

# Recursion II


CSE 120 Spring 2017

**Instructor:**

Justin Hsia

**Teaching Assistants:**

Anupam Gupta, Braydon Hall, Eugene Oh, Savanna Yee

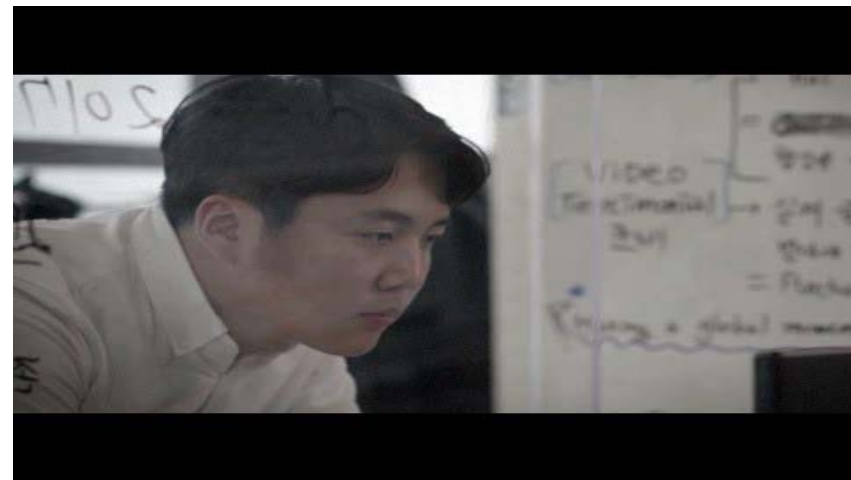
's' → 0x73 → 

## Braille Smartwatch Lets Users Feel Time, Texts And GPS Directions

Facebook, texting and GPS are commonplace tech that most of us use. But these apps... can be incredibly alienating for the visually impaired and blind.

The Dot smartwatch, which one could connect to a smartphone via Bluetooth, has a touch display where rising and falling dots spell out words in Braille.

Other smartwatches... [utilize] sound to read information on the screen through a speaker, but this method often robs disabled users of privacy.



- [http://www.huffingtonpost.com/entry/braille-smartwatch-feel-messages-screen\\_us\\_58b489d1e4b060480e0aeceb](http://www.huffingtonpost.com/entry/braille-smartwatch-feel-messages-screen_us_58b489d1e4b060480e0aeceb)

# Administrivia

## ❖ Assignments:

- Mid-Quarter Survey due tonight (5/3)
- Recursive Tree due Thursday (5/4)
- Color Checker due Saturday (5/6)
- Living Computers Museum Report (5/14)

## ❖ Guest lecture on Friday: Proofs and Computation

- Reading Check (5/4): mathematics

## ❖ Midterm re-grade requests due tonight (5/3)

- Adjusted scores will be uploaded to Canvas after regrade requests are handled

# Recursion Review

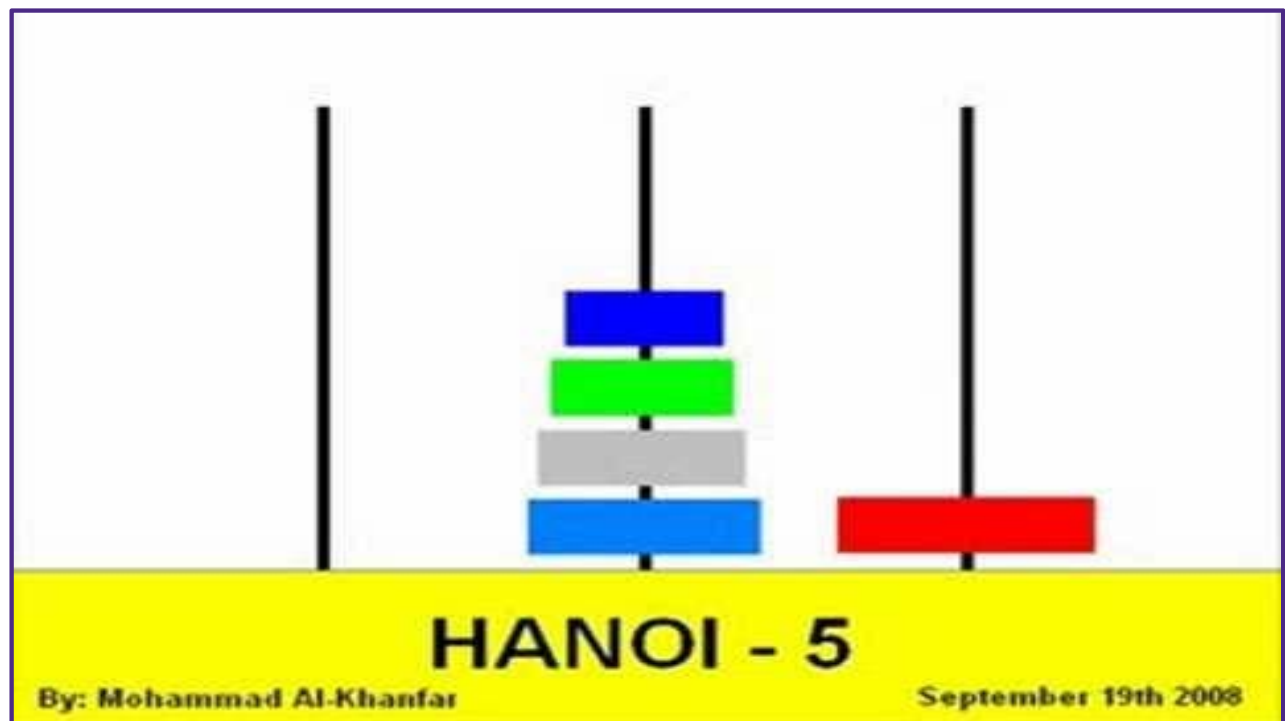
- ❖ A *recursive* function calls itself to solve its problem
- ❖ Base Case:
  - What happens for special/simple inputs
  - Need base case(s) to prevent infinite recursion
- ❖ Recursive Case:
  - Function calls itself one or more times on “smaller” problems
    - How to make the problem smaller varies ← this is the tricky part!

