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

Problem 1: Nearest Zombie

Given an `m x n` binary matrix `grid` where `1`s represent humans and `0`s represent zombies, return the distance of the nearest zombie to each square in the grid.

The distance between two adjacent cells horizontally/vertically is `1`.

```python
def nearest_zombie(grid):
    pass
```

Example Usage:

```
grid_1 = [
    [0,0,0],
    [0,1,0],
    [0,0,0]
    ]

grid_2 = [
    [0,0,0],
    [0,1,0],
    [1,1,1]
    ]

print(nearest_zombie(grid_1))
print(nearest_zombie(grid_2))
```

Example Output:

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

[
    [0,0,0],
    [0,1,0],
    [1,2,1]
    ]
```

Problem 2: Defending the Safehouse

The city is under attack by zombies, and you've built a secure underground safehouse with tunnels connecting key areas. The city is represented as a binar `m x n` matrix `city` where:

- A cell with value `1` represents a passage of the safehouse that's accessible
- A cell with value `0` represents a blocked area or wall

You can move between accessible passages either downward `(row + 1, col)` or to the right `(row, col + 1)`. You start in the top-left corner `(0, 0)`, and your goal is to reach the safehouse at the bottom-right corner `(m - 1, n - 1)`.

However, the zombies are closing in, and you need to check if it’s possible to disconnect the safehouse by flipping the value of **at most one cell** (except for the entrance `(0, 0)` and the safehouse `(m - 1, n - 1)`).

Flipping a cell changes its value from `0` to `1` or from `1` to `0`. Your task is to determine if you can make the city **disconnect**, meaning that no path exists between the entrance and the safehouse.

Return `True` if you can disconnect the city, otherwise return `False`.

```python
def can_disconnect_safehouse(city):
    pass
```

Example Usage:

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

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

print(can_disconnect_safehouse(city_1))
print(can_disconnect_safehouse(city_2))
```

Example Output:

```
False
True
Example 2 Explanation: Flipping the cell at (1, 1) disconnects the safehouse
```

Problem 3: Zombie Infested City Regions

The city is in chaos due to the zombie apocalypse, and has been fenced off into sections according to the severity of the zombie infestation. The city is represented as an `n x n` `grid`, where each `1x1` square in the grid represents a part of the city and contains either a fence or an open area:

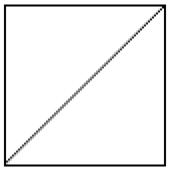
- A `'/'` or `'\\'` (forward or backslash) represents a *fence* dividing the square into two triangular zones.
- A `' '` represents an *open area* with no division in the square.

Given the `grid` represented as an array of strings where each substring is a row and each character in a substring is a column in the row, return the total number of contiguous regions.

Note that backslashes are represented as `'\\'` instead of `'\'` because backslashes are escaped characters.

```python
def count_regions(grid):
    pass
```

Example Usage 1:



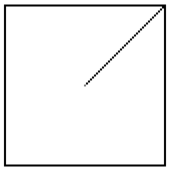
```
grid_1 = [" /","/ "]

print(count_regions(grid_1))
```

Example Output 1:

```
2
```

Example Usage 2:



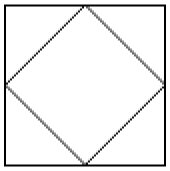
```
grid_2 = [" /","  "]

print(count_regions(grid_2))
```

Example Output 2:

```
1
```

Example Usage 3:



```
grid_3 = ["/\","\/"]

print(count_regions(grid_3))
```

Example Output 2:

```
5
Example Explanation: Recall that because \ characters are escaped, "\/" refers to \/
and "/\" refers to /\.
```

▶ 💡 **Hint: Understanding the Input**

Problem 4: Escape the Infected Zone

You are trapped in a rectangular zone that has been quarantined because it is infected with zombies. The infected zone borders two safe zones: the *Pacific Safety Zone* and the *Atlantic Safety Zone*. The Pacific Safety Zone borders the left and top edges of the infected zone, while the Atlantic Safety Zone borders the right and bottom edges.

The infected zone is partitioned into a grid of square subzones, and you are given an `m x n` integer matrix `safety` where `safety[row][column]` represents the safety level of the subzone at coordinate `(row, column)`. Higher values mean the zone is safer from the zombie outbreak.

