Topic Introduction

Today, we're diving into a trio of graph relationship and property problems. At first glance, these seem like classic "find the odd person out" puzzles, but they're all about understanding connections between nodes (people, townsfolk, etc.) in a special kind of graph.

Core Concept: Graph Relationship Patterns

A *graph* is a collection of nodes (also called vertices) and edges (connections between nodes). In many problems, the nodes represent people or objects, and the edges show how they're related (like "knows", "trusts", or "is connected to").

In coding interviews, graph problems often ask you to:

- Find nodes with unusual in/out degree patterns (e.g., someone trusted by everyone, but who trusts no one).
- Identify special roles (like a judge or celebrity).
- Check if the graph has certain properties (like being bipartite).

How does this work?

- **In-degree**: Number of edges coming *into* a node.
- **Out-degree**: Number of edges going *out* of a node.

For example, if Alice trusts Bob, that's an outgoing edge from Alice and an incoming edge to Bob.

When is this useful in interviews?

Many real-world problems can be modeled as graphs: social networks, tasks with dependencies, city maps, etc. Recognizing in/out degree patterns and properties like bipartiteness is a key skill for quickly narrowing down solutions.

Simple Example (not from the target problems):

Suppose you have three people and a list of who likes whom. Who is the "most liked" person?

If you draw this as a graph, the person with the highest in-degree is the answer.

Why are these 3 problems grouped together?

All three focus on analyzing relationships in a graph to find a node with a special property:

- **Town Judge**: Find the person trusted by everyone, but who trusts no one.
- **Celebrity**: Find the person known by everyone, but who knows no one.
- **Is Graph Bipartite**: Check if you can split the graph into two groups with no internal edges (a coloring property).

Let's explore each!

Problem 1: Find the Town Judge

LeetCode Link

Problem Restated

You are given `n` people labeled 1 to n in a town. Some people trust others, represented as pairs `[a, b]` meaning person `a` trusts person `b`.

Goal: Find the town judge, who:

- Is trusted by everyone else (n-1 people).
- Trusts no one.

If no such person exists, return -1.

Example

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

Output:
3

Explanation:
- Person 3 is trusted by 1 and 2 (so 2 incoming).
- Person 3 trusts no one.
- So, 3 is the judge.

How to Solve

Try it with pen and paper for a small example! Write out who trusts whom. Look for someone trusted by all others, but who doesn't trust anyone.

Additional Test Case

n = 4
trust = [[1,3],[2,3],[3,1],[4,3]]

Who is the judge? Try on your own!

Brute Force Approach

Check every person:

- For each person, count how many trust them.
- Also check if they trust anyone.

This is O(n^2) time.

Optimal Approach

Pattern: Track trust counts for each person.

- For each trust pair [a, b]:
    - a trusts someone, so a cannot be judge.
    - b is trusted by someone, so increment their trust count.

Key Idea:
Maintain an array where each index represents a person. For each trust pair:

- Decrement the trust count for the truster (since they trust someone).
- Increment the trust count for the trustee (since they are trusted).

At the end, the judge will have a trust count of n-1.

Step-by-Step Logic

- Create a trust score array of size n+1 (since people are labeled 1...n).
- For each trust [a, b]:
    - Decrement trust[a] by 1 (since a trusts someone).
    - Increment trust[b] by 1 (since b is trusted).
- Loop through 1 to n:
    - If trust[i] == n-1, that's your judge.

Python Solution

```python
def findJudge(n, trust):
    # Step 1: Initialize trust counts for each person
    trust_score = [0] * (n + 1)  # index 0 is unused

    # Step 2: Update trust scores
    for a, b in trust:
        trust_score[a] -= 1  # a trusts someone
        trust_score[b] += 1  # b is trusted

    # Step 3: Find the judge
    for person in range(1, n + 1):
        if trust_score[person] == n - 1:
            return person
    return -1
```

Time Complexity: O(n + t), where t is the number of trust pairs.

Space Complexity: O(n) for the trust_score array.

Code Explanation

- `trust_score`: Tracks for each person, their net trust status.
    - +1 for each person who trusts them.
    - -1 for each person they trust (since the judge trusts no one).
- If someone is trusted by everyone else and trusts no one, their score will be n-1.
- Go through each person to check.

Walkthrough Example

n = 3, trust = [[1,3],[2,3]]

- trust_score = [0, 0, 0, 0]
- [1,3]: trust_score[1] -= 1 (now -1), trust_score[3] += 1 (now 1)
- [2,3]: trust_score[2] -= 1 (now -1), trust_score[3] += 1 (now 2)
- trust_score = [0, -1, -1, 2]
- Check person 1: -1 != 2

person 2: -1 != 2

person 3: 2 == 2 → return 3

Try This Test Case

n = 4

trust = [[1,3],[2,3],[3,1],[4,3]]

What is the trust_score array? Who is the judge?

Now, take a moment to solve this yourself before checking the solution!

Problem 2: Find the Celebrity

LeetCode Link

Problem Restated

You are at a party with n people (labeled 0 to n-1). A celebrity is defined as someone who:

- Is known by everyone else.
- Knows no one.

You can only ask if person A knows person B via a function `knows(a, b)`.

Goal: Find the celebrity, or return -1 if there isn't one.

Example

n = 3

Let's say:

- 0 knows 1 and 2
- 1 knows 2
- 2 knows no one

In this case, person 2 is known by everyone, but knows no one. So, 2 is the celebrity.

