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;
```

Dry Run:

Let's dry run the algorithm with the input:

```cpp
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"}
};
```

Step 1: Initialize Disjoint Set:

- Rank array: `[0, 0, 0, 0, 0, 0, 0]`
- Parent array: `[0, 1, 2, 3, 4, 5, 6]`
- Size array: `[1, 1, 1, 1, 1, 1, 1]`

Step 2: Loop through the accounts:

Account 1: `{"John", "j1@com", "j2@com", "j3@com"}`

- For `j1@com`, map it to account `0`.
- For `j2@com`, map it to account `0`.
- For `j3@com`, map it to account `0`.

Account 2: `{"John", "j4@com"}`

- For `j4@com`, map it to account `1`.

Account 3: `{"Raj", "r1@com", "r2@com"}`

- For `r1@com`, map it to account `2`.
- For `r2@com`, map it to account `2`.

Account 4: `{"John", "j1@com", "j5@com"}`

- For `j1@com`, it already maps to account `0`. **Union** account `3` and `0`.
- For `j5@com`, map it to account `3`.

Account 5: `{"Raj", "r2@com", "r3@com"}`

- For `r2@com`, it already maps to account `2`. **Union** account `4` and `2`.
- For `r3@com`, map it to account `4`.

Account 6: `{"Mary", "m1@com"}`

- For `m1@com`, map it to account `5`.

```
        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;
```

Step 3: Union-find operations:

- Union operations are performed for common emails. For example:
  - `j1@com` in Account 1 and Account 4, so **union** Account 0 and Account 3.
  - `r2@com` in Account 3 and Account 4, so **union** Account 2 and Account 4.

After performing all unions, the parent array is updated as follows:

- Parent array: `[0, 1, 2, 0, 2, 5]`
- Rank array: `[1, 0, 1, 0, 0, 0]`
- Size array: `[4, 1, 3, 1, 2, 1]`

Step 4: Group emails by the root parent:

- For each email, find the root parent and group them.
  - Group 0: `{"j1@com", "j2@com", "j3@com", "j5@com"}`
  - Group 2: `{"r1@com", "r2@com", "r3@com"}`
  - Group 5: `{"m1@com"}`
  - Group 1: `{"j4@com"}`

Step 5: Sort and return:

- Sort each group of emails.
- Sort the result by the names (account names).

}	

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}
```

Dry Run:

Let's dry run the algorithm with the following graph represented by edges:

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

The graph can be visualized as:

```
yaml
Copy code
    0
    |
    1
   / \
  4---2
      |
      3
```

Step 1: Initialize Variables

- `vis`: A boolean vector initialized to `[0, 0, 0, 0, 0]` (all nodes unvisited).
- `tin`: A vector initialized to `[-1, -1, -1, -1, -1]`.
- `low`: A vector initialized to `[-1, -1, -1, -1, -1]`.
- `mark`: A vector initialized to `[0, 0, 0, 0, 0]` (articulation points).
- `timer`: Set to `1`, used to assign discovery times.

Step 2: DFS Traversal

- Start DFS from node `0`:
  - For node `0`:
    - Set `tin[0] = low[0] = 1`.
    - Visit neighbors: `1` (child).
  - For node `1`:
    - Set `tin[1] = low[1] = 2`.
    - Visit neighbors: `0` (parent) and `4` (child).
  - For node `4`:
    - Set `tin[4] = low[4] = 3`.
    - Visit neighbors: `1` (parent), `2` (child).

```
    };

    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;
}
```

- For node `2`:
    - Set `tin[2] = low[2] = 4`.
    - Visit neighbors: `4` (parent), `3` (child).
- For node `3`:
    - Set `tin[3] = low[3] = 5`.
    - Visit neighbors: `2` (parent).
    - DFS ends for node `3`, return to `2`.
- For node `2`, update `low[2]` as `min(low[2], low[3]) = 4`.
- As `low[3] >= tin[2]`, mark node `2` as an articulation point.
- For node `4`, update `low[4]` as `min(low[4], low[2]) = 3`.
- As `low[2] >= tin[4]`, mark node `4` as an articulation point.
- For node `1`, update `low[1]` as `min(low[1], low[4]) = 2`.
- As `low[4] >= tin[1]`, mark node `1` as an articulation point.

Step 3: Collect and Sort Results

- After DFS completes, `mark` contains `[0, 1, 1, 0, 1]`, indicating that nodes `1`, `2`, and `4` are articulation points.
- The final output will be `1 4` (sorted articulation points).

Output:-
1 4

BellmanFord in C++

```
#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 << " ";
```

Dry Run:

Let's dry run the given code with the input:

```
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;
```

Step 1: Initialize Variables

- `dist[]`: Distance array initialized to `{1e8, 1e8, 1e8, 1e8, 1e8, 1e8}`.
- Set `dist[0] = 0` (since `S = 0`).

Step 2: Relaxation (V-1) Times

- **First iteration (`i = 0`)**: Relax all edges.
  - Relax edge `(3, 2, 6)`: No change.
  - Relax edge `(5, 3, 1)`: No change.
  - Relax edge `(0, 1, 5)`: `dist[1] = min(1e8, dist[0] + 5) = 5.`
  - Relax edge `(1, 5, -3)`: `dist[5] = min(1e8, dist[1] - 3) = 2.`
  - Relax edge `(1, 2, -2)`: `dist[2] = min(1e8, dist[1] - 2) = 3.`
  - Relax edge `(3, 4, -2)`: `dist[4] = min(1e8, dist[3] - 2) = 3.`
  - Relax edge `(2, 4, 3)`: No change.
- **Second iteration (`i = 1`)**: Relax all edges again.
  - Relax edge `(3, 2, 6)`: No change.
  - Relax edge `(5, 3, 1)`: No change.
  - Relax edge `(0, 1, 5)`: No change.
  - Relax edge `(1, 5, -3)`: No change.
  - Relax edge `(1, 2, -2)`: No

```
	}
	cout << endl;

	return 0;
}
```

change.
- Relax edge `(3, 4, -2)`: No change.
- Relax edge `(2, 4, 3)`: No change.

(No updates during the second iteration.)

- **Third to Fifth iterations (`i = 2, 3, 4`)**: Relax all edges again.
  - No further changes, as all shortest paths are already updated.

Step 3: Negative Cycle Detection

- **Nth iteration (`i = 5`)**: Perform one more relaxation round.
  - All distances are unchanged, meaning no negative cycle exists.

Step 4: Return the Result

- Final `dist[]` array: `{0, 5, 3, 3, 1, 2}`.

