TIP102 | Intermediate Technical Interview Prep

Intermediate Technical Interview Prep Summer 2025 (@ Section 1b | Tuesdays and Thursdays 3PM - 5PM PDT)
Personal Member ID#: 126663

Need help? Post on our class slack channel or email us at support@codepath.org

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.

▼ **Note on Expectations**

Please Note: **It is not required or expected** that you complete all of the practice problems! In some sessions you may only complete 1 problem and that's okay.

Strengthening your **approach** to problems, and your **ability to speak and engage through the process** are key skills most often underdeveloped for engineers at this stage - focus on those in our small groups for your long term success!

You can always return to problems independently, after class time, to embrace the technical concepts and gain additional practice.

Close Section

🔎 Problem Solving Approach

We will approach problems using the six steps in the UMPIRE approach.

UMPIRE: Understand, Match, Plan, Implement, Review, Evaluate.

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

We will learn to:

- **Understand** the problem
- **Match** identifies common approaches you've seen/used before
- **Plan** a solution step-by-step, and
- **Implement** the solution
- **Review** your solution
- **Evaluate** your solution's time and space complexity and think critically about the advantages and disadvantages of your chosen approach.

Breakout Problems Session 1

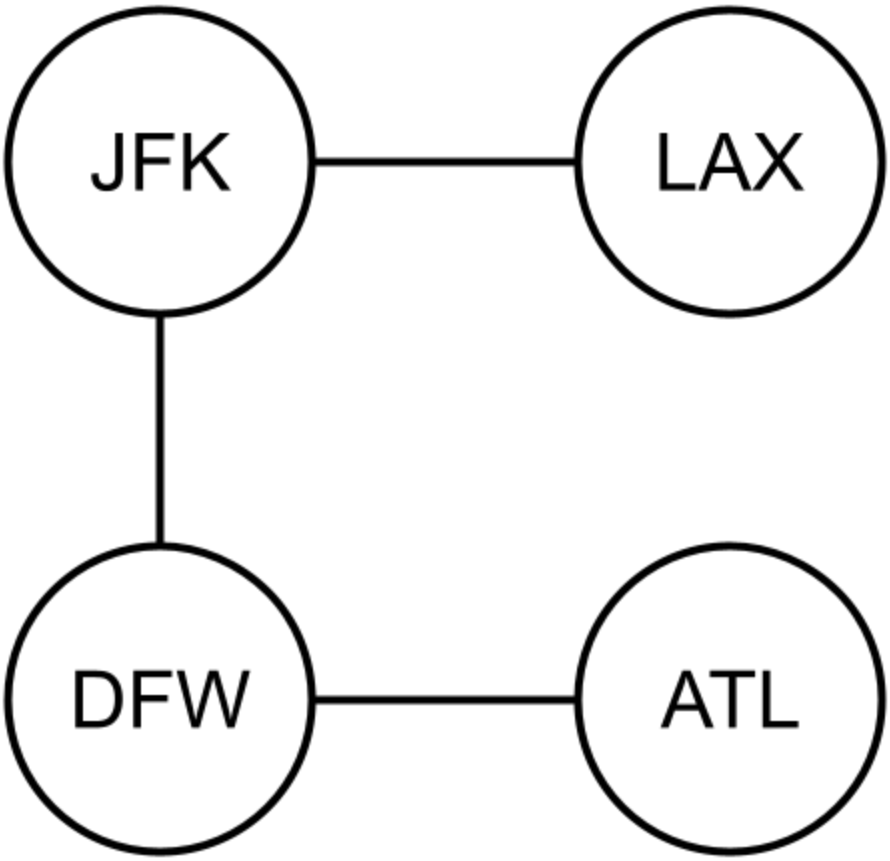
▼ Standard Problem Set Version 1

Problem 1: Graphing Flights

The following graph represents the different flights offered by CodePath Airlines. Each node or vertex represents an airport (JFK - New York City, LAX - Los Angeles, DFW - Dallas Fort Worth, and ATL -

Atlanta), and an edge between two vertices indicates that CodePath airlines offers flights between those two airports.

Create a variable `flights` that represents the undirected graph below as an adjacency dictionary, where each node's value is represented by a string with the airport's name (ex. `"JFK"`).




```
"""
JFK ----- LAX
|
|
DFW ----- ATL
"""
# No starter code is provided for this problem
# Add your code here
```

Example Usage:

```
print(list(flights.keys()))
print(list(flights.values()))
print(flights["JFK"])
```

Example Output:

```
['JFK', 'LAX', 'DFW', 'ATL']
[['LAX', 'DFW'], ['JFK'], ['ATL', 'JFK'], ['DFW']]
```

▼ 💡 **Hint: Introduction to Graphs**

This problem requires you to be familiar with the graph data structure and the different methods for representing graphs. Check out the Unit 10 Cheatsheet if you are unfamiliar with these concepts.

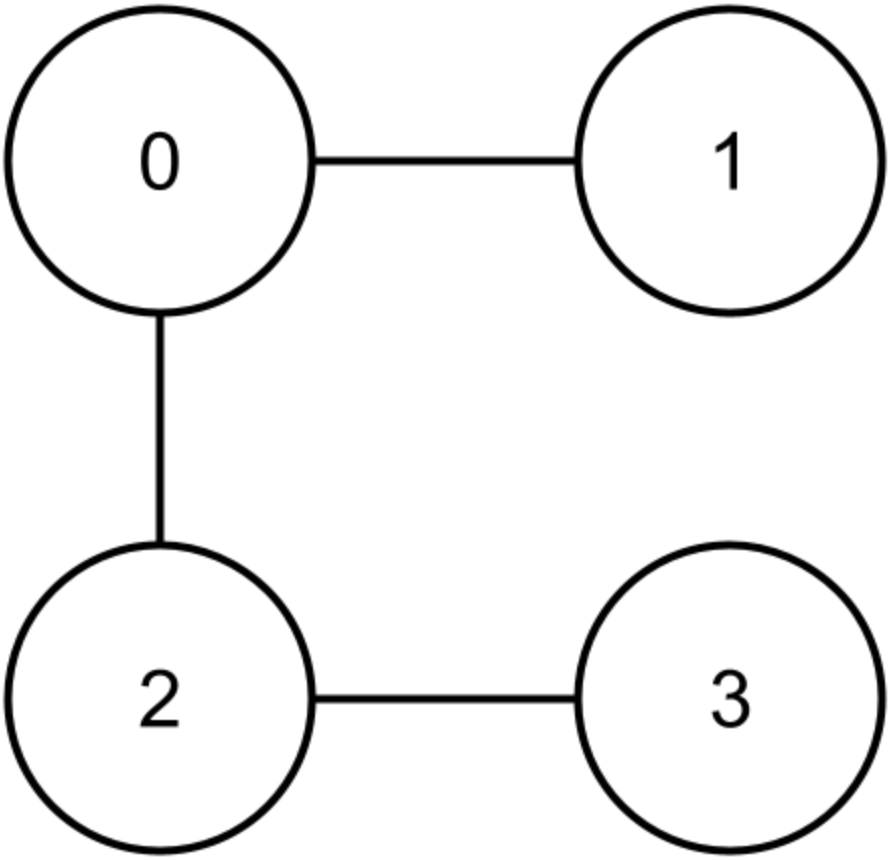
Problem 2: There and Back

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.

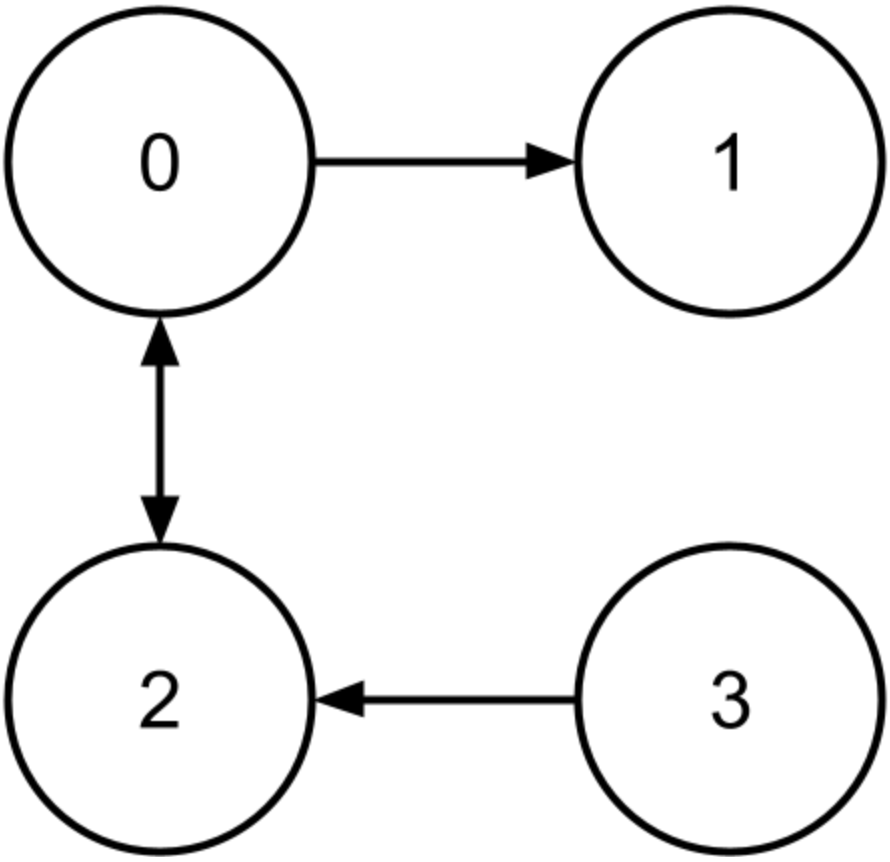
```
def bidirectional_flights(flights):
    pass
```

Example Usage:

