RECURSION (CONTINUED)

Lecture 9

CS2110 – Fall2016

Overview references to sections in text

- What is recursion? 7.1-7.39   slide 1-7
- Base case   7.1-7.10 slide 13
- How Java stack frames work 7.8-7.10 slide 28-32

Gries slides on A2 function evaluate and evaluate itself
are on pinned Piazza note Supplementary Material

```
// invariant: p = product of c[0..k-1]
```

what's the product when k == 0?

Why is the product of an empty bag of values 1?

Suppose bag b contains 2, 2, 5 and p is its product: 20.
Suppose we want to add 4 to the bag and keep p the product.
We do:

insert 4 in the bag;
p= 4 * p;

Suppose bag b is empty and p is its product: what value?.
Suppose we want to add 4 to the bag and keep p the product.
We want to do the same thing:

insert 4 in the bag;
p= 4 * p;

For this to work, the product of the empty bag has to be 1,
since 4 * 1 = 4

same thing in python

0 is the identity of + because	0 + x = x
1 is the identity of * because	1 * x = x
false is the identity of || because	false || b = b
true is the identity of && because	true && b = b
1 is the identity of gcd because	gcd({1, x}) = x

For any such operator **o**, that has an identity,
o of the empty bag is the identity of **o**.

Sum of the empty bag = 0

Product of the empty bag = 1

OR (||) of the empty bag = false.

gcd of the empty bag = 1

gcd: greatest common divisor of the elements of the bag

Primitive vs Reference Types

Primitive Types:	Reference Types:
`char`	`Object`
`boolean`	`JFrame`
`int`	`String`
`float`	`PHD`
`double`	`int[]`
`byte`	`Animal`
`short`	`Animal[]`
`long`	`...` (*everything* else!)

A variable of the type contains:

	the value of the variable is reference
A value of that type	A *reference to an object* of that type

`==` vs `equals`

Once you understand primitive vs reference types, there are only two things to know:

`a == b` compares a and b's **values**

may cause problem when a and b are reference object

`a.equals(b)` compares the two *objects* using the `equals` method

== vs `equals`: Reference types

For reference types, p1 == p2 determines whether p1 and p2 contain the same reference (i.e., point to the same object or are both null).

p1.equals(p2) tells whether the objects contain the same information (as defined by whoever implemented equals).

```
Pt a0 = new Pt(3,4);
Pt a1 = new Pt(3,4);
```

p1 $\boxed{\text{a0}}$

p2 $\boxed{\text{a0}}$

p3 $\boxed{\text{a1}}$

p4 $\boxed{\text{null}}$

same reference for p2 and p1: all a0

different reference for p3 and p1: p3 have a1 and p1 have a0

p2 == p1	true	p2.equals(p1)	true
p3 == p1	false	p3.equals(p1)	true
p4 == p1	false	p4.equals(p1)	**NullPointerException!**

cannot call a equals method on a null object, will cause NullPointerException; but if we do p1.equals(p4), we will get a FALSE but not the exception

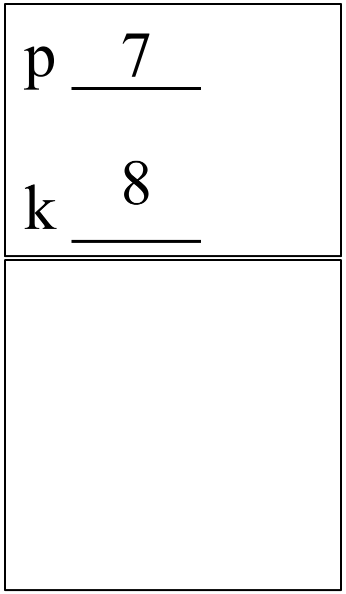
the primitive type can not call a .equals method

Recap: **Executing** Recursive Methods

- 1. Push frame for call onto call stack.
- 2. Assign arg values to pars.
- 3. Execute method body.
- 4. Pop frame from stack and (for a function) push return value on the stack.
- For function call: When control given back to call, pop return value, use it as the value of the function call.

```
public int m(int p) {
    int k= p+1;
    return p;
}
```

m(5+2)



call stack

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

Let's write some recursive methods!

