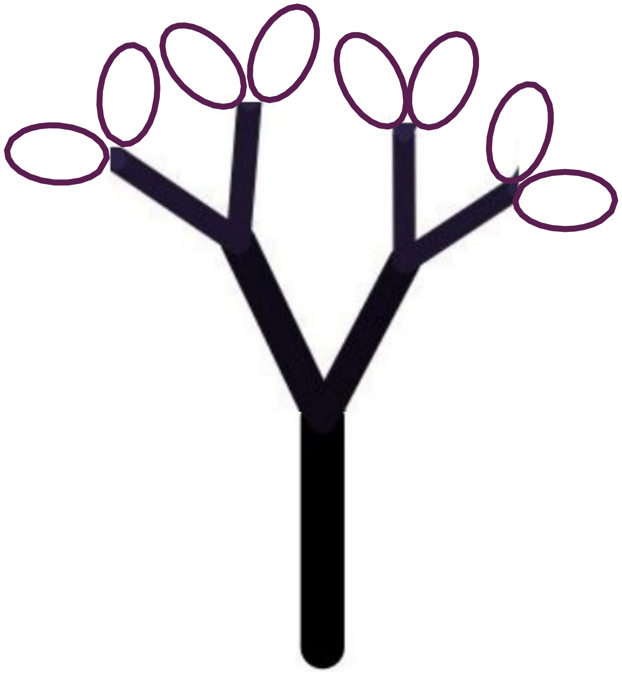
Programming with Data Structures

CS 241-03

OHLONE
COLLEGE

Data Structures



Why is recursion such a powerful problem-solving tool?

