Graph

Name:- Kuldeep Day 7

UID:- 22BCS10071

Section:- KPIT_901/A

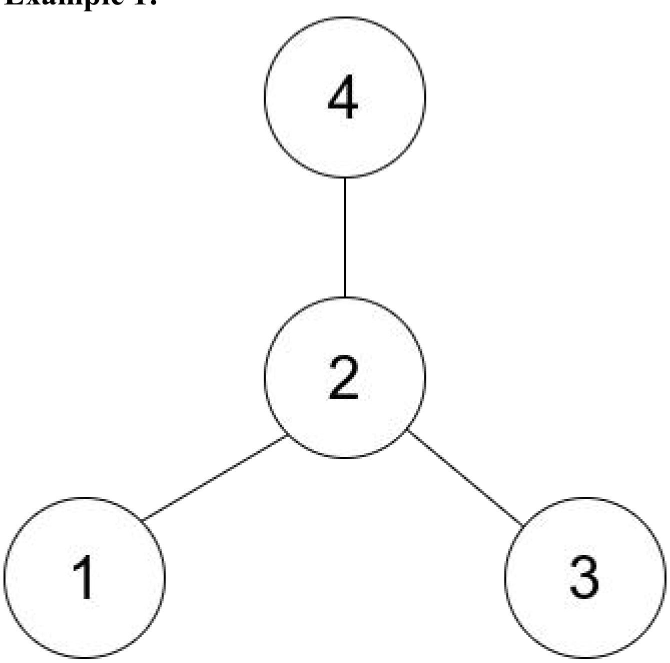
Very Easy:

Find Center of Star Graph

There is an undirected star graph consisting of n nodes labeled from 1 to n. A star graph is a graph where there is one center node and exactly n - 1 edges that connect the center node with every other node.

You are given a 2D integer array edges where each edges[i] = [ui, vi] indicates that there is an edge between the nodes ui and vi. Return the center of the given star graph.

Example 1:



Input: edges = [[1,2],[2,3],[4,2]]
Output: 2
Explanation: As shown in the figure above, node 2 is connected to every other node, so 2 is the center.

Example 2:
Input: edges = [[1,2],[5,1],[1,3],[1,4]]
Output: 1

Constraints:

- 3 <= n <= 1e5
- edges.length == n - 1
- edges[i].length == 2
- 1 <= ui, vi <= n
- ui != vi
- The given edges represent a valid star graph.

Code:-

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

int findCenter(vector<vector<int>>& edges) {
    // Check the first two edges
    if (edges[0][0] == edges[1][0] || edges[0][0] == edges[1][1]) {
        return edges[0][0];
    }
    return edges[0][1];
}

int main() {
    vector<vector<int>> edges1 = {{1, 2}, {2, 3}, {4, 2}};
    vector<vector<int>> edges2 = {{1, 2}, {5, 1}, {1, 3}, {1, 4}};

    cout << findCenter(edges1) << endl; // Output: 2
    cout << findCenter(edges2) << endl; // Output: 1

    return 0;
}
```

Output:-

```
[Running] cd "c:\Users\Rohit Thakur\Desktop\cpp code\winning camp questi
Thakur\Desktop\cpp code\winning camp question\"tempCodeRunnerFile
2
1

