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: Advanced Graphs

Session Overview

In this session, students will continue to practice using BFS and DFS to solve intermediate 2D matrices problems. Students who choose the Advanced track will be introduced to additional graph algorithms including Dijkstra's Algorithm, Topological Sort, and Union Find.

> 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: Crafting Survival Gear

In the aftermath of the zombie apocalypse, your survival depends on crafting essential gear from limited resources. You are given a list of strings, `gear` that you can craft and a 2D list `components` where `components[i]` contains the materials you need to craft `gear[i]`.

Some components are basic supplies that you already have, while others need to be crafted from other gear. Additionally, you are given a string array `supplies` containing all the components you initially have in your stash (infinite supply).

Return a list of all the gear you can craft. You can return the list in any order.

```
def craftable_gear(gear, components, supplies):
    pass
```

Example Usage:

```
gear_1 = ["weapon"]
components_1 = [["metal", "wood"]]
supplies_1 = ["metal", "wood", "rope"]

gear_2 = ["weapon", "trap"]
components_2 = [["metal", "wood"], ["weapon", "wire"]]
supplies_2 = ["metal", "wood", "wire"]

gear_3 = ["weapon", "trap", "shelter"]
components_3 = [["metal", "wood"], ["weapon", "wire"], ["trap", "wood", "metal"]]
supplies_3 = ["metal", "wood", "wire"]

print(craftable_gear(gear_1, components_1, supplies_1))
print(craftable_gear(gear_2, components_2, supplies_2))
print(craftable_gear(gear_3, components_3, supplies_3))
```

Example Output:

```
["weapon"]
Example 1 Explanation: You can craft "weapon" since you have the components "metal" a

["weapon", "trap"]
Example 2 Explanation:
- You can craft "weapon" first since you have "metal" and "wood."
- After crafting "weapon", you can craft "trap" since you have "wire" and the "weapo

["weapon", "trap", "shelter"]
Example 3 Explanation:
- You can craft "weapon" first.
- With the "weapon" and "wire", you can craft "trap."
- With the "trap", "wood", and "metal," you can craft "shelter."
```

▶ 💡 **Hint: Topological Sort**

Problem 2: Spread the Zombie Cure

In the aftermath of the zombie apocalypse, a cure has been developed to stop the infection, but it must be distributed across a network of survivor camps. The camps are connected by communication lines, and you need to determine how quickly the cure can reach all the camps. Each camp is represented as a node in the network, and each communication line is represented as a directed edge with a travel time.

You are given an integer `n`, representing the number of camps (nodes), and a list of travel times `times`, where `times[i] = (u, v, w)` indicates that it takes `w` time for the cure to travel from camp `u` to camp `v`. You will send the cure from a starting camp `k`.

Return the minimum time it takes for all camps to receive the cure. If it is impossible for all camps to receive the cure, return `-1`.

```
def spread_cure(n, times, k):
    pass
```

Example Usage:

```
times_1 = [(2, 1, 1), (2, 3, 1), (3, 4, 1)]
times_2 = [(1, 2, 1), (2, 3, 2), (1, 3, 4)]
times_3 = [(1, 2, 1)]

print(spread_cure(1, times_1, 2))
print(spread_cure(3, times_2, 1))
print(spread_cure(2, times_3, 2))
```

Example Output:

```
2
Example 1 Explanation: Starting from camp 2, the cure travels to camp 1 in 1 unit
of time, to camp 3 in 1 unit of time, and from camp 3 to camp 4 in 1 additional unit
The cure reaches all camps within 2 units of time.

3
Example 2 Explanation: Starting from camp 1, the cure can reach camp 2 in 1 unit
of time and camp 3 via camp 2 in a total of 3 units of time (1+2). Therefore, the
minimum time to reach all camps is 3 units.

-1
Example 3 Explanation: It is impossible for the cure to travel from camp 2 to camp 1
since there is no direct or indirect communication line to connect the camps.
```

▶ 💡 **Hint: Dijkstra's Algorithm**

▶ 💡 **Hint: Priority Queues**

Problem 3: Number of Survival Camps

There are a series of survival camps that have cropped up around the city since the zombie apocalypse setarted. Given an `m x n` 2D binary grid `city` which represents a map of `1`s (survival camps) and `0`s (zombie controlled land), use Union Find to return the number of islands.