Due to constant zombie movement, survivors can only move from one zone to an adjacent zone (north, south, east, or west) if the neighboring zone's safety level is *less than or equal to* the current zone's safety level. This means survivors can escape to a more dangerous zone but not to a safer one.

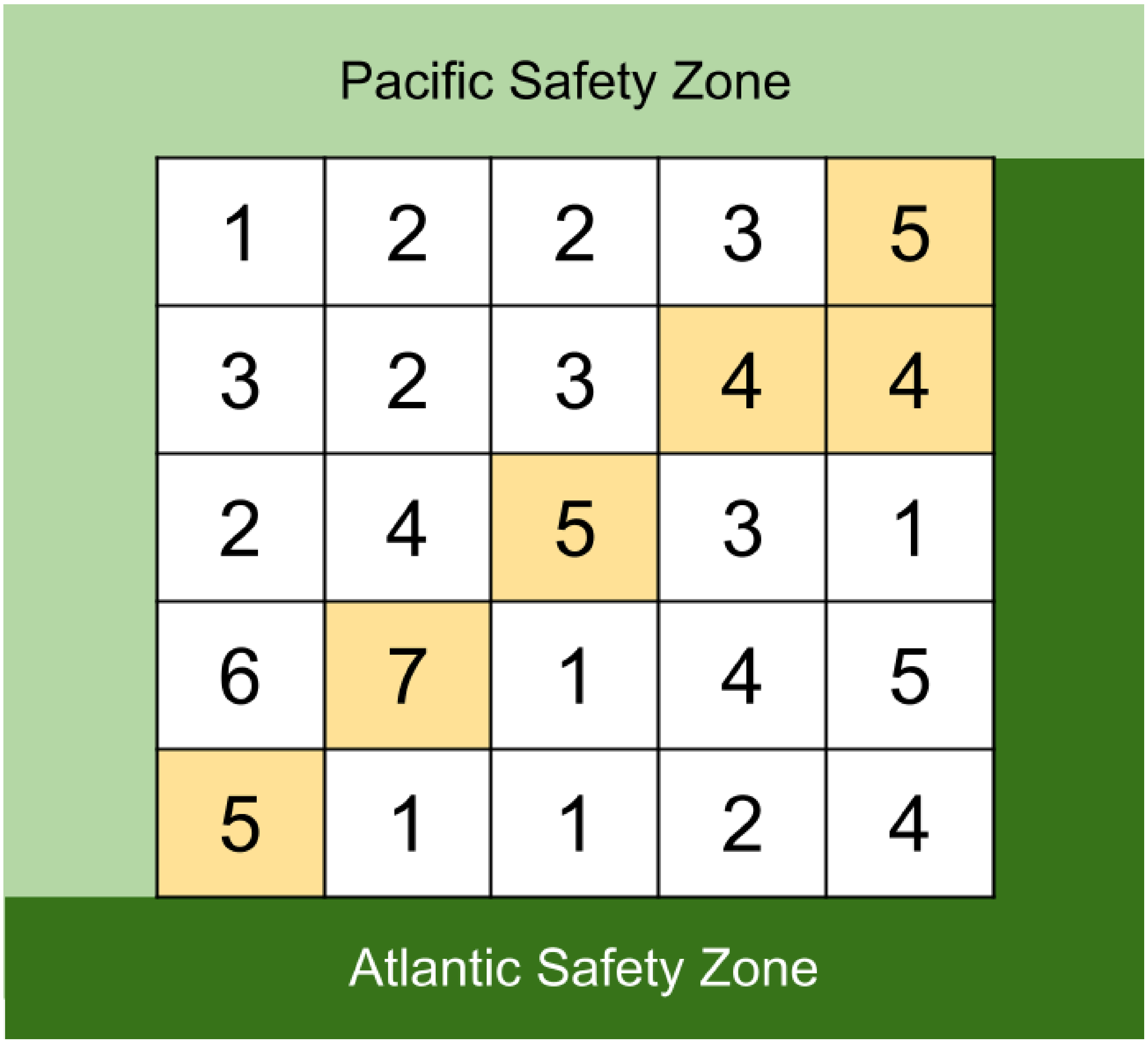
Your goal is to identify all subzones where survivors can potentially escape the island by reaching **both** the Pacific and Atlantic Safety Zones.

