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 2: Graphs

Session Overview

In this session, students will continue to explore using Breadth First Search (BFS) and Depth First Search (DFS) algorithms to solve common graph problems. They will learn to manipulate the base algorithm and explore the different use cases of these two traversal algorithms.

> 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**

🔎 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: Can Rebook Flight

Oh no! You're flight has been cancelled and you need to rebook. Given an adjacency matrix of today's flights `flights` where each flight is labeled `0` to `n-1` and `flights[i][j] = 1` indicates that there is an available flight from location `i` to location `j`, return `True` if there exists a path from your current location `source` to your final destination `dest`. Otherwise return `False`.

Evaluate the time complexity of your function. Define your variables and provide a rationale for why you believe your solution has the stated time complexity.

```
def can_rebook(flights, source, dest):
    pass
```

Example Usage:

```
flights1 = [
    [0, 1, 0], # Flight 0
```

```
]

flights2 = [
    [0, 1, 0, 1, 0],
    [0, 0, 0, 1, 0],
    [0, 0, 0, 0, 1],
    [0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0]
]

print(can_rebook(flights1, 0, 2))
print(can_rebook(flights2, 0, 2))
```

Example Output:

```
True
False
```

▼ 💡 **Hint: Graph Traversal Algorithms**

This problem requires you to use one of the two graph traversal algorithms, Breadth First Search or Depth First Search. If you need an introduction to these two algorithms, check out the unit cheatsheet.

Problem 2: Can Rebook Flight II

If you solved the above problem `can_rebook()` using Breadth First Search, try solving it using Depth First Search. If you solved it using Depth First Search, solve it using Breadth First Search.

Evaluate the time complexity of your function. Define your variables and provide a rationale for why you believe your solution has the stated time complexity.

```
def can_rebook(flights, source, dest):
    pass
```

Example Usage:

```
flights1 = [
    [0, 1, 0], # Flight 0
    [0, 0, 1], # Flight 1
    [0, 0, 0]  # Flight 2
]

flights2 = [
```

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

print(can_rebook(flights1, 0, 2))
print(can_rebook(flights2, 0, 2))
```

Example Output:

```
True
False
```

Problem 3: 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] = 1` 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))
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
```

▼ 💡 **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 4: 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],
    [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
```

▼ 💡 **Hint: Finding Components**

To find disconnected components in a graph, we need to perform the base BFS/DFS algorithm multiple times. BFS and DFS will only find *reachable* nodes from the start node. To traverse all nodes in the graph, including nodes unconnected to the start node, we must run the algorithm multiple times. We can do this by checking which nodes have not yet been visited after our initial execution of the algorithm, and choose an unvisited node as our new starting node. We can continue this pattern until we find all nodes have been visited.

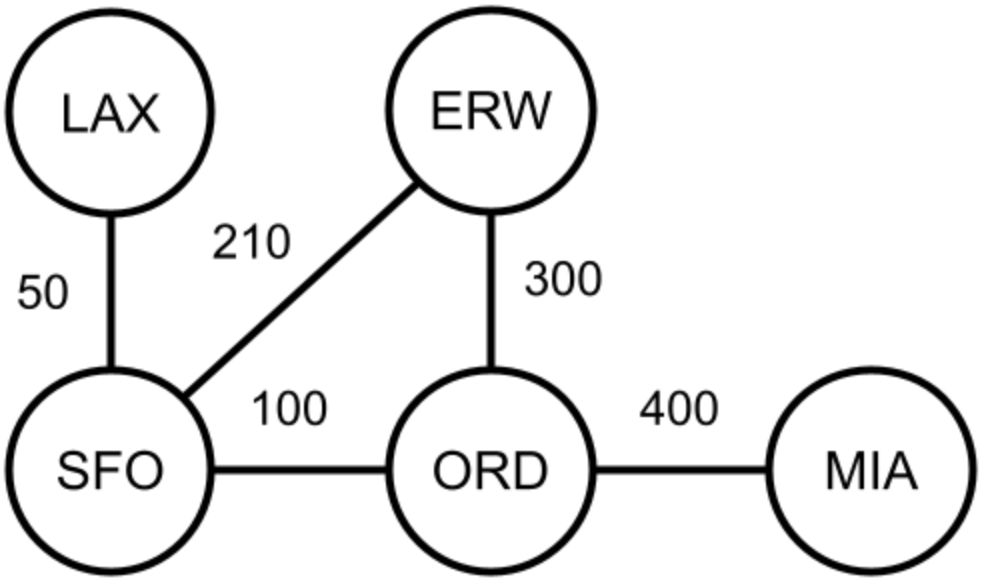
Problem 5: Get Flight Cost

You are given an adjacency dictionary `flights` where for any location `source`, `flights[source]` is a list of tuples in the form `(destination, cost)` indicating that there exists a flight from `source` to `destination` at ticket price `cost`.

Given a starting location `start` and a final destination `dest` return the total cost of flying from `start` to `dest`. If it is not possible to fly from `start` to `dest`, return `-1`. If there are multiple possible paths from `start` to `dest`, return any of the possible answers.

```
def calculate_cost(flights, start, dest):
    pass
```

Example Usage:



Path Courses

```
flights = {
    'LAX': [('SFO', 50)],
    'SFO': [('LAX', 50), ('ORD', 100), ('ERW', 210)],
    'ERW': [('SFO', 210), ('ORD', 100)],
    'ORD': [('ERW': 300), ('SFO', 100), ('MIA', 400)],
    'MIA': [('ORD', 400)]
}

print(calculate_cost(flights, 'LAX', 'MIA'))
```

Example Output:

```
550
Explanation: There is a path from LAX -> SFO -> ORD -> MIA with ticket prices 50 + 100 + 400
960 would also be an acceptable answer following the path from LAX -> SFO -> ERW -> ORD -> MI
```

▼ ✨ **AI Hint: Weighted Graphs**

Key Skill: Use AI to explain code concepts

This problem requires you to be familiar with weighted graphs. Learn more quickly by referencing the Unit 10 Cheatsheet

To explore further, try asking an AI tool like ChatGPT or GitHub Copilot to explain the concept of weighted graphs, how they work, and how to implement them in Python.

Problem 6: Fixing Flight Booking Software

CodePath Airlines uses Breadth First Search to suggest the route with the least number of layovers to its customers. But their software has a bug and is malfunctioning. Help the airline by identifying and fixing the bug.

When properly implemented, the function should accept an adjacency dictionary `flights` and returns a list with the shortest path from a `source` location to a `destination` location.

For this problem:

1. Identify and fix any bug(s) in the code.
2. Evaluate the time complexity of the function. Evaluate the time complexity of your function. Define your variables and provide a rationale for why you believe your solution has the stated time complexity.

```
from collections import deque

def find_shortest_path(flights, source, destination):
    queue = deque([(source, [])])
    visited = set()

    while queue:
        current, path = queue.popleft()

        if current == destination:
            return path

        visited.add(current)

        for neighbor in flights.get(current, []):
            if neighbor not in visited:
                queue.append((neighbor, [neighbor]))

    return []
```

Example Usage:

```
flights = {
    'LAX': ['SFO'],
    'SFO': ['LAX', 'ORD', 'ERW'],
    'ERW': ['SFO', 'ORD'],
    'ORD': ['ERW', 'SFO', 'MIA'],
    'MIA': ['ORD']
}

print(find_shortest_path(flights, 'LAX', 'MIA'))
```

Expected Output:

```
['LAX', 'SFO', 'JFK', 'MIA']
```

Problem 7: Expanding Flight Offerings

CodePath Airlines wants to expand their flight offerings so that for any airport they operate out of, it is possible to reach all other airports. They track their current flight offerings in an adjacency dictionary `flights` where each key is an airport `i` and `flights[i]` is an array indicating that there is a flight from destination `i` to each destination in `flights[i]`. Assume that if there is flight from airport `i` to airport `j`, the reverse is also true.

Given `flights`, return the minimum number of flights (edges) that need to be added such that there is flight path from each airport in `flights` to every other airport.

