MATH 495R, HOMEWORK 22: NIM (CONTINUED)

In this lab, we will complete our program to teach a computer to learn to play the game of Nim. Note that we will *not* program the computer to play perfectly–indeed, we will begin with a computer that plays randomly (and probably loses most of the time). We will then train the computer to recognize winning moves, until the computer plays perfectly.

Before beginning this lab, be sure that your lab from last week is working correctly. In particular, you should have code that allows two human players or a human and the computer to play, and it should allow two computer players to play multiple games.

For this week, use last week's lab as a template. We will start with what was part 4 last week. You have probably already written this, but we give a bit more detail about it here—in order for it to work with the other functions that we will write, make sure that it matches this description.

4. We now teach the computer the rules of the game, so that it can play very badly. The function `computer(n,position,moves,strategy)` takes as input an integer `n` indicating the player number, a vector `position` giving the board configuration, a vector `moves` which we describe below, and the vector `strategy` that tells the computer all valid moves.

   The variable `moves` should be a vector of length two. The first component of this vector will consist of moves made so far by player one, and the second will consist of moves made so far by player 2. As the computer makes a move, it will add information describing that move to `moves[n]`.

   This routine should first convert `position` into a three-digit integer `pos`. It will then look at the possible moves in `strategy[pos]`. Set `mx` equal to the largest value of the third component of the possible moves, and then create a vector `bestmoves` containing the index in `strategy[pos]` of all the possible moves whose third component matches that maximum value. As an example, if `position=[3,1,2]` then `pos=312` and we might have

   ```
   strategy[pos]=[[1,3,80],[1,2,20],[1,1,40],[2,1,80],[3,2,60],[3,1,80]]
   ```

   The maximum value of the third component is 80, and we should end up with `bestmoves=[0,3,5]`, indicating the the 0, 3, and 5 entries of the vector achieve the maximum.

   Randomly choose an element `m` of `bestmoves` (this can be done, for instance, with the command `m=bestmoves[random.randint(0,len(bestmoves)-1)]`, and execute the indicated move (i.e. the move described in `strategy[pos][m]` by subtracting the appropriate number of objects from the appropriate pile. Print out "Computer player `n` takes `a` objects from pile `b`", and append the vector `[pos,m]` to the vector `moves[n]`.

   Finally, if the current value of `position` is equal to `[0,0,0]`, print "Computer player `n` wins" and exit the function. Note that no value needs to be returned.

   Your function should now allow you to play a computer. the computer should, at this point, be selecting random moves from all possible moves (because every possible move has a "quality" of 50). You should be able to defeat this random computer easily.

5. Copy your function `computer` into the function `computertrainer`. We will modify it slightly to make it more suitable to training the computer. To do this, we will want to do two things differently.

   First, eliminate the print statements. We will train the computer by having it play thousands of games against itself, and we don't want to see the output of those games.

   Second, when selecting the best moves, choose all the moves with a quality greater than `mx-90`. If we only select the best moves, then the computer will only ever try one winning move of all the possible moves; namely the first one that leads it to a victory. There could be other, even better moves available, and we want the computer to try them out. Note that the value 90 is arbitrary, after the next step you may want to experiment with it, to see how best to train the computer. Basically a value closer to 100 reserves judgement on whether a move is good or bad until later; a smaller value judges a move to be permanently bad as soon as there is one that is marginally better. Using too low a value here can cause the computer to get stuck trying a marginal move many times, before finally deciding that there is a better move.
6. Our final function will be `traincomputer(strategy)`. This function will teach the computer which moves are the best, so that it can consistently win.

   We want the computer to play thousands of games against itself. Each time the computer wins, it should increase the quality of the moves that it made; each time it loses, it should decrease the quality. Eventually, the best moves will become apparent. Although this is the basic idea, we will make some modifications so that the best moves become apparent faster.

   The function `traincomputer(strategy)` will consist of a large loop, which will cause the computer to play against itself over and over, becoming a little bit better each time.

   The first thing to do inside the loop is to set the starting position, `position=[7,5,3]`, the player number `playernumber=2`, and initialize the list of moves that have been made, `moves=[[],[]]`. Note that the moves made are divided between moves made by player one and moves made by player 2.

   Next, have the computer play a game against itself, using the `computertrainer` function. Put the function inside a loop that does the following, until the game is over:
   (a) Switch the player number (from 1 to 2, or from 2 to 1).
   (b) Execute `computertrainer(playernumber,position,moves,strategy)`.

   When the game ends, keep track of which player was the winner, and which was the loser.

   Now, examine the moves made by the winner. Each of these moves will consist of a pair, `[pos,move]`, where `pos` is the position before the move, and `mov` is the number of the move that was made from that position. For each of them increase the value of `strategy[pos][mov][2]` (i.e. the quality of that move) by 29 (but only up to a maximum of 100).

   Now examine the moves made by the loser. Each of these moves will consist of a pair, `[pos,move]`, where `pos` is the position before the move, and `mov` is the number of the move that was made from that position. For each of them decrease the value of `strategy[pos][mov][2]` (i.e. the quality of that move) by 11 (but only down to a minimum of 0).

   As a further innovation, the last move made by the winner is a certain win; adjust the quality of that move to 1000 (we always want to make that move if we are in this position, regardless of how good the other moves are). The last move made by the loser leads immediately to a win by the other player, decrease the quality of that move to -1000 (since we would prefer to never make that move again), and don't need any more practice to know that it is a bad move.

With the parameters listed above, the strategy should produce to a good strategy after about $15,000$ games have been played. Changing the parameters (the 90 in the previous part, and the 29 and 11 in this part) will adjust the speed at which the strategy approaches a good strategy.