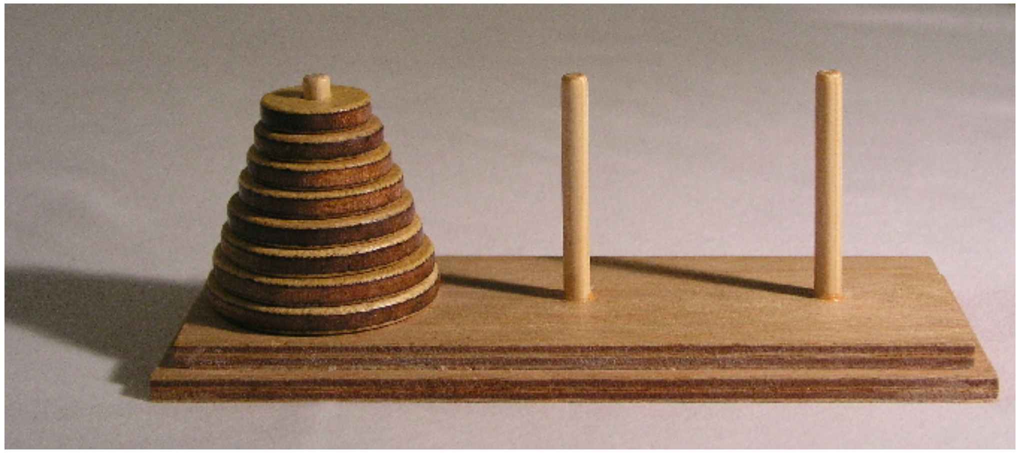
RECURSION (CONTINUED)

Lecture 9

CS2110 – Summer 2019

Plus…Tower of Hanoi

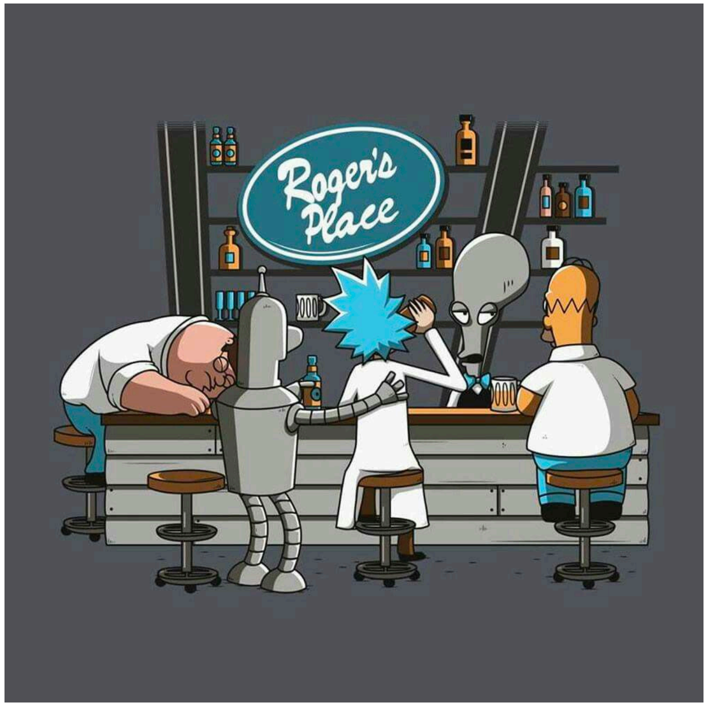


The objective of the puzzle is to move the entire stack to another rod, obeying the following simple rules:

1. Only one disk can be moved at a time.
2. Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack or on an empty rod.
3. No larger disk may be placed on top of a smaller disk.

Btw…another joke about recursion

Three logicians walked into the bar, and the waiter asked, "Does everyone wanna grab a beer?" The first logician said, "I don't know." The second said: "I don't know." The last one said: "Yes!"



AND…we will play a game after the class!

TOP SECRET!

