Nim, part 2

CMPUT 355: Games, Puzzles, and Algorithms

Lecture Outline

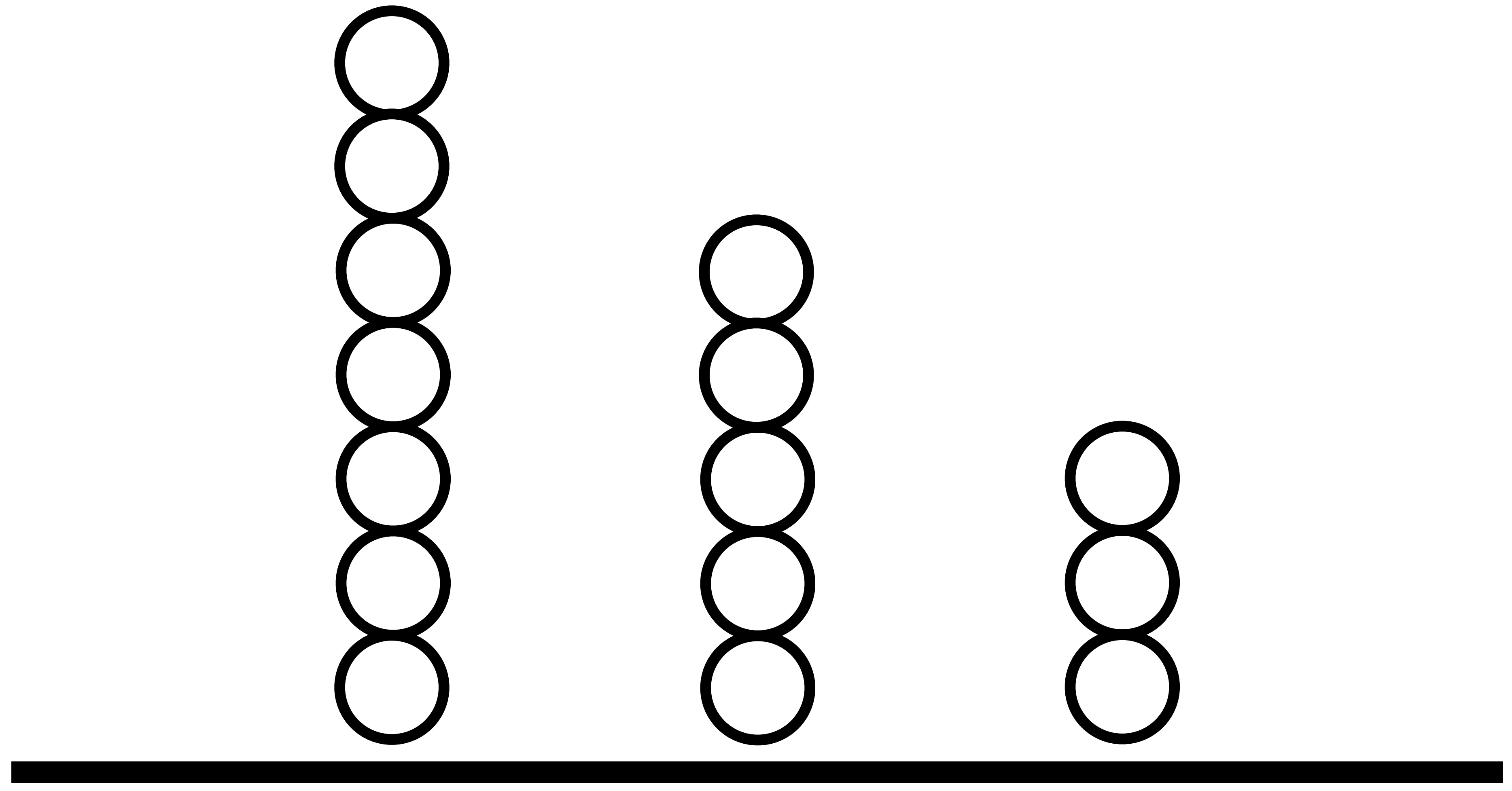
1. Logistics & Recap
2. Dynamic programming
3. Nim Formula & Theorem

