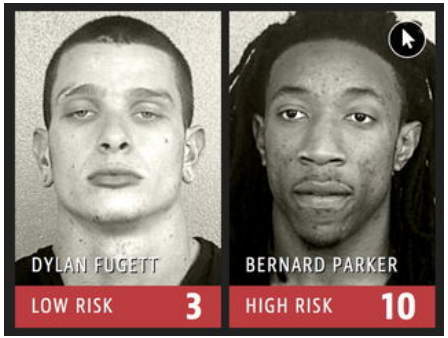
CS Responsibility: Algorithmic Bias

- Reports from subcontracted workers for Google collecting data for facial recognition tech
- Asked to target people with “darker skin tones”, people more likely to be enticed by $5 gift card
- “I feel like they wanted us to prey on the weak”
- Connected to larger issue: algorithmic bias
  - Search engines
  - Criminal risk assessment
  - Facial recognition

Sources:

https://towardsdatascience.com/mitigating-algorithmic-bias-in-predictive-justice-ux-design-principles-for-ai-fairness-machine-learning-d2227ce28099
https://www.theguardian.com/technology/2019/oct/03/google-data-harvesting-facial-recognition-people-of-color
https://www.newscientist.com/article/2166207-discriminating-algorithms-5-times-ai-showed-prejudice/
https://www.wired.com/story/the-real-reason-tech-struggles-with-algorithmic-bias/
https://towardsdatascience.com/mitigating-algorithmic-bias-in-predictive-justice-ux-design-principles-for-ai-fairness-machine-learning-d2227ce28099
https://9to5google.com/2019/10/04/google-wants-tiktok-competitor/

More on algorithmic bias



Disparate impact vs. disparate treatment

Search engines: jobs on Google

- Image search for “CEO”: 11% female, vs. 27% of actual CEOs in U.S.
- High-income jobs shown to men much more often than to women

Criminal Risk Assessment: COMPAS model by Northpointe

- High-risk, didn’t offend: 44.9% Black, 23.5% White
- Low-risk, did offend: 28% black, 47.7% white

Facial recognition: Study in 2018 on three of the biggest gender-recognition AI’s

- 99% accuracy on white men, 35% accuracy on dark-skinned women

Back to Google: Attempted to resolve racist facial recognition technology…by being racist?

How should algorithmic bias be addressed?

Important Course Reminders

- Please don't address TAs or HTAs outside of TA/Conceptual Hours, Sections, HTA Hours, or e-mail. TAs and HTAs are students, just as you, and their personal time should be dedicated to their own coursework.
- **Starting early is essential for the next projects of the course**. The difficulty grows exponentially. We strongly suggest taking advantage of conceptual hours, TA hours, and Piazza.
- Confused by a lecture topic**? Come to conceptual hours**! Stuck on part of DoodleJump? Come to conceptual hours! Want to discuss containment with others? Come to conceptual hours!
- The cutoff doesn't mean that you won't be seen in hours, it just means that there is **a chance** that you won't be seen!
- Section and labs **are mandatory!** Not going to sections and labs will affect your grade in the course. You can work on labs previously and get checked off at the beginning of your lab, but you can't skip it entirely.
- We know that TA hours lines have been getting longer, and that this course is difficult; however, always be mindful about the way that you address TAs, they are here to help and they deserve to be treated with respect **at all times.**

Lecture 14

Recursion

Jim, Pam, Dwight & Michael Like Cookies (1/2)

- They would each like to have one of these cookies:

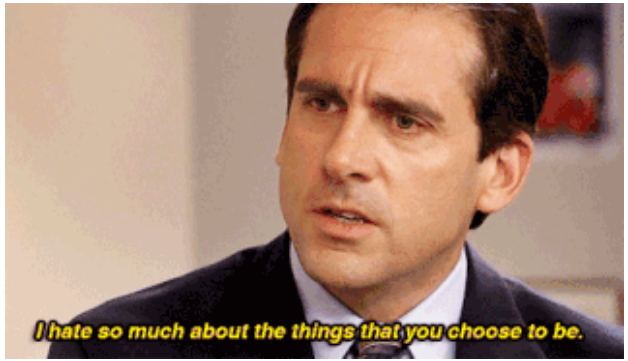



- How many ways can they distribute the cookies amongst themselves?
  - the first character who picks has four choices
  - three choices are left for the second character
  - two choices left for the third character
  - the last character has to take what remains (poor Michael!)

Jim, Pam Dwight & Michael Like Cookies (2/2)

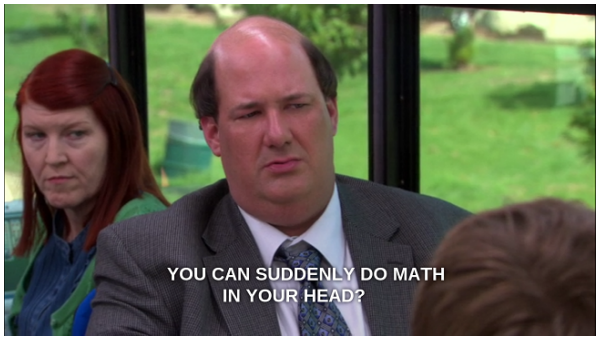
- Thus we have 24 different ways the characters can choose cookies (4! = 4 x 3 x 2 x 1 = 24)

- What if we wanted to solve this problem for all of Michael's enemies– all of the employees at Dunder Mifflin?



Factorial Function

- Model this problem mathematically:
  factorial (n!) calculates the total number of unique *permutations*, or the number of different ways to arrange/order *n* items



- Small examples:
  1! = 1
  2! = 2*1 = 2
  3! = 3*2*1 = 6
  4! = 4*3*2*1 = 24
  5! = 5*4*3*2*1 = 120

- *Iterative* definition: n! = n * (n-1) * (n-2) * … * 1

- *Recursive* definition: n! = n * (n-1)! for n >= 1 and 0! = 1

Recursion (1/2)

- Models problems that are self-similar
  - decompose a whole task into smaller, similar sub-tasks
  - each subtask can be solved by applying the same technique
- Whole task solved by combining solutions to sub-tasks
  - special form of divide and conquer at every level

Recursion (2/2)

