RECURSION (CONTINUED)

Lecture 9

CS2110 – Spring 2018

Prelim two weeks from today: 13 March.

1. Visit Exams page of course website, check what time your prelim is, complete assignment P1Conflict ONLY if necessary.
2. Review session Sunday, 11 March, 1-3PM.
3. A3 is due 2 days from now, on Thursday.
4. If appropriate, please check the JavaHyperText before posting a question on the Piazza. You can get your answer instantaneously rather than have to wait for a Piazza answer.
   Example:“default”, “access”, “modifier”, “private” are well-explained the JavaHyperText .

```
// 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:

put 4 into 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 do the same thing:

put 4 into the bag;
p= 4 * p;

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

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

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. 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

Number of recursive calls is ~ 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);
}
```

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

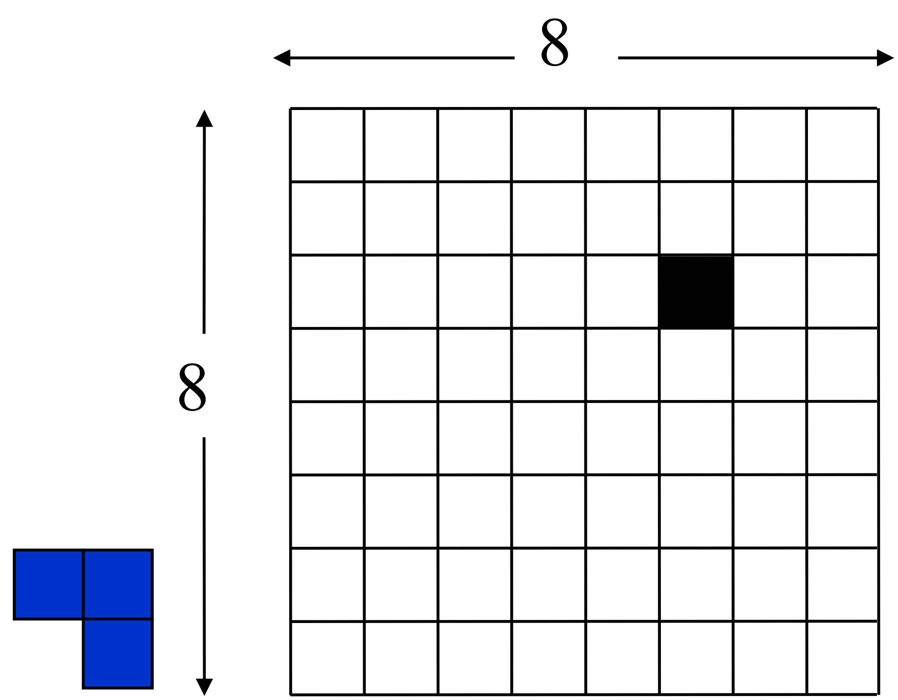
Kitchen in Gries's house: 8 x 8. Fridge sits on one of 1x1 squares

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.

