TIP102 | Intermediate Technical Interview Prep

Intermediate Technical Interview Prep Spring 2025 (@ Section 3 | Tuesdays and Thursdays 6PM - 8PM PT)
Personal Member ID#: **117667**

Session 1: Graphs

Session Overview

In this session, students will learn about how to build and manipulate different representations of graphs, including lists of edges, adjacency lists, and adjacency matrices. Students will also be introduced to the two primary traversal algorithms for graphs: Breadth First Search and Depth First Search.

> You can find all resources from today including session slide decks, session recordings, and more on the resources tab

🎢 Part 1 : Instructor Led Session

We'll spend the first portion of the synchronous class time in large groups, where the instructor will lead class instruction for 30-45 minutes.

👨‍💻 Part 2: Breakout Session

In breakout sessions, we will explore and collaboratively solve problem sets in small groups. Here, the **collaboration, conversation, and approach** are just as important as "solving the problem" - please engage warmly, clearly, and plentifully in the process!

In breakout rooms you will:

- Screen-share the problem/s, and verbally review them together
- Screen-share an interactive coding environment, and talk through the steps of a solution approach
    - ProTip: - An Integrated Development Environment (IDE) is a fancy name for a tool you could use for shared writing of code - like Replit.com, Collabed.it, CodePen.io, or other - your staff team will specify which tool to use for this class!
- Screen-share an implementation of your proposed solution
- Independently follow-along, or create an implementation, in your own IDE.

Your program leader/s will indicate which code sharing tool/s to use as a group, and will help break down or provide specific scaffolding with the main concepts above.

▶ **Note on Expectations**

🔎 Problem Solving Approach

To build a long-term organized approach to problem solving, we'll start with three main steps. We'll refer to them as **UPI: Understand, Plan, and Implement**.

We'll apply these three steps to most of the problems we'll see in the first half of the course.

We will learn to:

- **Understand** the problem,
- **Plan** a solution step-by-step, and
- **Implement** the solution

▶ **Comment on UPI**

▶ **UPI Example**

Breakout Problems Session 1

▶ Standard Problem Set Version 1

▶ Standard Problem Set Version 2

▼ Advanced Problem Set Version 1

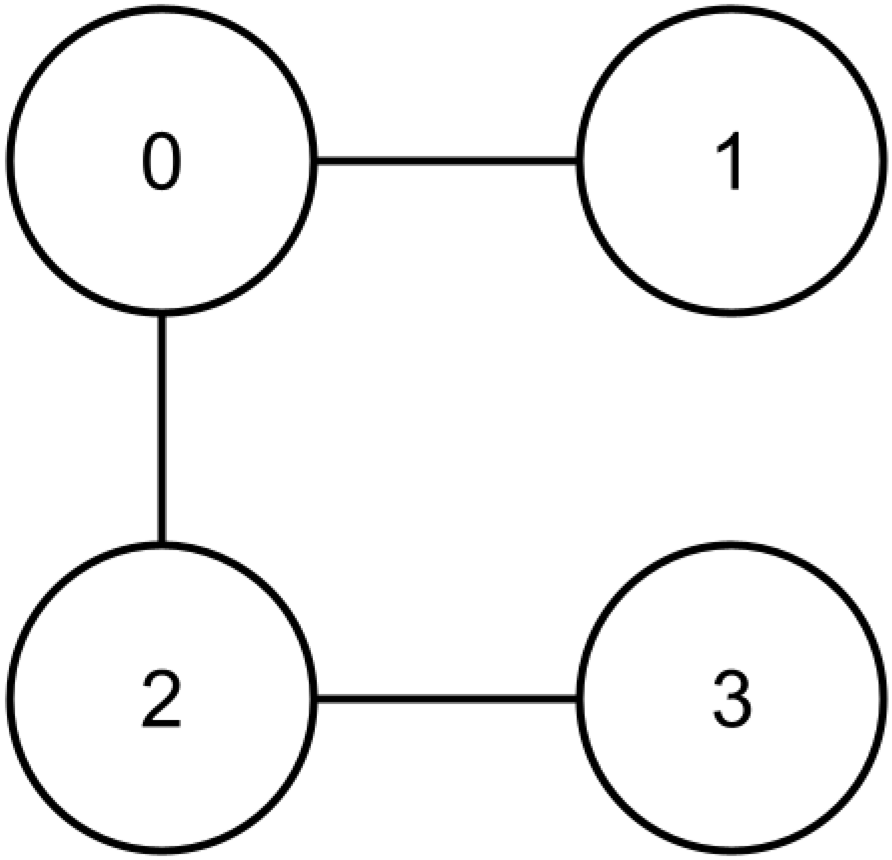
Problem 1: There and Back

As a flight coordinator for CodePath airlines, you have a 0-indexed adjacency list `flights` with `n` nodes where each node represents the ID of a different destination and `flights[i]` is an integer array indicating that there is a flight from destination `i` to each destination in `flights[i]`. Write a function `bidirectional_flights()` that returns `True` if for any flight from a destination `i` to a destination `j` there also exists a flight from destination `j` to destination `i`. Return `False` otherwise.

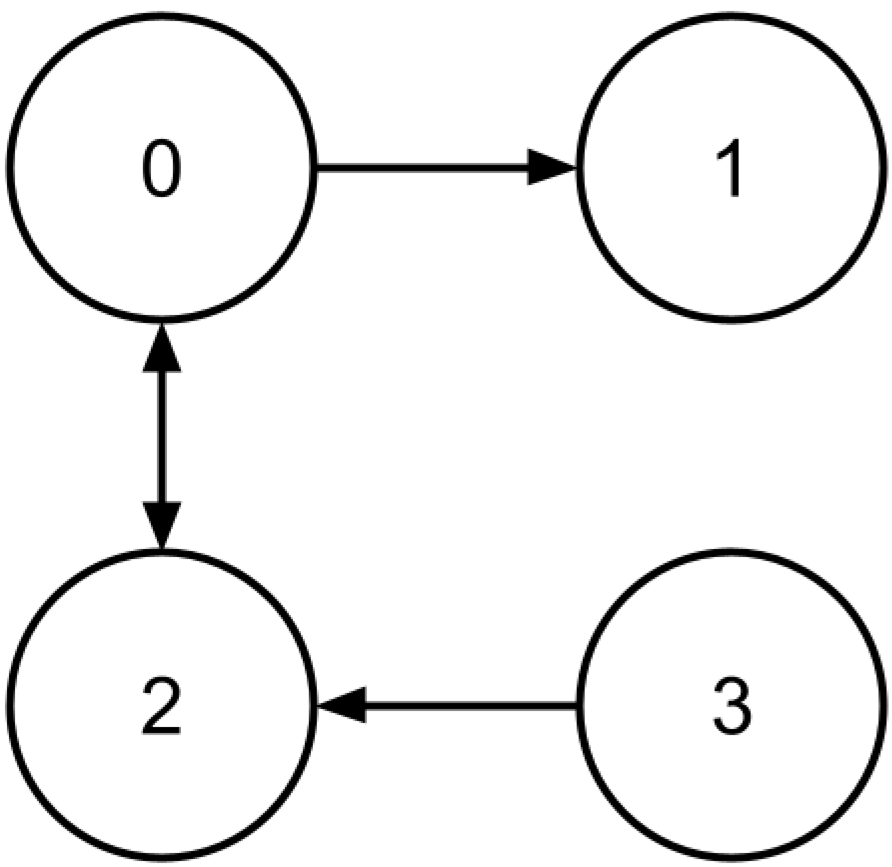
```
def bidirectional_flights(flights):
    pass
```

