TIP102 | Intermediate Technical Interview Prep

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

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.

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

▶ **Note on Expectations**

🔎 Problem Solving Approach

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

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

We will learn to:

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

▶ **Comment on UPI**

▶ **UPI Example**

Breakout Problems Session 1

▼ Standard Problem Set Version 1

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
    [0, 0, 1], # Flight 1
    [0, 0, 0]  # Flight 2
]

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**

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, 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
```

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
Explanation: Cannot fly from Airport 4 to Airport 0
```

▶ 💡 **Hint: BFS or DFS?**

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**

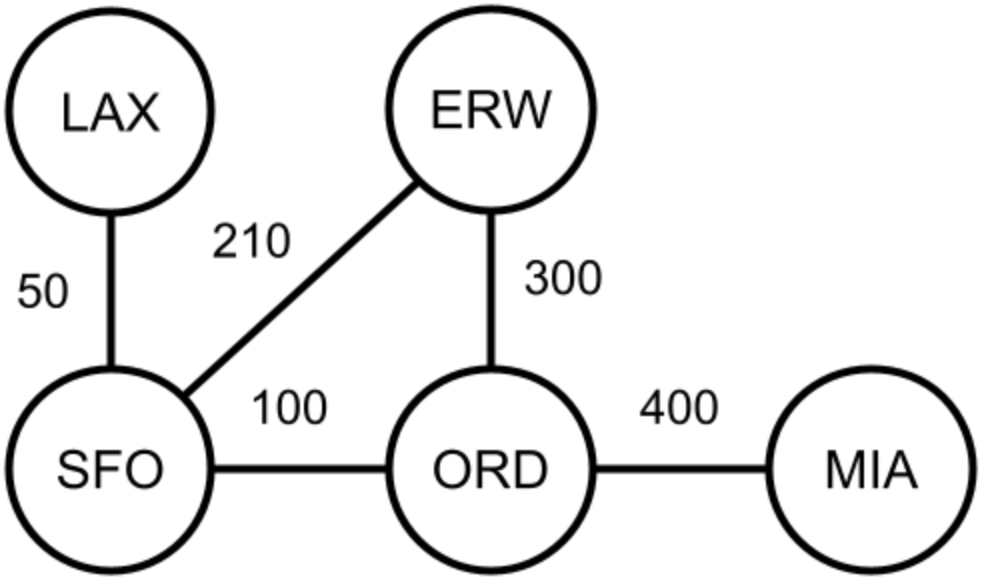
Problem 5: Get Flight Cost

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:




```
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 + 1(
960 would also be an acceptable answer following the path from LAX -> SFO -> ERW -> (
```

▶ 💡 **Hint: Weighted Graphs**

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.
3. If CodePath Airlines used an adjacency matrix instead of an adjacency dictionary/list, would the time complexity change? Why or why not?

```
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):
    pass
```