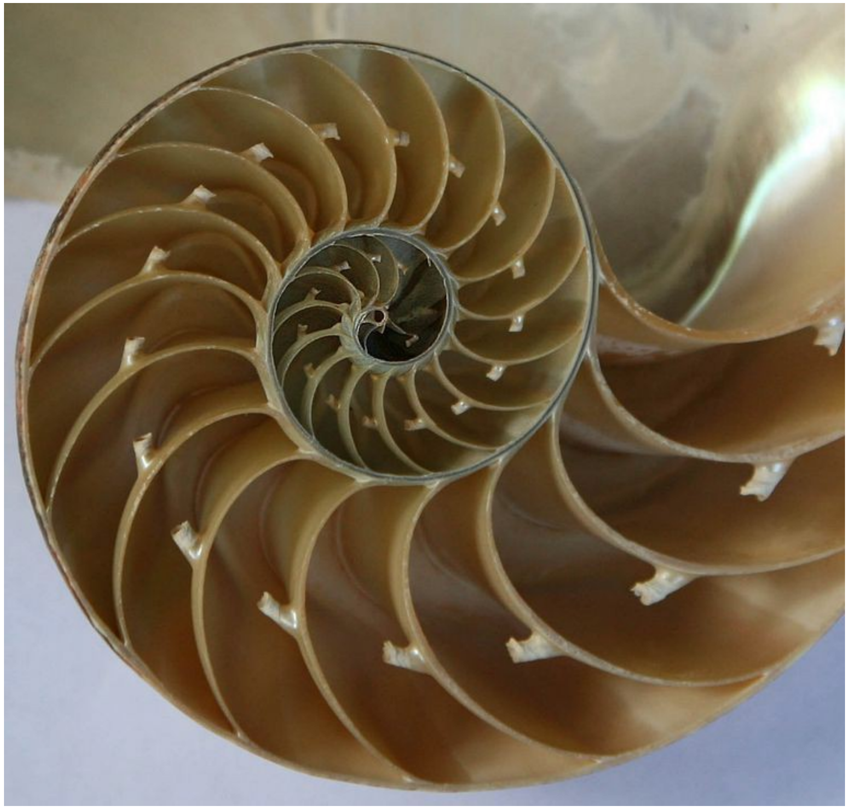
Geri Lamble

CS 124-03

Self-Similarity

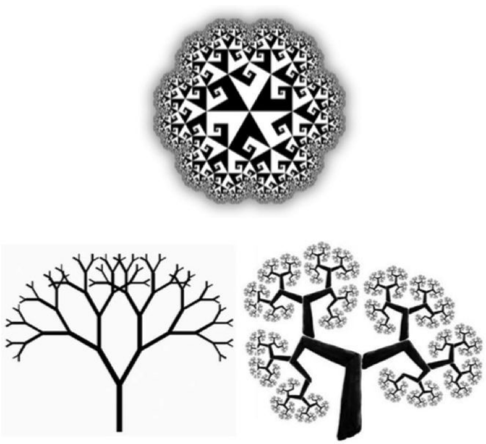
- Solving problems recursively and analyzing recursive phenomena involves identifying **self-similarity**

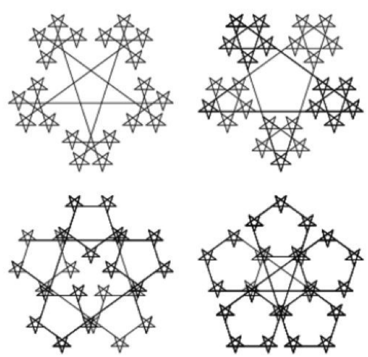
- An object is **self-similar** if it contains a smaller copy of itself.

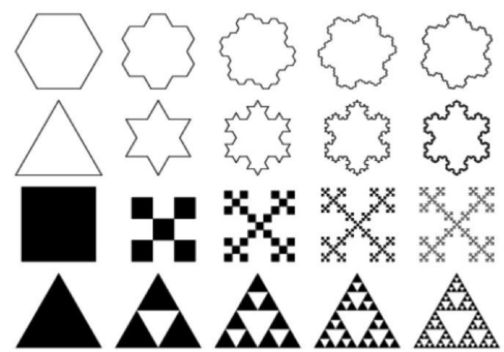


Fractals

- A **fractal** is any repeated, graphical pattern.
- A fractal is composed of **repeated instances of the same shape or pattern**, arranged in a structured way.







Why do we use recursion?

Why do we use recursion?

- Elegance
  - Allows us to solve problems with very clean and concise code
- Efficiency
  - Allows us to accomplish better runtimes when solving problems
- Dynamic
  - Allows us to solve problems that are hard to solve iteratively

Problem Solving with Recursion

THINKING RECURSIVELY

Recursive Decomposition

- Recursion works well when:
  - Problem can be written in terms of smaller sub-problems
  - Sub-problems have the *same structure* as original
  - Solving all sub-problems solves original problem
- Examples (from previous lectures)
  - Factorial as product of number and factorial of smaller number
  - Palindrome test written as: "check outer two characters, then test for smaller palindrome"

Recursion as Induction

The basic form of recursion follows that of induction:

- Recursive base case(s) == inductive base case(s)
  - If we apply our function to problem of size 1, then we get the correct answer
  - E.g., if a string is size 1 or 0, then it is a palindrome
- Recursive step == inductive step
  - If we are correct on problem of size n, then we are correct on a problem of size n + 1
  - Palindromes are a bit tricky here, because we actually prove 2 cases, one for odd numbers and one for evens:
    - If our program works for strings of n letters, then prove it works for strings of n + 2 letters

Example: The Towers of Hanoi

Object: move all disks from left spindle to middle spindle.

