Guidelines on Using External Resources

Our goal is to see how you solve problems in a realistic setting. You are encouraged to use the tools you would normally use on the job, such as online documentation and AI assistants.

- **Online Documentation (Google, Stack Overflow, etc.):** You are free and expected to use online resources to look up syntax, library documentation, and standard algorithms.
- **AI Assistants (ChatGPT, Cursor, etc.):** These tools are also permitted, with one key principle: **you must own your solution.**
    - Feel free to use AI as an assistant for debugging, brainstorming, or generating boilerplate code. Do not use it to generate the entire solution.
    - You must be able to explain the logic and design choices behind every line of code you submit.
    - Please include a brief note in your `README.md` acknowledging how you used AI tools, if at all.

Ultimately, we are evaluating your problem-solving approach, code quality, testing practices, and the clarity of your documentation—not just the final answer. Good luck!

Mercor Challenge: Referral Network

This coding challenge is designed to assess your engineering and problem-solving skills in a context relevant to our work at Mercor. You will build the core logic for a referral network, progressing from data structure design to algorithmic analysis and simulation.

The project is divided into five parts that build upon each other. Please read all instructions carefully before beginning.

Part 1: Referral Graph

Your first task is to design and build the core data structure for our referral network. At this stage, focus only on direct, one-to-one referral relationships.

Required Functionality

Design and implement a class or equivalent data structure that provides the ability to:

- Store and manage users and their referral relationships.
- Add a new, directed referral link from a `referrer` to a `candidate`.
- Query for a user's **direct referrals** (the immediate users they have referred).

Implementation Constraints

Your implementation must enforce the following rules:

- **No Self-Referrals:** A user cannot refer themselves.
- **Unique Referrer:** A candidate can only be referred by one user.
- **Acyclic Graph:** The graph must remain acyclic. Any operation that would create a cycle must be rejected.

Part 2: Full Network Reach

In Part 1, you focused on direct connections. Now, you will extend the analysis to consider a user's full influence by including their **indirect referrals** (referrals of referrals, and so on). This complete downstream network is defined as a user's **reach**.

Required Functionality

Extend your implementation to support the following calculations:

- **Total Referral Count:** Add a method or function to calculate the total number of referrals for any given user, including both direct and all indirect referrals. A Breadth-First Search (BFS) is a standard approach for this kind of traversal.
- **Top Referrers by Reach:** Implement a method or function to return a ranked list of the top k referrers based on their total referral count. In your code comments or README, explain how the caller should pick an appropriate k.

Part 3: Identify Influencers

Raw reach is a simplistic metric for identifying important users. This section introduces more sophisticated methods for finding true influencers within the network.

Metric 1: Unique Reach Expansion

- **Goal:** To identify a set of referrers who, together, cover the maximum number of **unique** candidates, thereby minimizing audience overlap.
- **Algorithm:** Implement a greedy algorithm. First, pre-compute the full downstream reach set for every user. Then, iteratively select the user who adds the largest number of *new, not-yet-covered* candidates to a global set of reached people.

Metric 2: Flow Centrality

- **Goal:** To identify the network's most critical "brokers"—the users who connect different parts of the network and whose removal would be most likely to cause it to fragment.
- **Algorithm Intuition:** Think of your network as a map; this metric finds the most critical intersections. A user is considered a broker if they lie on the most shortest paths between other users.
- **Implementation Guide:** A key property is that a user `v` lies on a shortest path between a source `s` and a target `t` if `dist(s, v) + dist(v, t) == dist(s, t)`. A straightforward way to implement this is to first pre-compute the shortest distance between all pairs of users (e.g., by running a BFS from every node). Then, you can iterate through every combination of `(s, t, v)` and increment a score for `v` each time the condition holds true.

Requirements

- **Implementation:** Implement functions or methods that return ranked lists of users based on these two metrics. For Flow Centrality, a clear, correct implementation is more important than a highly optimized one.
- **Comparison:** In your code comments or README, write a concise comparison of the three influence metrics (`reach`, `unique_reach`, `flow_centrality`). For each, describe a specific business scenario where it would be the preferred metric.