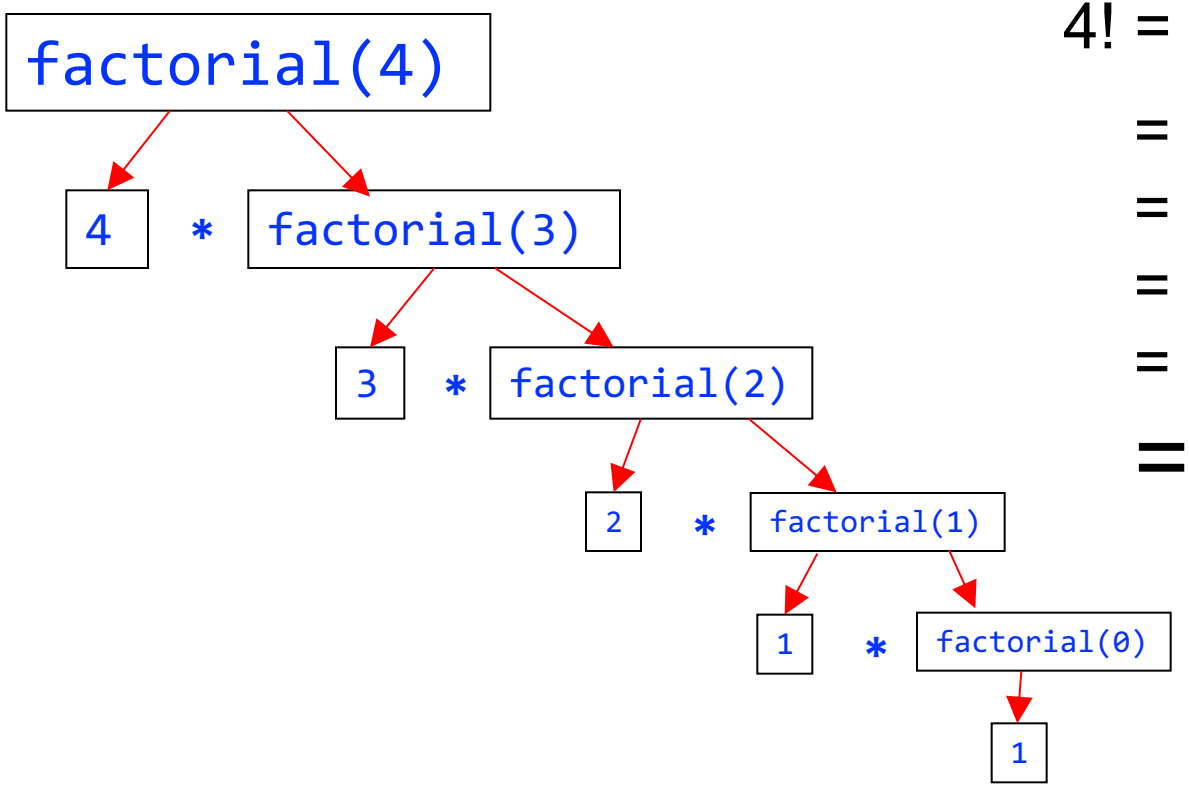
- Task is defined in terms of itself
  - in Java, recursion is modeled by method that calls itself, but each time with simpler case of the problem, hence the recursion will "bottom out" with a **base case** eventually
  - **base case** is a case simple enough to be solved directly, without recursion; otherwise infinite recursion and `StackOverflowError`
  - what is the base case of the factorial problem?
  - Java will bookkeep each invocation of the same method just as it does for nested methods that differ, so there is no confusion
  - usually you combine the results from the separate invocations

Factorial Function Recursively (1/2)

- Recursive factorial algorithm
  - the factorial function is not defined for negative numbers
    - the first conditional checks for this precondition
    - it is good practice to document and test preconditions (see code example)
  - number of times method is called is the depth of recursion (1 for 0!)
    - what is depth of (4!)?

```
public class RecursiveMath{
    //instance variables, other code elided
    public int factorial (int num) {
        if (num < 0){
            System.out.println("Input must be >= 0");
            return -1; // return -1 for invalid input
        }

        int result = 0;
        if (num == 0){ // base case: 0! = 1
            result = 1;
        }
        else{  //general case
            result = num * this.factorial(num - 1);
        }
        return result;
    }
}
```

Factorial Function Recursively (2/2)



4! = factorial(4)

= 4 * 3!

= 4 * 3 * 2!

= 4 * 3 * 2 * 1!

= 4 * 3 * 2 * 1 * 0!

= 24

TopHat Question

Given the following non-practical code:

```
public class RecursiveMath {

    public int recursiveAddition(int n) {
        if (n<=1) {
            return 1;
        } else {
            return recursiveAddition(n-1);
        }
    }

}
```

What is the output of `this.recursiveAddition(4)`?

A. 1
B. 9
C. 10
D. `StackOverflowError`

TopHat Question

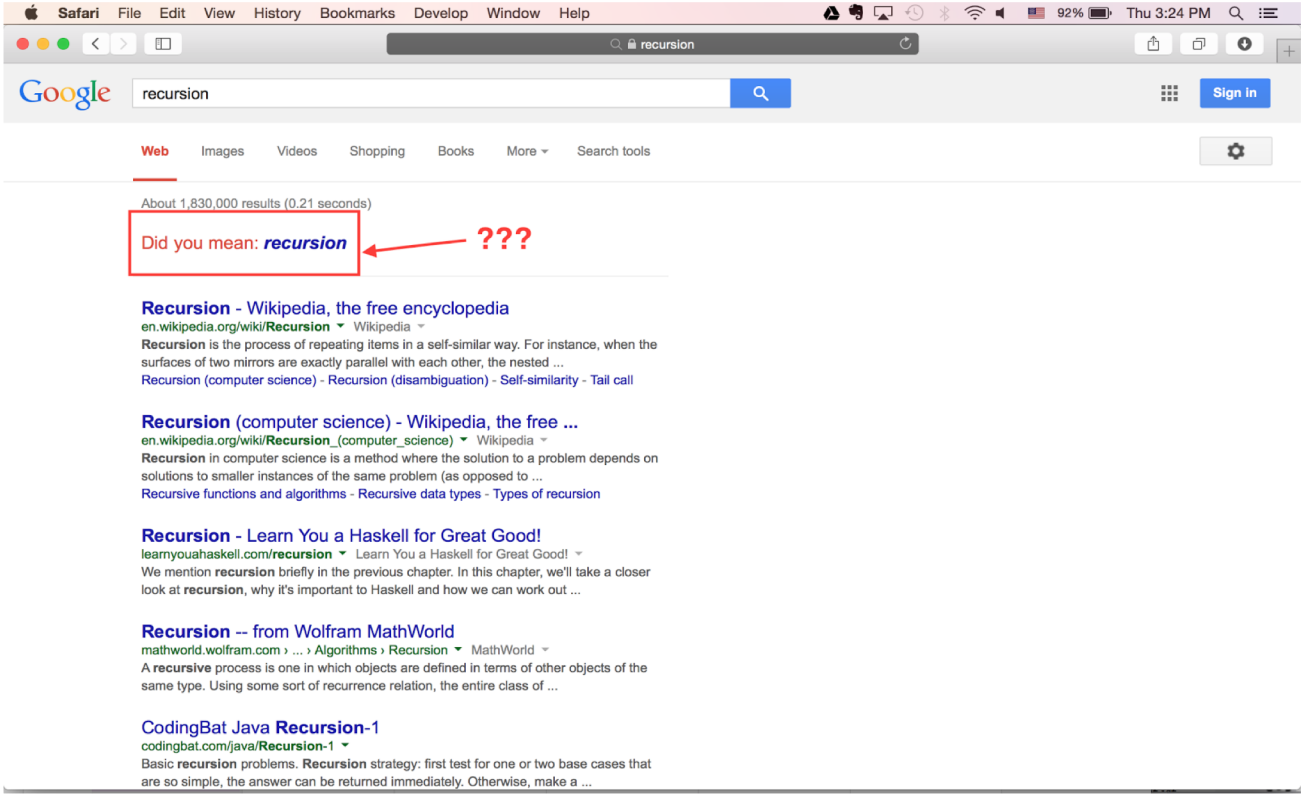
Given the following code:

```
public class RecursiveMath {

    public int funkyFactorial(int n) {
        if (n == 0) {
            return 1;
        } else {
            return n * this.funkyFactorial(n-2);
        }
    }

}
```

