<https://leetcode.com/problems/cherry-pickup/>

You are given an n x n grid representing a field of cherries, each cell is one of three possible integers.

0 means the cell is empty, so you can pass through,

1 means the cell contains a cherry that you can pick up and pass through, or

-1 means the cell contains a thorn that blocks your way.

Return *the maximum number of cherries you can collect by following the rules below*:

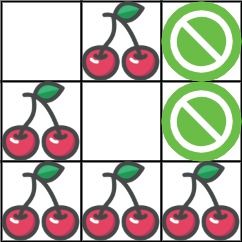
Starting at the position (0, 0) and reaching (n - 1, n - 1) by moving right or down through valid path cells (cells with value 0 or 1).

After reaching (n - 1, n - 1), returning to (0, 0) by moving left or up through valid path cells.

When passing through a path cell containing a cherry, you pick it up, and the cell becomes an empty cell 0.

If there is no valid path between (0, 0) and (n - 1, n - 1), then no cherries can be collected.

**Example 1:**



**Input:** grid = [[0,1,-1],[1,0,-1],[1,1,1]]

**Output:** 5

**Explanation:** The player started at (0, 0) and went down, down, right right to reach (2, 2).

4 cherries were picked up during this single trip, and the matrix becomes [[0,1,-1],[0,0,-1],[0,0,0]].

Then, the player went left, up, up, left to return home, picking up one more cherry.

The total number of cherries picked up is 5, and this is the maximum possible.

**Example 2:**

**Input:** grid = [[1,1,-1],[1,-1,1],[-1,1,1]]

**Output:** 0

**Constraints:**

n == grid.length

n == grid[i].length

1 <= n <= 50

grid[i][j] is -1, 0, or 1.

grid[0][0] != -1

grid[n - 1][n - 1] != -1

**Attempt 1: 2023-08-25**

**Solution 1: Native DFS (360 min, TLE 16/59)**

class Solution {

    public int cherryPickup(int[][] grid) {

        int n = grid.length;

        // Why need to compare with 0 ?

        // Input: [[1,1,-1],[1,-1,1],[-1,1,1]]

        // Output: -2147483645

        // Expected: 0 -> expect no way to reach bottom right corner

        return Math.max(helper(grid, n, 0, 0, 0, 0), 0);

    }

    private int helper(int[][] grid, int n, int r1, int c1, int r2, int c2) {

        // Since we're only going down and to the right, no need to check for < 0

        // if we went out of the grid or hit a thorn, discourage this path by returning Integer.MIN\_VALUE

        if(r1 >= n || c1 >= n || r2 >= n || c2 >= n || grid[r1][c1] == -1 || grid[r2][c2] == -1) {

            return Integer.MIN\_VALUE;

        }

        // If person 1 reached the bottom right, return what's in there (could be 1 or 0)

        if(r1 == n - 1 && c1 == n - 1) {

            return grid[r1][c1];

        }

        // If person 2 reached the bottom right, return what's in there (could be 1 or 0)

        if(r2 == n - 1 && c2 == n - 1) {

            return grid[r2][c2];

        }

        // If both persons standing on the same cell, don't double count and return what's in this cell (could be 1 or 0)

        int cherries = 0;

        if(r1 == r2 && c1 == c2) {

            cherries = grid[r1][c1];

        // otherwise, number of cherries collected by both of them equals the sum of what's on their cells

        } else {

            cherries = grid[r1][c1] + grid[r2][c2];

        }

        // since each person of the 2 person can move only to the bottom or to the right, then the total number of cherries

        // equals the max of the following possibilities:

        //    P1    |      P2

        //  DOWN    |    DOWN

        //  DOWN    |    RIGHT

        //  RIGHT  |    DOWN

        //  RIGHT  |    RIGHT

        cherries += Math.max(Math.max(helper(grid, n, r1 + 1, c1, r2 + 1, c2), helper(grid, n, r1 + 1, c1, r2, c2 + 1)),

Math.max(helper(grid, n, r1, c1 + 1, r2 + 1, c2), helper(grid, n, r1, c1 + 1, r2, c2 + 1)));

        return cherries;

    }

}

Time Complexity: O(N^4)

Space Complexity: O(N^4)

**Solution 2: DFS + Memoization (10 min)**

class Solution {

    public int cherryPickup(int[][] grid) {

        int n = grid.length;

        // Why need to compare with 0 ?

        // Input: [[1,1,-1],[1,-1,1],[-1,1,1]]

        // Output: -2147483645

        // Expected: 0 -> expect no way to reach bottom right corner

        Integer[][][][] memo = new Integer[n][n][n][n];

        return Math.max(helper(grid, n, 0, 0, 0, 0, memo), 0);

    }

    private int helper(int[][] grid, int n, int r1, int c1, int r2, int c2, Integer[][][][] memo) {

        // Since we're only going down and to the right, no need to check for < 0

        // if we went out of the grid or hit a thorn, discourage this path by returning Integer.MIN\_VALUE

        if(r1 >= n || c1 >= n || r2 >= n || c2 >= n || grid[r1][c1] == -1 || grid[r2][c2] == -1) {

            return Integer.MIN\_VALUE;

        }

        // If person 1 reached the bottom right, return what's in there (could be 1 or 0)

        if(r1 == n - 1 && c1 == n - 1) {

            return grid[r1][c1];

        }

        // If person 2 reached the bottom right, return what's in there (could be 1 or 0)

        if(r2 == n - 1 && c2 == n - 1) {

            return grid[r2][c2];