[Done] exited with code=0 in 0.74 seconds
```

Easy

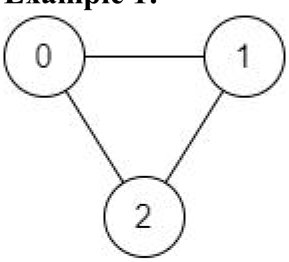
[Find if Path Exists in Graph]

There is a bi-directional graph with n vertices, where each vertex is labeled from 0 to n - 1 (inclusive). The edges in the graph are represented as a 2D integer array edges, where each edges[i] = [ui, vi] denotes a bi-directional edge between vertex ui and vertex vi. Every vertex pair is connected by at most one edge, and no vertex has an edge to itself.

You want to determine if there is a valid path that exists from vertex source to vertex destination.

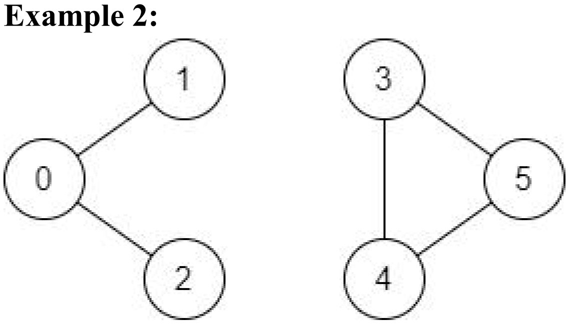
Given edges and the integers n, source, and destination, return true if there is a valid path from source to destination, or false otherwise.

Example 1:



Input: n = 3, edges = [[0,1],[1,2],[2,0]], source = 0, destination = 2
Output: true
Explanation: There are two paths from vertex 0 to vertex 2:
- 0 → 1 → 2
- 0 → 2

Example 2:



Input: n = 6, edges = [[0,1],[0,2],[3,5],[5,4],[4,3]], source = 0, destination = 5

Output: false
Explanation: There is no path from vertex 0 to vertex 5.
Constraints:

- 1 <= n <= 2 * 1e5
- 0 <= edges.length <= 2 * 1e5
- edges[i].length == 2
- 0 <= ui, vi <= n - 1
- ui != vi
- 0 <= source, destination <= n - 1
- There are no duplicate edges.
- There are no self edges.

Code:-

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

bool validPath(int n, vector<vector<int>>& edges, int source, int destination) {
    // Create an adjacency list
    vector<vector<int>> graph(n);
    for (const auto& edge : edges) {
        graph[edge[0]].push_back(edge[1]);
        graph[edge[1]].push_back(edge[0]);
    }

    // BFS setup
    queue<int> q;
    vector<bool> visited(n, false);

    q.push(source);
    visited[source] = true;

    // BFS traversal
    while (!q.empty()) {
        int node = q.front();
        q.pop();

        // If destination is found, return true
        if (node == destination) {
            return true;
        }
```

```
        // Visit neighbors
        for (int neighbor : graph[node]) {
            if (!visited[neighbor]) {
                visited[neighbor] = true;
                q.push(neighbor);
            }
        }
    }

    // If traversal completes without finding the destination
    return false;
}

int main() {
    // Example 1
    int n1 = 3;
    vector<vector<int>> edges1 = {{0, 1}, {1, 2}, {2, 0}};
    int source1 = 0, destination1 = 2;
    cout << (validPath(n1, edges1, source1, destination1) ? "true" : "false") << endl;

    // Example 2
    int n2 = 6;
    vector<vector<int>> edges2 = {{0, 1}, {0, 2}, {3, 5}, {5, 4}, {4, 3}};
    int source2 = 0, destination2 = 5;
    cout << (validPath(n2, edges2, source2, destination2) ? "true" : "false") << endl;

    return 0;
}
```

Output:-

```
[Running] cd "c:\Users\Rohit Thakur\Desktop\cpp code\winning camp question\" && g++
code\winning camp question\"easyd7
true
false

[Done] exited with code=0 in 0.896 seconds
```

Medium

01 Matrix

Given an m x n binary matrix mat, return the distance of the nearest 0 for each cell.

The distance between two adjacent cells is 1.

Example 1:

0	0	0
0	1	0
0	0	0

Input: mat = [[0,0,0],[0,1,0],[0,0,0]]
Output: [[0,0,0],[0,1,0],[0,0,0]]

Example 2:

0	0	0
0	1	0
1	1	1

Input: mat = [[0,0,0],[0,1,0],[1,1,1]]
Output: [[0,0,0],[0,1,0],[1,2,1]]

Constraints:

- m == mat.length
- n == mat[i].length
- 1 <= m, n <= 104
- 1 <= m * n <= 104
- mat[i][j] is either 0 or 1.