# Outline

- ❖ **Example: Tower of Hanoi**
- ❖ Variable Scope Revisited
- ❖ Example: Fibonacci
- ❖ Example: Snowflake Fractal

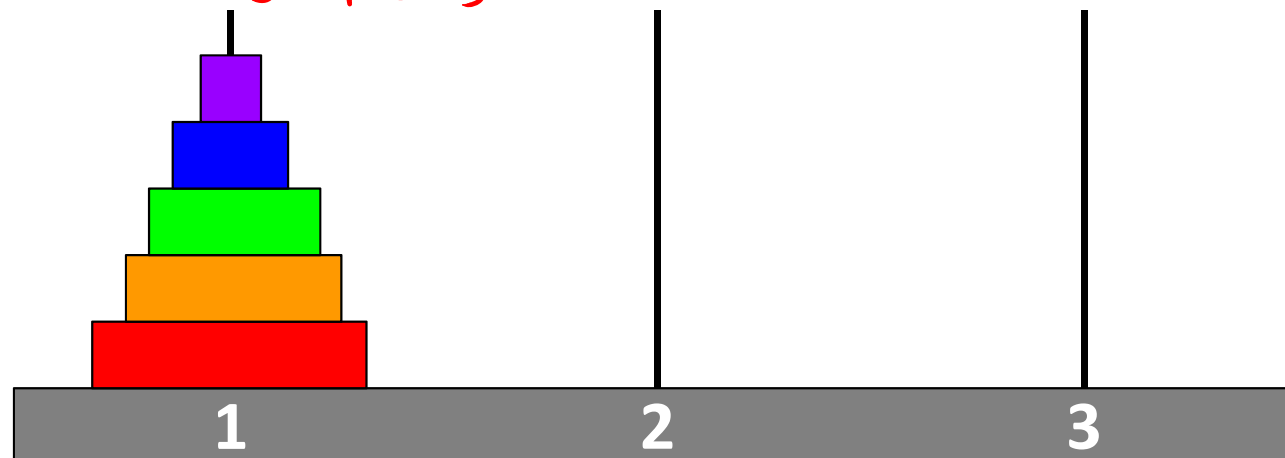
# Tower of Hanoi

- ❖ Mathematical puzzle/game
  - Goal is to move entire stack from one peg to any other peg
- ❖ Rules:
  - There are only 3 available pegs
  - Can only move one disk at a time
  - A disk cannot sit on top of a smaller one



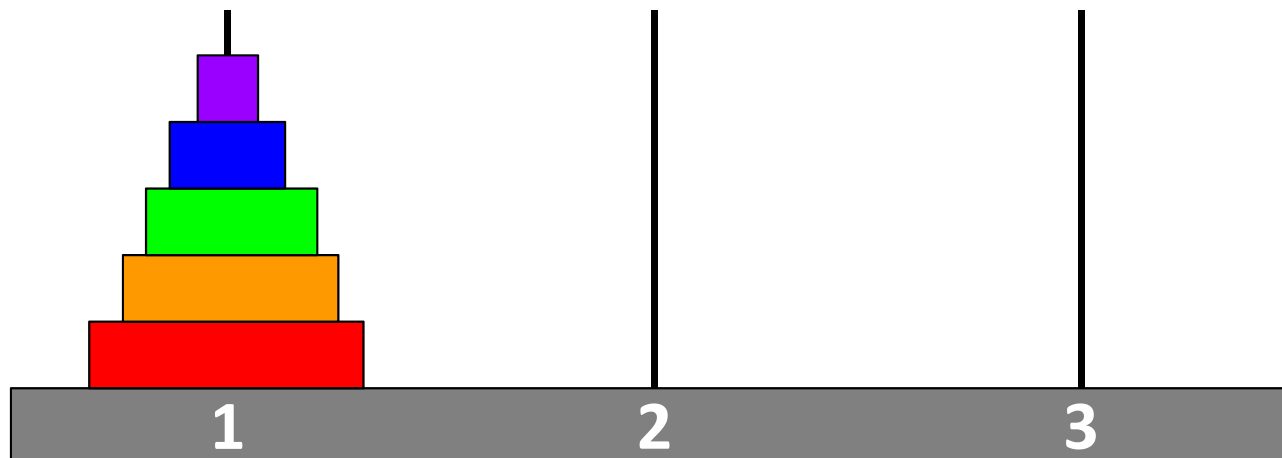
# Solving the Tower of Hanoi

- ❖ The animation was probably daunting, but the recursive solution is surprisingly clean
  - Can still be mind-blowingly confusing to understand
  - For illustrative purposes – you're not responsible for knowing this
- ❖ Goal: Move the tower of height 5 from peg 1 to peg 3
  - Let's assume our solution looks something of the form:  
`moveTower(int height, int startPeg, int endPeg)`  
`moveTower(5, peg1, peg3);` ← this should solve the problem



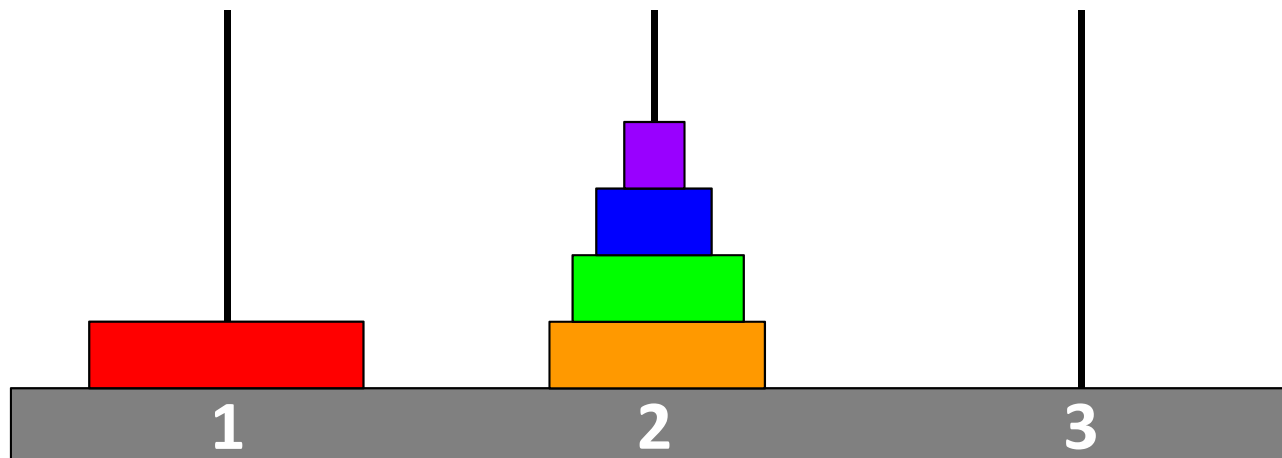
# Solving the Tower of Hanoi

- ❖ To reconstruct the tower on peg 3, we first need to get the largest disk (red) onto peg 3
  - Can't do this while the other disks are on top
  - Solution: First move the 4 disks on top to peg 2



