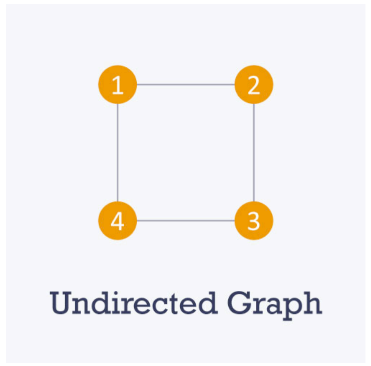
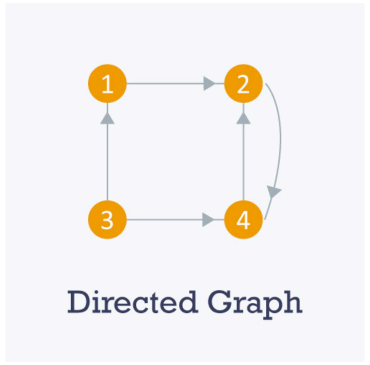
GRAPH ALGORITHMS

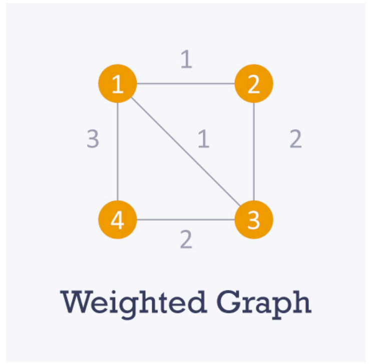
Oct 19, 2019

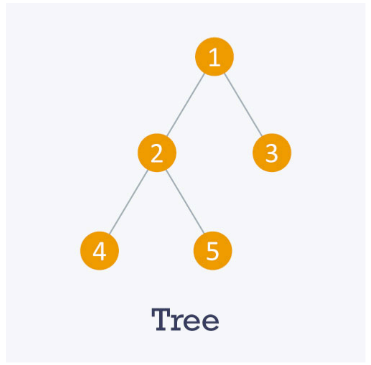
GRAPH DEFINITIONS

- A Graph is a non-linear data structure consisting of nodes and edges.
    - Undirected graph: bi-directional edges
    - Directed graph: uni-directional edges
    - Weighted graph: weighted edges
    - Tree: A special kind of graph with a root node that has no parent and all other nodes have a unique parent node.





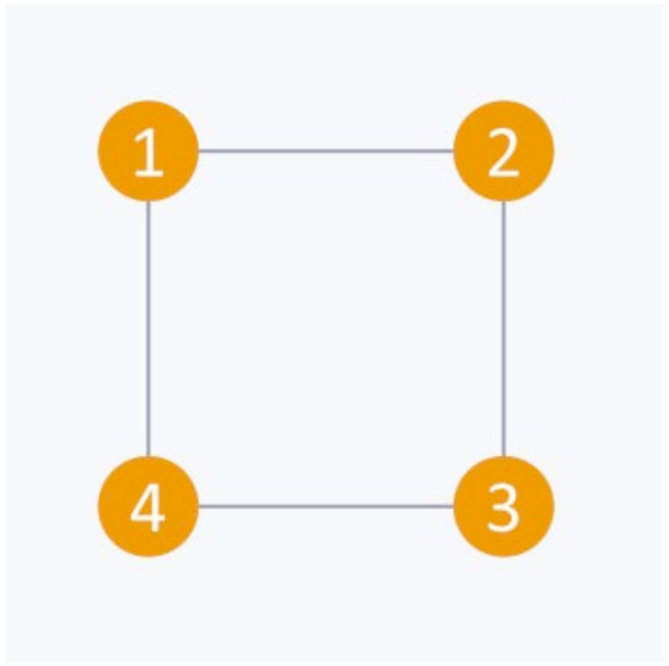




GRAPH REPRESENTATION

[[0, 1, 0, 1],
[1, 0, 1, 0],
[0, 1, 0, 1],
[1, 0, 1, 0]]

(Adjacency matrix)



[[2, 4],
[1, 3],
[2, 4],
[1, 3]]

