TIP102 | Intermediate Technical Interview Prep

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

Session 1: Matrices

Session Overview

In this session, students will learn how to use graph concepts and algorithms to solve 2D matrix problems.

> 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.

ur program leader/s will indicate which code sharing tool/s to use as a group, and will help break own 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: Escape to the Safe Haven

You've just learned of a safe haven at the bottom right corner of the city represented by an `m x n` matrix `grid`. However, the city is full of zombie-infected zones. Safe travel zones are marked on the grid as `1`s and infected zones are marked as `0`s. Given your current `position` as a tuple in the form `(row, column)`, return `True` if you can reach the safe haven traveling only through safe zones and `False` otherwise. From any zone (cell) in the `grid` you may move up, down, left, or right.

Evaluate the time and space complexity of your solution. Define your variables and provide a rationale for why you believe your solution has the stated time and space complexity. Assume the input tree is balanced when calculating time and space complexity.

```
def can_move_safely(position, grid):
    pass
```

Example Usage:

```
grid = [
    [1, 0, 1, 1, 0], # Row 0
    [1, 1, 1, 1, 0], # Row 1
    [0, 0, 1, 1, 0], # Row 2
    [1, 0, 1, 1, 1]  # Row 3
]

position_1 = (0, 0)
position_1 = (0, 4)
position_2 = (3, 0)

print(can_move_safely(position_1, grid))
print(can_move_safely(position_2, grid))
print(can_move_safely(position_3, grid))
```

Example Output:

```
True
Example 1 Explanation: Can follow the path (0, 0) -> (1, 0) -> (1, 1) -> (1, 2) ->
(2, 2) -> (3, 2) -> (3, 3) -> (3, 4)

True
Example 2 Explanation: Although we start in an unsafe position, we can immediately
arrive in a safe position and from there safely travel to the bottom right corner (3

False
```

▶ 💡 **Hint: Transforming Matrices into Graphs**

▶ 💡 **Hint: Create a Helper to Find Neighbors**

Problem 2: List All Escape Routes

Having arrived at the safe haven, you are immediately put to work evaluating how many civilians can be evacuated to the safe haven. Given an `m x n` `grid` representing the city, return a list of tuples of the form `(row, column)` representing every starting position in the `grid` from which there exists a valid path of safe zones (`1`s) to the safe haven in the bottom-right corner of the grid.

If the starting cell has value `0`, they are considered infected and cannot reach the safe haven.

```
def list_all_escape_routes(grid):
    pass
```

Example Usage:

```
grid = [
    [1, 0, 1, 0, 1], # Row 0
    [1, 1, 1, 1, 0], # Row 1
    [0, 0, 1, 0, 0], # Row 2
    [1, 0, 1, 1, 1]  # Row 3
]

print(list_all_escape_routes(grid))
```

Example Output:

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

▶ 💡 **Hint: Repeating Traversal**

Problem 3: Zombie Spread

The zombie infection is spreading rapidly! Given a city represented as a 2D `grid` where `0` represents an obstacle where neither humans nor zombies can live, `1` represents a human safe zone and `2` represents a zone that has already been infected by zombies, determine how long it will take for the infection to spread across the entire city.

The infection spreads from each infected zone to its adjacent safe zones (up, down, left, right) in one hour. Return the number of hours it takes for all safe zones to be infected. If there are still safe zones remaining after the infection has spread everywhere it can, return `-1`.

```
def time_to_infect(grid):
    pass
```

Example Usage:

Example 1:

```
grid_1 = [
        [2,1,1],
        [1,1,0],
        [0,1,1]]

grid_2 = [
        [2,1,1],
        [0,1,1],
        [1,0,1]]

grid_3 = [[0,2]]

print(time_to_infect(grid_1))
print(time_to_infect(grid_2))
print(time_to_infect(grid_3))
```

Example Output:

```
4
Example 1 Explanation: See image included above.

-1
Example 2 Explanation: The safe zone in the bottom left corner (row 2, column 0)
is never infected because infection only happens up, left, right, and down.

0
Example 3 Explanation: Since there are already no safe zones at minute 0, the answer
```

Problem 4: 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:

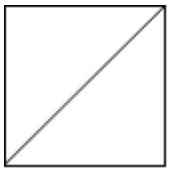
- A `'/'` or `'\\'` (forward or backslash) represents a *fence* dividing the square into two triangular zones.
- A `' '` represents an *open area* with no division in the square.

