Nim, nim.py and games.py

Rules of Nim

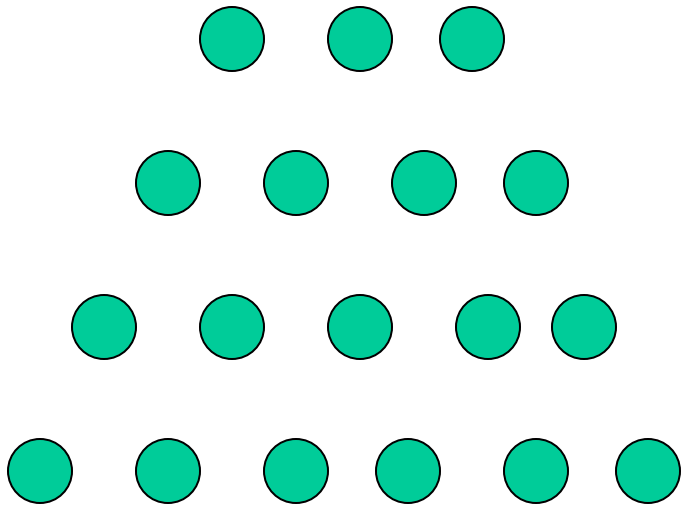
- Impartial two-player game of mathematical strategy
- Alternate turns, removing some items from ONE heap until no pieces remain
- Must remove at least one item per turn
- Last player able to move wins
- Variations:
  - Initial number of heaps and items in each
  - Misère version: last player who can move loses
  - Limit on number of items that can be removed

History of Nim Games

- Believed to have been created in China; unknown date of origin
- First actual recorded date- 15th century Europe
- Originally known as *Tsyanshidzi* meaning "picking stones game"
- Presently comes from German word "nimm" meaning "take"

Adapted from a presentation by Tim Larson and Danny Livarchik

Demonstration



Player 1 wins!