A survival camp is surrounded by zombie controlled land and is formed by connecting adjacent `1`s horizontally or vertically. You may assume all four edges of the grid are all surrounded by zombie controlled land.

```
def num_camps(city):
    pass
```

Example Usage:

```
city_1 = [
  [1,1,1,1,0],
  [1,1,0,1,0],
  [1,1,0,0,0],
  [0,0,0,0,0]
]

city_2 = [
  [1,1,0,0,0],
  [1,1,0,0,0],
  [0,0,1,0,0],
  [0,0,0,1,1]
]

print(num_camps(city_1))
print(num_camps(city_2))
```

Example Output:

```
1
3
```

▶ 💡 **Hint: Union Find/Disjoint Set Union**

Problem 4: Find Eventual Safe Locations

There is a directed graph of `n` nodes where each node represents a location in a zombie infested city. Each node is labeled from `0` to `n - 1`. The graph is represented by a 0-indexed 2D integer array `city` where `city[i]` is an integer array of nodes adjacent to node `i`, meaning there is a directed edge from node `i` to each node in `city[i]`.

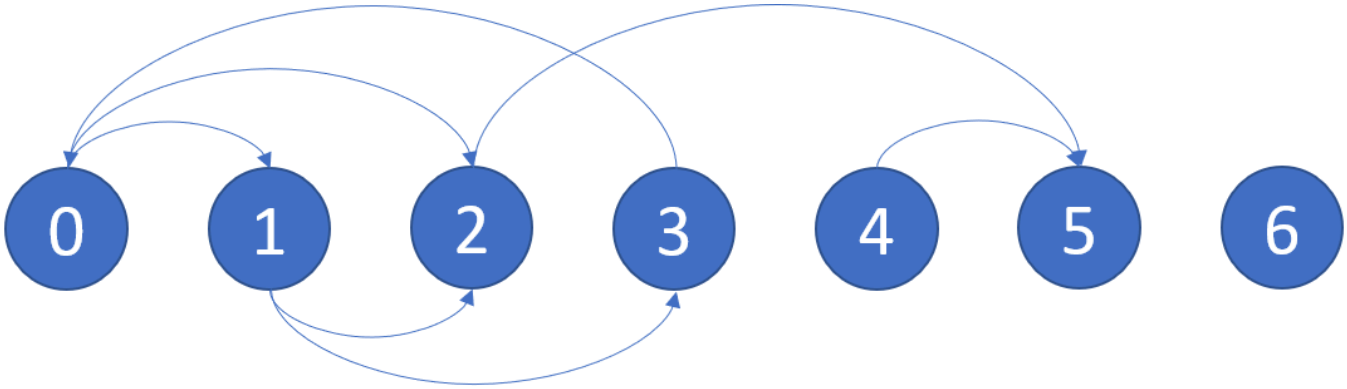
A node is a *terminal location* if it has no outgoing edges. A node is a *safe location* if every possible path starting from that node leads to a terminal location (or another safe location).