- There is at least one 0 in mat.

Code:-

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

vector<vector<int>> updateMatrix(vector<vector<int>>& mat) {
    int m = mat.size();
    int n = mat[0].size();
    vector<vector<int>> dist(m, vector<int>(n, INT_MAX));
    queue<pair<int, int>> q;

    // Initialize the queue with all 0 cells
    for (int i = 0; i < m; ++i) {
        for (int j = 0; j < n; ++j) {
            if (mat[i][j] == 0) {
                dist[i][j] = 0;
                q.push({i, j});
            }
        }
    }

    // Directions for moving in the matrix
    vector<pair<int, int>> directions = {{0, 1}, {1, 0}, {0, -1}, {-1, 0}};

    // BFS
    while (!q.empty()) {
        auto [x, y] = q.front();
        q.pop();

        for (auto [dx, dy] : directions) {
            int nx = x + dx;
            int ny = y + dy;

            // Check bounds and whether the cell is visited
            if (nx >= 0 && ny >= 0 && nx < m && ny < n && dist[nx][ny] == INT_MAX) {
                dist[nx][ny] = dist[x][y] + 1;
                q.push({nx, ny});
            }
        }
    }

    return dist;
}

int main() {
```

```
    vector<vector<int>> mat1 = {{0, 0, 0}, {0, 1, 0}, {0, 0, 0}};
    vector<vector<int>> mat2 = {{0, 0, 0}, {0, 1, 0}, {1, 1, 1}};

    vector<vector<int>> result1 = updateMatrix(mat1);
    vector<vector<int>> result2 = updateMatrix(mat2);

    // Print results
    for (const auto& row : result1) {
        for (int cell : row) cout << cell << " ";
        cout << endl;
    }
    cout << endl;

    for (const auto& row : result2) {
        for (int cell : row) cout << cell << " ";
        cout << endl;
    }

    return 0;
}
```

Output:-

```
0 0 0
0 1 0
0 0 0

0 0 0
0 1 0
1 2 1

[Done] exited with code=0 in 0.807 seconds
```

Hard

Accounts Merge

Given a list of accounts where each element accounts[i] is a list of strings, where the first element accounts[i][0] is a name, and the rest of the elements are emails representing emails of the account.

Now, we would like to merge these accounts. Two accounts definitely belong to the same person if there is some common email to both accounts. Note that even if two accounts have the same name, they may belong to different people as people could have the same name. A person can have any number of accounts initially, but all of their accounts definitely have the same name.

After merging the accounts, return the accounts in the following format: the first element of each account is the name, and the rest of the elements are emails in sorted order. The accounts themselves can be returned in any order.

Example 1:

Input: accounts =
[["John","johnsmith@mail.com","john_newyork@mail.com"],["John","johnsmith@mail.com","john00@mail.com"],["Mary","mary@mail.com"],["John","johnnybravo@mail.com"]]
Output:
[["John","john00@mail.com","john_newyork@mail.com","johnsmith@mail.com"],["Mary","mary@mail.com"],["John","johnnybravo@mail.com"]]
Explanation:
The first and second John's are the same person as they have the common email "johnsmith@mail.com".
The third John and Mary are different people as none of their email addresses are used by other accounts.
We could return these lists in any order, for example the answer [['Mary', 'mary@mail.com'], ['John', 'johnnybravo@mail.com'],
['John', 'john00@mail.com', 'john_newyork@mail.com', 'johnsmith@mail.com']] would still be accepted.

Example 2:

Input: accounts =
[["Gabe","Gabe0@m.co","Gabe3@m.co","Gabe1@m.co"],["Kevin","Kevin3@m.co","Kevin5@m.co","Kevin0@m.co"],["Ethan","Ethan5@m.co","Ethan4@m.co","Ethan0@m.co"],["Hanzo","Hanzo3@m.co","Hanzo1@m.co","Hanzo0@m.co"],["Fern","Fern5@m.co","Fern1@m.co","Fern0@m.co"]]
Output:
[["Ethan","Ethan0@m.co","Ethan4@m.co","Ethan5@m.co"],["Gabe","Gabe0@m.co","Gabe1@m.co","Gabe3@m.co"],["Hanzo","Hanzo0@m.co","Hanzo1@m.co","Hanzo3@m.co"],["Kevin","Kevin0@m.co","Kevin3@m.co","Kevin5@m.co"],["Fern","Fern0@m.co","Fern1@m.co","Fern5@m.co"]]

Constraints:

- 1 <= accounts.length <= 1000
- 2 <= accounts[i].length <= 10
- 1 <= accounts[i][j].length <= 30
- accounts[i][0] consists of English letters.
- accounts[i][j] (for j > 0) is a valid email.

Code:-.

