University of Illinois at Urbana-Champaign
Dept. of Electrical and Computer Engineering

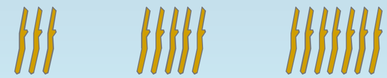
ECE 220: Computer Systems & Programming

More Recursion Examples

Ready to Play a Game?

Let's play Nim!

In Nim, there are three piles of sticks…



On their turn, each player
- takes as many sticks as they want
- from one of the piles.

The last player to take sticks wins.

Is Nim a Forced Win or a Forced Loss

Nim starts with 3, 5, and 7 sticks in the piles.

There is no way to tie.

A **forced win** means that,
- if a player plays correctly,
- they are **guaranteed to win**.

Is Nim
- **a forced win** (for the first player),
- **or a forced loss?**

Let's Use Recursion to Evaluate Nim

There's a fairly easy and intuitive mathematical solution to Nim.

But … maybe you don't know it?

Fortunately, now you know recursion.

So **let's**
- **write a recursive function**
- to answer the question!

Let's Use Recursion to Evaluate Nim

Here's how our function works:
- **given the number of sticks**
- in each of the three piles,
- the function **nim returns**
- the **value of the game**.

Since Nim is a zero-sum game,
- **1** can represent the **first player winning**,
- and **-1** can represent the **second player**.

Value is the Maximum of Negated Recursive Evaluations

In **nim**,
- the **current player makes one move**
- then **calls nim to evaluate** the new piles.

Since Nim is a zero-sum game,
- the **value returned** by the recursive call
- **is simply negated**:
- the value to one player
- is negative the value to the other player.

The **value of the game is the maximum value over all possible moves** (the best move).

Piles are Empty? Current Player Has Lost

```
int32_t nim (int32_t p[3])
{
    int32_t max = -2;
    int32_t pnum;
    int32_t count;
    int32_t value;
    if (0 == p[0] && 0 == p[1] &&
        0 == p[2]) {
        return -1;
    }
```

max move value seen

pile for a move

sticks for a move

value for a move

stopping condition: piles are empty

Try Every Possible Move and Choose the Best

```
    for (pnum = 0; 3 > pnum; pnum++) {
        for (count = 1;
             p[pnum] >= count;
             count++) {
            // Try one move
            // and update max.
        }
    }
    return max;
}
```

Try each valid number of sticks.

Try moves for each pile.

Return best move.

Make One Move, Evaluate, and Update

```
p[pnum] -= count;
value = -nim (p);
p[pnum] += count;
if (max < value) {
    max = value;
}
```

Modify the array in place (rather than creating a copy in our stack frame).

Recurse.

Restore the original array value.

If this move's value is better than any previous move, record it.

`nim` is On the Web Page

The code is on the web page.

A Few Other Applications of Recursion

Other applications of recursion include…
- puzzles, such as Sudoku,
- code generation, and
- code optimization.

Generally, recursion is useful for wide searches (many children).

Deep searches (many levels) tend to break the stack.

Time for Another Think-Pair-Share

As before, let's do a group exercise in lecture.

The process:
1. I give you a problem.
2. You form groups of 3-4 people.
3. Talk about ways to solve the problem.
4. Once enough of the groups have finished, one group volunteers to share their answer.
5. We go over the group's answer together.

The Task: Check a Maze for Cycles

The task: check for cycles in a maze
- using our earlier maze representation:

```
static uint8_t maze[10][10]; // maze
// L = 1, R = 2, U = 4, D = 8
```

- Is a cycle reachable from starting point?
- You define the function signature.
- You probably should use (initialized to 0):

```
static uint8_t found[10][10];
```


Review: How the Bit Vector Representation Works

We can **represent the maze with an array**:

```
static uint8_t maze[10][10];
```

Each space in the array **is a bit vector** composed of the following bits:
- // **1** – the space has a **left wall**
- // **2** – the space has a **right wall**
- // **4** – the space has an **upper wall**
- // **8** – the space has a **lower wall**
- // **16** – the space is the **exit**

Tasks on Your Own

More things to try on your own:
1. Remove all cycles (by adding walls as necessary).
2. Make all parts of the maze connected (by removing walls as necessary).