        }

        if(memo[r1][c1][r2][c2] != null) {

            return memo[r1][c1][r2][c2];

        }

        // If both persons standing on the same cell, don't double count and return what's in this cell (could be 1 or 0)

        int cherries = 0;

        if(r1 == r2 && c1 == c2) {

            cherries = grid[r1][c1];

        // otherwise, number of cherries collected by both of them equals the sum of what's on their cells

        } else {

            cherries = grid[r1][c1] + grid[r2][c2];

        }

        // since each person of the 2 person can move only to the bottom or to the right, then the total number of cherries

        // equals the max of the following possibilities:

        //    P1    |      P2

        //  DOWN    |    DOWN

        //  DOWN    |    RIGHT

        //  RIGHT  |    DOWN

        //  RIGHT  |    RIGHT

        cherries += Math.max(Math.max(helper(grid, n, r1 + 1, c1, r2 + 1, c2, memo), helper(grid, n, r1 + 1, c1, r2, c2 + 1, memo)),

Math.max(helper(grid, n, r1, c1 + 1, r2 + 1, c2, memo), helper(grid, n, r1, c1 + 1, r2, c2 + 1, memo)));

        return memo[r1][c1][r2][c2] = cherries;

    }

}

Time Complexity: O(N^4)

Space Complexity: O(N^4)

**Refer to**

<https://leetcode.com/problems/cherry-pickup/solutions/329945/very-easy-to-follow-step-by-step-recursive-backtracking-with-memoization-n-4/>

/\*

It is easy to see:

Instead of having two paths starting from 0,0 and then other path from N,N.

We can have two people starting from 0,0 and find two paths that collects maximum cherries.

First  person finds the path to collect maximum cherries and mark those cherries collected then

Second person finds another path to collect maximum cherries.

Though here is the case where local maximum is not global maximum.

So having cherry pick up by person1 and then person2 won't give the correct result.

This approach fails to find the best answer to this case.

Reference : https://leetcode.com/problems/cherry-pickup/solution/

Approach #1: Greedy [Wrong Answer]

11100

00101

10100

00100

00111

In above example we should be able to pick all cherries. I leave it up to you to figure out two paths that collect all cherries.

But, with our approach person1 will collect 9 cherries leaving once that is on the right(1,4) and on the left(2,0).

Then person2 won't be able to collect both cherries he can collect only right one or only left one.

Approach #2:

Now, we know that we want collectively maximum cherries.

So, we have to do the traversal of both paths at the same time and select maximum global answer.

The potential problem of this approach is double counting (if we collect same cherry in 2 paths), but this can be easily avoided in code.

If both are at the same cell we count cherry only once.

Following code is backtracking brute force so it is TLE.

Time Complexity is : 4^N\*N. As we are calling cherryPickup 4 times recursively with problem size N\*N.

\*/

class Solution1 {

  public int cherryPickup(int[][] grid) {

    return Math.max(0, cherryPickup(grid, grid.length, 0, 0, 0, 0));

  }

  private int cherryPickup(int[][] grid, int n, int r1, int c1, int r2, int c2) {

    // since we're only going down and to the right, no need to check for < 0

    // if we went out of the grid or hit a thorn, discourage this path by returning Integer.MIN\_VALUE

    if(r1 >= n || c1 >= n || r2 >= n || c2 >= n || grid[r1][c1] == -1 || grid[r2][c2] == -1)

      return Integer.MIN\_VALUE;

    // if person 1 reached the bottom right, return what's in there (could be 1 or 0)

    if(r1 == n - 1 && c1 == n - 1)

      return grid[r1][c1];

    // if person 2 reached the bottom right, return what's in there (could be 1 or 0)

    if(r2 == n - 1 && c2 == n - 1)

      return grid[r2][c2];

    int cherries;

    // if both persons standing on the same cell, don't double count and return what's in this cell (could be 1 or 0)

    if(r1 == r2 && c1 == c2)

      cherries = grid[r1][c1];

    else

      // otherwise, number of cherries collected by both of them equals the sum of what's on their cells

      cherries = grid[r1][c1] + grid[r2][c2];

    // since each person of the 2 person can move only to the bottom or to the right, then the total number of cherries

    // equals the max of the following possibilities:

    //    P1    |      P2

    //  DOWN    |    DOWN

    //  DOWN    |    RIGHT

    //  RIGHT  |    DOWN

    //  RIGHT  |    RIGHT

    cherries += Math.max(

        Math.max(cherryPickup(grid, n, r1 + 1, c1, r2 + 1, c2), cherryPickup(grid, n, r1 + 1, c1, r2, c2 + 1)),

        Math.max(cherryPickup(grid, n, r1, c1 + 1, r2 + 1, c2), cherryPickup(grid, n, r1, c1 + 1, r2, c2 + 1)));

    return cherries;

  }

}

/\*

Let that above solution sink in. Now think about memoization.

To make this solution use memoization, we have to think what states we have to preserve.

Here we want to track r1,c1 and r2,c2 positions.

So, if we create Integer[][][][] dp = new Integer[N][N][N][N];

and track all these states then it will reduce the time complexity to N^4.

dp[r1][c1][r2][c2] will identify each state.

if dp[r1][c1][r2][c2] is null then that means we haven't computed subanswer for that state.

if dp[r1][c1][r2][c2] is NOT null then we just return calculated subanswer.

