

Graph

Very Easy:

Find the Town Judge

In a town, there are n people labeled from 1 to n . There is a rumor that one of these people is secretly the town judge.

If the town judge exists, then:

1. The town judge trusts nobody.
2. Everybody (except for the town judge) trusts the town judge.
3. There is exactly one person that satisfies properties 1 and 2.

You are given an array `trust` where `trust[i] = [ai, bi]` representing that the person labeled `ai` trusts the person labeled `bi`. If a trust relationship does not exist in `trust` array, then such a trust relationship does not exist.

Return the label of the town judge if the town judge exists and can be identified, or return -1 otherwise.

Example 1:

Input: $n = 2$, `trust = [[1,2]]`

Output: 2

Example 2:

Input: $n = 3$, `trust = [[1,3],[2,3]]`

Output: 3

Example 3:

Input: $n = 3$, `trust = [[1,3],[2,3],[3,1]]`

Output: -1

Constraints:

- $1 \leq n \leq 1000$
- $0 \leq \text{trust.length} \leq 1e4$
- `trust[i].length == 2`
- All the pairs of `trust` are unique.
- $a_i \neq b_i$
- $1 \leq a_i, b_i \leq n$

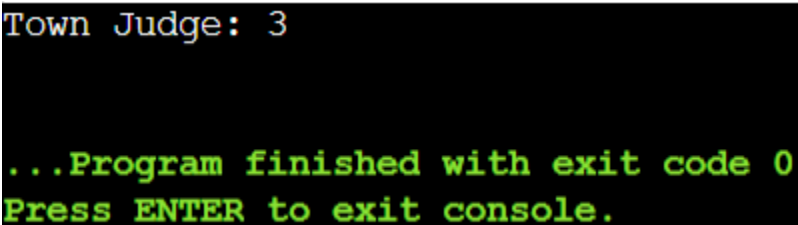
Code:-

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

int findJudge(int n, vector<vector<int>>& trust) {
    vector<int> trustScores(n + 1, 0);
    for (auto& t : trust) {
        trustScores[t[0]]--;
        trustScores[t[1]]++;
    }
    for (int i = 1; i <= n; ++i) {
        if (trustScores[i] == n - 1) {
            return i;
        }
    }
    return -1;
}

int main() {
    int n = 3;
    vector<vector<int>> trust = {{1, 3}, {2, 3}};
    cout << "Town Judge: " << findJudge(n, trust) << endl;
    return 0;
}
```

Output:-



```
Town Judge: 3

...Program finished with exit code 0
Press ENTER to exit console.
```

Easy

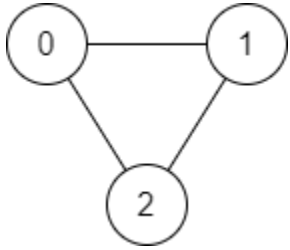
Find if Path Exists in Graph

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

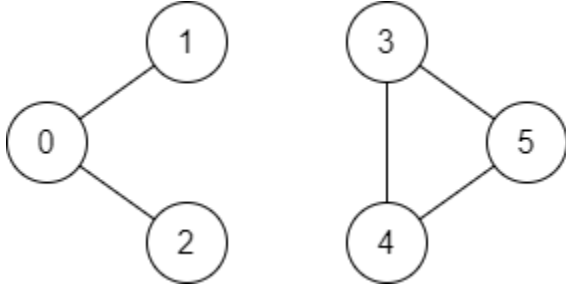
Output: true

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:

- $1 \leq n \leq 2 * 10^5$
- $0 \leq \text{edges.length} \leq 2 * 10^5$
- $\text{edges}[i].\text{length} == 2$
- $0 \leq u_i, v_i \leq n - 1$
- $u_i \neq v_i$
- $0 \leq \text{source}, \text{destination} \leq n - 1$
- There are no duplicate edges.
- There are no self edges.

Code:-

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

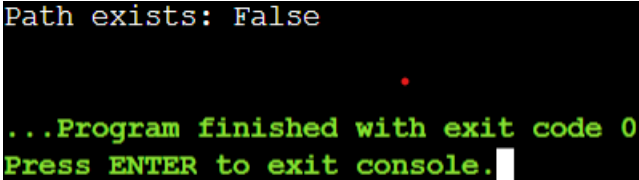
```
using namespace std;
```

```
bool validPath(int n, vector<vector<int>>& edges, int source, int destination) {
    unordered_map<int, vector<int>> graph;
    for (auto& edge : edges) {
        graph[edge[0]].push_back(edge[1]);
        graph[edge[1]].push_back(edge[0]);
    }

    vector<bool> visited(n, false);
    queue<int> q;
    q.push(source);
    while (!q.empty()) {
        int node = q.front();
        q.pop();
        if (node == destination) return true;
        if (!visited[node]) {
            visited[node] = true;
            for (int neighbor : graph[node]) {
                if (!visited[neighbor]) q.push(neighbor);
            }
        }
    }
    return false;
}

int main() {
    int n = 6;
    vector<vector<int>> edges = {{0, 1}, {0, 2}, {3, 5}, {5, 4}, {4, 3}};
    int source = 0, destination = 5;
    cout << "Path exists: " << (validPath(n, edges, source, destination) ? "True" : "False") << endl;
    return 0;
}
```

Output:-



