

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

```

int findJudge(int n, vector<vector<int>>& trust) {
    vector<int> trustCount(n + 1, 0);

    for (const auto& t : trust) {
        trustCount[t[0]]--; // Person t[0] trusts someone, so decrease their trust score
        trustCount[t[1]]++; // Person t[1] is trusted by t[0], so increase their trust score
    }

    for (int i = 1; i <= n; i++) {
        if (trustCount[i] == n - 1) {
            return i; // The town judge is the person with trustCount == n-1
        }
    }

    return -1; // No town judge exists
}

int main() {
    vector<vector<int>> trust = {{1, 2}};
    int n = 2;
    cout << findJudge(n, trust) << endl; // Output: 2
    return 0;
}

```

OUTPUT :

2

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

Easy

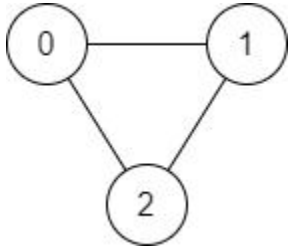
Find if Path Exists in Graph

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

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

```
bool validPath(int n, vector<vector<int>>& edges, int source, int destination) {
    if (source == destination) return true;
    vector<vector<int>> graph(n);
    for (const 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);
visited[source] = true;
while (!q.empty()) {
    int node = q.front();
    q.pop();
    for (int neighbor : graph[node]) {
        if (!visited[neighbor]) {
            if (neighbor == destination) return true;
            visited[neighbor] = true;
            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;
    int destination = 5;
    cout << (validPath(n, edges, source, destination) ? "true" : "false") << endl;
    return 0;
}

```

OUTPUT :



```

false

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

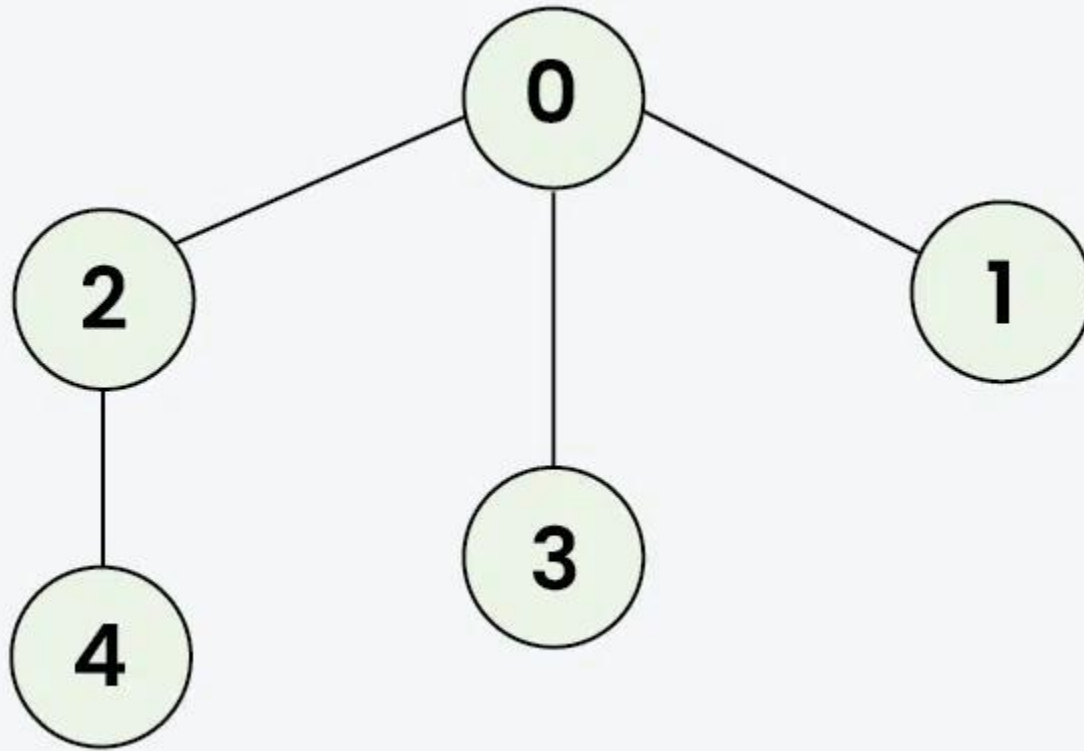
```

DFS of Graph

Given a connected undirected graph represented by an adjacency list `adj`, which is a vector of vectors where each `adj[i]` represents the list of vertices connected to vertex `i`. Perform a Depth First Traversal (DFS) starting from vertex 0, visiting vertices from left to right as per the adjacency list, and return a list containing the DFS traversal of the graph.

