Code Jam 2020 - Virtual World Finals

Analysis: Adjacent and Consecutive

As is the case in many problems about games, this problem revolves around evaluating a given state of the game and deciding whether it is winning or losing. The problem defines mistakes as transitions from winning to losing states, so once we have determined the type of each state, we can easily count up the mistakes.

Test Set 1

In all zero-sum finite two person games with no ties, the basic framework to decide whether a state is winning is the same. Let us call a state *A-winning* (or equivalently, *B-losing*) if player A can ensure a win from it, or *B-winning*/*A-losing* otherwise. For a given state, if the game is over, check the winning condition. Otherwise, if it is player X's turn, try every possible play. If any of them leads to an X-winning state, then the current state is also X-winning. Otherwise, the current state is X-losing. We can implement this as a recursive function, but given that there can be O($\mathbf{N}^2$) possible plays on any given turn, this will only work quickly enough for small values of **N**.

We can see immediately that there can be a lot of repeated work in that recursive evaluation. For example, consider a state S and another state S' such that S' is the result of making P plays on S. Since there are P! ways to make those plays, state S' potentially needs to be evaluated P! times to evaluate S once. Memoizing the function or turning it into a dynamic programming implementation would thus save a lot of recursive calls.

In addition, there are many states that can be evaluated without the need for a recursive call. Some simple cases include states in which there already are two adjacent and consecutive tiles placed (the "already won" condition for player A). We can prune the recursive calls tree by simply returning those states as A-winning. We can prune it further: if there is a play that creates such a pair during A's turn (the "can win immediately" condition), we just do that instead of trying all possible plays for A. We can also restrict B to only try plays that do not leave a known A-winning condition. There are ways to prune the tree further, some of which we explore in the Test Set 2 section below.

We can pass Test Set 1 either with a combination of simple pruning and memoization/dynamic programming, or a lot of pruning.

Test Set 2

The number of possible states in Test Set 2 is just too big to even fit in memory, even after careful pruning. However, pruning enough can help us find sets of equivalent states, that is, groups of states that we can prove have the same winner. By doing that, we can dramatically reduce the number of memoized states.

Consider only A's turns first. We have already mentioned the "already won" and "can win immediately" conditions above. On top of those, we can notice that if there is some set of three unplayed tiles available with three consecutive numbers, and there is some block of three consecutive cells available, A can win by playing the middle number in the middle cell (we call this the "can win in 2 moves" condition). Whatever B plays next, A's following turn will fulfill either the "already won" condition or the "can win immediately" condition.

The conditions above allow us to find some states which are definitely A-winning. Notice that in any other state where it is A's turn, it's impossible for any of the already-placed tiles to end the game as part of an adjacent and consecutive pair. Therefore, they could be replaced by "unavailable" cells without a specific number on them. So, the remaining cells can be represented as the multiset of sizes of the groups of consecutive cells that are left; call it L_C. The exact locations of those groups on the board are not relevant to the final outcome. Since each tile can only form adjacent and consecutive pairs with other leftover tiles, we can similarly represent the remaining tiles with a multiset of lengths of consecutive runs of tiles; call it L_T. For example, the state

```
7 2 6 _ _ 3 _ 4 _ _ 5
```

has the leftover tiles 1, 8, 9, 10, 11, so its leftover cells can be represented by the multiset L_C = {1, 2, 2} and its leftover tiles by the multiset L_T = {1, 4}.

In addition, since the "can win in 2 moves" condition has already been checked, we know that at least one of L_C and L_T does not contain any integer greater than or equal to 3. To simplify the algorithm, notice that the game in which we swap tile numbers and cell numbers is equivalent (because the winning condition is symmetric). Therefore, states with L_C and L_T swapped are equivalent. We can therefore assume it is always L_C that has only integers 1 and 2.