Thus, the shortest distances from source `0` to all other nodes are:

```
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";

    return 0;
}
```

Graph:

```
    0
   / \
  2   3
       \
        1
```

Adj list:
adj[0] = {2, 3}
adj[1] = {3}
adj[2] = {0, 3}

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

Step-by-Step DFS Traversal:

1. **Node 0:** Start DFS at node 0 and color it 0:

   color = [0, -1, -1, -1]

   Adjacent nodes: {2, 3}.

2. **Node 2:** Visit node 2 from node 0, and color it 1 (opposite of 0):

   color = [0, -1, 1, -1]

   Adjacent nodes: {0, 3}.

   - **Node 0** is already colored 0, which does not conflict.
   - Move to node 3.

3. **Node 3:** Visit node 3 from node 2, and color it 0 (opposite of 1):

   color = [0, -1, 1, 0]

   Adjacent nodes: {0, 2, 1}.

   - **Node 0** is already colored 0, which does not conflict.
   - **Node 2** is already colored 1, which does not conflict.
   - Move to node 1.

4. **Node 1:** Visit node 1 from node 3, and color it 1 (opposite of 0):

   color = [0, 1, 1, 0]

	Adjacent nodes: {3}. ◦ **Node 3** is already colored 0, which does not conflict. **Conflict in the Graph:** Now, backtrack to **Node 3**: • Adjacent nodes: {0, 2, 1}. • Both **Node 2** and **Node 1** are adjacent to **Node 3**, but they are colored the **same** (1). This is a violation of the bipartite condition because two nodes (1 and 2) that are both connected to 3 have the same color. **Conclusion:** The graph is **not bipartite**, and the output is correctly: 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:
 1 -- 2 -- 3
Adj list:
adj[0] = {}   // Node 0 (no connections)
adj[1] = {2}   // Node 1 connected to Node 2
adj[2] = {1, 3}   // Node 2 connected to Nodes 1 and 3

adj[3] = {2}   // Node 3 connected to Node 2

Dry Run

Step 1: Initialization

- `vis[] = {0, 0, 0, 0}` (all nodes unvisited initially).

Step 2: Check Nodes

1. Start with `i = 0`:
   - `vis[0] = 0` (unvisited), but `adj[0]` is empty (no neighbors), so skip.
2. Move to `i = 1`:
   - `vis[1] = 0` (unvisited), start a DFS from node `1`.

DFS Traversal (from Node 1)

Node 1:

- Mark `vis[1] = 1`.
- Neighbors: `2`.
- `vis[2] = 0` (unvisited), call `dfs(2, 1)`.

Node 2:

- Mark `vis[2] = 1`.
- Neighbors: `1, 3`.
- `1` is the parent, so skip.
- `vis[3] = 0` (unvisited), call `dfs(3, 2)`.

Node 3:

- Mark `vis[3] = 1`.
- Neighbors: `2`.
- `2` is the parent, so skip.
- Return `false` (no cycle detected in

	this branch). **Backtrack**: • Return `false` from `dfs(2, 1)` to `dfs(1, -1)`. **Step 3: Continue Checking** 3. Move to `i = 2` and `i = 3`: o Both nodes are already visited, so skip. **Result** Since no cycle was detected in any connected component, the output is:
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:
1 → 3 → 4 → 5
↓
2
Adjacency list:
adj[1] = {3, 2}
adj[2] = {}
adj[3] = {4}
adj[4] = {5}
adj[5] = {}

Execution Steps

1. Initialize vis = {false, false, false, false, false, false} (1-based indexing).
2. Start iterating from i = 1 to i = 5.

DFS Starting from Node 1:

- Call dfs(1, vis, adj, ls):
    - Mark vis[1] = true, add 1 to ls.
    - Visit neighbors 3 and 2 of node 1.

Visit Node 3:

- Call dfs(3, vis, adj, ls):
    - Mark vis[3] = true, add 3 to ls.
    - Visit neighbor 4.

Visit Node 4:

- Call dfs(4, vis, adj, ls):
    - Mark vis[4] = true, add 4 to ls.
    - Visit neighbor 5.

Visit Node 5:

- Call dfs(5, vis, adj, ls):
    - Mark vis[5] = true, add 5 to ls.
    - No more neighbors to visit; return.

Backtrack:

- Backtrack to node 4, then to 3, and finally to 1.

Visit Node 2:

- Call dfs(2, vis, adj, ls):
    - Mark vis[2] = true, add 2 to ls.
    - No more neighbors to visit;

<table>
<tr><td></td><td>return.

Result for DFS from Node 1:
First connected component: [1, 3, 4, 5, 2].
Remaining Iterations:
For i = 2, 3, 4, 5, all nodes are already visited, so no new DFS is initiated.</td></tr>
<tr><td colspan="2">Output:-
1 3 4 5 2</td></tr>
</table>

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

Adj list:-
adj[0] = {{1, 1}, {2, 6}}
adj[1] = {{2, 3}, {0, 1}}

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

Initialization

- `distTo` array (stores the shortest distance to each vertex):

  ```
  distTo = [INT_MAX, INT_MAX, 0]
  // Source vertex S=2 distance initialized to 0
  ```

- Priority queue `pq` (min-heap):

  ```
  pq = {(0, 2)}  // {distance, node}
  ```

Iteration 1: Process Node 2

- Pop `(0, 2)` from `pq`.
- For adjacent nodes of `2`:
  - **Node 1** (`weight = 3`):

    ```
    plaintext
    Copy code
    distTo[1] = min(INT_MAX, 0 + 3) = 3
    pq = {(3, 1)}
    ```

  - **Node 0** (`weight = 6`):

    ```
    plaintext
    Copy code
    distTo[0] = min(INT_MAX, 0 + 6) = 6
    pq = {(3, 1), (6, 0)}
    ```

Iteration 2: Process Node 1

- Pop `(3, 1)` from `pq`.
- For adjacent nodes of `1`:
  - **Node 2** (`weight = 3`):

    ```
    plaintext
    Copy code
    distTo[2] = min(0, 3 + 3) = 0  // No update, already shorter
    pq = {(6, 0)}
    ```

  - **Node 0** (`weight = 1`):