Logistics

- **Quiz 2** marks are **released** on Canvas
    - A scan of your quiz is attached to the Canvas assignment
- **Practice questions #3** are available
    - Solutions released last Friday
- **Quiz 3** is **this Friday** (Feb 27)
    - *Coverage:* up to and including **Feb 13** (Move Ordering & Nim)
    - Bring your student ID!
    - No calculators or other devices
- **TA Office hours:** every Thursday 1pm-2pm in **UCOMM-3-136**

Recap: Nim

- There are k piles of stones
- A move is removing some **positive** number of stones from a **single pile**
- Players alternate moves
- Last player to move wins
  - i.e., if there are **no stones left** and you are player-to-move, you lose
- Example at right is position $7{,}5{,}3$
- Every position is either **winning** (i.e., player-to-move can force a win from here) or **losing** (every child position is a winning position for the opponent)



Dynamic Programming

- **Dynamic programming:** Solve a problem by first solving **overlapping subproblems**
  - **overlapping**: used more than once
  - so it's useful to store subproblem solutions rather than re-solving them
- **Memoization** (caching) is an easy way to speed up a recursive function with overlapping subproblems

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n - 1)
```

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

```
def fib(n, D):
  if n in D:
    return D[n]
  elif n <= 1:
      return 1
  else:
    f = fib(n-1, D) + fib(n-2, D)
    D[n] = f
    return f
```

Questions:

1. Which of these two functions exhibit overlapping subproblems?
2. How long will **factorial(N)** take?