What is the output of `this.funkyFactorial(5)`?

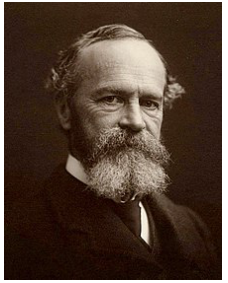
A. 1
B. 5
C. 15
D. `StackOverflowError`

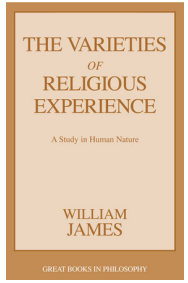
If you want to know more about recursion...



Turtles in Recursion – from Wikipedia

The following anecdote is told of William James. After a lecture on cosmology and the structure of the solar system, James was accosted by a little old lady.
"Your theory that the sun is the centre of the solar system, and the earth is a ball which rotates around it has a very convincing ring to it, Mr. James, but it's wrong. I've got a better theory," said the little old lady.
"And what is that, madam?" inquired James politely.
"That we live on a crust of earth which is on the back of a giant turtle."
Not wishing to demolish this absurd little theory by bringing to bear the masses of scientific evidence he had at his command, James decided to gently dissuade his opponent by making her see some of the inadequacies of her position.
"If your theory is correct, madam," he asked, "what does this turtle stand on?"
"You're a very clever man, Mr. James, and that's a very good question," replied the little old lady, "but I have an answer to it. And it's this: The first turtle stands on the back of a second, far larger, turtle, who stands directly under him."
"But what does this second turtle stand on?" persisted James patiently.
To this, the little old lady crowed triumphantly,
"It's no use, Mr. James — it's turtles all the way down."
— *J. R. Ross, Constraints on Variables in Syntax 1967*

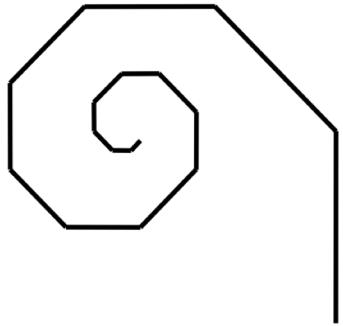




William James (January 11, 1842 – August 26, 1910)
Earliest psychologist

Call Out the Turtles

- Benoit Mandelbrot developed Fractals, a mathematical branch whose principle characteristic is self-similarity at any scale, one of the characteristics of recursions. Fractals are common in nature (botany, lungs, blood vessels, kidneys…), cosmology, antennas, $Z_{n+1} = Z_n^2 + C$ in the complex plane (x, i) where i = sqrt(-1)…
  - check out:
    - https://www.youtube.com/watch?v=2JUAojvFpCohttp://matek.hu/xaos/doku.php
    - http://projects.delimited.io/experiments/dragon-curves/dragons-single.html
    - https://www.youtube.com/watch?v=aSg2Db3jF_4
    - http://bl.ocks.org/syntagmatic/3736720
    - https://www.youtube.com/watch?v=4LQvjSf6SSw
- Some simpler, non-fractal, but still self-similar shapes composed of smaller, simpler copies of some pattern are simple spirals, trees, and snowflakes
- We can draw these using Turtle graphics
  - iteratively: Start at a particular point, facing in a chosen direction (here up). Draw successively shorter lines, each line at a given angle to the previous one
  - recursively: Start at a particular point, in a given direction. Draw a line of passed-in length, turn the passed-in angle, decrement length and call spiral recursively



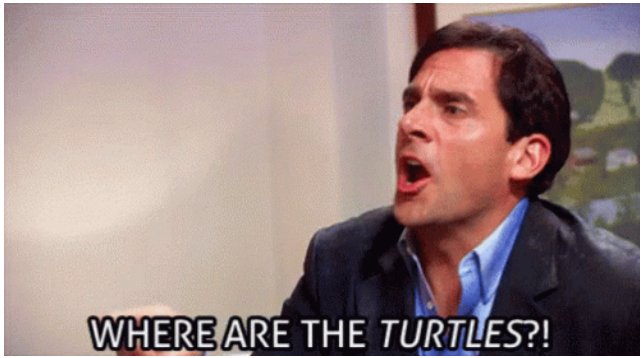
Designing `Spiral` Class (1/2)

- `Spiral` class defines single draw method
  - turtle functions as pen to draw spiral, so class needs reference to turtle instance

- Constructor's parameters to control its properties:
  - position at which spiral starts is turtle's position
  - length of spiral's starting side
  - angle between successive line segments
  - amount to change length of spiral's side at each step
  - Note: this info is passed to each invocation of recursive method, so next method call depends on previous one