After this compression, the number of states is dramatically reduced. Notice that because we know that there are no three consecutive leftover numbers, at least **N**/3 numbers have been played already. That means that the sum of the integers in both L_C and L_T (which is the same for both multisets) is at most 2×**N**/3. Therefore, the total number of possible multisets L_T under those conditions is bounded by the sum of partitions(K) for every K between 0 and 2×**N**/3, which is partitions(2×**N**/3+1). Given such an L_T multiset, the number of possible multisets L_C of only 1s and 2s such that the sum of L_C and L_T is the same is bounded by **N**/3 (which is the maximum number of 2s). So, the number of states is no greater than (**N**/3)×partitions(2×**N**/3+1), which is a fairly small number for the given limits for **N**. Moreover, we need only one memoization or dynamic programming table for all test cases.

There are multiple sufficiently fast ways to implement the second player's turns (which are the slowest ones), and there are further restrictions that can be placed on plays to optimize our strategy even more. Some of those options result in an algorithm whose complexity makes it clear that it runs within the time limit. Some other options have a theoretical complexity that is either too large or hard to estimate tightly enough to be convinced it's fast enough. Fortunately, it's possible to compute the recursive function for all possible states without reading any data, and be sure that it runs in time before submitting.

The Test Set 3 that wasn't

We considered having a third test set for this problem requiring a polynomial solution. We progressively found solutions taking O($\mathbf{N}^3$) time, O($\mathbf{N}^2$) time and even one requiring only O(**N** log **N**) time. We ultimately decided against adding the test set because the possibility of solving it by guessing the right theorem without proof was a significant concern. The benefit of the extra challenge without making contestants have to read a full extra statement is of course a significant benefit, but it was dampened by our estimation that the likelihood of it being solved legitimately was small. If we had made it worth a small number of points, it would not have been worth the relative effort to solve it. If we had made it worth a lot, though, that would have diminished the value of the work needed for our favorite part of the problem, which is solving Test Set 2.

If you are up to try an extra challenge without spoilers, stop here. If you want some hints on how those solutions go, read ahead!

The first theorem further compresses the states considered in the Test Set 2 solution on A's turns. Consider a state that is not under the "already won", "can win immediately" or "can win in 2 moves" conditions represented by some multisets L_C and L_T, where only L_T can contain integers greater than 2. That state is A-winning if and only if the state represented by multisets L_C and L'_T is A-winning, where L'_T is obtained from L_T by replacing each integer X in L_T with floor(X / 2) copies of a 2 and a 1 if X is odd. This reduces the number of states of the dynamic programming to $O(\mathbf{N}^3)$, and we can process each of those states in constant time because there is only a bounded number of effectively different plays to make.

If we pursue that line of reasoning further, we can find that we can check A-winningness with a small number of cases. Let L_{Ci} be the number of times i is in L_C, and L_{Ti} be the number of times i is in L_T. Because we already assumed that L_C had no integers greater than 2, $L_{Ci} = 0$ for all $i \geq 3$. Let $K = L_{C1}+2\times L_{C2}$ = sum of $i\times L_{Ti}$ over all i be the number of turns left and $Z = (L_{T2}+L_{T3}) + 2\times(L_{T4}+L_{T5}) + 3\times(L_{T6}+L_{T7}) + \ldots$ be the number of times 2 appears in L'_T as described in the previous paragraph. Then, the second theorem we can prove is that:

- If K = 2, then the state is A-winning if and only if $L_{C2} = L_{T2} = 1$.
- If K > 2, then the state is A-winning if and only if K is odd and $2\times(L_{C2}+Z) > K$.

From the need to avoid the "already won", "can win immediately" and "can win in 2 moves" conditions, it is possible to reduce the number of plays on B's turn to a set of a bounded number of options that always contains a winning move if there is one. This, together with the fact that the conditions above can be checked in constant time, yields an algorithm that decides for any state whether it is winning or losing in $O(\mathbf{N})$ time, making the overall process of a test case take $O(\mathbf{N}^2)$ time. With some technical effort, we can represent the board in a way that we can update and use in $O(\log \mathbf{N})$ time to check all the necessary conditions, yielding an algorithm that requires only $O(\mathbf{N} \log \mathbf{N})$ time to process a full test case.

The proofs for the theorems on the paragraphs above are left as an exercise to the reader. As a hint, it is easier to prove the second more general theorem, which is already nicely framed, and then obtain the first theorem as a corollary.