Lab 11: Playing a Two Player Game

In this lab, you will be tasked with searching through all possible states of the 2-player game Nim. However, you will be the writing code in such a manner that it will be easy to extend to other two player games as well. In the homework, you will extend this code to determine the best possible move given a current state in the game.

Choices made in a game naturally form a tree. The parent of a particular state in the game is the state of the game on the previous move. Going back in time leads back to the root of this tree, which is the starting configuration of the game. Each move creates new possible moves that branch out to form the different possible histories of the game.

Nim

In the game of Nim, there are rows of objects with a number of objects in each row. An example configuration is a row of 2, a row of 3, and a row of 4 objects. Two players take turns removing objects from any one row they choose. Each player must remove at least one object and can remove as many objects as they want. The person to remove the last object loses.

Representing a Generic Two Player Game

In order to represent a generic two player game, the `GameState` abstract data type is as follows.

- `moves` - return a list of all possible moves the current player can make. A move is another `GameState`.
- `isover` - return `True` if and only if the game is over and the current player has lost.
- `draw` - return True if and only if the current game state is a draw (that is, a tie). *Note, there are no ties in Nim*.

Representing Nim

A skeleton class, `Nim`, that represents a state of Nim will be provided. Your job will to fill in the following methods:

- `__init__(self)` - The initializer that takes in the board configuration and whose turn it is
- `moves(self)` - return an iterable containing all Nim states that can be reached from the current Nim state for all possible moves the current player can make.
- `isover(self)` - returns `True` if the game is over and the current player has lost. For Nim, this means just one object is left. Otherwise, return `False`.

Searching Through Nim

A skeleton class, `GameTree`, that will search through game states will also be provided. You will fill in the following methods. It is important that this class is not depended on what `GameState` it works with. In other words, it should work with abstract data type of a game state as defined above.

`__init__(self, gamestate)` - takes in a `GameState` of any two player game and generates a tree of all possible moves from the current game state.

The `GameTree` class can be thought of as both a tree and a node. The subtree rooted at a given `GameTree` describes all possible ways the game could end from that game state. You will need to use the public methods on the gamestate (especially `moves`) in order to build the tree. Store the children of a `GameTree` in a public attribute called `moves`.