Recap: **Understanding** Recursive Methods

1. Have a precise **specification**

2. Check that the method works in **the base case(s)**.

3. Look at the **recursive case(s)**. In your mind, replace each recursive call by what it does according to the spec and verify correctness.

4. (No infinite recursion) Make sure that the args of recursive calls are in some sense smaller than the pars of the method.

http://codingbat.com/java/Recursion-1

Problems with recursive structure

Code will be available on the course webpage.

1. exp - exponentiation, the slow way and the fast way
2. tile-a-kitchen – place L-shaped tiles on a kitchen floor
3. perms – list all permutations of a string
4. drawSierpinski – drawing the Sierpinski Triangle

Computing b^n for $n >= 0$

Power computation:

- $b^0 = 1$
- If $n != 0$, $b^n = b * b^{n-1}$
- If $n != 0$ and even, $b^n = (b*b)^{n/2}$

Judicious use of the third property gives far better algorithm

Example: $3^8 = (3*3) * (3*3) * (3*3) * (3*3) = (3*3)^4$

Computing b^n for n >= 0

Power computation:

- $b^0 = 1$
- If $n \neq 0$, $b^n = b\, b^{n-1}$
- If $n \neq 0$ and even, $b^n = (b*b)^{n/2}$

```
/** = b**n. Precondition: n >= 0 */
static int power(double b, int n) {
   if (n == 0) return 1;
   if (n%2 == 0) return power(b*b, n/2);
   return b * power(b, n-1);
}
```

Suppose n = 16
Next recursive call: 8
Next recursive call: 4
Next recursive call: 2
Next recursive call: 1
Then 0

16 = 2**4
Suppose n = 2**k
Will make k + 2 calls

Computing b^n for n >= 0

If n = 2**k
k is called the logarithm (to base 2)
of n: k = log n or k = log(n)

Suppose n = 16
Next recursive call: 8
Next recursive call: 4
Next recursive call: 2
Next recursive call: 1
Then 0

16 = 2**4
Suppose n = 2**k
Will make k + 2 calls

```
/** = b**n. Precondition: n >= 0 */
static int power(double b, int n) {
   if (n == 0) return 1;
   if (n%2 == 0) return power(b*b, n/2);
   return b * power(b, n-1);
}
```

Difference between linear and log solutions?

```
/** = b**n. Precondition: n >= 0 */
static double power(double b, int n) {
   if (n == 0) return 1;
   return b * power(b, n-1);
}
```

Number of recursive calls is n

```
/** = b**n. Precondition: n >= 0 */
static double power(double b, int n) {
   if (n == 0) return 1;
   if (n%2 == 0) return power(b*b, n/2);
   return b * power(b, n-1);
}
```

Number of recursive calls is ~ log n.

To show difference, we run linear version with bigger n until out of stack space. Then run log one on that n. See demo.

Table of log to the base 2

k	n = 2^k	log n (= k)
0	1	0
1	2	1
2	4	2
3	8	3
4	16	4
5	32	5
6	64	6
7	128	7
8	256	8
9	512	9
10	1024	10
11	2148	11
15	32768	15

