
           }
       }
       for (int i = 0; i < V; i++) {
           if (check[i] == 1) safeNodes.push_back(i);
       }
       return safeNodes;
   }
};

int main() {

   //V = 12;
   vector<int> adj[12] = {{1}, {2}, {3}, {4, 5}, {6}, {6}, {7}, {},
{1, 9}, {10},
       {8},{9}};
   int V = 12;
   Solution obj;
   vector<int> safeNodes = obj.eventualSafeNodes(V,
adj);
   for (auto node : safeNodes) {
```

Goal

We want to find all the **eventual safe nodes** in a **directed graph**, i.e., nodes from which **every path eventually ends in a terminal node** (a node with no outgoing edges). This is solved using **DFS cycle detection**.

🔍 Key Concepts

- vis[] → marks if a node has been visited.
- pathVis[] → tracks the current recursion path.
- check[] → 1 if node is *safe*, 0 if not.

A node is **not safe** if:

- A cycle is detected starting from it (or reachable from it).

📊 Input Graph (Adjacency List)

0 → 1
1 → 2
2 → 3
3 → 4,5
4 → 6
5 → 6
6 → 7
7 → {}   ← terminal node
8 → 1,9
9 → 10
10 → 8
11 → 9

🔄 DFS Cycle Detection

Let's go through the DFS starting from each unvisited node:

Node	Path	Cycle Detected	Safe?
0	0→1→2→3→4→6→7	No	✅ Yes
1	Already visited from 0	-	✅ Yes
2	Already visited from 0	-	✅ Yes
3	Already visited from 0	-	✅ Yes
4	Already visited from 0	-	✅ Yes
5	5→6→7	No	✅ Yes
6	Already visited	-	✅ Yes
7	Terminal	No	✅ Yes
8	8→1→… (already	✅ Yes	❌ No

```
        cout << node << " ";
    }
    cout << endl
   return 0;
}
```

	visited) AND 8→9→10→8 (cycle)		
9	9→10→8→9	✅ Yes	❌ No
10	10→8→9→10	✅ Yes	❌ No
11	11→9→cycle	✅ Yes	❌ No

✅ Safe Nodes

From the table above, the safe nodes are:

0 1 2 3 4 5 6 7

Output:-

0 1 2 3 4 5 6 7

Floyd-Warshall in C++

```cpp
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
    void shortest_distance(vector<vector<int>>&matrix) {
        int n = matrix.size();
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                if (matrix[i][j] == -1) {
                    matrix[i][j] = 1e9;
                }
                if (i == j) matrix[i][j] = 0;
            }
        }

        for (int k = 0; k < n; k++) {
            for (int i = 0; i < n; i++) {
                for (int j = 0; j < n; j++) {
                    matrix[i][j] = min(matrix[i][j],
                                       matrix[i][k] + matrix[k][j]);
                }
            }
        }



        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                if (matrix[i][j] == 1e9) {
                    matrix[i][j] = -1;
                }
            }
        }
    }
};

int main() {

    int V = 4;
    vector<vector<int>> matrix(V, vector<int>(V, -1));
    matrix[0][1] = 2;
    matrix[1][0] = 1;
    matrix[1][2] = 3;
    matrix[3][0] = 3;
    matrix[3][1] = 5;
    matrix[3][2] = 4;

    Solution obj;
    obj.shortest_distance(matrix);

    for (auto row : matrix) {
        for (auto cell : row) {
            cout << cell << " ";
        }
        cout << endl;
    }

    return 0;
}
```

Objective

You are given a directed weighted graph in the form of an **adjacency matrix**. You are using the **Floyd-Warshall algorithm** to compute **shortest distances between every pair of vertices**.

📌 Input Matrix (after setup)

The initial matrix setup (after setting the given edges):

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

Converted to:

```
    0    1    2    3
0 |  0    2   1e9  1e9
1 |  1    0    3   1e9
2 | 1e9  1e9   0    1e9
3 |  3    5    4    0
```

🧠 Floyd-Warshall Algorithm Dry Run

We'll now go through each intermediate node k and update the matrix.

🔁 For k = 0

Try to go i → 0 → j

No new updates help here, as 0 is only connected to 1.

🔁 For k = 1

Try i → 1 → j:

- 0 → 1 → 2 = 2 + 3 = 5 → Update matrix[0][2] from 1e9 → 5
- 3 → 1 → 2 = 5 + 3 = 8 → Update matrix[3][2] from 4 → 4 (already smaller, no change)

🔁 **For k = 2**

Only relevant updates:

- 3 → 2 → 0 = 4 + 1e9 → no update
- Nothing meaningful added as 2 is a disconnected node

🔁 **For k = 3**

- 0 → 3 → 0 → Not reachable
- But let's try:
  - 0 → 3 → 2: matrix[0][3] + matrix[3][2] = 1e9 + 4 = 1e9 → No update
  - Same for others, no improvement.

✅ **Final Matrix (replace 1e9 with -1)**

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

📤 **Output**

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

Output:-
0 2 5 -1
1 0 3 -1
-1 -1 0 -1
3 5 4 0

Check graph is bipartite using Breadth First Search in C++

```
#include<bits/stdc++.h>
using namespace std;

class Solution {
   // colors a component
   private:
   bool check(int start, int V, vector<int>adj[], int
color[]) {
       queue<int> q;
       q.push(start);
       color[start] = 0;
       while(!q.empty()) {
          int node = q.front();
          q.pop();

          for(auto it : adj[node]) {
             // if the adjacent node is yet not colored
             // you will give the opposite color of the
node
             if(color[it] == -1) {

                color[it] = !color[node];
                q.push(it);
             }
             // is the adjacent guy having the same
color
             // someone did color it on some other path
             else if(color[it] == color[node]) {
                return false;
             }
          }
       }
       return true;
   }
public:
   bool isBipartite(int V, vector<int>adj[]){
       int color[V];
       for(int i = 0;i<V;i++) color[i] = -1;

       for(int i = 0;i<V;i++) {
          // if not coloured
          if(color[i] == -1) {
             if(check(i, V, adj, color) == false) {
                return false;
             }
          }
       }
       return true;
   }