Personally I think if you come up with N^4 solution in the interview then it is awesome.

Runtime: 100 ms, faster than 6.59% of Java online submissions for Cherry Pickup.

Memory Usage: 135.1 MB, less than 5.09% of Java online submissions for Cherry Pickup.

\*/

class Solution {

  public int cherryPickup(int[][] grid) {

    int N = grid.length;

    return Math.max(0, cherryPickup(grid, grid.length, 0, 0, 0, 0, new Integer[N][N][N][N]) );

  }

  private int cherryPickup(int[][] grid, int n, int r1, int c1, int r2, int c2, Integer dp[][][][]) {

    /\* First two lines are same as above \*/

    if(dp[r1][c1][r2][c2]!=null)

        return dp[r1][c1][r2][c2];

  /\*

  This part is same as above

  \*/

    dp[r1][c1][r2][c2] = new Integer(cherries);

    return dp[r1][c1][r2][c2];

  }

}

**Solution 3: DFS + Memoization + Optimize N^4 Space Complexity to N^3 (10 min)**

class Solution {

    public int cherryPickup(int[][] grid) {

        int n = grid.length;

        // Why need to compare with 0 ?

        // Input: [[1,1,-1],[1,-1,1],[-1,1,1]]

        // Output: -2147483645

        // Expected: 0 -> expect no way to reach bottom right corner

        Integer[][][] memo = new Integer[n][n][n];

        return Math.max(helper(grid, n, 0, 0, 0, memo), 0);

    }

    private int helper(int[][] grid, int n, int r1, int c1, int c2, Integer[][][] memo) {

        int r2 = r1 + c1 - c2;

        // Since we're only going down and to the right, no need to check for < 0

        // if we went out of the grid or hit a thorn, discourage this path by returning Integer.MIN\_VALUE

        if(r1 >= n || c1 >= n || r2 >= n || c2 >= n || grid[r1][c1] == -1 || grid[r2][c2] == -1) {

            return Integer.MIN\_VALUE;

        }

        // If person 1 reached the bottom right, return what's in there (could be 1 or 0)

        if(r1 == n - 1 && c1 == n - 1) {

            return grid[r1][c1];

        }

        // If person 2 reached the bottom right, return what's in there (could be 1 or 0)

        if(r2 == n - 1 && c2 == n - 1) {

            return grid[r2][c2];

        }

        if(memo[r1][c1][c2] != null) {

            return memo[r1][c1][c2];

        }

        // If both persons standing on the same cell, don't double count and return what's in this cell (could be 1 or 0)

        int cherries = 0;

        if(r1 == r2 && c1 == c2) {

            cherries = grid[r1][c1];

        // otherwise, number of cherries collected by both of them equals the sum of what's on their cells

        } else {

            cherries = grid[r1][c1] + grid[r2][c2];

        }

        // since each person of the 2 person can move only to the bottom or to the right, then the total number of cherries

        // equals the max of the following possibilities:

        //    P1    |      P2

        //  DOWN    |    DOWN

        //  DOWN    |    RIGHT

        //  RIGHT  |    DOWN

        //  RIGHT  |    RIGHT

        cherries += Math.max(Math.max(helper(grid, n, r1 + 1, c1, c2, memo), helper(grid, n, r1 + 1, c1, c2 + 1, memo)), Math.max(helper(grid, n, r1, c1 + 1, c2, memo), helper(grid, n, r1, c1 + 1, c2 + 1, memo)));

        return memo[r1][c1][c2] = cherries;

    }

}

Time Complexity: O(N^3)

Space Complexity: O(N^3)

**Refer to**

<https://leetcode.com/problems/cherry-pickup/solutions/329945/very-easy-to-follow-step-by-step-recursive-backtracking-with-memoization-n-4/comments/406520>

**One small improvement: since you can only go down and right, we have r1+c1 = r2+c2.**

**Then for the 4-dimensional dp, you can simply drop the last dimension to become dp[r1][c1][r2]. Becomes O(n^3).**

class Solution {

    public int cherryPickup(int[][] grid) {

        int[][][] dp = new int[grid.length][grid.length][grid.length];

        // return 0 if there is no path from 0,0 to n-1,n-1

        return Math.max(0, helper(grid, dp, 0, 0, 0));

    }

    // Instead of going once from 0,0 to n-1,n-1 and then back, we simply go twice

    // from 0,0 to n-1,n-1 because every path from n-1,n-1 to 0,0 can be interpreted

    // as a path from 0,0 to n-1,n-1

    // Note that the one person can never cross the past path of the other person

    // (they can only meet at the same position) so we don't need to worry about one

    // person picking up an already picked up cherry from the past

    // What does a state represent? dp[r1][c1][c2] represents the max number of

    // cherries that can be collected by 2 people going from r1,c1 and r2,c2 to n-1,n-1

    // Transitions between states? we collect cherries on current positions of the

    // two people (r1,c1 and r2,c2), then we go through all possible next states and

    // choose the best one (max number of cherries) as the next state (we do this by

    // adding the number of cherries of the best next state to the number of cherries

    // we picked up on the current two positions of the people). In the end, the state

    // dp[0][0][0] will contain the max number of cherries that can be picked up by

    // going from 0,0 to n-1,n-1 and back.