Code will be available on the course webpage.

1. len – length of a string without .length()
2. dup – repeat each character in a string twice
3. isPal – check whether a string is a palindrome
4. rev – reverse a string

Check palindrome-hood

A String palindrome is a String that reads the same backward and forward:

isPal("racecar") → true isPal("pumpkin") → false

A String with at least two characters is a palindrome if

- (0) its first and last characters are equal and
- (1) chars between first & last form a palindrome:

have to be the same

e.g. AMANAPLANACANALPANAMA

have to be a palindrome

A recursive definition!

- A man a plan a caret a ban a myriad a sum a lac a liar a hoop a pint a catalpa a gas an oil a bird a yell a vat a caw a pax a wag a tax a nay a ram a cap a yam a gay a tsar a wall a car a luger a ward a bin a woman a vassal a wolf a tuna a nit a pall a fret a watt a bay a daub a tan a cab a datum a gall a hat a fag a zap a say a jaw a lay a wet a gallop a tug a trot a trap a tram a torr a caper a top a tonk a toll a ball a fair a sax a minim a tenor a bass a passer a capital a rut an amen a ted a cabal a tang a sun an ass a maw a sag a jam a dam a sub a salt an axon a sail an ad a wadi a radian a room a rood a rip a tad a pariah a revel a reel a reed a pool a plug a pin a peek a parabola a dog a pat a cud a nu a fan a pal a rum a nod an eta a lag an eel a batik a mug a mot a nap a maxim a mood a leek a grub a gob a gel a drab a citadel a total a cedar a tap a gag a rat a manor a bar a gal a cola a pap a yaw a tab a raj a gab a nag a pagan a bag a jar a bat a way a papa a local a gar a baron a mat a rag a gap a tar a decal a tot a led a tic a bard a leg a bog a burg a keel a doom a mix a map an atom a gum a kit a baleen a gala a ten a don a mural a pan a faun a ducat a pagoda a lob a rap a keep a nip a gulp a loop a deer a leer a lever a hair a pad a tapir a door a moor an aid a raid a wad an alias an ox an atlas a bus a madam a jag a saw a mass an anus a gnat a lab a cadet an em a natural a tip a caress a pass a baronet a minimax a sari a fall a ballot a knot a pot a rep a carrot a mart a part a tort a gut a poll a gateway a law a jay a sap a zag a fat a hall a gamut a dab a can a tabu a day a batt a waterfall a patina a nut a flow a lass a van a mow a nib a draw a regular a call a war a stay a gam a yap a cam a ray an ax a tag a wax a paw a cat a valley a drib a lion a saga a plat a catnip a pooh a rail a calamus a dairyman a bater a canal Panama

Problems with recursive structure

Code will be available on the course webpage.

1. exp - exponentiation, the slow way and the fast way
2. perms – list all permutations of a string
3. tile-a-kitchen – place L-shaped tiles on a kitchen floor
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 != 0, $b^n = b\ b^{n-1}$
- If n != 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)

```
/** = 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

Difference between linear and log solutions?