# Solving the Tower of Hanoi

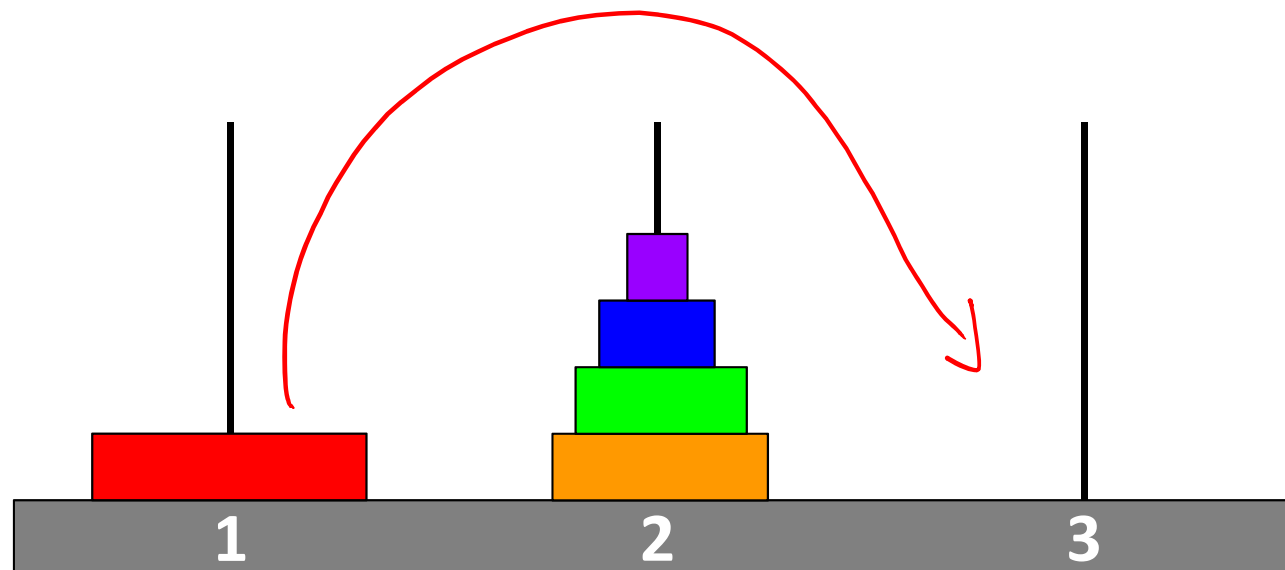
- ❖ To reconstruct the tower on peg 3, we first need to get the largest disk (red) onto peg 3
  - Can't do this while the other disks are on top
  - Solution: First move the 4 disks on top to peg 2
    - `moveTower(4, peg1, peg2)`; ← just assume it works!





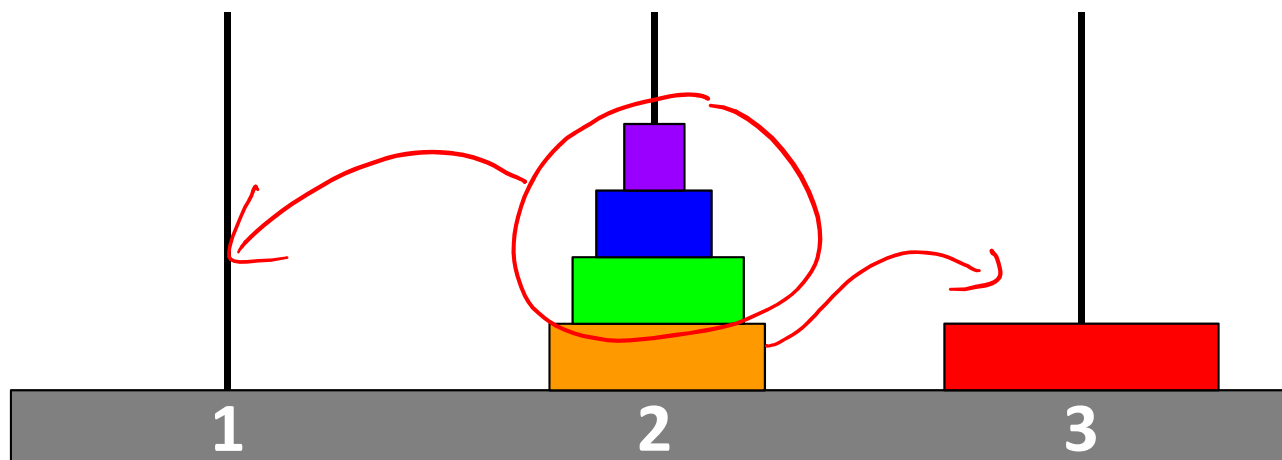
# Solving the Tower of Hanoi

- ❖ Now we can move the red disk to peg 3



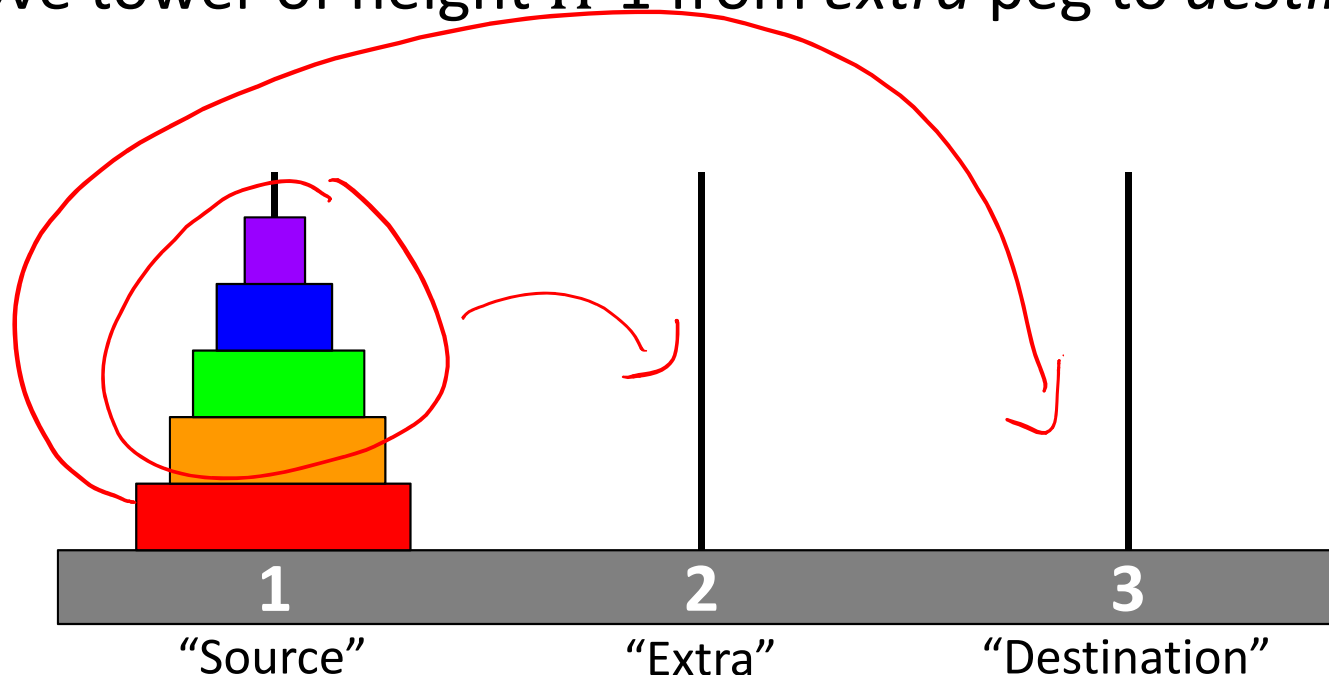
# Solving the Tower of Hanoi

- ❖ Now we can move the red disk to peg 3
  - `moveTower(1, peg1, peg3);`
- ❖ Next Goal: Move the tower of height 4 from peg 2 to peg 3
  - Solution: First move the 3 disks on top to peg 1, then move the orange disk to peg 3