Example Usage:

Example 1: `flights1`



Example 2: `flights2`



```python
flights1 = [[1, 2], [0], [0, 3], [2]]
flights2 = [[1, 2], [], [0], [2]]

print(bidirectional_flights(flights1))
print(bidirectional_flights(flights2))
```

Example Output:

```
True
False
```

▶ 💡 **Hint: Introduction to Graphs**

Problem 2: Find Center of Airport

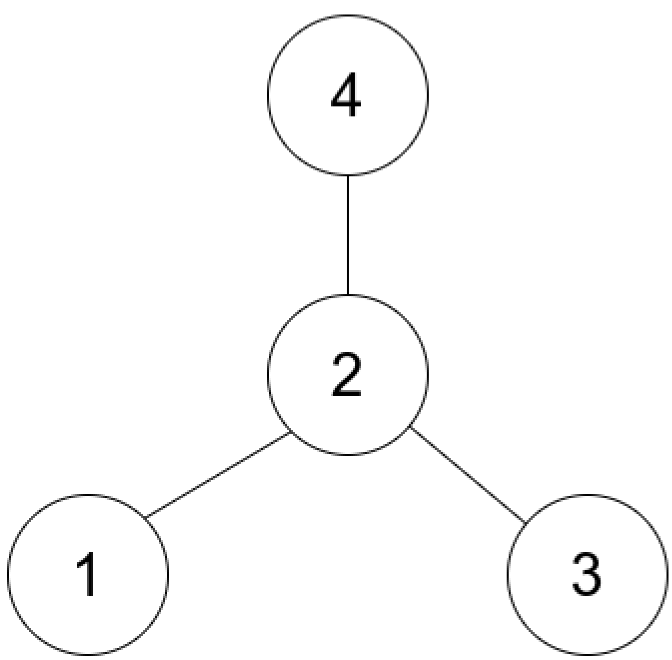
You are a pilot navigating a new airport and have a map of the airport represented as an undirected star graph with `n` nodes where each node represents a terminal in the airport labeled from `1` to `n`. You want to find the center terminal in the airport where the pilots' lounge is located.

Given a 2D integer array `terminals` where each `terminal[i] = [u, v]` indicates that there is a path (edge) between terminal `u` and `v`, return the center of the given airport.

A star graph is a graph where there is one center node and exactly `n-1` edges connecting the center node ot every other node.

```python
def find_center(terminals):
    pass
```

Example Usage:



```
terminals1 = [[1,2],[2,3],[4,2]]
terminals2 = [[1,2],[5,1],[1,3],[1,4]]

print(find_center(terminals1))
print(find_center(terminals2))
```

Example Output:

```
2
1
```

▶ 💡 **Hint: Star Graph Properties**

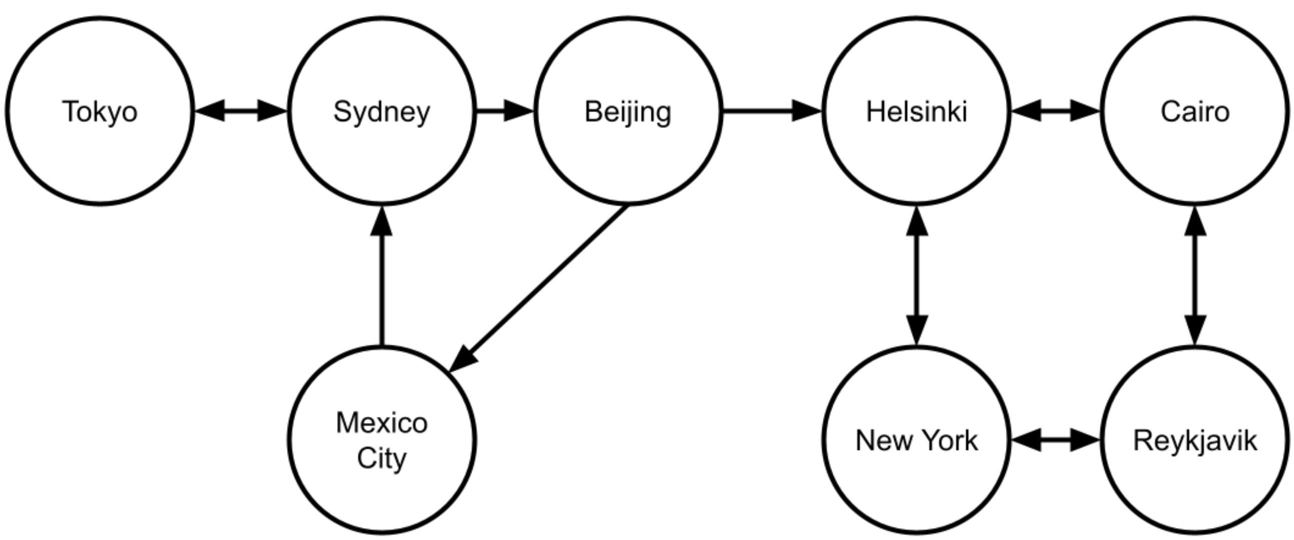
Problem 3: Finding All Reachable Destinations

You are a travel coordinator for CodePath Airlines, and you're helping a customer find all possible destinations they can reach from a starting airport. The flight connections between airports are represented as an adjacency dictionary `flights`, where each key is a destination, and the corresponding value is a list of other destinations that are reachable through a direct flight.

