



# Nim, nim.py and games.py

# Rules of Nim

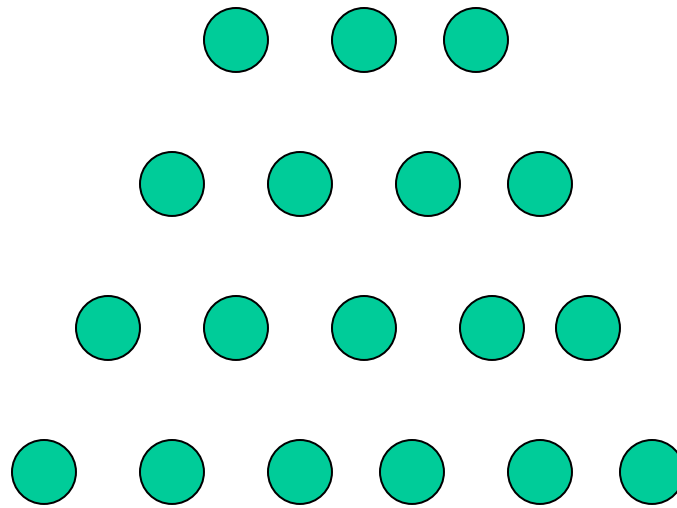
- Impartial two-player game of mathematical strategy
- Alternate turns removing some items from ONE heap until no pieces remain
- Must remove at least one item per turn
- Last player to be able to remove a wins
- Variations:
  - Initial number of heaps and items in each
  - Misere play: last player who 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- 15<sup>th</sup> century Europe
- Originally known as Tsyanshidzi meaning “picking stones game”
- Presently comes from German word “nimm” meaning “take”

# Demonstration

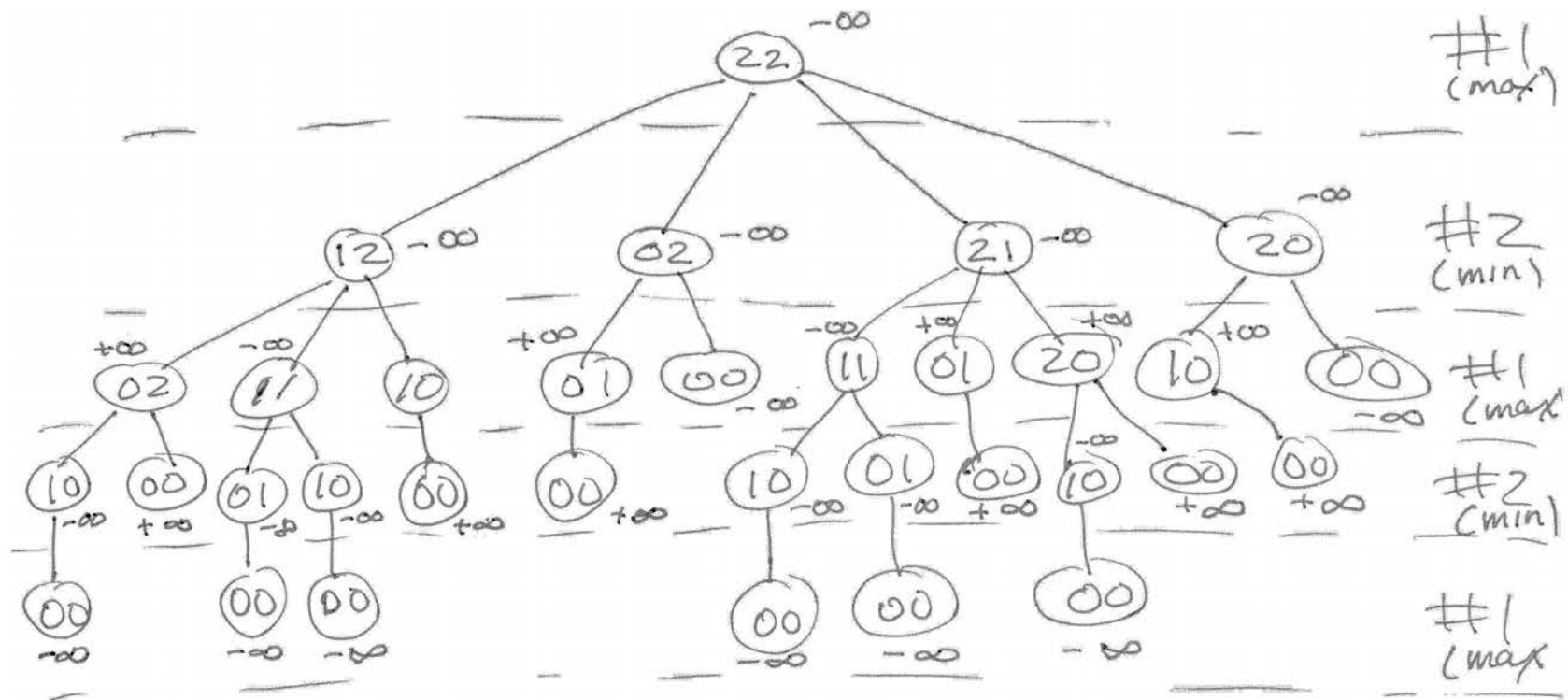


Player 1 wins!

# Theoretical Approach

- Theorem developed by Charles Bouton in 1901
- To win, the 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

# Tree for (2,2)



# games.py

- Peter Norvig'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
  - A state is (minimally) a board and whose turn to move
  - Write methods to (1) initialize game instance, (2) generate legal moves from 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

# Assumptions about states

- games.py assumes you represent a state as a **namedtuple** with at least two fields: `to_move` and `board`

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



# namedtuples

```
>>> from collections import namedtuple
>>> Person = namedtuple('PER', 'name age sex')
>>> p1 = Person(name='john', sex='male', age='20')
>>> 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')
```

```
>>> p2
```

```
PER(name='mary', age='female', sex='21')
```

- Like lightweight objects
- Also like tuples
- Immutable, so can serve as dictionary keys

# 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 first

```
new_board = board[:]
```

```
new_board[3] = new_board[3] - 1
```

# Players

The games.py framework defines several players

- `random_player`: chooses a random move from among legal moves
- `alphabeta_player`: uses `alpha_beta` to choose best move, optional args specify cutoff depth (default is 8) and some other variations
- `human_player`: asks user to enter move

# Variations

```
def make_alpha_beta_player(N):
```

```
    """ returns a player function using alpha_beta search to depth N """
```

```
    return lambda game, state: alpha_beta_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_alpha_beta_player(i)
```