Cherry Pickup

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

Your task is to collect maximum number of cherries possible 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.
```

Note:

- `grid` is an `N` by `N` 2D array, with `1 .`
- Each `grid[i][j]` is an integer in the set `{-1, 0, 1}`.
- It is guaranteed that grid[0][0] and grid[N-1][N-1] are not -1.
-

Solution 1

It takes me quite some time to understand the DP solution to this problem this afternoon.

So I post the solution here with annotations for anyone who might find helpful.

Updated Dec 05, 2017. Thanks for @张博文 @xu8908 @szyyn95 for pointing out the overflow case.

```cpp
class Solution {
public:
    int cherryPickup(vector<vector<int>>& grid) {
        int n = grid.size();
        // dp holds maximum # of cherries two k-length paths can pickup.
        // The two k-length paths arrive at (i, k - i) and (j, k - j),
        // respectively.
        vector<vector<int>> dp(n, vector<int>(n, -1));

        dp[0][0] = grid[0][0]; // length k = 0

        // maxK: number of steps from (0, 0) to (n-1, n-1).
        const int maxK = 2 * (n - 1);

        for (int k = 1; k <= maxK; k++) { // for every length k
            vector<vector<int>> curr(n, vector<int>(n, -1));

            // one path of length k arrive at (i, k - i)
            for (int i = 0; i < n && i <= k; i++) {
                if ( k - i >= n) continue;
                // another path of length k arrive at (j, k - j)
                for (int j = 0; j < n && j <= k; j++) {
                    if (k - j >= n) continue;
                    if (grid[i][k - i] < 0 || grid[j][k - j] < 0) { // keep away fr
om thorns
                        continue;
                    }

                    int cherries = dp[i][j]; // # of cherries picked up by the two
(k-1)-length paths.

                    // See the figure below for an intuitive understanding
                    if (i > 0) cherries = std::max(cherries, dp[i - 1][j]);
                    if (j > 0) cherries = std::max(cherries, dp[i][j - 1]);
                    if (i > 0 && j > 0) cherries = std::max(cherries, dp[i - 1][j -
1]);

                    // No viable way to arrive at (i, k - i)-(j, k-j).
                    if (cherries < 0) continue;

                    // Pickup cherries at (i, k - i) and (j, k -j ) if i != j.
                    // Otherwise, pickup (i, k-i).
                    cherries += grid[i][k - i] + (i == j ? 0 : grid[j][k - j]);

                    curr[i][j] = cherries;
                }
            }
            dp = std::move(curr);
        }

        return std::max(dp[n-1][n-1], 0);
    }
};
```

written by storypku original link here

Solution 2

It is the same as two path start from (0, 0) to (n-1, n-1)
dp[x1][x2] means two length p path , currently one arrived at [..., x1] , the other is at [..., x2], the max value we can get

loop on length p, update max.

bottom to up, updated to use one n*n array.

```
class Solution {
public:
    int cherryPickup(vector<vector<int>>& grid) {
        int N = grid.size();
        int P_LEN = N + N - 1;
        vector<vector<int> > dp = vector<vector<int>>(N, vector<int>(N, -1));
        dp[0][0] = grid[0][0];

        for (int p = 2; p <= P_LEN; p++) {//p is the length of the path
            for (int x1 = N - 1; x1 >= 0; x1--) {
                for (int x2 = x1; x2 >= 0; x2--) {
                    int y1 = p - 1 - x1;
                    int y2 = p - 1 - x2;
                    if (y1 < 0 || y2 < 0 || y1 >= N || y2 >= N)
                        continue;
                    if (grid[y1][x1] < 0 || grid[y2][x2] < 0) {
                        dp[x1][x2] = -1;
                        continue;
                    }
                    int best = -1, delta = grid[y1][x1];
                    if (x1 != x2)
                        delta += grid[y2][x2];
                    if (x1 > 0 && x2 > 0 && dp[x1 - 1][x2 - 1] >= 0) //from left le
ft
                        best = max(best, dp[x1 - 1][x2 - 1] + delta);
                    if (x1 > 0 && y2 > 0 && dp[x1 - 1][x2] >= 0) //from left up
                        best = max(best, dp[x1 - 1][x2] + delta);
                    if (y1 > 0 && x2 > 0 && dp[x1][x2 - 1] >= 0) //from up left
                        best = max(best, dp[x1][x2 - 1] + delta);
                    if (y1 > 0 && y2 > 0 && dp[x1][x2] >= 0) //from up up
                        best = max(best, dp[x1][x2] + delta);
                    dp[x1][x2] = best;
                }
            }
        }
        return dp[N - 1][N - 1] < 0 ? 0 : dp[N - 1][N - 1];
    }
};
```

written by jordandong original link here

Solution 3

`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 `N` of 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 `grid` and 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 `1` to `0` after we pick up the cherry? That is, modify the `grid` matrix 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 `1` to `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 `6` cherries. 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 `8` cherries 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 `grid` matrix when that position is reached. This is because the `grid` matrix 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 `grid` matrix now.

Here is a simple example to illustrate this. Assume we have this `grid` matrix:

```
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 `grid` matrix 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

`grid` matrix 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 `1` for the former and `2` for the latter.

So now each of our subproblems can be denoted symbolically as `T(i, j, grid.status)`, where the status of the `grid` matrix 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 `d` depends on which leg we are during the round trip (`d = +1` for the first leg and `d = -1` for the second leg), both `grid.status1` and `grid.status2` can be obtained from `grid.status`.

To cache the intermediate results, we may create an `N-by-N` matrix of HashMaps, where the one at position `(i, j)` will map each `grid.status` to 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.status` is very large -- in fact, it is exponential too, as each path may lead to a unique `grid.status` and 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 `grid` matrix 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 `grid` matrix. 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 `grid` matrix.

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)`. Similarly 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:

1. `i < p && j > q`
2. `i == p && j == q`
3. `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 `a` such 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 + q` is indeed a subset of the above conditions. (Note in this subset of conditions, `n` can be interpreted as the number of steps from the source position `(0, 0)`. I have also tried other anti-correlated functions for `i` and `j` such 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`, `p` and `q`, are in the range `[0, N)`, meaning `n` will be in the range `[0, 2N-1)` (remember it is the sum of `i` and `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 `-1` to indicate that a two-leg trip cannot be completed, and iterate in backward direction for indices `i` and `p` to get rid of the temporary matrix that is otherwise required for updating the `dp` matrix.

```
public int cherryPickup(int[][] grid) {
    int N = grid.length, M = (N << 1) - 1;
    int[][] dp = new int[N][N];
    dp[0][0] = grid[0][0];

    for (int n = 1; n < M; n++) {
 for (int i = N - 1; i >= 0; i--) {
     for (int p = N - 1; p >= 0; p--) {
  int j = n - i, q = n - p;

  if (j < 0 || j >= N || q < 0 || q >= N || grid[i][j] < 0 || grid[p][q] < 0) {
                    dp[i][p] = -1;
                    continue;
                }

   if (i > 0) dp[i][p] = Math.max(dp[i][p], dp[i - 1][p]);
   if (p > 0) dp[i][p] = Math.max(dp[i][p], dp[i][p - 1]);
   if (i > 0 && p > 0) dp[i][p] = Math.max(dp[i][p], dp[i - 1][p - 1]);

   if (dp[i][p] >= 0) dp[i][p] += grid[i][j] + (i != p ? grid[p][q] : 0)
             }
  }
    }

    return Math.max(dp[N - 1][N - 1], 0);
}
```

written by fun4LeetCode original link here