Given a starting location `start`, return a list of all destinations reachable from the `start` location either through a direct flight or connecting flights with layovers. The list should be provided in ascending order by number of layovers required.

```
def get_all_destinations(flights):
    pass
```

Example Usage:



```
flights = {
    "Tokyo": ["Sydney"],
    "Sydney": ["Tokyo", "Beijing"],
    "Beijing": ["Mexico City", "Helsinki"],
    "Helsinki": ["Cairo", "New York"],
    "Cairo": ["Helsinki", "Reykjavik"],
    "Reykjavik": ["Cairo", "New York"],
    "Mexico City": ["Sydney"]
}

print(get_all_destinations(flights, "Beijing"))
print(get_all_destinations(flights, "Helsinki"))
```

Example Output:

```
['Beijing', 'Mexico City', 'Helsinki', 'Sydney', 'Cairo', 'New York', 'Tokyo',
'Reykjavik']
['Helsinki', 'Cairo', 'New York', 'Reykjavik']
```

▶ 💡 **Hint: Breadth First Search**

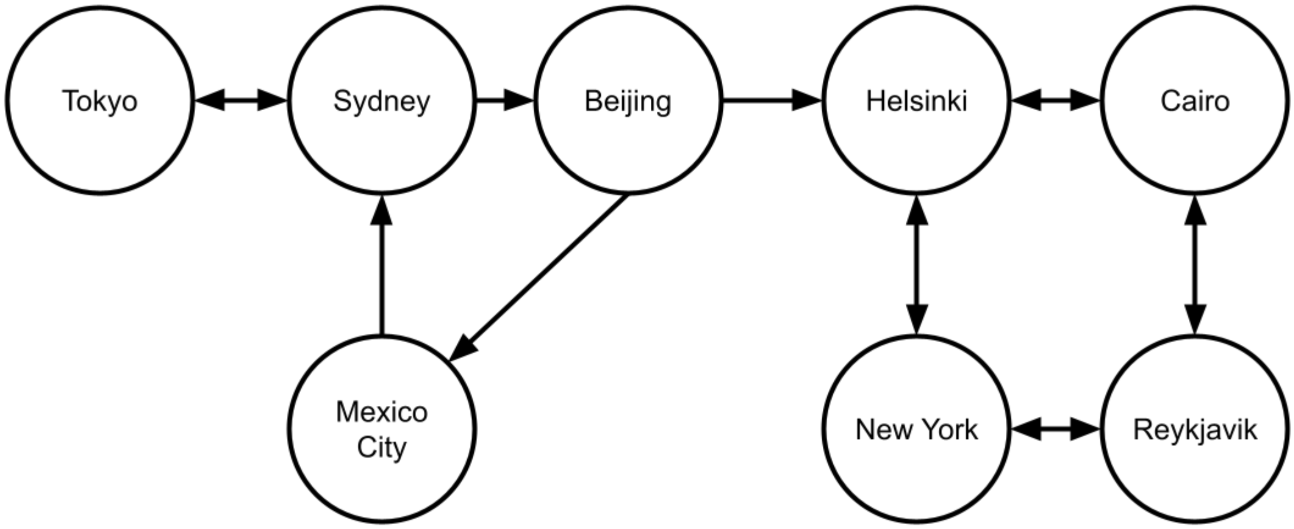
Problem 4: Finding All Reachable Destinations II

You are a travel coordinator for CodePath Airlines, and you're helping a customer find all possible destinations they can reach from a starting airport. The flight connections between airports are represented as an adjacency dictionary `flights`, where each key is a destination, and the corresponding value is a list of other destinations that are reachable through a direct flight.

Given a starting location `start`, write a function `get_all_destinations()` that uses Depth First Search (DFS) to return a list of all destinations that can be reached from `start`. The list should include both direct and connecting flights and should be ordered based on the order in which airports are visited in DFS.

```
def get_all_destinations(flights, start):
    pass
```

Example Usage:




```
flights = {
    "Tokyo": ["Sydney"],
    "Sydney": ["Tokyo", "Beijing"],
    "Beijing": ["Mexico City", "Helsinki"],
    "Helsinki": ["Cairo", "New York"],
    "Cairo": ["Helsinki", "Reykjavik"],
    "Reykjavik": ["Cairo", "New York"],
    "Mexico City": ["Sydney"]
}

print(get_all_destinations(flights, "Beijing"))
print(get_all_destinations(flights, "Helsinki"))
```

Example Output:

```
['Beijing', 'Mexico City', 'Sydney', 'Tokyo', 'Helsinki', 'Cairo', 'Reykjavik',
'New York']
['Helsinki', 'Cairo', 'Reykjavik', 'New York']
```

▶ 💡 **Hint: Depth First Search**

Problem 5: Find Itinerary

You are a traveler about to embark on a multi-leg journey with multiple flights to arrive at your final travel destination. You have all your boarding passes, but their order has gotten all messed up! You want to organize your boarding passes in the order you will use them, from your first flight all the way to your last flight that will bring you to your final destination.

Given a list of edges `boarding_passes` where each element `boarding_passes[i] = (departure_airport, arrival_airport)` represents a flight from `departure_airport` to `arrival_airport`, return an array with the itinerary listing out the airports you will pass through in the order you will visit them. Assume that departure is scheduled from every airport except the final destination, and each airport is visited only once (i.e., there are no cycles in the route).

```python
def find_itinerary(boarding_passes):
    pass
```

Example Usage:

```python
boarding_passes_1 = [
                    ("JFK", "ATL"),
                    ("SFO", "JFK"),
                    ("ATL", "ORD"),
                    ("LAX", "SFO")]

boarding_passes_2 = [
                    ("LAX", "DXB"),
                    ("DFW", "JFK"),
                    ("LHR", "DFW"),
                    ("JFK", "LAX")]

print(find_itinerary(boarding_passes_1))
print(find_itinerary(boarding_passes_2))
```

Example Output:

```
['LAX', 'SFO', 'JFK', 'ATL', 'ORD']
['LHR', 'DFW', 'JFK', 'LAX', 'DXB']
```

▶ 💡 **Hint: Pseudocode**

Problem 6: Finding Itinerary II

If you implemented `find_itinerary()` in the previous problem without using Depth First Search, solve it using DFS. If you solved it using DFS, try solving it using an alternative approach.

```python
def find_itinerary(boarding_passes):
    pass
```

Example Usage:

```python
boarding_passes_1 = [
                    ("JFK", "ATL"),
                    ("SFO", "JFK"),
                    ("ATL", "ORD"),
                    ("LAX", "SFO")]

boarding_passes_2 = [
                    ("LAX", "DXB"),
                    ("DFW", "JFK"),
                    ("LHR", "DFW"),
                    ("JFK", "LAX")]

print(find_itinerary(boarding_passes_1))
print(find_itinerary(boarding_passes_2))
```

Example Output:

```
['LAX', 'SFO', 'JFK', 'ATL', 'ORD']
['LHR', 'DFW', 'JFK', 'LAX', 'DXB']
```

▶ 💡 **Hint: Pseudocode**

Problem 7: Number of Flights

You are a travel planner and have an adjacency matrix `flights` with `n` airports labeled `0` to `n-1` where `flights[i][j]` indicates CodePath Airlines offers a flight from airport `i` to airport `j`. You are planning a trip for a client and want to know the minimum number of flights (edges) it will take to travel from airport `start` to their final destination airport `destination` on CodePath Airlines.

