Topic Introduction

Today's Theme: Dynamic Programming on Grids and Triangles

Dynamic Programming (DP) is a powerful technique for solving complex problems by breaking them down into overlapping subproblems, storing solutions to subproblems, and combining them to solve the original problem. One particularly common setting for DP is when you need to make optimal choices while traversing a grid, triangle, or matrix—often to maximize or minimize a path sum.

What does this look like?

Imagine you're standing at the top of a mountain, and below you is a snowy slope arranged in a triangle. At each step, you can ski down to one of two positions beneath you. Each patch of snow has its own cost or reward. The challenge: find the way down with the lowest total cost.

This is the heart of today's topic: **DP for path optimization in grids and triangles**. The key idea is to use previously computed optimal paths to build up the solution, moving either from the bottom up or from the top down.

Why is this so useful in interviews?

- **Common pattern:** Grid/triangle DP is a classic interview pattern.
- **Tests understanding of recursion, memoization, tabulation.**
- **Real-world application:** Pathfinding, scheduling, resource allocation, and more.
- **Great for edge case reasoning:** How do you handle boundaries? What about obstacles?

Quick Example (not one of our main problems):

Suppose you have a 2x2 grid:

```
1 3
2 4
```

Find the minimal path sum from the top-left to bottom-right, moving only right or down.

Using DP, you build up the minimal path sum at each cell:

- Top-left (1): 1
- Move right: 1 + 3 = 4
- Move down: 1 + 2 = 3
- Bottom-right: min(4+4, 3+4) = 7

This is the essence: breaking the problem into smaller steps, using past results to build up the answer.

Why These Problems Are Grouped Together

Today's trio—Triangle, Minimum Falling Path Sum, and Cherry Pickup—are all about **optimizing a path through a grid or triangle** using DP.

- **Triangle:** Find the minimum path sum from top to bottom of a triangle.

• **Minimum Falling Path Sum:** Find the minimum sum from top to bottom in a square matrix, where you can move down, down-left, or down-right.

• **Cherry Pickup:** Collect as many cherries as possible by moving from the top-left to the bottom-right and back in a grid, picking up cherries only once.

They all use similar grid DP patterns, but increase in complexity and subtlety. Mastering them will sharpen your skills for a huge class of interview questions!

Problem 1: Triangle

Problem:

Leetcode 120: Triangle

Given a triangle array, return the minimum path sum from top to bottom. At each step, you may move to adjacent numbers on the row below.

Example:

```
Input:
[
     [2],
    [3,4],
   [6,5,7],
  [4,1,8,3]
]
Output: 11
Explanation: 2 + 3 + 5 + 1 = 11
```

How to Approach:

• Imagine starting at the top. At each row, you can move to the same position or one to the right in the row below.

• Your goal: pick a path down that gives the smallest sum.

Try This Manually:

Input:

```
[
   [1],
  [2,3],
 [3,6,7],
[8,9,6,1]
]
```

What's the minimum path sum?

Brute-Force Approach:

• Try all possible paths from top to bottom.

- At each element, branch to both possibilities in the next row.
- Time complexity: Exponential (O(2^n)). Not practical for large triangles.

Optimal Approach: Bottom-Up DP

- Start from the bottom row and work upwards.
- At each cell, choose the minimum sum of its two children and add the current value.
- Replace the triangle's rows as you go, or use a separate array.

Step-by-Step Logic

- Copy the last row as your initial DP array.
- For each row above, update each cell with its value plus the minimum of its two "children" below.
- The top element of the DP array will contain the answer.

Python Solution:

```
def minimumTotal(triangle):
    # Copy the last row as our DP array
    dp = triangle[-1][:]
    # Start from the second last row and move upwards
    for row in range(len(triangle) - 2, -1, -1):
        for col in range(len(triangle[row])):
            # For each cell, choose the min path from its two children
            dp[col] = triangle[row][col] + min(dp[col], dp[col + 1])
    return dp[0]
```

Time Complexity: O(n^2) where n is the number of rows.
Space Complexity: O(n), for the DP array.

How Does This Work?

