Code Jam 2017 - Round 1A

Analysis: Play the Dragon

Play the Dragon: Analysis

Small dataset

The Small limits are large enough to foil pure simulation of all possible choices, so we need to have some insights before proceeding.

- The dragon should cure only when it is forced to — that is, when the knight's next attack would defeat it, and attacking or debuffing would not prevent that. Otherwise, it is always better to do something else.
- All buffs should come before all attacks, so that each of the dragon's attacks gets the benefit of all of the buffs.
- The number of buffs directly determines the number of attacks needed.
- All debuffs should come before all buffs/attacks, so that the total amount of damage the dragon must withstand is minimized.
- If the knight's first attack will defeat the dragon even if the dragon attacks or debuffs in the first turn, the case is impossible.
- If the dragon is forced to cure two turns in a row, then the case is impossible, since that implies that the dragon will have to cure every turn.

These observations add up to a strategy: spend some number D' of turns debuffing, then some number B' of turns buffing, then some number A' of turns attacking, and interleave cures only as needed to not be defeated. Since B' determines A', we only need to consider (D', B') pairs. Since $\mathbf{A_k}$ cannot exceed 100, there is no reason to ever do more than 100 debuffs or 100 buffs; moreover, the worst-case scenario can't possibly require more than a couple hundred turns (D' + B' + A'). We can place much smaller upper bounds than those with a little more thought, but it is already clear that direct simulation should be fast enough for the Small dataset.

So, we can proceed with translating the above strategy into code. We must take care to prioritize actions in the right order. In particular, we must avoid curing when we do not need to or failing to cure when we should. Once that is written, we can simulate each possible (D', B') pair and find the global minimum number of turns, or determine that the case is IMPOSSIBLE.

Large dataset

We noted above that all debuffs should come before all buffs/attacks, and that the number of buffs determines the number of attacks. In fact, the buff/attack part of the problem is independent of the debuff part of the problem. Changing the number of debuffs may change the number of cures, but regardless of how many times we debuff, we have nothing to gain by using more than the minimum number of buff + attack turns; that would just make us waste more turns curing.

We can find this minimum number of buff + attack turns as follows. First, we suppose that we will buff 0 times, and we determine the total number of attacks needed to defeat the knight. Then, we can repeatedly increase B' by 1 and calculate the number of attack turns A' required at that new level of attack power. As soon as this causes the total to get *larger*, we can stop (and take the previous total). It is safe to stop at that point because the total number of turns is given by

B' + ceil($\mathbf{H_k}$ / ($\mathbf{A_d}$ + B' Ã— **B**))

The B' part contributes a line with positive slope; the rest contributes a decaying step function. If that step function were a smooth curve, there would be one point at which the rate of decrease from the curve exactly matched the rate of increase from the linear part, and the function would take on our desired minimum there. Because of the discretization, there may actually be multiple values of B' that yield the minimum number of B' + A' turns, but it does not matter which one we choose; only the total matters.

Finding the minimum B' + A' in this way takes O(sqrt(N)) time, where N is the common upper limit for all of the parameters (10^9 for the Large). This is because once we have raised B' to about sqrt(N), we can defeat the knight in about sqrt(N) attack rounds, and there is no need to buff further. It is also possible to solve this part of the problem using binary search or ternary search, or by solving a quadratic equation.

What about the number D' of debuffs? The key observation here is that we do not need to consider every possible value of D'. For instance, suppose that $\mathbf{H_d}$ is 100, $\mathbf{A_k}$ = 50, and **D** = 1. Reducing $\mathbf{A_k}$ to 49 (which takes 1 turn of debuffing) is as good as reducing $\mathbf{A_k}$ to 48 or 34; in all of these cases, the dragon has to cure every other turn. However, reducing $\mathbf{A_k}$ to 33 (which takes 17 turns of debuffing) means that the dragon only has to cure on every third turn. We might as well only consider these threshold values of D' = 0, 1, 17, 26, 31...; we can leave out the others. We can find each of these values formulaically in constant time.

Once we have these values, we do not even need to simulate them independently. Note that a simulation with D' = 17 begins by looking like a simulation with D' = 1, since we have to do one debuff before we do the other sixteen. So we can perform a single simulation in which we keep track of a number T of turns used so far, and repeat the following:

- Pretend that we will do no more debuffing. Figure out how many additional turns are needed to buff + attack while curing enough to survive. Compare the total (T plus that number) to the best total we have seen so far.
- Figure out how many turns are needed to increase the number of debuffs to the next threshold value, while curing enough to survive. Add that number to T.

It takes additional turns to debuff more, but that debuffing may "pay for itself" by saving curing turns during the buff + attack phase. Our strategy will find the right balance.

Instead of actually simulating the turns, we can take advantage of the way we have chosen threshold values of D': in the debuffing period between two of our values of D', the frequency of curing remains constant. So we can calculate the total number of debuff turns + cures via a formula, and we can do the same for the number of cures in the buff + attack phase. With this optimization, the complexity of this step is O(sqrt(N)); since we also took O(sqrt(N)) time to find the optimal number of B' + A' turns, the algorithm is O(sqrt(N)) overall.

All that remains is to actually implement the above, which is perhaps even harder than coming up with the algorithm; there are many opportunities to make off-by-one errors! Although this is not generally the case in Code Jam, in this particular problem, the limits do allow solutions with some but not all of the above insights, and additional low-level optimizations, to pass within the 8-minute window.