Return a 2D list of grid coordinates `result` where `result[i] = [r_i, c_i]` denotes that survivors in zone `(r_i, c_i)` can escape to both the Pacific and Atlantic Safety Zones.

```python
def escape_subzones(safety):
    pass
```

Example Usage:

Example 1:



```
safety_1 = [
    [1, 2, 2, 3, 5],
    [3, 2, 3, 4, 4],
    [2, 4, 5, 3, 1],
    [6, 7, 1, 4, 5],
    [5, 1, 1, 2, 4]
]

safety_2 = [
    [2, 1],
    [1, 2]
]

print(escape_subzones(safety_1))
print(escape_subzones(safety_2))
```

Example Output:

```
[[0, 4], [1, 3], [1, 4], [2, 2], [3, 0], [3, 1], [4, 0]]
Example 1 Explanation: Survivors can escape from several zones on the island.
[0,4]: [0,4] -> Pacific Safety Zone
       [0,4] -> Atlantic Safety Zone
[1,3]: [1,3] -> [0,3] -> Pacific Safety Zone
       [1,3] -> [1,4] -> Atlantic Safety Zone
[1,4]: [1,4] -> [1,3] -> [0,3] -> Pacific Safety Zone
       [1,4] -> Atlantic Safety Zone
[2,2]: [2,2] -> [1,2] -> [0,2] -> Pacific Safety Zone
       [2,2] -> [2,3] -> [2,4] -> Atlantic Safety Zone
[3,0]: [3,0] -> Pacific Safety Zone
       [3,0] -> [4,0] -> Atlantic Safety Zone
[3,1]: [3,1] -> [3,0] -> Pacific Safety Zone
       [3,1] -> [4,1] -> Atlantic Safety Zone
[4,0]: [4,0] -> Pacific Safety Zone
       [4,0] -> Atlantic Safety Zone

[[0, 0], [1, 1]]
Example 2 Explanation:
- From zone `[0, 0]`, survivors can reach the Pacific Safety Zone (by moving left or
and the Atlantic Safety Zone (by moving right or down).
- From zone `[1, 1]`, survivors can also escape to both zones.
```

Problem 5: Decreasing Zombie Path

Given an `m x n` matrix `city` where each cell holds an integer representing the number of zombies in that area of the city, return the length of the longest decreasing path we can travel through the `city`.

The distance between two adjacent cells horizontally/vertically is `1`. For the path to be considered decreasing, the number of zombies in each subsequent area traveled must be strictly less than the number of zombies in the preceding area. You may not visit any areas twice.

```python
def longest_decreasing_path(city):
    pass
```

Example Usage:

```
city_1 = [
    [4, 3],
    [1, 2]
]

city_ = [
    [1, 2, 18, 3],
    [26, 6, 7, 15],
    [9, 10, 17, 18],
    [14, 15, 16, 22]
]

print(longest_decreasing_path(city_1))
print(longest_decreasing_path(city_2))
```

Example Output:

```
4
Example 1 Explanation: The longest decreasing path is 4 -> 3 -> 2 -> 1 which has
length 4

9
Example 2 Explanation: The longest decreasing path is 22 -> 18 -> 17 -> 16 -> 15
-> 10 -> 6 -> 2 -> 1
```

Close Section

▾ Standard Problem Set Version 2

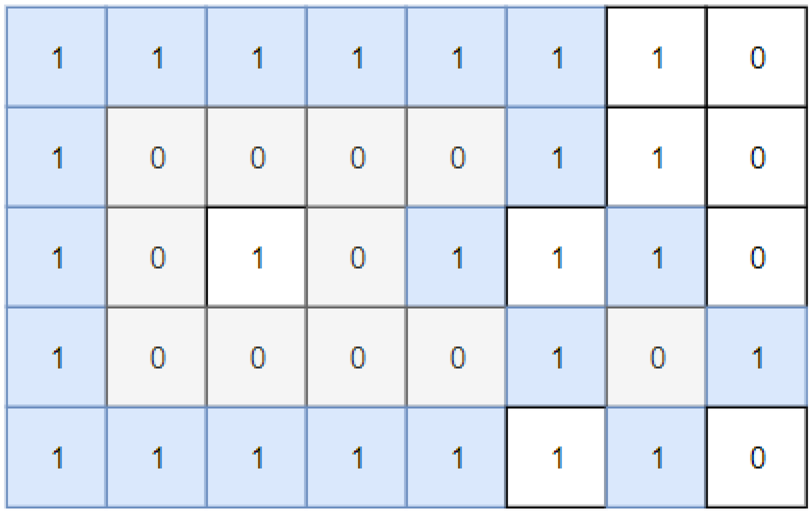
Problem 1: Number of Protected Towns

Youare given an `m x n` binary grid `kingdom` where a *town* is a maximally vertically/horizontally connected group of `0`s and a *protected town* is a town that is surround by `1`s on all sides.