Example 1: `flights1`



Example 2: `flights2`



Path Courses

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

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

Example Output:

```
True
False
```

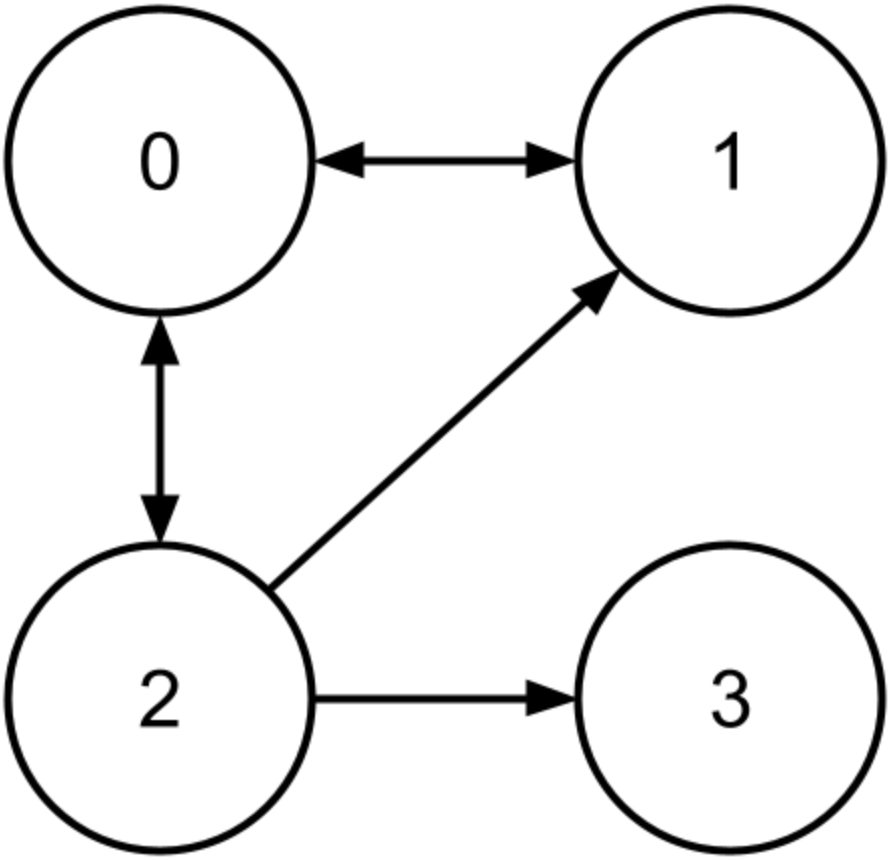
Problem 3: Finding Direct Flights

Given an adjacency matrix `flights` of size `n x n` where each of the `n` nodes in the graph represent a distinct destination and `n[i][j] = 1` indicates that there exists a flight from destination `i` to destination `j` and `n[i][j] = 0` indicates that no such flight exists. Given `flights` and an integer `source` representing the destination a customer is flying out of, return a list of all destinations the customer can reach from `source` on a direct flight. You may return the destinations in any order.

A customer can reach a destination on a direct flight if that destination is a neighbor of `source`.

```
def get_direct_flights(flights, source):
    pass
```

Example Usage:




```
flights = [
            [0, 1, 1, 0],
            [1, 0, 0, 0],
            [1, 1, 0, 1],
            [0, 0, 0, 0]]

print(get_direct_flights(flights, 2))
print(get_direct_flights(flights, 3))
```

Example Output:

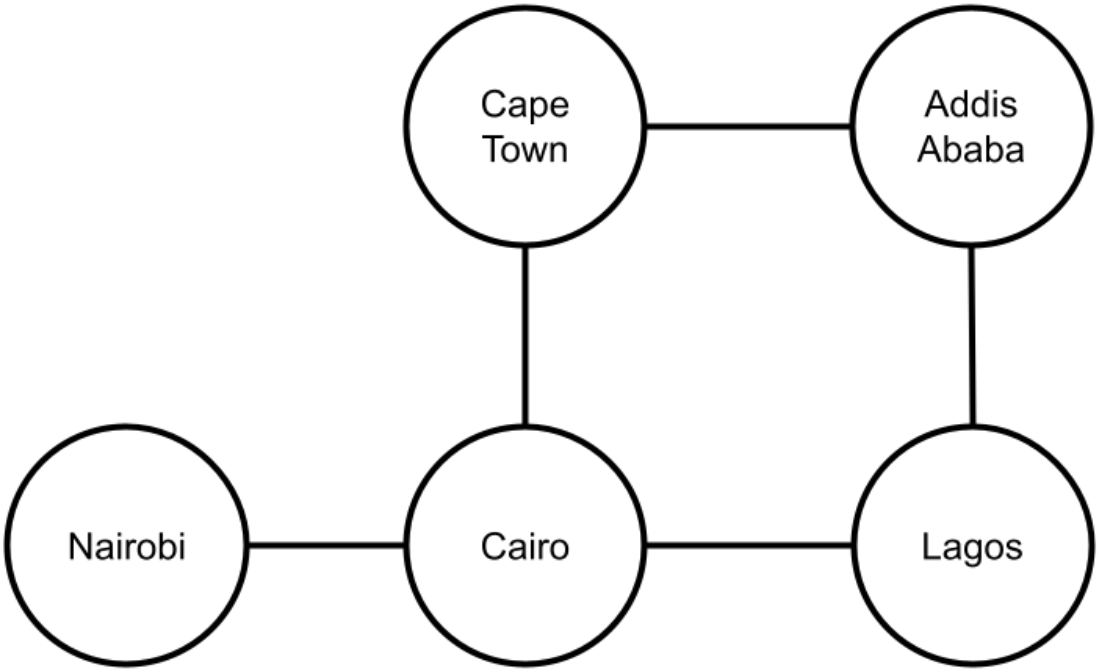
```
[0, 1, 3]
[]
```

Problem 4: Converting Flight Representations

Given a list of edges `flights` where `flights[i] = [a, b]` denotes that there exists a bidirectional flight (incoming and outgoing flight) from city `a` to city `b`, return an adjacency dictionary `adj_dict` representing the same flights graph where `adj_dict[a]` is an array denoting there is a flight from city `a` to each city in `adj_dict[a]`.

```
def get_adj_dict(flights):
```

Example Usage:




```
flights = [['Cape Town', 'Addis Ababa'], ['Cairo', 'Lagos'], ['Lagos', 'Addis Ababa'],
           ['Nairobi', 'Cairo'], ['Cairo', 'Cape Town']]
print(get_adj_dict(flights))
```

Example Output:

```
{
    'Cape Town': ['Addis Ababa', 'Cairo'],
    'Addis Ababa': ['Cape Town', 'Lagos'],
    'Lagos': ['Cairo', 'Addis Ababa'],
    'Cairo': ['Cape Town', 'Nairobi'],
    'Nairobi': ['Cairo']
}
```

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

This problem requires you to convert between two different graph representations. Converting between graph representations is a common subproblem when solving more advanced problems. This is especially true when you are given a list of edges and need to easily find a node's neighbors.

Problem 5: Find Center of Airport

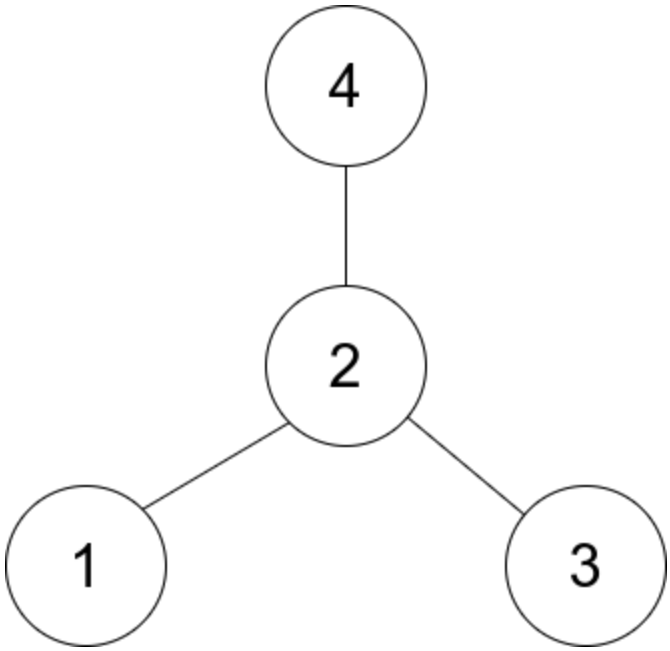
You are a pilot navigating a new airport and have a map of the airport represented as an undirected star graph with `n` nodes where each node represents a terminal in the airport labeled from `1` to `n`. You want to find the center terminal in the airport where the pilots' lounge is located.

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.

```
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**

Observe that in a star graph the center node is connected to all other nodes. It must appear in all but one of the edges.

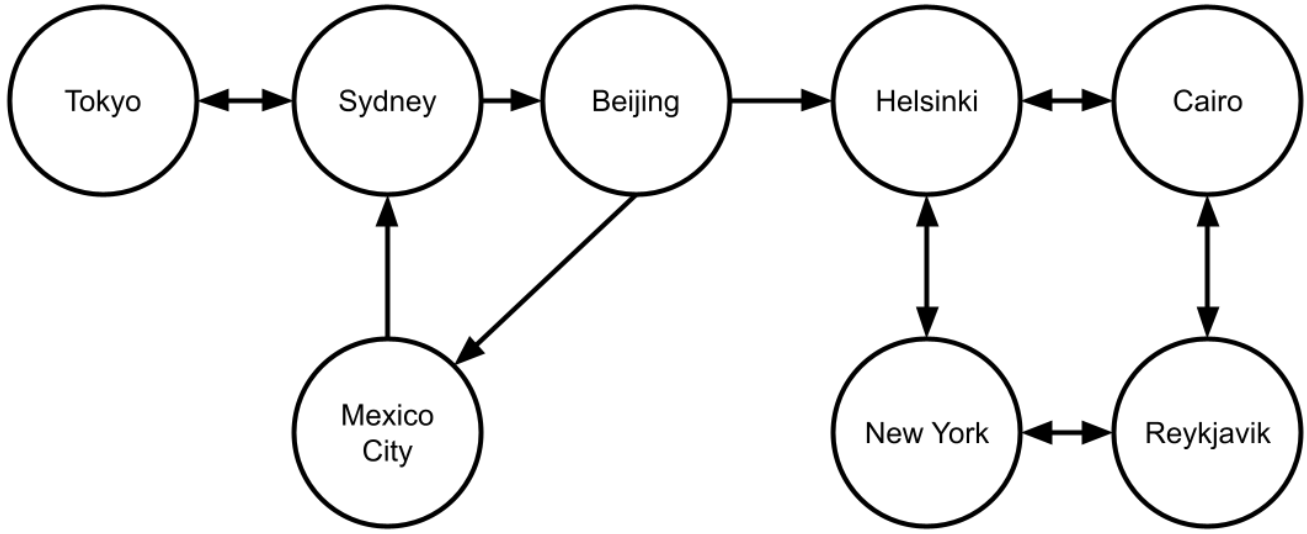
Problem 6: Finding All Reachable Destinations

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, 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"],
    "New York": []
}

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

Example Output:

```
['Helsinki', 'Cairo', 'New York', 'Reykjavik']
```

▼ ✨ AI Hint: Breadth First Search Traversal

Key Skill: Use AI to explain code concepts

To solve this problem, it may be helpful to understand both the **queue** data structure and **breadth first search** algorithm. To learn more about these concepts, visit the Queues and Breadth First Search sections of the Cheatsheet.

If you still have questions, try explaining what you're doing, and ask an AI tool like ChatGPT or GitHub Copilot to help you understand what's confusing you. For example, you might ask:

"I'm trying to understand how to use a queue to implement breadth first search, but I'm confused about why I wouldn't just use a list. Can you explain the difference?"

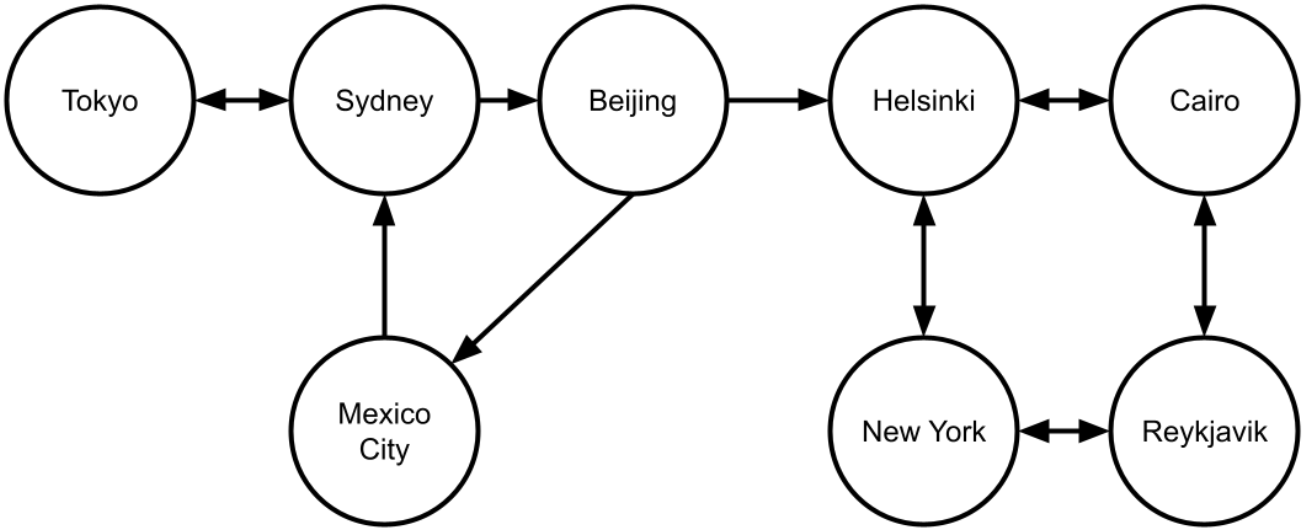
Problem 7: Finding All Reachable Destinations II

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**

This problem requires you to perform a depth first search traversal of a graph. If you need a primer on how to perform DFS on a graph, check out the unit cheatsheet.

Problem 8: 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

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).

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

Example Usage:

```
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**

One possible approach to this problem is to use a dictionary.

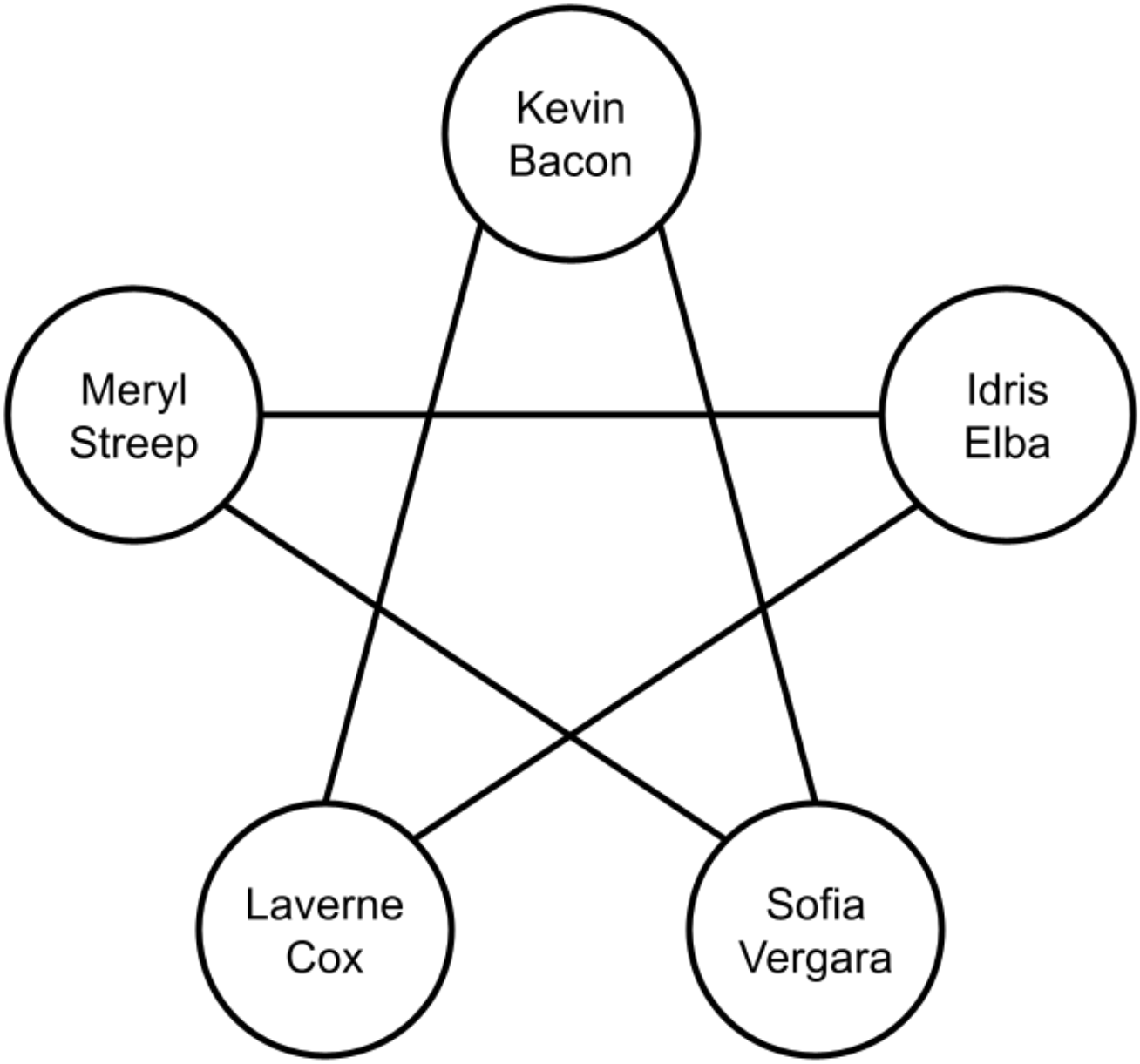
1. Create a dictionary that maps each deaprture airport to its corresponding arrival airport for efficient lookup.
2. Identify the starting airport. It is the only airport that is only a departure airport and never an arrival airport.
3. Trace the itinerary by following the mapping from departure to arrival until there are no more flights.

Standard Problem Set Version 2

Problem 1: Hollywood Stars

The following graph illustrates connections between different Hollywood stars. Each node represents a celebrity, and an edge between two nodes indicates that the celebrities know each other.

Create a variable `hollywood_stars` that represents the undirected graph below as an adjacency dictionary, where each node's value is represented by a string with the airport's name (ex. `"Kevin Bacon"`).




```
# No starter code is provided for this problem
# Add your code here
```

```
print(list(hollywood_stars.keys()))
print(list(hollywood_stars.values()))
print(hollywood_stars["Kevin Bacon"])
```

Example Output:

```
['Kevin Bacon', 'Meryl Streep', 'Idris Elba', 'Laverne Cox', 'Sofia Vergara']
[['Laverne Cox', 'Sofia Vergara'], ['Idris Elba', 'Sofia Vergara'], ['Meryl Streep', 'Laverne
['Kevin Bacon', 'Idris Elba'], ['Kevin Bacon', 'Meryl Streep']]
['Laverne Cox', 'Sofia Vergara']
```

▼ 💡 **Hint: Introduction to Graphs**

This problem requires you to be familiar with the graph data structure and the different methods for representing graphs. Check out the Unit 10 Cheatsheet if you are unfamiliar with these concepts.