```
/** = b**n. Precondition: n >= 0 */
static int 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 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);
}
```

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

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.

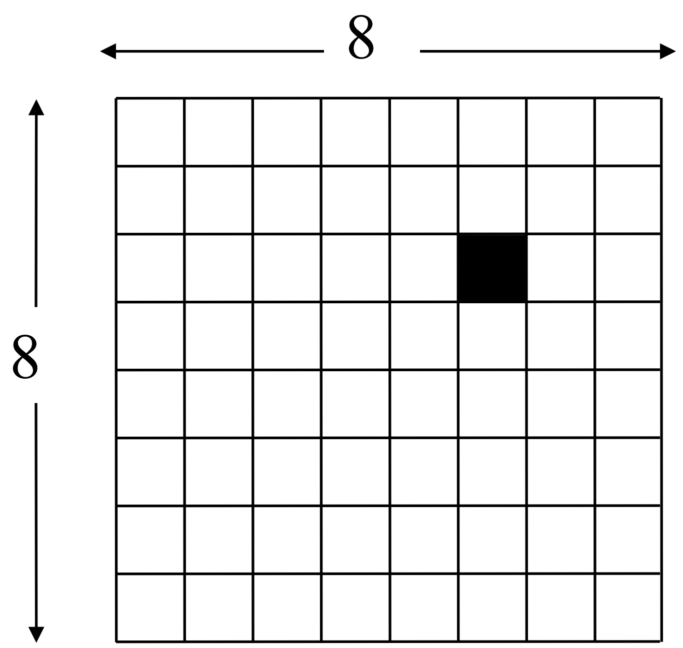
Tiling Elaine's kitchen

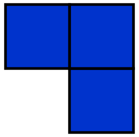
Kitchen in Gries's house: 8 x 8. Fridge sits on one of 1x1 squares

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)

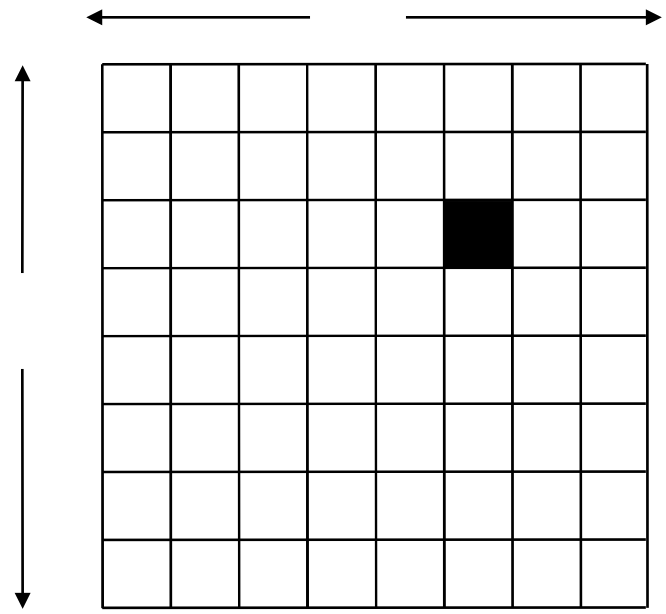
We abstract away keeping track of where the filled square is, etc.



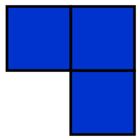


Tiling Elaine's kitchen


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

    if (n == 0) return;



}
```

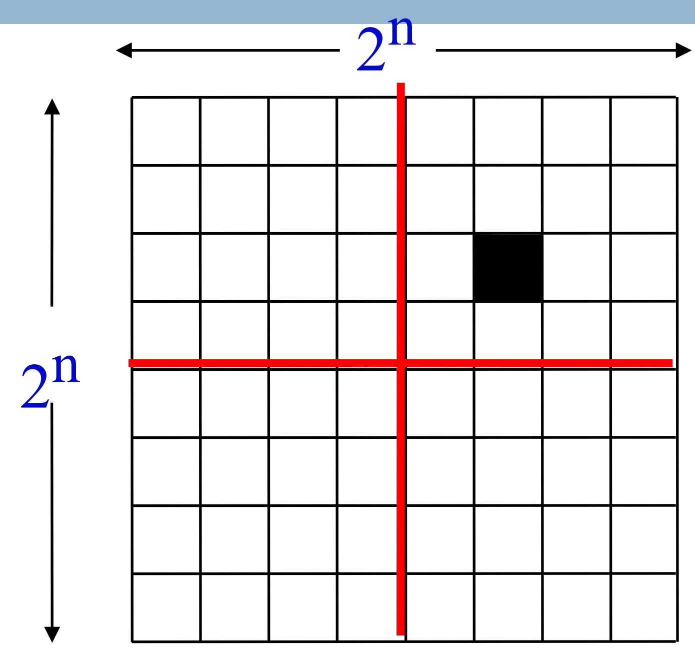


Base case?