    private int helper(int[][] grid, int[][][] dp, int r1, int c1, int c2) {

        // we can deduce r2 because r1 + c1 == r2 + c2, since with each move either

        // r or c of a person gets incremented by exactly one (Manhattan distance to

        // origin stays equal) this way we reduce the 4D dp problem to a 3D one

        // (we save space by reducing the number of things we store in a state)

        int r2 = r1 + c1 - c2;

        // check if current state is out of bounds or on thorns

        if (r1 >= grid.length || c1 >= grid.length || r2 >= grid.length

            || c2 >= grid.length || grid[r1][c1] == -1 || grid[r2][c2] == -1) {

            // current state should not be included in the solution

            return Integer.MIN\_VALUE;

        }

        // check if we have already computed a solution for this state

        if (dp[r1][c1][c2] != 0) return dp[r1][c1][c2];

        // check if we reached the end state (note that if r1,c1 reached the end,

        // this implies that r2,c2 also reached the end)

        if (r1 == grid.length - 1 && c1 == grid.length - 1) {

            return grid[r1][c1];

        }

        // compute and return answer for current state

        int result = grid[r1][c1];

        // in case the second person is on the same position, don't pick up the

        // same cherry twice. Note that r1 == r2 <--> c1 == c2 (eg. they can't

        // be on the same row without also being on the same column)

        if (r1 != r2) {

            result += grid[r2][c2];

        }

        // pick best possible next state

        int bestNextState = Math.max(helper(grid, dp, r1 + 1, c1, c2),  // down, down

                                    helper(grid, dp, r1, c1 + 1, c2 + 1));  // right, right

        bestNextState = Math.max(bestNextState, helper(grid, dp, r1 + 1, c1, c2 + 1));   // down, right

        bestNextState = Math.max(bestNextState, helper(grid, dp, r1, c1 + 1, c2));   // right, down

        result += bestNextState;

        dp[r1][c1][c2] = result;    // store current state such that it can be re-used

        return result;

    }

}

**Refer to Deepseek**

**Optimizing Memoization Space from O(N⁴) to O(N³)**

Yes! We can optimize the space complexity by observing that both paths always move the same number of steps (since they both start at (0,0) and must reach (N-1,N-1)). This means we can use step = r1 + c1 = r2 + c2 as a key dimension, reducing our state from (r1,c1,r2,c2) to (step,r1,r2).

class Solution {

private int[][][] memo;

private int[][] grid;

private int N;

public int cherryPickup(int[][] grid) {

this.N = grid.length;

this.grid = grid;

this.memo = new int[2\*N-1][N][N];

// Initialize memo table with -1

for (int i = 0; i < 2\*N-1; i++) {

for (int j = 0; j < N; j++) {

Arrays.fill(memo[i][j], -1);

}

}

int result = dfs(0, 0, 0);

return Math.max(0, result);

}

private int dfs(int step, int r1, int r2) {

int c1 = step - r1;

int c2 = step - r2;

// Check if out of bounds or on thorn

if (r1 >= N || c1 >= N || r2 >= N || c2 >= N ||

grid[r1][c1] == -1 || grid[r2][c2] == -1) {

return Integer.MIN\_VALUE;

}

// Check if we've already computed this state

if (memo[step][r1][r2] != -1) {

return memo[step][r1][r2];

}

// Add cherries from current positions

int result = grid[r1][c1];

if (r1 != r2 || c1 != c2) {

result += grid[r2][c2];

}

// Base case: both paths reached destination

if (step == 2\*N-2) {

return result;

}

// Try all combinations of moves

int max = Integer.MIN\_VALUE;

max = Math.max(max, dfs(step+1, r1, r2)); // both right

max = Math.max(max, dfs(step+1, r1, r2+1)); // P1 right, P2 down

max = Math.max(max, dfs(step+1, r1+1, r2)); // P1 down, P2 right

max = Math.max(max, dfs(step+1, r1+1, r2+1)); // both down

// Store result in memo before returning

if (max != Integer.MIN\_VALUE) {

result += max;

memo[step][r1][r2] = result;

return result;

} else {

memo[step][r1][r2] = Integer.MIN\_VALUE;

return Integer.MIN\_VALUE;

}

}

}

**Key Optimizations:**

**State Reduction**:

Instead of tracking (r1,c1,r2,c2), we track (step,r1,r2)

Calculate c1 = step - r1 and c2 = step - r2

Total steps needed: 2N-2 (from (0,0) to (N-1,N-1))

**Space Improvement**:

Original: O(N⁴) → N×N×N×N

Optimized: O(N³) → (2N-1)×N×N

**Same Correctness**:

Maintains all possible path combinations

Still properly handles cherry counting when paths cross

**Complexity Analysis:**

**Time Complexity**: O(N³) - We have N steps and N×N possible (r1,r2) combinations at each step

**Space Complexity**: O(N³) - For the memoization table

This optimized solution maintains the same correctness as the original while significantly reducing memory usage, making it feasible for larger grid sizes.