Return the minimum number of flights needed to travel from airport `i` to airport `j`. If it is not possible to fly from airport `i` to airport `j`, return `-1`.

```python
def counting_flights(flights, i, j):
    pass
```

Example Usage:

```python
# Example usage
flights = [
    [0, 1, 1, 0, 0], # Airport 0
    [0, 0, 1, 0, 0], # Airport 1
    [0, 0, 0, 1, 0], # Airport 2
    [0, 0, 0, 0, 1], # Airport 3
    [0, 0, 0, 0, 0]  # Airport 4
]

print(counting_flights(flights, 0, 2))
print(counting_flights(flights, 0, 4))
print(counting_flights(flights, 4, 0))
```

Example Output:

```
1
Example 1 Explanation: Flight path: 0 -> 2
3
Example 2 Explanation: Flight path 0 -> 2 -> 3 -> 4
-1
Explanation: Cannot fly from Airport 4 to Airport 0
```

▶ 💡 **Hint: BFS or DFS?**

Problem 8: Number of Airline Regions

CodePath Airlines operates in different regions around the world. Some airports are connected directly with flights, while others are not. However, if airport `a` is connected directly to airport `b`, and airport `b` is connected directly to airport `c`, then airport `a` is indirectly connected to airport `c`.

An airline region is a group of directly or indirectly connected airports and no other airports outside of the group.

You are given an `n x n` matrix `is_connected` where `is_connected[i][j] = 1` if CodePath Airlines offers a direct flight between airport `i` and airport `j`, and `is_connected[i][j] = 0` otherwise.

Return the total number of airline regions operated by CodePath Airlines.

```python
def num_airline_regions(is_connected):
    pass
```

Example Usage:

```
is_connected1 = [
    [1, 1, 0],
    [1, 1, 0],
    [0, 0, 1]
]

is_connected2 = [
    [1, 0, 0, 1],
    [0, 1, 1, 0],
    [0, 1, 1, 0],
    [1, 0, 0, 1]
]

print(num_airline_regions(is_connected1))
print(num_airline_regions(is_connected2))
```

Example Output:

```
2
2
```

Close Section

▾ Advanced Problem Set Version 2

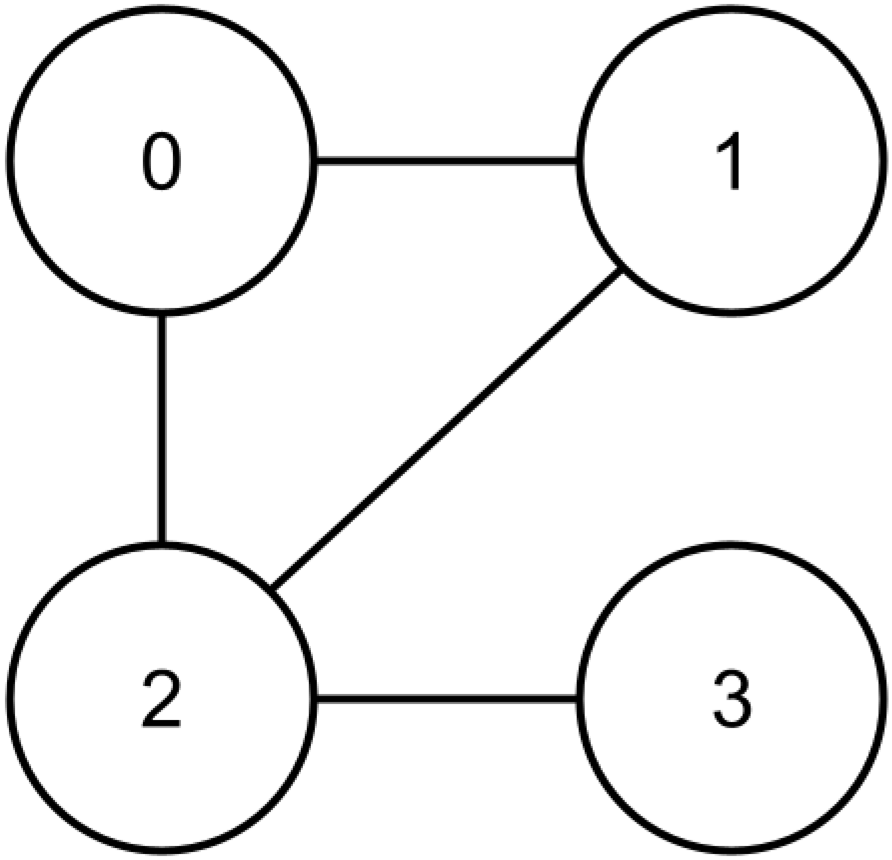
Problem 1: The Feeling is Mutual

You are given an insider look into the Hollywood gossip with an adjacency matrix `celebrities` where each node labeled 0 to `n` represents a celebrity. `celebrities[i][j] = 1` indicates that celebrity `i` likes celebrity `j` and `celebrities[i][j] = 0` indicates that celebrity `i` dislikes or doesn't know celebrity `j`. Write a function `is_mutual()` that returns `True` if all relationships between celebrities are mutual and `False` otherwise. A relationship between two celebrities is mutual if for any celebrity `i` that likes celebrity `j`, celebrity `j` also likes celebrity `i`.