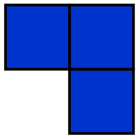


We generalize to a 2^n by 2^n kitchen

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;

}
```



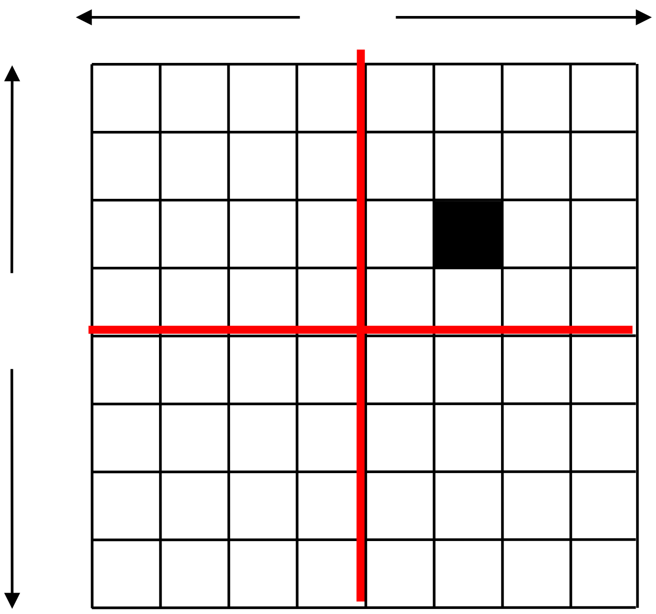
n > 0. What can we do to get kitchens of size 2^{n-1} by 2^{n-1}

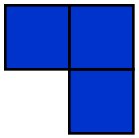
Tiling Elaine's kitchen

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

if (n == 0) return;

}





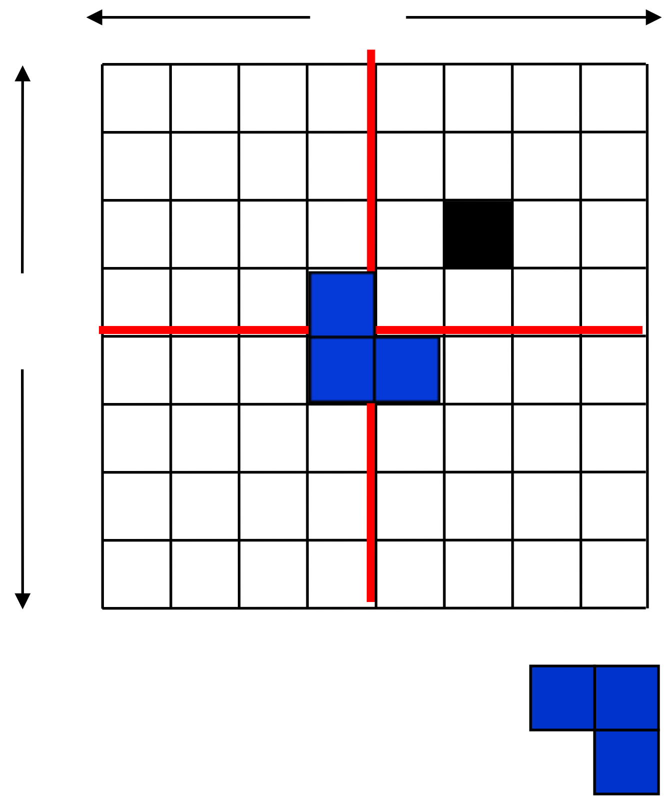
We can tile the upper-right 2^{n-1} by 2^{n-1} kitchen recursively.

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

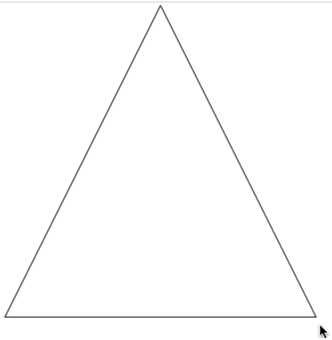
What can we do? Remember, the idea is to tile the kitchen!

Tiling Elaine's kitchen

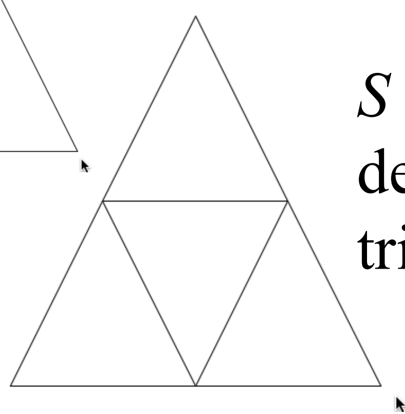

