Code Jam 2019 - Round 1C

Analysis: Bacterial Tactics

Test set 1

On a player's turn, the grid will be in some *state*, according to whether any bacteria have already been placed, and how they have spread. We can determine whether a state is *losing* via the following recursive definition:

- the player has no moves because there are no empty cells
- any of the player's moves would either cause a mutation, or give the opponent a *winning* state.
    - A winning state is a state that has at least one *winning move* — that is, a move that would leave the other player in a losing state.

Observe that if a state is not losing, it must be winning, since it has at least one move that does not cause a mutation and does not give the opponent a winning state.

To find the number of winning opening moves (if any) for Becca, we can check each move to see whether it is a winning move. Of course, to do this, we have to investigate the resulting state recursively per the above definition. However, since there are up to two moves per empty cell per state, the naive implementation that recursively counts the number of winning moves for each state may not be fast enough to handle even the 4 × 4 grids in test set 1, so we should optimize it.

Notice that whether a state is winning or losing does not depend on who the player is or on any previous moves. Since the same state may come up multiple times, we should consider memoizing our findings about each state to use in the future. It may be daunting that the number of possible states is intractably large. However, for any given case, there can be at most 16 initially empty cells, each of which can be either filled in by bacteria or not. (After a colony has been placed and has spread, it no longer matters what type it was.) So, we can put an upper bound of 2^{16} on the number of states per case. In practice, there will be even fewer because not all states are reachable.

Moreover, we can save some time by not computing the exact number of winning moves for every state we examine. We only care about this value for an initial state; for every other state, it suffices to determine whether it is winning or losing. If we are investigating a non-initial state's moves and we find a winning move, we can declare the state to be winning, and stop. This optimization alone may be enough to solve test set 1.

Test set 2

When a player makes a legal move, the bacteria spread across the entire width or length of the row or column, up until the line of bacteria reaches reaches the edge of the grid or a cell that is already infected. Therefore, each move creates up to two subproblems that are independent in the sense that a move in one subproblem does not affect the state of the other.

Each subproblem can be expressed as a rectangle contained within the full grid. There are therefore at most $O(\mathbf{R}^2\mathbf{C}^2)$ subproblems. How can we use the results of these subproblems to determine the overall winner of the game?

The goal of the game is to force the opponent into a situation in which there is no move they can make that leads them down a path to victory. The game is *impartial*: both players have access to

the same set of moves. It is therefore apt to draw a comparison between Bacterial Tactics and the ancient game Nim, an impartial game with similar types of decisions. The mathematics of Nim are well-studied. A discovery particularly useful to us is the Sprague–Grundy Theorem, which says that any impartial game can be mapped to a state of Nim. Every state in Nim corresponds to a non-negative *Grundy number*, or *nimber*, where any nonzero nimber indicates a winnable game.

According to nimber addition, the nimber of a game state after we place a colony is equal to the *XOR* of the two subproblems. The nimber of a game state before we place a colony is the *minimum excludant*, or *MEX*, of the set of possible nimbers after placing colonies. We can therefore solve Bacterial Tactics recursively using the following pseudocode:

```
let solve(state) be a function:
  let s = Ã˜
  for each legal colony placement:
    add [solve(first subproblem) XOR solve(second subproblem)] to s
  return MEX(s)
```

Given this general framework, we can now optimize our implementation.

First, as in test set 1, we can memoize the game states, which are now defined using rectangles of various sizes within the original grid. Note that it is not possible to have bacteria from previous moves in a subproblem, because we always cut the rectangle along the row or column of cells infected by a colony placement. We may also want to pre-compute the nimbers of all sizes of an empty rectangle (no radioactive cells), which is information that can be shared across all test cases.

Second, observe that if it is legal to place a V colony in a cell, then it is also legal to place a V colony in any cell in that column, and similarly for H colonies in a row, within the boundary of the current subproblem's rectangle. We therefore need to check only the rows and columns for legal colony placements, not each individual cell.

Third, we can construct a data structure that allows us to determine whether a colony placement is legal for any row or column in a given rectangle in O(1) time, allowing us to evaluate any game state in O(**R**+**C**) operations. For each row and column in the full grid, create an array. Check the cells in the row or column in ascending order, appending the 1-indexed position of the most recently seen radioactive cell, or 0 if a radioactive cell has not been encountered yet. For example, for the row `.#..#`, the array would be [0, 2, 2, 2, 5]. Suppose we have a rectangle that includes the third and fourth cells of that row. The fourth entry of the array is a 2. Since cell 2 is not in our rectangle (we have cells 3 and 4 only), we can conclude that it is safe to place an H colony in this row of our rectangle. This data structure can be pre-computed for each test case in O(**RC**) time.

To summarize, there are O($\mathbf{R}^2\mathbf{C}^2$) subproblems, and each subproblem takes O(**R**+**C**) operations. If we let N be *max*(**R**, **C**), this leads to O(N^5) total time complexity, sufficient for test set 2. Less efficient solutions might still pass, depending on their implementations.