    ```
    distTo[0] = min(6, 3 + 1)
    ```

<pre>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; }</pre>	<pre>= 4 pq = {(4, 0), (6, 0)}</pre> **Iteration 3: Process Node 0** • Pop `(4, 0)` from `pq`. • For adjacent nodes of `0`: ○ **Node 1** (`weight = 1`): <pre>distTo[1] = min(3, 4 + 1) = 3 // No update, already shorter</pre> ○ **Node 2** (`weight = 6`): <pre>distTo[2] = min(0, 4 + 6) = 0 // No update, already shorter</pre> **Final State** • `distTo` array: <pre>distTo = [4, 3, 0]</pre> **Output** The shortest distances from source vertex `S = 2` to all vertices are: <pre>4 3 0</pre>
Output:- **4 3 0**	

Disjoint Set in C++	
<pre>#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)) {</pre>	1. **makeSet(n)** o Initializes: ▪ parent = [0, 1, 2, 3, 4, 5, 6, 7] ▪ rankVec = [0, 0, 0, 0, 0, 0, 0, 0] o Each element is its own parent initially, and the rank is 0. 2. **unionByRank(1, 2)** o findUPar(1) returns 1 (root of 1). o findUPar(2) returns 2 (root of 2). o rankVec[1] (0) < rankVec[2] (0), so parent[2] = 1. o Updated: ▪ parent = [0, 1, 1, 3, 4, 5, 6, 7] ▪ rankVec = [0, 1, 0, 0, 0, 0, 0, 0] 3. **unionByRank(2, 3)** o findUPar(2) returns 1 (after path compression). o findUPar(3) returns 3. o rankVec[1] (1) > rankVec[3] (0), so parent[3] = 1. o Updated: ▪ parent = [0, 1, 1, 1, 4, 5, 6, 7] ▪ rankVec = [0, 1, 0, 0, 0, 0, 0, 0] 4. **unionByRank(4, 5)** o findUPar(4) returns 4. o findUPar(5) returns 5. o rankVec[4] (0) < rankVec[5] (0), so parent[5] = 4. o Updated: ▪ parent = [0, 1, 1, 1, 4, 4, 6, 7] ▪ rankVec = [0, 1, 0, 0, 1, 0, 0, 0] 5. **unionByRank(6, 7)** o findUPar(6) returns 6. o findUPar(7) returns 7. o rankVec[6] (0) < rankVec[7] (0), so parent[7] = 6. o Updated: ▪ parent = [0, 1, 1, 1, 4, 4, 6, 6] ▪ rankVec = [0, 1, 0, 0, 1, 0, 1, 0] 6. **unionByRank(5, 6)** o findUPar(5) returns 4 (path compression for 5). o findUPar(6) returns 6. o rankVec[4] (1) > rankVec[6] (1), so parent[6] = 4. o Updated: ▪ parent = [0, 1, 1, 1, 4, 4, 4, 6] ▪ rankVec = [0, 1, 0, 0, 2, 0, 0, 0]

<pre> cout << "Same\n"; } else { cout << "Not same\n"; } return 0; }</pre>	7. **Checking if 3 and 7 are in the same set** ○ findUPar(3) returns 1. ○ findUPar(7) returns 6 (path compression for 7 → 6 → 4). ○ They are not in the same set, so it prints "Not same". 8. **unionByRank(3, 7)** ○ findUPar(3) returns 1. ○ findUPar(7) returns 4 (path compression for 7 → 6 → 4). ○ rankVec[1] (1) < rankVec[4] (2), so parent[1] = 4. ○ Updated: ▪ parent = [0, 4, 1, 1, 4, 4, 4, 4] ▪ rankVec = [0, 1, 0, 0, 2, 0, 0, 0] 9. **Checking if 3 and 7 are in the same set again** ○ findUPar(3) returns 4 (path compression for 3 → 1 → 4). ○ findUPar(7) returns 4. ○ They are now in the same set, so it prints "Same". **Final Parent and Rank Arrays:** • parent = [0, 4, 1, 1, 4, 4, 4, 4] • rankVec = [0, 1, 0, 0, 2, 0, 0, 0]

Output:-
Not same
Same

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) {`	**Dry Run:** Let's dry-run the code with the given graph: **Adjacency List** for the graph: 0 -> 1 1 -> 2 2 -> 3 3 -> 4, 5 4 -> 6 5 -> 6 6 -> 7 7 -> (no outgoing edges) 8 -> 1, 9 9 -> 10 10 -> 8 11 -> 9 **DFS Exploration**: 1. **Starting DFS from node 0**: ○ vis[0] = 1, pathVis[0] = 1 ○ Go to node 1: vis[1] = 1, pathVis[1] = 1 ○ Go to node 2: vis[2] = 1, pathVis[2] = 1 ○ Go to node 3: vis[3] = 1, pathVis[3] = 1 ○ Go to node 4: vis[4] = 1, pathVis[4] = 1 ○ Go to node 6: vis[6] = 1, pathVis[6] = 1 ○ Go to node 7: vis[7] = 1, pathVis[7] = 1 ▪ Node 7 has no outgoing edges, so it is safe: check[7] = 1 ○ Node 6 is safe as it leads to safe node 7: check[6] = 1 ○ Node 4 is safe as it leads to safe node 6: check[4] = 1 ○ Node 3 is safe as it leads to safe nodes 4 and 5: check[3] = 1 ○ Node 2 is safe as it leads to safe node 3: check[2] = 1 ○ Node 1 is safe as it leads to safe node 2: check[1] = 1 ○ Node 0 is safe as it leads to safe node 1: check[0] = 1 2. **DFS from node 8**: ○ vis[8] = 1, pathVis[8] = 1 ○ Go to node 1, but node 1 is already visited and part of the current path (cycle detected). ○ Hence, node 8 is unsafe. 3. **DFS from node 9**: ○ vis[9] = 1, pathVis[9] = 1 ○ Go to node 10: vis[10] = 1,

<pre> cout << node << " "; } cout << endl return 0; }</pre>	pathVis[10] = 1 ◦ Go to node 8, and since 8 is already visited and part of the current DFS path, node 9 is unsafe. 4. **DFS from node 10**: ◦ Same as node 9, it leads to node 8, so it's unsafe. 5. **DFS from node 11**: ◦ vis[11] = 1, pathVis[11] = 1 ◦ Go to node 9, which is unsafe. ◦ Therefore, node 11 is unsafe. **Final Results:** • The safe nodes are **[0, 1, 2, 3, 4, 5, 6, 7]**. **Output:** 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;
}
```

Dry Run:

Input Matrix:

The input adjacency matrix is:

```
matrix = [
    [0, 2, -1, -1],
    [1, 0, 3, -1],
    [-1, -1, 0, -1],
    [3, 5, 4, 0]
]
```

Step 1: Initialize the matrix

Replace -1 with 1e9 and set matrix[i][i] = 0 for all i:

```
matrix = [
    [0, 2, 1e9, 1e9],
    [1, 0, 3, 1e9],
    [1e9, 1e9, 0, 1e9],
    [3, 5, 4, 0]
]
```

Step 2: Floyd-Warshall Algorithm

Iterate over each intermediate node k and update the matrix.

- For k = 0 (Intermediate node 0):
    - Check each pair (i, j) and update the matrix.
    - No changes to the matrix as no shorter paths through node 0 are found.
- For k = 1 (Intermediate node 1):
    - For each pair (i, j):
        - Update matrix[0][2] to matrix[0][1] + matrix[1][2] = 2 + 3 = 5.
        - Update matrix[2][3] to matrix[2][1] + matrix[1][3] = 1e9 + 1e9 = 1e9 (no update).
- For k = 2 (Intermediate node 2):
    - For each pair (i, j):
        - No changes as there are no shorter paths through node 2.
- For k = 3 (Intermediate node 3):
    - For each pair (i, j):
        - Update matrix[2][1] to matrix[2][3] + matrix[3][1] = 1e9 + 5 = 1e9 (no update).
        - Update matrix[3][1] to matrix[3][3] + matrix[3][1] = 0 + 5 = 5 (no

<table>
<tr><td></td><td>update).

Step 3: Final Matrix:

After the Floyd-Warshall algorithm finishes, the matrix is:

matrix = [
[0, 2, 5, 8],
[1, 0, 3, 6],
[6, 8, 0, 9],
[3, 5, 4, 0]
]

Step 4: Convert 1e9 back to -1:

If matrix[i][j] == 1e9, set matrix[i][j] = -1.

Final output matrix:

matrix = [
[0, 2, 5, 8],
[1, 0, 3, 6],
[6, 8, 0, 9],
[3, 5, 4, 0]
]

Output:

Copy code
0 2 5 8
1 0 3 6
6 8 0 9
3 5 4 0</td></tr>
<tr><td colspan="2">Output:-
0 2 5 -1
1 0 3 -1
-1 -1 0 -1
3 5 4 0</td></tr>
</table>

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 looks like:-

```
0 - 1
|   |
2 - 3 – 4
```

Adjacency list looks like:-

0 -> 1, 2
1 -> 0, 3, 4
2 -> 0, 3
3 -> 2, 1, 4
4 -> 1, 3
5 -> (no neighbors)
6 -> (no neighbors)

Dry Run of BFS (Start Vertex = 0):

Initialization:

- deque<int> q: Initially contains 0 (q = {0}).
- vector<bool> visited: All elements are false, except visited[0] = true.

Steps:

1. **Process Vertex 0:**
    - rem = q.front() → rem = 0.
    - Print 0.
    - Add neighbors of 0 (1 and 2) to q:
        - Mark visited[1] = true and visited[2] = true.
        - q = {1, 2}.
2. **Process Vertex 1:**
    - rem = q.front() → rem = 1.
    - Print 1.
    - Add unvisited neighbors of 1 (3 and 4) to q:
        - Mark visited[3] = true and visited[4] = true.
        - q = {2, 3, 4}.
3. **Process Vertex 2:**
    - rem = q.front() → rem = 2.
    - Print 2.
    - Add unvisited neighbors of 2 (none, as 3 is already visited).
        - q = {3, 4}.
4. **Process Vertex 3:**
    - rem = q.front() → rem = 3.
    - Print 3.
    - Add unvisited neighbors of 3 (none, as 4 is already visited).
        - q = {4}.
5. **Process Vertex 4:**
    - rem = q.front() → rem = 4.
    - Print 4.
    - Add unvisited neighbors of 4 (none).
        - q = {} (empty).

Output:-
0 1 2 3 4

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

```cpp
#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];
```

Dry Run:

Input Graph:

Edges:

- (0, 2)
- (0, 3)
- (2, 3)
- (3, 1)

Adjacency List:

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

Execution:

1. Initialize color array: [-1, -1, -1, -1]
2. Start BFS from node 0:
   - Assign color 0 to node 0: color = [0, -1, -1, -1].
   - Visit node 2, assign color 1: color = [0, -1, 1, -1].
   - Visit node 3, assign color 1: color = [0, -1, 1, 1].
   - At this point, node 3 and node 2 (adjacent nodes) have the same color (1). Therefore, the graph is **not bipartite**.

Output:

- Since the graph contains an odd-length cycle (e.g., 0 → 2 → 3 → 0), it is not bipartite.
- The function isBipartite() returns false, and the output is:

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++	
(code below)	(explanation below)

```
#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 looks like:-

```
1 → 2 → 3 → 4
    ↑   ↓
    └→ 5
```

Adjacency list looks like:-
adj[0] = {}
adj[1] = {2}
adj[2] = {3}
adj[3] = {4, 5}
adj[4] = {2}
adj[5] = {}

Step 1: Calculate Indegree

- Initialize indegree[] = {0, 0, 0, 0, 0, 0}.
- Traverse adjacency list to calculate indegree:
  - 1 → 2: indegree[2]++ → indegree[] = {0, 0, 1, 0, 0, 0}
  - 2 → 3: indegree[3]++ → indegree[] = {0, 0, 1, 1, 0, 0}
  - 3 → 4: indegree[4]++ → indegree[] = {0, 0, 1, 1, 1, 0}
  - 3 → 5: indegree[5]++ → indegree[] = {0, 0, 1, 1, 1, 1}
  - 4 → 2: indegree[2]++ → indegree[] = {0, 0, 2, 1, 1, 1}
- Final indegree[]: {0, 0, 2, 1, 1, 1}.

Step 2: Add Nodes with indegree == 0 to Queue

- Nodes with indegree == 0: 0, 1.
- Initialize queue = {0, 1}.

Step 3: Process Queue (Topological Sort)

1. **Process Node 0**:
   - Dequeue 0, cnt++ → cnt = 1.
   - Node 0 has no outgoing edges; no changes to indegree[].
   - queue = {1}.
2. **Process Node 1**:
   - Dequeue 1, cnt++ → cnt = 2.
   - Node 1 → Node 2: Decrease indegree[2]-- → indegree[] = {0, 0, 1, 1, 1, 1}.
   - Node 2 has indegree != 0, so it is not added to the queue.
   - queue = {}.