```
def min_flights_to_expand(flights):
```

Example Usage:

```
flights = {
    'JFK': ['LAX', 'SFO'],
    'LAX': ['JFK', 'SFO'],
    'SFO': ['JFK', 'LAX'],
    'ORD': ['ATL'],
    'ATL': ['ORD']
}

print(min_flights_to_expand(flights))
```

Example Output:

```
1
```

Problem 8: Get Flight Itinerary

Given an adjacency dictionary of flights `flights` where each key is an airport `i` and `flights[i]` is an array indicating that there is a flight from destination `i` to each destination in `flights[i]`, return an array with the flight path from a given `source` location to a given `destination` location.

If there are multiple flight paths from the `source` to `destination`, return any flight path.

```
def get_itinerary(flights, source, dest):
    pass
```

Example Usage:

```
flights = {
    'LAX': ['SFO'],
    'SFO': ['LAX', 'ORD', 'ERW'],
    'ERW': ['SFO', 'ORD'],
    'ORD': ['ERW', 'SFO', 'MIA'],
    'MIA': ['ORD']
}

print(get_itinerary(flights, 'LAX', 'MIA'))
```

Example Output:

```
['LAX', 'SFO', 'ORD', 'MIA']
Explanation: ['LAX', 'SFO', 'ERW', 'ORD', 'MIA'] is also a valid answer
```

▼ 💡 **Hint: Path Reconstruction**

This problem requires you to reconstruct the path taken by either BFS or DFS. To reconstruct

discovered) during the search. After reaching the target, backtrack from the target node to the start using the parent pointers to trace the path.

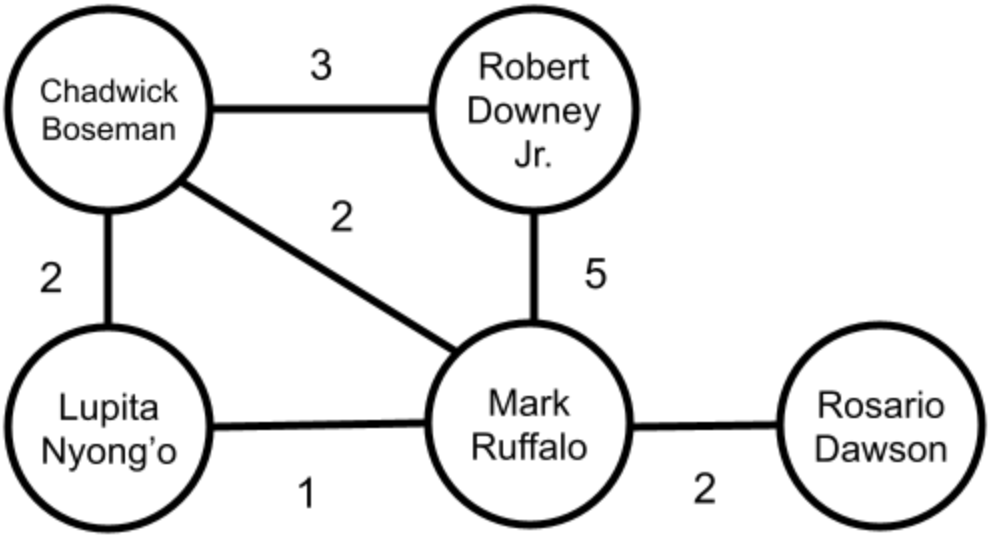
Close Section

▼ Standard Problem Set Version 2

Problem 1: Celebrity Collaborations

In the graph depicted below, each vertex represents a different actor and each undirected edge indicates that they have costarred together in one or more films. The weight of each edge represents the number of films they have costarred in together.

Build an adjacency dictionary `collaborations` that represents the given graph. Each key in the dictionary should be a string representing a actor in the graph, and each corresponding value a list of tuples where `collaborations[actor][i] = (costar, num_collaborations)`.




```
# There is no starter code for this problem
# Add code to build your graph here
```

Example Usage:

```
print(collaborations["Chadwick Boseman"])
```

Example Output:

```
[("Lupita Nyong'o", 2), ("Robert Downey Jr.", 3), ("Mark Ruffalo", 2)]
```

▼ ✨ AI Hint: Weighted Graphs

This problem requires you to be familiar with weighted graphs. Learn more quickly by referencing the Unit 10 Cheatsheet

To explore further, try asking an AI tool like ChatGPT or GitHub Copilot to explain the concept of weighted graphs, how they work, and how to implement them in Python.

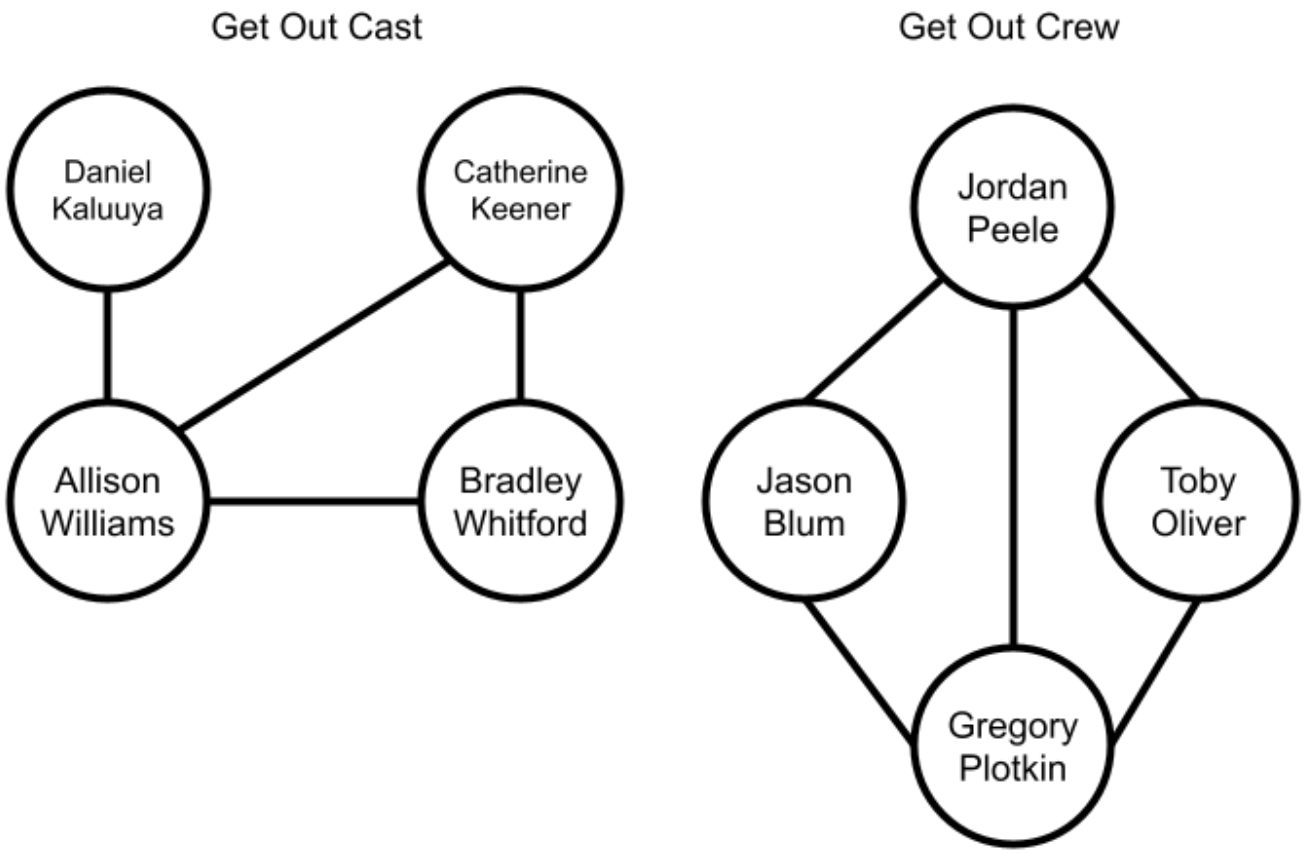
Problem 2: Cast vs Crew

You are given an adjacency list `cast_and_crew` where each node represents a cast or crew member of a particular movie. There exists a path from every cast member to every other cast member in the cast. There also exists a path from every crew member to every other crew member in the crew. Cast and crew are not connected by any edges.

Using Depth First Search, return two lists, one with all cast members in `cast_and_crew`, and a second with all crew members in `cast_and_crew`. You may return the two lists in any order.

