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

▶ Standard Problem Set Version 2

▼ Advanced Problem Set Version 1

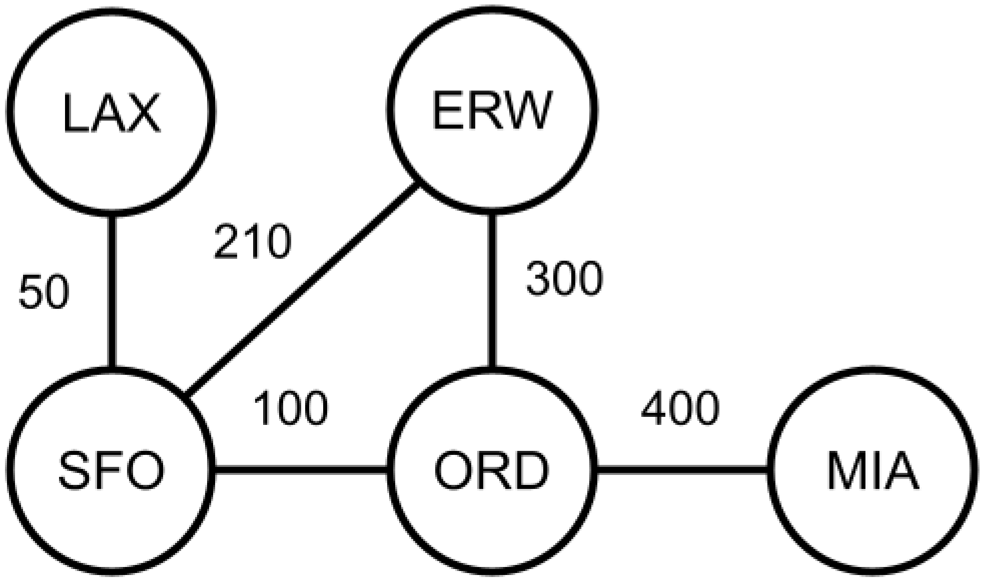
Problem 1: Get Flight Cost

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.

```python
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**

▶ 💡 **Hint: Breadth First Search**

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:

```
flights = {
    'JFK': ['LAX', 'SFO'],
    '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**

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`.

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

Example Usage:

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

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 Maneuv
False
Example 1 Explanation: There are 2 flight training courses. To take *Advanced Maneuv
```

▶ 💡 **Hint: BFS or DFS?**

Problem 5: Reorient Flight Routes

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`.

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

Example Usage:

```python
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 air
```

Problem 6: Find All Flight Routes

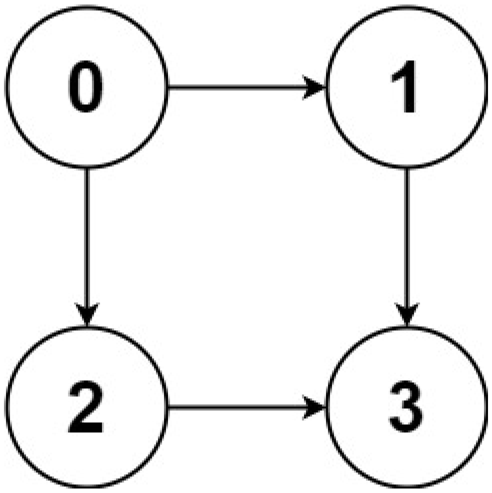
You are given a flight network represented as a directed acyclic graph (DAG) with `n` airports, labeled from `0` to `n - 1`. Your goal is to find all possible flight paths from airport `0` (the starting point) to airport `n - 1` (the final destination) and return them in any order.

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`.

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

Example Usage 1:



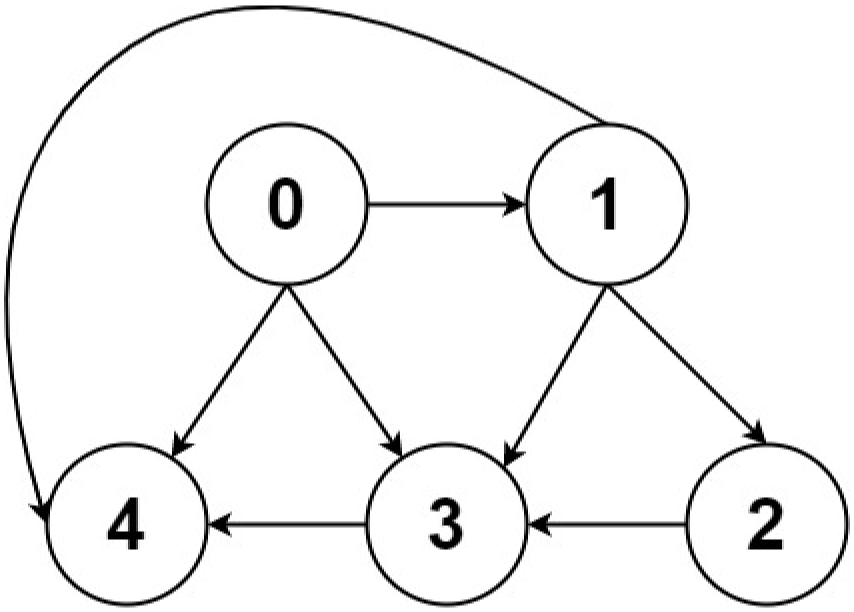
```
flight_routes_1 = [[1, 2], [3], [3], []]