};

void addEdge(vector <int> adj[], int u, int v) {
   adj[u].push_back(v);
   adj[v].push_back(u);
}

int main(){

   // V = 4, E = 4
   vector<int>adj[4];
```

Graph Structure

Vertices: V = 4
Edges:

- 0 ↔ 2
- 0 ↔ 3
- 2 ↔ 3
- 3 ↔ 1

Adjacency List:

0: [2, 3]
1: [3]
2: [0, 3]
3: [0, 2, 1]

🧪 Dry Run of check() Function (BFS for Coloring)

We want to color the graph with **2 colors (0 and 1)** such that no two adjacent nodes have the same color.

Step	Node	Queue	Color Status	Action
1	0	[0]	[-1, -1, -1, -1]	Start BFS with node 0 → color[0] = 0
2	0	[2, 3]	[0, -1, 1, 1]	2 & 3 uncolored → assign opposite color
3	2	[3]	[0, -1, 1, 1]	0 already colored & valid → continue
4	2	[3]	[0, -1, 1, 1]	3 already colored **with same color →** 🚨
				Conflict found → graph is **not bipartite**

❌ Output:

0

<pre> addEdge(adj, 0, 2); addEdge(adj, 0, 3); addEdge(adj, 2, 3); addEdge(adj, 3, 1); Solution obj; bool ans = obj.isBipartite(4, adj); if(ans)cout << "1\n"; else cout << "0\n"; return 0; }</pre>	
Output:- **0**	

Cycle detection in undirected graph using Breadth First Search in C++

```
#include <bits/stdc++.h>
using namespace std;
class Solution {
public:
    // Function to detect cycle in a directed graph.
    bool isCyclic(int V, vector<int> adj[]) {
        int indegree[V] = {0};
        for (int i = 0; i < V; i++) {
            for (auto it : adj[i]) {
                indegree[it]++;
            }
        }
        queue<int> q;
        for (int i = 0; i < V; i++) {
            if (indegree[i] == 0) {
                q.push(i);
            }
        }
        int cnt = 0;
        // o(v + e)
        while (!q.empty()) {
            int node = q.front();
            q.pop();
            cnt++;
            // node is in your topo sort
            // so please remove it from the indegree
            for (auto it : adj[node]) {
                indegree[it]--;
                if (indegree[it] == 0) q.push(it);
            }
        }

        if (cnt == V) return false;
        return true;
    }
};
int main() {
    //V = 6;
    vector<int> adj[6] = {{}, {2}, {3}, {4, 5}, {2}, {}};
    int V = 6;
    Solution obj;
    bool ans = obj.isCyclic(V, adj);
    if (ans) cout << "True";
    else cout << "Flase";
    cout << endl;
    return 0;
}
```

Graph Details

From your adj array:

```
vector<int> adj[6] = {
    {},      // 0
    {2},     // 1 → 2
    {3},     // 2 → 3
    {4, 5},  // 3 → 4, 5
    {2},     // 4 → 2  ← Cycle!
    {}       // 5
};
```

🔢 **Number of vertices: V = 6**

🧮 **Step 1: Calculate In-Degrees**

Node	Incoming Edges	in-degree
0	—	0
1	—	0
2	from 1, 4	2
3	from 2	1
4	from 3	1
5	from 3	1

📌 **Initial in-degree array**: [0, 0, 2, 1, 1, 1]

📥 **Step 2: Initialize Queue with in-degree = 0**

q = [0, 1]   // because indegree[0] = 0 and indegree[1] = 0

🔄 **Step 3: BFS Traversal & Count Nodes Processed**

Iteration	Queue	Node Popped	Neighbors	Action	Updated in-degree	Count
1	[0,1]	0	—	No neighbors	[0, 0, 2, 1, 1, 1]	1
2	[1]	1	[2]	indegree[2] = 2 → 1 (not zero yet)	[0, 0, 1, 1, 1, 1]	2
3	[]	—	—	Queue is empty — loop ends		2

🛑 **Step 4: Final Check**

- Nodes processed (cnt) = 2
- Total nodes (V) = 6

	📌 Since cnt != V, there **is a cycle** in the graph.

Output:-
True
The graph contains a cycle

Cycle detection in undirected graph using Breadth First Search in C++

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
  private:
  bool detect(int src, vector<int> adj[], int vis[])
{
      vis[src] = 1;
      // store <source node, parent node>
      queue<pair<int,int>> q;
      q.push({src, -1});
      // traverse until queue is not empty
      while(!q.empty()) {
          int node = q.front().first;
          int parent = q.front().second;
          q.pop();

          // go to all adjacent nodes
          for(auto adjacentNode: adj[node]) {
              // if adjacent node is unvisited
              if(!vis[adjacentNode]) {
                  vis[adjacentNode] = 1;
                  q.push({adjacentNode, node});
              }
              // if adjacent node is visited and is not
it's own parent node
              else if(parent != adjacentNode) {
                  // yes it is a cycle
                  return true;
              }
          }
      }
      // there's no cycle
      return false;
  }
  public:
    // Function to detect cycle in an undirected
graph.
    bool isCycle(int V, vector<int> adj[]) {
        // initialise them as unvisited
        int vis[V] = {0};
        for(int i = 0;i<V;i++) {
            if(!vis[i]) {
                if(detect(i, adj, vis)) return true;
            }
        }
        return false;
    }
};

int main() {
    // V = 4, E = 2
    vector<int> adj[4] = {{}, {2}, {1, 3}, {2}};
    Solution obj;
    bool ans = obj.isCycle(4, adj);
    if (ans)
        cout << "1\n";
    else
        cout << "0\n";
    return 0;
}
```

Graph Definition (Adjacency List)

```
vector<int> adj[4] = {
    {},       // 0 → No neighbors
    {2},      // 1 → 2
    {1, 3},   // 2 → 1, 3
    {2}       // 3 → 2
};
```

Visual graph:

1 -- 2 -- 3

- It's a **linear graph**, no cycle expected.