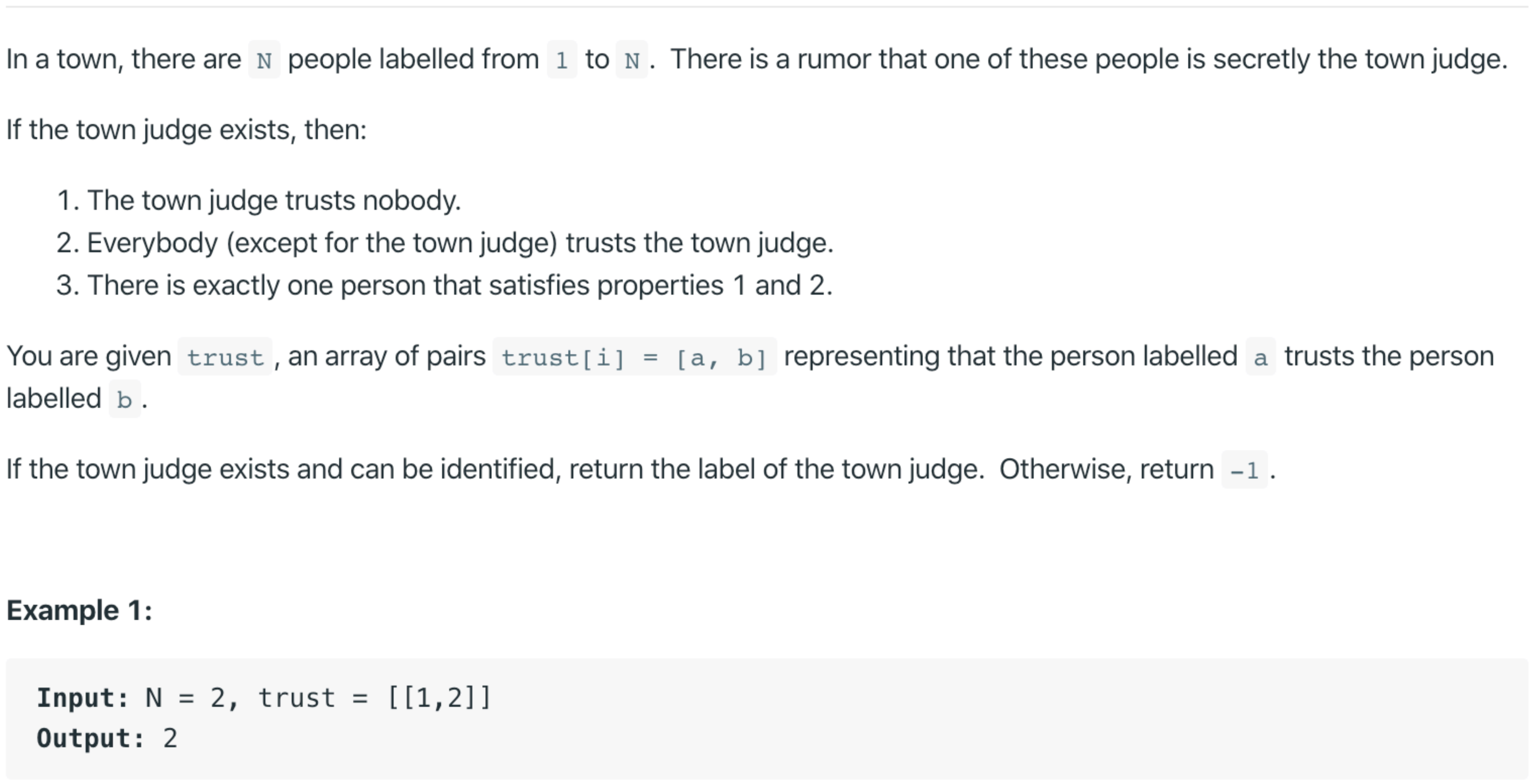
(Adjacency list)

- Adjacency matrix: a V by V binary matrix A, when element A(i, j) is 1 if there is an edge from vertex I to vertex j.
- Adjacency list: an array A of array, each element of the array A(i) is an array, which contains all the vertices that are adjacent to vertex i.

POPULAR GRAPH RELATED PROBLEMS AND METHODS

- Graph Traversal
  - Task: traverse every node of the graph
  - Method: BFS/DFS
- Find cycles in a graph
  - Task: Find cycles
  - Method: Union find
- Shortest Path
  - Task: finding a path between 2 vertices in a graph such that the total sum of the edges weights is minimum.
  - Method: Bellman Ford's Algorithm. Dijkstra's Algorithm
- Maximum Flow
  - Task: A flow network is defined as a directed graph with a source, a sink and several other nodes connected with edges. Each edge has a flow capacity. Find the maximum flow allowed.
  - Method: Ford-Fulkerson Algorithm. Dinic's Algorithm

1. FIND THE TOWN JUDGE (ADJACENT LIST)



In a town, there are `N` people labelled 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.

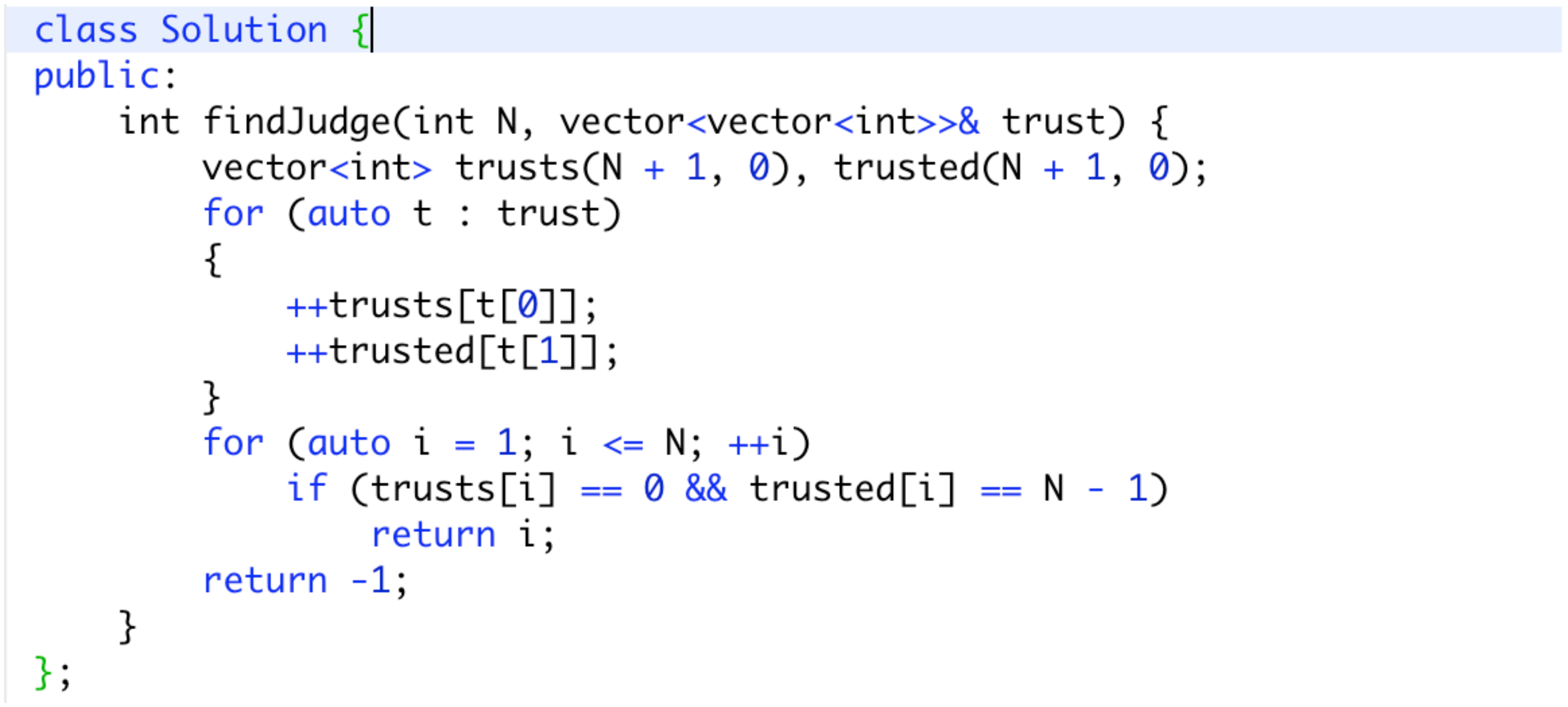
You are given `trust`, an array of pairs `trust[i] = [a, b]` representing that the person labelled `a` trusts the person labelled `b`.