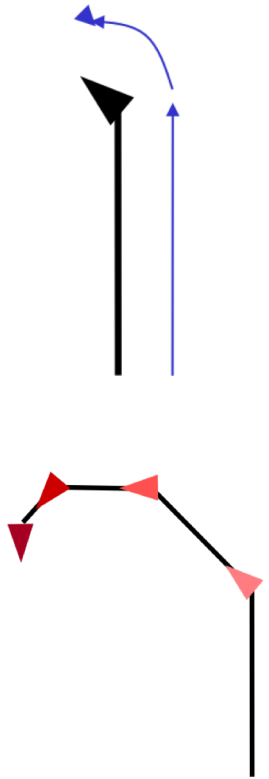
Designing `Spiral` Class (2/2)

```
public class Spiral {
    private Turtle _turtle;
    private double _angle;
    private int _lengthDecrement;
     // passing in parameters to set the properties of the spiral
    public Spiral(Turtle myTurtle, double myAngle, int myLengthDecrement) {
        _turtle = myTurtle;
        _angle = myAngle;
        _lengthDecrement = 1; // default, handles bad parameters
        if (myLengthDecrement > 0){
            _lengthDecrement = myLengthDecrement;
        }
        // draw method defined soon...
    }
}
```



Drawing `Spiral`

- First Step: Move turtle forward to draw line and turn some degrees. What's next?
- Draw smaller line and turn! Then another, and another…



Sending Recursive Messages (1/2)

- `draw` method uses turtle to trace spiral
- How does `draw` method divide up work?
  - draw first side of spiral
  - then draw smaller spiral (this is where we implement recursion)

```
public void draw(int sideLen){

    // general case: move sideLen, turn
    // angle and draw smaller spiral
    _turtle.forward(sideLen);
    _turtle.left(_angle);
    this.draw(sideLen - _lengthDecrement);
}
```

Sending Recursive Messages (2/2)

- What is the base case?
  - when spiral is too small to see, conditional statement stops method so no more recursive calls are made
  - since side length must approach zero to reach the **base case** of the recursion, the `draw` method is called recursively, passing in a smaller side length each time

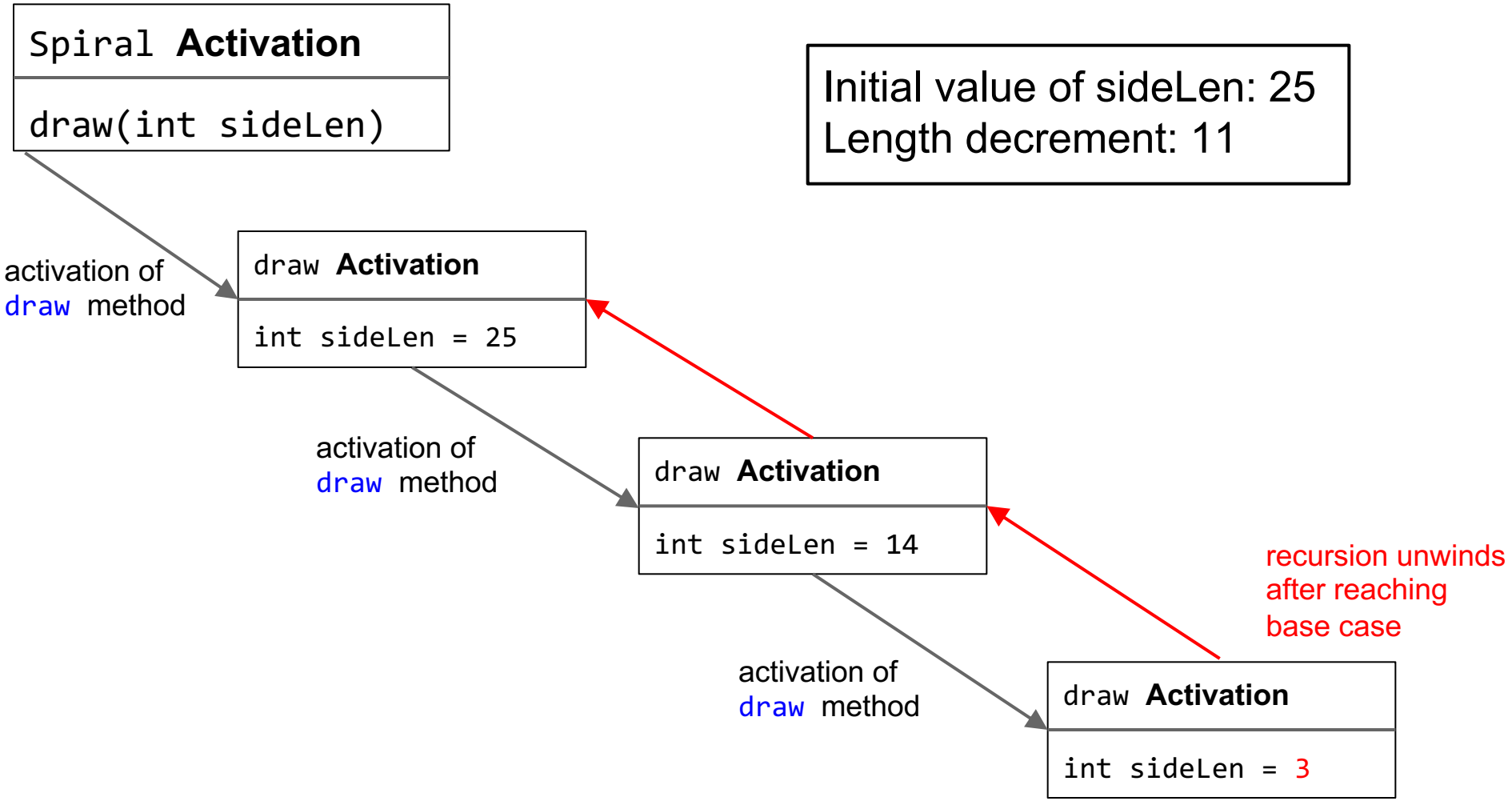
```
public void draw(int sideLen){
    // base case: spiral too small to see
    if (sideLen <= 3) {
        return;
    }

    // general case: move sideLen, turn
    // angle and draw smaller spiral
    _turtle.forward(sideLen);
    _turtle.left(_angle);
    this.draw(sideLen - _lengthDecrement);
}
```

Recursive Methods

- We are used to seeing a method call other methods, but now we see a method calling itself
- Method must handle successively smaller versions of original task

Method's Variable(s)

- As with separate methods, each invocation of the method has its own copy of parameters and local variables, and shares access to instance variables
- Parameters let method invocations (i.e., successive recursive calls) "communicate" with, or pass info between, each other
- Java's record of current place in code and current values of parameters and local variables is called the *activation record*
  - with recursion, multiple activations of a method may exist at once
  - at base case, as many exist as depth of recursion
  - each activation of a method is stored on the activation stack (you'll learn about stacks soon)


Spiral Activation
draw(int sideLen)
Initial value of sideLen: 25
Length decrement: 11
activation of
draw method
draw Activation
int sideLen = 25
activation of
draw method
draw Activation
int sideLen = 14
recursion unwinds
after reaching
base case
activation of
draw method
draw Activation
int sideLen = 3

