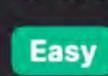
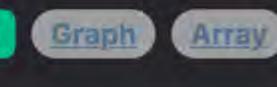
Hash Table





Problem Description

In this problem, we are given a scenario where there is a small town with n people indexed from 1 to n. Among them, there is a rumor of one person being the town judge. The town judge is defined by two distinct characteristics:

2. Every other person in the town trusts the judge, that means all individuals aside from the judge themselves trust this judge.

1. The town judge trusts no one, meaning they do not express trust towards any other person in the town.

- 3. There is only one individual who satisfies both of the above conditions, making them the town judge.
- The problem provides us with trust data in the form of an array, where each subarray trust[i] = [a_i, b_i] represents that person

a_i trusts person b_i. This array represents all the trust relationships that exist in the town. If no trust relationship exists between two individuals, it will not be represented in this array. The goal is to find out who the town judge is, if one exists, based on these trust relationships. The output should be the label of the

town judge. If no town judge can be identified, based on the rules described, the output should be -1. Intuition

The solution approach follows from the given properties of the town judge.

1. Since every person except for the town judge trusts the judge, the town judge will be the one who is trusted by n-1 others

- (everybody else). 2. Since the town judge trusts no one, the judge will not show up as the first element in any trust pair [a_i, b_i].
- cnt1[i]: Count of people that person i trusts. This will be increased when person i is the a_i in a trust pair.

• cnt2[i]: Count of people who trust person i. This will be increased when person i is the b_i in a trust pair.

To find the town judge, we can keep two counts for each person:

- As we iterate through the array, we populate these two count arrays. Then we look for a person i for whom cnt1[i] is 0 (trusts no
- one) and cnt2[i] is n-1 (is trusted by everyone else). This person is the town judge.

Solution Approach

The implementation of the solution uses a simple counting algorithm, which is based on the properties that define who the town

The judge trusts no one, so they will not appear as the first person in any of the trust pairs.

 Everyone trusts the judge, so the judge will be the second person in n-1 trust pairs if there are n people in the town. The solution uses two list data structures, cnt1 and cnt2. Both lists, indexed from 1 to n, are initialized to 0. The lists are used in the

- cnt1[i] represents the number of people that person i trusts. cnt2[i] represents the number of people who trust person i.
- For each trust pair [a, b] found in the trust list, we increase cnt1[a] by 1 because person a is shown to trust another person. Simultaneously, we increase cnt2[b] by 1 because it shows that person b is trusted by someone (person a in this case).

When the iteration of trust pairs is complete, we need to find the person who meets the criteria for being the town judge:

Person i satisfies the condition cnt1[i] == 0, indicating they do not trust anyone else, and

If such a person exists, their index i is returned as the town judge. If no one person satisfies both conditions, the function returns -1,

• Person i satisfies the condition cnt2[i] == n - 1, indicating that they are trusted by everyone else in town.

signaling there is no identifiable town judge.

given by the array trust = [[1, 3], [2, 3]]. Here's how the algorithm finds the town judge:

Example Walkthrough Let's use a small example to illustrate the solution approach. Suppose the town has n = 3 people, and the trust relationships are

1. We initialize two count arrays, cnt1 and cnt2, with size equal to the number of people in the town (3 in this case), filled with

zeroes. cnt1 tracks the number of people a person trusts, while cnt2 tracks the number of people that trust a person.

3. Next, we process [2, 3]:

Using the template described:

2. We iterate over each trust relationship. First, we look at [1, 3]:

 Increment cnt2[3] by 1 since person 3 is trusted by person 1. After this step: cnt1 = [1, 0, 0] cnt2 = [0, 0, 1]

2 Imagine a town with 3 people. We are given the trust array `trust = [[1, 3], [2, 3]]`, which tells us person 1 trusts person 3, and p

- Increment cnt1[2] by 1 since person 2 trusts person 3. Increment cnt2[3] by 1 since person 3 is trusted by person 2.
- 4. Now, with the counts updated, we look for the person who is trusted by n-1 others and trusts no one: Person 3 satisfies cnt1[3] == 0 (trusts nobody) and cnt2[3] == 2 (is trusted by others).
 - 1. We start by initializing two arrays to track the trust counts, `cnt1` and `cnt2`. Initially, no one trusts anyone, so both arrays 2. For the first pair `[1, 3]`, person 1 trusts person 3. We increment `cnt1[1]` because person 1 is showing trust, and `cnt2[3]` because person 1 is shown and the contact person 1 is shown and the

12 Therefore, the algorithm identifies person 3 as the town judge in this example.

```
class Solution:
       def findJudge(self, n: int, trust: List[List[int]]) -> int:
           # Initialize two lists to count the trust votes.
           # trust_received[i] will hold the number of people who trust person i.
6
8
9
10
11
12
```

trust_given[i] will hold the number of people person i trusts. $trust_received = [0] * (n + 1)$ $trust_given = [0] * (n + 1)$ # Iterate through each trust relationship in the trust list. for giver, receiver in trust: # Increment the trust count: giver trusts another person, # and the receiver is trusted by another person. trust_given[giver] += 1 trust_received[receiver] += 1 # Iterate over each person to find the potential town judge. for person in range(1, n + 1): # The town judge should not trust anyone (trust_given[person] == 0) # and should be trusted by everyone else (trust_received[person] == n - 1). if trust_given[person] == 0 and trust_received[person] == n - 1:

int personTrusts = relation[0]; // Person that trusts someone int personTrusted = relation[1]; // Person that is trusted 13 14

```
Java Solution
   class Solution {
       public int findJudge(int n, int[][] trust) {
           // Array to keep track of the number of people each person trusts
           int[] trustCount = new int[n + 1];
           // Array to keep track of the number of people who trust each person
           int[] trustedByCount = new int[n + 1];
           // Iterate over the trust relationships
           for (int[] relation : trust) {
               // Increment the trust count for the person who trusts someone
15
               trustCount[personTrusts]++;
```