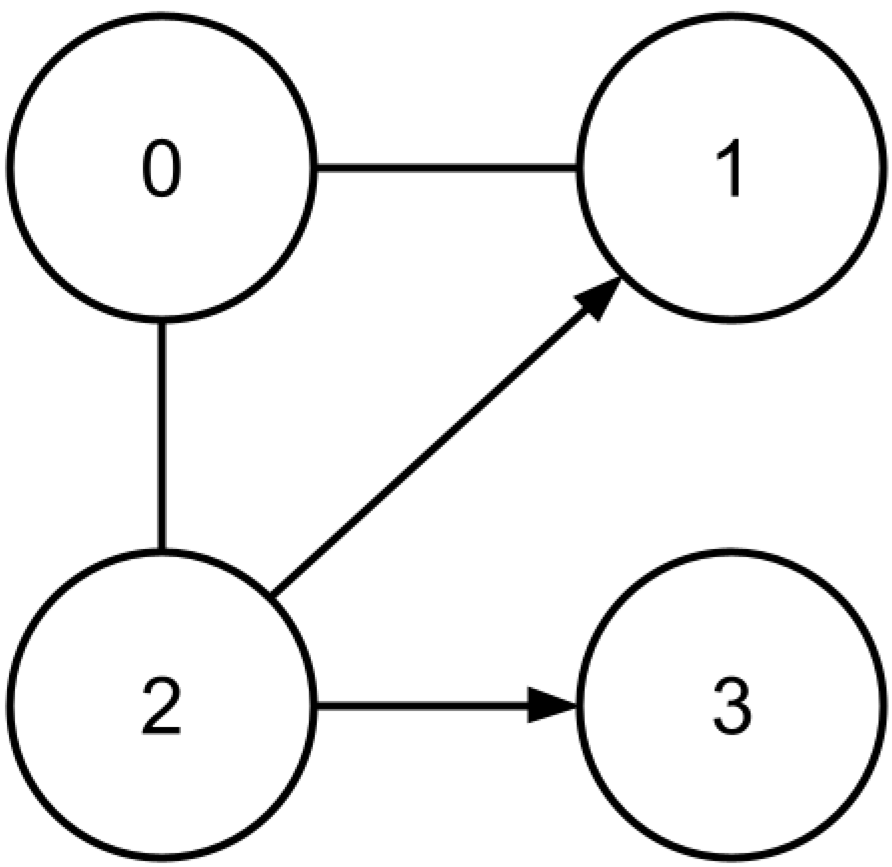
```
def is_mutual(celebrities):
    pass
```

Example Usage:

Example 1: `celebrities1`



Example 2: `celebrities2`



```python
celebrities1 = [
                [0, 1, 1, 0],
                [1, 0, 1, 0],
                [1, 1, 0, 1],
                [0, 0, 1, 0]]

celebrities2 = [
                [0, 1, 1, 0],
                [1, 0, 0, 0],
                [1, 1, 0, 1],
                [0, 0, 0, 0]]

print(is_mutual(celebrities1))
print(is_mutual(celebrities2))
```

Example Output:

```
True
False
```

▶ 💡 **Hint: Introduction to Graphs**

Problem 2: Network Lookup

You work for a talent agency and have a 2D list `clients` where `clients[i] = [celebrity_a, celebrity_b]` indicates that `celebrity_a` and `celebrity_b` have worked with each other. You want to create a more efficient lookup system for your agency by transforming `clients` into an equivalent adjacency matrix.

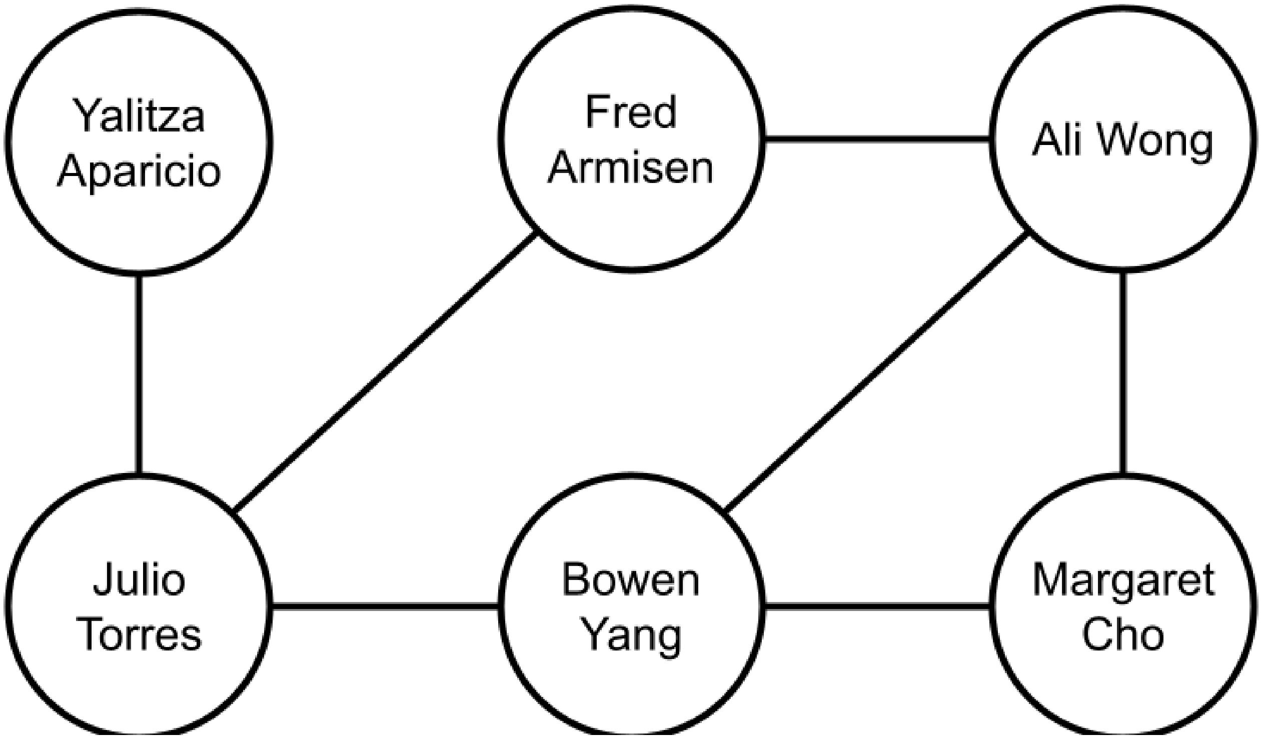
Given `contacts`:

1. Create a map of each unique celebrity in `contacts` to an integer ID with values `0` through `n`.
2. Using the celebrities' IDs, create an adjacency matrix where `matrix[i][j] = 1` indicates that celebrity with ID `i` has worked with celebrity with ID `j`. Otherwise, `matrix[i][j]` should have value `0`.