Step 4: Check for Remaining Nodes

- **Cycle Exists**:
  - Processed nodes (cnt = 2) < Total nodes (V = 6).
  - A cycle exists, as some nodes (like 2, 3, 4, 5) were never processed.

Output:-

True The graph contains a cycle

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

```cpp
#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 looks like:-

1 -- 2 -- 3

0 (disconnected)

Adjacency list looks like:-

adj[0] = {}   // Node 0 has no connections

adj[1] = {2}   // Node 1 is connected to Node 2

adj[2] = {1, 3}   // Node 2 is connected to Nodes 1 & 3

adj[3] = {2}   // Node 3 is connected to Node 2

Step 1: Initialization

- vis[] = {0, 0, 0, 0} (all nodes initially unvisited).

Step 2: Iteration over Nodes (in isCycle)

1. **Check Node 0:**
    - vis[0] = 0 → call detect(0, adj, vis):
        - Node 0 has no edges (adj[0] is empty).
        - No cycle can be detected here. Return false.
    - Continue to next node.
2. **Check Node 1:**
    - vis[1] = 0 → call detect(1, adj, vis):
        - vis[1] = 1 → mark Node 1 as visited.
        - Initialize queue: q = {{1, -1}} (Node 1 with parent -1).
        - **Process Queue:**
            - Dequeue q.front() → node = 1, parent = -1.
            - Adjacent to Node 1 → Node 2.
                - vis[2] = 0 → mark Node 2 as visited, push {2, 1} to q.
                    - Queue: q = {{2, 1}}.
            - Dequeue q.front() → node = 2, parent = 1.
                - Adjacent to Node 2 → Nodes 1 and 3.
                    - **Node 1:** Already visited, but parent == 1 → No cycle detected here.
                    - **Node 3:** vis[3] = 0 → mark Node 3 as visited, push {3, 2} to q.
                        - Queue: q = {{3, 2}}.
            - Dequeue q.front() → node = 3, parent = 2.
                - Adjacent to Node 3 → Node 2.

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

- **Node 2:** Already visited, but parent == 2 → No cycle detected here.
- Queue is empty, no cycle found. Return false.

3. **Check Nodes 2 and 3:**
   - Both are already visited (vis[2] = 1, vis[3] = 1).
   - Skip further checks.

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

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

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

Depth First Search in C++

```cpp
#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 looks like: -

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

Adjacency list looks like:-
adj[0] = {1, 4}
adj[1] = {0, 2, 3}
adj[2] = {1}
adj[3] = {1}
adj[4] = {0}

Step-by-Step Execution

1. **Start BFS from Node 0**:
   - Mark 0 as visited: vis[0] = 1.
   - Enqueue 0: q = {0}.
2. **Process Node 0**:
   - Dequeue 0: q = {}.
   - Add 0 to BFS result: bfs = {0}.
   - Neighbors of 0: {1, 4}.
     - 1 is unvisited, mark as visited and enqueue: vis[1] = 1, q = {1}.
     - 4 is unvisited, mark as visited and enqueue: vis[4] = 1, q = {1, 4}.
3. **Process Node 1**:
   - Dequeue 1: q = {4}.
   - Add 1 to BFS result: bfs = {0, 1}.
   - Neighbors of 1: {0, 2, 3}.
     - 0 is already visited, skip.
     - 2 is unvisited, mark as visited and enqueue: vis[2] = 1, q = {4, 2}.
     - 3 is unvisited, mark as visited and enqueue: vis[3] = 1, q = {4, 2, 3}.
4. **Process Node 4**:
   - Dequeue 4: q = {2, 3}.
   - Add 4 to BFS result: bfs = {0, 1, 4}.
   - Neighbors of 4: {0}.
     - 0 is already visited, skip.
5. **Process Node 2**:
   - Dequeue 2: q = {3}.
   - Add 2 to BFS result: bfs = {0, 1, 4, 2}.
   - Neighbors of 2: {1}.
     - 1 is already visited, skip.
6. **Process Node 3**:
   - Dequeue 3: q = {}.
   - Add 3 to BFS result: bfs = {0, 1,

	4, 2, 3}. ○ Neighbors of 3: {1}. ▪ 1 is already visited, skip. 7. **Queue is Empty**: ○ End BFS traversal.
Output:- **0 1 4 2 3**	

Cycle detection in undirected graph using Depth First Search 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 looks like: -
1 -- 2 – 3

Adjacency list looks like:-
adj[0] = {}
adj[1] = {2}
adj[2] = {1, 3}
adj[3] = {2}

Step-by-Step Execution:

1. **Initialization**:
   - vis = {0, 0, 0, 0} (all nodes unvisited).
2. **Node 0**:
   - vis[0] = 0 (no edges from node 0, skip).
3. **Node 1**:
   - vis[1] = 0, start DFS from node 1.
4. **DFS from Node 1**:
   - node = 1, parent = -1.
   - Mark 1 as visited: vis = {0, 1, 0, 0}.
   - Visit adjacent node 2 (unvisited):
     - Call dfs(2, 1).
5. **DFS from Node 2**:
   - node = 2, parent = 1.
   - Mark 2 as visited: vis = {0, 1, 1, 0}.
   - Visit adjacent nodes:
     - Node 1: Already visited, but it's the parent node (skip).
     - Node 3: Unvisited:
       - Call dfs(3, 2).
6. **DFS from Node 3**:
   - node = 3, parent = 2.
   - Mark 3 as visited: vis = {0, 1, 1, 1}.
   - Visit adjacent nodes:
     - Node 2: Already visited, but it's the parent node (skip).
7. **DFS Ends**:
   - Backtrack to node 2, then to node 1.
8. **Node 1 Ends**:
   - Continue checking other nodes in isCycle().
   - Node 0, 2, and 3 are already visited.
9. **Cycle Check**:
   - No cycles found during traversal.