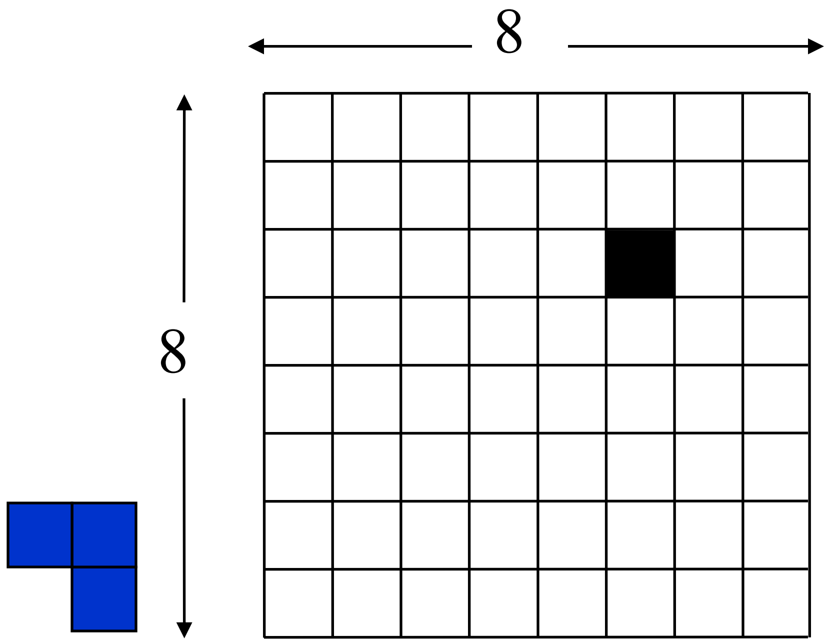
Tiling Elaine's kitchen

Kitchen in Gries's house: 8 x 8. Fridge sits on one of 1x1 squares

His wife, Elaine, wants kitchen tiled with el-shaped tiles –every square except where the refrigerator sits should be tiled.

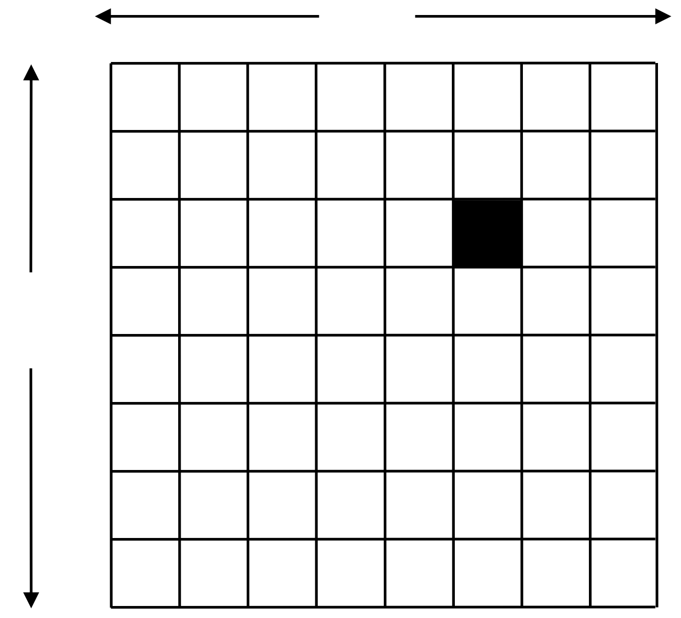
```
/** tile a 2^3 by 2^3 kitchen with 1
      square filled. */
public static void tile(int n)
```

We abstract away keeping track of where the filled square is, etc.

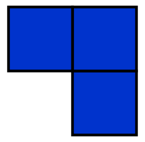


Tiling Elaine's kitchen

```
/** tile a 2^n by 2^n kitchen with 1
      square filled. */
public static void tile(int n) {

    if (n == 0) return;



}
```



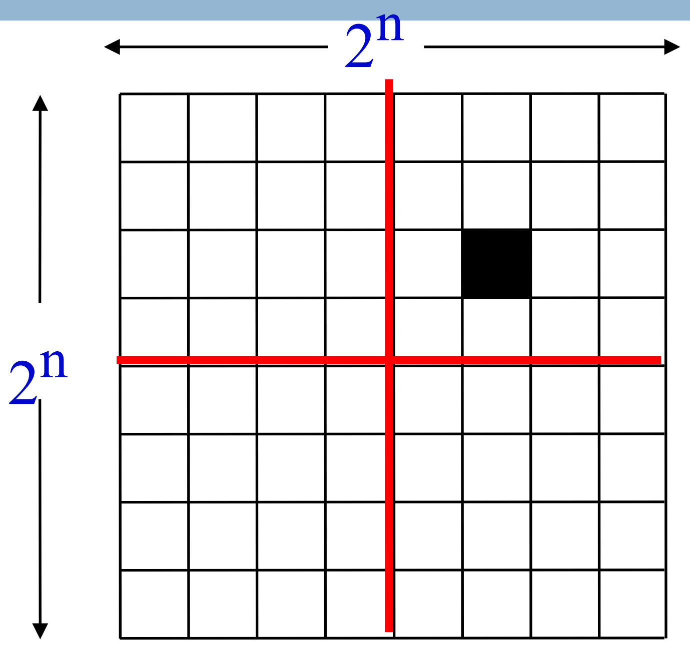
Base case?