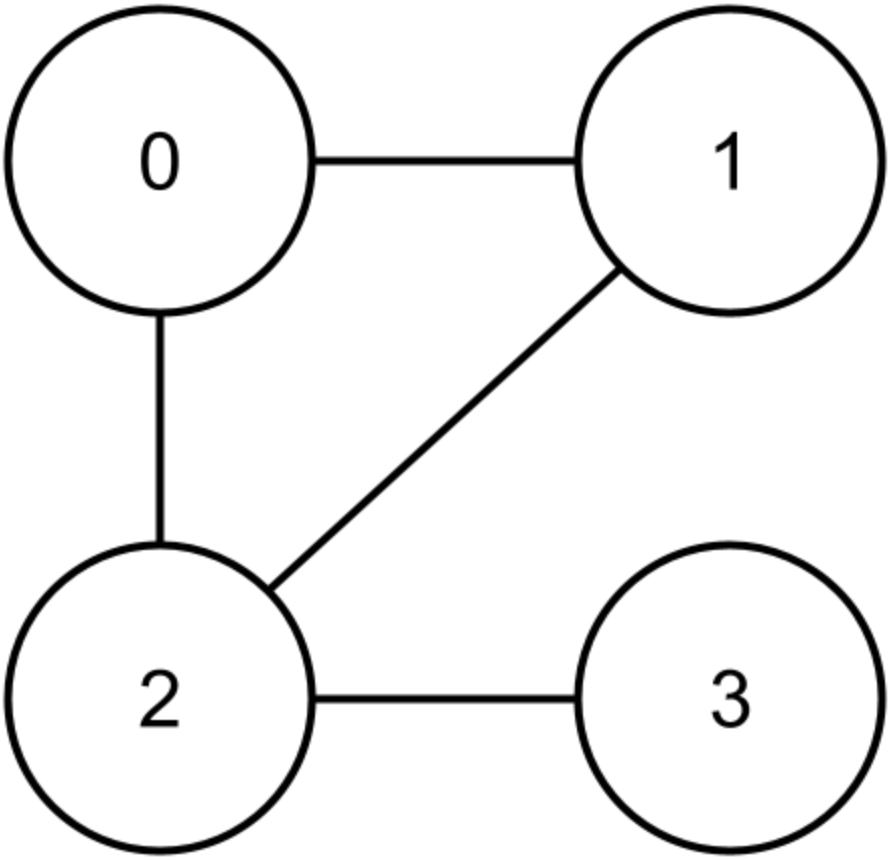
Problem 2: The Feeling is Mutual

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`.

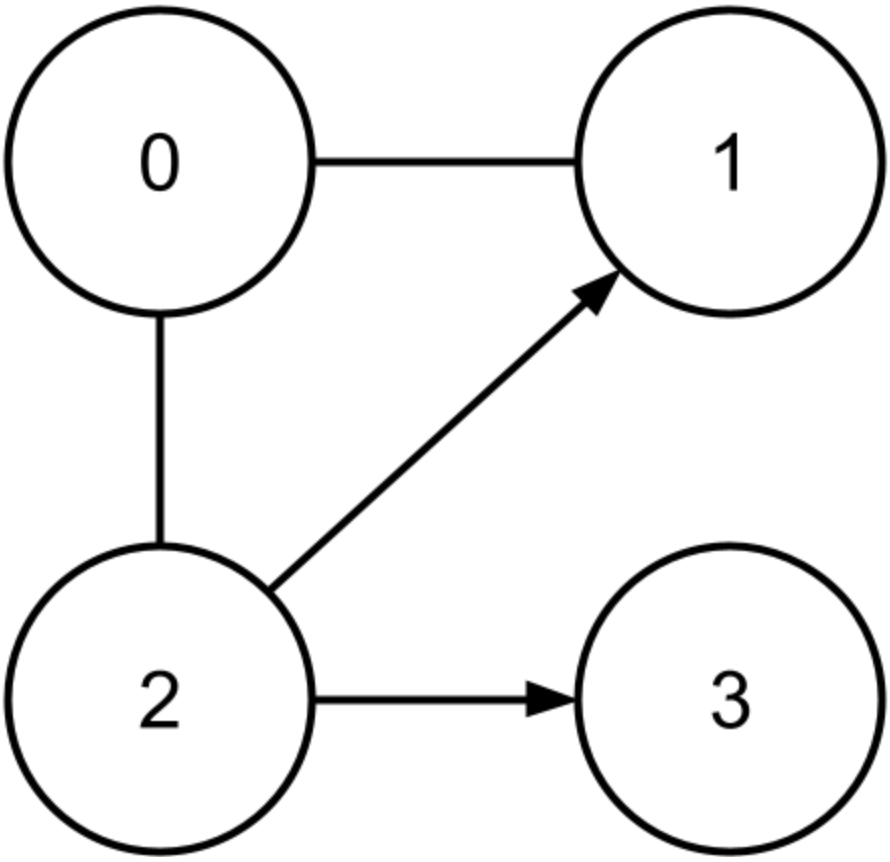
```
def is_mutual(celebrities):
    pass
```

Example Usage:

Example 1: `celebrities1`



Example 2: `celebrities2`



Path Courses

```
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
```

Problem 3: Closest Friends

You are a talented actor looking for your next big movie and want to leverage your connections to see if there are any good roles available. To increase your chances, you want to ask your closest friends first.

You have a 2D list `contacts` where `contacts[i] = [celebrity_a, celebrity_b]` indicates that there is a mutual relationship (undirected edge) between `celebrity_a` and `celebrity_b`. Given a celebrity `celeb`, return a list of the celebrity's closest friends.

`celebrity_b` is a close friend of `celebrity_a` if they are neighbors in the graph.

```
def get_close_friends(contacts, celeb):
    pass
```

Example Usage:

```
contacts = [["Lupita Nyong'o", "Jordan Peele"], ["Meryl Streep", "Jordan Peele"], ["Meryl Str
["Greta Gerwig", "Meryl Streep"], ["Ali Wong", "Greta Gergwig"]]

print(get_close_friends(contacts, "Lupita Nyong'o"))
print(get_close_friends(contacts, "Greta Gerwig"))
```

Example Output:

```
['Jordan Peele', 'Meryl Streep']
['Meryl Streep', 'Ali Wong']
```

Problem 4: 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.

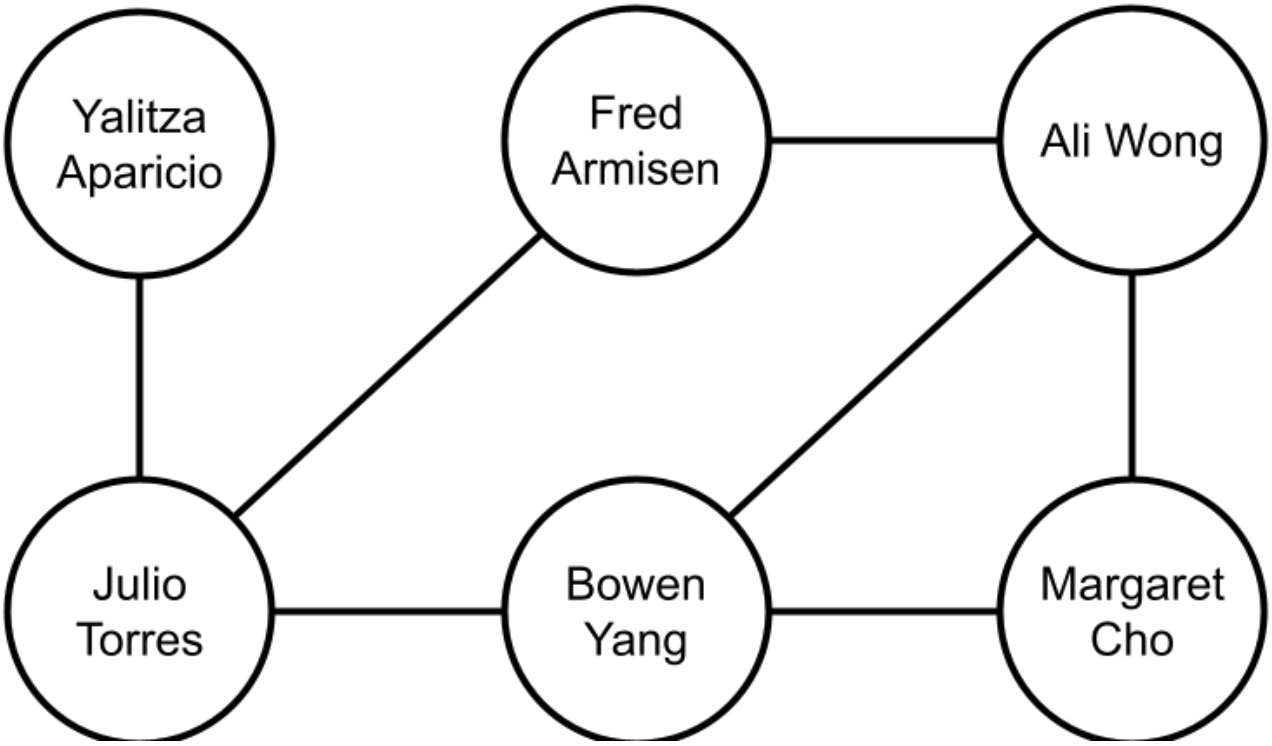
Given `contacts`:

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.

```
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"]]
```

```
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 look diffe
```

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

This problem requires you to convert between two different graph representations. Converting between graph representations is a common subproblem when solving more advanced problems. This is especially true when you are given a list of edges and need to easily find a node's neighbors.

Problem 5: 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.

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`.

```
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
```

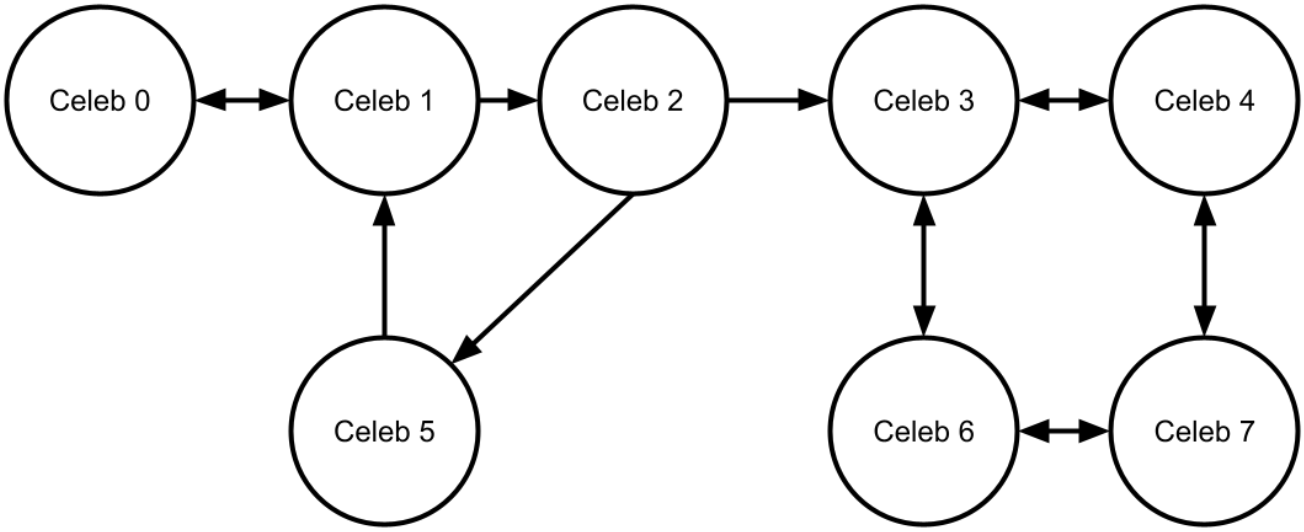
Problem 6: Casting Call Search

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`.