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**

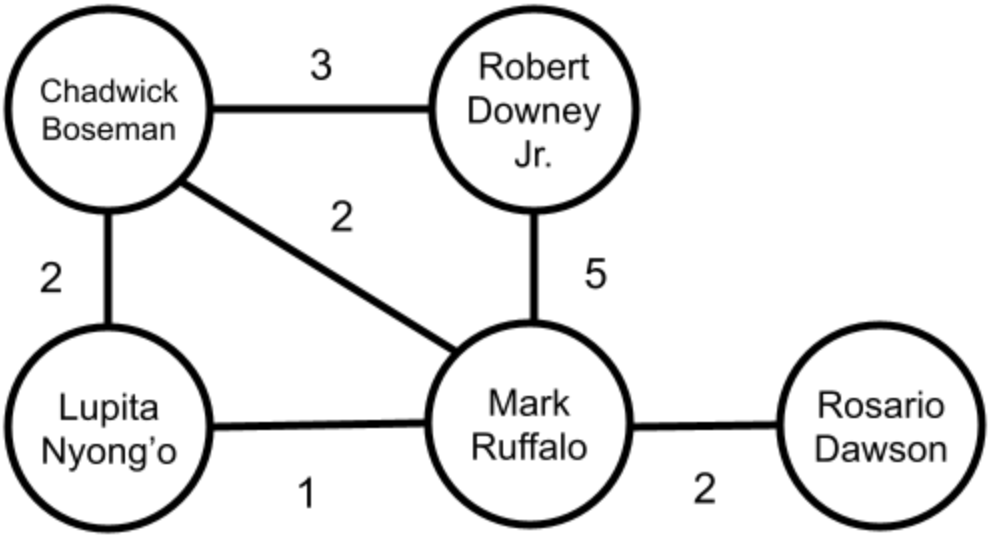
Close Section

▼ 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)]
```

▶ 💡 **Hint: Weighted Graphs**

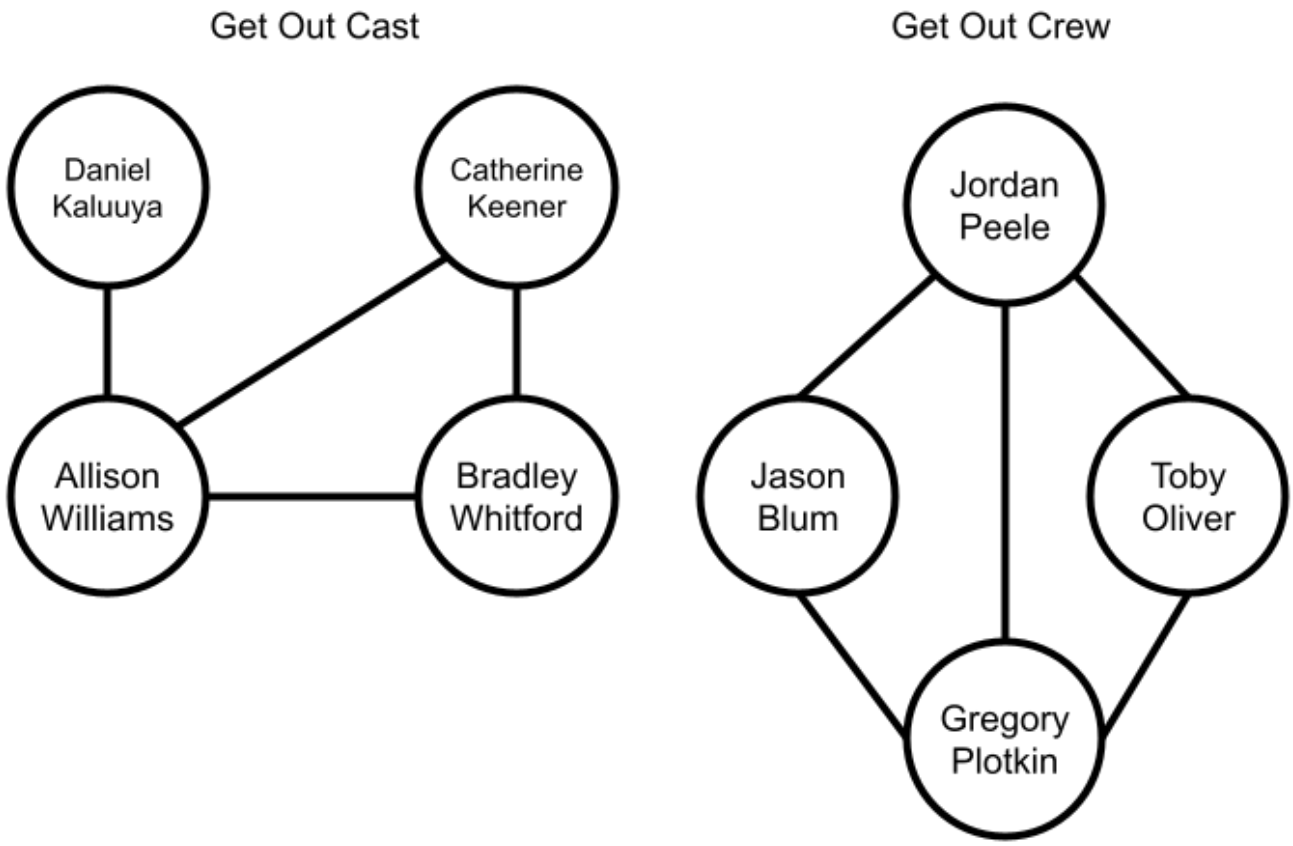
Problem 2: Cast vs Crew

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 = {
    "Daniel Kaluuya": ["Allison Williams"],
    "Allison Williams": ["Daniel Kaluuya", "Catherine Keener", "Bradley Whitford"],
    "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**

▶ 💡 **Hint: Finding Components**

Problem 3: Bacon Number

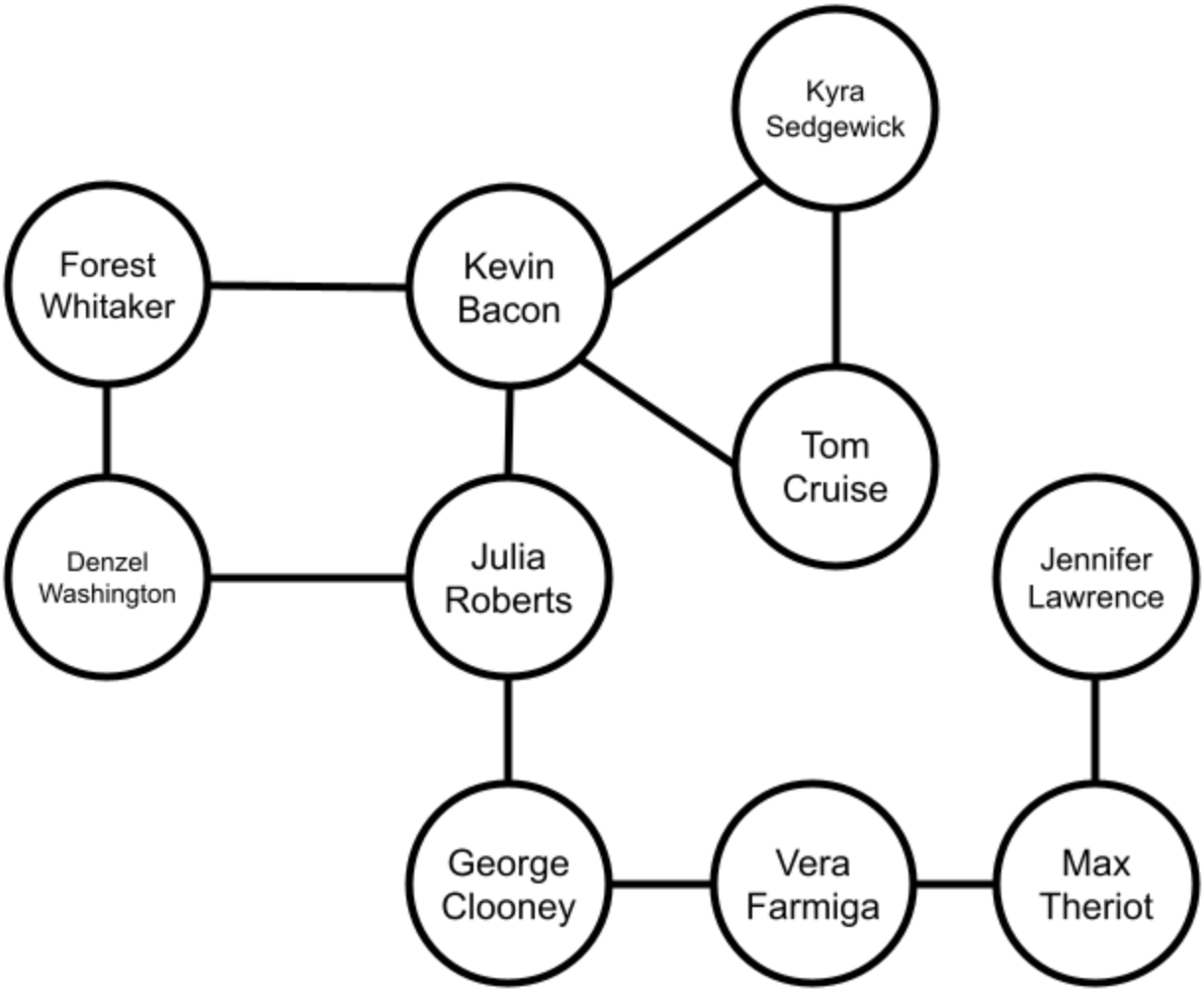
Six Degrees of Kevin Bacon is a game where you try to find a path of mutual connections between some actor or person to the actor Kevin Bacon in six steps or less. You are given an adjacency dictionary `bacon_network`, where each key represents an `actor` and the corresponding list `bacon_network[actor]` 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.

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 Cruis
    "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
```