```
Path exists: False

...Program finished with exit code 0
Press ENTER to exit console.
```

Medium

Course Schedule II

There are a total of numCourses courses you have to take, labeled from 0 to numCourses - 1. You are given an array prerequisites where prerequisites[i] = [ai, bi] indicates that you must take course bi first if you want to take course ai.

For example, the pair [0, 1], indicates that to take course 0 you have to first take course 1.

Return the ordering of courses you should take to finish all courses. If there are many valid answers, return any of them. If it is impossible to finish all courses, return an empty array.

Example 1:

Input: numCourses = 2, prerequisites = [[1,0]]

Output: [0,1]

Explanation: There are a total of 2 courses to take. To take course 1 you should have finished course 0. So the correct course order is [0,1].

Example 2:

Input: numCourses = 4, prerequisites = [[1,0],[2,0],[3,1],[3,2]]

Output: [0,2,1,3]

Explanation: There are a total of 4 courses to take. To take course 3 you should have finished both courses 1 and 2. Both courses 1 and 2 should be taken after you finished course 0.

So one correct course order is [0,1,2,3]. Another correct ordering is [0,2,1,3].

Example 3:

Input: numCourses = 1, prerequisites = []

Output: [0]

Constraints:

- $1 \leq \text{numCourses} \leq 2000$
- $0 \leq \text{prerequisites.length} \leq \text{numCourses} * (\text{numCourses} - 1)$
- $\text{prerequisites}[i].\text{length} == 2$

- $0 \leq a_i, b_i < \text{numCourses}$
- $a_i \neq b_i$
- All the pairs $[a_i, b_i]$ are distinct.

Code:-

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

vector<int> findOrder(int numCourses, vector<vector<int>>& prerequisites) {
    vector<vector<int>> graph(numCourses);
    vector<int> indegree(numCourses, 0);
    for (auto& prereq : prerequisites) {
        graph[prereq[1]].push_back(prereq[0]);
        indegree[prereq[0]]++;
    }

    queue<int> q;
    for (int i = 0; i < numCourses; ++i) {
        if (indegree[i] == 0) q.push(i);
    }

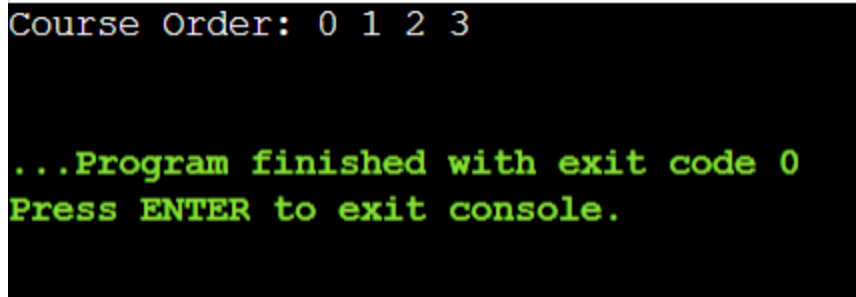
    vector<int> order;
    while (!q.empty()) {
        int course = q.front();
        q.pop();
        order.push_back(course);
        for (int next : graph[course]) {
            indegree[next]--;
            if (indegree[next] == 0) q.push(next);
        }
    }

    return order.size() == numCourses ? order : vector<int>();
}

int main() {
    int numCourses = 4;
    vector<vector<int>> prerequisites = {{1, 0}, {2, 0}, {3, 1}, {3, 2}};
    vector<int> order = findOrder(numCourses, prerequisites);
    cout << "Course Order: ";
    for (int course : order) {
        cout << course << " ";
    }
}
```

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

Output:-



```
Course Order: 0 1 2 3  
  
...Program finished with exit code 0  
Press ENTER to exit console.
```

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:

- $1 \leq \text{accounts.length} \leq 1000$
- $2 \leq \text{accounts}[i].\text{length} \leq 10$
- $1 \leq \text{accounts}[i][j].\text{length} \leq 30$
- $\text{accounts}[i][0]$ consists of English letters.
- $\text{accounts}[i][j]$ (for $j > 0$) is a valid email.

