Approach #1: Greedy [Wrong Answer]

**Intuition**

Let's find the most cherries we can pick up with one path, pick them up, then find the most cherries we can pick up with a second path on the remaining field.

Though a counter example might be hard to think of, this approach fails to find the best answer to this case:

11100

00101

10100

00100

00111

**Algorithm**

We can use dynamic programming to find the most number of cherries dp[i][j] that can be picked up from any location (i, j) to the bottom right corner. This is a classic question very similar to [Minimum Path Sum](https://leetcode.com/problems/minimum-path-sum/description/), refer to the link if you are not familiar with this type of question.

After, we can find an first path that maximizes the number of cherries taken by using our completed dp as an oracle for deciding where to move. We'll choose the move that allows us to pick up more cherries (based on comparing dp[i+1][j] and dp[i][j+1]).

After taking the cherries from that path (and removing it from the grid), we'll take the cherries again.

**Complexity Analysis**

* Time Complexity: O(N^2)*O*(*N*2), where N*N* is the length of grid. Our dynamic programming consists of two for-loops of length N.
* Space Complexity: O(N^2)*O*(*N*2), the size of dp.

Approach #2: Dynamic Programming (Top Down) [Accepted]

**Intuition**

Instead of walking from end to beginning, let's reverse the second leg of the path, so we are only considering two paths from the beginning to the end.

Notice after t steps, each position (r, c) we could be, is on the line r + c = t. So if we have two people at positions (r1, c1) and (r2, c2), then r2 = r1 + c1 - c2. That means the variables r1, c1, c2 uniquely determine 2 people who have walked the same r1 + c1 number of steps. This sets us up for dynamic programming quite nicely.

**Algorithm**

Let dp[r1][c1][c2] be the most number of cherries obtained by two people starting at (r1, c1) and (r2, c2) and walking towards (N-1, N-1) picking up cherries, where r2 = r1+c1-c2.

If grid[r1][c1] and grid[r2][c2] are not thorns, then the value of dp[r1][c1][c2] is (grid[r1][c1] + grid[r2][c2]), plus the maximum of dp[r1+1][c1][c2], dp[r1][c1+1][c2], dp[r1+1][c1][c2+1], dp[r1][c1+1][c2+1] as appropriate. We should also be careful to not double count in case (r1, c1) == (r2, c2).

Why did we say it was the maximum of dp[r+1][c1][c2] etc.? It corresponds to the 4 possibilities for person 1 and 2 moving down and right:

* Person 1 down and person 2 down: dp[r1+1][c1][c2];
* Person 1 right and person 2 down: dp[r1][c1+1][c2];
* Person 1 down and person 2 right: dp[r1+1][c1][c2+1];
* Person 1 right and person 2 right: dp[r1][c1+1][c2+1];

**Complexity Analysis**

* Time Complexity: O(N^3)*O*(*N*3), where N*N* is the length of grid. Our dynamic programming has O(N^3)*O*(*N*3)states.
* Space Complexity: O(N^3)*O*(*N*3), the size of memo.