# Solving the Tower of Hanoi

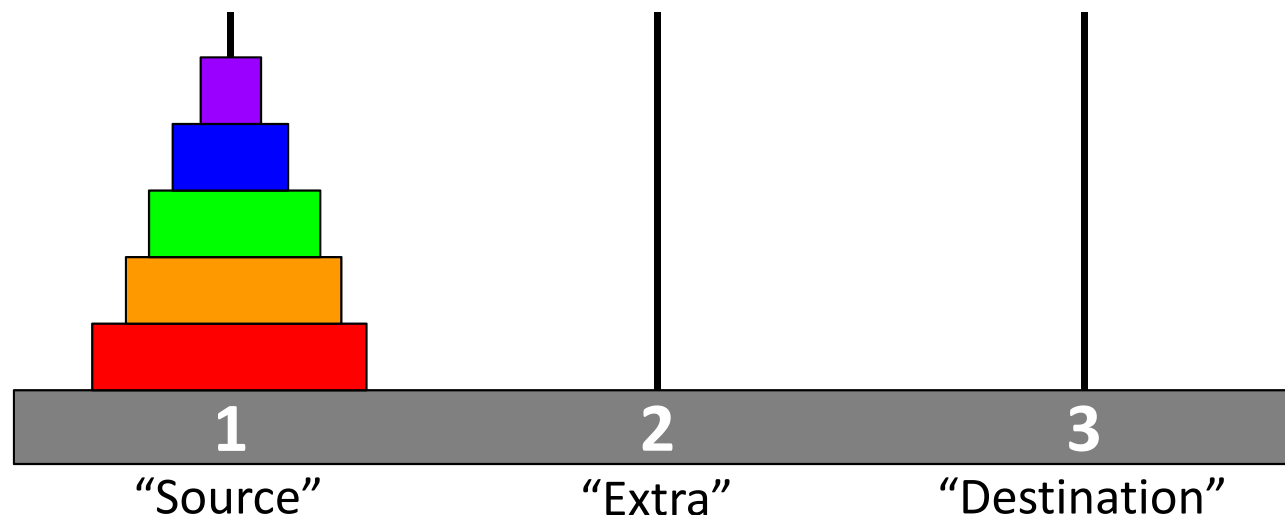
- ❖ Generalized recursive solution to move tower of height  $H$  from *source* peg to *destination* peg:
  - Move tower of height  $H-1$  from *source* peg to *extra* peg
  - Move the remaining bottom disk from *source* peg to *destination* peg
  - Move tower of height  $H-1$  from *extra* peg to *destination* peg



# Solving the Tower of Hanoi

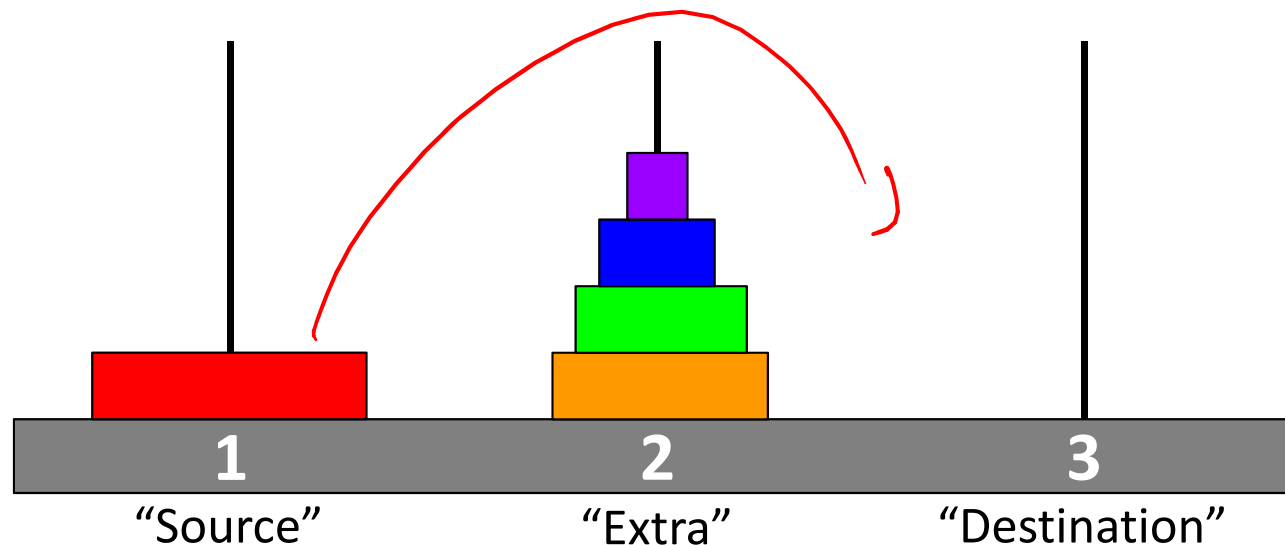
❖ Generalized recursive solution to move tower of height  $H$  from *source* peg to *destination* peg:

- `moveTower(H-1, peg1, peg2);`
  - `moveTower(1, peg1, peg3);`
  - `moveTower(H-1, peg2, peg3);`
- these are all contained in the body of `moveTower(H, peg1, peg3);`



# Solving the Tower of Hanoi

- ❖ What's the base case?
  - Don't recurse (but still move disk) when  $H == 1$



# Outline

- ❖ Example: Tower of Hanoi
- ❖ **Variable Scope Revisited**
- ❖ Example: Fibonacci
- ❖ Example: Snowflake Fractal

# Variable Scope Revisited

- ❖ Internal variables (*i.e.* parameters) only exist within the function they are declared
  - The variables “cease to exist” when the function returns
- ❖ Each individual call of a recursive function contains a *separate* set of parameters, even though they have the same variable names
  - Parameters have initial values set by the passed arguments

# Variable Scope Revisited

- ❖ Local variables take precedent over variables of the same name
  - Detail Removal: internal variable names are independent of external variable names, even if the same names are used
- ❖ We can think of every function call as creating a new function *environment*, which later disappears once the function returns
  - Global variables exist outside of these environments and are accessible to all of them

