Code Jam 2013 - Round 2

Analysis: Many Prizes

Let's begin by an observation that will make our life a bit easier: if we reverse all the team numbers (that is, team 0 becomes team $2^{\mathbf{N}}$-1, team 1 becomes team $2^{\mathbf{N}}$-2, etc.), without changing the tournament list. This will result in the final ranks of the teams also being reversed, since it's relatively easy to see that all the records will be reversed (wins becoming losses and vice versa). This means that the problem of having a team rank as low as possible to get into the first **P** is equivalent to the problem of a team ranked as high as possible to get into the bottom **P** (or, in other words, not get into the top $2^{\mathbf{N}}$ - **P**). Thus, if we are able to answer the question what is the lowest rank that can possibly get into the top **P**, we can run the same code to see what's the lowest rank that can possibly be in the top $2^{\mathbf{N}}$ - **P**, subtract one, and reverse, and this will be the lowest rank that will always get a prize. This means we only have to solve one problem — lowest ranked team guaranteed to be in the top **P**

For the small dataset we have at most 1024 teams, which means we can try to figure out the tournament tree that will give a given team the best position, do this for all teams, and see which team was the lowest-ranked one to get a prize. Let's skip this, however, and jump straight to the solution for the large dataset, where 2^{50} teams clearly disallow any direct approaches.

The key observation we can make here is that if we have a team we want to be as high as possible, we can do it with a record that includes a string of wins followed by a string of losses, and nothing else. This sounds surprising, but is actually true. Imagine, to the contrary, that the team (let's call them **A**) has a loss (against some team **B**) followed by a win against **C**, who played **D** in the previous round. Note that up to the round where **A** plays **B** the records of the four teams were identical. Also note that all the tournament trees of the four teams so far are disjoint, and so we can swap them around. In particular, we can swap team **C** and all its tree with team **B** and all its tree. Since we swap whole trees, the records of teams don't change, so now team **A** will play **C** in the first match and win — and so its record is going to be strictly better than it was, no matter what happens next. Thus, any ordering in which team **A** has a loss followed by a win is suboptimal.

This allows us to solve the problem of getting the highest possible rank for a given team. We simply need to greedily win as much as we can. If we're the worst team, we can't win anything. If we're not, we can certainly win the first match. The second match will be played against the winner of some match, so in order to win it we need to be better than three teams. To win two matches, we need to be better than seven teams, and so on.

We can also reverse the reasoning to get the lowest-ranked team that can win the prize. First, let's ask how many matches does one need to win in order to win a prize. If you win no matches, you are in the top $2^{\mathbf{N}}$ (not a huge achievement!). If you win one, you are in the top $2^{\mathbf{N}-1}$. And so on. Once we know how many matches you need to win, you directly know how many teams you need to be better than. Simple python code follows:

```
def LowRankCanWin(N, P):
  if P == 0:
    return -1
  matches_won = 0
  size_of_group = 2 ** N
  while size_of_group > P:
    matches_won += 1
    size_of_group /= 2
  return 2 ** N - 2 ** matches_won

def ManyPrizes(N, P):
  print 2 ** N - LowRankCanWin(N, 2 ** N - P) - 2, LowRankCanWin(N, P)
```