CMPUT 396 – Nim Game

```
        *
        *
        *
        *
    *   *
*   *   *
a   b   c
```

Start with *k* piles of stones.

On a move, a player selects a pile and removes any number of stones from that pile.

The last player to move wins.

In this game, how do we win?

Algorithm tips:

- Use recursion
- Avoid code duplication (e.g. negamax)
- Avoid recomputation (e.g. memoization)

Dynamic Programming (memoization): tackle subproblems, write down intermediate results so you don't recompute it later (recursion can be helpful in such solutions)

Memoization Example:

```
def fib(n):
  if n<=1: return n
  return fib(n-2)+fib(n-1)

for j in range(40):
  print(j, fib(j))

how long does this take to execute?

memoize, don't recompute

def dpfib(n):
  F = [0,1]
  for j in range(2,n+1):
    F.append(F[j-1] + F[j-2])
  return F[n]
```

How long does it take to compute fib(40) with no memoization? Just count the number of function calls to estimate.

- Recurrence Relation
    - Number of calls for fib(10): 1 + # for fib(9) + # for fib(8)
- Number of calls for fib(2): 1 + # for fib(1) (1, base case) + # for fib(0) (1, base case)

So far for Tic-Tac-Toe we could solve it with simple minimax, alphabeta, and negamax. Does this work with nim?

Consider the game state search tree for nim (10 10 10 10 10).

What moves can we make? We can take 1-10 from any of the 5 piles. So, the root state has 50 children, 50*49 grandchildren…
At depth 10, this tree already has 50*49*48*47*…*41 > 3 e16 nodes. This tree is too big to solve in a reasonable amount of time. What else can we do?

We can exploit isomorphisms and memoize.
iso = same, morph = shape

As with Tic-Tac-Toe, we can use a transposition table to generate only one node per equivalence class (set of isomorphic states).

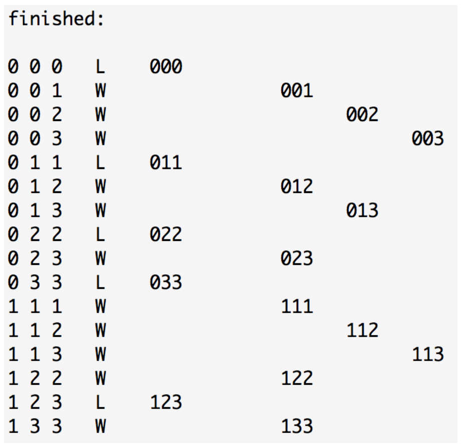
e.g. nim (1 2 3)

- From (1 2 3), some possible moves: 012 013 023 112 113 122 …
    - There are some isomorphic states here: 102 120 012 (only need one)
    - In the tree, like in tic-tac-toe, 2 paths can lead to different nodes that are the same isomorphic state (look at transposition table example in ttt notes).
- From (0 1 3) (we've chosen a move), some possible moves: 003 012 011 010 …

Solving nim (1 3 3):

```
0 0 0
0 0 1
0 0 2
0 0 3
0 1 1
0 1 2
0 1 3
0 2 2
0 2 3
0 3 3
1 1 1
1 1 2
1 1 3
1 2 2
1 2 3
1 3 3

start
0 0 0  L
0 0 1  ?
0 0 2  ?
0 0 3  ?
0 1 1  ?
...

next: states that get to losing state win
0 0 0  L
0 0 1  W
0 0 2  W
0 0 3  W
0 1 1  ?
...
```

```
next: states that get only to winning states lose
0 0 0  L
0 0 1  W
0 0 2  W
0 0 3  W
0 1 1  L   <-
...

next: states that get to losing state win
0 0 0  L
0 0 1  W
0 0 2  W
0 0 3  W
0 1 1  L
0 1 2  W   <-
0 1 3  W   <-
0 2 2
0 2 3
...

next: states that get only to winning states lose
0 0 0  L
0 0 1  W
0 0 2  W
0 0 3  W
0 1 1  L
0 1 2  W
0 1 3  W
0 2 2  L   <-
0 2 3
...
```



```
finished:

0 0 0   L    000
0 0 1   W                001
0 0 2   W                         002
0 0 3   W                                  003
0 1 1   L    011
0 1 2   W                012
0 1 3   W                         013
0 2 2   L    022
0 2 3   W                023
0 3 3   L    033
1 1 1   W                111
1 1 2   W                         112
1 1 3   W                                  113
1 2 2   W                122
1 2 3   L    123
1 3 3   W                133
```

In Nim, it is easier to work from the bottom-up because we know what the losing state is (board is empty and it is your turn to play, this is the terminal node.) It's a game that gets simpler as the game progresses (same with Chess, because pieces are taken off the board).