🧠 Variables

- vis[4] = {0, 0, 0, 0} (all unvisited initially)
- Queue for BFS: stores pairs {node, parent}

🔁 Step-by-Step Traversal Table

Iter	Queue	node	parent	Neighbours	Action
1	{1, -1}	1	-1	[2]	2 is unvisited → mark visited, enqueue {2, 1}
2	{2, 1}	2	1	[1, 3]	1 is parent → skip; 3 is unvisited → mark visited, enqueue {3, 2}
3	{3, 2}	3	2	[2]	2 is parent → skip
4	empty	—	—	—	Loop ends

Visited array after traversal: [0, 1, 1, 1]

No condition parent != adjacentNode && vis[adjacentNode] == 1 was met.

✅ Final Output

0 // No cycle found

📋 Summary Table

Node	Parent	Visited	Notes
1	-1	✅	Starting node
2	1	✅	Connected from node 1

Node	Parent	Visited	Notes
3	2	✓	Connected from node 2
0	-	✗	Isolated node (not connected)

🧠 **Conclusion**

- **No cycle** detected — the output is 0.

Output:-
0
No cycle was found in any component of the graph

Breadth First Search in C++

```
#include <iostream>
#include <vector>
#include <queue>
#include <deque>
using namespace std;
// Function to add an edge between two vertices u and v
void addEdge(vector<vector<int>>& adj, int u, int v) {
    adj[u].push_back(v);
    adj[v].push_back(u);
}
// Function to perform BFS traversal
void bfs(vector<vector<int>>& adj, int v, int s)
{
    deque<int> q;
    vector<bool> visited(v, false);
    q.push_back(s);
    visited[s] = true;
    while (!q.empty()) {
        int rem = q.front();
        q.pop_front();
        cout << rem << " ";
        for (int nbr : adj[rem]) {
            if (!visited[nbr]) {
                visited[nbr] = true;
                q.push_back(nbr);
            }
        }
    }
    cout << endl; // Print newline after traversal
}
int main() {
    int V = 7;
    vector<vector<int>> adj(V);
    // Adding edges to the graph
    addEdge(adj, 0, 1);
    addEdge(adj, 0, 2);
    addEdge(adj, 2, 3);
    addEdge(adj, 1, 3);
    addEdge(adj, 1, 4);
    addEdge(adj, 3, 4);
    cout << "Following is Breadth First Traversal: \n";
    bfs(adj, V, 0);
    return 0;
}
```

Graph Structure

Adjacency List:

0: [1, 2]
1: [0, 3, 4]
2: [0, 3]
3: [2, 1, 4]
4: [1, 3]
5: []
6: []

(Nodes 5 and 6 are isolated)

🧠 **BFS Dry Run Table**

Step	Queue	Visited Nodes	Node Processed	Neighbors Added	Output
1	[0]	{}	-	-	
2	[1, 2]	{0}	0	1, 2	0
3	[2, 3, 4]	{0, 1}	1	3, 4 (0 already done)	0 1
4	[3, 4]	{0, 1, 2}	2	- (0, 3 already done)	0 1 2
5	[4]	{0,1,2,3}	3	- (2,1,4 already done)	0 1 2 3
6	[]	{0,1,2,3,4}	4	- (1,3 already done)	0 1 2 3 4

📜 **Final Output**

Following is Breadth First Traversal:
0 1 2 3 4

Output:-
0 1 2 3 4

Cycle detection in undirected graph using Depth First Search in C++

```cpp
#include <bits/stdc++.h>
using namespace std;

class Solution {
  private:
    bool dfs(int node, int parent, int vis[], vector<int>
adj[]) {
        vis[node] = 1;
        // visit adjacent nodes
        for(auto adjacentNode: adj[node]) {
            // unvisited adjacent node
            if(!vis[adjacentNode]) {
                if(dfs(adjacentNode, node, vis, adj) == true)
                    return true;
            }
            // visited node but not a parent node
            else if(adjacentNode != parent) return true;
        }
        return false;
    }
  public:
    // Function to detect cycle in an undirected graph.
    bool isCycle(int V, vector<int> adj[]) {
       int vis[V] = {0};
       // for graph with connected components
       for(int i = 0;i<V;i++) {
           if(!vis[i]) {
               if(dfs(i, -1, vis, adj) == true) return true;
           }
       }
       return false;
    }
};

int main() {

    // V = 4, E = 2
    vector<int> adj[4] = {{}, {2}, {1, 3}, {2}};
    Solution obj;
    bool ans = obj.isCycle(4, adj);
    if (ans)
        cout << "1\n";
    else
        cout << "0\n";
    return 0;
}
```

Input Graph (Adjacency List)

```cpp
vector<int> adj[4] = {
  {},        // 0 → no connections
  {2},       // 1 → connected to 2
  {1, 3},    // 2 → connected to 1 and 3
  {2}        // 3 → connected to 2
};
```

Graph in visual form:

1 -- 2 -- 3

(0 is isolated and not connected to any node.)