**Please elaborate below statement ? how we get this relation ?**

**This means we can use step = r1 + c1 = r2 + c2 as a key dimension, reducing our state from (r1,c1,r2,c2) to (step,r1,r2).**

The key insight that allows us to reduce the state space from O(N⁴) to O(N³) comes from observing the relationship between the positions of the two paths. Let me break this down in detail:

**Understanding the Step Relationship**

**Single Path Movement**:

For any path moving from (0,0) to (N-1,N-1) using only right/down moves:

Each move increases either the row or column by 1

After k moves: row + column = k

Total steps to reach end: 2N-2 (N-1 right + N-1 down)

**Two Paths Moving Simultaneously**:

Both paths start at (0,0) at step 0

Both must reach (N-1,N-1) at step 2N-2

At any intermediate step k:

Path 1 satisfies: r1 + c1 = k

Path 2 satisfies: r2 + c2 = k

**State Space Reduction**

Original state: (r1, c1, r2, c2)

Reduced state: (step, r1, r2)

Why this works:

**Step Determines Position Sum**:

Given step = k, we know:

c1 = k - r1

c2 = k - r2

This means we can reconstruct c1 and c2 when needed

**Dimensionality Reduction**:

Instead of tracking 4 variables (r1,c1,r2,c2)

We track 3 variables (step,r1,r2) and compute the others

Eliminates one entire dimension from our state space

**Mathematical Justification**

For any valid positions:

r1 + c1 = r2 + c2 = step

Therefore:

c1 = step - r1

c2 = step - r2

This relationship holds because:

Each move (right or down) increments the step count

The sum r+c always equals the number of steps taken

**Practical Implications**

**Memory Savings**:

Original: N × N × N × N = N⁴

Optimized: (2N-1) × N × N ≈ 2N³ (but we consider as O(N³))

**Same Information Content**:

No loss of information - we can always compute c1 and c2

All possible path combinations are still represented

**Example Walkthrough**

Consider N=3 (3x3 grid):

At step=2 (third move):

Possible positions where r+c=2:

(0,2), (1,1), (2,0)

If we store state as (step=2, r1=1, r2=0):

We know:

c1 = 2 - 1 = 1 → Path1 at (1,1)

c2 = 2 - 0 = 2 → Path2 at (0,2)

This maintains all necessary information while using less memory.

**Why This Optimization Works**

**Physical Constraint**:

Both paths move in lockstep (same number of moves)

Their position sums must always be equal

**Mathematical Dependency**:

The column positions are dependent on row positions and step

No need to store them explicitly

This optimization is crucial for making the solution feasible for larger grids while maintaining correctness.

**Step by step guidance of the O(N^3) time and O(N^2) space solution**

**Refer to**

<https://leetcode.com/problems/cherry-pickup/solutions/109903/step-by-step-guidance-of-the-o-n-3-time-and-o-n-2-space-solution/>

I -- A naive idea towards the solution

To begin with, you may be surprised by the basic ideas to approach the problem: simply simulate each of the round trips and choose the one that yields the maximum number of cherries.

But then what's the difficulty of this problem? The biggest issue is that there are simply too many round trips to explore -- the number of round trips scales exponentially as the size Nof the grid. This is because each round trip takes (4N-4)steps, and at each step, we have two options as to where to go next (in the worst case). This puts the total number of possible round trips at 2^(4N-4). Therefore a naive implementation of the aforementioned idea would be very inefficient.

II -- Initial attempt of DP

Fortunately, a quick look at the problem seems to reveal the two features of dynamic programming: optimal substructure and overlapping of subproblems.

Optimal substructure: if we define T(i, j)as the maximum number of cherries we can pick up starting from the position (i, j)(assume it's not a thorn) of the gridand following the path (i, j) ==> (N-1, N-1) ==>(0, 0), we could move one step forward to either (i+1, j)or (i, j+1), and recursively solve for the subproblems starting from each of those two positions (that is, T(i+1, j)and T(i, j+1)), then take the sum of the larger one (assume it exists) together with grid[i][j]to form a solution to the original problem. (**Note**: the previous analyses assume we are on the first leg of the round trip, that is, (0, 0) ==> (N-1, N-1); if we are on the second leg, that is, (N-1, N-1) ==> (0, 0), then we should move one step backward from (i, j)to either (i-1, j)or (i, j-1).)

Overlapping of subproblems: two round trips may overlap with each other in the middle, leading to repeated subproblems. For example, the position (i, j)can be reached from both positions (i-1, j)and (i, j-1), which means both T(i-1, j)and T(i, j-1)are related to T(i, j). Therefore we may cache the intermediate results to avoid recomputing these subproblems.

This sounds promising, since there are at most O(N^2)starting positions, meaning we could solve the problem in O(N^2)time with caching. But there is an issue with this naive DP -- it failed to take into account the constraint that "**once a cherry is picked up, the original cell (value 1) becomes an empty cell (value 0)**", so that if there are overlapping cells between the two legs of the round trip, those cells will be counted twice. In fact, without this constraint, we can simply solve for the maximum number of cherries of the two legs of the round trip separately (they should have the same value), then take the sum of the two to produce the answer.

III -- Second attempt of DP that modifies the grid matrix

So how do we account for the aforementioned constraint? I would say, why don't we reset the value of the cell from 1to 0after we pick up the cherry? That is, modify the gridmatrix as we go along the round trip.

**1. Can we still divide the round trip into two legs and maximize each of them separately ?**

Well, you may be tempted to do so as it seems to be right at first sight. However, if you dig deeper, you will notice that the maximum number of cherries of the second leg actually depends on the choice of path for the first leg. This is because if we pluck some cherry in the first leg, it will no longer be available for the second leg (remember we reset the cell value from 1to 0). So the above greedy idea only maximize the number of cherries for the first leg, but not necessarily for the sum of the two legs (that is, local optimum does not necessarily lead to global optimum).

Here is a counter example:

grid = ​[[1,1,1,0,1],

​ ​[0,0,0,0,0],

​ ​[0,0,0,0,0],

​ ​[0,0,0,0,0],

​ ​[1,0,1,1,1]].