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

Example Usage:




```
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
```

▼ ✨ AI Hint: Breadth First Search Traversal

Key Skill: Use AI to explain code concepts

To solve this problem, it may be helpful to understand both the **queue** data structure and **breadth first search** algorithm. To learn more about these concepts, visit the Queues and Breadth First Search sections of the Cheatsheet.

If you still have questions, try explaining what you're doing, and ask an AI tool like ChatGPT or GitHub Copilot to help you understand what's confusing you. For example, you might ask:

"I'm trying to understand how to use a queue to implement breadth first search, but I'm confused about why I wouldn't just use a list. Can you explain the difference?"

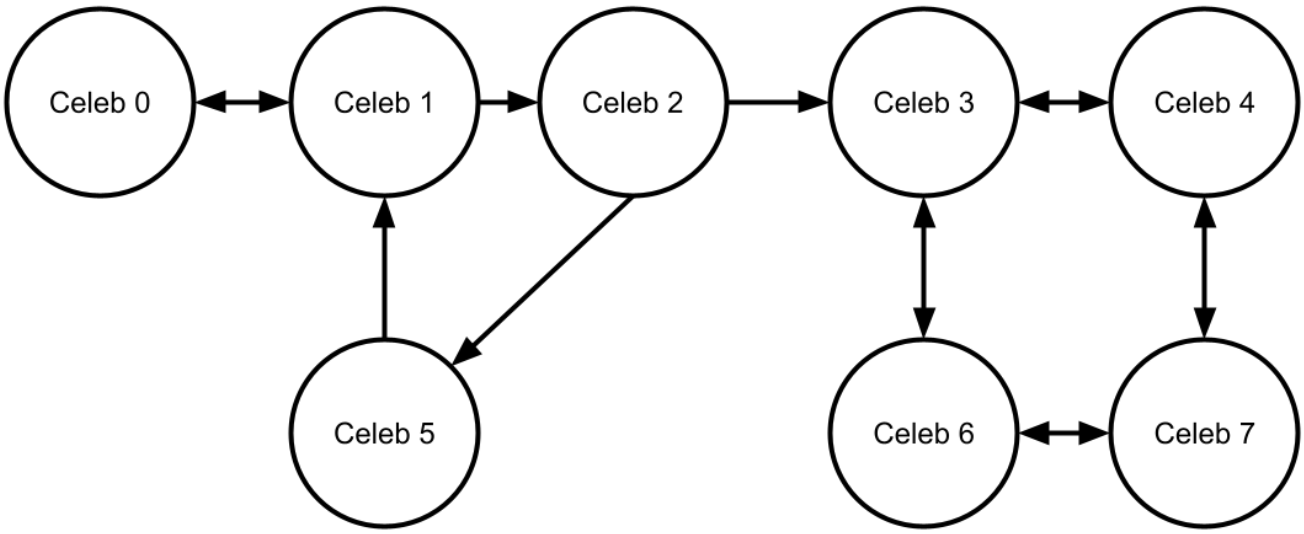
Problem 7: Casting Call Search II

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`.

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

Example Usage:




```
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**

This problem requires you to perform a depth first search traversal of a graph. If you need a primer on how to perform DFS on a graph, check out the unit cheatsheet.

Problem 8: 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.

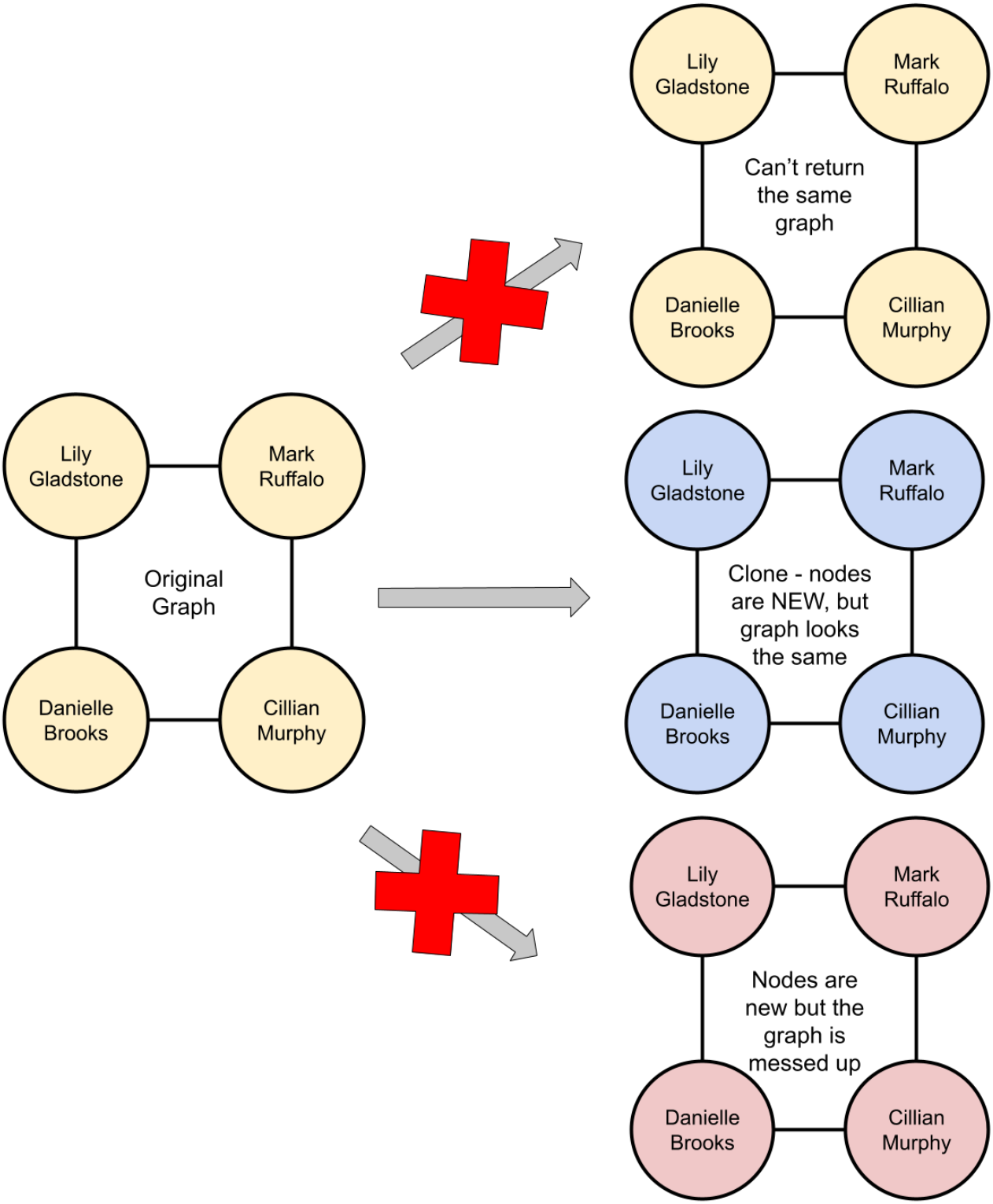
Given a reference to a `Node` in the original seating arrangement `seat`, make a deep copy (clone) of the seating arrangement. Return the copy of the given node.

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):
```

```
def copy_seating(seat):
    pass
```

Example Usage:




```
lily = Node("Lily Gladstone")
```

Path Courses

```
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
```

Close Section

▼ Advanced Problem Set Version 1

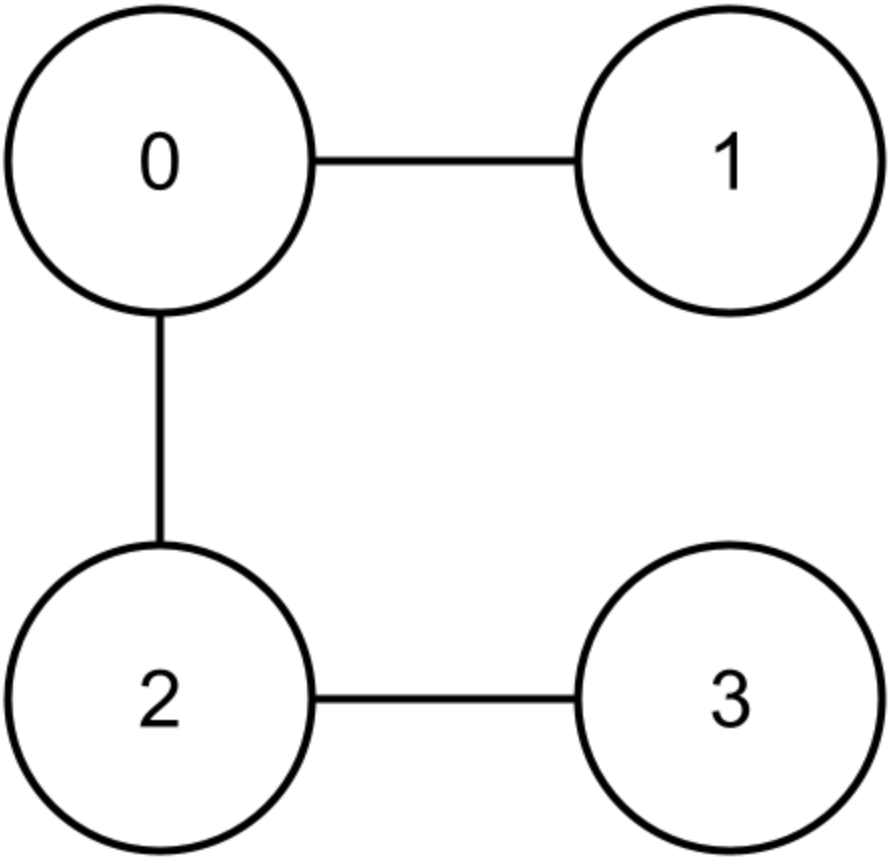
Problem 1: There and Back

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.