Code:-

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

class Solution {
public:
    vector<int> parent;

    int find(int x) {
        if (parent[x] != x) {
            parent[x] = find(parent[x]);
        }
        return parent[x];
    }

    void unionSets(int x, int y) {
        parent[find(x)] = find(y);
    }
};
```



```

}

vector<vector<string>> accountsMerge(vector<vector<string>>& accounts) {
    unordered_map<string, int> emailToIndex;
    unordered_map<string, string> emailToName;
    int id = 0;
    parent.resize(10001);
    for (int i = 0; i < 10001; ++i) parent[i] = i;

    for (auto& account : accounts) {
        string name = account[0];
        for (int i = 1; i < account.size(); ++i) {
            emailToName[account[i]] = name;
            if (!emailToIndex.count(account[i])) {
                emailToIndex[account[i]] = id++;
            }
            unionSets(emailToIndex[account[1]], emailToIndex[account[i]]);
        }
    }

    unordered_map<int, vector<string>> mergedAccounts;
    for (auto& [email, idx] : emailToIndex) {
        mergedAccounts[find(idx)].push_back(email);
    }

    vector<vector<string>> result;
    for (auto& [_, emails] : mergedAccounts) {
        sort(emails.begin(), emails.end());
        vector<string> account = {emailToName[emails[0]]};
        account.insert(account.end(), emails.begin(), emails.end());
        result.push_back(account);
    }
    return result;
}
};

```

```

int main() {
    Solution sol;
    vector<vector<string>> accounts = {
        {"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>> merged = sol.accountsMerge(accounts);
    for (auto& account : merged) {
        cout << account[0] << ": ";
        for (int i = 1; i < account.size(); ++i) {
            cout << account[i] << " ";
        }
    }
}

```

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

Output:-

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

...Program finished with exit code 0
Press ENTER to exit console.
```

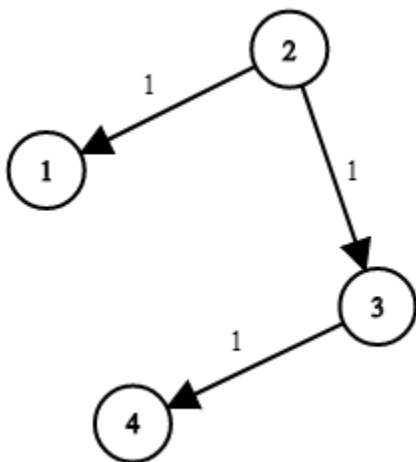
Very Hard

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 $\text{times}[i] = (u_i, v_i, w_i)$, where u_i is the source node, v_i is the target node, and w_i is the time it takes for a signal to travel from source to target.

We will send a signal from a given node k . Return the 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: $\text{times} = [[2,1,1],[2,3,1],[3,4,1]]$, $n = 4$, $k = 2$

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:

- $1 \leq k \leq n \leq 100$
- $1 \leq \text{times.length} \leq 6000$
- $\text{times}[i].\text{length} == 3$
- $1 \leq u_i, v_i \leq n$
- $u_i \neq v_i$
- $0 \leq w_i \leq 100$
- All the pairs (u_i, v_i) are unique. (i.e., no multiple edges.)

Code:-

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

int networkDelayTime(vector<vector<int>>& times, int n, int k) {
    vector<vector<pair<int, int>>> graph(n + 1);
    for (auto& time : times) {
        graph[time[0]].emplace_back(time[1], time[2]);
    }

    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);

    while (!pq.empty()) {
        auto [time, node] = pq.top();
        pq.pop();
        if (time > dist[node]) continue;
        for (auto& [neighbor, weight] : graph[node]) {
            if (dist[node] + weight < dist[neighbor]) {
                dist[neighbor] = dist[node] + weight;
            }
        }
    }

    return dist[n] == INT_MAX ? -1 : dist[n];
}
```

```

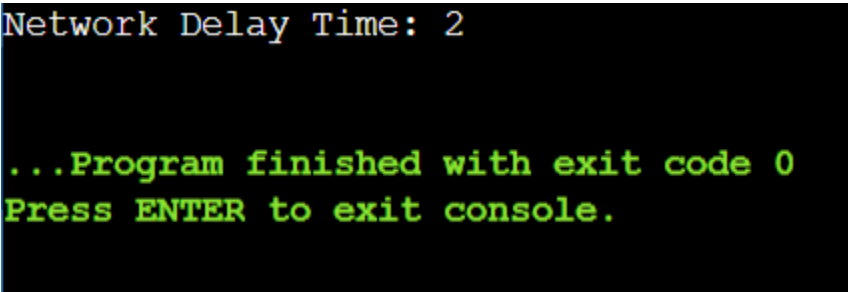
        pq.emplace(dist[neighbor], neighbor);
    }
}

int maxTime = 0;
for (int i = 1; i <= n; ++i) {
    if (dist[i] == INT_MAX) return -1;
    maxTime = max(maxTime, dist[i]);
}
return maxTime;
}

int main() {
    vector<vector<int>> times = {{2, 1, 1}, {2, 3, 1}, {3, 4, 1}};
    int n = 4, k = 2;
    cout << "Network Delay Time: " << networkDelayTime(times, n, k) << endl;
    return 0;
}

```

Output:-



```

Network Delay Time: 2

...Program finished with exit code 0
Press ENTER to exit console.

```