🧠 DFS Function Signature

```cpp
bool dfs(int node, int parent, int vis[],
vector<int> adj[]);
```

- node: current node being explored
- parent: node from which we came
- vis[]: visited array
- adj[]: adjacency list

📊 Dry Run Table

Initial:

- vis[4] = {0, 0, 0, 0}

DFS Call Stack Trace

Call	Node	Parent	Visited Array	Action
1	0	-1	[1, 0, 0, 0]	No neighbors → return false
2	1	-1	[1, 1, 0, 0]	Visit 2 from 1
3	2	1	[1, 1, 1, 0]	1 is parent → skip; visit 3
4	3	2	[1, 1, 1, 1]	2 is parent → skip; DFS returns false
3↑	2	1	[1, 1, 1, 1]	DFS from 3 returned false → continue → DFS returns false
2↑	1	-1	[1, 1, 1, 1]	DFS from 2 returned false → continue → DFS

	<table><tr><td></td><td></td><td></td><td></td><td>returns false</td></tr></table> ✅ **Final State** • All nodes visited: vis = [1, 1, 1, 1] • No back-edge found (no adjacent visited node that's not the parent) 📜 **Output:** 0
Output:- **0** **No cycle**	

Depth First Search in C++

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
  public:
    // Function to return Breadth First Traversal of given graph.
    vector<int> bfsOfGraph(int V, vector<int> adj[]) {
        int vis[V] = {0};
        vis[0] = 1;
        queue<int> q;
        // push the initial starting node
        q.push(0);
        vector<int> bfs;
        // iterate till the queue is empty
        while(!q.empty()) {
           // get the topmost element in the queue
            int node = q.front();
            q.pop();
            bfs.push_back(node);
            // traverse for all its neighbours
            for(auto it : adj[node]) {
                // if the neighbour has previously not been visited,
                // store in Q and mark as visited
                if(!vis[it]) {
                    vis[it] = 1;
                    q.push(it);
                }
            }
        }
        return bfs;
    }
};

void addEdge(vector<int> adj[], int u, int v) {
    adj[u].push_back(v);
    adj[v].push_back(u);
}

void printAns(vector <int> &ans) {
    for (int i = 0; i < ans.size(); i++) {
        cout << ans[i] << " ";
    }
}

int main()
{
    vector<int> adj[6];

    addEdge(adj, 0, 1);
    addEdge(adj, 1, 2);
    addEdge(adj, 1, 3);
    addEdge(adj, 0, 4);

    Solution obj;
    vector <int> ans = obj.bfsOfGraph(5, adj);
    printAns(ans);

    return 0;
}
```

Graph Definition (Adjacency List)

```
vector<int> adj[6];
addEdge(adj, 0, 1);
addEdge(adj, 1, 2);
addEdge(adj, 1, 3);
addEdge(adj, 0, 4);
```

Adjacency List:

0 → [1, 4]
1 → [0, 2, 3]
2 → [1]
3 → [1]
4 → [0]

🧠 BFS Variables

- vis[5] = {1, 0, 0, 0, 0} → Only node 0 marked visited initially
- Queue: q = [0]
- Result vector: bfs = []

🔁 BFS Traversal Table

Step	Queue	Node Popped	BFS List	Neighbors	Action
1	[0]	0	[0]	[1, 4]	Visit 1 & 4 → mark visited, enqueue → Queue: [1, 4]
2	[1, 4]	1	[0, 1]	[0, 2, 3]	0 already visited; Visit 2 & 3 → mark visited, enqueue → Queue: [4, 2, 3]
3	[4, 2, 3]	4	[0, 1, 4]	[0]	0 already visited → nothing added
4	[2, 3]	2	[0, 1, 4, 2]	[1]	1 already visited
5	[3]	3	[0, 1, 4, 2, 3]	[1]	1 already visited
6	[]	-	Done	-	Queue

					empty → BFS complete

✅ **Final BFS Output**

[0, 1, 4, 2, 3]

🧠 **Summary Table**

Node	Visited	Enqueued	When
0	✅	✅	Start
1	✅	✅	From 0
4	✅	✅	From 0
2	✅	✅	From 1
3	✅	✅	From 1

📌 **Output on Console:**

0 1 4 2 3

Output:-
0 1 4 2 3

Kahn in C++

```cpp
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
    //Function to return list containing vertices in
Topological order.
    vector<int> topoSort(int V, vector<int> adj[])
    {
        int indegree[V] = {0};
        for (int i = 0; i < V; i++) {
            for (auto it : adj[i]) {
                indegree[it]++;
            }
        }

        queue<int> q;
        for (int i = 0; i < V; i++) {
            if (indegree[i] == 0) {
                q.push(i);
            }
        }
        vector<int> topo;
        while (!q.empty()) {
            int node = q.front();
            q.pop();
            topo.push_back(node);
            // node is in your topo sort
            // so please remove it from the indegree

            for (auto it : adj[node]) {
                indegree[it]--;
                if (indegree[it] == 0) q.push(it);
            }
        }

        return topo;
    }
};

int main() {

    //V = 6;
    vector<int> adj[6] = {{}, {}, {3}, {1}, {0, 1}, {0, 2}};
    int V = 6;
    Solution obj;
    vector<int> ans = obj.topoSort(V, adj);

    for (auto node : ans) {
        cout << node << " ";
    }
    cout << endl;

    return 0;
}
```

Input Graph (Adjacency List)