- We reduce the triangle problem to repeated choices between two possibilities.
- By working upwards, we ensure every subproblem is solved before it's needed.
- The DP array always represents the minimal path sum from the current row to the bottom.

Trace Example:

Let's trace with the main example:

```
[
     [2],
    [3,4],
   [6,5,7],
  [4,1,8,3]
]
```

Start with last row: `[4,1,8,3]`

- Move up to `[6,5,7]`:
  - For 6: 6 + min(4,1) = 6 + 1 = 7

- For 5: 5 + min(1,8) = 5 + 1 = 6
- For 7: 7 + min(8,3) = 7 + 3 = 10
- Now dp = `[7,6,10,3]`

- Now `[3,4]`:
  - For 3: 3 + min(7,6) = 3 + 6 = 9
  - For 4: 4 + min(6,10) = 4 + 6 = 10
  - Now dp = `[9,10,10,3]`
- Now `[2]`:
  - 2 + min(9,10) = 2 + 9 = 11

Answer: `11`

Try This Test Case:

```
[
  [5],
 [9,6],
[4,6,8],
[0,7,1,5]
]
```

What is the minimum path sum? Try drawing it out!

Take a moment to solve this on your own before jumping into the solution.

Problem 2: Minimum Falling Path Sum

Problem:

Leetcode 931: Minimum Falling Path Sum

Given a square matrix of integers, find the minimum falling path sum from any cell in the first row to any cell in the last row. At each step, you may move down, down-left, or down-right.

How is this Similar or Different?

- **Similar:** Both problems use DP to find optimal paths in a 2D structure.
- **Different:** Here you can start from any cell in the top row, and at each step, you have three choices instead of two. The structure is a matrix, not a triangle.

Example Input:

```
matrix = [
  [2,1,3],
  [6,5,4],
  [7,8,9]
]
```

Output: 13

Explanation: Path: 1 -> 4 -> 8 = 1+4+8=13

Try This Test Case:

```
matrix = [
  [1,2,3],
  [4,5,6],
  [7,8,9]
]
```

What is the minimum falling path sum?

Brute-Force Approach:

- Try every possible path starting from every cell in the first row.
- At each step, branch to up to three choices.
- Time complexity: Exponential (O(3^n)), not efficient for large matrices.

Optimal Approach: Bottom-Up DP

- Build a DP array of the same size as the matrix.
- Start from the bottom row, working upwards.
- For each cell, its minimum path is the value plus the minimum of its three possible children in the row below.

Step-by-Step Logic:

- Copy the last row as the initial DP array.
- For each row above, update each cell:
  - For each cell in the row, check the three possible child positions (down, down-left, down-right).
  - Choose the minimum and add the current cell's value.
- The answer is the minimum value in the top row after processing.

Pseudocode:

```
dp = last row of matrix
for row = n-2 to 0:
    for col = 0 to n-1:
        min_child = min(
            dp[col],               # directly below
            dp[col-1] if col > 0,  # down-left
            dp[col+1] if col < n-1 # down-right
        )
        dp[col] = matrix[row][col] + min_child
return min(dp)
```

Example Trace:

Input:

```
[
  [2,1,3],
  [6,5,4],
  [7,8,9]
]
```

Start with `[7,8,9]`

- Next row up: `[6,5,4]`
  - col=0: 6 + min(7,8) = 6+7=13
  - col=1: 5 + min(7,8,9) = 5+7=12
  - col=2: 4 + min(8,9) = 4+8=12
  - dp = `[13,12,12]`
- Next row up: `[2,1,3]`
  - col=0: 2 + min(13,12) = 2+12=14
  - col=1: 1 + min(13,12,12) = 1+12=13
  - col=2: 3 + min(12,12) = 3+12=15
  - dp = `[14,13,15]`

Answer: `min(14,13,15) = 13`

Try This Test Case:

```
[
  [10,8,12,5],
  [1,2,3,4],
  [6,1,2,3],
  [4,5,6,7]
]
```

What's the minimum falling path sum?