Return the number of connected towns.

```
def count_protected(kingdom):
    pass
```

Example Usage 1:



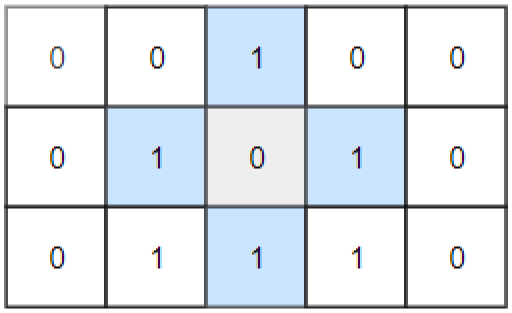
```
kingdom_1 = [
    [1,1,1,1,1,1,1,0],
    [1,0,0,0,0,1,1,0],
    [1,0,1,0,1,1,1,0],
    [1,0,0,0,0,1,0,1],
    [1,1,1,1,1,1,1,0]]

print(count_protected(kingdom_1))
```

Example Output 1:

```
2
Explanation: In the image above, islands in grey are closed because they are complet
```

Example 2:



```
kingdom_2 = [
    [0,0,1,0,0],
    [0,1,0,1,0],
    [0,1,1,1,0]]

print(count_protected(kingdom_2))
```

Example Output 2:

```
2
```

Problem 2: Cyclical Roads in Kingdom

Here's a themed version of the problem set in a kingdom context:

Your kingdom is represented by an `m x n` grid `kingdom`, where each square contains a character. Each unique character represents a different road in the kingdom.

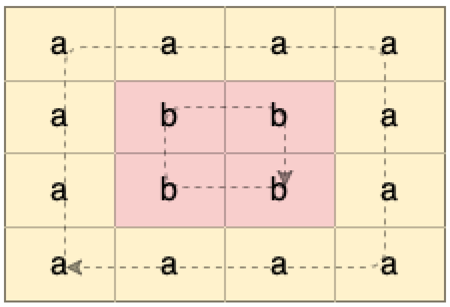
A *cyclical* road is a path of length 4 or more that starts and ends at the same square, where each square along the path contains the same road (character). You can move between squares in one of the four directions: up, down, left, or right, as long as the neighboring square contains the same symbol as the current one.

However, you cannot return to the square you just visited in the last move. For example, the cycle between `(row, column)` coordinates `(1, 1) -> (1, 2) -> (1, 1)` is invalid because you're returning directly to the last visited square.

Your task is to determine if there are any cyclical roads in the kingdom. Return `True` if a cycle exists, and `False` if no such cycle can be found.

```
def detect_cyclical_roads(kingdom):
    pass
```

Example Usage 1:



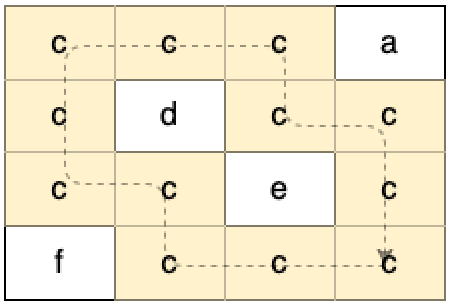
```
kingdom_1 = [
    ["a","a","a","a"],
    ["a","b","b","a"],
    ["a","b","b","a"],
    ["a","a","a","a"]]

print(detect_cyclical_roads(kingdom_1))
```

Example Output 1:

```
True
```

Example Usage 2:



c	c	c	a
c	d	c	c
c	c	e	c
f	c	c	c

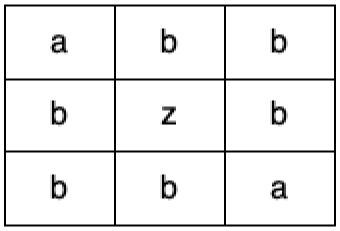
```
kingdom_2 = [
    ["c","c","c","a"],
    ["c","d","c","c"],
    ["c","c","e","c"],
    ["f","c","c","c"]]

print(detect_cyclical_roads(kingdom_2))
```