Rules:

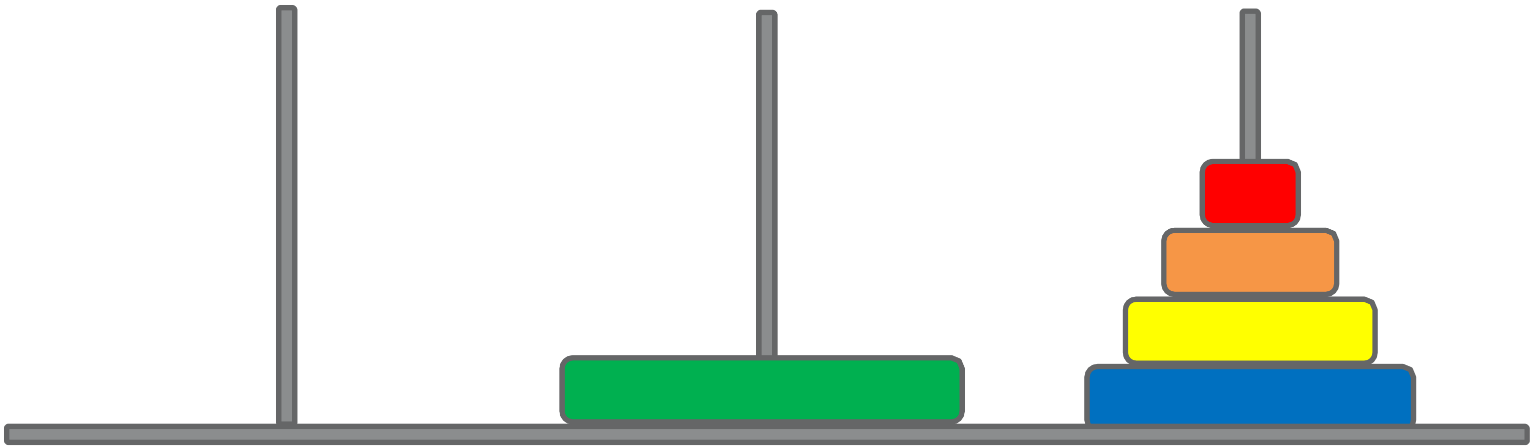
1. Move only 1 disk at a time.
2. A disk can only be moved onto an empty spindle or a larger disk.

Ways of Thinking About Recursion

Top down or **bottom up**:

- **Top down:** think of the whole problem and how it can be decomposed.
  - If I could solve a smaller problem *of the same form*, could I solve the whole problem?
- **Bottom up:** what is the smallest problem I *can* solve?
  - If I can solve that, would it help me solve larger problems?

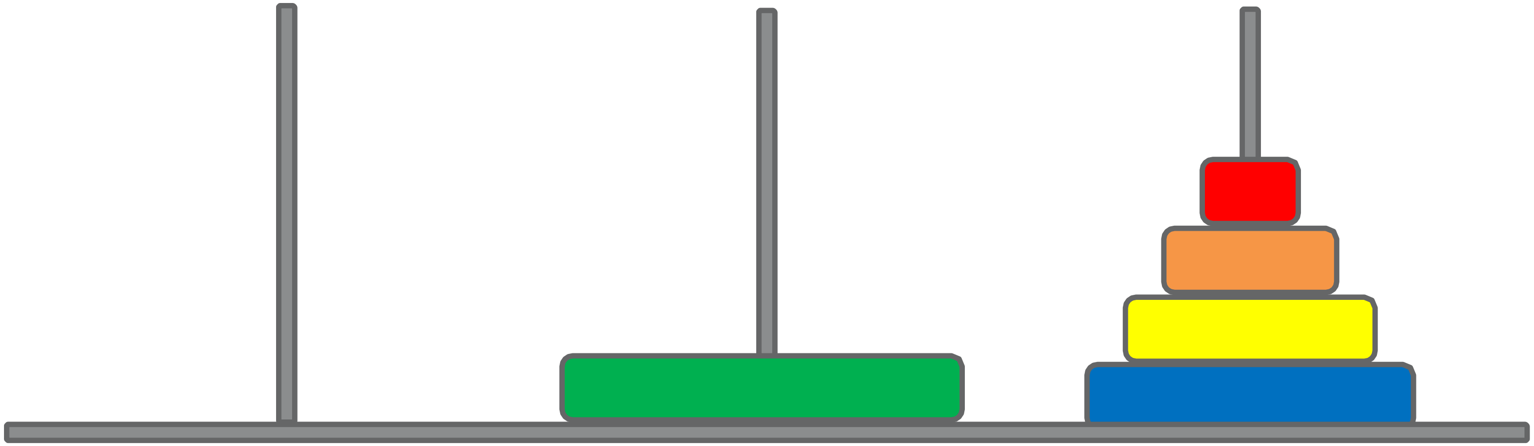
Top Down



Consider this arrangement:

- If we could move the right stack onto the green disk, we'd be done.
- This is a *smaller problem* of the original form:
  - The stack to be moved has one fewer than the original.
  - Although the middle spindle is occupied, every disk to be moved is smaller than the green disk – so no problem!

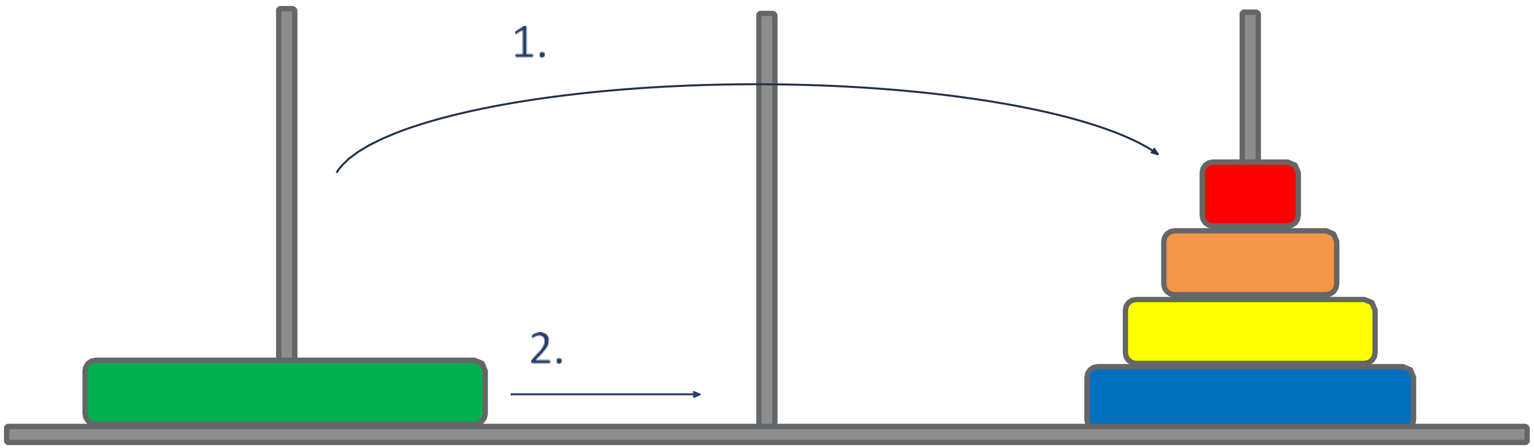
Top Down



Does this help us?

If we can get to this partial solution, it does…

Top Down



Well, we can, in 2 steps:

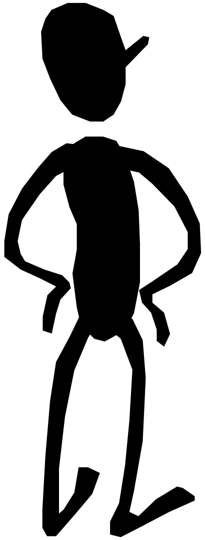
1. Move the smaller stack to the right spindle, leaving the green disk in place (hey, that's *another* instance of the same problem!)
2. Move the green disk to the center. Moving just 1 disk is easy – that must be the base case!