// Increment the count of being trusted for the person who is trusted

// and the person who is trusted by everyone else (trustedByCount is n-1)

return i; // The index represents the person, so return it as the judge's identity

// The judge is the person who trusts nobody (trustCount is 0)

if (trustCount[i] == 0 && trustedByCount[i] == n - 1) {

return person # Return the town judge.

If no town judge is found, return -1.

trustedByCount[personTrusted]++;

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

// If no judge is found, return -1

return -1;

// Iterate over all the people to find the judge

return -1

```
34 }
35
C++ Solution
1 class Solution {
2 public:
       int findJudge(int N, vector<vector<int>>& trust) {
           // Create two vectors to store trust counts
           vector<int> trustByOthers(N + 1, 0); // Counts the number of people who trust 'i'
           vector<int> trustsOthers(N + 1, 0); // Counts the number of people 'i' trusts
           // Calculate trust counts
           for (const auto& relation : trust) {
10
               int truster = relation[0]; // Person who trusts
               int trustee = relation[1]; // Person being trusted
11
12
               trustsOthers[truster]++; // Increment trust count for the truster
13
               trustByOthers[trustee]++; // Increment count of being trusted for the trustee
14
15
           // Find the town judge
16
17
           for (int i = 1; i \le N; ++i) {
               // The judge is the person who trusts no one else (trustsOthers[i] == 0)
18
               // and is trusted by everyone else (trustByOthers[i] == N - 1)
19
               if (trustsOthers[i] == 0 && trustByOthers[i] == N - 1) {
20
21
                   return i; // Judge found
23
24
25
           // If no judge found, return -1
26
           return -1;
27
28 };
29
```

// Process trust relationships. for (const [truster, trusted] of trust) { trustCounts[truster]++; trustedByCount[trusted]++; 18

Typescript Solution

1 // Function to find the judge in a town.

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

return -1; // No judge found

return i; // Judge found

2 // In a town of n people, a judge is known by everyone but knows no one.

// Array to track the number of people that trust each person.

// Find the judge who is trusted by everyone but trusts nobody.

if (trustedByCount[i] === n - 1 && trustCounts[i] === 0) {

const trustCounts: number[] = new Array(n + 1).fill(0);

// Array to track the number of people each person trusts.

const trustedByCount: number[] = new Array(n + 1).fill(0);

Time and Space Complexity

The time complexity of the code can be analyzed as follows:

length of the trust array. This loop has a time complexity of O(m).

• A second loop is run to go through each person from 1 to n, which does constant work for each person. Hence, the time complexity is O(n) for this loop.

• There is one loop executed over the trust array, which in the worst-case scenario will go through m relationships, where m is the

Since these are two separate loops, we add the time complexities resulting in a total time complexity of 0(m + n).

The extra space used by the code includes two arrays cnt1 and cnt2, each of size n + 1 to keep counts of trust received and given. Therefore, the space complexity of the code is 0(n + n), which simplifies to 0(n), as generally the constants are not considered in

Space Complexity

In summary: Time Complexity: 0(m + n)

the Big O notation, and we take the highest order term.

Space Complexity: 0(n)

- If there isn't anyone who meets these conditions, we return -1 to indicate that there's no identifiable town judge based on the given trust relationships.
- judge could be:
- following way: As we iterate through the trust list, we update these counts:
- The algorithm's time complexity is O(T + n), where T is the number of trust relationships (the length of the trust list) and n is the number of people in the town. The space complexity is O(n) for the two count lists used to store the trust counts for each person.
- At the start: cnt1 = [0, 0, 0] cnt2 = [0, 0, 0]Increment cnt1[1] by 1 since person 1 trusts person 3.
- After this step: cnt1 = [1, 1, 0] cnt2 = [0, 0, 2]5. Finally, we conclude that person 3 is the town judge, as they meet both conditions for being the judge.
- 8 3. Next, we process the second pair `[2, 3]`. Similarly, we increment `cnt1[2]` and `cnt2[3]`, reflecting that person 2 trusts person 10 4. After analyzing the trust pairs, we see that person 3 is trusted by both other people (`cnt2[3] = 2`), and person 3 trusts nobody 11 Python Solution
- 13 14 15 16
- 17 18 19 20 21 23 24
- Note: List needs to be imported from typing if not done already in the code. To adhere to Python coding standards, the import statement would look like this: from typing import List

24

25

26

27

28

29

30

31

32

33

3 // Parameters: 4 // n: number - The number of people in the town. 5 // trust: number[][] - Array of pairwise trusts; trust[i] = [a, b] represents person a trusts person b. 6 // Returns: number - The label of the judge; returns -1 if no judge is found. 7 function findJudge(n: number, trust: number[][]): number { 10 11 12 13 14 16 17

19

20

21

22

23

24

25

26

29

Time Complexity