If the town judge exists and can be identified, return the label of the town judge. Otherwise, return `-1`.

Example 1:

```
Input: N = 2, trust = [[1,2]]
Output: 2
```

1. FIND THE TOWN JUDGE (ADJACENT LIST)



```cpp
class Solution {
public:
    int findJudge(int N, vector<vector<int>>& trust) {
        vector<int> trusts(N + 1, 0), trusted(N + 1, 0);
        for (auto t : trust)
        {
            ++trusts[t[0]];
            ++trusted[t[1]];
        }
        for (auto i = 1; i <= N; ++i)
            if (trusts[i] == 0 && trusted[i] == N - 1)
                return i;
        return -1;
    }
};
```

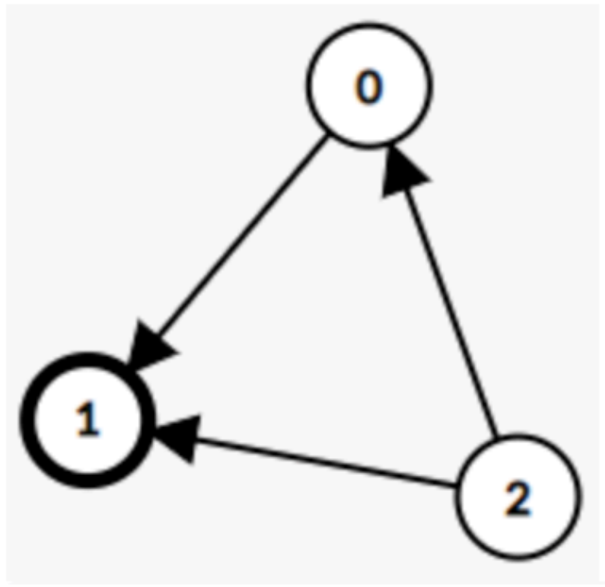
2. FIND THE CELERITY (ADJACENT MATRIX)

Suppose you are at a party with `n` people (labeled from `0` to `n - 1`) and among them, there may exist one celebrity. The definition of a celebrity is that all the other `n - 1` people know him/her but he/she does not know any of them.

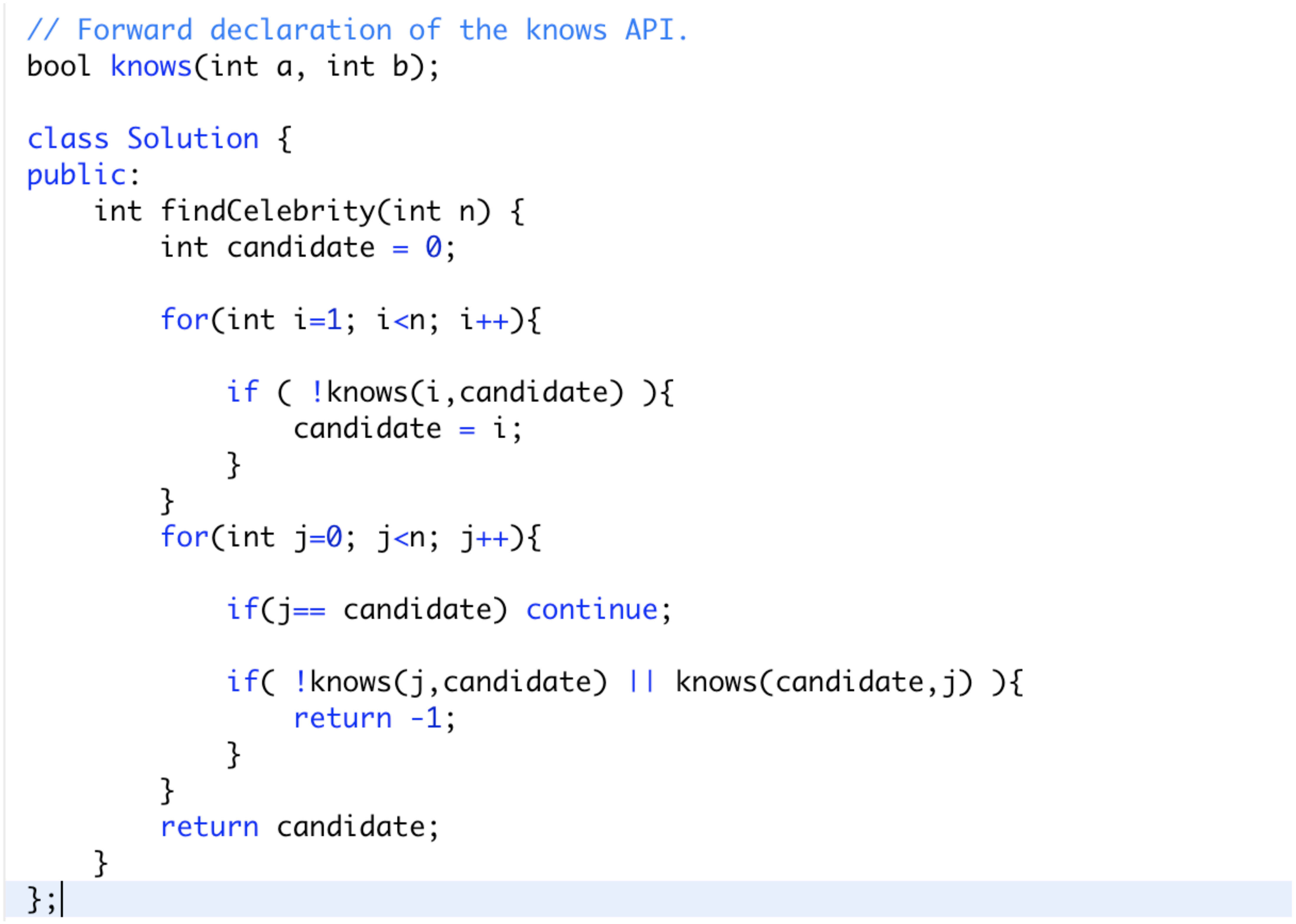
Now you want to find out who the celebrity is or verify that there is not one. The only thing you are allowed to do is to ask questions like: "Hi, A. Do you know B?" to get information of whether A knows B. You need to find out the celebrity (or verify there is not one) by asking as few questions as possible (in the asymptotic sense).