Example Output 2:

```
True
```

Example Usage 3:



a	b	b
b	z	b
b	b	a

```
kingdom_3 = [
    ["a","b","b"],
    ["b","z","b"],
    ["b","b","a"]]

print(detect_cyclical_roads(kingdom_3))
```

Example Output 3:

```
False
```

Problem 3: Escape the Dungeon

You find yourself trapped in a pitch-black castle dungeon that is represented by an `m x n` grid `dungeon`, where:

- 0 represents an open passage
- 1 represents a wall

You have the ability to move in one of four directions: up, down, left, or right. To prevent yourself from getting lost, once you start moving in a direction, you will continue moving in that direction until you hit a wall. After stopping, you can choose a new direction to continue your escape.

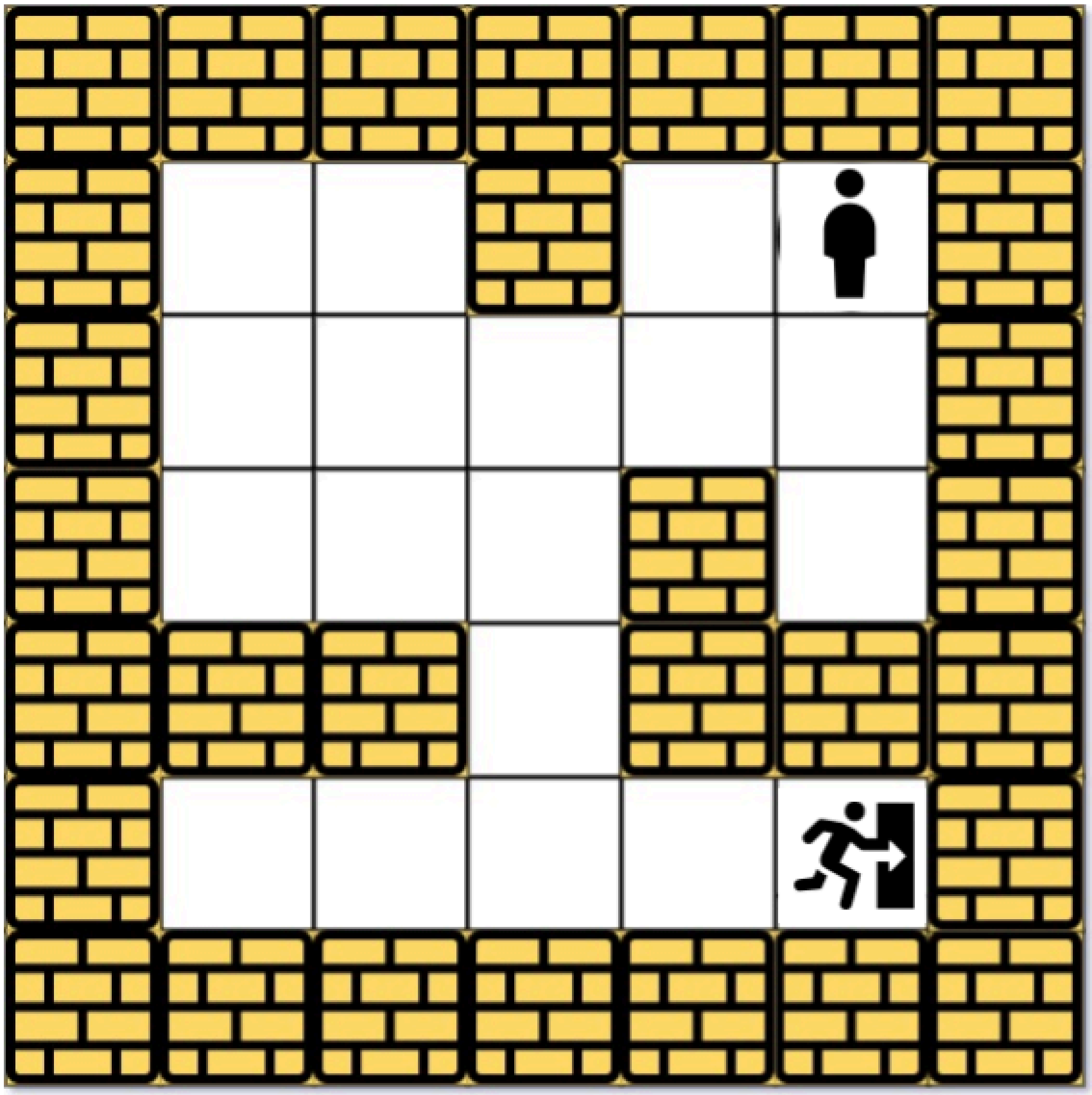
You are given your current `position` in the form `(start_row, start_col)` and the `exit` location in the form `(exit_row, exit_column)`.