Additional Test Case

n = 3, where:

- 0 knows 1
- 1 knows 0
- 2 knows 0 and 1

Who is the celebrity? Try to dry-run it!

How is this similar to Town Judge?

- Both look for a node with in-degree n-1, out-degree 0.
- Both can be solved by analyzing in-out relationships.
- But here, the only way to get info is by calling `knows(a, b)`.

Brute Force

For each person, check if they know anyone (shouldn't know anyone) and if everyone knows them.

O(n^2) calls to `knows`.

Optimal Approach

Pattern Used: *Candidate Elimination via Simulation*

- If A knows B, then A can't be the celebrity.
- If A doesn't know B, then B can't be the celebrity.
- You can eliminate one non-celebrity per call!

Step-by-Step Logic

- Set candidate to 0.
- For i = 1 to n-1:
    - If candidate knows i, candidate can't be celebrity. Set candidate = i.
- After this loop, candidate is the only possible celebrity.
- Verify: For all others, does everyone know candidate, and candidate knows no one?

Pseudocode

```
candidate = 0
for i from 1 to n-1:
    if knows(candidate, i):
        candidate = i

# Check if candidate is actual celebrity
for i from 0 to n-1:
    if i == candidate:
        continue
    if knows(candidate, i) or not knows(i, candidate):
        return -1
return candidate
```

Example Trace

Suppose n = 3:

- candidate = 0
- knows(0,1)? Yes -> candidate = 1
- knows(1,2)? Yes -> candidate = 2

Now check:

- does 0 know 2? yes
- does 1 know 2? yes
- does 2 know anyone? no
- does everyone know 2? yes
- So 2 is the celebrity.

Try This Test Case

n = 3,
knows(0,1): True
knows(1,0): True
knows(2,0): True
knows(2,1): True
Everyone knows someone else. Is there a celebrity?

Time Complexity: O(n) calls to `knows` to find candidate, O(n) to verify — total O(n).

Space Complexity: O(1).

Problem 3: Is Graph Bipartite?

LeetCode Link

Problem Restated

Given an undirected graph as an adjacency list, can you split the nodes into two groups so that no two adjacent nodes are in the same group?

This is equivalent to asking: Can you 2-color the graph so that no edge connects nodes of the same color?

Example

Input:
graph = [[1,3],[0,2],[1,3],[0,2]]

Here, node 0 is connected to 1 and 3, 1 to 0 and 2, etc.

Output: True (it is bipartite)

Additional Test Case

graph = [[1,2,3],[0,2],[0,1,3],[0,2]]

Is this bipartite?

How is this similar?

- This is about properties of the graph, not just individual nodes.
- Still, we analyze relationships — does the pattern hold for all nodes and their neighbors?

- Instead of in/out degree, we're assigning colors/groups.

Brute Force

Try all possible 2-colorings. Exponential time.

Optimal Approach

Pattern: Use BFS or DFS to try coloring the graph. If you ever find an edge connecting two nodes of the same color, it's not bipartite.

Step-by-Step Logic

- Assign colors (say, 0 and 1) as you traverse.
- For each unvisited node, assign a color and traverse its neighbors:
    - If a neighbor has the same color, return False.
    - If uncolored, assign the opposite color and continue.
- If you finish without conflict, return True.

Pseudocode

```
Initialize color array with -1 for all nodes

for each node in graph:
    if color[node] == -1:
        color[node] = 0
        queue = [node]
        while queue is not empty:
            current = queue.pop()
            for neighbor in graph[current]:
                if color[neighbor] == -1:
                    color[neighbor] = 1 - color[current]
                    queue.append(neighbor)
                elif color[neighbor] == color[current]:
                    return False
return True
```

Example Trace

graph = [[1,3],[0,2],[1,3],[0,2]]

- Start at node 0, color 0.
- Node 1 (neighbor): color 1.
- Node 3 (neighbor): color 1.
- Node 1's neighbor: 2 (color 0).
- Continue; no conflicts found.

Try This Test Case

graph = [[1],[0,3,4],[4],[1],[1,2]]

Is this bipartite? (Try coloring by hand!)

Time Complexity: O(V + E), V = nodes, E = edges.
Space Complexity: O(V) for color array and queue.

Summary and Next Steps

These problems are all about understanding how nodes relate to each other in a graph:

- **Town Judge** and **Celebrity**: Find a special node with a unique in/out degree pattern (trusted/known by all, trusts/knows no one).
- **Is Graph Bipartite**: Can you split the graph into two groups where every edge connects different groups?

Key Patterns:

- Use in-degree and out-degree to identify special nodes.
- Candidate elimination: Efficiently narrow down possibilities (as in Celebrity).
- BFS/DFS coloring: Check for graph properties (as in Bipartite).

Common Traps:

- Forgetting to check both conditions (in-degree and out-degree).
- Not verifying the candidate in the Celebrity problem.
- Missing disconnected components in Bipartite check.

Action List

- Solve all three problems on your own — even the one with code above.
- For Problem 2 and 3, try both BFS and DFS approaches.
- Explore similar problems: "Course Schedule" (topological sort), "Number of Connected Components."
- After solving, compare your code with community solutions for new insights.
- If you get stuck, step back and draw the graph or relationships by hand. Practice helps!

Keep at it! These graph relationship and property skills are powerful tools in your interview toolkit.