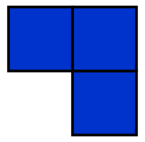


We generalize to a 2^n by 2^n kitchen

Tiling Elaine's kitchen

```
/** tile a 2^n by 2^n kitchen with 1
      square filled. */
public static void tile(int n) {

      if (n == 0) return;



}
```

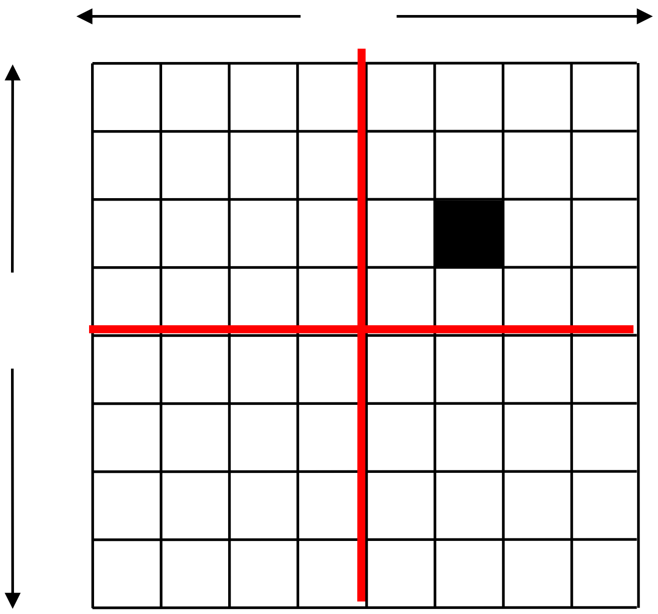


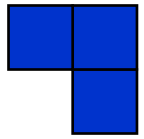


n > 0. What can we do to get kitchens of size 2^{n-1} by 2^{n-1}

Tiling Elaine's kitchen

```
/** tile a 2^n by 2^n kitchen with 1
      square filled. */
public static void tile(int n) {

     if (n == 0) return;



}
```





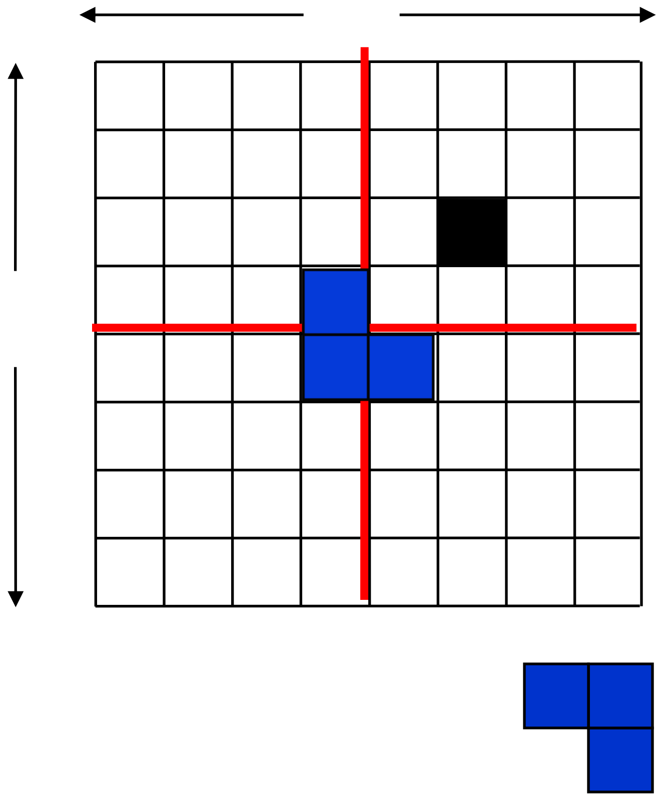
We can tile the upper-right 2^{n-1} by 2^{n-1} kitchen recursively.