We can build up possibilities from there. Look under "Finished:", we can build up from (0 0 0) to figure out which states are winning and which states are losing. Note that we are only considering optimal play (so some states like 002 and 003 don't make sense).

The search space for Nim is narrow at the bottom so it is more productive to use a bottom-up approach and always take the best move.

Game Space Search

DAG vs Tree:

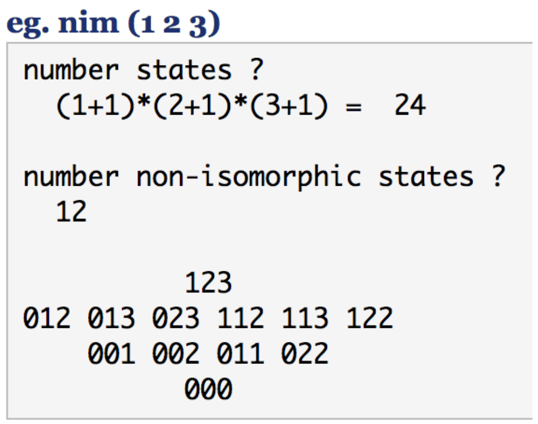
- DAG = directed acyclic graph
- Better to use DAG than tree because in a directed acyclic graph different paths from the root can lead to the same child (this is the case in TTT and Nim).

Pro: fewer nodes in DAG than tree, so less time to examine all the nodes

Con: DAG code usually more complicated than tree code, especially if information from child is backed up to parent (in a tree each child has one parent and in a DAG each child can possibly have many parents).

State Equivalence: we can compress the search space by group nodes into equivalence classes.

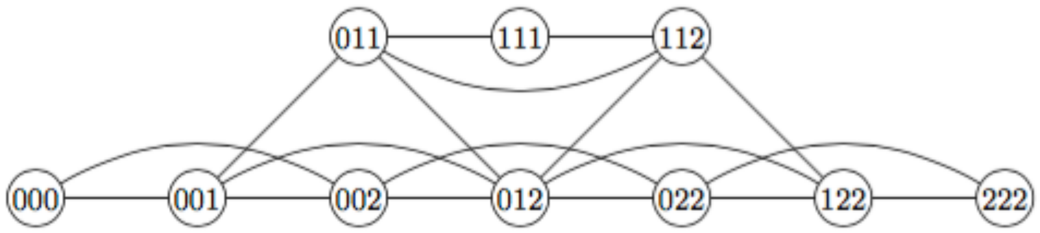
e.g. Nim, put isomorphic states in same class
(1 2 3) = (1 3 2) = (2 1 3) = (2 3 1) = (3 1 2) = (3 2 1)



eg. nim (1 2 3)

```
number states ?
  (1+1)*(2+1)*(3+1) =  24

number non-isomorphic states ?
  12

          123
012 013 023 112 113 122
    001 002 011 022
          000
```

Nim(2 2 2) DAG:



- Root node is (2 2 2), the start of the game
- All arcs for from right to left, for example, 222 has two children: 122 and 022
- Exercise (solution at the end): write X under each losing state, draw arrow on each winning edge (winning edges go to losing edges)

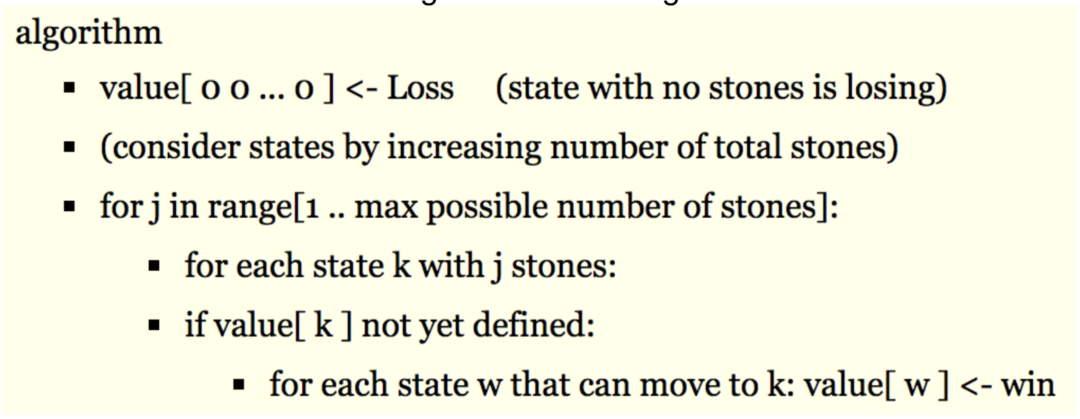
Algorithm

Memoization is also called **dynamic programming**. Instead of recomputing, we write values when we compute them the first time so we can look them up later. Memoization works if states can be ordered so that the value of a state depends only on values of previous states.

