Minimum Path Sum

Think "shortest path on a DAG" where edges only go right or down.

Core idea (recurrence):

Let best[i][j] be the minimum path sum to reach cell (i, j) from (0, 0).

Transition (only from top to left):

- For interior cells:

  best[i][j] = grid[i][j] + min(best[i-1][j], best[i][j-1]).

- First row (no top):

  best[0][j] = grid[0][j] + best[0][j-1]

- First column (no left):

  best[i][0] = grid[i][0] + best[i-1][0]

- Base: best[0][0] = grid[0][0].

Answer: best[m-1][n-1].

Space-optimized versions:

You don't need the whole table:

1) One row DP (O(n) extra space):

   Keep a 1D array dp[j] meaning "min sum to current row's column j".

   - Initialize for row 0 by running cumulative sums across columns.
   - For each next row i:
     - Update dp[0] += grid[i][0] (only from above).
     - For j = 1 .. n-1:

       dp[j] = grid[i][j] + min(dp[j] /* from above */,
                                dp[j-1] /* from left */)

     Final answer ends in dp[n-1].

2) In-place (O(1) extra space):

   If you're allowed to modify grid, you can store best back into grid:

   - Accumulate along first row and first column
   - For each interior (i, j), write:

     grid[i][j] += min(grid[i-1][j], grid[i][j-1]).

   - Return grid[m-1][n-1].

Why it works:

Because moves only go right / down, each cell depends only on already computed cells (top / left). That makes the grid a DAG with a natural topological order.

Optional: recover the path:

If you need the actual path, either:

- Keep a parent pointer (came-from: up or left) while filling best, then backtrack from $(m-1, n-1)$ or
- Reconstruct afterwards by walking backward: at (i,j), move to the neighbor (up or left) whose best is smaller (ties are fine).