```
/** tile a 2^n by 2^n kitchen with 1
      square filled. */
public static void tile(int n) {
    if (n == 0) return;
    Place one tile so that each kitchen
    has one square filled;

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



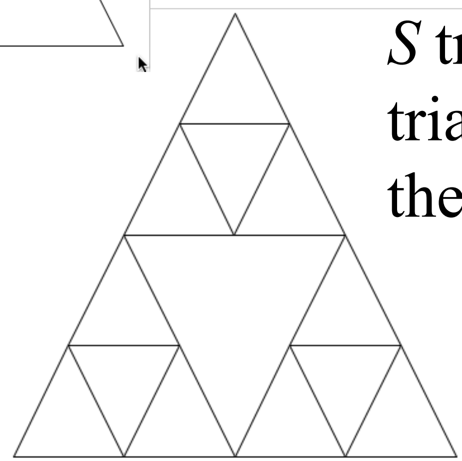
Sierpinski triangles



S triangle of depth 0



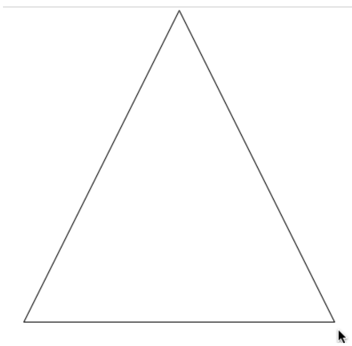
S triangle of depth 1: 3 S triangles of depth 0 drawn at the 3 vertices of the triangle