TopHat Question

Given the following code for the Collatz conjecture:

```
public class RecursiveMath{
    private int _count;
    //constructor elided. _count gets 0, it records
    //number of calls on collatzCounter
    public int collatzCounter(int n) {
        _count += 1;
        if (n == 1) { //base case
            return 1;
        } else {
            if (n % 2 == 0) { //if n is even
                return collatzCounter(n / 2);
            } else {
                return collatzCounter(3 * n + 1);
            }
        }
    }
}
```

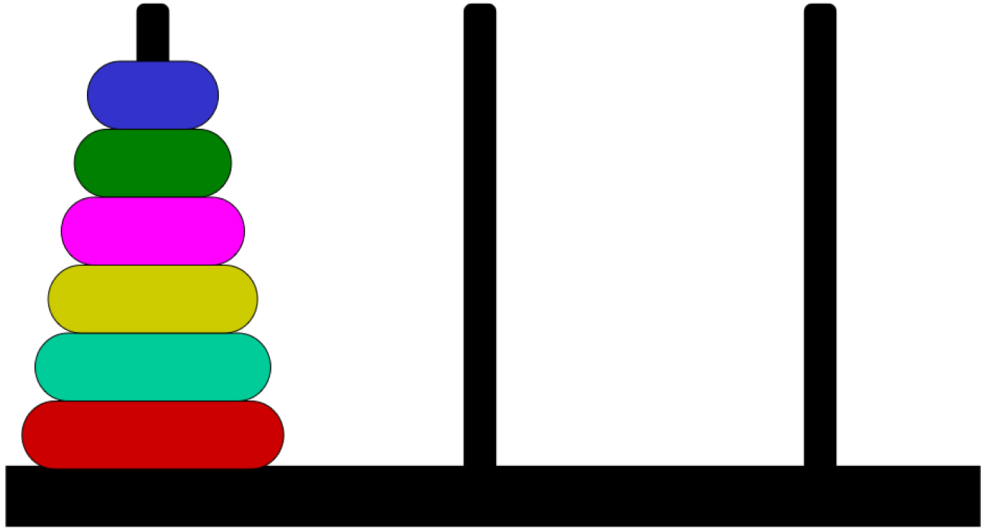
What is the value of `_count` after calling `collatzCounter(5)`?

A. 4
B. 5
C. 6
D. `StackOverflowError`

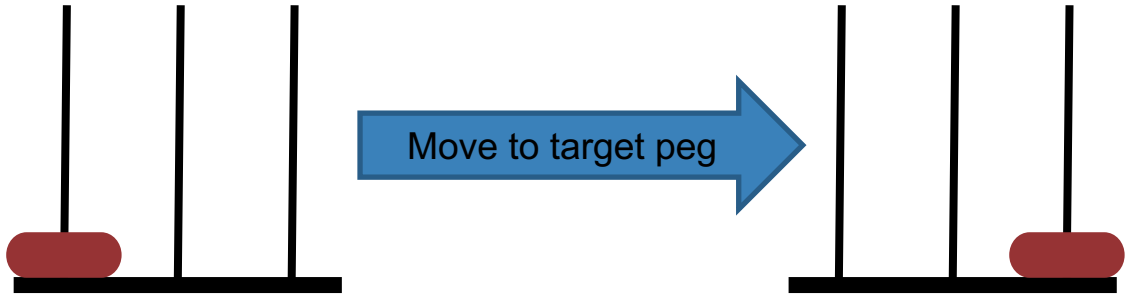
"The **Collatz conjecture** is a conjecture in mathematics named after Lothar Collatz. It concerns a sequence defined as follows: start with any positive integer *n*. Then each term is obtained from the previous term as follows: if the previous term is even, the next term is one half the previous term. Otherwise, the next term is 3 times the previous term plus 1. The conjecture is that no matter what value of *n*, the sequence will always reach 1." (From Wikipedia)

Towers of Hanoi (1/4)

- Game invented by French mathematician Edouard Lucas in 1883
- **Goal**: move tower of n disks, each of a different size (in order, with smallest at top), from left-most peg to right-most peg
- **Rule 1:** no disk can be placed on top of a smaller disk to win
- **Rule 2:** only one disk can be moved at a time