Return an array containing all the safe locations (nodes) of the graph. The answer should be sorted in ascending order.

```
def eventual_safe_locations(city):
    pass
```

Example Usage:

Example 1:




```
city_1 = [
    [1,2], # Location 0
    [2,3], # Location 1
    [5],   # Location 2
    [0],   # Location 3
    [5],   # Location 4
    [],    # Location 5
    []     # Location 6
    ]

city_2 = [
    [1,2,3,4], # Location 0
    [1,2],     # Location 1
    [3,4],     # Location 2
    [0,4],     # Location 3
    []         # Location 4
    ]

print(eventual_safe_locations(city_1))
print(eventual_safe_locations(city_2))
```

Example Output:

```
[2,4,5,6]
Example 1 Explanation: The given graph is shown above.
Nodes 5 and 6 are terminal locations as there are no outgoing edges from either of t
Every path starting at locations 2, 4, 5, and 6 all lead to either location 5 or 6.

[4]
Example 2 Explanation:
Only location 4 is a terminal location, and every path starting at node 4 leads to n
```

▶ 💡 **Hint: Topological Sort**

Problem 5: Maximizing Zombie Avoidance

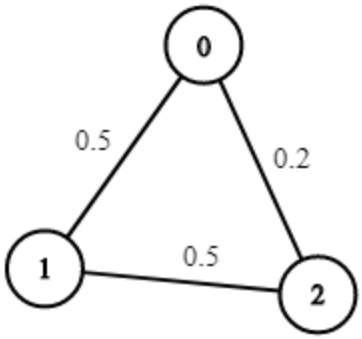
In a post-apocalyptic world, you need to find the safest path through a series of zombie-infested zones. The zones are connected by roads, but each passage has a probability of success for safely making it through without encountering zombies. You must find the path that maximizes your chances of survival when traveling from one safe zone to another.

You are given an undirected graph with `n` nodes where each node represents a safe zone and each edge represents a road between two zones and has a probability of success. The graph is represented by an edge list, where `edges[i] = [a, b]` means that there is an undirected passage between zone `a` and zone `b` with a probability of success `succ_prob[i]`.

Given two zones, `start` and `end`, find the path that maximizes the probability of survival when traveling from the `start` zone to the `end` zone and return that maximum probability. If there is no safe path from start to end, return `0`. Round your answer to the nearest hundredth.

```
def max_survival_probability(n, edges, succ_prob, start, end):
    pass
```

Example Usage 1:



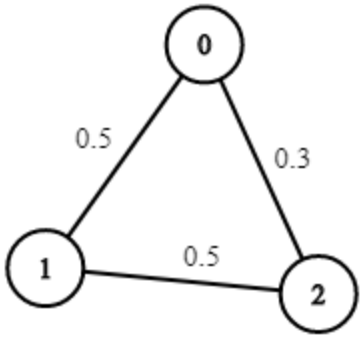

```
edges_1 = [[0, 1], [1, 2], [0, 2]]
succ_prob_1 = [0.5, 0.5, 0.2]

print(max_survival_probability(3, edges_1, succ_prob_1, 0, 2))
```

Example Output 1:

```
0.25
Example 1 Explanation:
The possible paths from zone 0 to zone 2 are:
- 0 -> 1 -> 2, with probability 0.5 * 0.5 = 0.25.
- 0 -> 2, with probability 0.2.
The safest path has a probability of 0.25.
```

Example Usage 2:



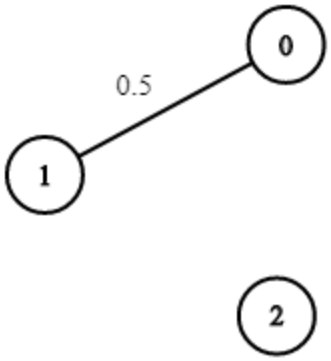

```
edges_2 = [[0, 1], [1, 2], [0, 2]]
succ_prob_2 = [0.5, 0.5, 0.3]

print(max_survival_probability(3, edges_2, succ_prob_2, 0, 2))
```

Example Output 2:

```
0.3
Example 2 Explanation:
The possible paths from zone 0 to zone 2 are:
- 0 -> 1 -> 2, with probability 0.5 * 0.5 = 0.25.
- 0 -> 2, with probability 0.3.
The safest path has a probability of 0.3.
```

Example Usage 3:




```
edges_3 = [[0, 1]]
succ_prob_3 = [0.5]

print(max_survival_probability(2, edges_3, succ_prob_3, 0, 1))
```