The greedy idea above would suggest a Z-shaped path for the first leg, i.e., (0, 0) ==> (0, 2) ==> (4, 2) ==> (4, 4), which garners 6cherries. Then for the second leg, the maximum number of cherries we can get is 1(the one at the lower-left or upper-right corner), so the sum will be 7. This is apparently less than the best route by traveling along the four edges, in which all 8cherries can be picked.

**2. What changes do we need to make on top of the above naive DP if we are modifying the grid matrix ?**

The obvious difference is that now the maximum number of cherries of the trip not only depends on the starting position (i, j), but also on the **status**of the gridmatrix when that position is reached. This is because the gridmatrix may be modified differently along different paths towards the same position (i, j), therefore, even if the starting position is the same, the maximum number of cherries may be different since we are working with different gridmatrix now.

Here is a simple example to illustrate this. Assume we have this gridmatrix:

grid = ​[[0,1,0],

​ ​[0,1,0],

​ ​[0,0,0].

and we are currently at position (1, 1). If this position is reached following the path (0, 0) ==> (0, 1) ==> (1, 1), the gridmatrix will be:

grid = ​[[0,0,0],

​ ​[0,1,0],

​ ​[0,0,0].

However, if it is reached following the path (0, 0) ==> (1, 0) ==> (1, 1), the gridmatrix will be the same as the initial one:

grid = ​[[0,1,0],

​ ​[0,1,0],

​ ​[0,0,0].

Therefore starting from the same initial position (1, 1), the maximum number of cherries will be 1for the former and 2for the latter.

So now each of our subproblems can be denoted symbolically as T(i, j, grid.status), where the status of the gridmatrix may be represented by a string with cell values joined row by row. Our original problem will be T(0, 0, grid.initial\_status)and the recurrence relations are something like:

T(i, j, grid.status) = -1, if grid[i][j] == -1 || T(i + d, j, grid.status1) == -1 && T(i + d, j, grid.status2) == -1;

T(i, j, grid.status) = grid[i][j] + max(T(i + d, j, grid.status1), T(i, j + d, grid.status2)), otherwise.

Here ddepends on which leg we are during the round trip (d = +1for the first leg and d = -1for the second leg), both grid.status1and grid.status2can be obtained from grid.status.

To cache the intermediate results, we may create an N-by-Nmatrix of HashMaps, where the one at position (i, j)will map each grid.statusto the maximum number of cherries obtained starting from position (i, j)on the grid with that particular status.

**3. What is the issue with this new version of DP ?**

While we can certainly develop a solution using this new version of DP, it does **NOT**help reduce the time complexity substantially. While it does help improve the performance, the worst case time complexity is still exponential. The reason is that the number of grid.statusis very large -- in fact, it is exponential too, as each path may lead to a unique grid.statusand the number of paths to some position is exponential. So the possibility of overlapping subproblems becomes so slim that we are forced to compute most of the subproblems, leading to the exponential time complexity.

IV -- Final attempt of DP without modifying the grid matrix

So we have seen that modifying the gridmatrix isn't really the way to go. But if we leave it intact, how do we account for the aforementioned constraint? The key here is to avoid the duplicate counting. But for now, let's pretend the constraint does not exist and see what we can do later to overcome it when it shows up.

**1. Can we reuse the definition of DP problem in Part II ?**

Not really. The recurrence relation of the DP problem in **Part II** says that

T(i, j) = grid[i][j] + max{T(i+1, j), T(i, j+1)}, which means we already counted grid[i][j]towards T(i, j). To avoid the duplicate counting, we somehow need to make sure that grid[i][j]will not be counted towards any of T(i+1, j)and T(i, j+1). This can only happen if the position (i, j)won't appear on the paths for either of the two trips: (i+1, j) ==> (N-1, N-1) ==>(0, 0)or (i, j+1) ==> (N-1, N-1) ==>(0, 0), which is something we cannot guarantee. For example, since we have no control over the path that will be chosen for the sub-trip (N-1, N-1) ==>(0, 0)of both trips, it may pass the position (i, j)again, resulting in duplicate counting.

**2. Can we shorten our round trip so that we don't have to go all the way to the lower right corner ?**

Maybe. We can redefine T(i, j)as the maximum number of cherries for the shortened round trip: (0, 0) ==> (i, j) ==> (0, 0)without modifying the gridmatrix. The original problem then will be denoted as T(N-1, N-1). To obtain the recurrence relations, note that for each position (i, j), we have two options for arriving at and two options for leaving it: (i-1, j)and (i, j-1), so the above round trip can be divide into four cases:

Case 1: (0, 0) ==> (i-1, j) ==> (i, j) ==> (i-1, j) ==> (0, 0)

Case 2: (0, 0) ==> (i, j-1) ==> (i, j) ==> (i, j-1) ==> (0, 0)

Case 3: (0, 0) ==> (i-1, j) ==> (i, j) ==> (i, j-1) ==> (0, 0)

Case 4: (0, 0) ==> (i, j-1) ==> (i, j) ==> (i-1, j) ==> (0, 0)

By definition, Case 1 is equivalent to T(i-1, j) + grid[i][j]and Case 2 is equivalent to T(i, j-1) + grid[i][j]. However, our definition of T(i, j)does not cover the last two cases, where the end of the first leg of the trip and the start of the second leg of the trip are different. This suggests we should generalize our definition from T(i, j)to T(i, j, p, q), which denotes the maximum number of cherries for the two-leg trip (0, 0) ==> (i, j); (p, q) ==> (0, 0)without modifying the gridmatrix.