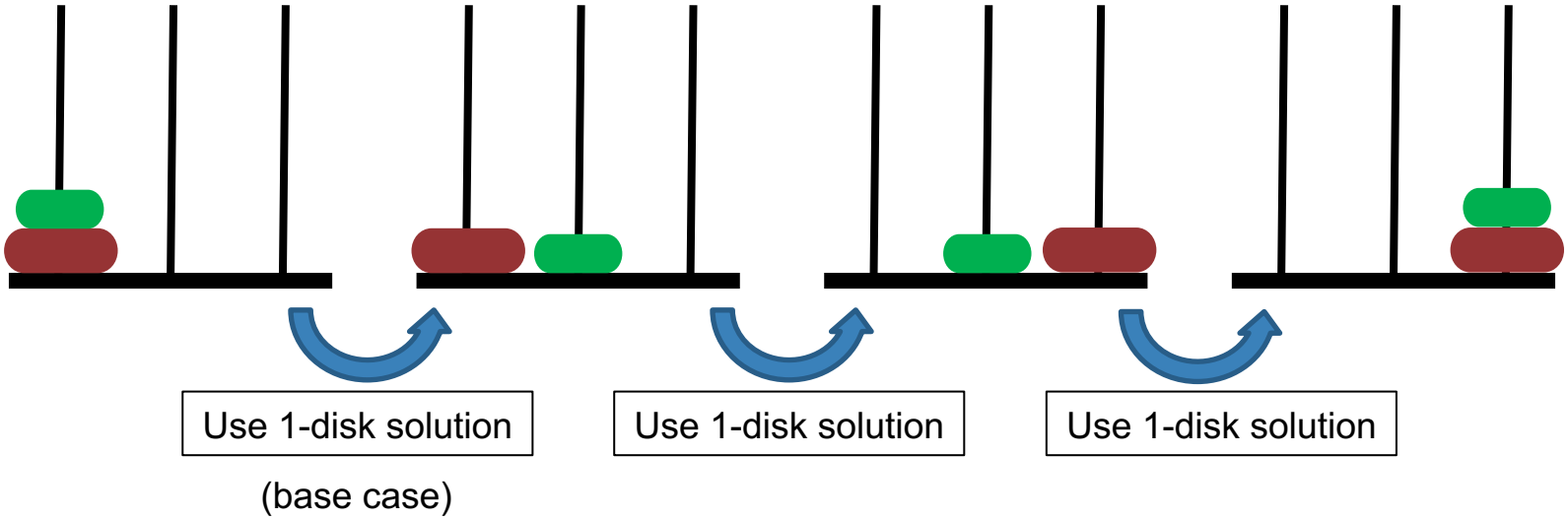


One Disk Solution

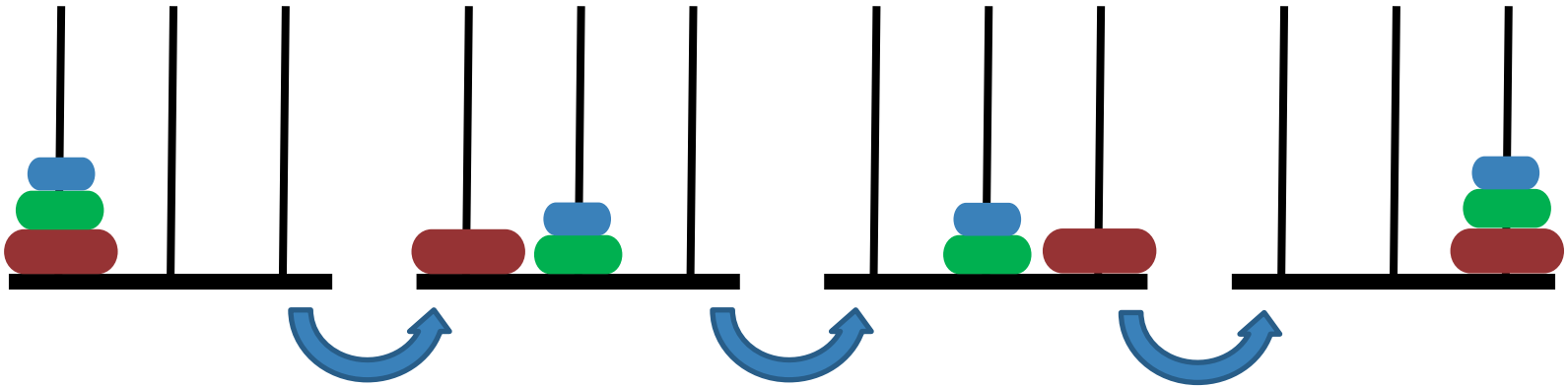


Base Case

Two Disk Solution



Three Disk Solution



Pseudocode for Towers of Hanoi (1/2)

- Try solving for 5 non-recursively…
- One disk:
  - move disk to final pole
- Two disks:
  - use one disk solution to move top disk to intermediate pole
  - use one disk solution to move bottom disk to final pole
  - use one disk solution to move top disk to final pole
- Three disks:
  - use two disk solution to move top disks to intermediate pole
  - use one disk solution to move bottom disk to final pole
  - use two disk solution to move top disks to final pole

Pseudocode for Towers of Hanoi (2/2)

- In general (for *n* disks)
  - use *n-1* disk solution to move top disks to intermediate pole
  - use one disk solution to move bottom disk to final pole
  - use *n-1* disk solution to move top disks to final pole

- Note: can have multiple recursive calls in a method

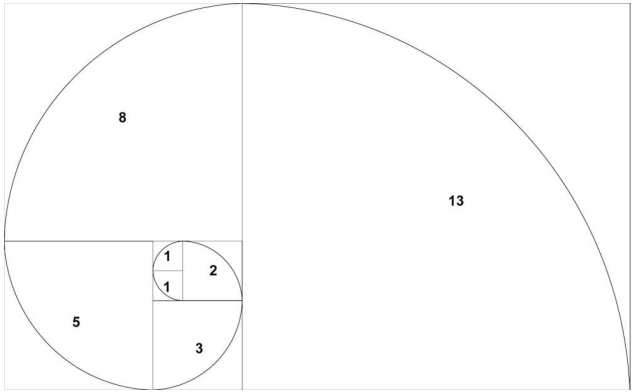
Lower level pseudocode

```
//n is number of disks, src is starting pole,
//dst is finishing pole
public void hanoi(int n, Pole src, Pole dst, Pole other){
    if (n==1) {
        this.move(src, dst);
    }
    else {
        this.hanoi(n-1, src, other, dst);
        this.move(src, dst);
        this.hanoi(n-1, other, dst, src);
    }
}

public void move(Pole src, Pole dst){
    //take the top disk on the pole src and make
    //it the top disk on the pole dst
}
```

- That's it! `otherPole` and `move` are fairly simple methods, so this is not much code.
- But try hand simulating this when n is greater than 4. Whoo boy, it is tedious (but not hard!)
- Iterative solution far more complex, and much harder to understand

Fibonacci Sequence (1/2)

- 1, 1, 2, 3, 5, 8, 13, 21…
- Each number is calculated by adding the two previous numbers
  - $F_n = F_{n-1} + F_{n-2}$



Fun application of fibonacci sequence:
https://www.npr.org/2018/08/10/637470699/let-this-percussionist-blow-your-mind-with-the-fibonacci-sequence

Fibonacci Sequence (2/2)

- What is the base case?
  - there are two: n=0 and n=1
- Otherwise, add two previous values of sequence together
  - this is also two recursive calls!

```
// returns nth value of Fibonacci sequence
public int fib(int n){
    if (n < 0) {
        System.out.println("input must be >= 0");
        return -1;
    }
    // base cases: n is 0 or 1
    if (n == 0 || n == 1) {
        return 1;
    }
    // general case: add previous two values
    // using two recursive calls
    return fib(n-1) + fib(n-2);
}
```

TopHat Question

Given the following code:

```
public int fib(int n){
    //error check
    if (n < 0) {
        return -1;
    }
    //base case
    if (n == 0 || n == 1) {
        return 1;
    }
    return fib(n-1) + fib(n-2);
}
```

What number would be returned if you **excluded** `n == 1` from the base case and called `fib(2)`?