```cpp
vector<int> adj[6] = {
    {},       // 0
    {},       // 1
    {3},      // 2 → 3
    {1},      // 3 → 1
    {0, 1},   // 4 → 0, 1
    {0, 2}    // 5 → 0, 2
};
```

🔢 Step 1: Calculate In-Degree of Each Node

Node	Incoming Edges from	In-degree
0	4, 5	2
1	3, 4	2
2	5	1
3	2	1
4	-	0
5	-	0

➡ Initial indegree[] = {2, 2, 1, 1, 0, 0}

📥 Step 2: Enqueue All Nodes With In-degree = 0

Initial Queue: q = [4, 5]

🔁 Step 3: BFS Loop & Topological Sorting

Iteration	Node Popped	Topo List	Decrease In-degree	Queue after Push
1	4	[4]	0→1, 1→1	[5]
2	5	[4, 5]	0→0 ✅, 2→0 ✅	[0, 2]
3	0	[4, 5, 0]	-	[2]
4	2	[4, 5, 0, 2]	3→0 ✅	[3]
5	3	[4, 5, 0, 2,	1→0 ✅	[1]

Iteration	Node Popped	Topo List	Decrease In-degree	Queue after Push
		3]		
6	1	[4, 5, 0, 2, 3, 1]	-	[] (done)

✅ Final Output

Topological Order = [4, 5, 0, 2, 3, 1]

🧠 Summary Table

Node	Final In-degree	Status
0	0	Printed
1	0	Printed
2	0	Printed
3	0	Printed
4	0	Printed
5	0	Printed

Output:-
4 5 0 2 3 1

Kruskal in C++

```cpp
#include <bits/stdc++.h>
using namespace std;


class DisjointSet {
    vector<int> rank, parent, size;
public:
    DisjointSet(int n) {
        rank.resize(n + 1, 0);
        parent.resize(n + 1);
        size.resize(n + 1);
        for (int i = 0; i <= n; i++) {
            parent[i] = i;
            size[i] = 1;
        }
    }

    int findUPar(int node) {
        if (node == parent[node])
            return node;
        return parent[node] =
findUPar(parent[node]);
    }

    void unionByRank(int u, int v) {
        int ulp_u = findUPar(u);
        int ulp_v = findUPar(v);
        if (ulp_u == ulp_v) return;
        if (rank[ulp_u] < rank[ulp_v]) {
            parent[ulp_u] = ulp_v;
        }
        else if (rank[ulp_v] < rank[ulp_u]) {
            parent[ulp_v] = ulp_u;
        }
        else {
            parent[ulp_v] = ulp_u;
            rank[ulp_u]++;
        }
    }

    void unionBySize(int u, int v) {
        int ulp_u = findUPar(u);
        int ulp_v = findUPar(v);
        if (ulp_u == ulp_v) return;
        if (size[ulp_u] < size[ulp_v]) {
            parent[ulp_u] = ulp_v;
            size[ulp_v] += size[ulp_u];
        }
        else {
            parent[ulp_v] = ulp_u;
            size[ulp_u] += size[ulp_v];
        }
    }
};
class Solution
{
public:
    //Function to find sum of weights of edges
of the Minimum Spanning Tree.
    int spanningTree(int V,
vector<vector<int>> adj[])
```

Input

You are given:

```
V = 5;
edges = {
    {0, 1, 2},
    {0, 2, 1},
    {1, 2, 1},
    {2, 3, 2},
    {3, 4, 1},
    {4, 2, 2}
};
```

📦 Step 1: Adjacency List Construction (Undirected Graph)

adj[i] stores {neighbour, weight}:

Node	**Adjacents**
0	[1, 2], [2, 1]
1	[0, 2], [2, 1]
2	[0, 1], [1, 1], [3, 2], [4, 2]
3	[2, 2], [4, 1]
4	[3, 1], [2, 2]

🧮 Step 2: Edge List Formation

Collected as {weight, {u, v}} (both directions included):

Edge	Format
0-1	{2, {0, 1}}
0-2	{1, {0, 2}}
1-2	{1, {1, 2}}
2-3	{2, {2, 3}}
3-4	{1, {3, 4}}
4-2	{2, {4, 2}}
🔁 duplicates (undirected, so reverse edges too!)	

🔽 Step 3: Sort Edges by Weight

Sorted edges:

```
edges = {
  {1, {0, 2}},
  {1, {1, 2}},
  {1, {3, 4}},
  {2, {0, 1}},
  {2, {2, 3}},
  {2, {4, 2}}
}
```

```
    {
        // 1 - 2 wt = 5
        /// 1 - > (2, 5)
        // 2 -> (1, 5)

        // 5, 1, 2
        // 5, 2, 1
        vector<pair<int, pair<int, int>>> edges;
        for (int i = 0; i < V; i++) {
            for (auto it : adj[i]) {
                int adjNode = it[0];
                int wt = it[1];
                int node = i;

                edges.push_back({wt, {node,
adjNode}});
            }
        }
        DisjointSet ds(V);
        sort(edges.begin(), edges.end());
        int mstWt = 0;
        for (auto it : edges) {
            int wt = it.first;
            int u = it.second.first;
            int v = it.second.second;

            if (ds.findUPar(u) != ds.findUPar(v)) {
                mstWt += wt;
                ds.unionBySize(u, v);
            }
        }

        return mstWt;
    }
};

int main() {

    int V = 5;
    vector<vector<int>> edges = {{0, 1, 2}, {0,
2, 1}, {1, 2, 1}, {2, 3, 2}, {3, 4, 1}, {4, 2, 2}};
    vector<vector<int>> adj[V];
    for (auto it : edges) {
        vector<int> tmp(2);
        tmp[0] = it[1];
        tmp[1] = it[2];
        adj[it[0]].push_back(tmp);

        tmp[0] = it[0];
        tmp[1] = it[2];
        adj[it[1]].push_back(tmp);
    }

    Solution obj;
    int mstWt = obj.spanningTree(V, adj);
    cout << "The sum of all the edge weights: "
<< mstWt << endl;
    return 0;
}
```

🛠 Step 4: Disjoint Set Initialization

- Each node starts as its own parent.
- parent[] = {0, 1, 2, 3, 4}
- size[] = {1, 1, 1, 1, 1}

🔁 Step 5: Process Edges

Edge	Find UParent(u)	Find UParent(v)	Cycle?	Union?	MST Weight
{1, {0, 2}}	0	2	No	Union(0, 2)	1
{1, {1, 2}}	1	0 (from 2)	No	Union(1, 0)	2
{1, {3, 4}}	3	4	No	Union(3, 4)	3
{2, {0, 1}}	0	0	**Yes**	✕ Skip	3
{2, {2, 3}}	0	3	No	Union(0, 3)	5
{2, {4, 2}}	0	0	**Yes**	✕ Skip	5

✅ Final MST Weight

The sum of all the edge weights: 5

🧠 Disjoint Set Status (Final)

Node	Parent
0	0
1	0
2	0
3	0
4	0

All nodes are connected — ✅ valid spanning tree.

Output:-
The sum of all the edge weights: 5

No of provinces in C++