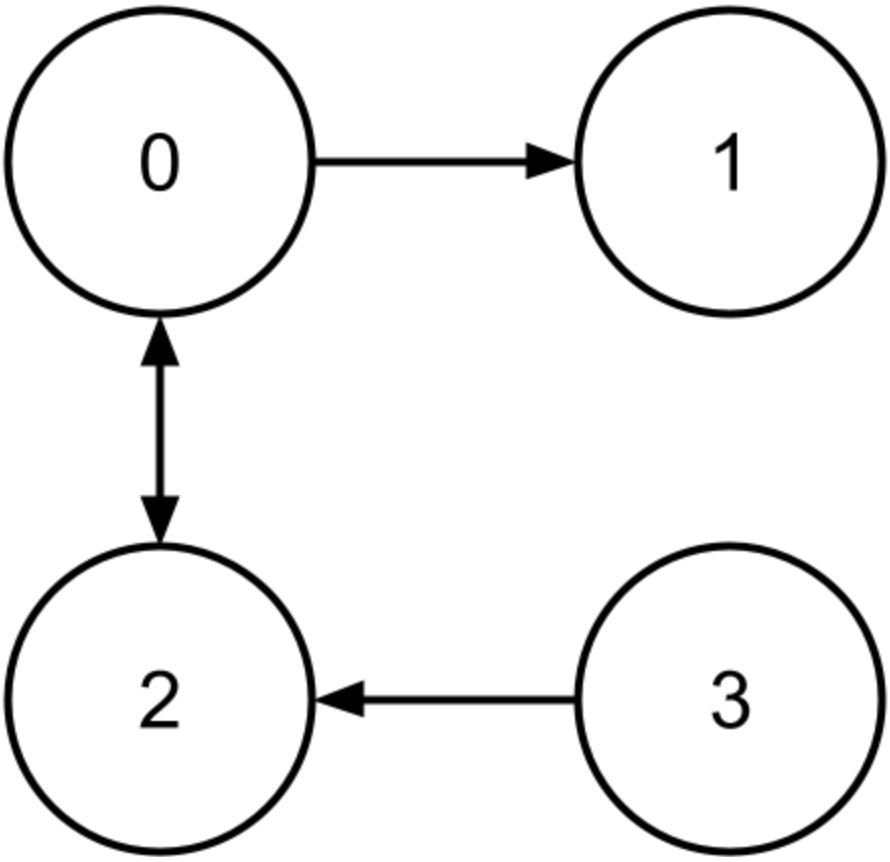
```
def bidirectional_flights(flights):
    pass
```

Example Usage:

Example 1: `flights1`



Example 2: `flights2`



Path Courses

```
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**

This problem requires you to be familiar with the graph data structure and the different methods for representing graphs. Check out the Unit 10 Cheatsheet if you are unfamiliar with these concepts.

Problem 2: Find Center of Airport

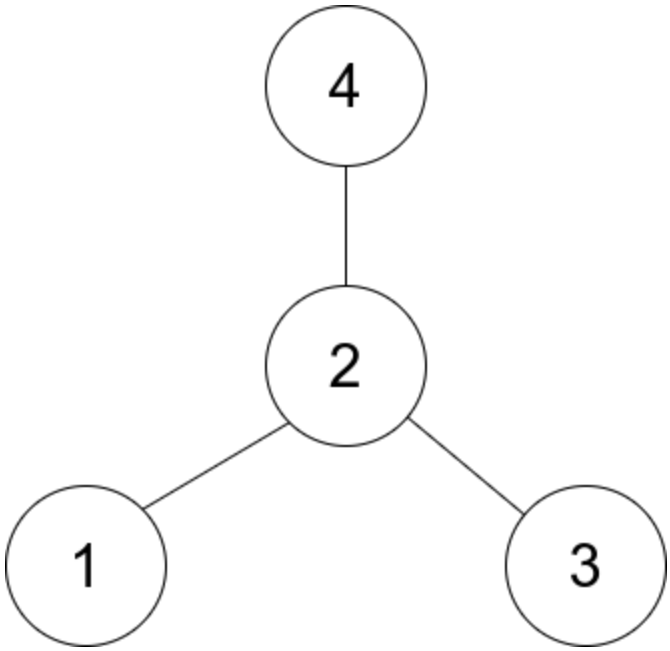
You are a pilot navigating a new airport and have a map of the airport represented as an undirected star graph with `n` nodes where each node represents a terminal in the airport labeled from `1` to `n`. You want to find the center terminal in the airport where the pilots' lounge is located.

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.

```
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**

Observe that in a star graph the center node is connected to all other nodes. It must appear in all but one of the edges.

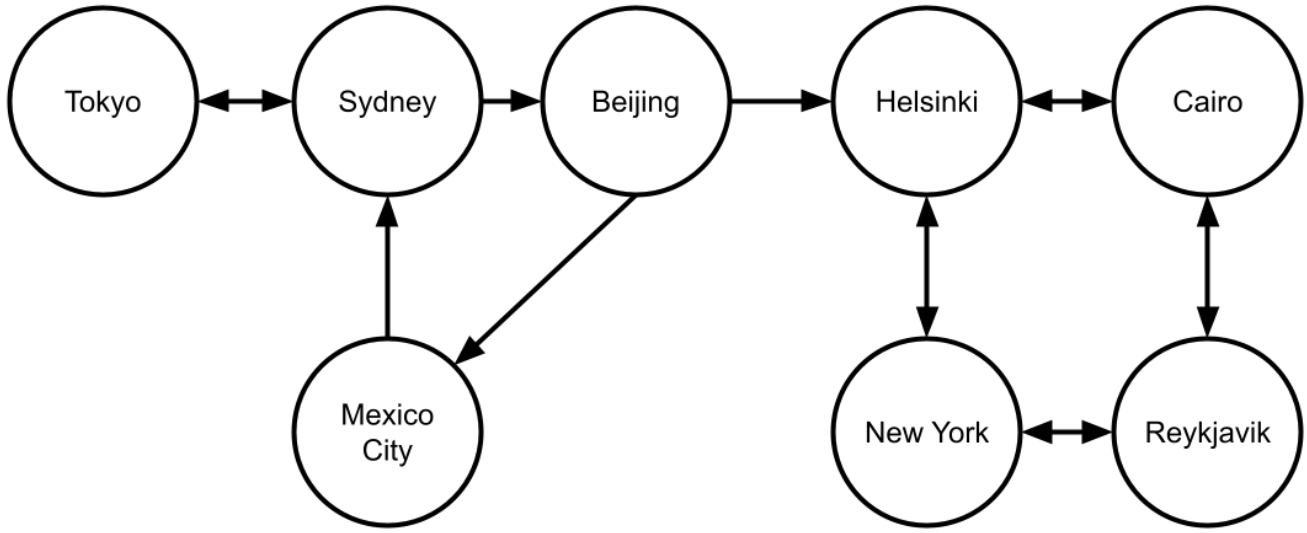
Problem 3: Finding All Reachable Destinations

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, 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"],
    "New York": []
}

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']
```

▼ ✨ AI Hint: Breadth First Search Traversal

Key Skill: Use AI to explain code concepts

To solve this problem, it may be helpful to understand both the **queue** data structure and **breadth first search** algorithm. To learn more about these concepts, visit the Queues and Breadth First Search sections of the Cheatsheet.

"I'm trying to understand how to use a queue to implement breadth first search, but I'm confused about why I wouldn't just use a list. Can you explain the difference?"

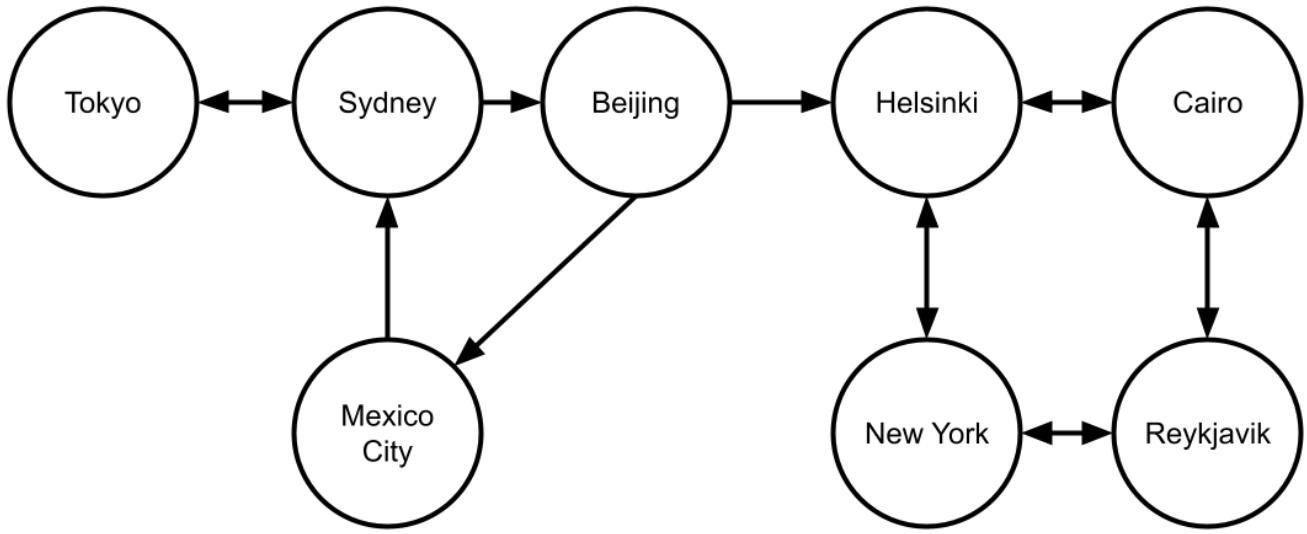
Problem 4: Finding All Reachable Destinations II

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"]
}
```

Example Output:

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

▼ 💡 **Hint: Depth First Search**

This problem requires you to perform a depth first search traversal of a graph. If you need a primer on how to perform DFS on a graph, check out the unit cheatsheet.