You are given a helper function `bool knows(a, b)` which tells you whether A knows B. Implement a function `int findCelebrity(n)`. There will be exactly one celebrity if he/she is in the party. Return the celebrity's label if there is a celebrity in the party. If there is no celebrity, return `-1`.

Example 1:

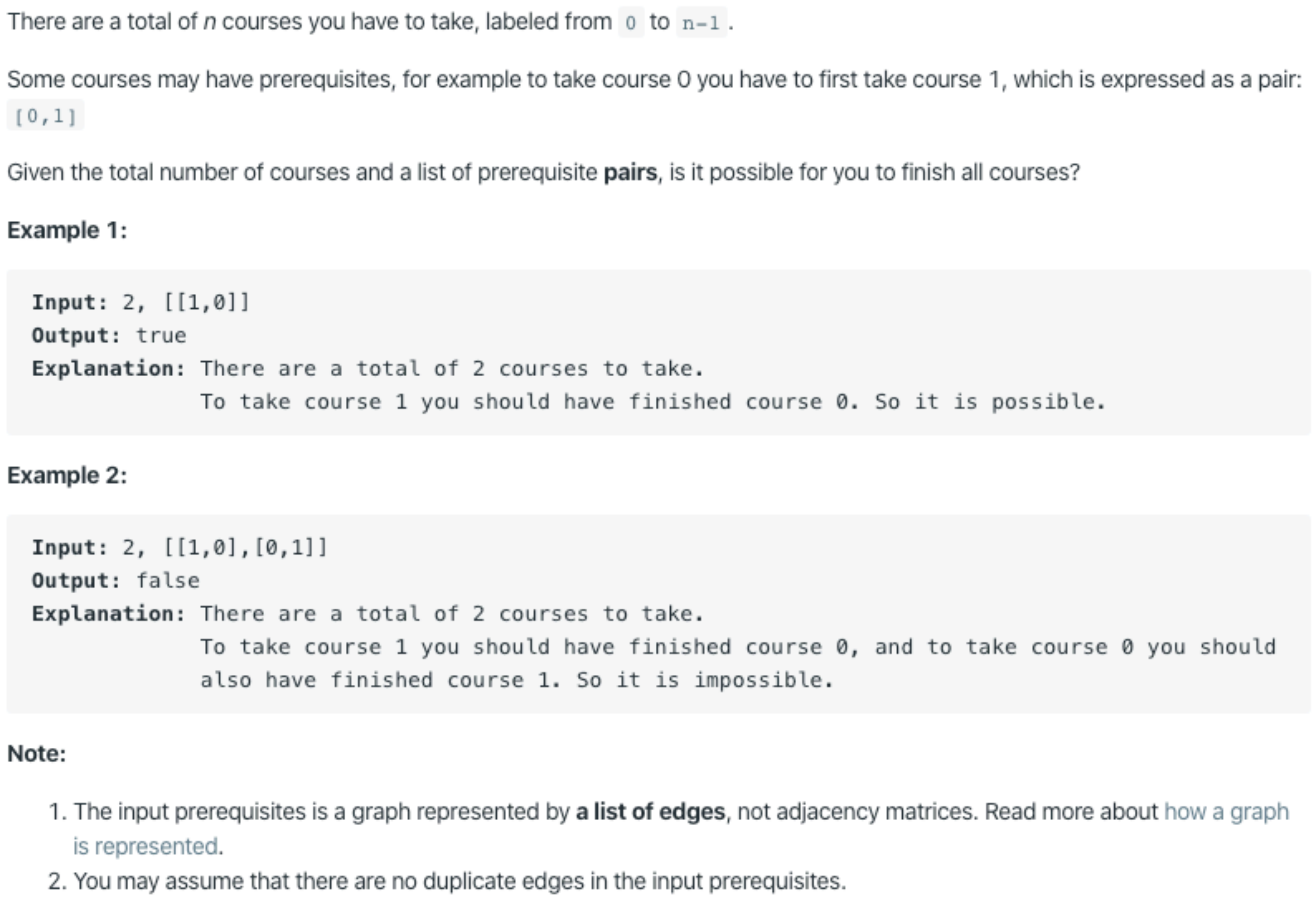


2. FIND THE CELERITY



```
// Forward declaration of the knows API.
bool knows(int a, int b);

class Solution {
public:
    int findCelebrity(int n) {
        int candidate = 0;

        for(int i=1; i<n; i++){

            if ( !knows(i,candidate) ){
                candidate = i;
            }
        }
        for(int j=0; j<n; j++){

            if(j== candidate) continue;

            if( !knows(j,candidate) || knows(candidate,j) ){
                return -1;
            }
        }
        return candidate;
    }
};
```

3. COURSE SCHEDULE (TOPOLOGICAL SORT)



There are a total of *n* courses you have to take, labeled from `0` to `n-1`.