3. How long will **fib(N)** take?
4. How would each change if we **cached** the results?

Dynamic Programming in Nim

- Determining whether a Nim position is **winning** or **losing** is a dynamic programming problem
- We have **already used memoization** to solve this problem (**how?**)
- The recursion-with-memoization approach to dynamic programming has some drawbacks (**what?**)
- **Bottom-up approach:**
  Solve the subproblems in a specific order such that you have already solved subproblem A,B,C,... before any other subproblems that depend on those solutions
- **Question:** What would that look like in the context of Nim?

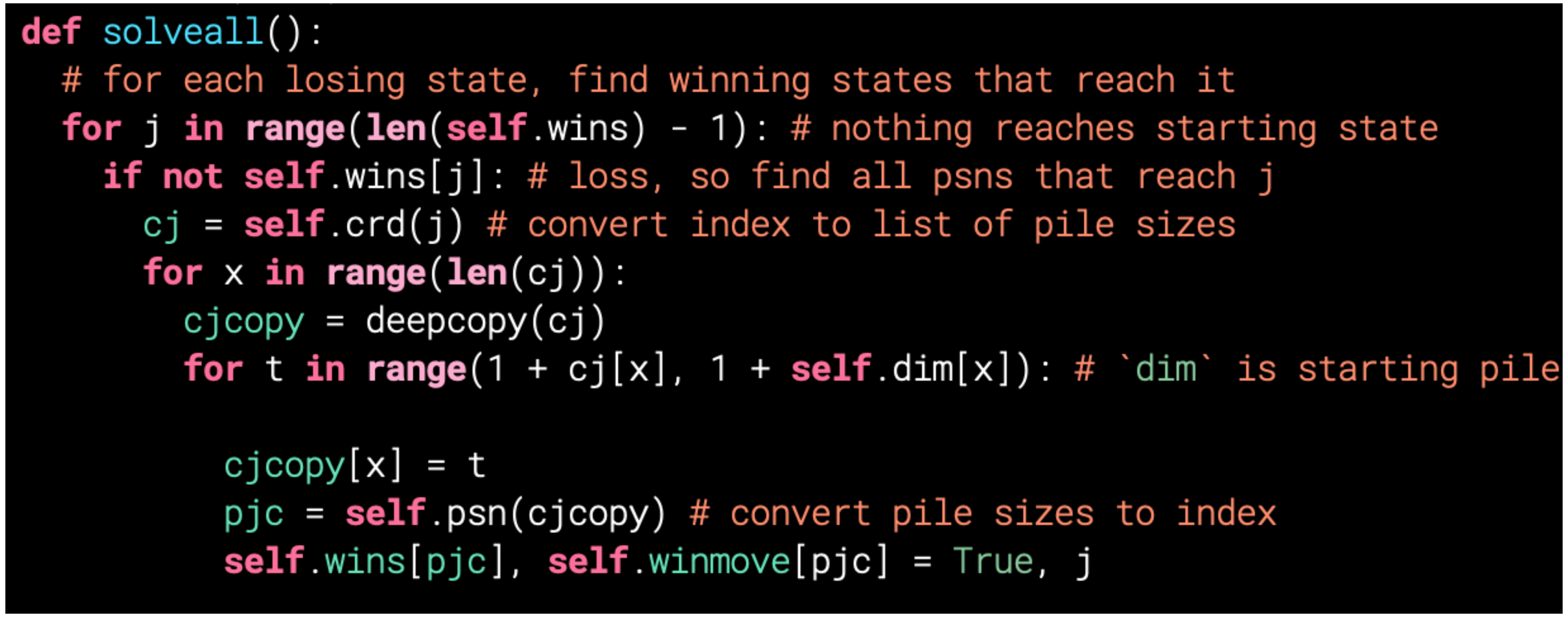
Questions:

1. What are the **subproblems** in Nim?
2. Why are they **overlapping**?
3. Is the win/lose/draw computation in **Tic-tac-toe** a dynamic programming problem?
   Why or why not?

Canonical Representations

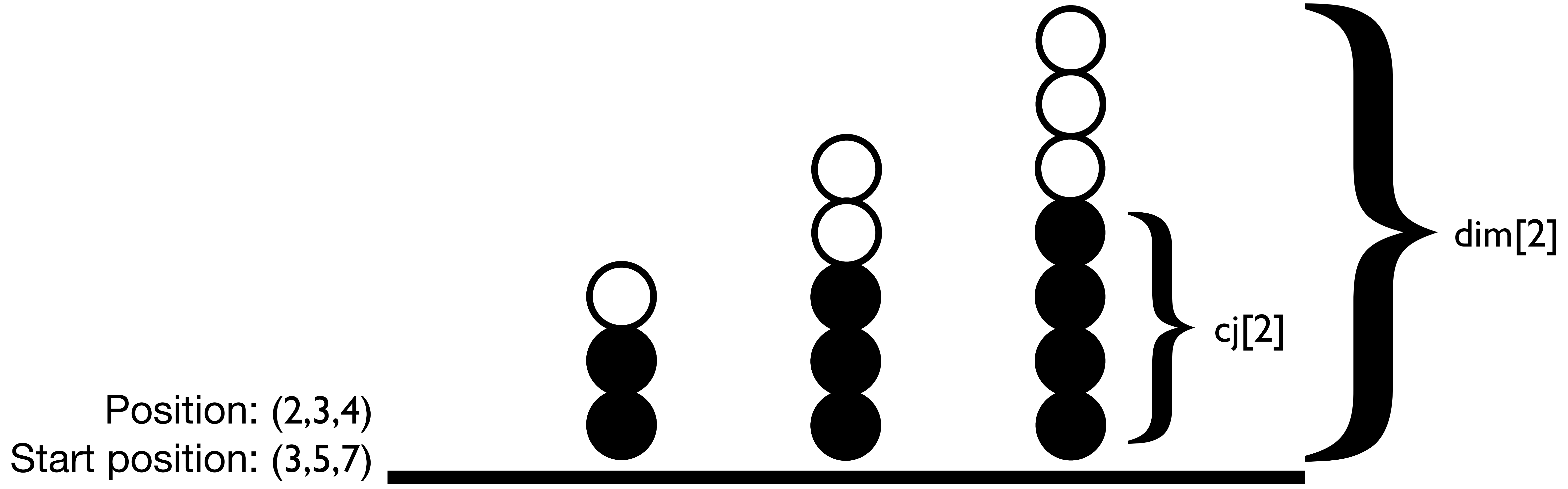
- Recall that we can convert any Nim position into a **canonical representation** just by **sorting** the piles in order of size
  - E.g.,(7,3,4) --> (3,4,7)
- Just like in tic-tac-toe, we can then assign a **number** to each representative
  - E.g.1: (3,4,7) --> 347
  - E.g.2: (3,4,7) --> $3 + 4 \times (3+1) + 7 \times \left((3+1) \times (5+1)\right) = 120$
    - (assumes starting position of (3,5,7)) (**why?**)
- **Claim:** every child position will have a **smaller number** than its parent (**why?**)
- So if we compute subproblems in **increasing numeric order**, then every subproblem will be computed **before we need it**

Dynamic Programming Implementation: **nim/nim.py**