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.

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

Example Usage 1:



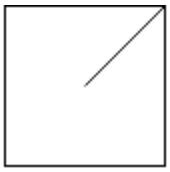
```
grid_1 = [" /","/ "]

print(count_regions(grid_1))
```

Example Output 1:

```
2
```

Example Usage 2:



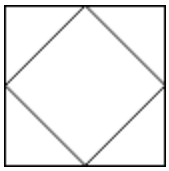
```
grid_2 = [" /","  "]

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 5: 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.

```
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.
```

Close Section

▾ Standard Problem Set Version 2

Problem 1: Castle Path

Your kingdom is represented by an `m x n` matrix `kingdom`. Each square in the matrix represents a different town in the kingdom. You wish to travel from a starting position `town` to the `castle`, however several towns have been overrun by bandits.

Towns that are safe to travel through are marked with `X`s and towns with dangerous bandits are marked with `0`s.

Given your current `town` and the `castle` location as tuples in the form `(row, column)`, return a list of tuples representing the shortest path from your `town` to the `castle` without traveling through any towns with bandits. If there are multiple paths with the shortest length, you may return any path. If no such path exists, return `None`.

From any town in the `grid` you may move to the neighboring towns up, down, left, or right. You may not move out of bounds of the `kingdom`.

Evaluate the time and space complexity of your solution. Define your variables and provide a rationale for why you believe your solution has the stated time and space complexity. Assume the input tree is balanced when calculating time and space complexity.

```
def path_to_castle(kingdom, town, castle):
    pass
```

Example Usage:

```
grid = [
    ['X', 'O', 'X', 'X', 'O'], # Row 'O'
    ['X', 'X', 'X', 'X', 'O'], # Row 1
    ['O', 'O', 'X', 'X', 'O'], # Row 2
    ['X', 'O', 'X', 'X', 'X']  # Row 3
]

town_1 = (0, 0)
town_2 = (0, 4)
town_3 = (3, 0)

print(path_to_castle(town_1, grid))
print(path_to_castle(town_2, grid))
print(path_to_castle(town_3, grid))
```

Example Output:

```
[0, 0, (10)]
Example 1 Explanation: Can follow the path (0, 0) -> (1, 0) -> (1, 1) -> (1, 2) ->
(2, 2) -> (3, 2) -> (3, 3) -> (3, 4)

True
Example 2 Explanation: Although we start in an unsafe position, we can immediately
arrive in a safe position and from there safely travel to the bottom right corner (3