Some courses may have prerequisites, for example to take course 0 you have to first take course 1, which is expressed as a pair: `[0,1]`

Given the total number of courses and a list of prerequisite **pairs**, is it possible for you to finish all courses?

Example 1:

```
Input: 2, [[1,0]]
Output: true
Explanation: There are a total of 2 courses to take.
             To take course 1 you should have finished course 0. So it is possible.
```

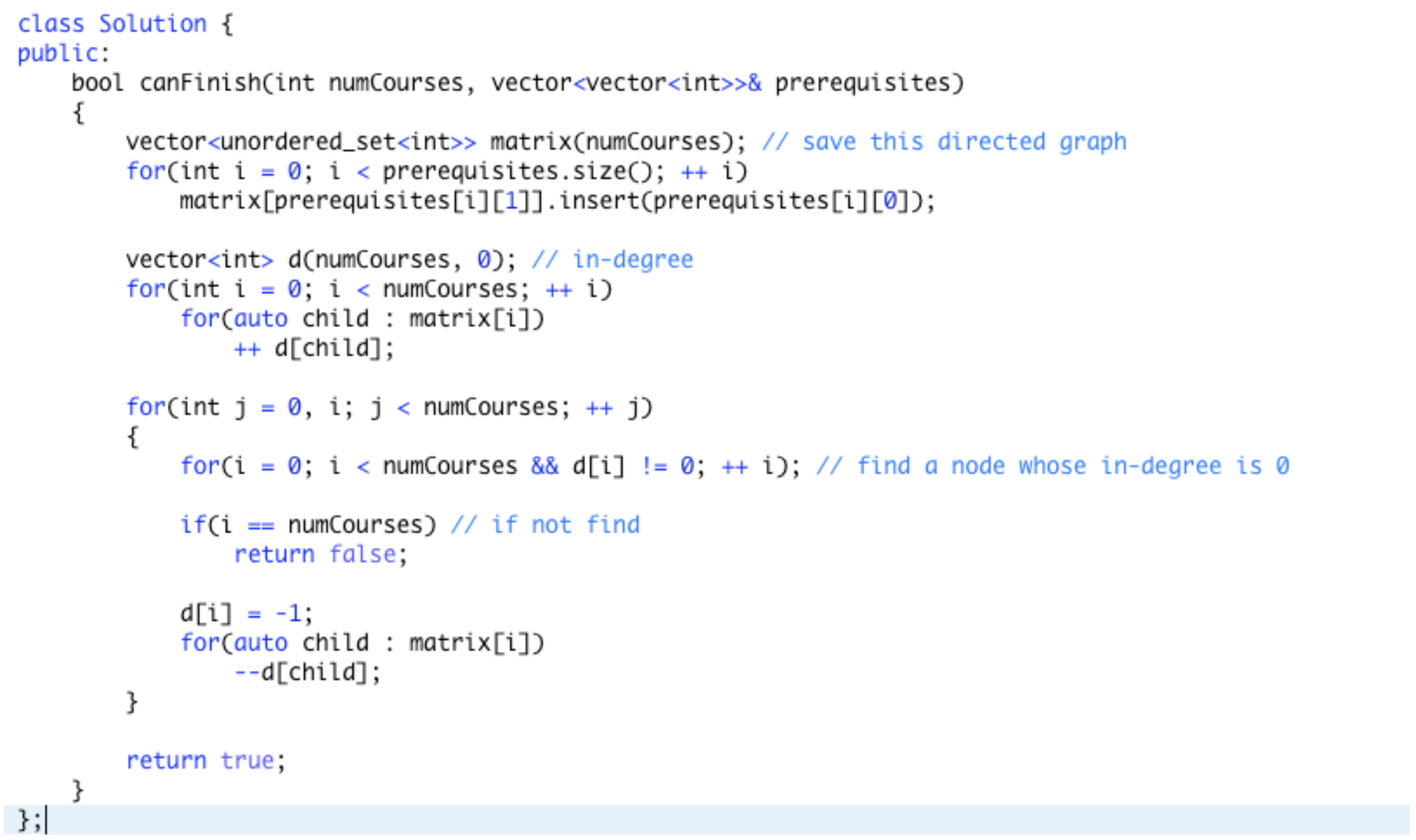
Example 2:

```
Input: 2, [[1,0],[0,1]]
Output: false
Explanation: There are a total of 2 courses to take.
             To take course 1 you should have finished course 0, and to take course 0 you should
             also have finished course 1. So it is impossible.
```

Note:

1. The input prerequisites is a graph represented by **a list of edges**, not adjacency matrices. Read more about how a graph is represented.
2. You may assume that there are no duplicate edges in the input prerequisites.