Example Output 3:

```
0.5
Example 3 Explanation:
There is only one path between zone 0 and zone 1, with a probability of 0.5.
```

▶ 💡 **Hint: Dijkstra's Algorithm**

Problem 6: Rebuilding the Safe Zones

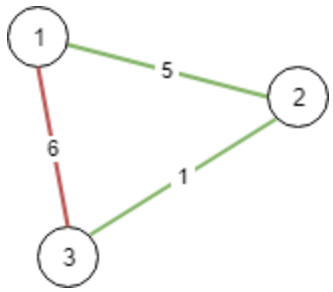
In the post-apocalyptic world, several survivor camps are scattered across the land. To unite these camps and form a network of safe zones, you need to rebuild the roads between them. However, the rebuilding process comes with a cost, and resources are limited.

You are given an integer `n` representing the number of survivor camps, labeled from `1` to `n`. You are also given an array `connections`, where `connections[i] = [x, y, cost]` indicates that rebuilding the road between camp `x` and camp `y` will cost `cost` resources.

Your goal is to minimize the total cost of connecting all the camps so that there is at least one safe path between every pair of camps. If it is impossible to connect all the camps, return `-1`.

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

Example Usage 1:



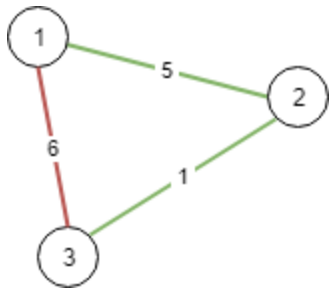

```
connections_1 = [[1, 2, 5], [1, 3, 6], [2, 3, 1]]

print(min_rebuilding_cost(3, connections_1))
```

Example Output 1:

```
6
Example 1 Explanation:
The minimum cost to connect all the camps is to rebuild the road between camp 2 and
```

Example Usage 2:




```
connections_2 = [[1, 2, 3], [3, 4, 4]]
print(min_rebuilding_cost(4, connections_2))
```

Example Output 2:

```
-1
Example 2 Explanation:
It is impossible to connect all 4 camps since there is no connection between camps 1
```

▶ 💡 **Hint: Kruskal's Algorithm**

Close Section

▼ Advanced Problem Set Version 2

Problem 1: Schedule Banquet Tasks

In preparation for the grand castle banquet, there are `tasks` that must be completed. Each task is represented by a string. You are given a list of dependencies called `prerequisites`, where `prerequisites[i] = [task_a, task_b]` means that you must complete `task_b` before you can complete `task_a`.

For example, if you have `prerequisites = [["prepare_dessert", "set_table"]]`, it means that you must set the table before you can prepare the dessert.

Return a list with the ordering of tasks such that all tasks can be completed in time for the banquet. If there are multiple valid orderings, return any of them. If it's impossible to complete all the tasks (due to circular dependencies), return an empty array.

```
def prepare_banquet(tasks, prerequisites):
    pass
```

Example Usage:

```
tasks_1 = ["set_table", "prepare_dessert"]
prerequisites_1 = [["prepare_dessert", "set_table"]]

tasks_2 = ["stock_pantry", "main_course", "decorations", "serve_food"]
prerequisites_2 = [["main_course", "stock_pantry"], ["decorations", "stock_pantry"],

tasks_3 = ["only_task"]
prerequisites_3 = []

print(prepare_banquet(tasks_1, prerequisites_1))
print(prepare_banquet(tasks_1, prerequisites_2))
print(prepare_banquet(tasks_1, prerequisites_3))
```

Example Output:

```
["set_table", "prepare_dessert"]
Example 1 Explanation: You need to set the table before you can prepare the dessert.

["stock_pantry", "main_course", "decorations", "serve_food"]
Example 2 Explanation:
["stock_pantry", "decorations", "main_course", "serve_food"] is also an acceptable
answer. You need to finish both cooking the main course and setting up the decoratio
before you can serve the food. Both tasks depend on stocking the pantry.

["only_task"]
Example 3 Explanation: There's only one task to complete, so you can proceed without
```