```python
def solveall():
  # for each losing state, find winning states that reach it
  for j in range(len(self.wins) - 1): # nothing reaches starting state
    if not self.wins[j]: # loss, so find all psns that reach j
      cj = self.crd(j) # convert index to list of pile sizes
      for x in range(len(cj)):
        cjcopy = deepcopy(cj)
        for t in range(1 + cj[x], 1 + self.dim[x]): # `dim` is starting pile

          cjcopy[x] = t
          pjc = self.psn(cjcopy) # convert pile sizes to index
          self.wins[pjc], self.winmove[pjc] = True, j
```



XorSum

- **xorsum** operation: Cumulative bitwise XOR of **each binary digit** of the pile sizes
- Examples:
  - xorsum(1,2,3) = 00
    - $\begin{array}{l} 1 = 01 \\ 2 = 10 \\ 3 = 11 \end{array}$ and $\begin{array}{l} 0 \oplus 1 \oplus 1 = 0 \\ 1 \oplus 0 \oplus 1 = 0 \end{array}$
  - xorsum(3,5,3) = 101b = 5
    - $\begin{array}{l} 3 = 011 \\ 5 = 101 \\ 3 = 011 \end{array}$ and $\begin{array}{l} 0 \oplus 1 \oplus 0 = 1 \\ 1 \oplus 0 \oplus 1 = 0 \\ 1 \oplus 1 \oplus 1 = 1 \end{array}$

Nim Formula Theorem

Theorem: A Nim position with pile sizes $p_1, \ldots, p_k$ is **losing** iff

$$\text{xorsum}(p_1, \ldots, p_k) = 0.$$

Proof sketch:

1. If $\text{xorsum}(\text{P}) = 0$, then $\text{xorsum}(C) \neq 0$ for **every** child C of P
2. If $\text{xorsum}(P) \neq 0$, then $\text{xorsum}(C) = 0$ for **some** child C of P
3. $\text{xorsum}(0,0,\ldots,0) = 0$
4. Any position that has $(0,0,\ldots,0)$ as a child is a **winning** position