3. COURSE SCHEDULE (TOPOLOGICAL SORT)



```cpp
class Solution {
public:
    bool canFinish(int numCourses, vector<vector<int>>& prerequisites)
    {
        vector<unordered_set<int>> matrix(numCourses); // save this directed graph
        for(int i = 0; i < prerequisites.size(); ++ i)
            matrix[prerequisites[i][1]].insert(prerequisites[i][0]);

        vector<int> d(numCourses, 0); // in-degree
        for(int i = 0; i < numCourses; ++ i)
            for(auto child : matrix[i])
                ++ d[child];

        for(int j = 0, i; j < numCourses; ++ j)
        {
            for(i = 0; i < numCourses && d[i] != 0; ++ i); // find a node whose in-degree is 0

            if(i == numCourses) // if not find
                return false;

            d[i] = -1;
            for(auto child : matrix[i])
                --d[child];
        }

        return true;
    }
};
```

4. GRAPH VALID TREE (UNION FIND)

Given `n` nodes labeled from `0` to `n-1` and a list of undirected edges (each edge is a pair of nodes), write a function to check whether these edges make up a valid tree.

Example 1:

```
Input: n = 5, and edges = [[0,1], [0,2], [0,3], [1,4]]
Output: true
```

Example 2:

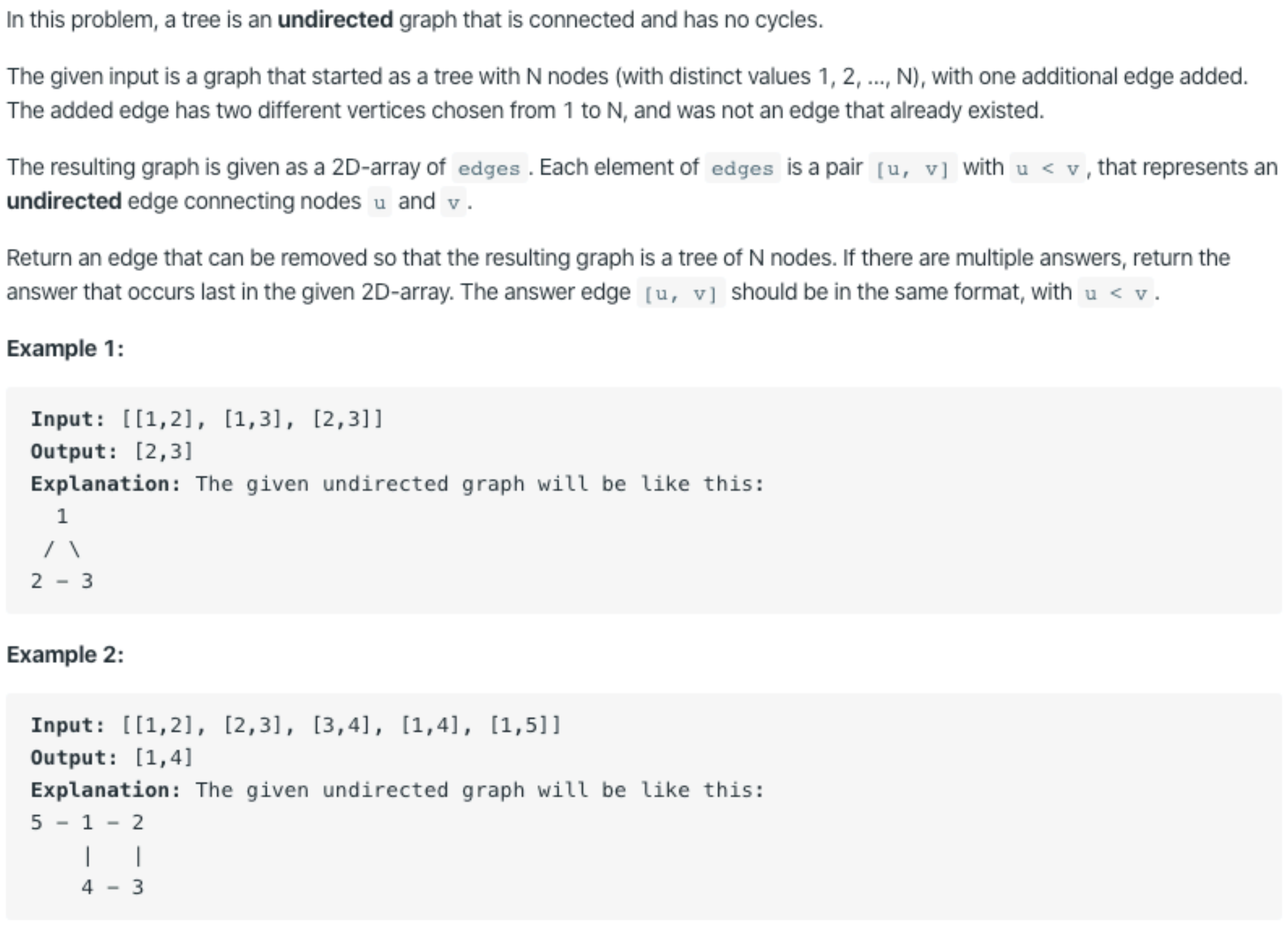
```
Input: n = 5, and edges = [[0,1], [1,2], [2,3], [1,3], [1,4]]
Output: false
```

Note: you can assume that no duplicate edges will appear in `edges`. Since all edges are undirected, `[0,1]` is the same as `[1,0]` and thus will not appear together in `edges`.

4. GRAPH VALID TREE (UNION FIND)

```cpp
class Solution {
public:
    bool validTree(int n, vector<vector<int>>& edges) {
        vector<int> nodes(n,0);
        for(int i=0; i<n; i++) nodes[i] = i;
        for(int i=0; i<edges.size(); i++){
            int f = edges[i][0];
            int s = edges[i][1];
            while(nodes[f]!=f) f = nodes[f];
            while(nodes[s]!=s) s = nodes[s];
            if(nodes[f] == nodes[s]) return false;
            nodes[s] = f;
        }
        return edges.size() == n-1;
    }
};
```

5. REDUNDANT CONNECTION (UNION FIND)



In this problem, a tree is an **undirected** graph that is connected and has no cycles.