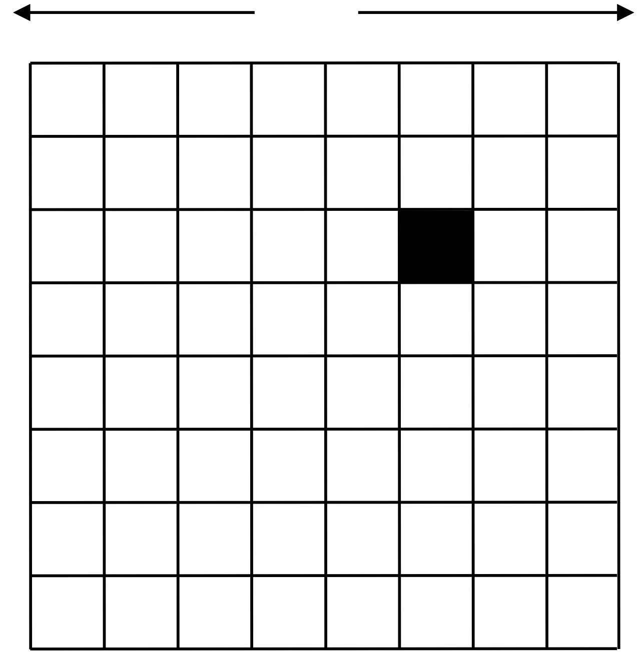


Tiling Elaine's kitchen

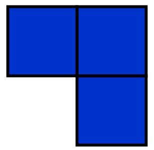
/** tile a 2^n by 2^n kitchen with 1
square filled. */
public static void tile(int n) {

if (n == 0) return;

}



Base case?



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

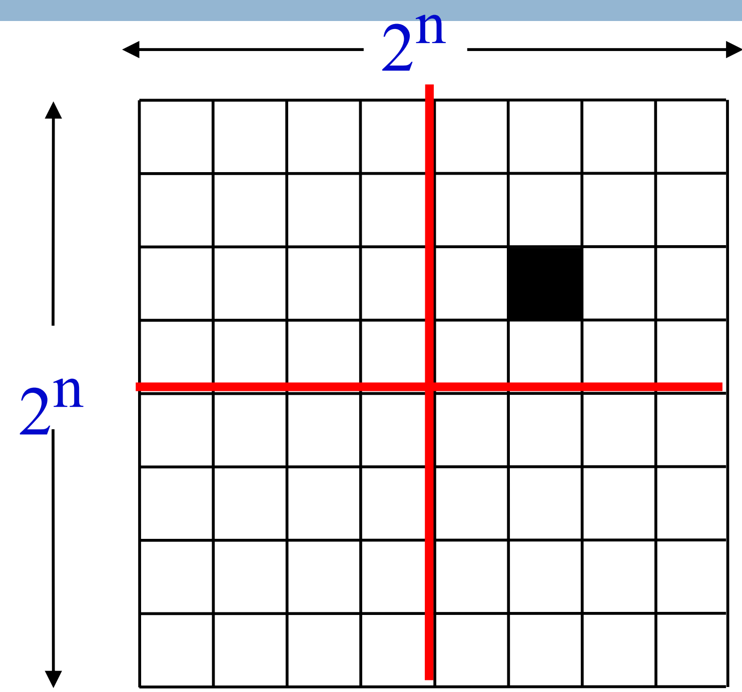
Tiling Elaine's kitchen

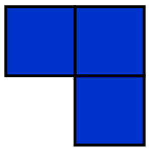

```
/** tile a 2^n by 2^n kitchen with 1
      square filled. */
public static void tile(int n) {

    if (n == 0) return;



}
```