```
#include <iostream>
#include <vector>
#include <string>
#include <unordered_map>
```

```cpp
#include <unordered_set>
#include <algorithm>
using namespace std;

class Solution {
public:
    vector<vector<string>> accountsMerge(vector<vector<string>>& accounts) {
        unordered_map<string, string> parent;   // Union-Find parent
        unordered_map<string, string> emailToName; // Email to account name
        unordered_map<string, vector<string>> groups;

        // Initialize union-find
        for (const auto& account : accounts) {
            string name = account[0];
            for (int i = 1; i < account.size(); ++i) {
                parent[account[i]] = account[i];
                emailToName[account[i]] = name;
                if (i > 1) {
                    unionFind(account[i - 1], account[i], parent);
                }
            }
        }

        // Group emails by their root parent
        for (const auto& [email, _] : parent) {
            string root = find(email, parent);
            groups[root].push_back(email);
        }

        // Prepare the result
        vector<vector<string>> result;
        for (auto& [root, emails] : groups) {
            sort(emails.begin(), emails.end());
            emails.insert(emails.begin(), emailToName[root]);
            result.push_back(emails);
        }
        return result;
    }

private:
    string find(string email, unordered_map<string, string>& parent) {
        if (parent[email] != email) {
            parent[email] = find(parent[email], parent); // Path compression
        }
        return parent[email];
    }

    void unionFind(string email1, string email2, unordered_map<string, string>& parent) {
        string root1 = find(email1, parent);
        string root2 = find(email2, parent);
```

```
        if (root1 != root2) {
            parent[root2] = root1; // Union
        }
    }
};

int main() {
    Solution sol;
    vector<vector<string>> accounts1 = {
        {"John", "johnsmith@mail.com", "john_newyork@mail.com"},
        {"John", "johnsmith@mail.com", "john00@mail.com"},
        {"Mary", "mary@mail.com"},
        {"John", "johnnybravo@mail.com"}
    };

    vector<vector<string>> accounts2 = {
        {"Gabe", "Gabe0@m.co", "Gabe3@m.co", "Gabe1@m.co"},
        {"Kevin", "Kevin3@m.co", "Kevin5@m.co", "Kevin0@m.co"},
        {"Ethan", "Ethan5@m.co", "Ethan4@m.co", "Ethan0@m.co"},
        {"Hanzo", "Hanzo3@m.co", "Hanzo1@m.co", "Hanzo0@m.co"},
        {"Fern", "Fern5@m.co", "Fern1@m.co", "Fern0@m.co"}
    };

    vector<vector<string>> result1 = sol.accountsMerge(accounts1);
    vector<vector<string>> result2 = sol.accountsMerge(accounts2);

    // Print results
    for (const auto& account : result1) {
        for (const string& email : account) {
            cout << email << " ";
        }
        cout << endl;
    }
    cout << endl;

    for (const auto& account : result2) {
        for (const string& email : account) {
            cout << email << " ";
        }
        cout << endl;
    }

    return 0;
}
```

Output:-

```
John john00@mail.com john_newyork@mail.com johnsmith@mail.com
Mary mary@mail.com
John johnnybravo@mail.com

