Code Jam 2016 - Round 2

Analysis: Rather Perplexing Showdown

Rather Perplexing Showdown: Analysis

There are multiple ways to attack this problem. We will present two methods for building the correct tournament tree, and a method for finding the arrangement of that tree that produces the alphabetically earliest lineup. It is possible to combine the tree building and tree optimization methods into a single algorithm, but we present the analysis this way for ease of explanation.

Building the tree: starting from the beginning

Let's start from the beginning of the tournament and create each new round. At any point, you have some number of `R`s, `P`s, and `S`s remaining, and you can only create `RP`, `RS`, and `PS` matches, because anything else would result in a tie. Call the number of `RP` matches you will create *x* — that is, you will make *x* of the `R`s match up with `x` of the `P`s. Then all other `R`s must face `S`s, so you will create **R**-*x* `RS` matches. There will be **P**-*x* leftover `P`s and **S**-(**R**-*x*) leftover **S**s, and these numbers must be equal to avoid creating tied matches, so **P**-*x* = **S**-**R**+*x* and *x* = (**R**+**P**-**S**)/2. If this *x* causes an impossible situation (e.g, there must be more `RP` matches than there are `R`s or `P`s), then the answer is `IMPOSSIBLE`. Otherwise, match the players accordingly, note the winners (all `RP`s become `P`s, all `RS`s become `R`s, and all `PS`s become `S`s), and then you have a smaller instance of the same problem. This strategy tells you whether the tournament will end, and how to make all your matchups; with that information and some careful bookkeeping along the way, you can generate the entire tree.

Building the tree: starting from the end

Let's start from the end of a tournament instead. Suppose that the winning player is a `P`. What do we know about the match that produced that winner? One of the participants must have been that `P`, and the other must have been the opponent that the `P` defeated, namely, an `R`. That `R` must have defeated an `S`, and so on. That is, for any node in the tournament tree, including the bottom (winning) node, we can easily regenerate the entire part of the tree that led to it!

This also implies that for a given **N**, there is only one possible (`R`, `P`, `S`) triplet that will produce a successful tournament ending in `R`, and likewise for `P` and `S`. Almost all triplets are doomed to fail! There are only three valid ones for any **N**, and each of them must produce a different winner.

So, we can try all three possible winners (`R`, `P`, and `S`) for every value of **N** from 1 to 12, and store the resulting tournament trees and their numbers of `R`s, `P`s, and `S`s. Then, for each test case, either the given **N**, **R**, **P**, and **S** values match one of the stored trees, or we know the case is `IMPOSSIBLE`.

Finding the alphabetically earliest lineup

Having the tournament tree is not enough, because a tree can generate many possible lineups. For any internal (non-leaf) node in the tree, you can swap the two branches; this does not change the tree, but it does change the initial lineup! For instance, the lineups `PSRS`, `PSSR`, `RSPS`, `RSSP`, `SPRS`, `SPSR`, `SRPS`, and `SRSP` all represent the same tree. There are $2^{\mathbf{N}}$-1 internal nodes in the tree, and we can't try all 2 to the ($2^{\mathbf{N}}$-1) ways of flipping or not flipping each of them. Fortunately, we don't have to.

Consider any pair of players who face off in the first round; let's say they're using moves X and Y, where X is alphabetically earlier than Y. These two players will contribute to two consecutive characters in the lineup; either XY or YX, depending on whether we flip their node. Flipping other nodes in the tree may move this pair of characters around in the final lineup, but it cannot reverse or separate them. So we have nothing to lose by choosing XY; this decision is totally independent of whatever we do with other nodes later. More generally, for any node, we should put the "alphabetically earlier" branch before the "alphabetically later" branch. Moreover, we should optimize shallower nodes in the tree before optimizing deeper nodes, so that we can be sure that we're only making decisions about branches that are already themselves alphabetically optimized.

So we can start with *any* lineup corresponding to our tree (ideally, whatever came out of our algorithm earlier), and first examine the lineup in $2^{\mathbf{N}-1}$ chunks of length 2 and swap the letters in each chunk whenever that would make the chunk alphabetically earlier. Then we can examine the lineup in $2^{\mathbf{N}-2}$ chunks of length 4, and swap the subchunks of length 2 in each chunk whever that would make the chunk alphabetically earlier. And so on, until we've examined and possibly swapped the 2 chunks of length $2^{\mathbf{N}-1}$; that final lineup will be our alphabetically earliest answer.