Return both the dictionary mapping celebrities to their ID and the adjacency matrix.

```python
def get_adj_matrix(clients):
    pass
```

Example Usage:




```
clients = [
            ["Yalitza Aparicio", "Julio Torres"],
            ["Julio Torres", "Fred Armisen"],
            ["Bowen Yang", "Julio Torres"],
            ["Bowen Yang", "Margaret Cho"],
            ["Margaret Cho", "Ali Wong"],
            ["Ali Wong", "Fred Armisen"],
            ["Ali Wong", "Bowen Yang"]]

id_map, adj_matrix = get_adj_matrix(clients)
print(id_map)
print(adj_matrix)
```

Example Output:

```
{
  'Fred Armisen': 0,
  'Yalitza Aparicio': 1,
  'Margaret Cho': 2,
  'Bowen Yang': 3,
  'Ali Wong': 4,
  'Julio Torres': 5
}

[
  [0, 0, 0, 0, 1, 1],  # Fred Armisen
  [0, 0, 0, 0, 0, 1],  # Yalitza Aparicio
  [0, 0, 0, 1, 1, 0],  # Margaret Cho
  [0, 0, 1, 0, 1, 1],  # Bowen Yang
  [1, 0, 1, 1, 0, 0],  # Ali Wong
  [1, 1, 0, 1, 0, 0]   # Julio Torres
]

Note: The order in which you assign IDs and consequently your adjacency matrix may l
```

▶ 💡 **Hint: Converting Between Graph Representations**

Problem 3: Secret Celebrity

A new reality show is airing in which a famous celebrity pretends to be a non-famous person. As contestants get to know each other, they have to guess who the celebrity among them is to win the show. An even bigger twist: there might be no celebrity at all! The show has `n` contestants labeled from 1 to `n`.

If the celebrity exists, then:

1. The celebrity trusts none of the contestants.
2. Due to the celebrity's charisma, all the contestants trust the celebrity.
3. There is exactly one person who satisfies rules 1 and 2.

You are given an array `trust` where `trust[i] = [a, b]` indicates that contestant `a` trusts contestant `b`. If a trust relationship does not exist in `trust` array, then such a trust relationship does not exist.

Return the label of the celebrity if they exist and can be identified. Otherwise, return `-1`.

```python
def identify_celebrity(trust, n):
    pass
```

Example Usage:

```
trust1 = [[1,2]]
trust2 = [[1,3],[2,3]]
trust3 = [[1,3],[2,3],[3,1]]

identify_celebrity(trust1, 2)
identify_celebrity(trust2, 3)
identify_celebrity(trust3, 3)
```

Example Output:

```
2
3
-1
```

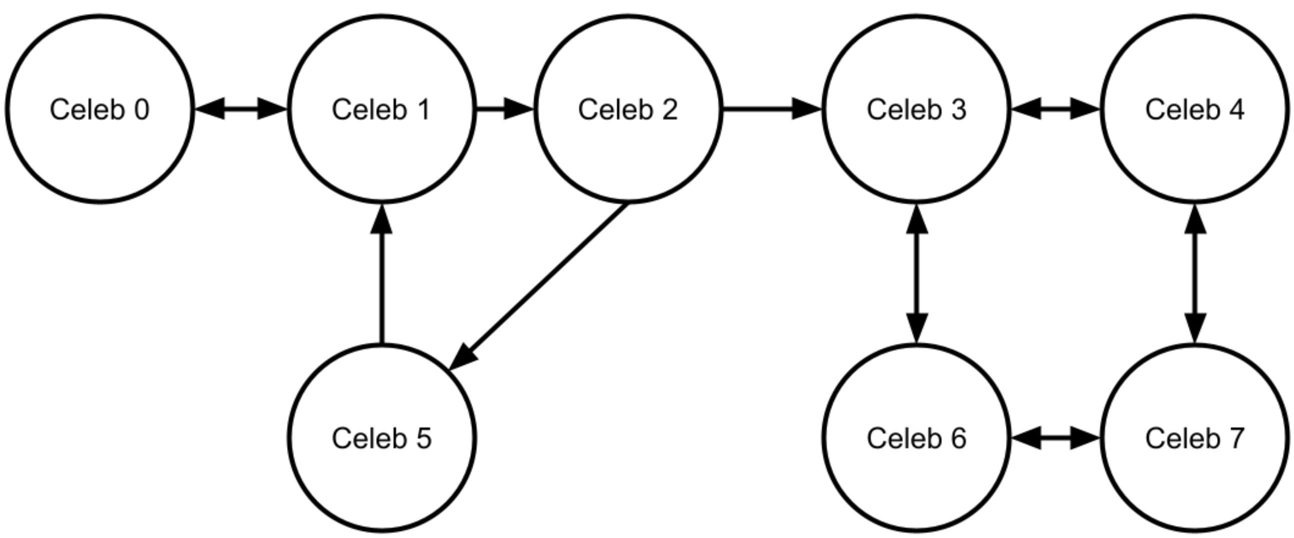
Problem 4: Casting Call Search

You are a casting agent for a major Hollywood production and the director has a certain celebrity in mind for the lead role. You have an adjacency matrix `celebs` where `celebs[i][j] = 1` means that celebrity `i` has a connection with celebrity `j`, and `celebs[i][j] = 0` means they don't. Connections are directed meaning that `celebs[i][j] = 1` does not automatically mean `celebs[j][i] = 1`.

Given a celebrity you know `start_celeb` and the celebrity the director wants to hire `target_celeb`, use Breadth First Search to return `True` if you can find a path of connections from `start_celeb` to `target_celeb`. Otherwise, return `False`.

```python
def have_connection(celebs, start_celeb, target_celeb):
    pass
```

Example Usage:



```python
celebs = [
            [0, 1, 0, 0, 0, 0, 0, 0], # Celeb 0
            [0, 1, 1, 0, 0, 0, 0, 0], # Celeb 1
            [0, 0, 0, 1, 0, 1, 0, 0], # Celeb 2
            [0, 0, 0, 0, 1, 0, 1, 0], # Celeb 3
            [0, 0, 0, 1, 0, 0, 0, 1], # Celeb 4
            [0, 1, 0, 0, 0, 0, 0, 0], # Celeb 5
            [0, 0, 0, 1, 0, 0, 0, 1], # Celeb 6
            [0, 0, 0, 0, 1, 0, 1, 0]  # Celeb 7
            ]

print(have_connection(celebs, 0, 6))
print(have_connection(celebs, 3, 5))
```