False
```

▶ 💡 **Hint: Transforming Matrices into Graphs**

▶ 💡 **Hint: Create a Helper to Find Neighbors**

Problem 2: Walls and Gates

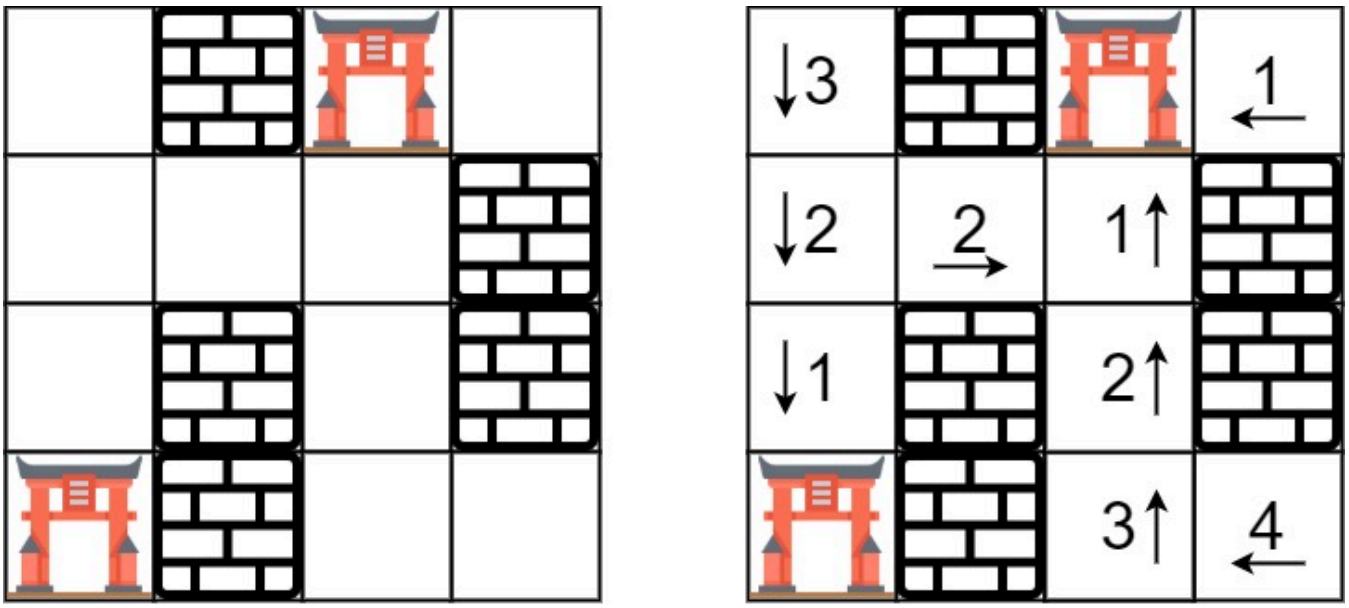
You have an `m x n` grid `castle` where each square represents a section of the castle. Each square has one of three possible values:

- `1` : a wall or an obstacle
- `0` : a gate
- `float('inf')` (infinity): an empty room

Return the `castle` matrix modified in-place such that each empty rooms value is its distance to its nearest gate. If it is impossible to reach a gate, it should have value infinity.

```
def walls_and_gates(castle):
    pass
```

Example Usage:




```
castle = [
    [float('inf'), -1, 0, float('inf')],            # Row 0
    [float('inf'), float('inf'), float('inf'), -1], # Row 1
    [float('inf'), -1, float('inf'), -1],           # Row 2
    [0, -1, float('inf'), float('inf')]             # Row 3
    ]

print(walls_and_gates(castle))
```

Example Output:

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

▶ 💡 **Hint: Repeating Traversal**

Problem 3: Surrounded Regions

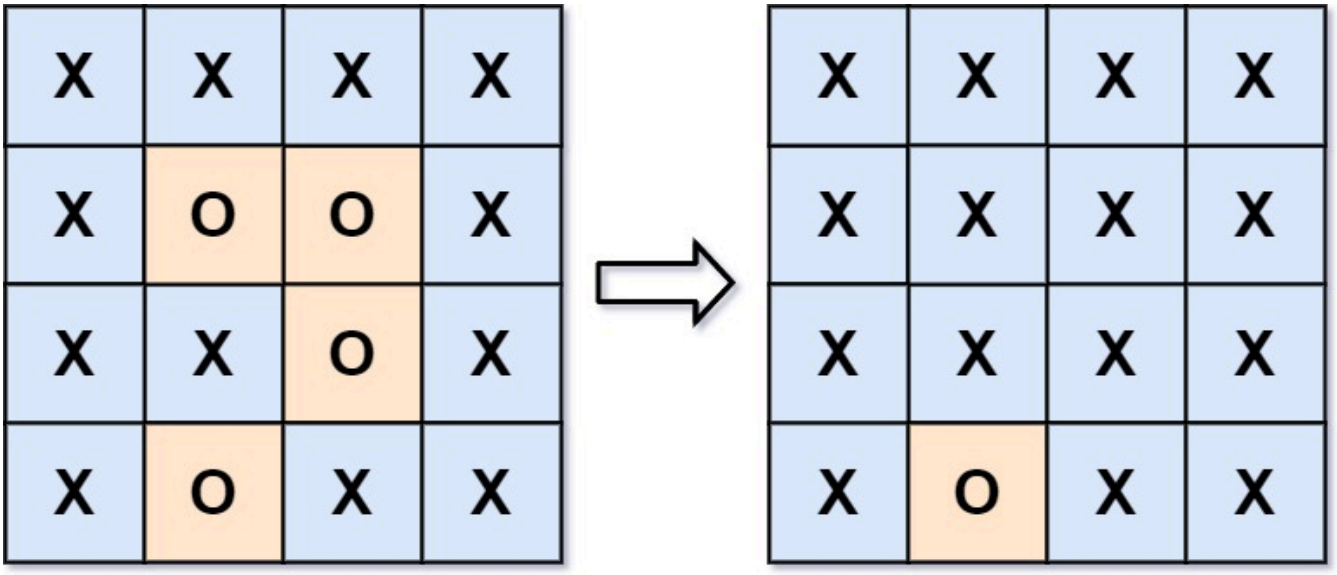
Your kingdom has been battling a neighboring kingdom. You are given an `m x n` matrix `map` containing letters `'X'` and `'O'`. Territory controlled by your kingdom is labeled with `'X'` while territory controlled by the opposing kingdom is labeled `'O'`.

Territories (cells) in the matrix are considered connected to horizontally and vertically adjacent territories. A *region* is formed by contiguously connected territories controlled by the same kingdom. Your kingdom can capture an `'O'` region if it is surrounded. The region is surrounded with `'X'` territories if you can connect the region with `'X'` cells and none of the region cells are on the edge of the `map`.

A surrounded region is captured by replacing all `'O'`s with `'X'`s in the input matrix `map`. Return `map` after modifying it in-place to capture all possible `'O'` regions.