Part 4: Network Growth Simulation

This section moves from static graph analysis to modeling how the network might grow over time.

Model Parameters

- **Initial Referrers:** The simulation starts with **100** active referrers.
- **Referral Capacity:** Each referrer has a lifetime capacity to make up to **10** successful referrals. After their capacity is exhausted, they become inactive.
- **Time Unit:** The simulation proceeds in discrete steps called **days**.

Required Functions

Implement the following functions:

- A function `simulate` that accepts a floating-point probability `p` and an integer `days`. It must return an array or list of numbers, where the element at index `i` is the **cumulative total expected referrals** at the end of day `i`.
- A function `days_to_target` that accepts a probability `p` and an integer `target_total`. It should efficiently calculate the **minimum number of days** required for the cumulative expected referrals to meet or exceed the target.

Part 5: Referral Bonus Optimization

The final part uses the simulation logic to solve a business problem: finding the minimum bonus amount required to achieve a hiring target.

Model Details

- A milestone bonus, offered in increments of $10, is paid when a referral passes a basic screening.
- The bonus amount influences participation. This is modeled by a function `adoption_prob(bonus)`, which returns the probability `p` that an active user will successfully refer someone on any given day.
- You can assume this `adoption_prob` function is a "black box" that is **monotonically increasing** (a higher bonus never results in lower participation).

Required Function

Efficiently implement a function `min_bonus_for_target`. This function should:

- **Accept:** an integer `days`, an integer `target_hires`, the `adoption_prob` function, and a small

floating-point number eps (e.g., 1e-3) for precision.

- **Return:** The minimum bonus (rounded UP to the nearest $10) required to achieve the target. If the target is unachievable with any finite bonus, it should return a null or equivalent empty value.
- **Use:** The simulation logic from Part 4. The initial referrers (100) and referral capacity (10) are fixed.

In the code comments or README, explain the time complexity of your solution.

Submission Guidelines

Please follow these instructions carefully. Submissions that do not adhere to this structure may be considered incomplete.

Submission Method

Your work must be submitted as a link to a **private GitHub repository**.

Please grant repository access to the following GitHub user: `yukuairoy@gmail.com`

Repository Structure

We expect your repository to be organized in a professional manner. The following is an *example* structure—please adapt it to the conventions of your chosen language and build system.

```
mercor-challenge/
├── README.md
├── .gitignore
├── source/
│   ├── ReferralNetwork.{js,java,go,ts,etc.}
│   └── Simulation.{js,java,go,ts,etc.}
└── tests/
    └── ... (language-specific test files)
```

Content Requirements

1. `README.md`

This is the entry point for your project. It must contain the following sections:

- **Language & Setup:** Specify the language, version, and any required compilers or runtimes. Provide a simple list of commands to install dependencies and build the project.
- **Running Tests:** A single, clear command to execute your entire test suite.
- **Design Choices:** A section where you provide the written justifications required by the prompt, particularly for your data structure and API design choices in Part 1.

2. Source Code

Organize your source code into logical modules or files within a `source` or `src` directory.

3. Tests

A comprehensive test suite is required. **Submissions without tests will not be considered complete.**

- Use a standard testing framework for your language (e.g., Jest for JavaScript, JUnit for Java, Go's built-in testing package).
- Your tests must validate the correctness of your implementation, including core functionality, all constraints from Part 1, and relevant edge cases.

4. Dependency Management

Include a file that declares your project's dependencies, such as a `package.json`, `pom.xml`, `go.mod`, or `requirements.txt` file.

Final Checklist

Before you submit, please verify the following:

- [ ] The GitHub repository is private and access has been granted.
- [ ] The `README.md` file is complete and includes all required sections.
- [ ] All project dependencies are declared in a manifest file.
- [ ] Your test suite is comprehensive and runs with a single command.