Nim Formula Theorem Proof: Step 1

1. If $\mathrm{xorsum}(\mathrm{P}) = 0$, then $\mathrm{xorsum}(C) \neq 0$ for every child C of P

(a) Consider any pile j with $p_j > 0$

(b) Then $\mathrm{xorsum}(p_1, \ldots, p_{j-1}, p_{j+1}, \ldots, p_k) = p_j$ (**why?**)

(c) So if you remove any stones from pile j, $p_j' \neq p_j$, and so $p_j \oplus p_j' \neq 0$

(d) $\mathrm{xorsum}(p_1, \ldots, p_{j-1}, p_j', p_{j+1}, \ldots, p_k) = \mathrm{xorsum}(p_1, \ldots, p_{j-1}, p_{j+1}, \ldots, p_k) \oplus p_j' = p_j \oplus p_j' \neq 0$

(e) But every child of $(p_1, \ldots, p_k)$ is made by removing some stones from a single pile j

$\blacksquare$

Nim Formula Theorem Proof Step 2

2. If $\operatorname{xorsum}(P) \neq 0$, then $\operatorname{xorsum}(C) = 0$ for **some** child C of P

(a) Let d be the digit number of the most significant 1 in the binary representation of $\operatorname{xorsum}(P)$

(b) There exists at least one j such that p_j has a 1 in digit d (**why?**)

(c) $\operatorname{xorsum}(p_1, \ldots, p_{j-1}, p_{j+1}, \ldots, p_k)$ has a 0 in the d-th column (**why?**)

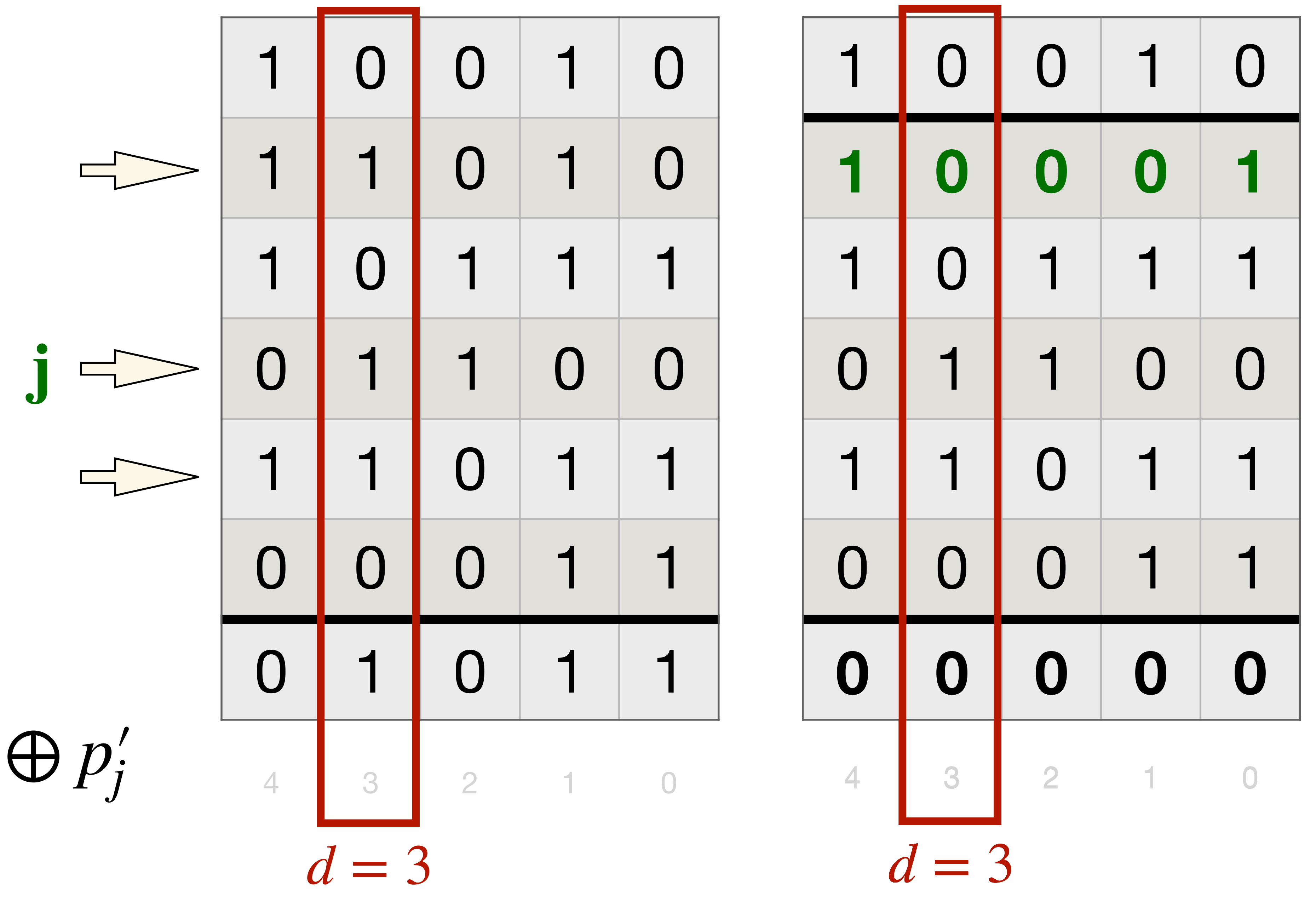
- and *also* in every more-significant column that p_j has a 0 in (**why?**)

(d) So $\operatorname{xorsum}(p_1, \ldots, p_{j-1}, p_{j+1}, \ldots, p_k) < p_j$ (**why?**)

(e) Set $p_j' = \operatorname{xorsum}(p_1, \ldots, p_{j-1}, p_{j+1}, \ldots, p_k)$

(f)
$$\begin{aligned}\operatorname{xorsum}(p_1, \ldots, p_{j-1}, \mathbf{p_j'}, p_{j+1}, \ldots, p_k) &= \operatorname{xorsum}(p_1, \ldots, p_{j-1}, p_{j+1}, \ldots, p_k) \oplus p_j' \\ &= p_j' \oplus p_j' \\ &= 0\end{aligned}$$