```
def get_groups(cast_and_crew):
    pass
```

Example Usage:




```
get_out_movie = {
```

Path Courses

```
    "Bradley Whitford": ["Allison Williams", "Catherine Keener"],
    "Catherine Keener": ["Allison Williams", "Bradley Whitford"],
    "Jordan Peele": ["Jason Blum", "Gregory Plotkin", "Toby Oliver"],
    "Toby Oliver": ["Jordan Peele", "Gregory Plotkin"],
    "Gregory Plotkin": ["Jason Blum", "Toby Oliver", "Jordan Peele"],
    "Jason Blum": ["Jordan Peele", "Gregory Plotkin"]
}

print(get_groups(cast_and_crew))
```

Example Output:

```
[
    ['Daniel Kaluuya', 'Allison Williams', 'Catherine Keener', 'Bradley Whitford'],
    ['Jordan Peele', 'Jason Blum', 'Gregory Plotkin', 'Toby Oliver']
]
```

▼ 💡 **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.

▼ 💡 **Hint: Finding Components**

To find disconnected components in a graph, we need to perform the base BFS/DFS algorithm multiple times. BFS and DFS will only find *reachable* nodes from the start node. To traverse all nodes in the graph, including nodes unconnected to the start node, we must run the algorithm multiple times. We can do this by checking which nodes have not yet been visited after our initial execution of the algorithm, and choose an unvisited node as our new starting node. We can continue this pattern until we find all nodes have been visited.

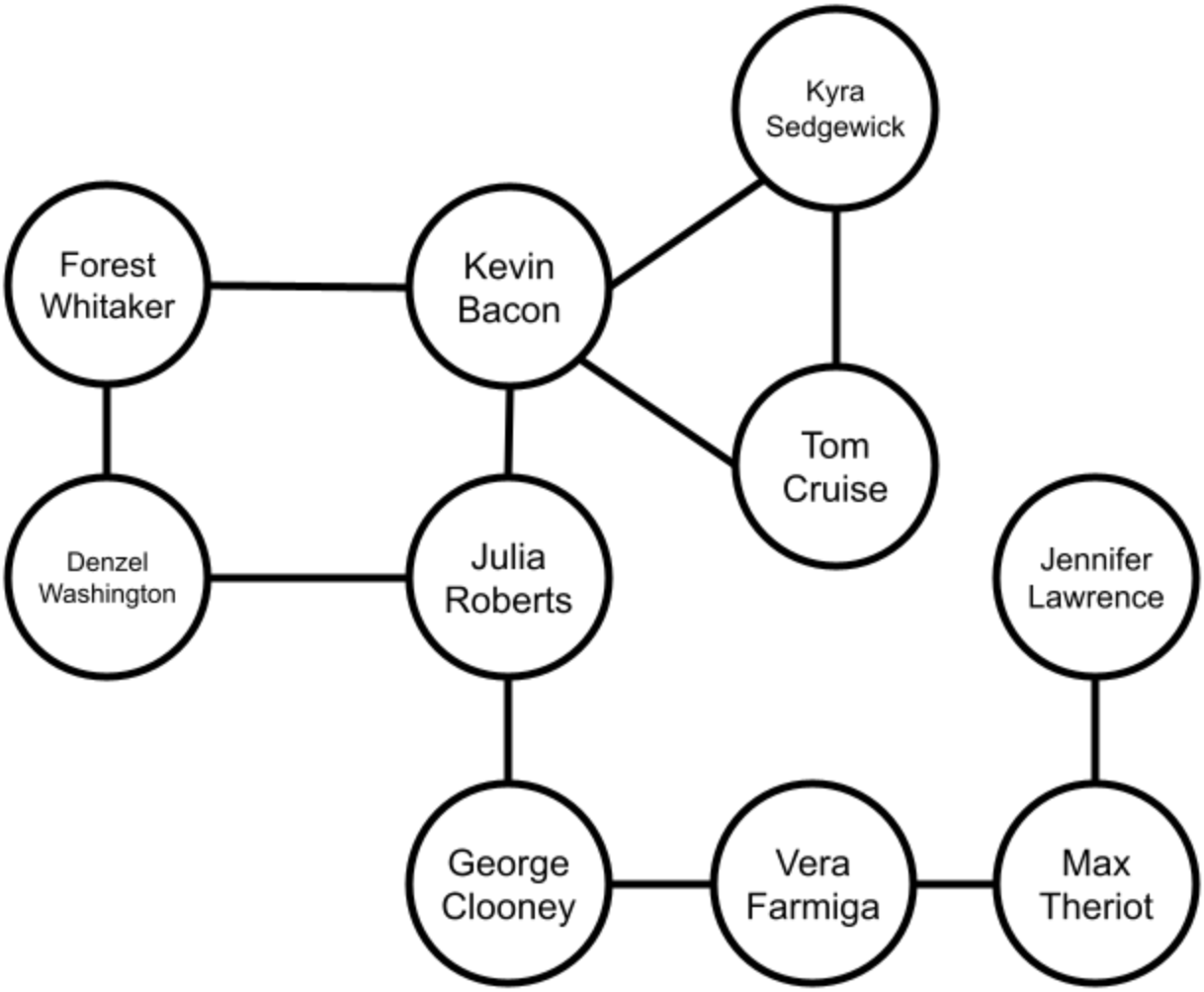
Problem 3: Bacon Number

Six Degrees of Kevin Bacon is a game where you try to find a path of mutual connections between some actor or person to the actor Kevin Bacon in six steps or less. You are given an adjacency dictionary `bacon_network`, where each key represents an `actor` and the corresponding list `bacon_network[actor]` represents an actor they have worked with. Given a starting actor `celeb`, find their Bacon Number. `'Kevin Bacon'` is guaranteed to be a vertex in the graph.

- Kevin Bacon himself has a Bacon Number of `0`.
- Actors who have worked directly with Kevin Bacon have a Bacon Number of `1`.
- If an individual has worked with `actor_b` and `actor_b` has a Bacon Number of `n`, the individual has a Bacon Number of `n+1`.
- If an individual cannot be connected to Kevin Bacon through a path of mutual connections, their Bacon Number is `-1`.

```
def bacon_number(bacon_network, celeb):
    pass
```

Example Usage:




```
bacon_network = {
    "Kevin Bacon": ["Kyra Sedgewick", "Forest Whitaker", "Julia Roberts", "Tom Cruise"],
    "Kyra Sedgewick": ["Kevin Bacon"],
    "Tom Cruise": ["Kevin Bacon", "Kyra Sedgewick"]
    "Forest Whitaker": ["Kevin Bacon", "Denzel Washington"],
    "Denzel Washington": ["Forest Whitaker", "Julia Roberts"],
    "Julia Roberts": ["Denzel Washington", "Kevin Bacon", "George Clooney"],
    "George Clooney": ["Julia Roberts", "Vera Farmiga"],
    "Vera Farmiga": ["George Clooney", "Max Theriot"],
```

```
}

print(bacon_number(bacon_network, "Jennifer Lawrence"))
print(bacon_number(bacon_network, "Tom Cruise"))
```

Example Output:

```
5
1
```

▼ ✨ 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?"

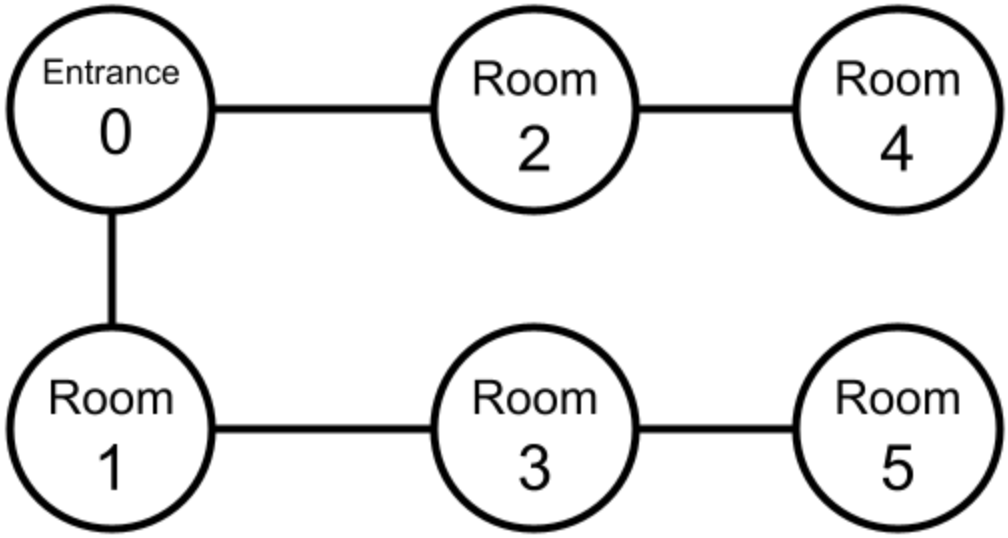
Problem 4: Press Junket Navigation

You've been invited to interview some of your favorite celebrities. Each group is stationed in a different room in the venue numbered `0` to `n-1`. To get to your assigned interview station, you need to navigate from the *entrance* which is room number `0` to your assigned room `target`.

Given an adjacency list `venue_map` where `venue_map[i]` indicates that there is a hallway between room `i` and each room in `venue_map[i]`, return a list representing the path from the entrance to your `target` room. If there are multiple paths, you may return any valid path.

```
def find_path(venue_map, target):
    pass
```

Example Usage:




```
venue_map = [
    [1, 2],
    [0, 3],
    [0, 4],
    [1, 5],
    [2],
    [3]
]

print(find_path(venue_map, 5))
print(find_path(venue_map, 2))
```

Example Output:

```
[0, 1, 3, 5]
[0, 2]
```

▼ 💡 **Hint: Path Reconstruction**

This problem requires you to reconstruct the path taken by either BFS or DFS. To reconstruct a path from BFS/DFS, we can keep track of each node's parent (the node from which it was discovered) during the search. After reaching the target, backtrack from the target node to the start using the parent pointers to trace the path.

Problem 5: 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

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),
    "Molly-Mae Hague": (3, 8),
    "Tommy Fury": (4, 7),
    "Ovie Soko": (5, 6),
    "Curtis Pritchard": (-1, -1),
    "Maura Higgins": (-1, -1)
}
```

Problem 6: 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
```

Problem 7: Maximizing Star Power

You are the producer of a big Hollywood film and want to maximize the star power of the cast. Each collaboration between two celebrities has a star power value. You want to maximize the total star power of the cast, while including two costars who have already signed onto the project `costar_a` and `costar_b`.

You are given a graph where:

- Each vertex represents a celebrity.
- Each edge between two celebrities represents a collaboration, with two weights:
    1. The star power (benefit) they bring when collaborating.
    2. The cost to hire them both for the project.

The graph is given as a dictionary `collaboration_map` where each key is a celebrity and the corresponding value is a list of tuples. Each tuple contains a connected celebrity, the star power of that collaboration, and the cost of the collaboration. Given `costar_a` and `costar_b`, return the maximum star power of any path between `costar_a` and `costar_b`.

Example Usage:

```
collaboration_map = {
    "Leonardo DiCaprio": [("Brad Pitt", 40), ("Robert De Niro", 30)],
    "Brad Pitt": [("Leonardo DiCaprio", 40), ("Scarlett Johansson", 20)],
    "Robert De Niro": [("Leonardo DiCaprio", 30), ("Chris Hemsworth", 50)],
    "Scarlett Johansson": [("Brad Pitt", 20), ("Chris Hemsworth", 30)],
    "Chris Hemsworth": [("Robert De Niro", 50), ("Scarlett Johansson", 30)]
}

print(find_max_star_power(collaboration_map, "Leonardo DiCaprio", "Chris Hemsworth"))
```

Example Output:

```
90
Explanation: The maximum star power path is from Leonardo DiCaprio -> Brad Pitt -> Scarlett J
(40 + 20 + 30 = 90).
The other path is Leonardo DiCaprio -> Robert De Niro -> Chris Hemsworth (30 + 50 = 80).
```

Problem 8: Celebrity Feuds

You are in charge of scheduling celebrity arrival times for a red carpet event. To make things easy, you want to split the group of `n` celebrities labeled from `1` to `n` into two different arrival groups.

However, your boss has just informed you that some celebrities don't get along, and celebrities who dislike each other may not be in the same arrival group. Given the number of celebrities who will be attending `n`, and an array `dislikes` where `dislikes[i] = [a, b]` indicates that the person labeled `a` does not get along with the person labeled `b`, return `True` if it is possible to split the celebrities into two arrival periods and `False` otherwise.

```
def can_split(n, dislikes):
    pass
```

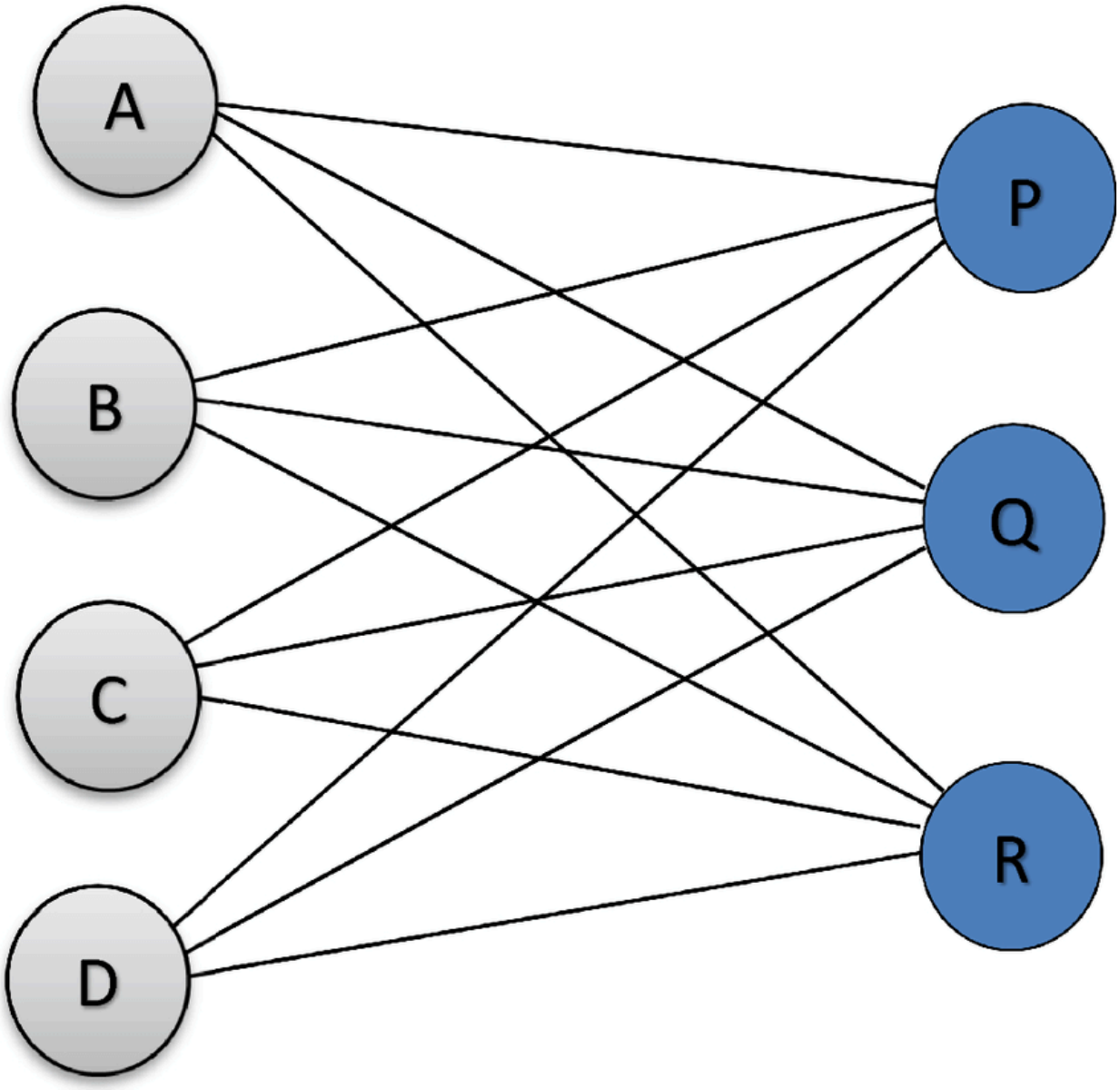
Example Usage:

```
dislikes_1 = [[1, 2], [1, 3], [2, 4]]
dislikes_2 = [[1, 2], [1, 3], [2, 3]]

print(can_split(4, dislikes_1))
print(can_split(3, dislikes_2))
```

Example Output:

```
True
False
```

This problem requires you to determine whether a graph is bipartite. A bipartite graph is a graph where the nodes can e divided into two distinct sets such that no two vertices in the saem set are connected by an edge. All edges must go between vertices in different sets.



We can determine whether a graph is bipartite using a technique called *graph coloring*. To determine if a graph is bipartite, we try coloring the graph using two colors: start from any node, color it one color, and color all its neighbors the opposite color. If at any point two adjacent nodes have the same color, the graph is not bipartite. You can do this using either BFS or DFS.

Close Section

▾ Advanced Problem Set Version 1

Path Courses

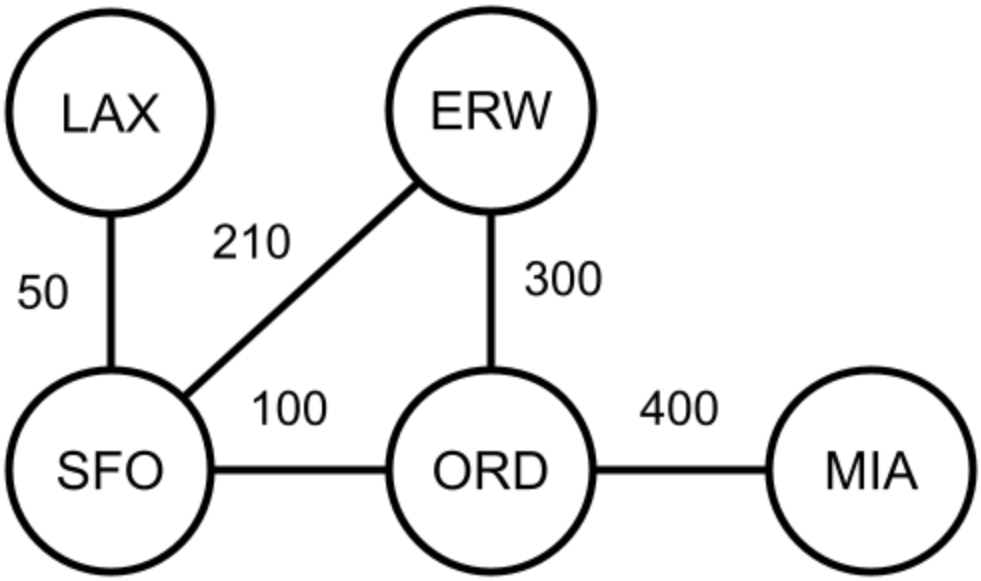
Problem 1: Get Flight Cost

You are given an adjacency dictionary `flights` where for any location `source`, `flights[source]` is a list of tuples in the form `(destination, cost)` indicating that there exists a flight from `source` to `destination` at ticket price `cost`.

Given a starting location `start` and a final destination `dest` return the total cost of flying from `start` to `dest`. If it is not possible to fly from `start` to `dest`, return `-1`. If there are multiple possible paths from `start` to `dest`, return any of the possible answers.

```
def calculate_cost(flights, start, dest):
    pass
```

Example Usage:




```
flights = {
    'LAX': [('SFO', 50)],
    'SFO': [('LAX', 50), ('ORD', 100), ('ERW', 210)],
    'ERW': [('SFO', 210), ('ORD', 100)],
    'ORD': [('ERW': 300), ('SFO', 100), ('MIA', 400)],
    'MIA': [('ORD', 400)]
}

print(calculate_cost(flights, 'LAX', 'MIA'))
```

Example Output:

```
550
Explanation: There is a path from LAX -> SFO -> ORD -> MIA with ticket prices 50 + 100 + 400
960 would also be an acceptable answer following the path from LAX -> SFO -> ERW -> ORD -> MI
```

▼ ✨ AI Hint: Weighted Graphs

Key Skill: Use AI to explain code concepts

This problem requires you to be familiar with weighted graphs. Learn more quickly by referencing the Unit 10 Cheatsheet

To explore further, try asking an AI tool like ChatGPT or GitHub Copilot to explain the concept of weighted graphs, how they work, and how to implement them in Python.

▼ ✨ 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 2: Expanding Flight Offerings

CodePath Airlines wants to expand their flight offerings so that for any airport they operate out of, it is possible to reach all other airports. They track their current flight offerings in an adjacency dictionary `flights` where each key is an airport `i` and `flights[i]` is an array indicating that there is a flight from destination `i` to each destination in `flights[i]`. Assume that if there is flight from airport `i` to airport `j`, the reverse is also true.

Given `flights`, return the minimum number of flights (edges) that need to be added such that there is flight path from each airport in `flights` to every other airport.

```
def min_flights_to_expand(flights):
    pass
```

Example Usage:

```
    'LAX': ['JFK', 'SFO'],
    'SFO': ['JFK', 'LAX'],
    'ORD': ['ATL'],
    'ATL': ['ORD']
}

print(min_flights_to_expand(flights))
```

Example Output:

```
1
```

Problem 3: Get Flight Itinerary

Given an adjacency dictionary of flights `flights` where each key is an airport `i` and `flights[i]` is an array indicating that there is a flight from destination `i` to each destination in `flights[i]`, return an array with the flight path from a given `source` location to a given `destination` location.

If there are multiple flight paths from the `source` to `destination`, return any flight path.

```
def get_itinerary(flights, source, dest):
    pass
```

Example Usage:

```
flights = {
    'LAX': ['SFO'],
    'SFO': ['LAX', 'ORD', 'ERW'],
    'ERW': ['SFO', 'ORD'],
    'ORD': ['ERW', 'SFO', 'MIA'],
    'MIA': ['ORD']
}

print(get_itinerary(flights, 'LAX', 'MIA'))
```

Example Output:

```
['LAX', 'SFO', 'ORD', 'MIA']
Explanation: ['LAX', 'SFO', 'ERW', 'ORD', 'MIA'] is also a valid answer
```

▼ 💡 **Hint: Path Reconstruction**

This problem requires you to reconstruct the path taken by either BFS or DFS. To reconstruct a path from BFS/DFS, we can keep track of each node's parent (the node from which it was discovered) during the search. After reaching the target, backtrack from the target node to the start using the parent pointers to trace the path.

Problem 4: Pilot Training

You are applying to become a pilot for CodePath Airlines, and you must complete a series of flight training courses. There are a total of `num_courses` flight courses you have to take, labeled from `0` to `num_courses - 1`. Some courses have prerequisites that must be completed before you can take the next one.

You are given an array `flight_prerequisites` where `flight_prerequisites[i] = [a, b]` indicates that you must complete course `b` first in order to take course `a`.

For example, the pair `["Advanced Maneuvers", "Basic Navigation"]` indicates that to take `"Advanced Maneuvers"`, you must first complete `"Basic Navigation"`.

Return `True` if it is possible to complete all flight training courses. Otherwise, return `False`.

