MATH 495R, HOMEWORK 20: NIM

In this Lab, we will program 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.

Rules of Nim. A games of Nim begins with three piles of objects; one having 7 objects, the other having 5 objects, and the last having three objects. Two players take turns removing objects from the piles. A turn consists of removing any positive number of objects from one of the piles. The winner is the last player that is able to make a valid move (i.e. the player who removes the last objects from the last nonempty pile).

There are many variations of Nim on the internet; we could use different number of objects in each pile, or have a different number of piles. In some versions, the person who takes the last object loses. For our game, we will use the rules listed above.

Nim is a completely solved game. With perfect play, depending on the starting configuration, either the first player will always win, or the second player will always win. For our starting configuration, unless the first player plays poorly, the first player will always win.

A template for this lab is available at

`http://math.byu.edu/~doud/Math495R/Lab20template.py`

1. The first function that we will write is `player(n,position)`. The input to this function is an integer `n` indicating whether the human player is player one or player two, and a list `position` indicating the position of the board prior to the players move.

   The function should ask the player "Player n, what move would you like to make?" (with "n" filled in with the playernumber) and accept an answer as a string. The string should be of the form "3 1" (two integers separated by a space). The first integer indicates which pile the player wishes to remove objects from, and the second indicates how many objects should be removed. The program should parse this string into two integers, and check whether it is a valid move for the indicated position. For example, if `position=[7,5,3]`, then "3 1" would be a valid move, but "3 5" would not (since there are not five objects on pile three to remove).

   If the move is not valid, the program should indicate this, and ask the player again for a valid move.

   If the move is valid, the program should subtract the given number of objects from the correct component of `position`. Note that the program does not need to return a value; `position` is a vector that was passed to the function, so changing a component will change the original vector.

2. The second function that we need to write is `\playgame(strategy)`. For the moment, we will not use the input variable `strategy`; it will be used to store the computer's strategy, once we have developed a strategy for the computer.

This function should begin by asking "Would you like to play a game? (Y/N) ". It should accept input in the form of a string. If the string is either "Y" or "y", it should proceed; if the input is "N" or "n" it should terminate, and other input should lead to the question being asked again.

If the program proceeds, is should ask

"Player 1: Computer (C) or Human (H)? (C/H) "

and accept a string as an answer. It should store this answer in the variable `P1`. If the answer is neither "C" nor "H", it should ask the question again.

It should then repeat the process for player 2, storing the response in the variable `P2`.

If both players are computers, the program should ask "How many games?" and accept an integer as a response. If either player is human, this question should not be asked–a single game will be played.

If more than one game is to be played (which should only happen with two computer players), the program should print

"—Beginning Game 1—".

The program should then initialize the board, setting the position to `[7,5,3]`. Beginning with player one, it should print the board position, in the format

"The piles have 7, 5, 3 objects"

(where the numbers are taken from the vector indicating the position) and then call the function `player(n,position)` (if player `n` is a human) or `computer(n,position,moves,strategy)` (if player `n` is a human) to allow the player to make a move. For instance, if `position=[5,2,1]`, it should print "The three piles have 5, 2, 1 stones." and then call `player(n,position)`, which will ask the player for a move. It should then change the player number, and repeat, until the position becomes `[0,0,0]`.

Once the game is over, if both players are computers and multiple games have been requested, it should print

"—Beginning Game 2–"

and initialize and play a new game with two computer players. It should repeat this until the desired number of games have been played. Once all of the computer vs. computer games are over, it should print

"—Completed n games—-"

where `n` is the number of games requested.

If either player is human, then once the game is over, to program should ask "Would you like to play again? (Y/N)", and depending on the answer, start over (asking if the players are human or computers), or terminate.

At this point, you should be able to play a two-person game (with no computer players) by running the program.

3. In order to have a computer play, we need to tell it how to play. We will create a vector called `strategy`, indexed by integers from 0 to 753. If the integer `n` is a valid board position for the game (i.e. the first digit is from 0 to 7, the second digit from 0 to 5, and the third digit from 0 to 3), then `strategy[n]` will itself be a vector containing all possible moves from the given position. Each move will be represented by a triple of

numbers, representing the pile, number of objects to remove, and the quality of the move (to be described later). For instance, if the integer `n` is equal to 312, the board position will be `[3,1,2]`, and `strategy[312]` will be the vector

$$[[1,3,50],[1,2,50],[1,1,50],[2,1,50],[3,2,50],[3,1,50]].$$

For now, we just set the third number in each triple to 50 (it will be used to determine which moves are best, later).

Create a function `initialize()` that will create and return the desired vector `strategy`.

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.

   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`, and execute the indicated move (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]`. Note that if `m` is the number that we select from bestmoves, then we want to execute the `m`th move of `strategy[pos]`.

   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.

THIS IS THE END OF LAB 20. EVERYTHING AFTER THIS WILL BE LAB 21.

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.

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 two.

   Next, have the computer play a game against itself, using the `computertrainer` function. 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]` 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]` 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 function `traincomputer(strategy)` should produce 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. Feel free to adjust these parameters to experiment, and see if you can get a good strategy in less than $15,000$ games.