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

Sierpinski triangles

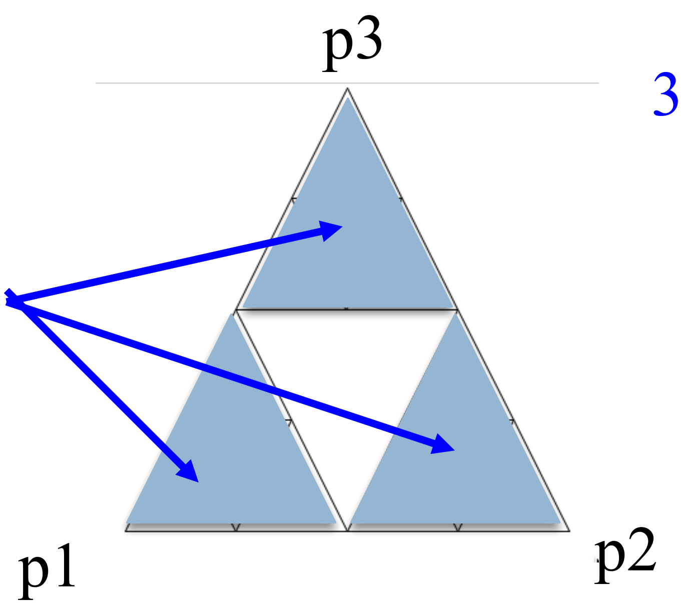
S triangle of depth 0: the triangle



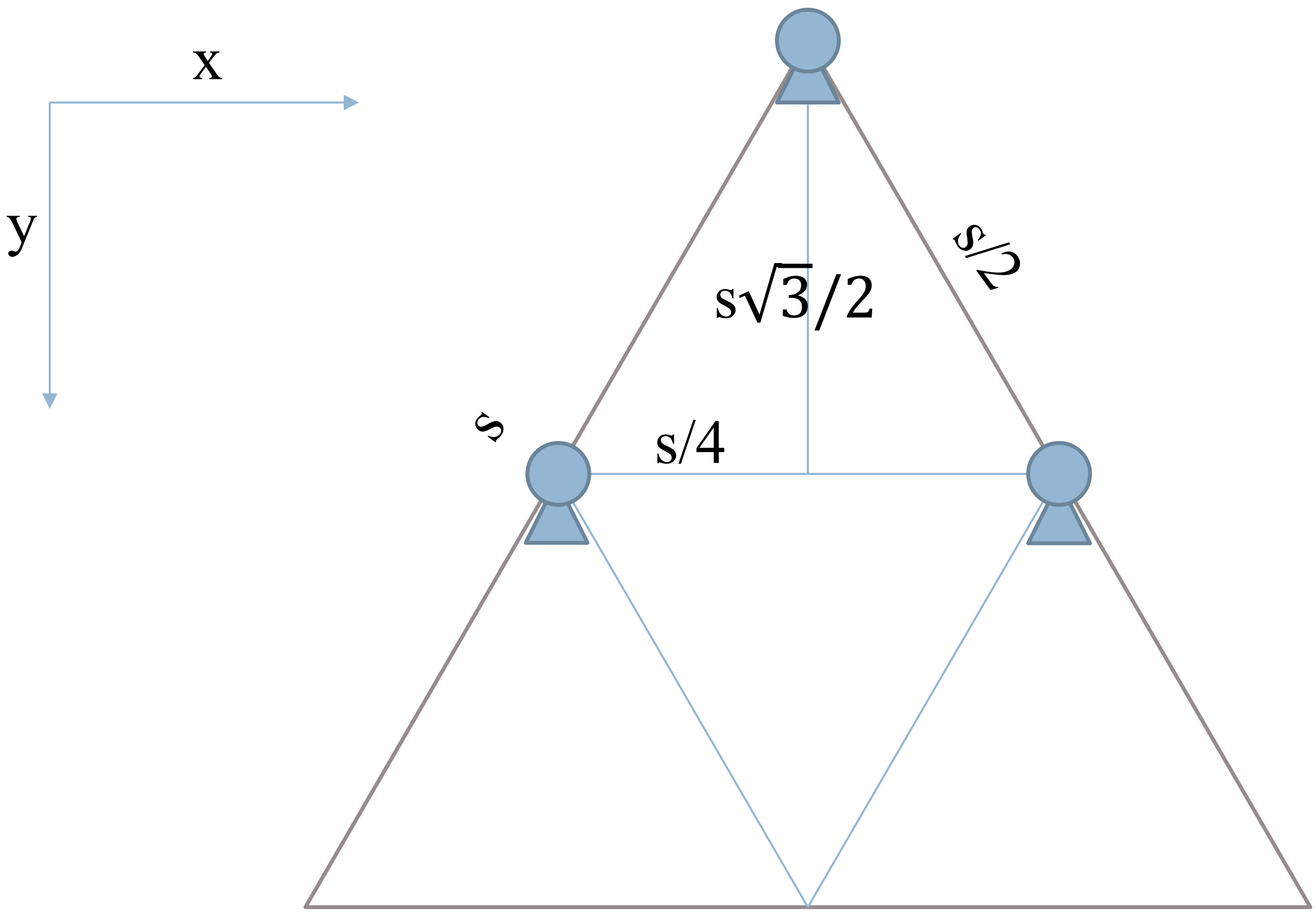
S triangle of depth d at points p1, p2, p3:

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

Sierpinski triangles of depth d-1



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