**3. Will this two-leg DP definition work ?**

We don't really know. But at least, we can work out the recurrence relations for T(i, j, p, q). Similar to the analyses above, there are two options for arriving at (i, j), and two options for leaving (p, q), so the two-leg trip again can be divided into four cases:

Case 1: (0, 0) ==> (i-1, j) ==> (i, j); (p, q) ==> (p-1, q) ==> (0, 0)

Case 2: (0, 0) ==> (i-1, j) ==> (i, j); (p, q) ==> (p, q-1) ==> (0, 0)

Case 3: (0, 0) ==> (i, j-1) ==> (i, j); (p, q) ==> (p-1, q) ==> (0, 0)

Case 4: (0, 0) ==> (i, j-1) ==> (i, j); (p, q) ==> (p, q-1) ==> (0, 0)

and by definition, we have:

Case 1 is equivalent to T(i-1, j, p-1, q) + grid[i][j] + grid[p][q];

Case 2 is equivalent to T(i-1, j, p, q-1) + grid[i][j] + grid[p][q];

Case 3 is equivalent to T(i, j-1, p-1, q) + grid[i][j] + grid[p][q];

Case 4 is equivalent to T(i, j-1, p, q-1) + grid[i][j] + grid[p][q];

Therefore, the recurrence relations can be written as:

T(i, j, p, q) = grid[i][j] + grid[p][q] + max{T(i-1, j, p-1, q), T(i-1, j, p, q-1), T(i, j-1, p-1, q), T(i, j-1, p, q-1)}

Now to make it work, we need to impose the aforementioned constraint. As mentioned above, since we already counted grid[i][j]and grid[p][q]towards T(i, j, p, q), to avoid duplicate counting, both of them should **NOT** be counted for any of T(i-1, j, p-1, q), T(i-1, j, p, q-1), T(i, j-1, p-1, q)and T(i, j-1, p, q-1). It is obvious that the position (i, j)won't appear on the paths of the trips (0, 0) ==> (i-1, j)or (0, 0) ==> (i, j-1), and similarly the position (p, q)won't appear on the paths of the trips (p-1, q) ==> (0, 0)or (p, q-1) ==> (0, 0). Therefore, if we can guarantee that (i, j)won't appear on the paths of the trips (p-1, q) ==> (0, 0)or (p, q-1) ==> (0, 0), and (p, q)won't appear on the paths of the trips (0, 0) ==> (i-1, j)or (0, 0) ==> (i, j-1), then no duplicate counting can ever happen. So how do we achieve that?

Take the trips (0, 0) ==> (i-1, j)and (0, 0) ==> (i, j-1)as an example. Although we have no control over the paths that will be taken for them, we do know the boundaries of the paths: all positions on the path for the former will be lying within the rectangle [0, 0, i-1, j]and for the latter will be lying within the rectangle [0, 0, i, j-1], which implies all positions on the two paths combined will be lying within the rectangle [0, 0, i, j], except for the lower right corner position (i, j). Therefore, if we make sure that the position (p, q)is lying outside the rectangle [0, 0, i, j](except for the special case when it overlaps with (i, j)), it will never appear on the paths of the trips (0, 0) ==> (i-1, j)or (0, 0) ==> (i, j-1).

The above analyses are equally applicable to the trips (p-1, q) ==> (0, 0)and (p, q-1) ==> (0, 0), so we conclude that the position (i, j)has to be lying outside the rectangle [0, 0, p, q](again except for the special case) in order to avoid duplicate counting. So in summary, one of the following three conditions should be true:

i < p && j > q

i == p && j == q

i > p && j < q

This indicates that our definition of the two-leg trip T(i, j, p, q)is not valid for all values of the four indices, but instead, they will be subjected to the above three conditions. This is problematic, as it would break the self-consistency of the original definition of T(i, j, p, q)when such conditions do not exist. A direct consequence is that the above recurrence relations derived for T(i, j, p, q)won't work anymore. For example, T(3, 1, 2, 3)is valid under these conditions but one of the terms in the recurrence relations, T(2, 1, 2, 2), would be invalid, and we have no idea how to get its value under current definition of T(i, j, p, q).

**4. Self-consistent two-leg DP definition**

Though the above two-leg DP definition does not work, we are pretty close to a real solution. We know that in order to avoid duplicate counting, the four indices, (i, j, p, q), have to be correlated to each other (i.e., they are not independent variables). The above three conditions are only the most general way for specifying what the correlations should be, but not necessarily the best one. We can as well choose a subset of those three conditions and define T(i, j, p, q)over that subset, then still no duplicate counting will ever happen for this new definition. This is because if the four indices fall within the range delimited by the subset, they will be guaranteed to satisfy the above three conditions, which is the most general form of conditions we have derived to eliminate the possibilities of duplicate counting (this is like to say, we want to have a < 10, and if we always choose asuch that a < 5, then it is guaranteed that a < 10).