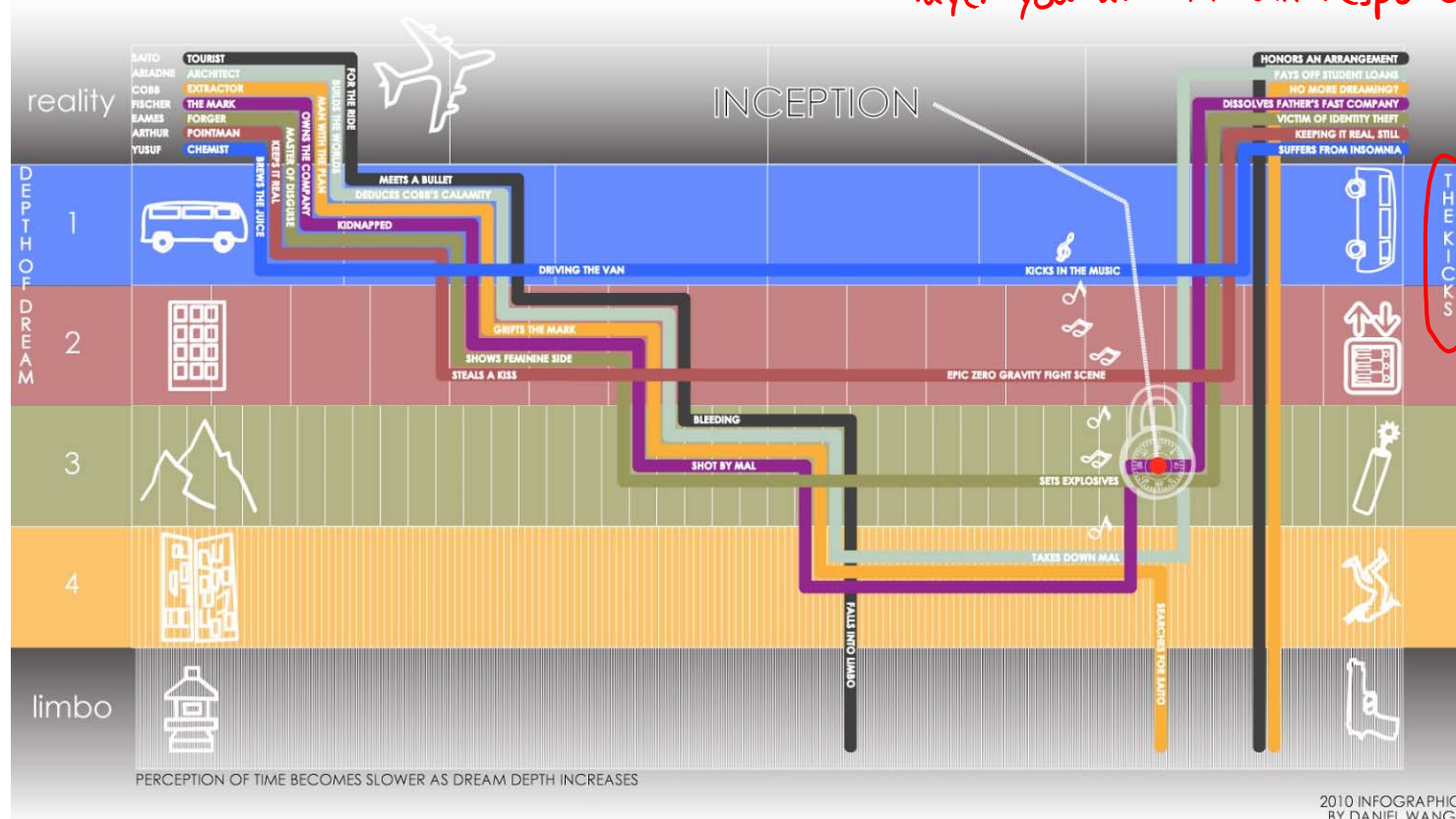


# 'Inception' Analogy (2010 film)

- ❖ Each dream is a function call, each "kick" is a function return
  - e.g. the 'reality' function calls the 'Robert Fischer dream' function
  - Characters are the parameters – they may have the same names, but are different (clothes?) in every layer

*when you call for Cobb, the Cobb in the current layer you are in will respond*

*function calls*



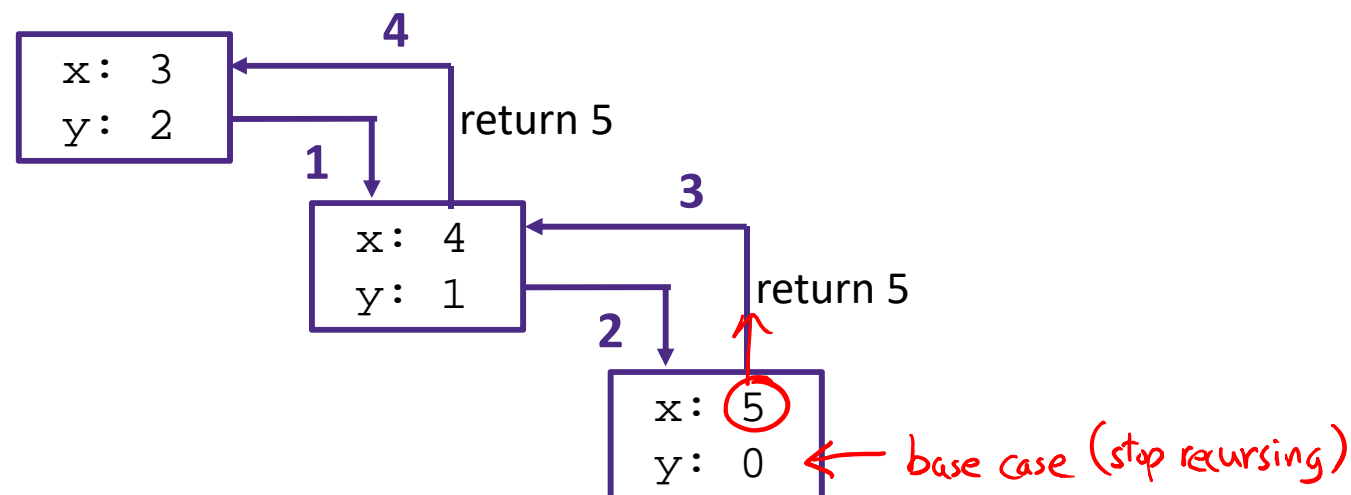
*← returning from a function*

# Add Example

## ❖ Recursive add ( ):

```
int add(int x, int y) {  
→ if (y==0) {  
    return x;  
} else {  
    return add(x+1, y-1);  
}  
}
```

## ❖ Environment diagram if we call add ( 3 , 2 ):



# Peer Instruction Question

❖ In the shown code, what will be printed after "3: " ?

▪ Vote at <http://PollEv.com/justinh>

- A. 0
- B. 1**
- C. 4
- D. 5

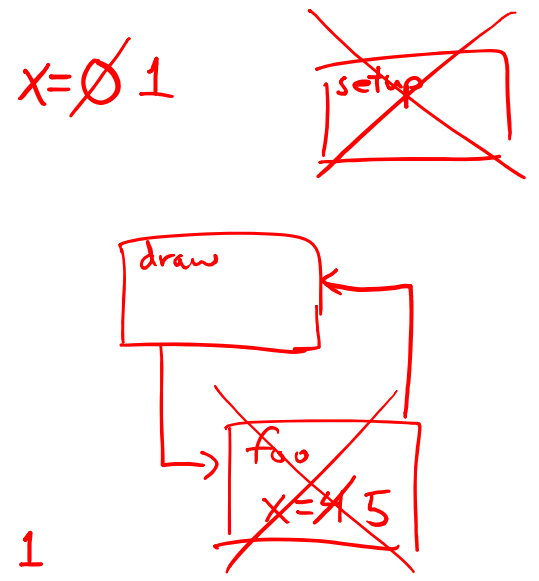
```

int x = 0; //global

void setup() {
  x = 1;
}

void draw() {
  println("1: " + x); → 1: 1
  foo(4);
  println("3: " + x); → 3: 1
  noLoop(); // stops here
}

void foo(int x) {
  x = 4x + 1;
  println("2: " + x); → 2: 5
}
    
```



# Outline

- ❖ Example: Tower of Hanoi
- ❖ Variable Scope Revisited
- ❖ **Example: Fibonacci**
- ❖ Example: Snowflake Fractal

