Topic Introduction

Today's PrepLetter is all about *grid traversal with dynamic programming*. This is a classic interview theme that shows up in many forms, usually involving finding the best or most numerous way to move from one part of a grid to another. Let's break it down!

What's the concept?
Dynamic programming (DP) is a way to solve problems by breaking them into smaller subproblems and reusing the solutions to those subproblems. When applied to grids, DP typically helps us count paths, minimize or maximize values, or determine reachability, all while avoiding redundant computation.

How does it work?
Imagine a grid (think of a chessboard). You're often allowed to move in certain directions, like right or down. For each cell, you want to know something: maybe the minimum cost to reach it, or how many ways you can arrive there. DP helps by letting you build up answers for each cell using the answers from previously computed cells.

Why is this useful in interviews?
Grid DP problems test your ability to spot overlapping subproblems and optimal substructure—two big DP ideas. They're approachable but can trip you up if you miss an edge case. Plus, they often have real-world analogs, so strong DP skills make you a better engineer.

A quick example (not from today's problems):
Say you want to count how many ways you can reach the bottom-right of a 3x3 grid, starting at the top-left, moving only right or down. For each cell, the number of ways to get there is the sum of ways to get to the cell above and the cell to the left. This is classic DP: you build up the answer, using previous results.

Why These Problems Go Together

Today's three problems are all about *traversing a grid with dynamic programming*:

- **Unique Paths** counts all possible ways from top-left to bottom-right.
- **Unique Paths II** adds obstacles that can block your way.
- **Minimum Path Sum** asks for the path with the smallest total value.

They share the same DP grid pattern, but with unique twists. If you master one, you're on the way to mastering them all!

Problem 1: Unique Paths

LeetCode Link

Problem Statement (in Plain English)

Given an `m x n` grid, you start in the top-left cell and want to reach the bottom-right cell. You can only move **right** or **down** at each step. How many different paths are there from start to finish?

Example

Suppose `m = 3`, `n = 2`:

Grid:

```
[ S   . ]
[ .   . ]
[ .   E ]
```

`S` = start, `E` = end.

Output: `3`

Explanation:
The three paths are:

- Right, Right, Down, Down
- Right, Down, Right, Down
- Down, Right, Right, Down

Wait, the grid is 3 rows and 2 columns, so possible paths:

- Right, Down, Down
- Down, Right, Down
- Down, Down, Right

So, paths = 3

Walkthrough

For each cell, the number of ways to get there is the sum of:

- The number of ways to get to the cell above (if any)
- The number of ways to get to the cell to the left (if any)

For the first row and first column, there's only one way to get to each cell: straight from the start along the edge.

Try This Test Case

`m = 4`, `n = 3`
How many unique paths?

Brute-force Approach

Try every possible sequence of right and down moves. With recursion, this is exponential: `O(2^(m+n))`. Not practical as grid size grows!

Optimal DP Approach

Let's use a 2D DP table.

Let `dp[i][j]` be the number of unique paths to cell `(i, j)`.

- If `i == 0` or `j == 0`, `dp[i][j] = 1` (only 1 way along the edge)
- Otherwise, `dp[i][j] = dp[i-1][j] + dp[i][j-1]`

We fill the table row by row, left to right.

Python Solution

```
def uniquePaths(m, n):
    # Create a 2D DP table with m rows and n columns, initialized to 1
    dp = [[1] * n for _ in range(m)]

    # Start from cell (1, 1) since first row and column are already set
    for i in range(1, m):
        for j in range(1, n):
            # Paths to (i, j) = from above + from left
            dp[i][j] = dp[i-1][j] + dp[i][j-1]

    # The answer is in the bottom-right cell
    return dp[m-1][n-1]
```

Time Complexity: O(m * n)

Space Complexity: O(m * n) (can be improved to O(n) with a 1D array)

Code Explanation

- We set up a `dp` table where each cell represents the number of ways to reach that cell from the start.
- The first row and first column are filled with 1s, since only one way exists to reach those.
- For every other cell, the value is the sum of the cell above and the cell to the left.
- The answer is the value in the bottom-right cell.

Example Trace (`m = 3`, `n = 2`):

Initial `dp`:

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

Fill in row 1, col 1:

`dp[1][1] = dp[0][1] + dp[1][0] = 1 + 1 = 2`

Now:

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

Next, row 2, col 1:

`dp[2][1] = dp[1][1] + dp[2][0] = 2 + 1 = 3`

Final table:

```
[1, 1]
[1, 2]
[1, 3]
```

Answer: `dp[2][1] = 3`

Try This Yourself

What about `m = 5`, `n = 3`? How many unique paths?