```
def capture(map):
    pass
```

Example Input:




```
map = [
    ["X","X","X","X"],
    ["X","O","O","X"],
    ["X","X","O","X"],
    ["X","O","X","X"]]

print(capture(map))
```

Example Output:

```
[
    ["X","X","X","X"],
    ["X","X","X","X"],
    ["X","X","X","X"],
    ["X","O","X","X"]
    ]
Example Explanation: The bottom region cannot be captured because it is on the edge
of the board and cannot be surrounded.
```

Problem 4: Maximum Number of Troops Captured

You are given a 2D matrix `battlefield` of size `m x n`, where `(row, column)` represents:

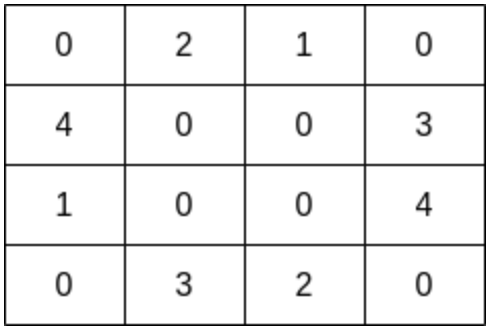
- An impassable obstacle if `battlefield[row][column] = 0`, or
- An square containing `battlefield[row][column]` enemy troops, if `battlefield[row][column] > 0`.

Your kingdom can start at any non-obstacle square `(row, column)` and can do the following operations any number of times: - Capture all the troops at square `battlefield[row][column]` or - Move to any adjacent cell with troops up, down, left, or right.

Return the maximum number of troops your kingdom can capture if they choose the starting cell optimally. Return `0` if no troops exist on the `battlefield`.

```
def capture_max_troops(battlefield):
    pass
```

Example Usage 1:



0	2	1	0
4	0	0	3
1	0	0	4
0	3	2	0

```
battlefield_1 = [
    [0,2,1,0],
    [4,0,0,3],
    [1,0,0,4],
    [0,3,2,0]]

print(capture_max_troops(battlefield_1))
```

Example Output 1:

```
7
Example 1 Explanation: You can start at square (1, 3) and capture 3 troops, then
move to square (2, 3) and capture 4 troops.
```

Example Usage 2:

1	0	0	0
0	0	0	0
0	0	0	0
0	0	0	1

```
battlefield_2 = [
    [1,0,0,0],
    [0,0,0,0],
    [0,0,0,0],
    [0,0,0,1]]

print(capture_max_troops(battlefield_2))
```

Example Output 2:

```
1
Example 2 Explanation: You can start at square (0,0) or (3,3) and capture a single
troop.
```

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.

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

Example Input:

```
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