# Fibonacci

❖ The Fibonacci Sequence is as follows:

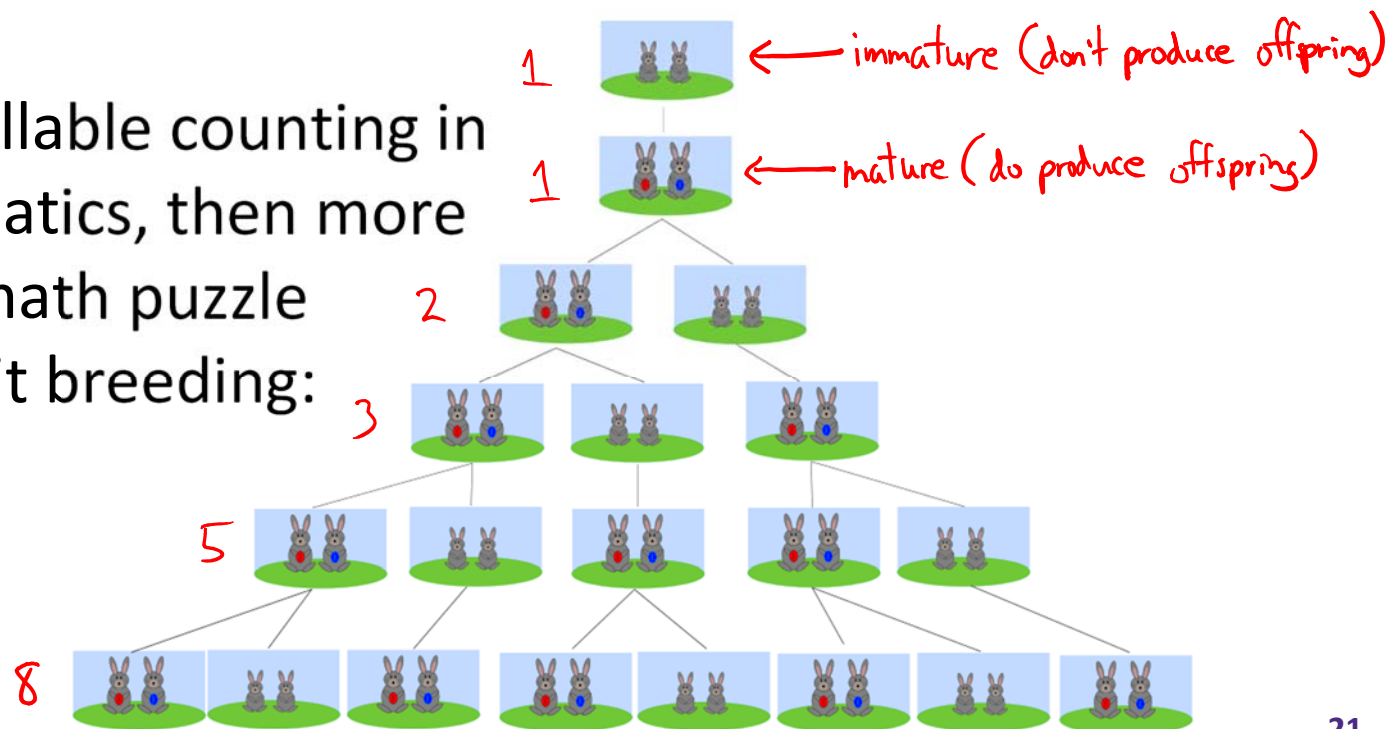
- 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ...

→ • The first two numbers are 0 and 1

→ • All following numbers are the sum of the previous two numbers

- [https://en.wikipedia.org/wiki/Fibonacci\\_number](https://en.wikipedia.org/wiki/Fibonacci_number)

- Appeared as syllable counting in Indian mathematics, then more famously in a math puzzle regarding rabbit breeding:



# Fibonacci

❖ The Fibonacci Sequence is as follows:

■ 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ...

•  $\text{fib}(0) = 0, \text{fib}(1) = 1$

• Otherwise,  $\text{fib}(n) = \text{fib}(n-1) + \text{fib}(n-2)$

*recursive  
definition!*

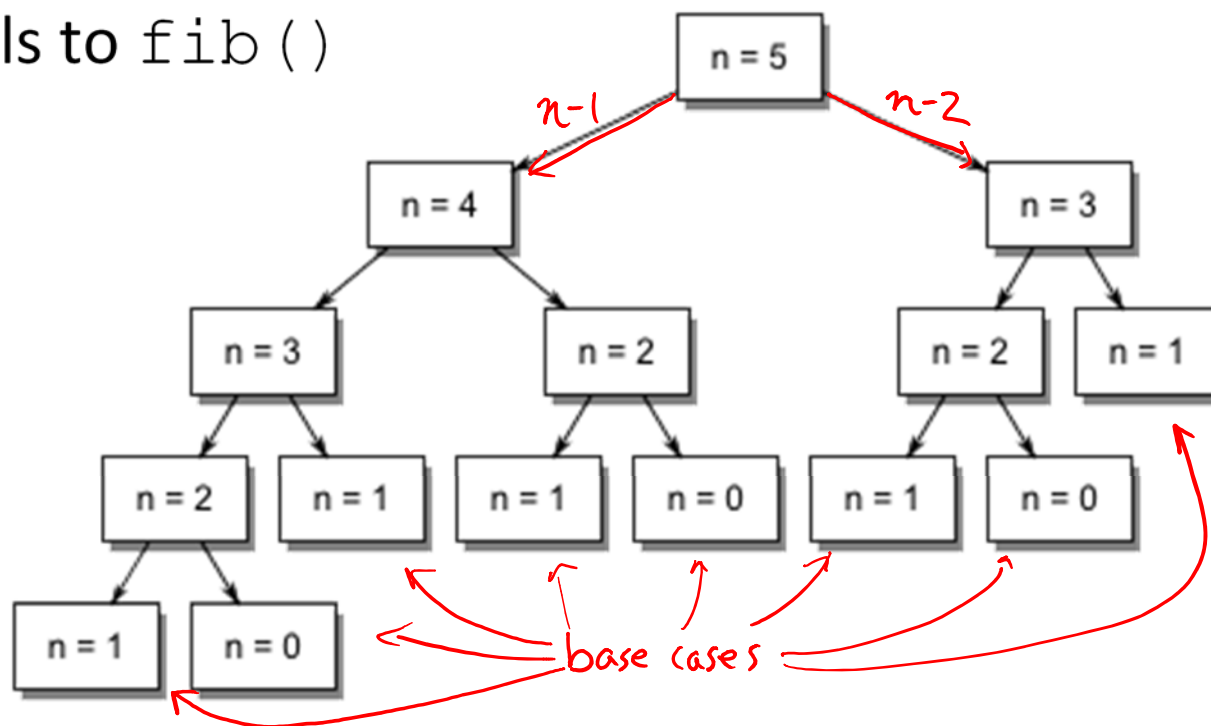
# Fibonacci Call Structure

- ❖ Call structure of `add ( )` looked like a call list
  - It contained one recursive call: `add (x+1, y-1)`