But we can't tile the other three because they don't have a filled square.

What can we do? Remember, the idea is to tile the kitchen!

Tiling Elaine's kitchen

```
/** tile a 2^n by 2^n kitchen with 1
      square filled. */
public static void tile(int n) {
    if (n == 0) return;
    Place one tile so that each kitchen
    has one square filled;

    Tile upper left kitchen recursively;
    Tile upper right kitchen recursively;
    Tile lower left kitchen recursively;
    Tile lower right kitchen recursively;
}
```



Permutations of a String

`perms(abc): abc, acb, bac, bca, cab, cba`

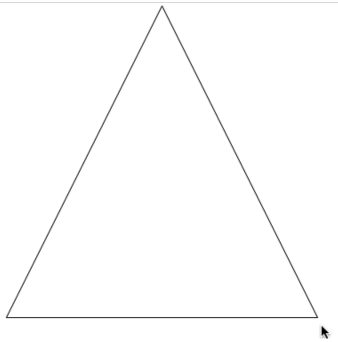
a**bc** a**cb**
b**ac** b**ca**
c**ab** c**ba**

Recursive definition:

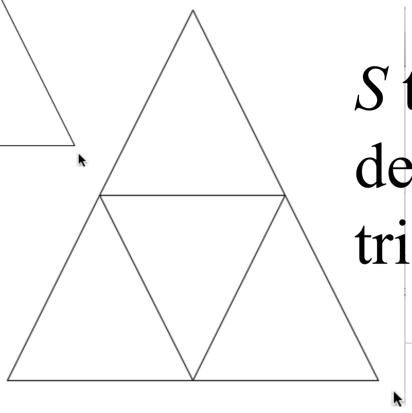
Each possible first letter, followed by all permutations of the remaining characters.

AND… a triangle example!

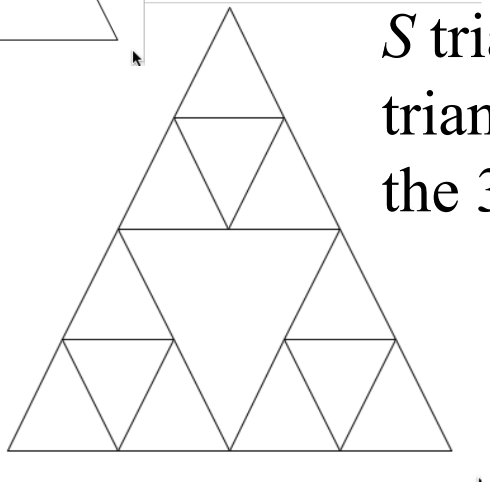
Sierpinski triangles



S triangle of depth 0



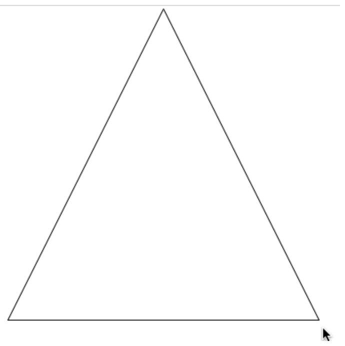
S triangle of depth 1: 3 S triangles of depth 0 drawn at the 3 vertices of the triangle