Example Output:

```
True
False
```

▶ 💡 **Hint: Breadth First Search**

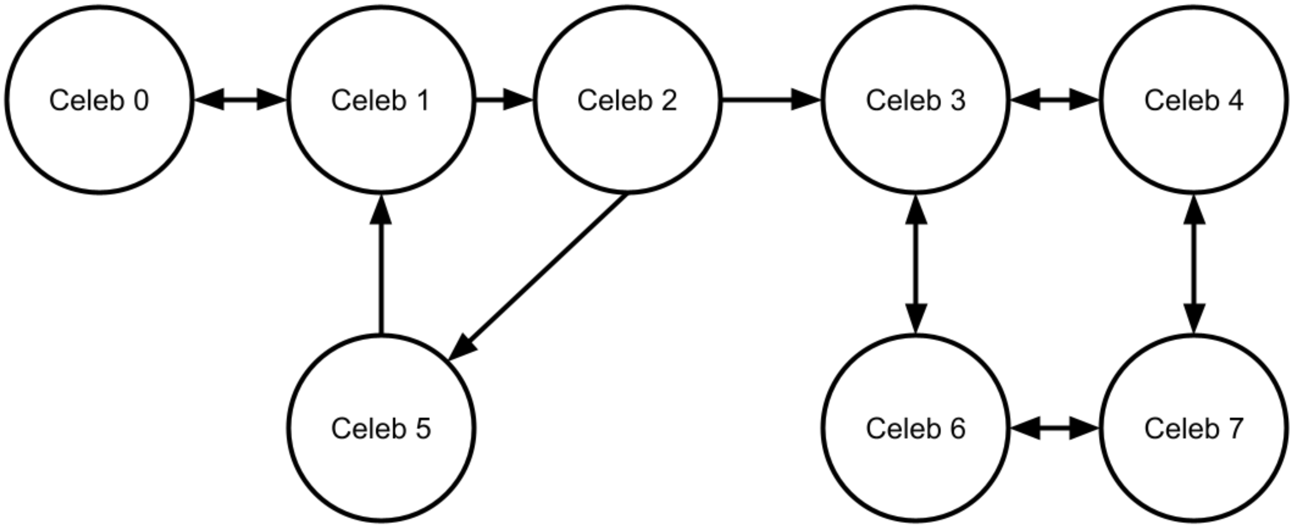
Problem 5: Casting Call Search II

You are a casting agent for a major Hollywood production and the director has a certain celebrity in mind for the lead role. You have an adjacency matrix `celebs` where `celebs[i][j] = 1` means that celebrity `i` has a connection with celebrity `j`, and `celebs[i][j] = 0` means they don't. Connections are directed meaning that `celebs[i][j] = 1` does not automatically mean `celebs[j][i] = 1`.

Given a celebrity you know `start_celeb` and the celebrity the director wants to hire `target_celeb`, use **Depth First Search** to return `True` if you can find a path of connections from `start_celeb` to `target_celeb`. Otherwise, return `False`.

```python
def have_connection(celebs, start_celeb, target_celeb):
    pass
```

Example Usage:




```python
celebs = [
            [0, 1, 0, 0, 0, 0, 0, 0], # Celeb 0
            [0, 1, 1, 0, 0, 0, 0, 0], # Celeb 1
            [0, 0, 0, 1, 0, 1, 0, 0], # Celeb 2
            [0, 0, 0, 0, 1, 0, 1, 0], # Celeb 3
            [0, 0, 0, 1, 0, 0, 0, 1], # Celeb 4
            [0, 1, 0, 0, 0, 0, 0, 0], # Celeb 5
            [0, 0, 0, 1, 0, 0, 0, 1], # Celeb 6
            [0, 0, 0, 0, 1, 0, 1, 0]  # Celeb 7
            ]

print(have_connection(celebs, 0, 6))
print(have_connection(celebs, 3, 5))
```

Example Output:

```
True
False
```

▶ 💡 **Hint: Depth First Search**

Problem 6: Copying Seating Arrangements

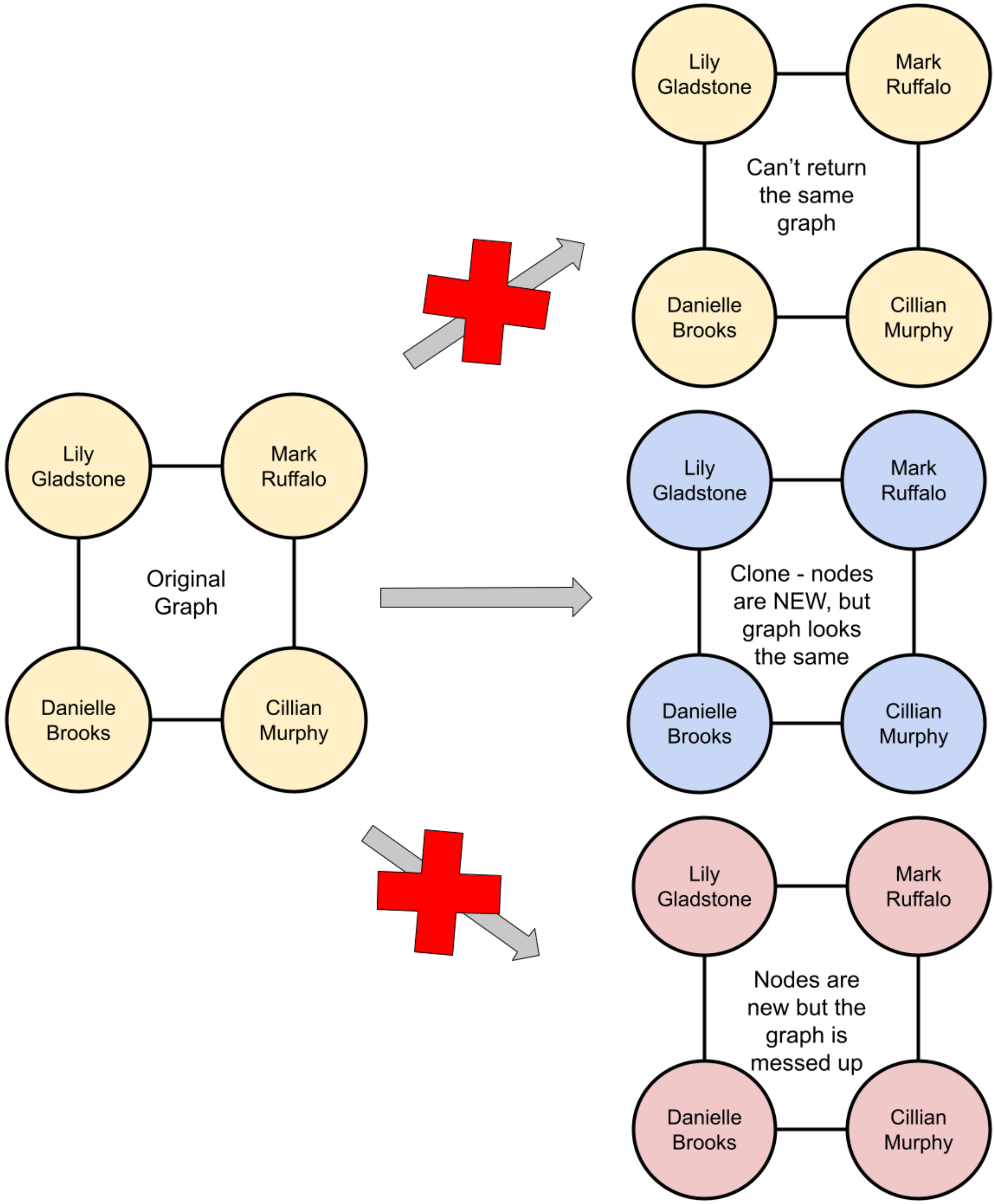
You are organizing the seating arrangement for a big awards ceremony and want to make a copy for your assistant. The seating arrangement is stored in a graph where each `Node` value `val` is the name of a celebrity guest at the ceremony and its array `neighbors` are all the guests sitting in seats adjacent to the celebrity.