- ❖ Fibonacci makes how many recursive calls? **2**

$$\text{fib}(n) = \text{fib}(n-1) + \text{fib}(n-2);$$

- `fib ( )` looks like a call *tree* – each recursive case makes two calls to `fib ( )`



each environment has its own separate value for the parameter *n*

# Outline

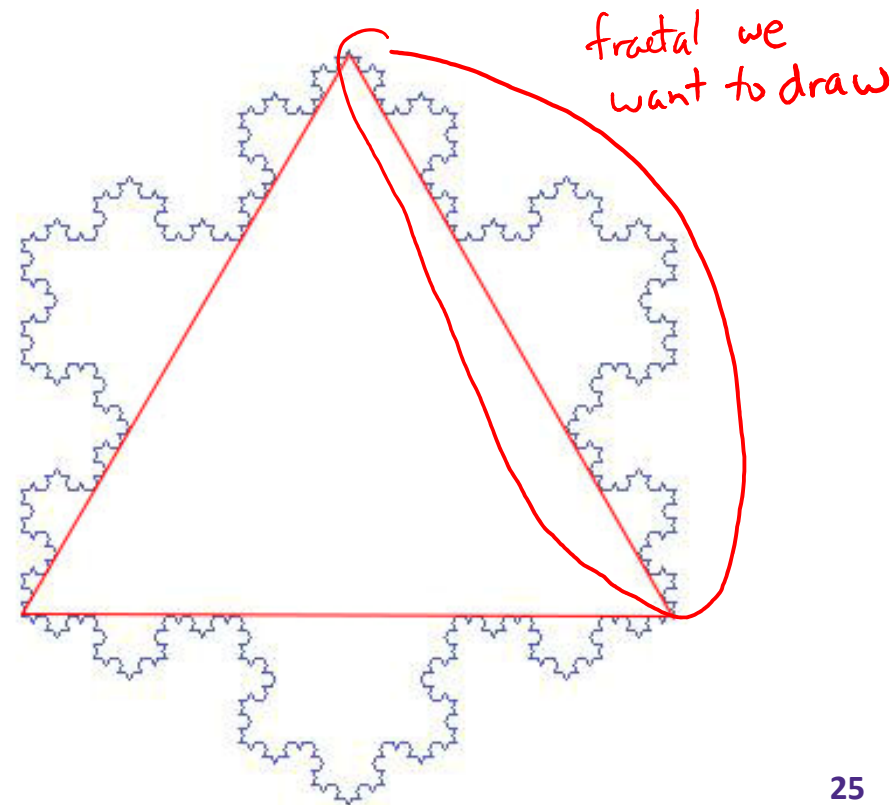
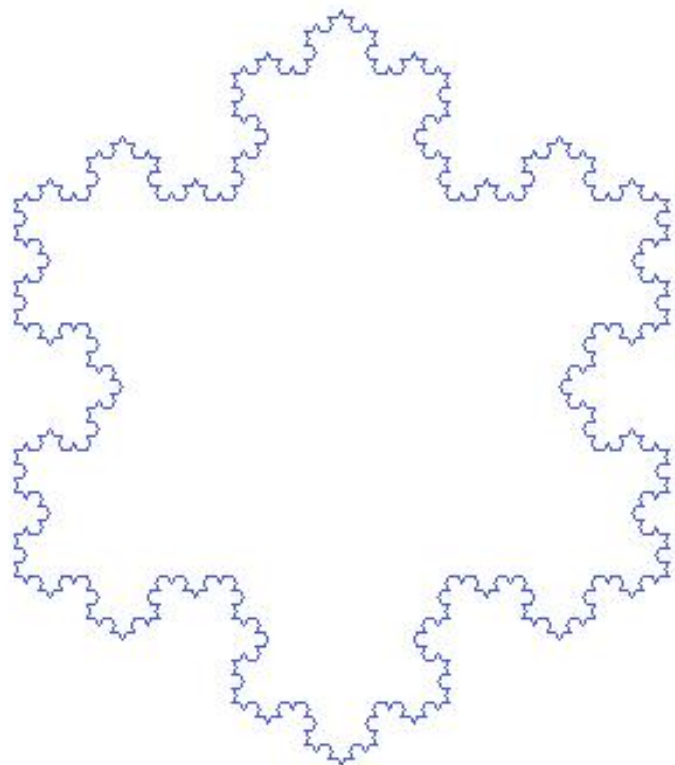
- ❖ Example: Tower of Hanoi
- ❖ Variable Scope Revisited
- ❖ Example: Fibonacci
- ❖ **Example: Snowflake Fractal**

The following exercise is from the Beauty and Joy of Computing (BJC) curriculum:  
<http://bjc.berkeley.edu/bjc-r/cur/programming/recur/fractals/snowflake.html>



# Koch Snowflake

- ❖ A mathematical curve that is one of the earliest fractal curves to have been described
  - [https://en.wikipedia.org/wiki/Koch\\_snowflake](https://en.wikipedia.org/wiki/Koch_snowflake)
  - 3 arranged copies of the same *fractal*



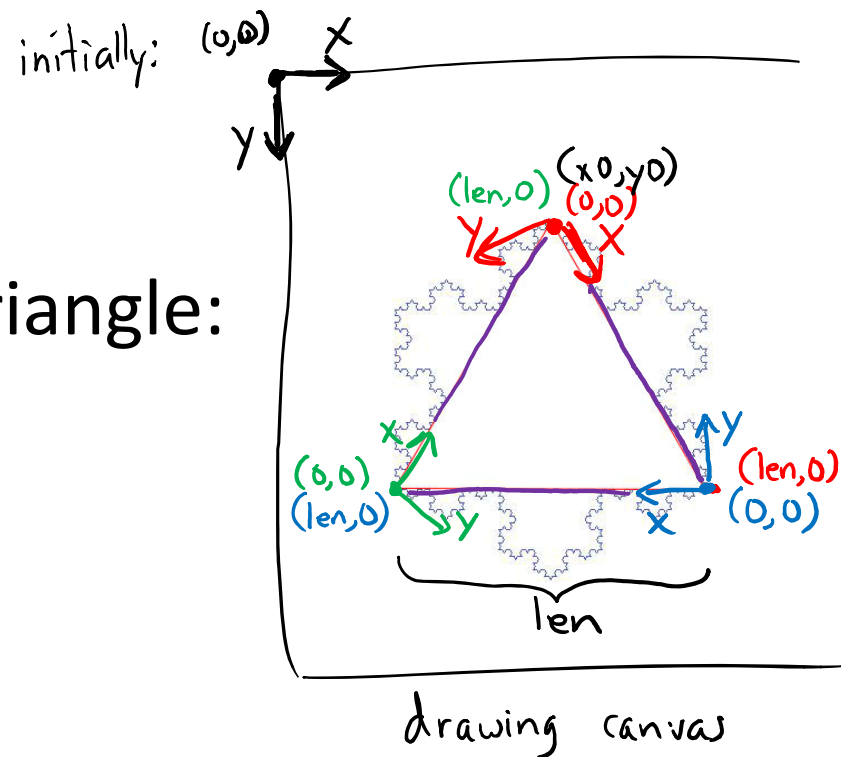
# Code: Triangle

- ❖ Copies of fractal arranged in a triangle:

```

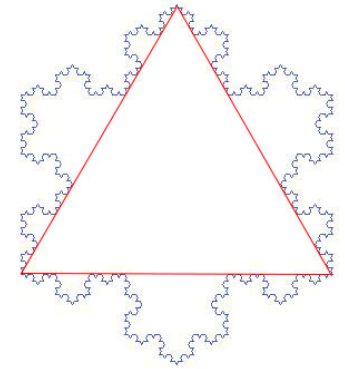
void draw() {
    translate(x0,y0);
    rotate(radians(60));
    for(int i=0; i<3; i=i+1) {
        line(0,0,len,0);
        translate(len,0);
        rotate(radians(120));
    }
    noLoop();
}