Take a Moment

Try solving this by hand and writing your own code before reading on!

Problem 2: Unique Paths II

LeetCode Link

Problem Statement

You're given an `m x n` grid with some obstacles (cells marked as 1). You still want to count unique paths from the top-left to the bottom-right, moving only right or down. You cannot step on or through an obstacle.

How This Differs

This is almost identical to Unique Paths, but now you may have to avoid cells!
Obstacles turn some cells into dead-ends.

Example

Grid:

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

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

Output: 2

Explanation: Two paths, both go around the obstacle at (1,1).

Brute-force

Recursively try all paths, but skip any path that hits an obstacle. Still exponential time.

Optimal DP Approach

Same as before, but:

- If a cell is an obstacle, `dp[i][j] = 0` (no way to step here)
- Otherwise, `dp[i][j] = dp[i-1][j] + dp[i][j-1]` (as before)

Special case: If the start or end cell is blocked, answer is 0.

Pseudocode

```
If start cell is obstacle: return 0
dp[0][0] = 1

For each cell (i, j):
    If obstacleGrid[i][j] == 1:
        dp[i][j] = 0
    Else if i == 0 and j == 0:
        continue
    Else:
        dp[i][j] = (dp[i-1][j] if i > 0 else 0) + (dp[i][j-1] if j > 0 else 0)

Return dp[m-1][n-1]
```

Example Trace

For the above grid, the DP table builds up like this:

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

Another Test Case

Try:

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

Answer: 1

Dry Run

Try building the DP table for this test case.

Time and Space Complexity

- Time: O(m * n)
- Space: O(m * n) (can be reduced to O(n))

Problem 3: Minimum Path Sum

LeetCode Link

Problem Statement

Given an m x n grid filled with non-negative numbers, find a path from top-left to bottom-right which minimizes the sum of all numbers along its path. You can only move right or down.

What's Different

Instead of counting paths, you want the *minimum sum*. Obstacles are now replaced by costs.

Example

Input:

```
[
  [1,3,1],
  [1,5,1],
  [4,2,1]
]
```

Output: 7

Explanation: Path is 1 → 3 → 1 → 1 → 1

Brute-force

Try every possible path, sum their values, keep the minimum. Exponential time.

DP Approach

Let `dp[i][j]` be the minimum path sum to reach cell `(i, j)`.

- `dp[0][0] = grid[0][0]`
- For each cell:
  - If first row, `dp[0][j] = dp[0][j-1] + grid[0][j]`
  - If first column, `dp[i][0] = dp[i-1][0] + grid[i][0]`
  - Else, `dp[i][j] = min(dp[i-1][j], dp[i][j-1]) + grid[i][j]`

Pseudocode

```
dp[0][0] = grid[0][0]
For i in 1 to m-1:
    dp[i][0] = dp[i-1][0] + grid[i][0]
For j in 1 to n-1:
    dp[0][j] = dp[0][j-1] + grid[0][j]
For i in 1 to m-1:
    For j in 1 to n-1:
        dp[i][j] = min(dp[i-1][j], dp[i][j-1]) + grid[i][j]
Return dp[m-1][n-1]
```

Try This Test Case

```
[
  [1,2,3],
  [4,5,6]
]
```

Expected output: 12

Dry Run Guidance

Build the DP table step by step and see how the minimums are chosen.

Time and Space Complexity

- Time: O(m * n)
- Space: O(m * n) (can be optimized to O(n))

Summary and Next Steps

These three problems are grouped together because they all use grid-based dynamic programming to traverse from the top-left to the bottom-right cell. You learned how:

- To use DP to count unique ways to traverse a grid
- To adapt that approach for obstacles
- To use DP to optimize for minimum sum, not just count paths

Key Patterns and Insights:

- Build a DP table where each cell depends only on its top and left neighbors
- Handle special cases: obstacles, blocked start/end cells, or minimum/maximum instead of counts
- You can often reduce space from O(mn) to O(n) if you process rows one at a time

Common Mistakes:

- Forgetting to handle obstacles or blocked cells properly
- Not initializing the first row or column correctly
- Confusing when to sum vs. when to take min/max

Action List

- Solve all 3 problems on your own — even the one with code provided!
- Try to implement Problem 2 and 3 with a 1D array for space optimization.
- Explore related problems like "Triangle Minimum Path Sum" or "Edit Distance" for more DP practice.
- Compare your solutions with others for style and edge cases.
- If you get stuck, take a break and revisit with fresh eyes — DP takes practice!

Happy coding, and remember: DP grids are a superpower in interviews!