Given a reference to a `Node` in the original seating arrangement `seat`, make a deep copy (clone) of the seating arrangement. Return the copy of the given node.

We have provided a function `compare_graphs()` to help with testing this function. To use this function, pass in the given node `seat` and the copy of that node your function `copy_seating()` returns. If the two graphs are clones of each other, the function will return `True`. Otherwise, the function will return `False`.

```
class Node():
    def __init__(self, val = 0, neighbors = None):
        self.val = val
        self.neighbors = neighbors if neighbors is not None else []

# Function to test if two seating arrangements (graphs) are identical
def compare_graphs(node1, node2, visited=None):
    if visited is None:
        visited = set()

    if node1.val != node2.val:
        return False

    visited.add(node1)

    if len(node1.neighbors) != len(node2.neighbors):
        return False

    for n1, n2 in zip(node1.neighbors, node2.neighbors):
        if n1 not in visited and not compare_graphs(n1, n2, visited):
            return False

    return True

def copy_seating(seat):
    pass
```

Example Usage:



Lily Gladstone
Mark Ruffalo
Can't return the same graph
Danielle Brooks
Cillian Murphy
Lily Gladstone
Mark Ruffalo
Original Graph
Danielle Brooks
Cillian Murphy
Lily Gladstone
Mark Ruffalo
Clone - nodes are NEW, but graph looks the same
Danielle Brooks
Cillian Murphy
Lily Gladstone
Mark Ruffalo
Nodes are new but the graph is messed up
Danielle Brooks
Cillian Murphy

```python
lily = Node("Lily Gladstone")
mark = Node("Mark Ruffalo")
cillian = Node("Cillian Murphy")
danielle = Node("Danielle Brooks")
lily.neighbors.extend([mark, danielle])
mark.neighbors.extend([lily, cillian])
cillian.neighbors.extend([danielle, mark])
danielle.neighbors.extend([lily, cillian])

copy = copy_seating(lily)
print(compare_graphs(lily, copy))
```

Example Output:

```
True
```

Problem 7: Gossip Chain

In Hollywood, rumors spread rapidly among celebrities through various connections. Imagine each celebrity is represented as a vertex in a directed graph, and the connections between them are directed edges indicating who spread the latest gossip to whom.

The arrival time of a rumor for a given celebrity is the moment the rumor reaches them for the first time, and the departure time is when all the celebrities they could influence have already heard the rumor, meaning they are no longer involved in spreading it.

Given a list of edges `connections` representing connections between celebrities and the number of celebrities in the the graph `n`, find the arrival and departure time of the rumor for each celebrity in a Depth First Search (DFS) starting from a given celebrity `start`.

Return a dictionary where each celebrity in `connections` is a key whose corresponding value is a tuple `(arrival_time, departure_time)` representing the arrival and departure times of the rumor for that celebrity. If a celebrity never hears the rumor their arrival and departure times should be `(-1, -1)`.

```python
def rumor_spread_times(connections, n, start):
    pass
```

Example Usage:

```python
connections = [
    ["Amber Gill", "Greg O'Shea"],
    ["Amber Gill", "Molly-Mae Hague"],
    ["Greg O'Shea", "Molly-Mae Hague"],
    ["Greg O'Shea", "Tommy Fury"],
    ["Molly-Mae Hague", "Tommy Fury"],
    ["Tommy Fury", "Ovie Soko"],
    ["Curtis Pritchard", "Maura Higgins"]
]

print(rumor_spread_times(connections, 7, "Amber Gill"))
```

Example Output:

```
{
    "Amber Gill": (1, 12),
    "Greg O'Shea": (2, 11),
    "Molly-Mae Hague": (3, 8),
    "Tommy Fury": (4, 7),
    "Ovie Soko": (5, 6),
    "Curtis Pritchard": (-1, -1),
    "Maura Higgins": (-1, -1)
}
```

Problem 8: Network Strength

Given a group of celebrities as an adjacency dictionary `celebrities`, return `True` if the group is strongly connected and `False` otherwise. The list `celebrities[i]` is the list of all celebrities celebrity `i` likes. Mutual like between two celebrities is not guaranteed. The graph is said to be strongly connected if every celebrity likes every other celebrity in the network.

```python
def is_strongly_connected(celebrities):
    pass
```

Example Usage:

```python
celebrities1 = {
    "Dev Patel": ["Meryl Streep", "Viola Davis"],
    "Meryl Streep": ["Dev Patel", "Viola Davis"],
    "Viola Davis": ["Meryl Streep", "Viola Davis"]
}

celebrities2 = {
    "John Cho": ["Rami Malek", "Zoe Saldana", "Meryl Streep"],
    "Rami Malek": ["John Cho", "Zoe Saldana", "Meryl Streep"],
    "Zoe Saldana": ["Rami Malek", "John Cho", "Meryl Streep"],
    "Meryl Streep": []
}


print(is_strongly_connected(celebrities1))
print(is_strongly_connected(celebrities2))
```

Example Output:

```
True
False
```

Close Section