```



# Code: Triangle

- ❖ Copies of fractal arranged in a triangle:

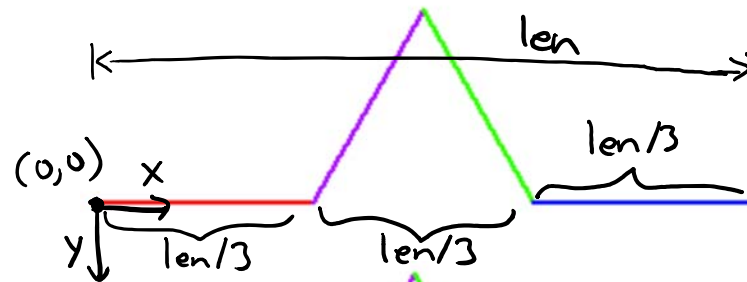


```
void draw() {  
    translate(250,100); // start at top point  
    rotate(radians(60));  
    for(int i=0; i<3; i=i+1) {  
        line(0,0,len,0); // replace with fractal  
        translate(len,0);  
        rotate(radians(120));  
    }  
    noLoop();  
}
```

# Drawing the Fractal

❖ Break each segment into 4 segments of equal length

■ First call:



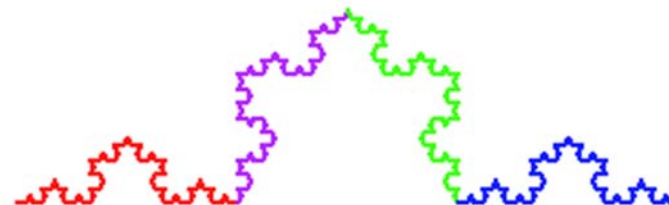
■ Second call:



■ Third call:



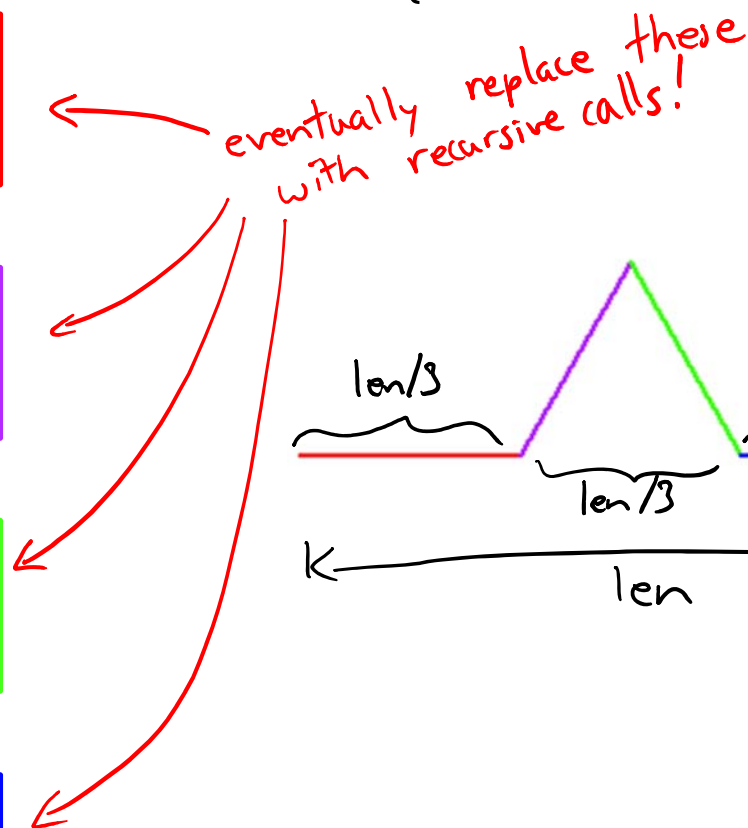
■ Fourth call:



# Code: Fractal

## ❖ First call:

```
void snowflake_fractal(float len) {  
    line(0,0,len/3,0);  
    translate(len/3,0);  
    rotate(radians(-60));  
    line(0,0,len/3,0);  
    translate(len/3,0);  
    rotate(radians(120));  
    line(0,0,len/3,0);  
    translate(len/3,0);  
    rotate(radians(-60));  
    line(0,0,len/3,0);  
    translate(len/3,0);  
}
```



# Code: Make It Recursive

## ❖ Recursive case

- Instead of drawing a line, draw the fractal!
  - Each smaller segment is  $1/3$  the length of the larger segment
  - Replace `line()` and `translate()` command pairs with calls to `snowflake_fractal()`

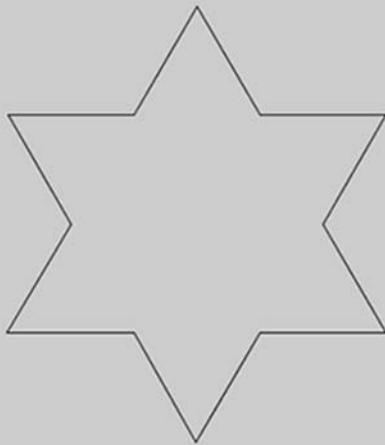
## ❖ Base case

- Introduce `level` variable
  - Arbitrarily tells us how deep to recurse
- When `level==0`, just draw line instead of fractal

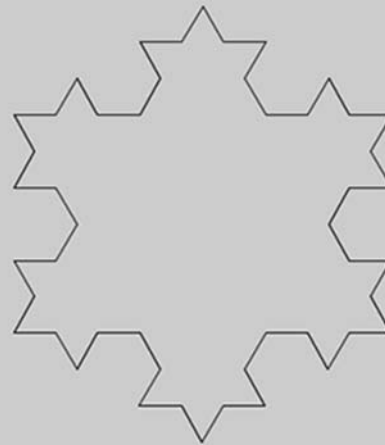
# The Result

- ❖ Can draw snowflake fractal of arbitrary depth!

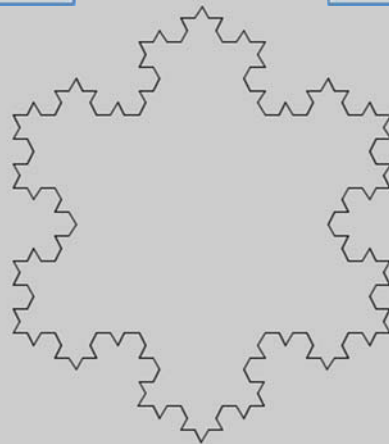
level=1



level=2



level=3



level=4