```cpp
#include <bits/stdc++.h>
using namespace std;

class Solution {
  private:
    // dfs traversal function
    void dfs(int node, vector<int> adjLs[], int vis[]) {
        // mark the more as visited
        vis[node] = 1;
        for(auto it: adjLs[node]) {
            if(!vis[it]) {
                dfs(it, adjLs, vis);
            }
        }
    }
  public:
    int numProvinces(vector<vector<int>> adj, int V) {
        vector<int> adjLs[V];

        // to change adjacency matrix to list
        for(int i = 0;i<V;i++) {
            for(int j = 0;j<V;j++) {
                // self nodes are not considered
                if(adj[i][j] == 1 && i != j) {
                    adjLs[i].push_back(j);
                    adjLs[j].push_back(i);
                }
            }
        }
        int vis[V] = {0};
        int cnt = 0;
        for(int i = 0;i<V;i++) {
            // if the node is not visited
            if(!vis[i]) {
                // counter to count the number of provinces
                cnt++;
               dfs(i, adjLs, vis);
            }
        }
        return cnt;

    }
};

int main() {

    vector<vector<int>> adj
    {
        {1, 0, 1},
        {0, 1, 0},
        {1, 0, 1}
    };

    Solution ob;
    cout << ob.numProvinces(adj,3) << endl;

    return 0;
}
```

Input:

```
adj = {
  {1, 0, 1},
  {0, 1, 0},
  {1, 0, 1}
};
V = 3
```

✅ Adjacency Matrix ➡ List Conversion:

i	j	adj[i][j]	i != j	Action	adjLs
0	0	1	❌	skip	
0	1	0	✅	skip	
0	2	1	✅	add edge 0–2 and 2–0	0→[2], 2→[0]
1	0	0	✅	skip	
1	1	1	❌	skip	
1	2	0	✅	skip	
2	0	1	✅	already added	
2	1	0	✅	skip	
2	2	1	❌	skip	

🔧 Final Adjacency List:

0 → [2]
1 → []
2 → [0]

🚀 DFS + Province Counting

i	vis[i]	Action	DFS Called	Updated vis	cnt
0	0	Not visited → DFS(0)	✓	[1, 0, 1]	1
1	0	Not visited → DFS(1)	✓	[1, 1, 1]	2
2	1	Already visited	✕	-	-

🔄 DFS Traversal Details

◆ **DFS(0)**

node	vis[node]	Neighbors	Action	vis
0	0 → 1	2	DFS(2)	[1, 0, 0]
2	0 → 1	0	Already vis	[1, 0, 1]

◆ **DFS(1)**

node	vis[node]	Neighbors	Action	vis
1	0 → 1	none	Done	[1, 1, 1]

📜 Final Result

Variable	Value
cnt	2 (Answer)
vis	[1, 1, 1]

🟩 **Output: 2 provinces**

Output:-
2

Prim in C++

```
#include <bits/stdc++.h>
using namespace std;

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, int>,
		               vector<pair<int, int> >, greater<pair<int, int>>> pq;

		vector<int> vis(V, 0);
		// {wt, node}
		pq.push({0, 0});
		int sum = 0;
		while (!pq.empty()) {
			auto it = pq.top();
			pq.pop();
			int node = it.second;
			int wt = it.first;

			if (vis[node] == 1) continue;
			// add it to the mst
			vis[node] = 1;
			sum += wt;
			for (auto it : adj[node]) {
				int adjNode = it[0];
				int edW = it[1];
				if (!vis[adjNode]) {
					pq.push({edW, adjNode});
				}
			}
		}
		return sum;
	}
};


int main() {

	int V = 5;
	vector<vector<int>> edges = {{0, 1, 2}, {0, 2, 1}, {1, 2, 1}, {2, 3, 2}, {3, 4, 1}, {4, 2, 2}};
	vector<vector<int>> adj[V];
	for (auto it : edges) {
		vector<int> tmp(2);
		tmp[0] = it[1];
		tmp[1] = it[2];
		adj[it[0]].push_back(tmp);

		tmp[0] = it[0];
		tmp[1] = it[2];
		adj[it[1]].push_back(tmp);
	}

	Solution obj;
```

Input Edges

```
edges = {
  {0, 1, 2},
  {0, 2, 1},
  {1, 2, 1},
  {2, 3, 2},
  {3, 4, 1},
  {4, 2, 2}
}
```

🔁 Adjacency List

Node	Neighbors
0	[1,2], [2,1]
1	[0,2], [2,1]
2	[0,1], [1,1], [3,2], [4,2]
3	[2,2], [4,1]
4	[3,1], [2,2]

🧠 Prim's MST Logic (Min-Heap)

We track:

- pq: min-heap for {weight, node}
- vis[]: visited array
- sum: total MST weight

📊 Dry Run Table

Step	pq (Min-Heap)	node	wt	vis	sum	Action Taken
1	{(0, 0)}	0	0	[1, 0, 0, 0, 0]	0	Add node 0, add neighbors 1 (wt=2), 2 (wt=1) to pq
2	{(1, 2), (2, 1)}	2	1	[1, 0, 1, 0, 0]	1	Add node 2, add unvisited neighbors: 1(wt=1), 3(wt=2), 4(wt=2)
3	{(1, 1), (2, 1), (2, 3), (2, 4)}	1	1	[1, 1, 1, 0, 0]	2	Add node 1, skip already visited 0 & 2
4	{(2, 1), (2, 3), (2, 4)}	1	2	Already visited	-	Skip