So our goal now is to select a subset of the conditions that can restore the self-consistency of T(i, j, p, q)so we can have a working recurrence relation. The key observation comes from the fact that when i(p) increases, we need to decrease j(q) in order to make the above conditions hold, and vice versa -- they are **anti-correlated**. This suggests we can set the sum of i(p) and j(q) to some constant, n = i + j = p + q. Then it is straightforward to verify that the above conditions is met automatically, meaning n = i + j = p + qis indeed a subset of the above conditions. (Note in this subset of conditions, ncan be interpreted as the number of steps from the source position (0, 0). I have also tried other anti-correlated functions for iand jsuch as their product is a constant but it did not work out. The recurrence relations here play a role and constant sum turns out to be the simplest one that works.)

With the new conditions in place, we can now redefine our T(i, j, p, q)such that n = i + j = p + q, which can be rewritten, in terms of independent variables, as T(n, i, p), where T(n, i, p) = T(i, n-i, p, n-p). Note that under this definition, we have:

T(i-1, n-i, p-1, n-p) = T(n-1, i-1, p-1)

T(i-1, n-i, p, n-p-1) = T(n-1, i-1, p)

T(i, n-i-1, p-1, n-p) = T(n-1, i, p-1)

T(i, n-i-1, p, n-p-1) = T(n-1, i, p)

Then from the recurrence relation for T(i, j, p, q), we obtain the recurrence relation for T(n, i, p)as:

T(n, i, p) = grid[i][n-i] + grid[p][n-p] + max{T(n-1, i-1, p-1), T(n-1, i-1, p), T(n-1, i, p-1), T(n-1, i, p)}.

Of course, in the recurrence relation above, only one of grid[i][n-i]and grid[p][n-p]will be taken if i == p(i.e., when the two positions overlap). Also note that all four indices, i, j, pand q, are in the range [0, N), meaning nwill be in the range [0, 2N-1)(remember it is the sum of iand j). Lastly we have the base case given by T(0, 0, 0) = grid[0][0].

Now using the recurrence relation for T(n, i, p), it is straightforward to code for the O(N^3)time and O(N^3)space solution. However, if you notice that T(n, i, p)only depends on those subproblems with n - 1, we can iterate on this dimension and cut down the space to O(N^2). So here is the final O(N^3)time and O(N^2)space solution, where we use -1to indicate that a two-leg trip cannot be completed, and iterate in backward direction for indices iand pto get rid of the temporary matrix that is otherwise required for updating the dpmatrix.

**Refer to Deepseek**

**Reducing Cherry Pickup from O(N³) to O(N²) Space**

While the time complexity remains O(N³), we can optimize the space complexity further to O(N²) by observing that we only need information from the previous step to compute the current step.

**Key Insight for Space Optimization**

**Step Dependency**:

To compute step k, we only need results from step k-1

We don't need to store all previous steps

**State Representation**:

Instead of storing a 3D array, we can use two 2D arrays:

One for the current step (curr)

One for the previous step (prev)

**Optimized O(N²) Space Solution**

class Solution {

public int cherryPickup(int[][] grid) {

int N = grid.length;

int[][] prev = new int[N][N];

for (int i = 0; i < N; i++) {

Arrays.fill(prev[i], Integer.MIN\_VALUE);

}

prev[0][0] = grid[0][0]; // Starting point

for (int step = 1; step <= 2\*N-2; step++) {

int[][] curr = new int[N][N];

for (int i = 0; i < N; i++) {

Arrays.fill(curr[i], Integer.MIN\_VALUE);

}

for (int r1 = Math.max(0, step-(N-1)); r1 <= Math.min(N-1, step); r1++) {

int c1 = step - r1;

if (grid[r1][c1] == -1) continue;

for (int r2 = Math.max(0, step-(N-1)); r2 <= Math.min(N-1, step); r2++) {

int c2 = step - r2;

if (grid[r2][c2] == -1) continue;

int max = Integer.MIN\_VALUE;

// Check all 4 possible previous states

for (int pr1 = r1-1; pr1 <= r1; pr1++) {

for (int pr2 = r2-1; pr2 <= r2; pr2++) {

if (pr1 >= 0 && pr2 >= 0) {

max = Math.max(max, prev[pr1][pr2]);

}

}

}

if (max != Integer.MIN\_VALUE) {

curr[r1][r2] = max + grid[r1][c1];

if (r1 != r2) curr[r1][r2] += grid[r2][c2];

}

}

}

prev = curr;

}

return Math.max(0, prev[N-1][N-1]);

}

}

**Why This Works**

**Space Reduction**:

Original: O(N³) (N×N×N)

Optimized: O(N²) (two N×N arrays)

**Correctness Preservation**:

Still considers all possible path combinations

Maintains proper cherry counting logic

Handles thorns and boundaries correctly

**Efficiency Gains**:

Much better cache locality

Reduced memory overhead

Same time complexity (O(N³))

**Implementation Notes**

**Step Processing**:

Processes steps from 1 to 2N-2

For each step, only considers valid (r,c) pairs where r + c = step

**State Transitions**:

For each current position pair (r1,c1) and (r2,c2)

Checks all 4 possible previous position combinations

**Cherry Counting**:

Adds cherries from both positions

Adjusts for shared cells (when r1=r2 and c1=c2)

This optimized solution maintains all the correctness of the original while significantly reducing space usage, making it suitable for very large grids (up to N≈200 on typical judging platforms).

**Refer to**

[L62.Unique Paths (Ref.L63,L64,L174,L2087,L2304,L2400)](note://E33F0B728ABE4CCF8B7E554A219CF1EA)

[L2435.Paths in Matrix Whose Sum Is Divisible by K (Ref.L62)](note://WEB2e1f15d82976dbe1f3c531b0aa3a7504)