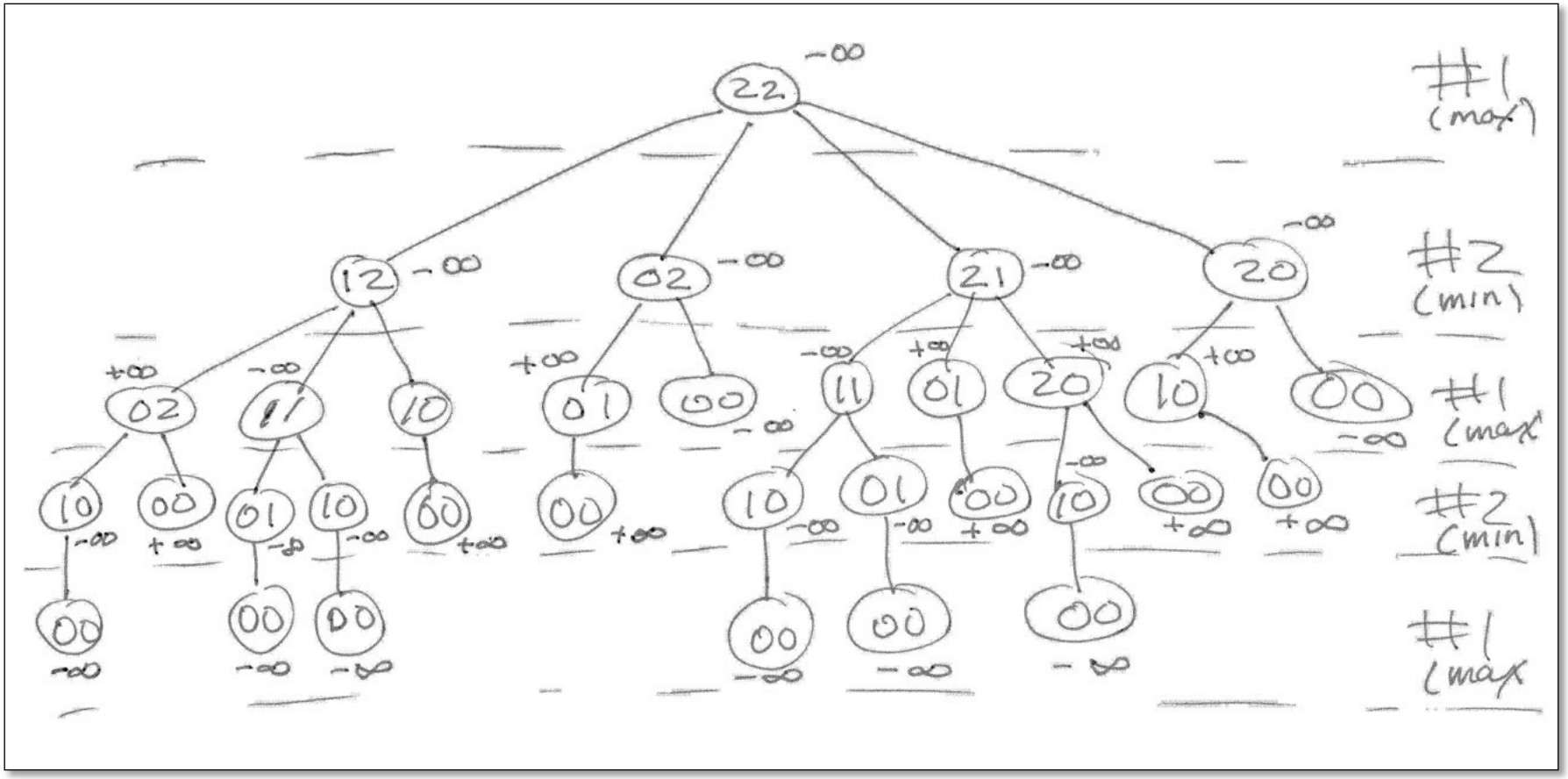
Theoretical Approach

- Theorem developed by Charles Bouton in 1901
- To win, goal is to reach a nim-sum of 0 after each turn until all turns are finished
- **Nim Sum:** exclusive-or of corresponding numbers when represented in binary

  Exclusive-or is used for adding two or more numbers in binary and ignores all carries
- This is a ***strong method***; we can also use the ***weak method*** of traditional game playing:

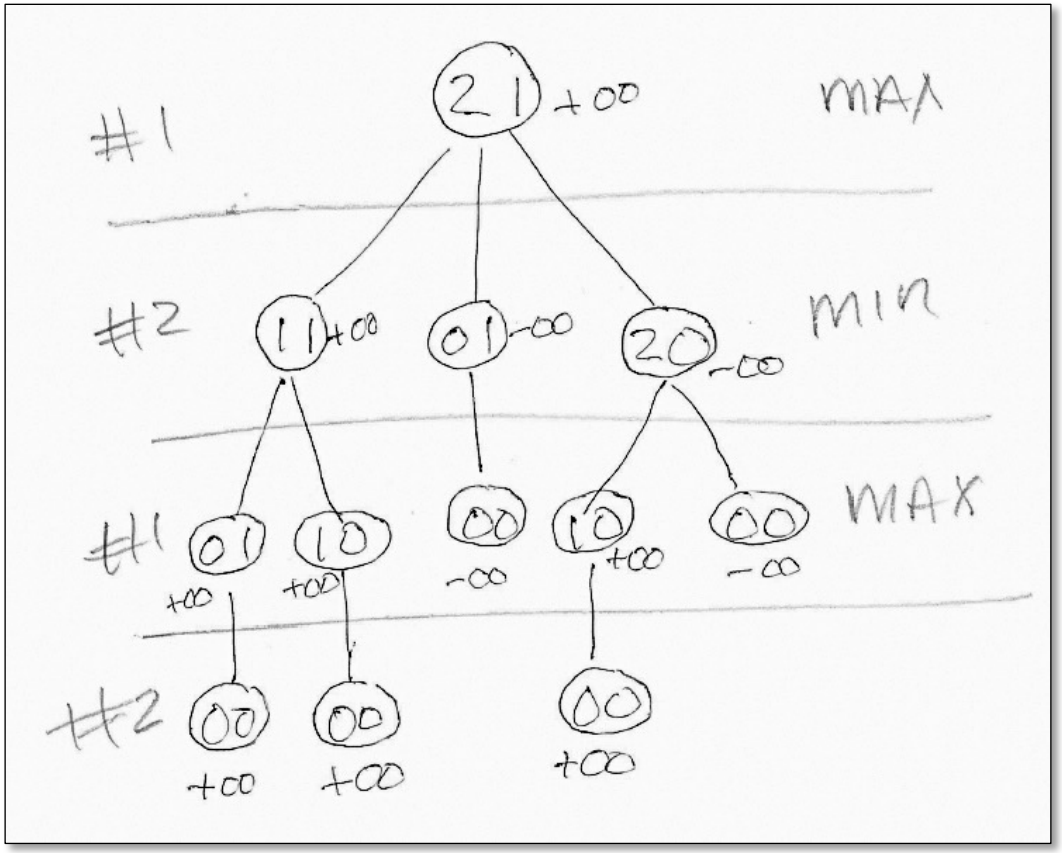
  Evaluation function + lookahead + minimax