Return `True` if you can stop at the exit; otherwise, return `False`.

You may assume the borders of the dungeon are all walls.

```python
def can_escape_dungeon(dungeon, position, exit):
    pass
```

Example Usage 1:



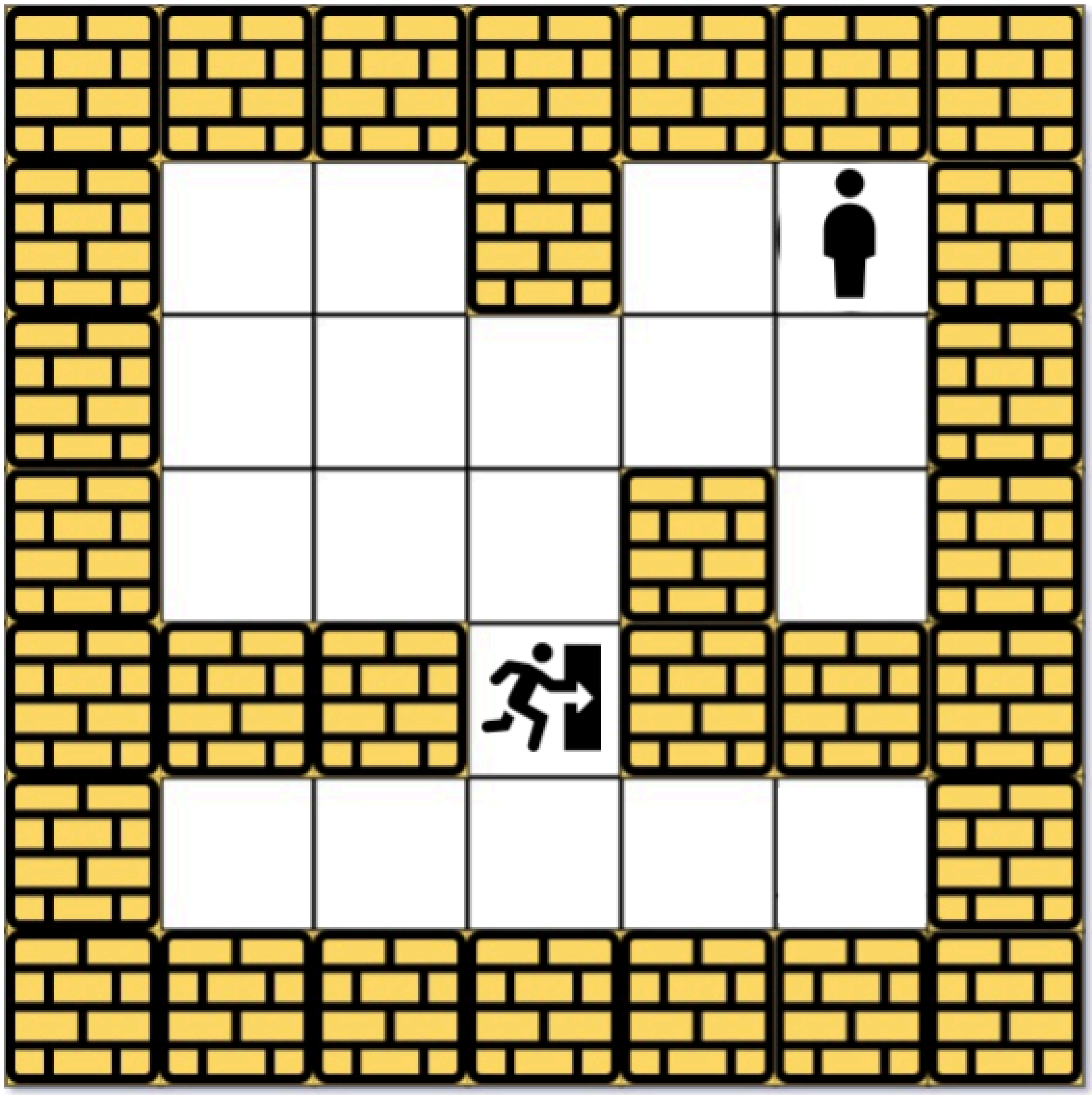
```
dungeon = [
    [0, 0, 1, 0, 0],
    [0, 0, 0, 0, 0],
    [0, 0, 0, 1, 0],
    [1, 1, 0, 1, 1],
    [0, 0, 0, 0, 0]
]

print(can_escape_dungeon(dungeon, (0, 4), (4, 4)))
```