Time Complexity: O(n^2)
Space Complexity: O(n)

Problem 3: Cherry Pickup

Problem:

Leetcode 741: Cherry Pickup

Given a square grid of 0s and 1s (with -1 for thorns), collect the maximum number of cherries by traveling from the top-left to the bottom-right, and then back again, picking up each cherry only once.

What's Different or More Challenging?

- Now you need to **optimize two trips**: out and back.
- You must avoid picking up the same cherry twice.
- There are obstacles (-1) that block your path.
- This problem is much more complex and requires **multi-dimensional DP**.

Example Input:

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

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

Output: 5

Explanation:

- Path 1: (0,0) -> (0,1) -> (1,1) -> (2,1) -> (2,2)
- Path 2 (back): (2,2) -> (1,2) -> (0,2) [blocked], so must find a way that does not overlap.
- Optimal: 5 cherries picked up.

Try This Test Case:

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

How many cherries can be collected?

Brute-Force Approach:

- Try every possible path for both trips, remembering which cherries have been picked.
- Time complexity: Exponential, not feasible.

Optimal Approach: DP with Multiple Agents

- Imagine **two people** start at (0,0) and try to reach (N-1,N-1) at the same time, simulating both trips together.
- At each step, both move right or down.
- Use a 3D DP: dp[r1][c1][c2] = max cherries collected when person 1 at (r1,c1) and person 2 at (r2,c2), with r2 = r1 + c1 - c2 (since total steps are the same).
- Add cherries at both positions, but if they overlap, only count once.
- Skip any path that hits a thorn (-1).

Pseudocode:

```
function dp(r1, c1, c2):
    r2 = r1 + c1 - c2
    if out of bounds or thorn at (r1,c1) or (r2,c2):
        return -infinity
    if both at (N-1,N-1): return grid[N-1][N-1]
    if result is memoized: return it
    cherries = grid[r1][c1]
    if (r1, c1) != (r2, c2): cherries += grid[r2][c2]
    max_next = max(
        dp(r1+1, c1, c2+1), # person1 down, person2 right
        dp(r1, c1+1, c2+1), # person1 right, person2 right
        dp(r1+1, c1, c2),   # person1 down, person2 down
        dp(r1, c1+1, c2)    # person1 right, person2 down
    )
```

```
    result = cherries + max_next
    memoize and return result
```

Time Complexity: O(n^3)
Space Complexity: O(n^3)

Trace Example:

For a simple 3x3 grid (see example above), try filling in the DP table recursively, and notice how the two agents simulate both trips.

Try This Test Case:

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

How many cherries can be picked?

Give it a try! Draw the grid, simulate the two agents' paths, and see how memoization can help.

Reflection Prompt:
Notice how this problem builds on the previous DP grid patterns, but requires a deeper level of memoization and state tracking. Can you think of other problems where *multi-agent* or *multi-dimensional* DP is useful?

Summary and Next Steps

Today, we tackled a family of problems that share a **grid/triangle DP path optimization pattern**:

- **Triangle:** DP bottom-up from a triangle's base to its tip.
- **Minimum Falling Path Sum:** DP on a square grid, with three movement choices per cell.
- **Cherry Pickup:** Multi-agent DP, simulating two trips with state-tracking and obstacles.

Key Patterns & Insights:

- Build solutions from base cases upward (bottom-up DP).
- Use memoization or tabulation to avoid redundant computation.
- When multiple agents/paths are involved, expand your DP state dimensions.
- Be careful with boundaries, obstacles, and overlapping paths.

Common Mistakes:

- Forgetting to handle edges/corners in the grid.
- Not memoizing enough state (especially with multiple agents).
- Double-counting or missing cherries in Cherry Pickup.

Action List

- **Solve all 3 problems on your own** — even the one with code provided here.

• **Try alternate approaches**: Can you solve Triangle or Minimum Falling Path Sum with top-down (recursive + memoization) DP?

• **Experiment with variations**: What if you could move left, or up? How would that change your DP?

• **Compare your solutions** with others, especially how they handle tricky cases.

• **If you get stuck, don't worry** — review the step-by-step logic, and try to trace smaller test cases with pen and paper.

Keep practicing — DP is a superpower for interviews!