Game Tree for (2,2): P2 wins



- **No matter what move P1 makes, it will lead to a win for P2**
- **Note: scores are always w.r.t. the root player, aka P1**

Game Tree for (2,1): P1 wins



- **If P1 removes 1 stone from the 1st pile, P1 will win**
- **Note: scores are always w.r.t. the root player, aka P1**

games.py

- AIMA's python framework for multiple-player, turn taking games
- Implements minimax and alphabeta
- For a new game, subclass the Game class and
  - Decide how to represent the "board"
  - Decide how to represent a move
  - State: (minimally) a board and whose turn to move
  - Write methods to (1) initialize game instance, (2) generate legal moves for state, (3) make move in state, (4) recognize terminal states (win, lose, or draw), (5) compute state's utility for player, (5) display a state

```
class Game:
    """A game is similar to a problem, but it has a utility for each state and a terminal test instead of a path cost and
a goal test. To create a game, subclass this class and implement actions, result, utility, and terminal_test. You may
override display and successors or you can inherit their default methods. You will also need to set the .initial
attribute to the initial state; this can  be done in the constructor."""

    def actions(self, state):
        """Return a list of the allowable moves at this point."""
        raise NotImplementedError
    def result(self, state, move):
        """Return the state that results from making a move from a state."""
        raise NotImplementedError
    def utility(self, state, player):
        """Return the value of this final state to player."""
        raise NotImplementedError
    def terminal_test(self, state):
        """Return True if this is a final state for the game."""
        return not self.actions(state)
    def to_move(self, state):
        """Return the player whose move it is in this state."""
        return state.to_move
    def display(self, state):
        """Print or otherwise display the state."""
        print(state)
    def __repr__(self):
        return '<{}>'.format(self.__class__.__name__)
```

play_game method

```
def play_game(self, *players):
    """Play an n-person, move-alternating game."""
    state = self.initial
    while True:
        for player in players:
            move = player(self, state)
            state = self.result(state, move)
            if self.terminal_test(state):
                self.display(state)
                return self.utility(state, self.to_move(self.initial))
```

Assumptions about states

- games.py assumes you represent a state as a **namedtuple** with at least two fields:
  - *to_move*: whose turn it is to move
  - *board*: current board configuration
- Example for Nim

  NimState = namedtuple('Nim', 'to_move board’)

namedtuples

```
>>> from collections import namedtuple
>>> Person = namedtuple('PER', 'name age sex')   # note order of properties
>>> p1 = Person(name='john', sex='male', age=20)   # note order of properties
>>> p1
PER(name='john', age=20, sex='male')
>>> p1.sex
'male'
>>> p1[1]
'20'
>>> p2 = Person()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: __new__() missing 3 required positional arguments: 'name', 'age', and 'sex'
>>> p2 = Person('mary', 'female', '21')  # note, wrong order!
>>> p2
PER(name='mary', age='female', sex=21)
>>> p2 = p2._replace(age=21, sex='female') # create a new tuple object to fix
>>> p2
PER(name='mary', age=21, sex='female')
```