```
def can_complete_flight_training(num_courses, flight_prerequisites):
    pass
```

Example Usage:

```
flight_prerequisites_1 = [['Advanced Maneuvers', 'Basic Navigation']]
flight_prerequisites_2 = [['Advanced Maneuvers', 'Basic Navigation'], ['Basic Navigation', 'A

print(can_complete_flight_training(2, flight_prerequisites_1))
print(can_complete_flight_training(2, flight_prerequisites_2))
```

Example Output:

```
True
Example 1 Explanation: There are 2 flight training courses. To take *Advanced Maneuvers*, you
False
Example 1 Explanation: There are 2 flight training courses. To take *Advanced Maneuvers*, you
```

▼ 💡 **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.

There are `n` airports numbered from `0` to `n - 1` and `n - 1` direct flight routes between airports such that there is exactly one way to travel between any two airports (this network forms a tree). Last year, the aviation authority decided to orient the flight routes in one direction due to air traffic regulations.

Flight routes are represented by `connections`, where `connections[i] = [airport_a, airport_b]` represents a one-way flight route from airport `airport_a` to airport `airport_b`.

This year, there will be a major aviation event at the central hub (airport `0`), and many flights need to reach this hub.

Your task is to reorient some flight routes so that every airport can send flights to airport `0`. Return the minimum number of flight routes that need to be reoriented.

It is guaranteed that after the reordering, each airport will be able to send a flight to airport `0`.

```
def min_reorient_flight_routes(n, connections):
    pass
```

Example Usage:

```
n = 6
connections = [[0, 1], [1, 3], [2, 3], [4, 0], [4, 5]]

print(min_reorient_flight_routes(n, connections))
```

Example Output:

```
3
Explanation:
- Initially, the flight routes are: 0 -> 1, 1 -> 3, 2 -> 3, 4 -> 0, 4 -> 5
- We need to reorient the routes [1, 3], [2, 3], and [4, 5] to ensure that every airport can
```

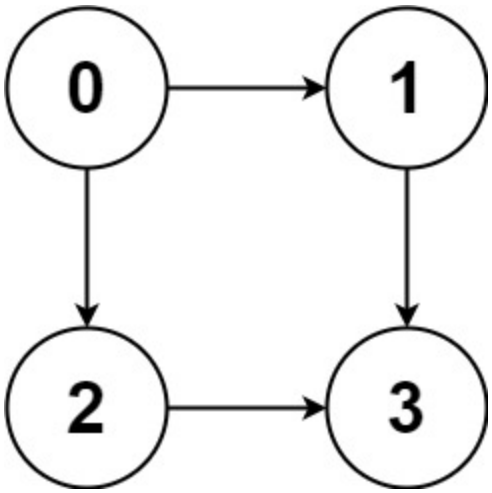
Problem 6: Find All Flight Routes

You are given a flight network represented as a directed acyclic graph (DAG) with `n` airports, labeled from `0` to `n - 1`. Your goal is to find all possible flight paths from airport `0` (the starting point) to airport `n - 1` (the final destination) and return them in any order.

The flight network is given as follows: `flight_routes[i]` is a list of all airports you can fly to directly from airport `i` (i.e., there is a one-way flight from airport `i` to airport `flight_routes[i][j]`).

Write a function that returns all possible flight paths from airport `0` to airport `n - 1`.

```
def find_all_flight_routes(flight_routes):
    pass
```

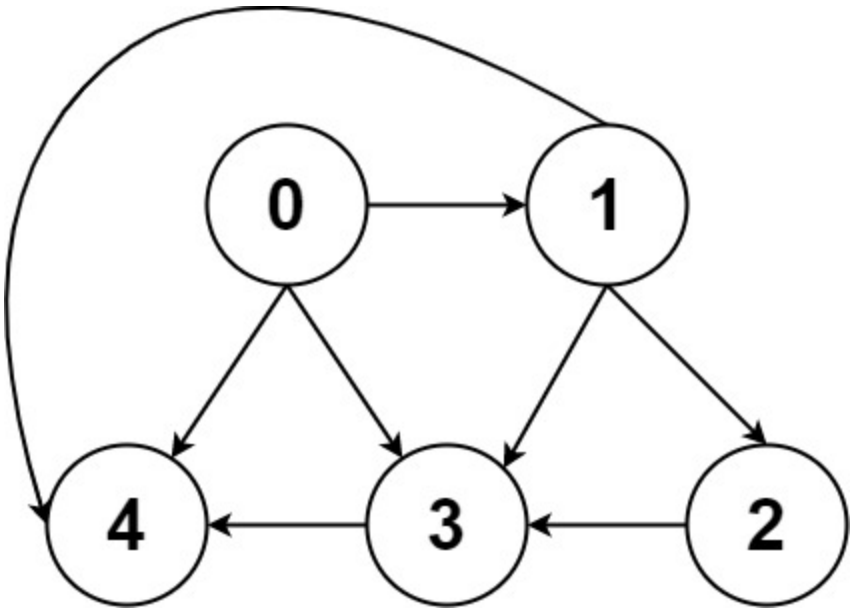



```
flight_routes_1 = [[1, 2], [3], [3], []]

print(find_all_flight_routes(flight_routes_1))
```

Example Output 1:

```
[[0, 1, 3], [0, 2, 3]]
Explanation:
- There are two possible paths from airport 0 to airport 3.
- The first path is: 0 -> 1 -> 3
- The second path is: 0 -> 2 -> 3
```

Example Usage 2:




```
flight_routes_2 = [[4,3,1],[3,2,4],[3],[4],[]]

print(find_all_flight_routes(flight_routes_2))
```

Example Output 2:

```
[[0,4],[0,3,4],[0,1,3,4],[0,1,2,3,4],[0,1,4]]
```

Close Section

Advanced Problem Set Version 2

Problem 1: Bacon Number

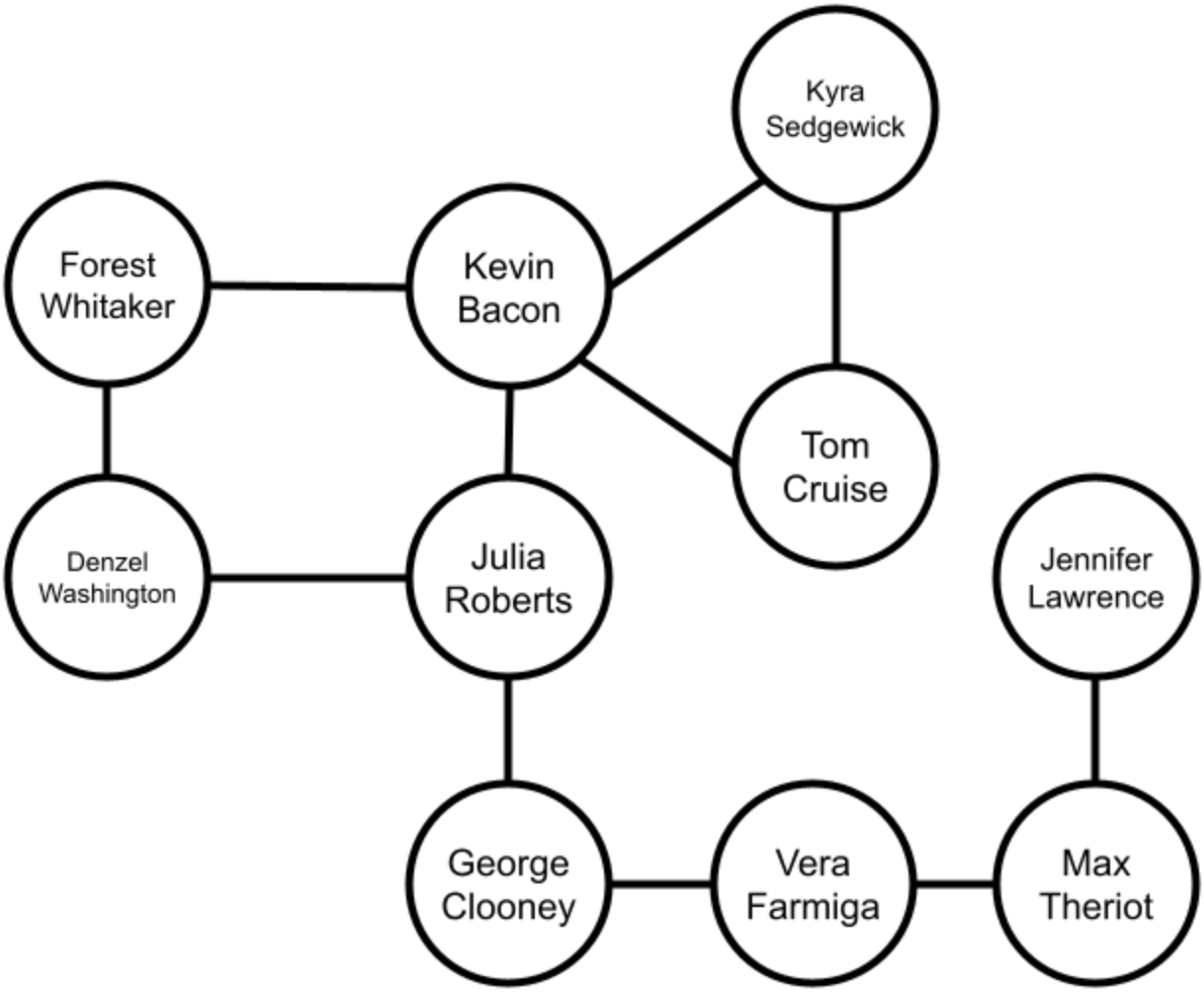
Six Degrees of Kevin Bacon is a game where you try to find a path of mutual connections between some actor or person to the actor Kevin Bacon in six steps or less. You are given an adjacency dictionary `bacon_network`, where each key represents an `actor` and the corresponding list `bacon_network[actor]` represents an actor they have worked with. Given a starting actor `celeb`, find their Bacon Number. `'Kevin Bacon'` is guaranteed to be a vertex in the graph.

To compute an individual's Bacon Number, assume the following:

- Kevin Bacon himself has a Bacon Number of `0`.
- Actors who have worked directly with Kevin Bacon have a Bacon Number of `1`.
- If an individual has worked with `actor_b` and `actor_b` has a Bacon Number of `n`, the individual has a Bacon Number of `n+1`.
- If an individual cannot be connected to Kevin Bacon through a path of mutual connections, their Bacon Number is `-1`.

```
def bacon_number(bacon_network, celeb):
    pass
```

Example Usage:




```
bacon_network = {
    "Kevin Bacon": ["Kyra Sedgewick", "Forest Whitaker", "Julia Roberts", "Tom Cruise"],
    "Kyra Sedgewick": ["Kevin Bacon"],
    "Tom Cruise": ["Kevin Bacon", "Kyra Sedgewick"]
    "Forest Whitaker": ["Kevin Bacon", "Denzel Washington"],
    "Denzel Washington": ["Forest Whitaker", "Julia Roberts"],
    "Julia Roberts": ["Denzel Washington", "Kevin Bacon", "George Clooney"],
    "George Clooney": ["Julia Roberts", "Vera Farmiga"],
    "Vera Farmiga": ["George Clooney", "Max Theriot"],
    "Max Theriot": ["Vera Farmiga", "Jennifer Lawrence"],
    "Jennifer Lawrence": ["Max Theriot"]
}

print(bacon_number(bacon_network, "Jennifer Lawrence"))
print(bacon_number(bacon_network, "Tom Cruise"))
```

Example Output:

```
5
1
```

▼ ✨ AI Hint: Breadth First Search Traversal

Path Courses

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?"

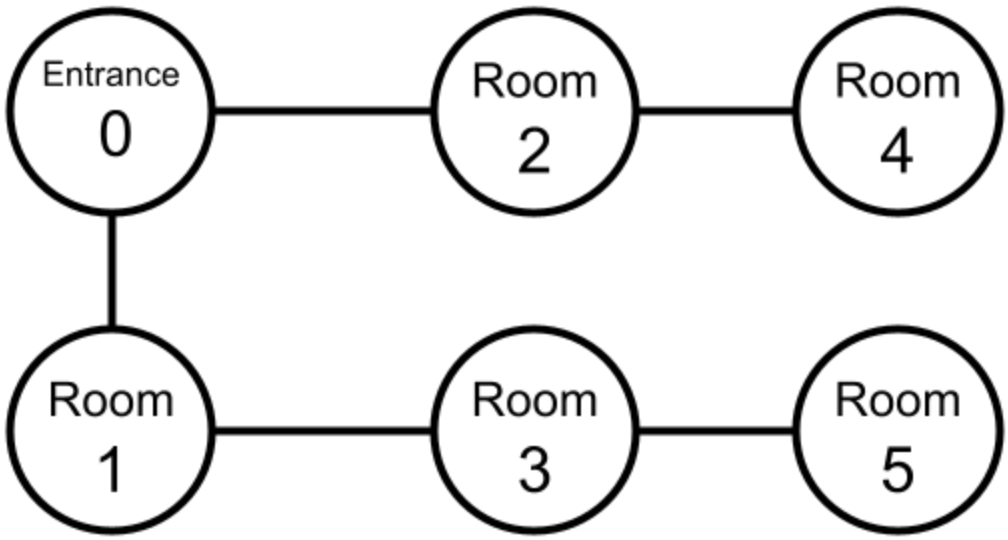
Problem 2: Press Junket Navigation

You've been invited to interview some of your favorite celebrities. Each group is stationed in a different room in the venue numbered `0` to `n-1`. To get to your assigned interview station, you need to navigate from the *entrance* which is room number `0` to your assigned room `target`.

Given an adjacency list `venue_map` where `venue_map[i]` indicates that there is a hallway between room `i` and each room in `venue_map[i]`, return a list representing the path from the entrance to your `target` room. If there are multiple paths, you may return any valid path.

```
def find_path(venue_map, target):
    pass
```

Example Usage:




```
venue_map = [
    [1, 2],
    [0, 3],
    [0, 4],
    [1, 5],
```

Path Courses

```
print(find_path(venue_map, 5))
print(find_path(venue_map, 2))
```

Example Output:

```
[0, 1, 3, 5]
[0, 2]
```

▼ 💡 **Hint: Path Reconstruction**

This problem requires you to reconstruct the path taken by either BFS or DFS. To reconstruct a path from BFS/DFS, we can keep track of each node's parent (the node from which it was discovered) during the search. After reaching the target, backtrack from the target node to the start using the parent pointers to trace the path.

Problem 3: Celebrity Rivalry Loops

In Hollywood, celebrity rivalries can escalate quickly. Sometimes, a rivalry between two stars leads to a chain reaction of other stars getting involved. You're tasked with determining if any group of celebrities is involved in a rivalry loop, where a rivalry escalates back to its origin.

You are given an adjacency list `rivalries`, where `rivalries[i]` represents the celebrities that celebrity `i` has a rivalry with. Write a function that detects whether any rivalry loops exist. A rivalry loop exists if there is a cycle of rivalries, where one celebrity's feud eventually leads back to themselves through others.

```
def has_rivalry_loop(rivalries):
    pass
```

Example Usage:

```
rivalries_1 = [
    [1],
    [0, 2],
    [1, 3],
    [2]
]

rivalries_2 = [
    [1],
    [2],
    [3],
```

```
print(has_rivalry_loop(rivalries_1))
print(has_rivalry_loop(rivalries_2))
```

Example Output:

```
False
True
```

▼ 💡 **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 4: Celebrity Feuds

You are in charge of scheduling celebrity arrival times for a red carpet event. To make things easy, you want to split the group of `n` celebrities labeled from `1` to `n` into two different arrival groups.

However, your boss has just informed you that some celebrities don't get along, and celebrities who dislike each other may not be in the same arrival group. Given the number of celebrities who will be attending `n`, and an array `dislikes` where `dislikes[i] = [a, b]` indicates that the person labeled `a` does not get along with the person labeled `b`, return `True` if it is possible to split the celebrities into two arrival periods and `False` otherwise.