Ethan Ethan0@m.co Ethan4@m.co Ethan5@m.co
Hanzo Hanzo0@m.co Hanzo1@m.co Hanzo3@m.co
Gabe Gabe0@m.co Gabe1@m.co Gabe3@m.co
Fern Fern0@m.co Fern1@m.co Fern5@m.co
Kevin Kevin0@m.co Kevin3@m.co Kevin5@m.co

[Done] exited with code=0 in 1.246 seconds
```

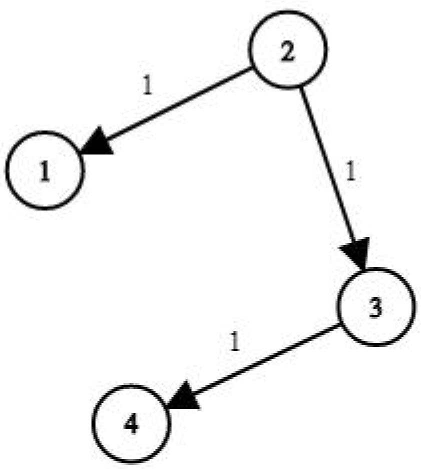
Very Hard

Network Delay Time

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

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

Example 1:



Input: times = [[2,1,1],[2,3,1],[3,4,1]], n = 4, k = 2
Output: 2

Example 2:

Input: times = [[1,2,1]], n = 2, k = 1
Output: 1

Example 3:

Input: times = [[1,2,1]], n = 2, k = 2
Output: -1

Constraints:

- 1 <= k <= n <= 100
- 1 <= times.length <= 6000
- times[i].length == 3
- 1 <= ui, vi <= n
- ui != vi
- 0 <= wi <= 100
- All the pairs (ui, vi) are unique. (i.e., no multiple edges.)

Code:-

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

int networkDelayTime(vector<vector<int>>& times, int n, int k) {
    // Step 1: Build the graph as an adjacency list
    unordered_map<int, vector<pair<int, int>>> graph;
    for (const auto& edge : times) {
        int u = edge[0], v = edge[1], w = edge[2];
        graph[u].emplace_back(v, w);
    }

    // Step 2: Initialize distances and priority queue
    vector<int> dist(n + 1, INT_MAX);
    dist[k] = 0;
    priority_queue<pair<int, int>, vector<pair<int, int>>, greater<>> pq;
    pq.emplace(0, k); // (distance, node)

    // Step 3: Dijkstra's algorithm
    while (!pq.empty()) {
        auto [currDist, node] = pq.top();
        pq.pop();

        // Skip if we already found a shorter path
        if (currDist > dist[node]) continue;

        for (const auto& [neighbor, weight] : graph[node]) {
```

```
            int newDist = currDist + weight;
            if (newDist < dist[neighbor]) {
                dist[neighbor] = newDist;
                pq.emplace(newDist, neighbor);
            }
        }
    }

    // Step 4: Find the maximum distance
    int maxDist = 0;
    for (int i = 1; i <= n; ++i) {
        if (dist[i] == INT_MAX) return -1; // Unreachable node
        maxDist = max(maxDist, dist[i]);
    }

    return maxDist;
}

int main() {
    vector<vector<int>> times1 = {{2, 1, 1}, {2, 3, 1}, {3, 4, 1}};
    int n1 = 4, k1 = 2;
    cout << networkDelayTime(times1, n1, k1) << endl; // Output: 2

    vector<vector<int>> times2 = {{1, 2, 1}};
    int n2 = 2, k2 = 1;
    cout << networkDelayTime(times2, n2, k2) << endl; // Output: 1

    vector<vector<int>> times3 = {{1, 2, 1}};
    int n3 = 2, k3 = 2;
    cout << networkDelayTime(times3, n3, k3) << endl; // Output: -1

    return 0;
}
```

Output:-

```
2
1
-1

[Done] exited with code=0 in 1.01 seconds
```