The given input is a graph that started as a tree with N nodes (with distinct values 1, 2, ..., N), with one additional edge added. The added edge has two different vertices chosen from 1 to N, and was not an edge that already existed.

The resulting graph is given as a 2D-array of `edges`. Each element of `edges` is a pair `[u, v]` with `u < v`, that represents an **undirected** edge connecting nodes `u` and `v`.

Return an edge that can be removed so that the resulting graph is a tree of N nodes. If there are multiple answers, return the answer that occurs last in the given 2D-array. The answer edge `[u, v]` should be in the same format, with `u < v`.

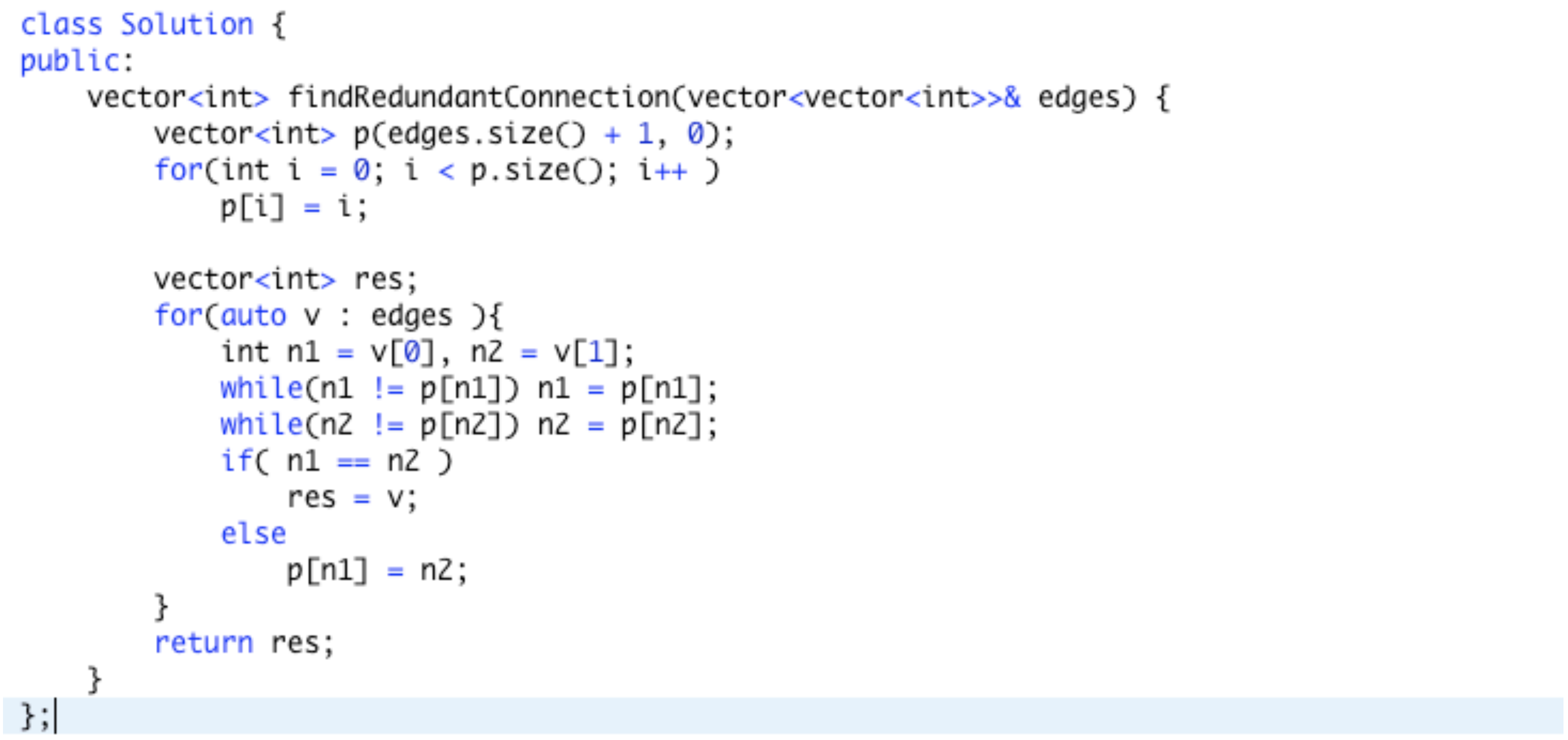
Example 1:

```
Input: [[1,2], [1,3], [2,3]]
Output: [2,3]
Explanation: The given undirected graph will be like this:
  1
 / \
2 - 3
```

Example 2:

```
Input: [[1,2], [2,3], [3,4], [1,4], [1,5]]
Output: [1,4]
Explanation: The given undirected graph will be like this:
5 - 1 - 2
    |   |
    4 - 3
```

5. REDUNDANT CONNECTION (UNION FIND)



```cpp
class Solution {
public:
    vector<int> findRedundantConnection(vector<vector<int>>& edges) {
        vector<int> p(edges.size() + 1, 0);
        for(int i = 0; i < p.size(); i++ )
            p[i] = i;

        vector<int> res;
        for(auto v : edges ){
            int n1 = v[0], n2 = v[1];
            while(n1 != p[n1]) n1 = p[n1];
            while(n2 != p[n2]) n2 = p[n2];
            if( n1 == n2 )
                res = v;
            else
                p[n1] = n2;
        }
        return res;
    }
};
```

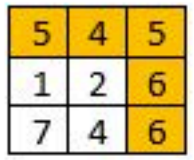
6.PATH WITH MAXIMUM MINIMUM VALUE (DIJKSTRA'S ALGORITHM)

Given a matrix of integers `A` with `R` rows and `C` columns, find the **maximum** score of a path starting at `[0,0]` and ending at `[R-1,C-1]`.