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

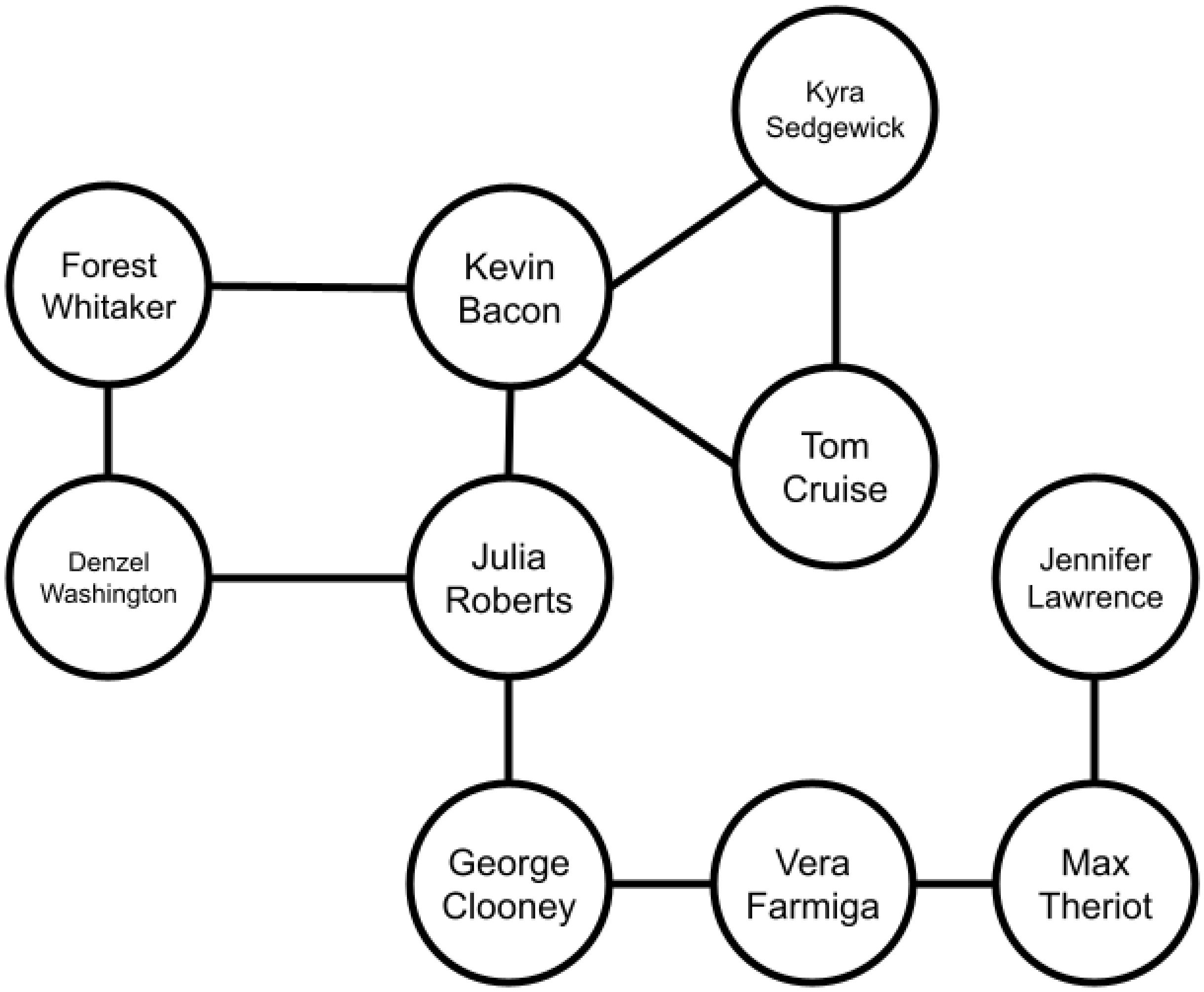
Six Degrees of Kevin Bacon is a game where you try to find a path of mutual connections between some actor or person to the actor Kevin Bacon in six steps or less. You are given an adjacency dictionary `bacon_network`, where each key represents an `actor` and the corresponding list `bacon_network[actor]`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

```python
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**

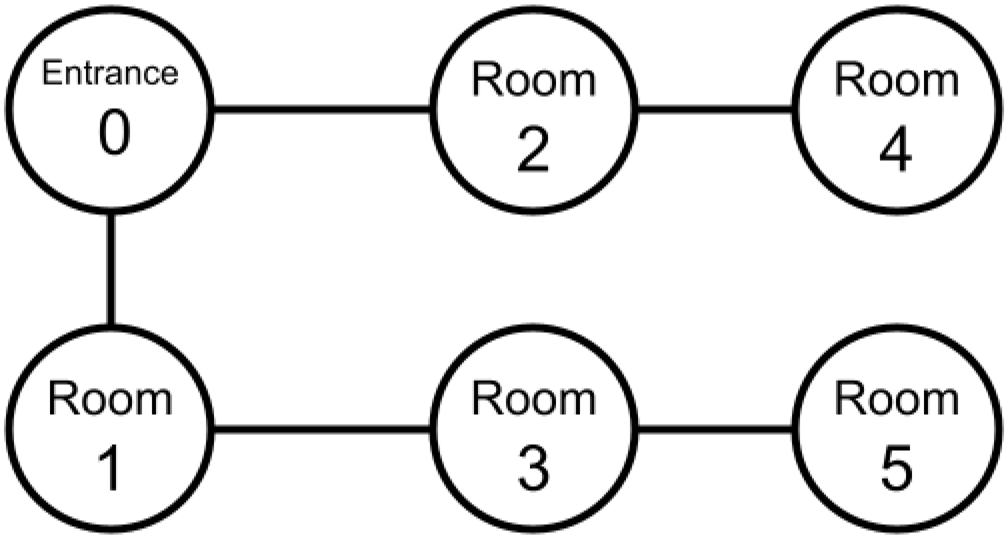
Problem 2: Press Junket Navigation

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],
    [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 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],
    [0]
]

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

Example Output:

```
False
True
```

▶ 💡 **Hint: BFS or DFS?**

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**

Problem 5: Maximizing Star Power Under Budget

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, and each celebrity demands a fee to be 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",
```

Example Output:

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

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 number of people that can travel in one car. Representatives can either drive their own car or join 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.

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

Example Usage:

```python
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**

Close Section