Note: Do traverse in the same order as they are in the adjacency list.

Example 1:



Input: `adj = [[2,3,1], [0], [0,4], [0], [2]]`

Output: `[0, 2, 4, 3, 1]`

Explanation: Starting from 0, the DFS traversal proceeds as follows:

Visit 0 → Output: 0

Visit 2 (the first neighbor of 0) → Output: 0, 2

Visit 4 (the first neighbor of 2) → Output: 0, 2, 4

Backtrack to 2, then backtrack to 0, and visit 3 → Output: 0, 2, 4, 3

Finally, backtrack to 0 and visit 1 → Final Output: 0, 2, 4, 3, 1

Constraints:

- $1 \leq \text{adj.size()} \leq 1e4$
- $1 \leq \text{adj}[i][j] \leq 1e4$

CODE -

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

void dfs(int node, vector<vector<int>>& adj, vector<bool>& visited, vector<int>& result) {
    visited[node] = true;
    result.push_back(node);
    for (int neighbor : adj[node]) {
        if (!visited[neighbor]) {
```

```

        dfs(neighbor, adj, visited, result);
    }
}

vector<int> dfsOfGraph(vector<vector<int>>& adj) {
    int n = adj.size();
    vector<bool> visited(n, false);
    vector<int> result;
    dfs(0, adj, visited, result);
    return result;
}

int main() {
    vector<vector<int>> adj = {{2, 3, 1}, {0}, {0, 4}, {0}, {2}};
    vector<int> result = dfsOfGraph(adj);

    for (int node : result) {
        cout << node << " ";
    }
    cout << endl;
    return 0;
}

```

OUTPUT -



```

0 2 4 3 1

...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.

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 <vector>
#include <unordered_map>
#include <unordered_set>
#include <string>
#include <algorithm>
#include <iostream>
using namespace std;
class Solution {
public:
    unordered_map<string, string> parent;
```

```
unordered_map<string, string> emailToName
string find(string email) {
if (parent[email] != email) {
```



```
parent[email] = find(parent[email]);  
  
    }  
  
    return parent[email];  
  
}
```

```
void unionSets(string email1, string email2) {  
  
    string root1 = find(email1);  
  
    string root2 = find(email2);  
  
    if (root1 != root2) {  
  
        parent[root2] = root1;  
  
    }  
  
}
```

```
vector<vector<string>> accountsMerge(vector<vector<string>>& accounts) {  
  
    for (const auto& account : accounts) {  
  
        string name = account[0];  
  
        for (int i = 1; i < account.size(); i++) {  
  
            string email = account[i];  
  
            if (parent.find(email) == parent.end()) {  
  
                parent[email] = email;  
  
            }  
  
        }  
  
    }  
  
}
```

```
}
```

```
    emailToName[email] = name;
```

```
    if (i > 1) {
```

```
        unionSets(account[i], account[i - 1]);
```

```
    }
```

```
}
```

```
}
```

```
unordered_map<string, unordered_set<string>> mergedAccounts;
```

```
for (const auto& entry : parent) {
```

```
string root = find(entry.first);
```

```
    mergedAccounts[root].insert(entry.first);
```

```
}
```

```
vector<vector<string>> result;
```

```
for (const auto& entry : mergedAccounts) {
```

```
    vector<string> account;
```

```
    account.push_back(emailToName[entry.first]);
```

```
    for (const string& email : entry.second) {
```

```
        account.push_back(email);
```

```

    }

    sort(account.begin() + 1, account.end());

    result.push_back(account);

}

return result;

}

};

int main() {

    Solution solution;

    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>> result = solution.accountsMerge(accounts);

    for (const auto& account : result) {

        for (const auto& email : account) {

```

```
        cout << email << " ";

    }

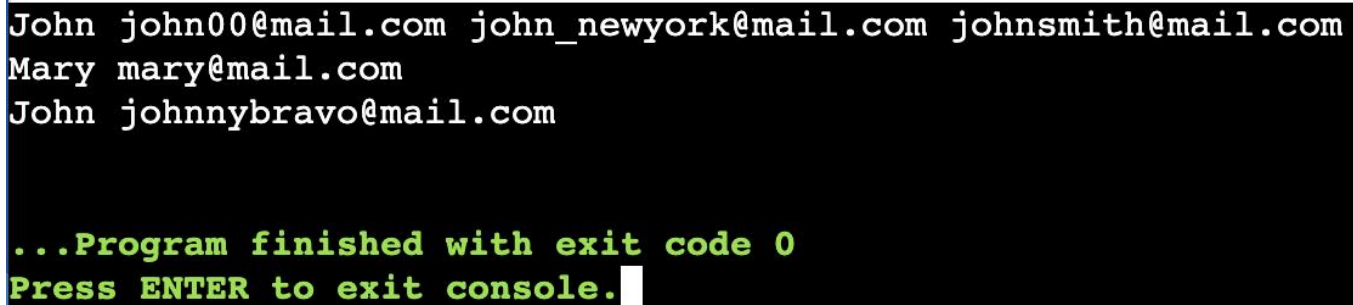
    cout << endl;