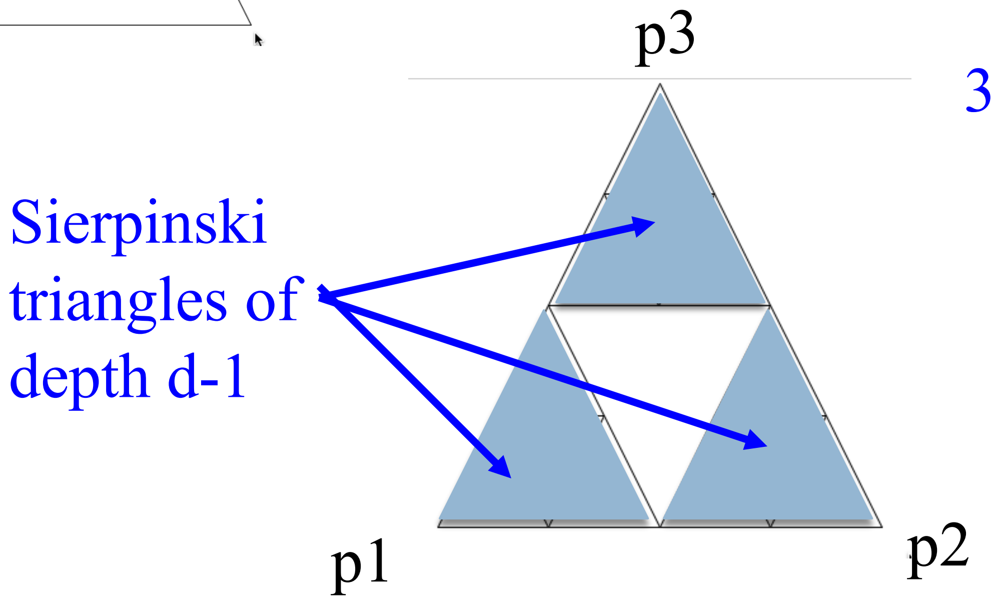
S triangle of depth 2: 3 S triangles of depth 1 drawn at the 3 vertices of the triangle