```
        int sum = obj.spanningTree(V, adj);
        cout << "The sum of all the edge weights: " << sum << endl;

        return 0;
}
```

Step	pq (Min-Heap)	node	wt	vis	sum	Action Taken
5	{(2, 3), (2, 4)}	3	2	[1, 1, 1, 1, 0]	4	Add node 3, add neighbor 4 (wt=1)
6	{(1, 4), (2, 4)}	4	1	[1, 1, 1, 1, 1]	5	Add node 4, skip visited 3, 2
7	{(2, 4)}	4	2	Already visited	-	Skip

✅ **Final Result:**

Variable	Value
sum	**5**
vis	[1,1,1,1,1] (All visited)

✅ **Output:**

The sum of all the edge weights: 5

Output:-
The sum of all the edge weights: 5

Reverse directed graph in C++

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

class ReverseDirectedGraph {
public:
    static vector<vector<int>>
reverseDirectedGraph(const vector<vector<int>>& adj,
int V) {
        vector<vector<int>> reversedAdj(V + 1);

        for (int i = 0; i <= V; ++i) {
            for (int j : adj[i]) {
                reversedAdj[j].push_back(i);
            }
        }

        return reversedAdj;
    }

    static void printGraph(const vector<vector<int>>&
graph, int V) {
        for (int i = 1; i <= V; ++i) {
            for (int j : graph[i]) {
                cout << i << " -> " << j << endl;
            }
        }
    }
};

int main() {
    int V = 5;
    vector<vector<int>> adj(V + 1);

    adj[1].push_back(3);
    adj[1].push_back(2);
    adj[3].push_back(4);
    adj[4].push_back(5);

    vector<vector<int>> reversedAdj =
ReverseDirectedGraph::reverseDirectedGraph(adj, V);

    cout << "Reversed Graph:" << endl;
    ReverseDirectedGraph::printGraph(reversedAdj, V);

    return 0;
}
```

Original Input Graph (Adjacency List)

We have a **directed graph** with 5 vertices (V = 5):

Vertex	**Edges**
1	→ 3, → 2
2	—
3	→ 4
4	→ 5
5	—

Graphically:

1 → 2
↓
3 → 4 → 5

🔁 Dry Run Table: reverseDirectedGraph(adj, V)

This function creates a reversed adjacency list where **every edge u → v becomes v → u**.

i (Source Node)	**j (adj[i])**	**reversedAdj[j] After Insertion**
1	3	reversedAdj[3] = {1}
1	2	reversedAdj[2] = {1}
3	4	reversedAdj[4] = {3}
4	5	reversedAdj[5] = {4}

📥 Final reversedAdj Table

Vertex	**reversedAdj[vertex] (Incoming Edges)**
1	—
2	1
3	1
4	3
5	4

🖨 Output of printGraph(reversedAdj, V)

This prints **destination → source** (reversed):

<table>
<tr><td></td><td>2 -> 1
3 -> 1
4 -> 3
5 -> 4</td></tr>
<tr><td colspan="2">Output:-
Reversed Graph:
2 -> 1
3 -> 1
4 -> 3
5 -> 4</td></tr>
</table>

Rotten Oranges in C++

```
#include<bits/stdc++.h>

using namespace std;

class Solution {
  public:
    //Function to find minimum time required to rot all
oranges.
    int orangesRotting(vector < vector < int >> & grid) {
      // figure out the grid size
      int n = grid.size();
      int m = grid[0].size();

      // store {{row, column}, time}
      queue < pair < pair < int, int > , int >> q;
      int vis[n][m];
      int cntFresh = 0;
      for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
          // if cell contains rotten orange
          if (grid[i][j] == 2) {
            q.push({{i, j}, 0});
            // mark as visited (rotten) in visited array
            vis[i][j] = 2;
          }
          // if not rotten
          else {
            vis[i][j] = 0;
          }
          // count fresh oranges
          if (grid[i][j] == 1) cntFresh++;
        }
      }

      int tm = 0;
      // delta row and delta column
      int drow[] = {-1, 0, +1, 0};
      int dcol[] = {0, 1, 0, -1};
      int cnt = 0;

      // bfs traversal (until the queue becomes empty)
      while (!q.empty()) {
        int r = q.front().first.first;
        int c = q.front().first.second;
        int t = q.front().second;
        tm = max(tm, t);
        q.pop();
        // exactly 4 neighbours
        for (int i = 0; i < 4; i++) {
          // neighbouring row and column
          int nrow = r + drow[i];
          int ncol = c + dcol[i];
          // check for valid cell and
          // then for unvisited fresh orange
          if (nrow >= 0 && nrow < n && ncol >= 0 && ncol <
m &&
            vis[nrow][ncol] == 0 && grid[nrow][ncol] == 1) {
            // push in queue with timer increased
             q.push({{nrow, ncol}, t + 1});
            // mark as rotten
            vis[nrow][ncol] = 2;
```

Input Grid
```
grid = {
    {0, 1, 2},
    {0, 1, 2},
    {2, 1, 1}
};
```

✅ Initial Setup

- Fresh oranges = **4**
- Rotten oranges start at:
  - (0, 2)
  - (1, 2)
  - (2, 0)
- Queue initialized with these rotten oranges (time = 0)