▶ 💡 **Hint: Breadth First Search**

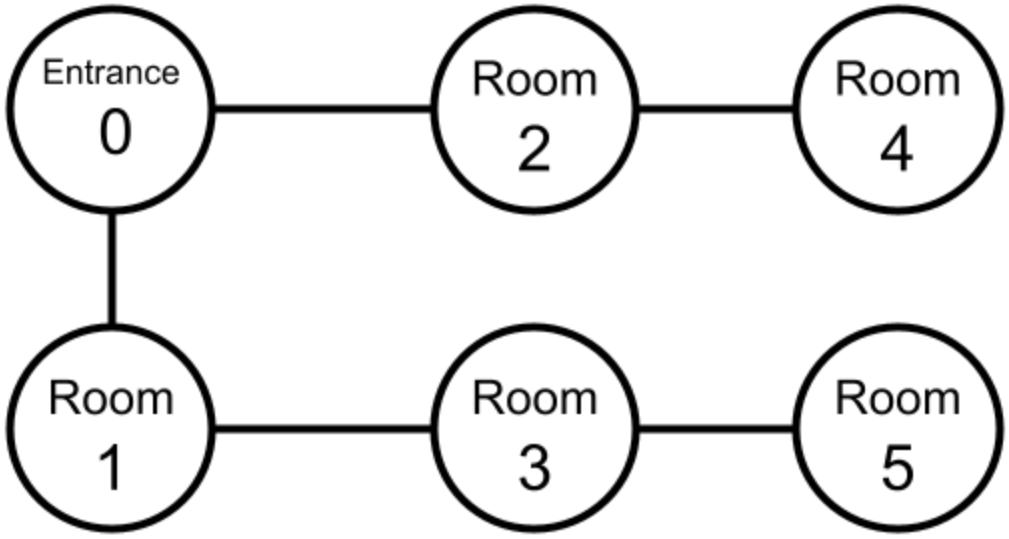
Problem 4: Press Junket Navigation

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**

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 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),
    "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`.

```
def find_max_star_power(collaboration_map, costar_a, costar_b):
    pass
```

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 -> S
(40 + 20 + 30 = 90).
The other path is Leonardo DiCaprio -> Robert De Niro -> Chris Hemsworth (30 + 50 =
```

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
```

▶ 💡 **Hint: Bipartite Graphs**

Close Section

▶ Advanced Problem Set Version 1

▶ Advanced Problem Set Version 2