A. 5
B. 3
C. 2
D. 1

Loops vs. Recursion (1/2)

- `Spiral` uses simple form of recursion
  - each sub-task only calls on one other sub-task
  - this form can be used for the same computational tasks as iteration
  - loops (iteration) and simple recursion are computationally equivalent in the sense of producing the same result, if suitably coded (not necessarily the same performance, though -- looping is more efficient)

Loops vs. Recursion (2/2)

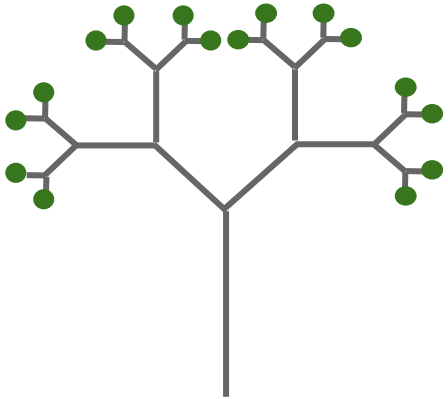
- Iteration is often more efficient in Java because recursion takes more method calls (each activation record takes up some of the computer's memory)
- Recursion is more concise and more elegant for tasks that are "naturally" self-similar (Towers of Hanoi is very difficult to solve iteratively!)

```
public void drawIteratively(int sideLen){

    while(sideLen > 3){
        _turtle.forward(sideLen);
        _turtle.left(_angle);
        sideLen -= _lengthDecrement;

    }
}
```

Indirect Recursion

- Two or more methods act recursively instead of just one
- For example, `methodA` calls `methodB` which calls `methodA` again
- Methods may be implemented in same or different classes
- Can be implemented with more than two methods too

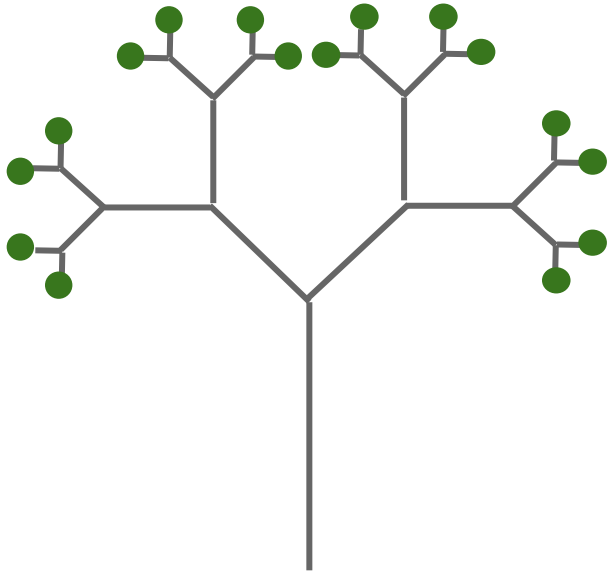
Recursive Binary Tree (1/2)



- The tree is composed of a trunk that splits into two smaller branches that sprout in opposite directions at the same angle
- Each branch then splits as the trunk did until sub-branch is deemed too small to be seen. Then it is drawn as a leaf
- The user can specify the length of a tree's main trunk, the angle at which branches sprout, and the amount by which to decrement each branch

Recursive Binary Tree (2/2)

- Compare each left branch to its corresponding right branch
  - right branch is simply rotated copy
- Branches are themselves smaller trees!
  - branches are themselves smaller trees!
    - branches are themselves smaller trees!
      - …
- Our tree is self-similar and can be programmed recursively!
  - base case is leaf



Designing the Tree Class

- Tree has properties that user can set:
  - start position (myTurtle's built in position)
  - angle between branches (myBranchAngle)
  - amount to change branch length (myTrunkDecrement)

- Tree class will define a single draw method
  - like Spiral, also uses a Turtle to draw

```
public class Tree{
private Turtle _turtle;
private double _branchAngle;
private int _trunkDecrement;
    public Tree(Turtle myTurtle, double myBranchAngle,
      int myTrunkDecrement){

        _turtle = myTurtle;
        _trunkDecrement = 1;

        if(myTrunkDecrement > 0){
            _trunkDecrement = myTrunkDecrement;
        }

        if(myBranchAngle > 0){
            _branchAngle = myBranchAngle;
        }
    }

      // draw method coming up…
}
```

`Tree`'s `draw` Method

- **Base case**: if branch size too small, add a leaf
- **General case**:
  - move `_turtle` forward
  - orient `_turtle` left
  - recursively draw left branch
  - orient `_turtle` right
  - recursively draw right branch
  - reset `_turtle` to starting orientation
  - back up to prepare for next branch

```
private void draw(int trunkLen){
     if (trunkLen <= 0) {
          this.addLeaf();
     } else {
          _turtle.forward(trunkLen);
          _turtle.left(_branchAngle);
          this.draw(trunkLen - _trunkDecrement);
          _turtle.right(2 * _branchAngle);
          this.draw(trunkLen - _trunkDecrement);
          _turtle.left(_branchAngle);
          _turtle.back(trunkLen);
     }
}
```

Tree's draw Method

```
private void draw(int trunkLen){
     if (trunkLen <= 0) {
          this.addLeaf();
     } else {
          _turtle.forward(trunkLen);
          _turtle.left(_branchAngle);
          this.draw(trunkLen - _trunkDecrement);
          _turtle.right(2 * _branchAngle);
          this.draw(trunkLen - _trunkDecrement);
          _turtle.left(_branchAngle);
          _turtle.back(trunkLen);
     }
}
```

TopHat Question

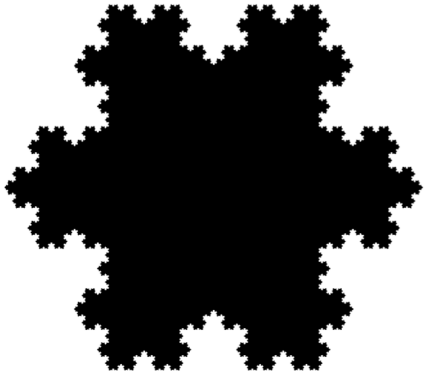
Given the following code:

```
private void draw(int trunkLen) {

   if (trunkLen <= 0) {
      this.addLeaf(); // creates a leaf
                      // at the current location
   } else {
      _turtle.forward(trunkLen);
      _turtle.left(_branchAngle);
      this.draw(trunkLen - _trunkDecrement);
      _turtle.right(2*_branchAngle);
      this.draw(trunkLen - _trunkDecrement);
      _turtle.left(_branchAngle);
      _turtle.back(trunkLen);
   }
}
```

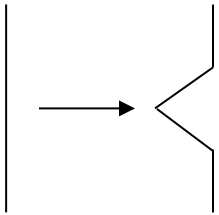
What would happen if you got rid of the first call `this.draw(trunkLen - _trunkDecrement)`?