The *score* of a path is the **minimum** value in that path. For example, the value of the path 8 → 4 → 5 → 9 is 4.

A *path* moves some number of times from one visited cell to any neighbouring unvisited cell in one of the 4 cardinal directions (north, east, west, south).

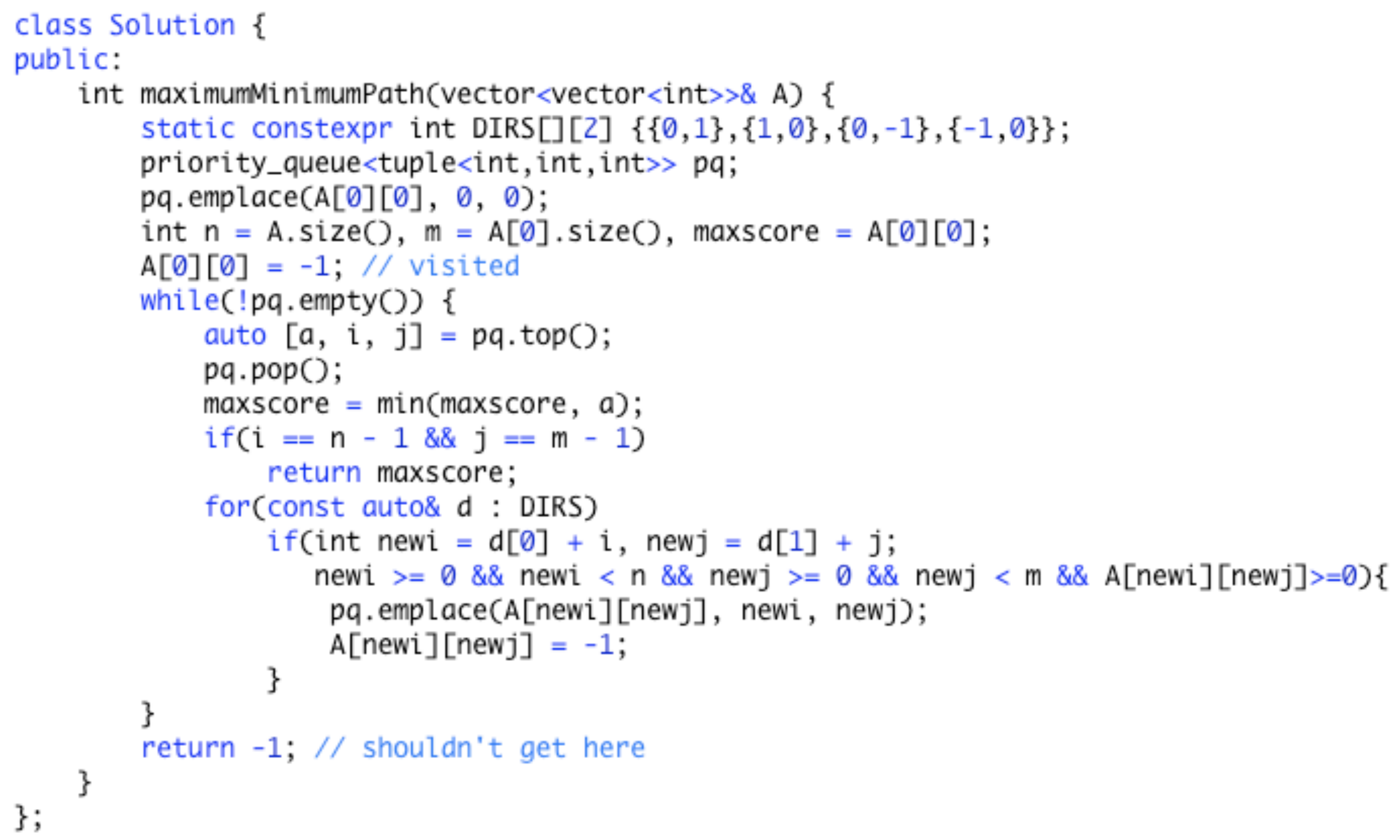
Example 1:



5	4	5
1	2	6
7	4	6

```
Input: [[5,4,5],[1,2,6],[7,4,6]]
Output: 4
Explanation:
The path with the maximum score is highlighted in yellow.
```

6.PATH WITH MAXIMUM MINIMUM VALUE (DIJKSTRA'S ALGORITHM)



```
class Solution {
public:
    int maximumMinimumPath(vector<vector<int>>& A) {
        static constexpr int DIRS[][2] {{0,1},{1,0},{0,-1},{-1,0}};
        priority_queue<tuple<int,int,int>> pq;
        pq.emplace(A[0][0], 0, 0);
        int n = A.size(), m = A[0].size(), maxscore = A[0][0];
        A[0][0] = -1; // visited
        while(!pq.empty()) {
            auto [a, i, j] = pq.top();
            pq.pop();
            maxscore = min(maxscore, a);
            if(i == n - 1 && j == m - 1)
                return maxscore;
            for(const auto& d : DIRS)
                if(int newi = d[0] + i, newj = d[1] + j;
                   newi >= 0 && newi < n && newj >= 0 && newj < m && A[newi][newj]>=0){
                    pq.emplace(A[newi][newj], newi, newj);
                    A[newi][newj] = -1;
                }
        }
        return -1; // shouldn't get here
    }
};
```

HOMEWORK

- 1042. Flower Planting With No Adjacent
- 133. Clone Graph
- 332. Reconstruct Itinerary
- 210. Course Schedule II
- 269. Alien Dictionary
- 1136. Parallel Courses

Thank you!