📊 Dry Run Table

Time	Queue Front (Cell)	Rotting New Oranges → Queue Update	Total Rotten
0	(0, 2)	(0,1) → push with t=1	1
0	(1, 2)	(1,1) → push with t=1	2
0	(2, 0)	(2,1) → push with t=1	3
1	(0, 1)	— (no new fresh)	—
1	(1, 1)	— (no new fresh)	—
1	(2, 1)	(2,2) → push with t=2	4
2	(2, 2)	—	—

🧾 Final Check

- Rotten count = 4
- Fresh count = 4
  ✅ All fresh oranges became rotten
- Max time = 2 (last t value added to queue)

✅ Final Output

Answer = 2

```
            cnt++;
          }
        }
      }

      // if all oranges are not rotten
      if (cnt != cntFresh) return -1;

      return tm;

    }
};

int main() {

  vector<vector<int>>grid{{0,1,2},{0,1,2},{2,1,1}};
  Solution obj;
  int ans = obj.orangesRotting(grid);
  cout << ans << "\n";

  return 0;
}
```

Output:-

1

Terminal Nodes in C++

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

class TerminalNodes {
private:
    unordered_map<int, vector<int>> adjacencyList;

public:
    TerminalNodes() {}

    void addEdge(int source, int destination) {
        adjacencyList[source].push_back(destination);
        adjacencyList[destination]; // Ensure destination is also in the map
    }

    void printTerminalNodes() {
        vector<int> terminalNodes;
        for (auto it = adjacencyList.begin(); it != adjacencyList.end(); ++it) {
            if (it->second.empty()) {
                terminalNodes.push_back(it->first);
            }
        }
        cout << "Terminal Nodes:" << endl;
        for (int node : terminalNodes) {
            cout << node << endl;
        }
    }
};

int main() {
    TerminalNodes graph;

    // Adding edges to the graph
    graph.addEdge(1, 2);
    graph.addEdge(2, 3);
    graph.addEdge(3, 4);
    graph.addEdge(4, 5);
    graph.addEdge(6, 7);

    graph.printTerminalNodes();

    return 0;
}
```

Step-by-Step Dry Run

Step	Operation	Affected Node(s)	Adjacency List State	Notes
1	addEdge(1, 2)	1, 2	{1: [2], 2: []}	1 → 2, ensure 2 is in the map
2	addEdge(2, 3)	2, 3	{1: [2], 2: [3], 3: []}	2 → 3, ensure 3 is in the map
3	addEdge(3, 4)	3, 4	{1: [2], 2: [3], 3: [4], 4: []}	3 → 4, ensure 4 is in the map
4	addEdge(4, 5)	4, 5	{1: [2], 2: [3], 3: [4], 4: [5], 5: []}	4 → 5, ensure 5 is in the map
5	addEdge(6, 7)	6, 7	{1: [2], 2: [3], 3: [4], 4: [5], 5: [], 6: [7], 7: []}	6 → 7, ensure 7 is in the map
6	printTerminalNodes()	Scan all nodes	Check which nodes have empty adjacency lists	Nodes 5 and 7 have no outgoing edges
7	Print	Terminal Nodes		Output: 5, 7

✅ **Final Output**

Terminal Nodes:

5
7

Output:-

```
Terminal Nodes:
7
5
```

Topological sort DFS in C++

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

class Topo_dfs {
public:
   // Helper function to perform DFS and populate stack
   static void dfs(int node, vector<int>& vis, stack<int>& st, vector<vector<int>>& adj) {
       vis[node] = 1; // Mark node as visited

       // Traverse all adjacent nodes
       for (int it : adj[node]) {
           if (vis[it] == 0) { // If adjacent node is not visited, perform DFS on it
               dfs(it, vis, st, adj);
           }
       }

       st.push(node); // Push current node to stack after visiting all its dependencies
   }

   // Function to perform topological sorting using DFS
   static vector<int> topoSort(int V, vector<vector<int>>& adj) {
       vector<int> vis(V, 0); // Initialize visited array
       stack<int> st; // Stack to store nodes in topological order

       // Perform DFS for each unvisited node
       for (int i = 0; i < V; ++i) {
           if (vis[i] == 0) {
               dfs(i, vis, st, adj);
           }
       }

       vector<int> topo(V);
       int index = 0;

       // Pop elements from stack to get topological order
       while (!st.empty()) {
           topo[index++] = st.top();
           st.pop();
       }

       return topo;
   }
};

int main() {
   int V = 6;
   vector<vector<int>> adj(V);

   adj[2].push_back(3);
   adj[3].push_back(1);
   adj[4].push_back(0);
   adj[4].push_back(1);
   adj[5].push_back(0);
   adj[5].push_back(2);
```

Revised Dry Run with DFS Call Order

DFS Start	Calls	Stack Push Order
0	No edges → push(0)	0
1	No edges → push(1)	1, 0
2	DFS(3) → DFS(1) already visited	3, 2, 1, 0
3	Already visited	
4	DFS(0, already visited), DFS(1)	4, 3, 2, 1, 0
5	DFS(0, 2) already visited	5, 4, 3, 2, 1, 0

✅ **Stack (Top to Bottom)**

5
4
2
3
1
0

➡ **Final Output**

```cpp
while (!st.empty()) {
   topo[index++] = st.top();
   st.pop();
}
```

🟩 **Output:**

5 4 2 3 1 0

🧠 **Why This Is Valid:**

Topological sort can have **multiple valid orders** as long as:

- For every edge u → v, u appears **before** v.

And in this case:

- 5 is before 2, 0
- 2 is before 3
- 3 is before 1
- 4 is before 0, 1

✅ All conditions are satisfied.

```
    vector<int> ans = Topo_dfs::topoSort(V, adj);

    for (int node : ans) {
        cout << node << " ";
    }
    cout << endl;

    return 0;
}
```

Output:-

5 4 2 3 1 0