▶ 💡 **Hint: Topological Sort**

Problem 2: Connecting Roads for Winter

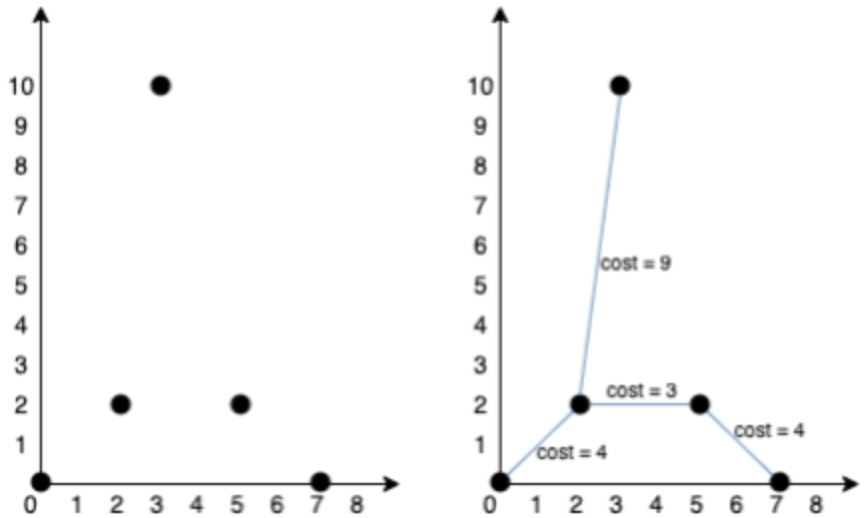
As winter approaches, the kingdom's roads must be reinforced to connect various outposts located across the kingdom. Each outpost is represented by a coordinate `[x_i, y_i]` on a 2D map. The cost of reinforcing a road between two outposts located at `[x_i, y_i]` and `[x_j, y_j]` is the *Manhattan distance* between them, which is calculated as `|x_i - x_j| + |y_i - y_j|`, where `|val|` denotes the absolute value of `val`.

Write a function `prepare_winter_roads()` that returns the minimum cost to connect all outputs. All outposts are considered connected if there is exactly one simple path between any two outposts.

```
def prepare_winter_roads(outposts):
    pass
```

Example Usage:

Example 1:



```
outposts_1 = [[0, 0], [2, 2], [3, 10], [5, 2], [7, 0]]
outposts_2 = [[3, 12], [-2, 5], [-4, 1], [5, 2], [9, 6]]

print(prepare_winter_roads(outposts_1))
print(prepare_winter_roads(outposts_2))
```

Example Output:

```
20
Example 1 Explanation: The total minimum cost of connecting all the outposts is 20,

26
Example 2 Explanation: The kingdom's outposts in this scenario are more spread out,
```

▶ 💡 **Hint: Dijkstra's Algorithm**

▶ 💡 **Hint: Priority Queues**

Problem 3: Number of Towns

As part of the royal survey, the kingdom is mapping out its territories. The kingdom's lands are represented by an `m x n` 2D binary grid `map`, where:

- **1** represents **urban** land, and
- **0** represents **rural** land.

Determine how many distinct towns exist in the kingdom. An town is defined as a group of connected urban land squares, and two squares are connected if they are adjacent horizontally or vertically. Assume that the entire map is surrounded by rural land.

To solve this problem, you must use the Union Find technique to efficiently group connected urban land squares into towns.