Output:-
0

No cycle

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:
     4 ----> 0
     |       ^
     |       |
     |       +----+
     v            |
     1 <--- 3 <---2
     ^
     |
     5 ----> 2
     |
     +----> 0s

V = 6
Adjacency List:
0 -> {}
1 -> {}
2 -> {3}
3 -> {1}
4 -> {0, 1}
5 -> {0, 2}
```

Step-by-Step Execution:

1. **Calculate Indegree**:
   - Traverse the adjacency list and compute indegrees:

     ```
     Indegree of node 0 = 2 (edges from 4, 5)
     Indegree of node 1 = 3 (edges from 3, 4, 5)
     Indegree of node 2 = 1 (edge from 5)
     Indegree of node 3 = 1 (edge from 2)
     Indegree of node 4 = 0 (no incoming edges)
     Indegree of node 5 = 0 (no incoming edges)
     ```

   - Indegree array: `[2, 3, 1, 1, 0, 0]`
2. **Initialize Queue**:
   - Nodes with `indegree = 0`: `[4, 5]`
   - Initial queue: `q = [4, 5]`
3. **Process Nodes in Topological Order**:
   - **Step 1**: Process node `4`:
     - Add `4` to `topo`: `topo = [4]`
     - Reduce indegree of `0` and `1`: `indegree[0] = 1`, `indegree[1] = 2`
     - Updated queue: `q = [5]`
   - **Step 2**: Process node `5`:

	- Add `5` to `topo`: `topo = [4, 5]` - Reduce indegree of `0` and `2`: `indegree[0] = 0`, `indegree[2] = 0` - Updated queue: `q = [0, 2]` ○ **Step 3**: Process node `0`: - Add `0` to `topo`: `topo = [4, 5, 0]` - No neighbors to update. - Updated queue: `q = [2]` ○ **Step 4**: Process node `2`: - Add `2` to `topo`: `topo = [4, 5, 0, 2]` - Reduce indegree of `3`: `indegree[3] = 0` - Updated queue: `q = [3]` ○ **Step 5**: Process node `3`: - Add `3` to `topo`: `topo = [4, 5, 0, 2, 3]` - Reduce indegree of `1`: `indegree[1] = 0` - Updated queue: `q = [1]` ○ **Step 6**: Process node `1`: - Add `1` to `topo`: `topo = [4, 5, 0, 2, 3, 1]` - No neighbors to update. - Updated queue: `q = []` 4. **Final Topological Order**: `topo = [4, 5, 0, 2, 3, 1]` **Output:** `4 5 0 2 3 1`

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[])
    {
        // 1 - 2 wt = 5
```

The graph represented by `edges` is:

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

Step 1: Create the Edge List

The adjacency list `adj[]` is converted into an edge list `edges[]`, which is a vector of pairs representing the edges:

```
edges = [(2, (0, 1)), (1, (0, 2)),
(1, (1, 2)), (2, (2, 3)), (1, (3,
4)), (2, (4, 2))]
```

Step 2: Sort the Edges by Weight

The edges are sorted in ascending order by their weights:

```
edges = [(1, (0, 2)), (1, (1, 2)),
(1, (3, 4)), (2, (0, 1)), (2, (2,
3)), (2, (4, 2))]
```

Step 3: Apply Kruskal's Algorithm with Disjoint Set

- Initialize the Disjoint Set for 5 vertices: `parent = [0, 1, 2, 3, 4]`, `size = [1, 1, 1, 1, 1]`.
- Process each edge:
  1. **Edge (0, 2, 1)**:
     - `find(0) != find(2)`, so add the edge to MST.
     - `parent[2] = 0`, `size[0] = 2`.
     - Add `1` to `mstWt`. Now `mstWt = 1`.
  2. **Edge (1, 2, 1)**:
     - `find(1) != find(2)`, so add the edge to MST.
     - `parent[2] = 1`, `size[1] = 2`.
     - Add `1` to `mstWt`. Now `mstWt = 2`.
  3. **Edge (3, 4, 1)**:
     - `find(3) != find(4)`, so add the edge to MST.

```cpp
        /// 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;
}
```

- `parent[4] = 3`, `size[3] = 2`.
- Add `1` to `mstWt`. Now `mstWt = 3`.

4. **Edge (0, 1, 2)**:
   - `find(0) == find(1)`, so ignore this edge (it forms a cycle).
5. **Edge (2, 3, 2)**:
   - `find(2) != find(3)`, so add the edge to MST.
   - `parent[3] = 2`, `size[2] = 4`.
   - Add `2` to `mstWt`. Now `mstWt = 5`.
6. **Edge (4, 2, 2)**:
   - `find(4) == find(2)`, so ignore this edge (it forms a cycle).

Step 4: Return the MST Weight

The total weight of the Minimum Spanning Tree is `5`.

Output:-

The sum of all the edge weights: 5

No of provinces in C++

```
#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;
}
```

Dry Run:

Input:

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

Adjacency Matrix to List Conversion:

- adj[0] has a 1 at indices 0 and 2, so node 0 is connected to node 2.
- adj[1] has a 1 at index 1, so node 1 is connected to itself.
- adj[2] has a 1 at indices 0 and 2, so node 2 is connected to node 0.

Adjacency List:

```
adjLs = {
    {2},  // Node 0 is connected to 2
    {0},  // Node 1 is connected to 0
    {0}   // Node 2 is connected to 0
}
```

DFS Execution:

1. Start DFS from node 0. Mark node 0 as visited and visit node 2.
2. In DFS traversal, we will also mark node 2 as visited.
3. Start DFS from node 1. Since it is unvisited, we increment the province counter. Mark node 1 as visited.
4. Finally, return the total count of provinces (2).

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:

We have 5 vertices (V = 5) and the edges:

edges = [{0, 1, 2}, {0, 2, 1}, {1, 2, 1}, {2, 3, 2}, {3, 4, 1}, {4, 2, 2}]

Graph Representation (Adjacency List):

adj[0] = {{1, 2}, {2, 1}}
adj[1] = {{0, 2}, {2, 1}}
adj[2] = {{0, 1}, {1, 1}, {3, 2}, {4, 2}}
adj[3] = {{2, 2}, {4, 1}}
adj[4] = {{3, 1}, {2, 2}}

Prim's Algorithm Process

1. **Initialization:**
   - Use a **priority queue** pq to process edges in increasing weight order. The queue stores {weight, node}.
   - Use a vis array to track visited nodes: vis = [0, 0, 0, 0, 0].
   - Start with node 0: push {0, 0} to pq.

Iteration 1:

- **Priority Queue:** pq = {{0, 0}}
- **Pop the top element:** {0, 0} → node = 0, weight = 0.
- **Check if node is visited:** It's not, so mark node 0 as visited: vis = [1, 0, 0, 0, 0].
- **Add weight to sum:** sum = 0 + 0 = 0.
- **Push adjacent edges to pq:**
  - From adj[0] = {{1, 2}, {2, 1}}:
    - Push {2, 1} (edge to node 1 with weight 2).
    - Push {1, 2} (edge to node 2 with weight 1).
- **Updated Priority Queue:** pq = {{1, 2}, {2, 1}}.