■



4	3	2	1	0
1	0	0	1	0
1	1	0	1	0
1	0	1	1	1
0	1	1	0	0
1	1	0	1	1
0	0	0	1	1
0	1	0	1	1

4	3	2	1	0
1	0	0	1	0
1	**0**	**0**	**0**	**1**
1	0	1	1	1
0	1	1	0	0
1	1	0	1	1
0	0	0	1	1
0	**0**	**0**	**0**	**0**

Nim Formula Theorem Proof Step 5

1. If $\text{xorsum}(\text{P}) = 0$, then $\text{xorsum}(C) \neq 0$ for **every** child C of P
2. If $\text{xorsum}(P) \neq 0$, then $\text{xorsum}(C) = 0$ for **some** child C of P
3. $\text{xorsum}(0,0,\ldots,0) = 0$
4. Any position that has $(0,0,\ldots,0)$ as a child is a **winning** position

Inductive hypothesis: suppose that for all positions P within k steps of $(0,\ldots,0)$,
$(\text{xorsum}(P) \neq 0 \implies P$ is winning) and $(\text{xorsum}(P) = 0 \implies P$ is losing)

Then for all P that are within $k+1$ steps of $(0,\ldots,0)$:

(a) $\text{xorsum}(P) \neq 0$:

- by (2), $\text{xorsum}(C) = 0$ for **some** child C
- C is within k of $(0,\ldots,0)$, so by IH C is **losing**
- So P has **at least one losing child**, and is therefore **winning**

(b) $\text{xorsum}(P) = 0$:

- by (1), $\text{xorsum}(C) \neq 0$ for **all** children C
- every C is within k of $(0,\ldots,0)$, so by IH every C is **winning**
- So **all of P's children are winning**, hence P is **losing**

Base case:

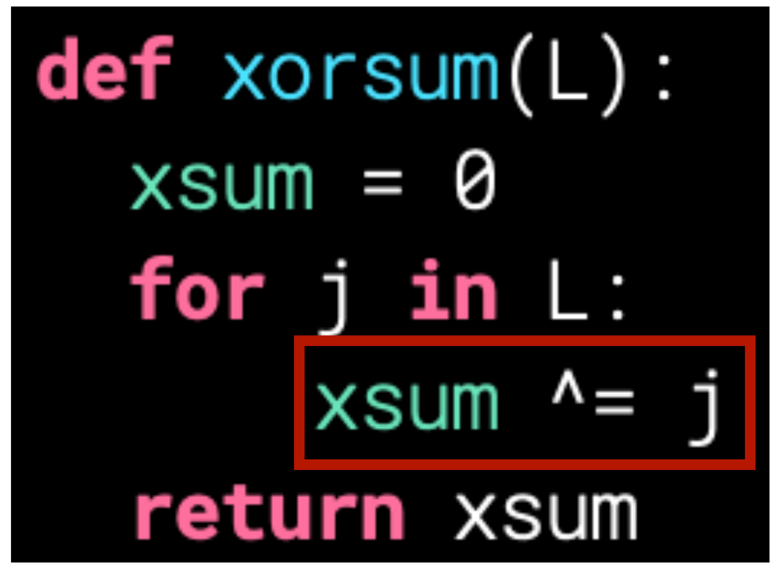
(a) By (3), every position P within 0 of $(0,\ldots,0)$ has $\text{xorsum}(C) = 0$, and by definition it is a losing position