Dynamic Programming Nim Algorithm:
- Order states by number of stones so each move results in a state with fewer stones
- In order, from fewest stones to most, compute the win/loss value for each state

Three definitions:
- Any position with no moves is losing.
- A position is **losing** is every move from that position is to a winning position.
- A position is **winning** is it is not losing, i.e. if some move from that position is to a losing position.

If we don't have a move to a state where the opponent loses, every move we make is to a state where the opponent will win. All the states we can reach from a losing state must be a winning state. For example, 001 (W), 002 (W), 003 (W) from 000 (L). All the states we can reach from our winning states are losing states.



```
algorithm
  ▪ value[ 0 0 ... 0 ] <- Loss     (state with no stones is losing)
  ▪ (consider states by increasing number of total stones)
  ▪ for j in range[1 .. max possible number of stones]:
      ▪ for each state k with j stones:
      ▪ if value[ k ] not yet defined:
          ▪ for each state w that can move to k: value[ w ] <- win
```

Correctness?
Because we consider states by increasing number of stones, if the value of a state is not yet defined by the time we get to it, it must be the case that every move from that state is to a state whose value we have already seen, and (since we would have set the value otherwise) all those values must be wins, so the value of the undefined state must be a loss.

```
for each state:
  value(state) = False

for k in range(1, sum_pilesizes):
  for each k-stone state:
    if not value(state): → if value is false, all reachable states must be a win (every other state is still false)
      for each state t reachable by adding stones to a pile of s:
        value(t) = True
```

Trace dpsolve(3 3 3):

```
showing sorted equiv classes only

 0 stones: 000 F, update 001 002 003 T

 1 stones: 001 T, no update

 2 stones: 002 T
           011 F  012 013 111 112 113 T

 3 stones: 003 T
           012 T
           111 T

 [exercise: show trace output for 4 stones]

 ...
```

```
4 stones: 013 T
          112 T
          022 F  023 122 222 223 T
```

From Python code:
```
# for each losing state, find winning states that reach it
      for j in range(len(self.wins) - 1): # nothing reaches last state
        if not self.wins[j]: # loss, so find all psns that reach j
          cj = self.crd(j)
          #print(cj,'loses, find all wins that reach this')
          for x in range(len(cj)):
            cjcopy = deepcopy(cj)
            for t in range(1+cj[x], 1+self.dim[x]):
              cjcopy[x] = t
              pjc = self.psn(cjcopy)
              self.wins[pjc], self.winmove[pjc] = True, j
```

Runtime:
- Each state causes an update at most one time
- Number of states at most product_pilesizes
- Each update takes time at most sum_pilesizez
- Total time O(product_pilesizes * sum_pilesizes)

$O(n^2)$ → runtime of n increases on average by some factor times n
$C * n^2$: $C * 100^2$ → $C * 200^2 = 200^2/100^2 = 4/1 = 4$ times as much

- So if n doubles, runtime is 4 times greater

$O(n)$: $C * 100$ → $C * 200 = 200/100 = 2$, increases by factor of 2
$O(n^3)$: $C * 100^3$ → $C * 200^3 = 200^3/100^3 = 8$, increases by factor of 8

Checkers:

- When does dynamic programming work well? When the number of positions (or equivalence classes) eventually drops to a manageable number.
- 1994 (1996) – Schaeffer's check program Chinook becomes the first bot to be a world champion of some game
- 2007: Schaeffer's program solves checkers (a draw)
- Two versions of checkers:
    - Freestyle: whoever goes first can make any move
    - 3-move: first move must be one of three moves (three opening moves chosen to make game fairly even)
        - Schaeffer's group figured out for these three moves who will win with optimal play
- Key idea to solving checkers: endgame databases (dynamic programming on steroids)
    - "For this position of my last king and your last king, who will win" (2 vs. 1, 3 vs. 1, 3 vs. 2, etc.)
- *"Using retrograde analysis, subsets of a game with a small number of pieces on the board can be exhaustively enumerated to compute which positions are wins, losses or draws. Whenever a search reaches a database position, instead of using an inexact heuristic evaluation function, the exact value of the position can be retrieved."*

Nim Formula:

Theorem: a nim position is losing if an only if the xorsum of the pile sizes is 0

e.g. xorsum(1 2 3) = 01 ^ 10 ^ 11 = 00, so (1 2 3) is losing
e.g. xorsum(3 5 3) = 011 ^ 101 ^ 111 = 001, so (3 5 7) is winning

How to find winning move? From (3 5 7) can move to:

- (0 5 7), xorsum 010, not losing position, so this is not a winning move
- (1 5 7), xorsum 011, not losing position, so this is not a winning move
- (2 5 7), xorsum 000, losing position, so yes, winning move

What other winning moves are there from (3 5 7) (do as exercise)?