1. Use Union Find to merge connected urban land squares.
2. Return the total number of separate towns in the map.

```
def count_towns(grid):
    pass
```

Example Usage:

```
kingdom_map_1 = [
    ["1", "1", "1", "1", "0"],
    ["1", "1", "0", "1", "0"],
    ["1", "1", "0", "0", "0"],
    ["0", "0", "0", "0", "0"]
]

kingdom_map_2 = [
    ["1", "1", "0", "0", "0"],
    ["1", "1", "0", "0", "0"],
    ["0", "0", "1", "0", "0"],
    ["0", "0", "0", "1", "1"]
]

print(count_towns(kingdom_map_1))
print(count_towns(kingdom_map_2))
```

Example Output:

```
1
3
```

▶ 💡 **Hint: Union Find/Disjoint Set Union**

Problem 4: Find the Royal Lineage

In the kingdom, the royal family is structured in a lineage represented by a Directed Acyclic Graph (DAG) with `n` members, where each member is identified by a string representing their name.

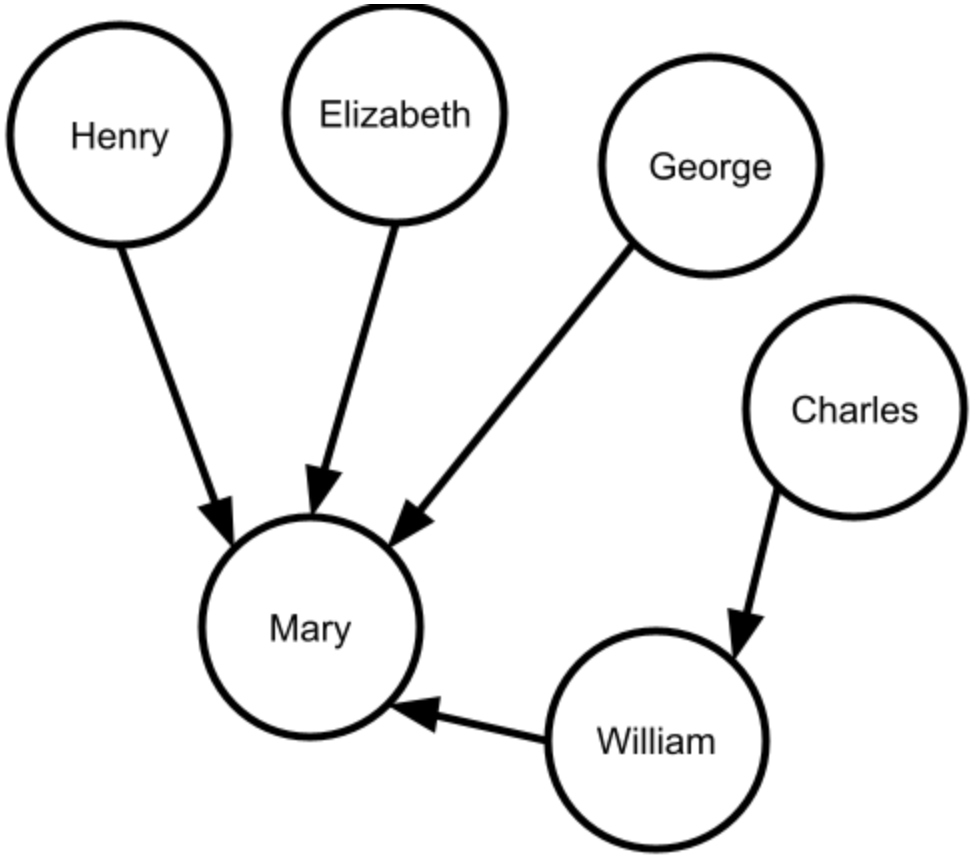
You are given a list of relationships in the form of a 2D array `relationships`, where `relationships[i] = [elder_name, descendant_name]` indicates that there is a unidirectional bond between `elder_name` and `descendant_name`. This means that `elder_name` is an ancestor of `descendant_name`.