(b) By (1), every position P within 1 of $(0,\ldots,0)$ has $\text{xorsum}(C) \neq 0$, and by (4) they are all winning ∎

Using the Nim Formula

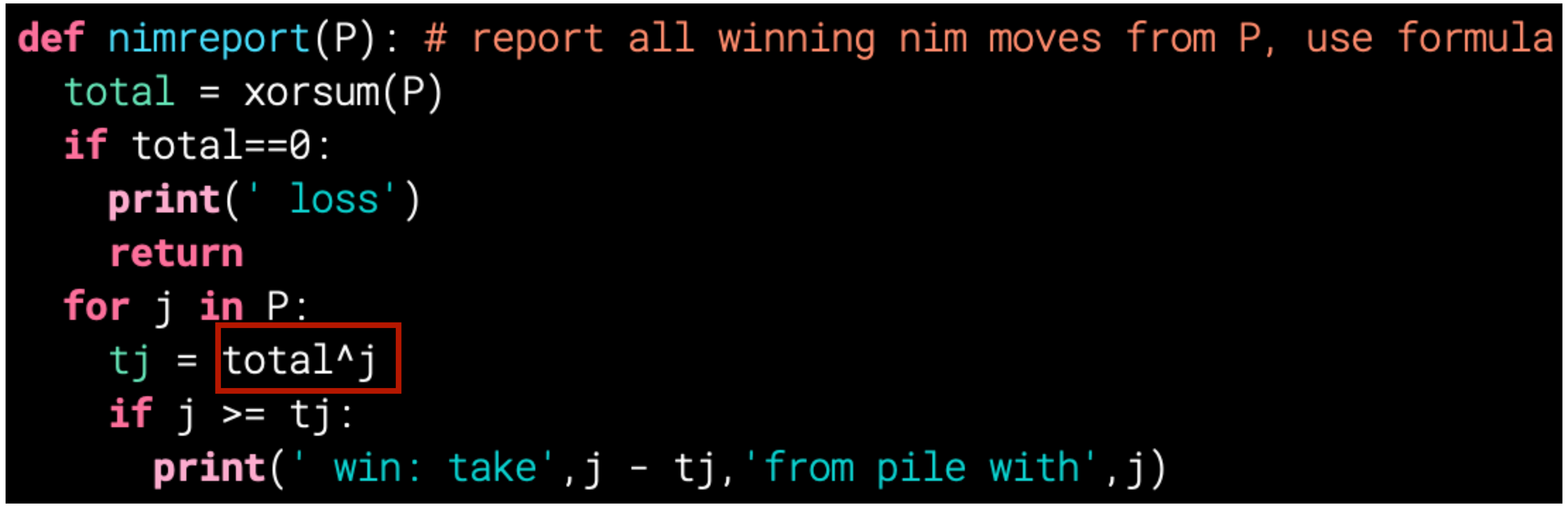
- **Checking** if a position P is **winning** or **losing**: Simply compare $\mathrm{xorsum}(P)$ to 0
- Finding a **winning move**:
  1. Compute $\mathrm{xorsum}(P)$
  2. Find d and a pile j with a 1 in digit d
     - Equivalently: a pile j with $p_j \geq \mathrm{xorsum}(p_1, \ldots, p_{j-1}, p_{j+1}, \ldots, p_k)$
  3. Set $p_j' = \mathrm{xorsum}(p_1, \ldots, p_{j-1}, p_{j+1}, \ldots, p_k)$
  4. Take $p_j - p_j'$ stones from pile j
     - i.e., move to position $(p_1, \ldots, p_{j-1}, \mathbf{p_j'}, p_{j+1}, \ldots, p_k)$

Nim Formula Implementation: **nim/nim.py**



```python
def xorsum(L):
  xsum = 0
  for j in L:
      xsum ^= j
  return xsum
```

(x ^ y) is the Python syntax for $(x \oplus y)$



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

Note: $\text{xorsum}(p_1, \ldots, p_{j-1}, p_{j+1}, \ldots, p_k) = \text{xorsum}(p_1, \ldots, p_k) \oplus p_j$ **(why?)**

Summary

- Nim can be solved using **dynamic programming**
    - recursion with memoization (i.e., recursion with transposition table)
    - bottom-up
- The **Nim formula** allows us to **directly compute** whether a Nim move is **winning**
    - *Without* having to solve all the subpositions!
- Can directly compute a **winning move** from any winning position
    - Again, without having to solve subpositions
    - Or even consider every possible child