Iteration 2:

- **Priority Queue:** pq = {{1, 2}, {2, 1}}
- **Pop the top element:** {1, 2} → node = 2, weight = 1.
- **Check if node is visited:** It's not, so mark node 2 as visited: vis = [1, 0, 1, 0, 0].

<table>
<tr>
<td>

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

        return 0;
}
```

</td>
<td>

- **Add weight to sum:** sum = 0 + 1 = 1.
- **Push adjacent edges to pq:**
  - From adj[2] = {{0, 1}, {1, 1}, {3, 2}, {4, 2}}:
    - Skip {0, 1} (node 0 is already visited).
    - Push {1, 1} (edge to node 1 with weight 1).
    - Push {2, 3} (edge to node 3 with weight 2).
    - Push {2, 4} (edge to node 4 with weight 2).
- **Updated Priority Queue:** pq = {{1, 1}, {2, 1}, {2, 3}, {2, 4}}.

Iteration 3:

- **Priority Queue:** pq = {{1, 1}, {2, 1}, {2, 3}, {2, 4}}
- **Pop the top element:** {1, 1} → node = 1, weight = 1.
- **Check if node is visited:** It's not, so mark node 1 as visited: vis = [1, 1, 1, 0, 0].
- **Add weight to sum:** sum = 1 + 1 = 2.
- **Push adjacent edges to pq:**
  - From adj[1] = {{0, 2}, {2, 1}}:
    - Skip {0, 2} and {2, 1} (nodes 0 and 2 are already visited).
- **Updated Priority Queue:** pq = {{2, 1}, {2, 3}, {2, 4}}.

Iteration 4:

- **Priority Queue:** pq = {{2, 3}, {2, 4}}
- **Pop the top element:** {2, 3} → node = 3, weight = 2.
- **Check if node is visited:** It's not, so mark node 3 as visited: vis = [1, 1, 1, 1, 0].
- **Add weight to sum:** sum = 2 + 2 = 4.
- **Push adjacent edges to pq:**
  - From adj[3] = {{2, 2}, {4, 1}}:
    - Skip {2, 2} (node 2 is already visited).
    - Push {1, 4} (edge to node 4 with weight 1).
- **Updated Priority Queue:** pq = {{1, 4}, {2, 4}}.

Iteration 5:

- **Priority Queue:** pq = {{1, 4}, {2, 4}}

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

	Pop the top element: {1, 4} → node = 4, weight = 1. **Check if node is visited:** It's not, so mark node 4 as visited: vis = [1, 1, 1, 1, 1]. **Add weight to sum:** sum = 4 + 1 = 5. **Push adjacent edges to pq:** - From adj[4] = {{3, 1}, {2, 2}}: - Skip {3, 1} and {2, 2} (nodes 3 and 2 are already visited). **Updated Priority Queue:** pq = {{2, 4}}. **Iteration 6:** **Priority Queue:** pq = {{2, 4}} **Pop the top element:** {2, 4} → node = 4, weight = 2. **Check if node is visited:** It is already visited, so skip this iteration. **Final Output:** **Sum of Weights of MST:** 5. **Visited Array:** vis = [1, 1, 1, 1, 1] (all nodes visited).
Output:- The sum of all the edge weights: 5	

Reverse directed graph in C++	
see below	see below

```
#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;
}
```

Input:

- **Number of nodes** (`V`) = 5
- **Edges** of the directed graph (adjacency list):
  - 1 → 3
  - 1 → 2
  - 3 → 4
  - 4 → 5

Step 1: Initialize Adjacency List

The adjacency list for the original graph (`adj`) is built as:

```
adj[1] = [3, 2]    // Node 1 has edges
to 3 and 2
adj[2] = []        // Node 2 has no
outgoing edges
adj[3] = [4]       // Node 3 has an
edge to 4
adj[4] = [5]       // Node 4 has an
edge to 5
adj[5] = []        // Node 5 has no
outgoing edges
```

Step 2: Call `reverseDirectedGraph()` Function

Now, the function `reverseDirectedGraph()` will reverse the edges of the graph. We will iterate over the adjacency list and for each edge from `u → v`, we will add an edge `v → u` in the reversed graph.

Iterating through the adjacency list:

- **i = 1** (For node 1):
  - For edge `1 → 3`, reverse it to `3 → 1`
  - For edge `1 → 2`, reverse it to `2 → 1`
  - So, `reversedAdj[3]` becomes `[1]` and `reversedAdj[2]` becomes `[1]`.
- **i = 2** (For node 2):
  - Node 2 has no outgoing edges, so no change.
- **i = 3** (For node 3):
  - For edge `3 → 4`, reverse it to `4 → 3`
  - So, `reversedAdj[4]` becomes `[3]`.
- **i = 4** (For node 4):
  - For edge `4 → 5`, reverse it to `5`

→ 4

- So, `reversedAdj[5]` becomes `[4]`.

- **i = 5** (For node 5):
    - Node 5 has no outgoing edges, so no change.

Reversed Graph:

After the reversal of the edges, the reversed adjacency list will be:

```
reversedAdj[1] = []        // No edges
coming into 1
reversedAdj[2] = [1]       // Node 2
has an edge coming from 1
reversedAdj[3] = [1]       // Node 3
has an edge coming from 1
reversedAdj[4] = [3]       // Node 4
has an edge coming from 3
reversedAdj[5] = [4]       // Node 5
has an edge coming from 4
```

Step 3: Print Reversed Graph Using `printGraph()` Function

Now, the `printGraph()` function will print the reversed adjacency list:

1. **For node 1**:
    - `reversedAdj[1] = []`, so no output for node 1.
2. **For node 2**:
    - `reversedAdj[2] = [1]`, so it will print `2 -> 1`.
3. **For node 3**:
    - `reversedAdj[3] = [1]`, so it will print `3 -> 1`.
4. **For node 4**:
    - `reversedAdj[4] = [3]`, so it will print `4 -> 3`.
5. **For node 5**:
    - `reversedAdj[5] = [4]`, so it will print `5 -> 4`.

Final Output:

The output of the program will be:

```
Reversed Graph:
2 -> 1
3 -> 1
4 -> 3
5 -> 4
```

	Summary of the Dry Run: 1. **Original graph** has edges: o 1 → 3, 1 → 2, 3 → 4, 4 → 5 2. **Reversed graph** has edges: o 2 → 1, 3 → 1, 4 → 3, 5 → 4
Output:- Reversed Graph: 2 -> 1 3 -> 1 4 -> 3 5 -> 4	

Rotten Oranges in C++	
<pre>#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;</pre>	**Step 1: BFS Traversal** The `queue` will be used to perform BFS, where we process the rotten oranges and spread the rot to adjacent fresh oranges. The variable `tm` will track the maximum time it takes to rot all oranges. **First BFS Iteration (Queue: `q = { {{0, 2}, 0}, {{1, 2}, 0}, {{2, 0}, 0} }`):** • **Processing rotten orange at (0, 2) at time 0**: ○ **Neighbors**: ▪ `(0, 1)` is a fresh orange (`grid[0][1] == 1`), so we rot it and add it to the queue with time 1: `q.push({{0, 1}, 1})`. ○ Updated state: <pre>vis = { {0, 2, 2}, {0, 1, 2}, {2, 1, 1} } q = { {{1, 2}, 0}, {{2, 0}, 0}, {{0, 1}, 1} }</pre> • **Processing rotten orange at (1, 2) at time 0**: ○ **Neighbors**: ▪ `(1, 1)` is a fresh orange (`grid[1][1] == 1`), so we rot it and add it to the queue with time 1: `q.push({{1, 1}, 1})`. ○ Updated state: <pre>vis = { {0, 2, 2}, {0, 2, 2}, {2, 1, 1} } q = { {{2, 0}, 0}, {{0, 1}, 1}, {{1, 1}, 1} }</pre> • **Processing rotten orange at (2, 0) at time 0**: ○ **Neighbors**: ▪ `(2, 1)` is a fresh orange (`grid[2][1] == 1`), so we rot it and

```
            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;
}
```

add it to the queue with time 1: `q.push({{2, 1}, 1})`.

- Updated state:

```
vis = {
    {0, 2, 2},
    {0, 2, 2},
    {2, 2, 1}
}
q = { {{0, 1}, 1}, {{1, 1}, 1}, {{2, 1}, 1} }
```

Second BFS Iteration (Queue: `q = { {{0, 1}, 1}, {{1, 1}, 1}, {{2, 1}, 1} }`):

- **Processing rotten orange at (0, 1) at time 1**:
  - **Neighbors**:
    - `(0, 0)` is empty (`grid[0][0] == 0`), so nothing happens.
  - Queue remains unchanged:

    ```
    q = { {{1, 1}, 1}, {{2, 1}, 1} }
    ```

- **Processing rotten orange at (1, 1) at time 1**:
  - **Neighbors**:
    - `(1, 0)` is empty (`grid[1][0] == 0`), so nothing happens.
  - Queue remains unchanged:

    ```
    q = { {{2, 1}, 1} }
    ```

- **Processing rotten orange at (2, 1) at time 1**:
  - **Neighbors**:
    - `(2, 2)` is a fresh orange (`grid[2][2] == 1`), so we rot it and add it to the queue with time 2: `q.push({{2, 2}, 2})`.
  - Updated state:

    ```
    vis = {
        {0, 2, 2},
        {0, 2, 2},
        {2, 2, 2}
    }
    q = { {{2, 2}, 2} }
    ```

	Final State: After the BFS traversal completes, the queue is empty and the `vis` array is: `vis = {` `    {0, 2, 2},` `    {0, 2, 2},` `    {2, 2, 2}` `}` **Step 2: Checking if All Oranges Are Rotten** • **Count of Rotten Oranges (`cnt`)**: The total number of rotten oranges in the grid is `cnt = 4` (after BFS propagation). • **Count of Fresh Oranges (`cntFresh`)**: The initial count of fresh oranges is `cntFresh = 4`. • **Result**: Since `cnt == cntFresh`, all fresh oranges have been rotted. **Step 3: Return the Time** • The maximum time it took to rot all the fresh oranges is `tm = 1`. Thus, the **minimum time required to rot all oranges is `1`**.
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;
}
```

Example Walkthrough

Let's consider the following graph representation:

```
1 -> 2 -> 3 -> 4 -> 5
6 -> 7
```

- **Graph Representation:**
  - Node 1 has an edge to node 2.
  - Node 2 has an edge to node 3.
  - Node 3 has an edge to node 4.
  - Node 4 has an edge to node 5.
  - Node 6 has an edge to node 7.
  - Node 7 has no outgoing edges.
- **Terminal Nodes:**
  - Nodes 5 and 7 are terminal nodes because they have no outgoing edges.

Code Execution:

1. The `addEdge` method is called multiple times to build the graph.
2. Then, the `printTerminalNodes()` method is called to iterate through the graph and check for terminal nodes.
3. The nodes 5 and 7 will be identified as terminal nodes and printed.

Output:-

```
Terminal Nodes:
7
5
```

Topological sort DFS in C++	
<pre>#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);</pre>	**Input** Vertices (V) = 6 Edges: • 2 → 3 • 3 → 1 • 4 → 0 • 4 → 1 • 5 → 0 • 5 → 2 Adjacency list: adj = [[], // Node 0 [], // Node 1 [3], // Node 2 [1], // Node 3 [0, 1], // Node 4 [0, 2] // Node 5] **Dry Run** **Step 1: Initialize Variables** • Visited array: vis = [0, 0, 0, 0, 0, 0] • Stack (st) is empty. **Step 2: Start DFS from Unvisited Nodes** **Iteration 1 (Node 0)**: • vis[0] = 1. Node 0 has no neighbors. • Push 0 to st: st = [0]. **Iteration 2 (Node 1)**: • vis[1] = 1. Node 1 has no neighbors. • Push 1 to st: st = [0, 1]. **Iteration 3 (Node 2)**: • vis[2] = 1. • Neighbor: Node 3. ○ Perform DFS on Node 3: ▪ vis[3] = 1. ▪ Neighbor: Node 1 (already visited). ▪ Push 3 to st: st = [0, 1,

<pre> vector<int> ans = Topo_dfs::topoSort(V, adj); for (int node : ans) { cout << node << " "; } cout << endl; return 0; }</pre>	3]. • Push 2 to st: st = [0, 1, 3, 2]. **Iteration 4 (Node 3)**: • Already visited. Skip. **Iteration 5 (Node 4)**: • vis[4] = 1. • Neighbors: Node 0 and Node 1 (both already visited). • Push 4 to st: st = [0, 1, 3, 2, 4]. **Iteration 6 (Node 5)**: • vis[5] = 1. • Neighbors: Node 0 and Node 2 (both already visited). • Push 5 to st: st = [0, 1, 3, 2, 4, 5]. **Step 3: Extract Topological Order** • Reverse the stack: topo = [5, 4, 2, 3, 1, 0].
Output:- 5 4 2 3 1 0	