Code Jam 2018 - Round 2

Analysis: Falling Balls

Test set 1

We can narrow down the scope of Test set 1 with only a couple of insights:

- We cannot put any ramps in the leftmost and rightmost columns, so the balls dropped into those columns must fall straight down to the bottom. So, if either of those columns has no balls, the case is `IMPOSSIBLE`. (Later on in this analysis, it will become clear that this is the only way for a case to be impossible!)
- Suppose **C** has its maximum value of 5. Since we know the leftmost and rightmost columns must have at least one ball each, the only variation among cases comes from where the other 3 balls end up. It turns out that there are only 35 ways to partition 3 balls among 5 columns, and some of these cases are mirror reflections of each other, so there are really only 19 distinct cases to consider. When **C** is smaller than 5, there are even fewer cases.

So, we can experiment offline, generating potential ramp arrangements via simulation or by hand, until we are confident in our solutions for all cases, and then submit; since Test set 1 is Visible, we have little to lose. However, we can avoid this extra work, as follows...

Test set 2

Consider the paths that each ball might take through the toy, and notice that whenever two balls' paths intersect, those balls must end up in the same final column. (Because we can never have a `\` ramp immediately to the left of a `/` ramp, paths cannot cross over each other.)

So, for example, if your friend says there are exactly K balls in the leftmost column, they must have been the balls that were originally dropped into the K leftmost columns. Suppose that one of those balls had instead come from, e.g., the (K+1)-th column; then the path taken by that ball would have crossed all of the paths taken by the balls in the K leftmost columns, so all K+1 of the balls would have ended up in the leftmost column, which is a contradiction.

With those observations in mind, let us think of the i-th column as both a *source* of one ball and a potential *target* of $\mathbf{B_i}$ balls (if $\mathbf{B_i} > 0$). We can scan our friend's list from left to right, and when we encounter a positive $\mathbf{B_i}$ value, we allocate the $\mathbf{B_i}$ leftmost source columns that we have not used so far to target column i. So, for example, if we have the data `3 2 0 0 0 0 2 1`, we map the source column range [1, 3] to target column 1, the source column range [4, 5] to target column 2, the source column range [6, 7] to target column 7, and the source column range [8, 8] to target column 8.

Once we have this mapping, we can move on to designing the toy to ensure that every source column's ball will end up at the bottom of the appropriate target column. We will start with an empty top row, and add ramps and more rows as needed.

For each source column range, we check whether the left endpoint is to the left of the target. If it is not, we do nothing. Suppose, on the other hand, that it is L columns to the left of the target. Then we add L `\` ramps, starting in the top row and the left endpoint's column, and proceeding diagonally, moving one cell to the right and one cell down each time. Notice that this line of ramps will catch all balls in that source column range that need to move to the right, so we do not need to explicitly worry about those balls. Then, we draw similar diagonal lines of `/` ramps running down and to the left from our right endpoints. Finally, if our bottom row is not empty, we add an empty bottom row, as the rules require.

Can we convince ourselves that this construction is optimal? We have already shown that the mapping of source columns to target columns is forced. Consider the largest absolute difference, in columns, between a target column and one of the endpoints of the range of source columns mapped to it. (In our example above, this value is |5 - 2| = 3.) The toy must have at least this many rows (in addition to the empty bottom row), because a ball can only interact with one ramp per row, and each ramp only moves a ball one cell in the desired direction. Observe that our construction uses exactly this many rows, plus the required empty bottom row. For our example, our method produces:

```
../././\..
./././....
../.....
........
```

Notice that there is no danger of creating a `\/` pattern within a range, since no target position could cause that.