Example Output 1:

```
True
Example 1 Explanation: You can escape the dungeon by rolling the following path:
  - Start at `(0, 4)`, roll left to `(0, 1)`, then roll down to `(4, 1)`, then roll
    to `(4, 4)`, where you stop exactly at the exit.
```

Example Usage 2:



```
dungeon = [
    [0, 0, 1, 0, 0],
    [0, 0, 0, 0, 0],
    [0, 0, 0, 1, 0],
    [1, 1, 0, 1, 1],
    [0, 0, 0, 0, 0]
]

print(can_escape_dungeon(dungeon, (0, 4), (3, 2)))
```

Example Output:

```
False
Example 2 Explanation: There is no way for the ball to stop at the destination.
Notice that you can pass through the destination but you cannot stop there.
```

Problem 4: Surveying the Kingdom

You are conducting an annual survey of your kingdom which is divided into `m x n` hectares of land, represented by a binary matrix `land`. Each hectare is either *forested land* (`0`) or *farmland* (`1`). As the kingdom expands, you've designated certain rectangular areas for farming, called *farmland groups*.

These farmland groups are rectangular plots consisting entirely of farmland, and no two farmland groups are adjacent to each other (farmland in one group is not horizontally/vertically adjacent to farmland in another group). To conduct a proper survey of the land, you need to identify the boundaries of each farmland group.

The kingdom is represented by a coordinate system where the top-left corner of the land is `(0, 0)` and the bottom-right corner is `(m-1, n-1)`. For each group of farmland, you must determine the coordinates of the top-left corner and the bottom-right corner.

A farmland group with a top-left corner at (r_1, c_1) and a bottom-right corner at (r_2, c_2) is represented by the 4-length array $[r_1, c_1, r_2, c_2]$.

Return a 2D array containing the coordinates of all the farmland groups. If there are no groups of farmland, return an empty array.

```
def find_farmland_groups(land):
    pass
```

Example Usage:

```
land = [
    [1, 0, 0],
    [1, 0, 1],
    [1, 1, 1]
]

print(find_farmland_groups(land))
```

Example Output:

```
[
    [0, 0, 2, 0],
    [1, 2, 2, 2],
]
Explanation:
- The first farmland group starts at `(0, 0)` and extends down to `(2, 0)`.
- The second farmland group starts at `(1, 2)` and extends down to `(2, 2)`.
```

Problem 5: Reinforce the Kingdom Walls

Your kingdom is represented by an `m x n` integer matrix `kingdom_grid`, where each square represents a fortified area with a particular defensive strength. The value at each square in the grid indicates the current level of fortification (color).

You are given three integers, `row`, `col`, and `new_strength`. The square at `kingdom_grid[row][col]` is part of a fortified section with a particular defensive strength, and you want to strengthen the border of this section.

The *border* of a section is defined as all the squares in the fortified section that either:

1. Are adjacent to a square with a different defensive strength, or
2. Lie on the outer edges of the kingdom.

Your task is to identify the fortified section containing `kingdom_grid[row][col]` and reinforce the border by updating its defensive strength to `new_strength`. Return the updated `kingdom_grid` after reinforcing the border.

From any square, you may move to adjacent squares in the four cardinal directions: up, down, left, and right. Two squares are considered part of the same fortified section if they have the same defensive strength and are adjacent.

```python
def reinforce_walls(kingdom_grid, row, col, new_strength):
    pass
```

Example Usage:

```python
kingdom_grid = [
    [1, 1, 1, 2],
    [1, 3, 1, 2],
    [1, 1, 1, 2]
]

print(reinforce_walls(kingdom_grid, 1, 1, 4))
```

Example Output:

```
[
    [1, 1, 1, 2],
    [1, 4, 1, 2],
    [1, 1, 1, 2]
]
```

Close Section

▸ Advanced Problem Set Version 1

▸ Advanced Problem Set Version 2