Your task is to return a dictionary with the complete royal lineage of each member. For each member, return a list of all their ancestors sorted alphabetically. A member `u` is an ancestor of another member `v` if `u` can reach `v` through the set of family ties.

```
def find_kingdom_lineage(names, relationships):
    pass
```

Example Usage:

Example 1:




```
names_1 = ["Henry", "Elizabeth", "George", "Mary", "Charles", "William"]
relationships_1 = [["Henry", "Mary"], ["Elizabeth", "Mary"], ["George", "Mary"], ["C

names_2 = ["Alice", "Bob", "Catherine", "Diana", "Edward"]
relationships_2 = [["Alice", "Bob"], ["Alice", "Catherine"], ["Bob", "Diana"], ["Cat

print(find_kingdom_lineage(names_1, relationships_1))
print(find_kingdom_lineage(names_2, relationships_2))
```

Example Output:

```
{
    "Henry": [],
    "Elizabeth": [],
    "George": [],
    "Mary": ["Elizabeth", "George", "Henry", "William"],
    "Charles": [],
    "William": ["Charles"]
}
Example 1 Explanation:
- Henry, Elizabeth, George, and Charles have no ancestors.
- Mary has ancestors Elizabeth, George, Henry, and William, sorted alphabetically.
- William has ancestor Charles.

{
    "Alice": [],
    "Bob": ["Alice"],
    "Catherine": ["Alice"],
    "Diana": ["Alice", "Bob", "Catherine"],
    "Edward": ["Alice", "Bob", "Catherine", "Diana"]
}
Example 2 Explanation:
- Alice has no ancestors.
- Bob has ancestor Alice.
- Catherine has ancestor Alice.
- Diana has ancestors Alice, Bob, and Catherine.
- Edward has ancestors Alice, Bob, Catherine, and Diana.
```

▶ 💡 **Hint: Topological Sort**

Problem 5: Minimum Effort Path to the Castle

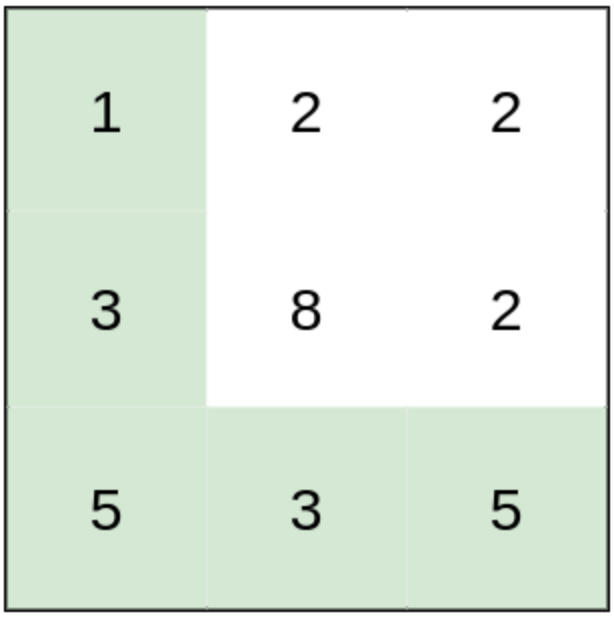
You are about to embark on a long journey to the castle to make a request of the queen. The kingdom's terrain is represented by a `m x n` 2D array `heights`, where `heights[row][col]` represents the height of the terrain at position `(row, col)`.

You start your journey at the top-left corner of the kingdom `(0, 0)` and need to safely reach the castle, which is located at the bottom-right corner `(rows-1, columns-1)`. You can move **up**, **down**, **left**, or **right** through the kingdom. However, you want to minimize the effort it takes to cross the rugged terrain.