```
def can_split(n, dislikes):
    pass
```

Example Usage:

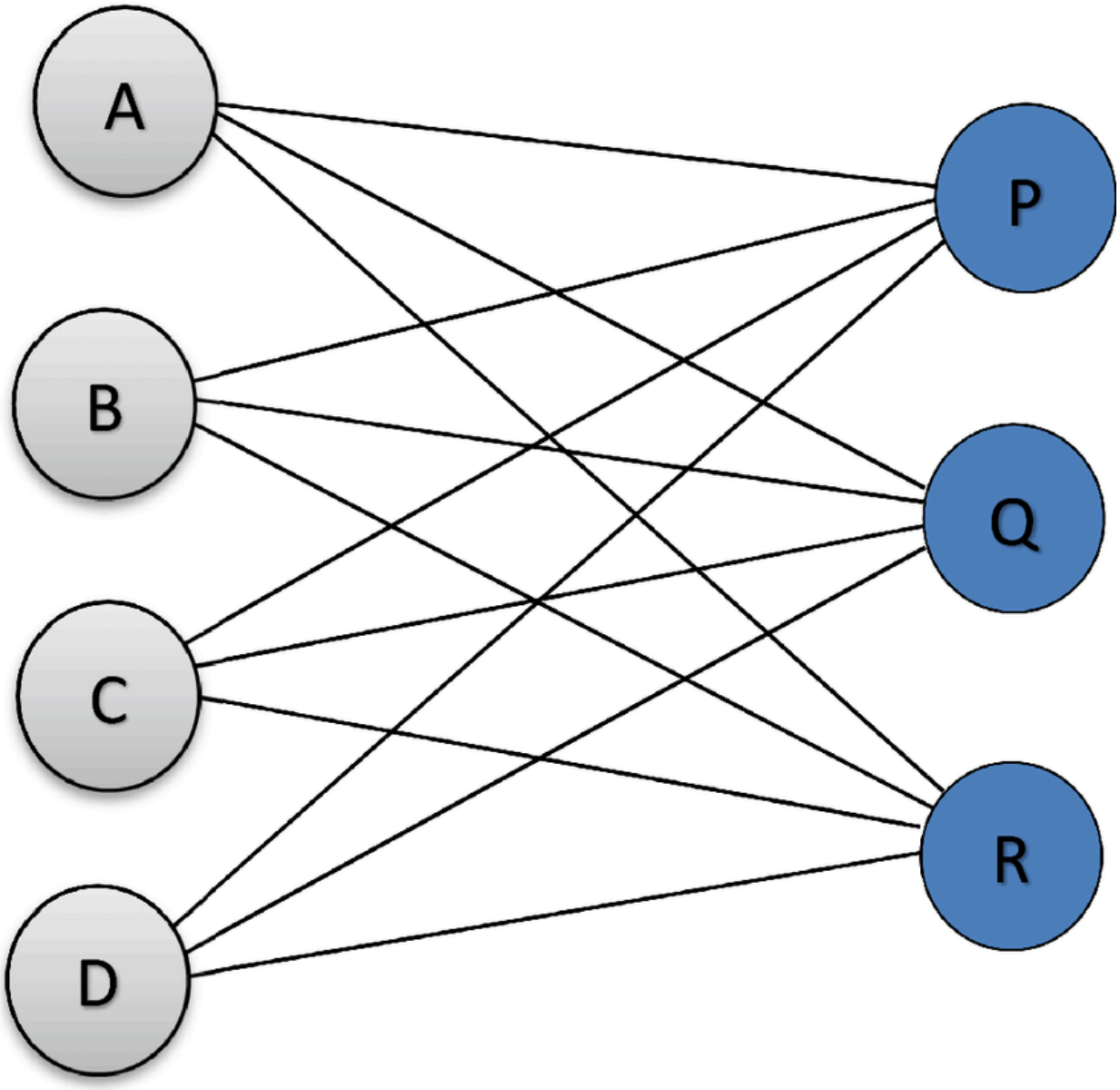
```
dislikes_1 = [[1, 2], [1, 3], [2, 4]]
dislikes_2 = [[1, 2], [1, 3], [2, 3]]

print(can_split(4, dislikes_1))
print(can_split(3, dislikes_2))
```

Example Output:

```
True
False
```

▼ 💡 **Hint: Bipartite Graphs**

This problem requires you to determine whether a graph is bipartite. A bipartite graph is a graph where the nodes can e divided into two distinct sets such that no two vertices in the saem set are connected by an edge. All edges must go between vertices in different sets.



We can determine whether a graph is bipartite using a technique called *graph coloring*. To determine if a graph is bipartite, we try coloring the graph using two colors: start from any node, color it one color, and color all its neighbors the opposite color. If at any point two adjacent nodes have the same color, the graph is not bipartite. You can do this using either BFS or DFS.

Problem 5: Maximizing Star Power Under Budget

part of the project. You want to maximize the total star power of the cast while ensuring that the total cost of hiring these celebrities stays under a given budget.

You are given a graph where:

- Each vertex represents a celebrity.
- Each edge between two celebrities represents a collaboration, with two weights:
  1. The star power (benefit) they bring when collaborating.
  2. The cost to hire them both for the project.

The graph is given as a dictionary `collaboration_map` where each key is a celebrity and the corresponding value is a list of tuples. Each tuple contains a connected celebrity, the star power of that collaboration, and the cost of the collaboration. Given a `start` celebrity, `target` celebrity, and maximum `budget`, return the maximum star power it is possible for you film to have from `start` to `target`.

```
def find_max_star_power(collaboration_map, start, target, budget):
    pass
```

Example Usage:

```
collaboration_map = {
    "Leonardo DiCaprio": [("Brad Pitt", 40, 300), ("Robert De Niro", 30, 200)],
    "Brad Pitt": [("Leonardo DiCaprio", 40, 300), ("Scarlett Johansson", 20, 150)],
    "Robert De Niro": [("Leonardo DiCaprio", 30, 200), ("Chris Hemsworth", 50, 350)],
    "Scarlett Johansson": [("Brad Pitt", 20, 150), ("Chris Hemsworth", 30, 250)],
    "Chris Hemsworth": [("Robert De Niro", 50, 350), ("Scarlett Johansson", 30, 250)]
}

print(find_max_star_power(collaboration_map, "Leonardo DiCaprio", "Chris Hemsworth", 500))
```

Example Output:

```
110
Explanation: The maximum star power while staying under budget is achieved on the path from L
```

Problem 6: Hollywood Talent Summit

Hollywood is hosting a major talent summit, and representatives from all production studios across various cities must travel to the capital city, Los Angeles (city 0). There is a tree-structured network of cities consisting of `n` cities numbered from `0` to `n-1`, with exactly `n-1` two-way roads connecting them. The roads are described by the 2D array `roads`, where `roads[i] = [a, b]` indicates a road connecting city `a` and city `b`.

Each studio has a car with limited seats, as described by the integer `seats`, which indicates the

another car along the way to save fuel.

The goal is to calculate the minimum number of liters of fuel needed for all representatives to travel to the capital city for the summit. Each road between cities costs one liter of fuel to travel.

Write a function that returns the minimum number of liters of fuel required for all representatives to reach the summit.

```
def minimum_fuel(roads, seats):
    pass
```

Example Usage:

```
roads_1 = [[0,1],[0,2],[0,3]]
seats_1 = 5

roads_2 = [[3,1],[3,2],[1,0],[0,4],[0,5],[4,6]]
seats_2 = 2

print(minimum_fuel(roads_1, seats_1))  # Output: 3
print(minimum_fuel(roads_2, seats_2))  # Output: 7
```

Example Output:

```
3
7
```

▼ 💡 **Hint: Finding the Ceiling**

This problem may benefit from importing the `math` library and using the `math.ceil()` method, which rounds a number **up** to the nearest integer.

Close Section