}

return 0;

}
```

OUTPUT -

A screenshot of a terminal window with a black background and white text. The output shows three lines of names and email addresses: 'John john00@mail.com john_newyork@mail.com johnsmith@mail.com', 'Mary mary@mail.com', and 'John johnnybravo@mail.com'. Below this, a green message states '...Program finished with exit code 0' and 'Press ENTER to exit console.' followed by a white cursor block.

```
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.
```

Evaluate Division

You are given an array of variable pairs equations and an array of real numbers values, where equations[i] = [Ai, Bi] and values[i] represent the equation $A_i / B_i = \text{values}[i]$. Each Ai or Bi is a string that represents a single variable.

You are also given some queries, where queries[j] = [Cj, Dj] represents the jth query where you must find the answer for $C_j / D_j = ?$.

Return the answers to all queries. If a single answer cannot be determined, return -1.0.

Note: The input is always valid. You may assume that evaluating the queries will not result in division by zero and that there is no contradiction.

Note: The variables that do not occur in the list of equations are undefined, so the answer cannot be determined for them.

Example 1:

Input: equations = [["a","b"],["b","c"]], values = [2.0,3.0], queries =
[["a","c"],["b","a"],["a","e"],["a","a"],["x","x"]]

Output: [6.00000,0.50000,-1.00000,1.00000,-1.00000]

Explanation:

Given: $a / b = 2.0$, $b / c = 3.0$

queries are: $a / c = ?$, $b / a = ?$, $a / e = ?$, $a / a = ?$, $x / x = ?$

return: [6.0, 0.5, -1.0, 1.0, -1.0]

note: x is undefined => -1.0

Constraints:

- $1 \leq \text{equations.length} \leq 20$
- $\text{equations}[i].\text{length} == 2$
- $1 \leq \text{Ai.length}, \text{Bi.length} \leq 5$
- $\text{values.length} == \text{equations.length}$
- $0.0 < \text{values}[i] \leq 20.0$
- $1 \leq \text{queries.length} \leq 20$
- $\text{queries}[i].\text{length} == 2$
- $1 \leq \text{Cj.length}, \text{Dj.length} \leq 5$
- Ai, Bi, Cj, Dj consist of lower case English letters and digits.

CODE -

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

class Solution {
public:
    unordered_map<string, unordered_map<string, double>>> graph;

    void dfs(const string& src, const string& dest, double value, unordered_map<string, bool>& visited,
double& result) {
        if (src == dest) {
```

```

        result = value;
        return;
    }
    visited[src] = true;
    for (const auto& neighbor : graph[src]) {
        if (!visited[neighbor.first]) {
            dfs(neighbor.first, dest, value * neighbor.second, visited, result);
            if (result != -1) return;
        }
    }
}

vector<double> calcEquation(vector<vector<string>>& equations, vector<double>& values,
vector<vector<string>>& queries) {
    for (int i = 0; i < equations.size(); ++i) {
        graph[equations[i][0]][equations[i][1]] = values[i];
        graph[equations[i][1]][equations[i][0]] = 1 / values[i];
    }

    vector<double> result;
    for (const auto& query : queries) {
        unordered_map<string, bool> visited;
        double res = -1;
        dfs(query[0], query[1], 1.0, visited, res);
        result.push_back(res);
    }

    return result;
}

};

int main() {
    Solution solution;
    vector<vector<string>> equations = {{ "a", "b"}, {"b", "c"}};
    vector<double> values = {2.0, 3.0};
    vector<vector<string>> queries = {{ "a", "c"}, {"b", "a"}, {"a", "e"}, {"a", "a"}, {"x", "x"}};

    vector<double> result = solution.calcEquation(equations, values, queries);

    for (double res : result) {
        cout << res << " ";
    }
    cout << endl;

    return 0;
}

```

OUTPUT -

```
6 0.5 -1 1 1
```

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