The effort of a path is defined as the maximum absolute difference in heights between two consecutive locations along the path. Your task is to find the minimum effort required to travel from the top-left corner to the castle at the bottom-right.

```
def min_effort(heights):
    pass
```

Example Usage 1:



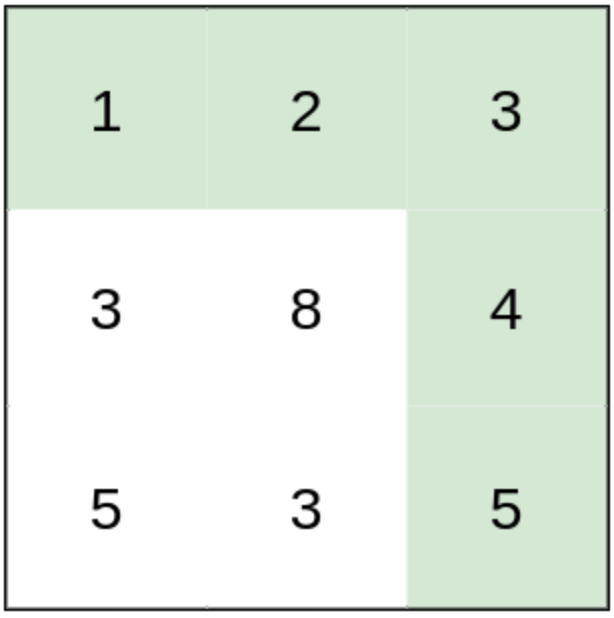

```
terrain_1 = [
    [1, 2, 2],
    [3, 8, 2],
    [5, 3, 5]
]

print(min_effort(terrain_1))
```

Example Output 1:

```
2
Example 1 Explanation: The path with the minimum effort of 2 can be found by traveli
through cells with minimal height differences.
```

Example Input 2:



```
terrain_2 = [
    [1, 2, 3],
    [3, 8, 4],
    [5, 3, 5]
]

print(min_effort(terrain_2))
```

Example Output 2:

```
1
Example 2 Explanation: The safest path to the castle in this scenario requires
only an effort of 1, as the height differences between consecutive cells are minimal
```

▶ 💡 **Hint: Dijkstra's Algorithm**

Problem 6: Reinforce the Kingdom's Strongholds

In the kingdom, there are `n` strongholds positioned at various integer coordinates on a 2D map. Each stronghold is represented as a stone located at $[x_i, y_i]$ on the map. No two strongholds occupy the same location.

To optimize the kingdom's defenses, you can remove a stronghold if it shares the same row or column as another stronghold that hasn't been removed yet. Your goal is to remove as many strongholds as possible while maintaining the defensive structure.

Given an array `strongholds` of length `n`, where $strongholds[i] = [x_i, y_i]$ represents the location of the `i`-th stronghold, return the largest possible number of strongholds that can be removed.

```
def reinforce_strongholds(strongholds):
    pass
```

Example Usage:

```
strongholds_1 = [[0,0], [0,1], [1,0], [1,2], [2,1], [2,2]]
strongholds_2 = [[0,0], [0,2], [1,1], [2,0], [2,2]]
strongholds_3 = [[0,0]]

print(reinforce_strongholds(strongholds_1))
print(reinforce_strongholds(strongholds_2))
print(reinforce_strongholds(strongholds_3))
```

Example Output:

```
5
Example 1 Explanation: One way to remove 5 strongholds is as follows:
1. Remove stronghold [2,2] because it shares the same row as [2,1].
2. Remove stronghold [2,1] because it shares the same column as [0,1].
3. Remove stronghold [1,2] because it shares the same row as [1,0].
4. Remove stronghold [1,0] because it shares the same column as [0,0].
5. Remove stronghold [0,1] because it shares the same row as [0,0].
Stronghold [0,0] cannot be removed since it no longer shares a row/column with any o

3
Example 2 Explanation: One way to remove 3 strongholds is as follows:
1. Remove stronghold [2,2] because it shares the same row as [2,0].
2. Remove stronghold [2,0] because it shares the same column as [0,0].
3. Remove stronghold [0,2] because it shares the same row as [0,0].
Strongholds [0,0] and [1,1] cannot be removed since they no longer share a row/colum

0
Example 3 Explanation: Stronghold [0,0] is the only stronghold on the map, so you ca
```

▶ 💡 **Hint: Union Find/Disjoint Set Union**

Close Section