A. We will only draw the right half of the tree
B. We will draw a spiral that terminates in a leaf
C. Stack Overflow!
D. None of the above

Recursive Snowflake



- Invented by Swedish mathematician, Helge von Koch, in 1904; also known as *Koch Island*
- Snowflake is created by taking an equilateral triangle and partitioning each side into three equal parts. Each side's middle part is then replaced by another equilateral triangle (with no base) whose sides are one third as long as the original.
  - this process is repeated for each remaining line segment
  - the user can specify the length of the initial equilateral triangle's side
  - "mathematical monster": infinite length with a bounded area



Snowflake's `draw` Method

- Can draw equilateral triangle iteratively
- `drawSnowFlake` draws the snowflake by drawing smaller, rotated triangles on each side of the triangle (compare to iterative `drawTriangle`)
- `for` loop iterates 3 times
- Each time, calls the `drawSide` helper method (defined in the next slide) and reorients `_turtle` to be ready for the next side

```
public void drawTriangle(int sideLen) {
    for (int i = 0; i < 3; i++) {
        _turtle.forward(sideLen);
        _turtle.right(120.0);
    }
}

public void drawSnowFlake(int sideLen){
    for(int i = 0; i < 3; i++){
        this.drawSide(sideLen);
         _turtle.right(120.0);
    }
}
```

Snowflake's `drawSide` method

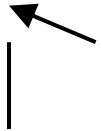
- `drawSide` draws single side of a recursive snowflake by drawing four recursive sides
- **Base case**: simply draw a straight side
- MIN_SIDE is a constant we set indicating the smallest desired side length
- **General case**: draw complete recursive side, then reorient for next recursive side

```
private void drawSide(int sideLen){
    if (sideLen < MIN_SIDE){
        _turtle.forward(sideLen);
    }
    else{
        this.drawSide(Math.round(sideLen / 3));
        _turtle.left(60.0);
        this.drawSide(Math.round(sideLen / 3));
        _turtle.right(120.0);
        this.drawSide(Math.round(sideLen / 3));
        _turtle.left(60.0);
        this.drawSide(Math.round(sideLen / 3));
     }
}
```

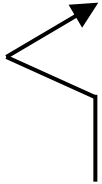
Hand Simulation

```
MIN_SIDE: 20
sideLen:  90
```

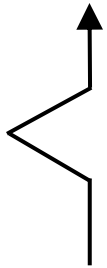
1) Call `draw(90)`, which calls `drawSide(90)`, which calls `drawSide(30)`, which calls `drawSide(10)`. Base case reached because 10 < `MIN_SIDE`



2) `drawSide(10)` returns to `drawSide(30)`, which tells `_turtle` to turn left 60 degrees and then calls `drawSide(10)` again.



3) `drawSide(10)` returns to `drawSide(30)`, which tells `_turtle` to turn right 120 and then calls `drawSide(10)` for a third time.

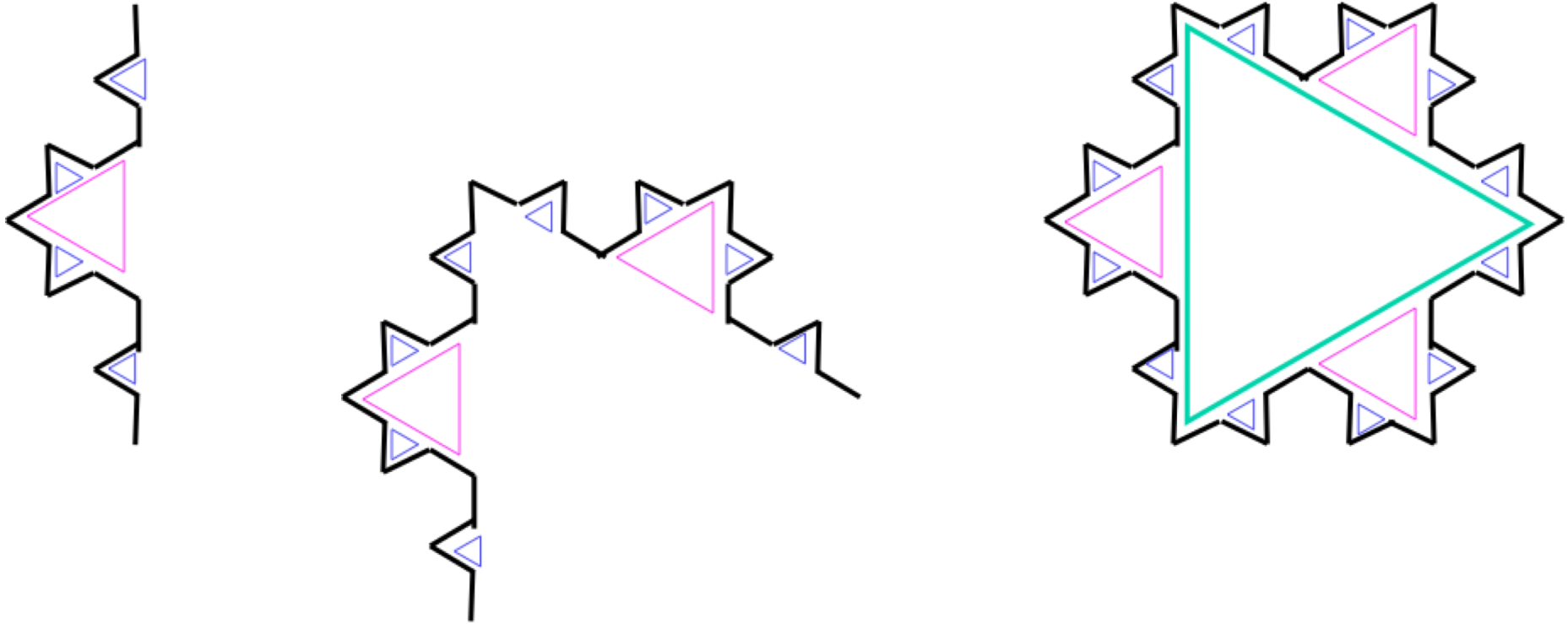


4) `drawSide(10)` returns to `drawSide(30)`, which tells `_turtle` to turn left 60 degrees and then calls `drawSide(10)` for a fourth time.

After this call, `drawSide(30)` returns to `drawSide(90)`, which reorients `_turtle` and calls `drawSide(30)` again.

Again: Koch Snowflake Progression

colored triangles added for emphasis only



Summary

- Recursion models problems that are self-similar, decomposing a task into smaller, similar sub-tasks.
- Whole task solved by combining solutions to sub-tasks (divide and conquer)
- Since every task related to recursion is defined in terms of itself, method will continue calling itself until it reaches its **base case**, which is simple enough to be solved directly

Announcements

- Design discussions for DoodleJump this week!
- DoodleJump due dates:
  - Early: 10/29, 11:59PM
  - On-time: 10/31, 11:59PM
  - Late: 11/2, 11:59PM