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).

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

Example Usage:

```
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))
```

Example Output:

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

▼ 💡 **Hint: Pseudocode**

One possible approach to this problem is to use a dictionary.

1. Create a dictionary that maps each deaprture airport to its corresponding arrival airport for efficient lookup.
2. Identify the starting airport. It is the only airport that is only a departure airport and never an arrival airport.
3. Trace the itinerary by following the mapping from departure to arrival until there are no more flights.

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.

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

Example Usage:

```
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:

```
['LHR', 'DFW', 'JFK', 'LAX', 'DXB']
```

▼ 💡 **Hint: Pseudocode**

One possible approach to this problem is to use Depth First Search. To use DFS:

1. Create an adjacency list where each airport is a key and its corresponding value is a list of destinations (flights) from that airport.
2. Identify the starting airport. It is the only airport that is only a departure airport and never an arrival airport.
3. Using the starting airport as your start point, begin a DFS traversal of the adjacency list. After visiting *all* destinations for a given airport, add the airport to the itinerary.
4. Since airports are added only after visiting all connected destinations, the resulting itinerary is in reverse order. Reverse the itinerary and return the result.

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`.

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

Example Usage:

```
# 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))
```

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?**

This problem requires you to use either BFS or DFS. But which should you choose? Check out the *BFS vs DFS* section of the unit cheatsheet or conduct your own research to determine which algorithm would best suit this problem.

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.

```
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],
```

```
    [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

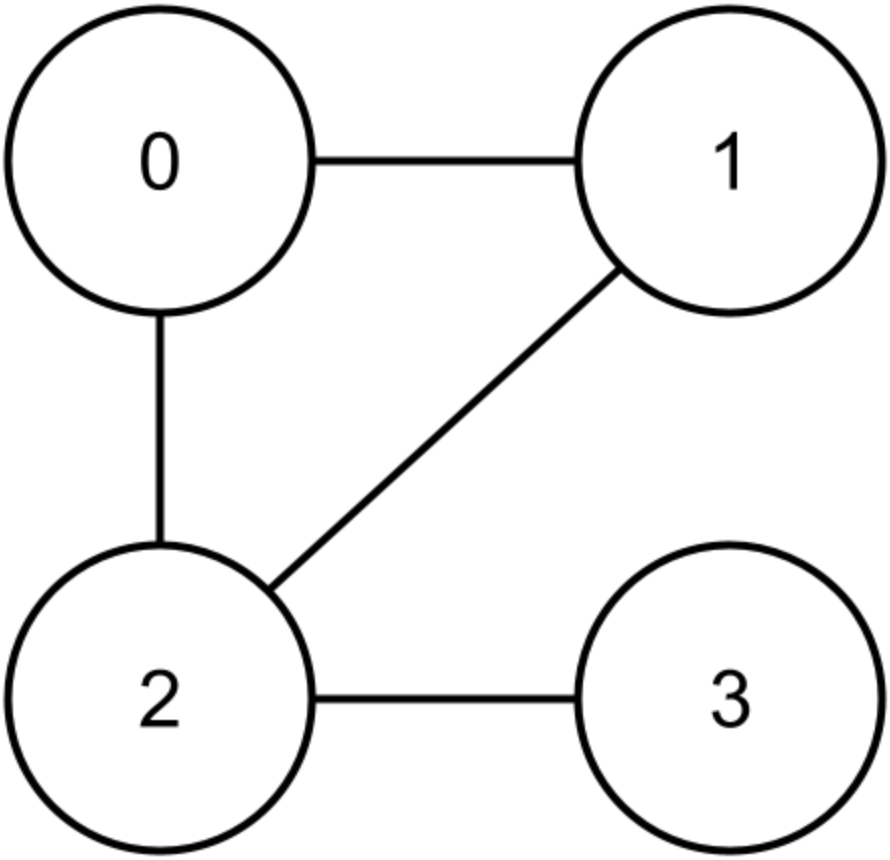
Problem 1: The Feeling is Mutual

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`.

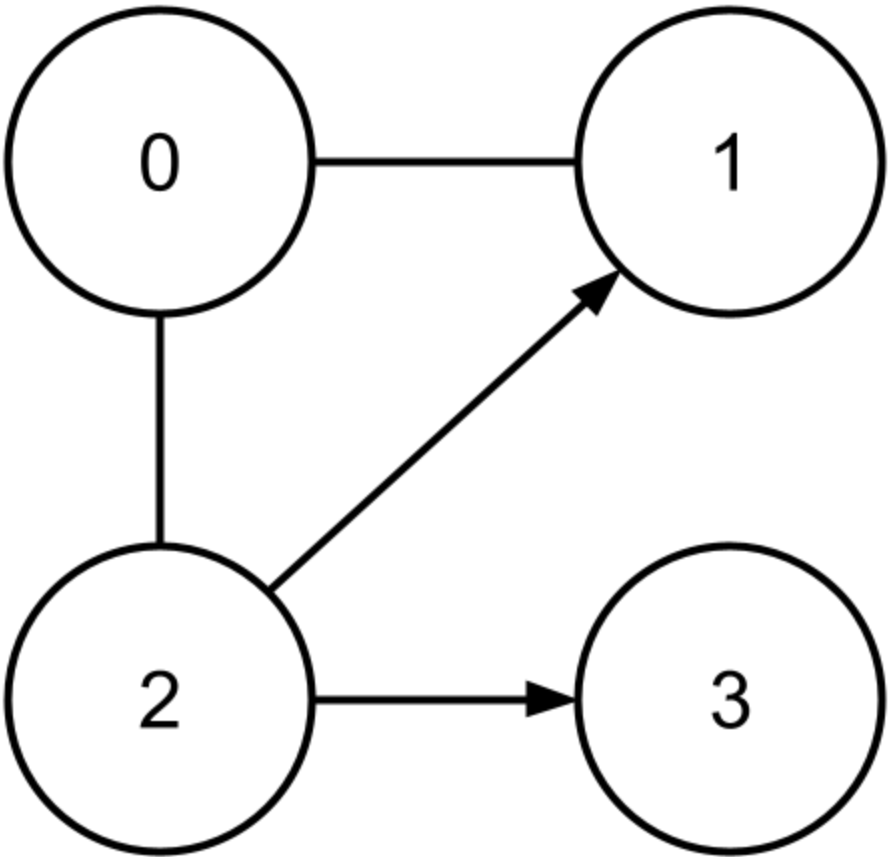
```
def is_mutual(celebrities):
    pass
```

Example Usage:

Example 1: `celebrities1`



Example 2: `celebrities2`



Path Courses

```
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**

This problem requires you to be familiar with the graph data structure and the different methods for representing graphs. Check out the Unit 10 Cheatsheet if you are unfamiliar with these concepts.

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.

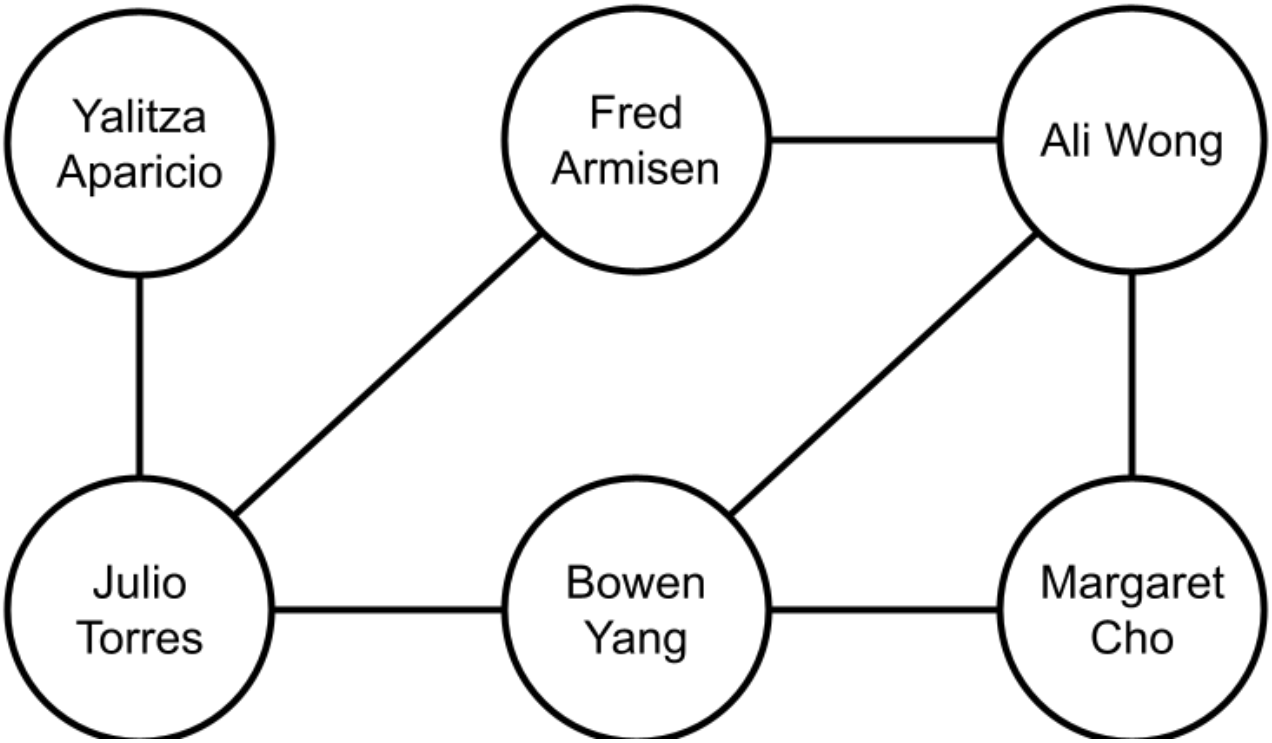
Given `contacts`:

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.