Proof:

show for each state P with xorsum 0, each move is to state with xorsum nonzero

- each move is from pile j with p_j > 0
- let c_j be column of leftmost 1 in binrep p_j
- remove any number of stones from p_j changes xorsum of column c_j from 0 to 1
- so xorsum(new P) has c_j 1, so nonzero

show for each state P with xorsum nonzero, some move is to state with xorsum 0

- let c_j be column of leftmost 1 in binrep xorsum(P)
- let p_x be any pile with 1 in c_j
- z = xorsum(P ^ p_x) has 0 in all columns c_1 ... c_j, so z < p_x, so can reduce p_x to z, leaves state with xorsum zero

All winning moves (bignim.py):

```
def nimreport(P): # all nim winmoves from P, use formula
  total = xorsum(P)  # xor all elements of P
  if total==0:
    print(' loss')
    return
  for j in P:
    tj = total ^ j  # xor
    if j >= tj:
      print(' win: take',j - tj,'from pile with',j)

[ code: simple/nim/bignim.py ]
```

Nim formula example:

```
all winmoves from (1 2 6)

  1    0 0 1
  2    0 1 0
  6    1 1 0
xor+   1 0 1   <-- nonzero, so exists winmove

1-pile winmove?
  after move, need new total xorsum 0, so
  need to remove k = (xorsum of all other piles) stones
  (xorsum of all other piles) = 110 ^ 010 = 100 = 4
  remove 4 stones from 1-pile?  4 > 1  *no*

2-pile winmove?
  remove 001 ^ 110 = 7 stones?  7 > 2  *no*

6-pile winmove?
  remove 001 ^ 010 = 3 stones?  6 > 3, *yes*

after remove 3 from 6-pile:
  1    0 0 1
  2    0 1 0
  3    0 1 1
sum    0 0 0  <--- losing state, as desired

- exercise: find all winmoves from (3 5 6 7)
```

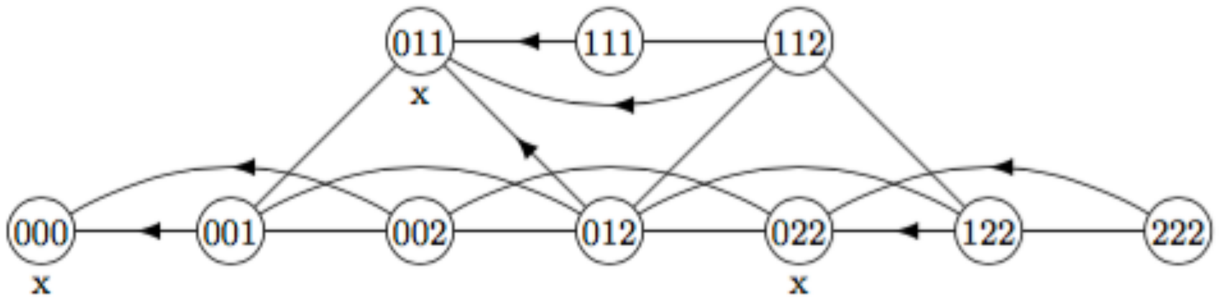
Combinatorial Game Theory:

Nim is one of the games whose math gave rise to combinatorial game theory. A combinatorial game is a game where **whoever cannot move loses (**variant: whoever cannot move wins). The math of combinatorial game theory is essential to solving Go endgame puzzles.

Some CGT games include:

- Amazons
- Breakthrough
- Clobber
- Dots and boxes

Solution to Nim DAG:



Solution to exercise on last page:

```
  3   0 1 1
  5   1 0 1
  6   1 1 0
  7   1 1 1
sum   1 1 1
      *      piles 5,6,7 have 1 in this column

winning: remove 3 from 5-pile, leaves
  3   0 1 1
  2   0 1 0
  6   1 1 0
  7   1 1 1
sum   0 0 0         or

         remove 5 from 6-pile, leaves
  3   0 1 1
  5   1 0 1
  1   0 0 1
  7   1 1 1
sum   0 0 0         or

         remove all from 7-pile, leaves
  3   0 1 1
  5   1 0 1
  6   1 1 0
  0   0 0 0
sum   0 0 0
```