Pseudo-code

```
// count is # to move
// a is "from" tower
// b is "to" tower
// c is the spare tower
moveTower (count, a, b, c):
  if count = 1 then
    just move it!
  else
    moveTower(count – 1, a, c, b)
    moveTower(1, a, b, c)
    moveTower(count – 1, c, b, a)
```

See it in Action

[Towers of Hanoi in Action]

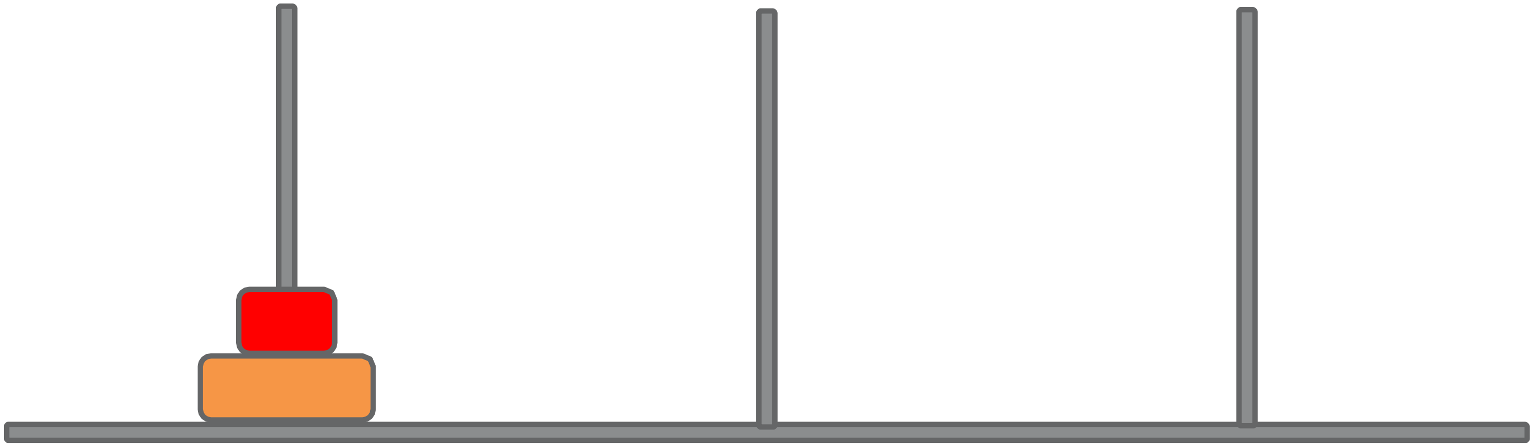


(You can find many others online, I just liked this one.)

Bottom Up

OK, moving 1 disk is easy.
Can I move 2 disks?