- Like lightweight objects
  no methods or inheritance
- Like tuples, immutable, so can serve as dictionary keys
- Reference elements with names or numbers
- Documentation

```
def minmax_decision(state, game):
    """Given a state in a game, calculate the best move by searching
    forward all the way to the terminal states. [Figure 5.3]"""”

    player = game.to_move(state)

    def max_value(state):
        if game.terminal_test(state):
            return game.utility(state, player)
        v = -infinity
        for a in game.actions(state):
            v = max(v, min_value(game.result(state, a)))
        return v

    def min_value(state):
        if game.terminal_test(state):
            return game.utility(state, player)
        v = infinity
        for a in game.actions(state):
            v = min(v, max_value(game.result(state, a)))
        return v

    # Body of minmax_decision:
    return max(game.actions(state), key=lambda a: min_value(game.result(state, a)))
```

Minimax
games4e.py

```
def minmax_decision(state, game):
    """Given a state in a game, calculate the best move by searching
    forward all the way to the terminal states. [Figure 5.3]"""”

    player = game.to_move(state)

    def max_value(state):
        if game.terminal_test(state):
            return game.utility(state, player)
        v = -infinity
        for a in game.actions(state):
            v = max(v, min_value(game.result(state, a)))
        return v

    def min_value(state):
        if game.terminal_test(state):
            return game.utility(state, player)
        v = infinity
        for a in game.actions(state):
            v = min(v, max_value(game.result(state, a)))
        return v

    # Body of minmax_decision:
    return max(game.actions(state), key=lambda a: min_value(game.result(state, a)))
```

Minimax
games4e.py

```
def minmax_decision(state, game):
    """Given a state in a game, calculate the best move by searching
    forward all the way to the terminal states. [Figure 5.3]"""”

    player = game.to_move(state)

    def max_value(state):
        if game.terminal_test(state):
            return game.utility(state, player)
        v = -infinity
        for a in game.actions(state):
            v = max(v, min_value(game.result(state, a)))
        return v

    def min_value(state):
        if game.terminal_test(state):
            return game.utility(state, player)
        v = infinity
        for a in game.actions(state):
            v = min(v, max_value(game.result(state, a)))
        return v

    # Body of minmax_decision:
    return max(game.actions(state), key=lambda a: min_value(game.result(state, a)))
```

Minimax
games4e.py

Python max/min with key

- **max(game.actions(state), key=lambda a: min_value(game.result(state, a)))**
- max/min with key like argmax/argmin for collections
- Example:

```
words = "the dog chased a cat".split()
>>> max(words, key=len)
'chased’
>>> argmax = lambda iterable, func: max(iterable, key=func)
>>> argmax(words, len)
‘chased’
```

Caution

- Python lists are mutable objects
- If you use a list to represent a board and want to generate a new board from it, you probably want to copy it fist

```
new_board = board[:]
# alternatively: new_board = board.copy()
new_board[3] = new_board[3] - 1
```

Players

games.py framework defines several players

- **random_player**: choses a random move from among legal moves
- **alpha_beta**: uses alpha_beta to choose best move, optional args specify cutoff depth (default is 4) and some other variations
- **human_player**: asks user to enter move

Variations

```
def make_alphabeta_player(N):
    """ returns a player function using alpha_beta search to depth N """
    return lambda game, state: alphabeta_search(state, game, d=N)

# add to the PLAYER dictionary player function named ab1,ab2,...ab20
# that use alpha_beta search with depth cutoffs between 1 and 20

for i in range(20):
    PLAYER['ab'+str(i)] = make_alphabeta_player(i)
```