```
def get_adj_matrix(clients):
```

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
```

```
  [1, 1, 0, 1, 0, 0]   # Julio Torres
]

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

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

This problem requires you to convert between two different graph representations. Converting between graph representations is a common subproblem when solving more advanced problems. This is especially true when you are given a list of edges and need to easily find a node's neighbors.

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`.

```
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)
```

Example Output:

```
2
3
-1
```

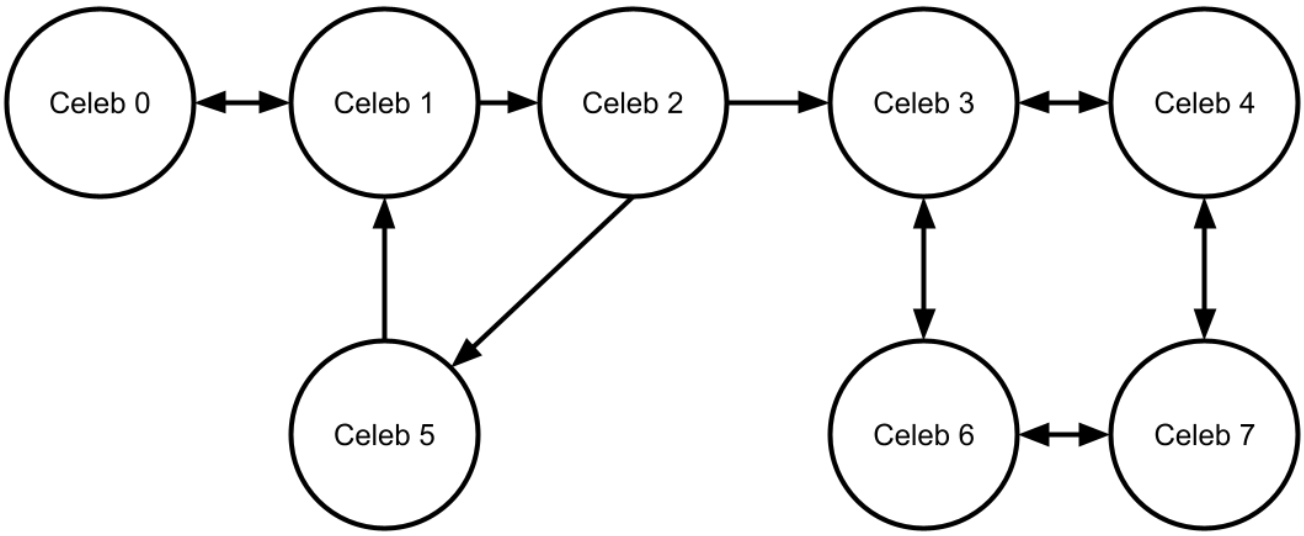
Problem 4: Casting Call Search

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`.

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

Example Usage:




```
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
```

▼ ✨ AI Hint: Breadth First Search Traversal

Key Skill: Use AI to explain code concepts

To solve this problem, it may be helpful to understand both the **queue** data structure and **breadth first search** algorithm. To learn more about these concepts, visit the Queues and Breadth First Search sections of the Cheatsheet.

If you still have questions, try explaining what you're doing, and ask an AI tool like ChatGPT or GitHub Copilot to help you understand what's confusing you. For example, you might ask:

"I'm trying to understand how to use a queue to implement breadth first search, but I'm confused about why I wouldn't just use a list. Can you explain the difference?"

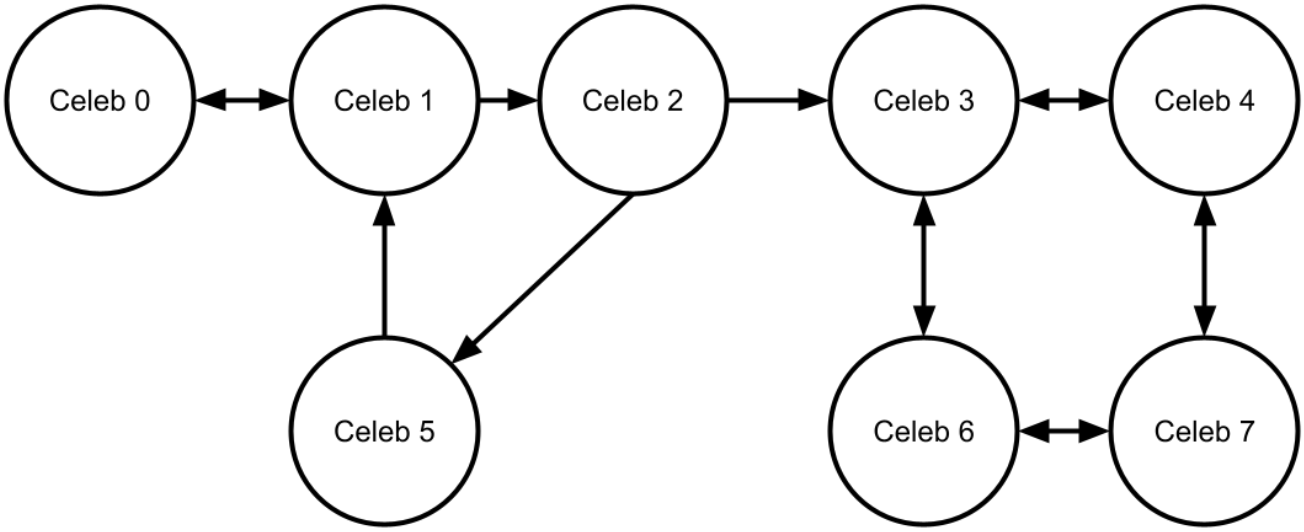
Problem 5: Casting Call Search II

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`.

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

Example Usage:




```
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**

This problem requires you to perform a depth first search traversal of a graph. If you need a primer on how to perform DFS on a graph, check out the unit cheatsheet.

Problem 6: Copying Seating Arrangements

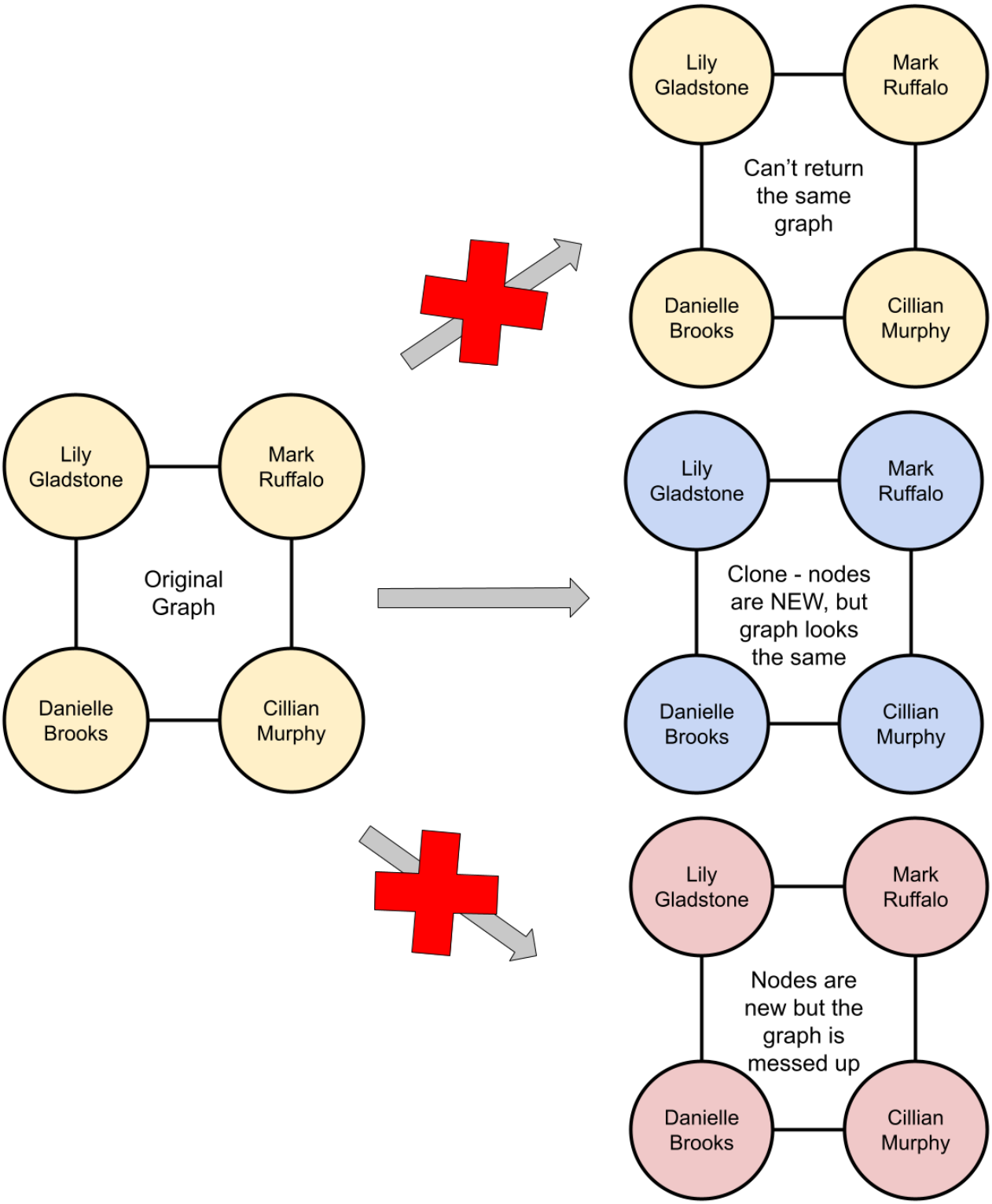
You are organizing the seating arrangement for a big awards ceremony and want to make a copy for your assistant. The seating arrangement is stored in a graph where each `Node` value `val` is the

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 = 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])
```

```
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)`.

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

Example Usage:

```
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),
```

```
    "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.

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

Example Usage:

```
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