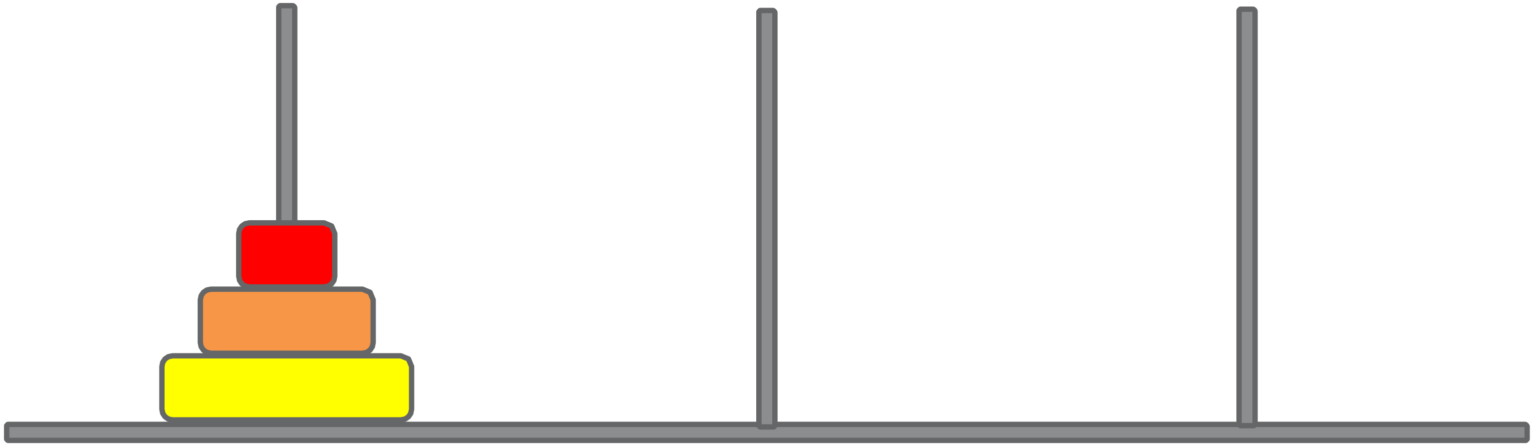
Bottom Up



Yes, in three moves:

1. Move the top disk to the third spindle.
2. Move the bottom disk to the middle (final) spindle.
3. Move the top disk to the middle spindle.

Bottom Up



If I have *3* disks, I need to:

1. Move the top *2* to the rightmost spindle
2. Move the bottom disk
3. Move the top 2 back

Generalize to n disks!

Example: Permutations

- Problem: find all permutations of an ordered set
  - E.g., what are all permutations of (a, b, c)?
    - Answer: (a,b,c), (a,c,b),(b,a,c),(b,c,a),(c,a,b),(c,b,a)
  - What about (a,b,c,d,e,f,g,h,…)?
    - Ugh. Let the computer do it.
    - OK… how?

Trying everything

BACKTRACKING

Maze Solving

Consider solving a maze:

- Assume potential loops, so right-hand rule fails
- Instead, have string and a marker
  - Mark where you've been, so you don't loop
  - Unroll string behind you so you can back up
  - Pick a passage, follow as far as you can until dead-ending or repeating yourself
  - Back-up to the last branching and try one you haven't tried (or back up further if no choices left)

Backtracking

- The maze solving algorithm above is an example of *backtracking*
- Essentially, try every possibility in a branching problem, avoiding repeats
- This sort of has the recursive sub-structure:
  - The problem is only made smaller by a little bit
  - We have to remember choices (or do we?)

Maze Solving Pseudocode

```
solve_2d_maze(maze, x,  y):
    if at exit, yay!
    else:
        mark maze[x][y] as visited
        if can go right (and right not visited):
            solve_2d_maze(maze, x+1, y)
        if can go down (and down not visited):
            solve_2d_maze(maze, x, y+1)
        etc.
```

Winning!

MINIMAX

Backtracking for Games

- For 2-player perfect information games
- Like trying every possibility, but:
  - Assume each player is trying to win😊
  - Each player has a different goal, so have to switch objective between moves
- Classic algorithm is called *minimax*

Example: Nim

- The game:
  - Put *n* tokens on the table
  - Each player gets to take 1, 2, or 3 tokens each turn
  - Player who takes the last token loses
- Work backwards from base case:
  - If 1 token left for other player, you win
  - Thus, if 2-4 tokens left for you, you can force win
  - However, if 5 tokens left for you, you lose, because any move you make leaves a good move for opponent…

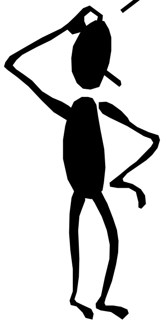
Solving Nim Recursively

```
find_good_move(ntokens):
  for j = 1 to min(3, ntokens):        // try possible moves
    // try for win in one move
    if ntokens – j == 1:               // base case: WIN 😊
      return j

    // next, see if opponent must lose if I make this move
    if find_good_move(ntokens – j) == NO_GOOD:       // WIN 😊
      return j

    // I tried everything, no luck: I must lose
    return NO_GOOD              // base case: LOSE ☹
```

Questions?



DATA STRUCTURE