Sierpinski triangles

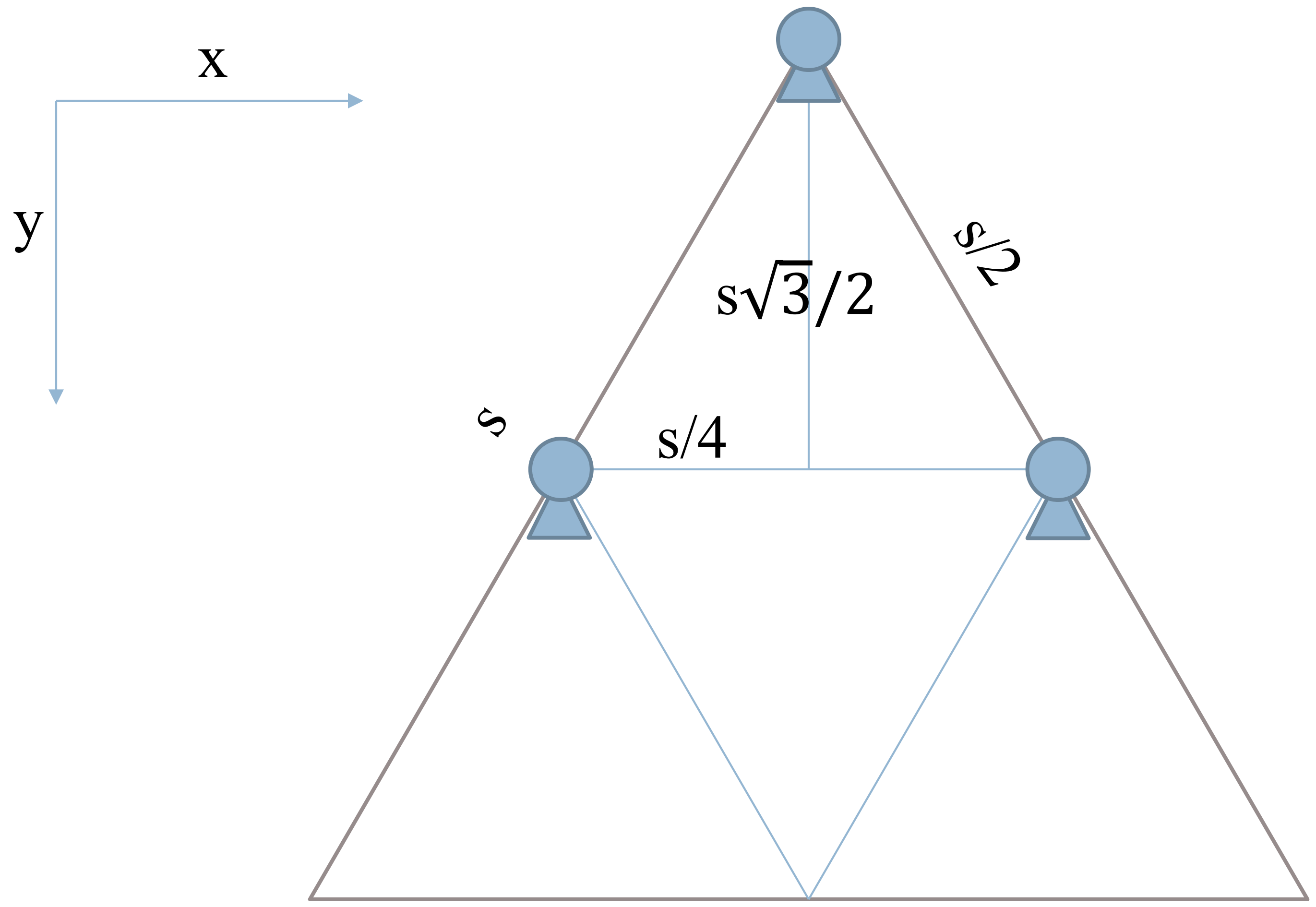
S triangle of depth 0: the triangle



S triangle of depth d at points p1, p2, p3:
3 S triangles of depth d-1 drawn at at p1, p2, p3



Sierpinski triangles



Conclusion

Recursion is a convenient and powerful way to define functions

Problems that seem insurmountable can often be solved in a "divide-and-conquer" fashion:

- Reduce a big problem to smaller problems of the same kind, solve the smaller problems
- Recombine the solutions to smaller problems to form solution for big problem

http://codingbat.com/java/Recursion-1

The Fibonacci Function: is there a more efficient way?

Mathematical definition:

fib(0) = 0
fib(1) = 1
fib(n) = fib(n – 1) + fib(n – 2) $n \geq 2$

two base cases!

Fibonacci sequence: 0 1 1 2 3 5 8 13 …

```
/** = fibonacci(n). Pre: n >= 0 */
static int fib(int n) {
   if (n <= 1) return n;
   // { 1 < n }
   return fib(n-1) + fib(n-2);
}
```

Fibonacci (Leonardo Pisano) 1170-1240?

Statue in Pisa Italy
Giovanni Paganucci
1863

AND…we will play a game after the class!

Five volunteers?

How do we divide the coins?

Five pirates (A,B,C,D,E)

100 GOLD Coins

A>B>C>D>E

1. most senior pirate first proposes a plan of distribution
2. VOTE TIME. If majority accepts, game ends. If tie, the proposer has the casting vote. If majority rejects, the proposer will be thrown Out!
3. Comes to the next most senior pirate recursion!
4. The process repeats until a plan is accepted or if there is one pirate left.

Some assumptions

1. Everyone wants to survive!
2. Given survival, maximize the coins u get!
3. prefer to throw others overboard!
4. Everyone is super super super smart!
   (just as smart as u guys!)

So what would u do if u are A?