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

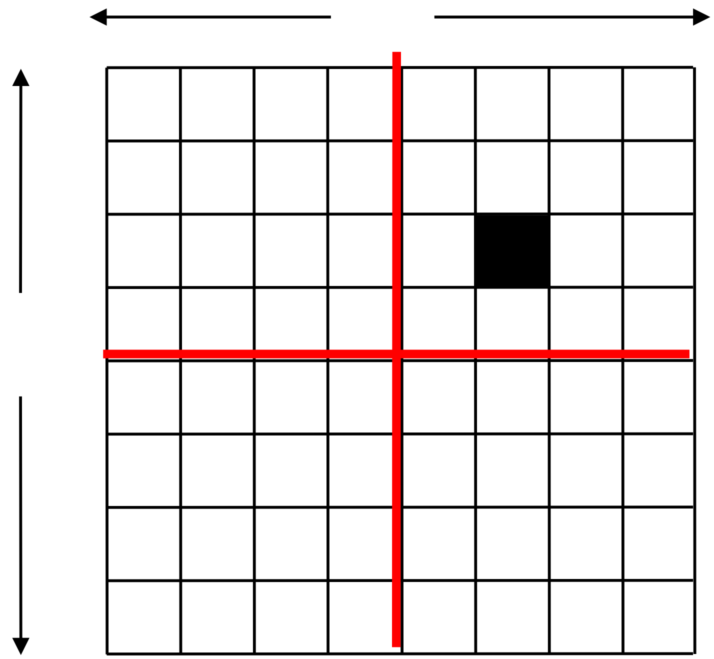
Tiling Elaine's kitchen

/** tile a 2^n by 2^n kitchen with 1 square filled. */

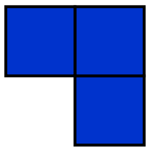
```
public static void tile(int n) {

    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!

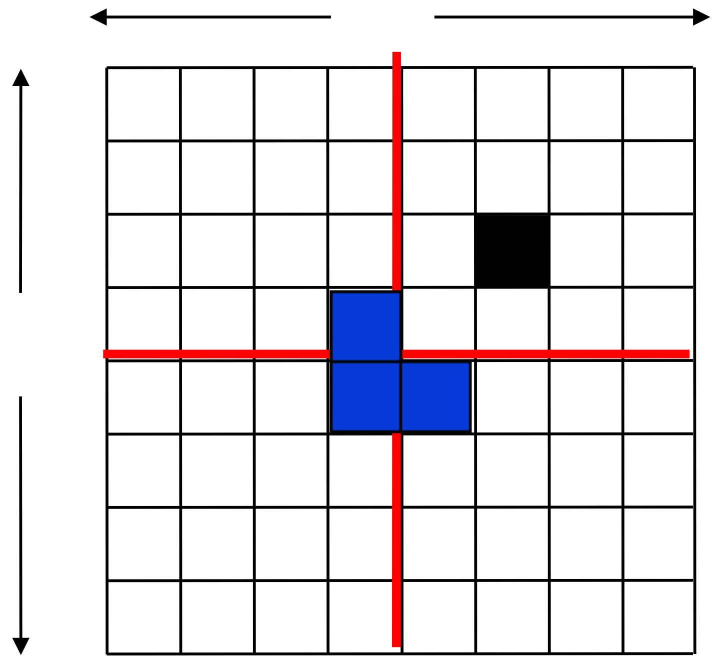
Tiling Elaine's kitchen

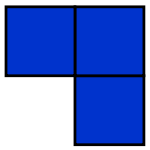
/** tile a 2^n by 2^n kitchen with 1
square filled. */
public static void tile(int n) {

if (n == 0) return;
Place one tile so that each kitchen
has one square filled;

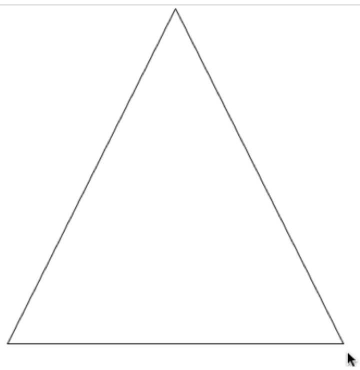
Tile upper left kitchen recursively;
Tile upper right kitchen recursively;
Tile lower left kitchen recursively;
Tile lower right kitchen recursively;

}

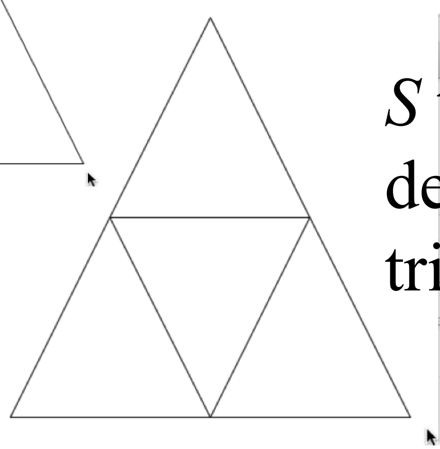




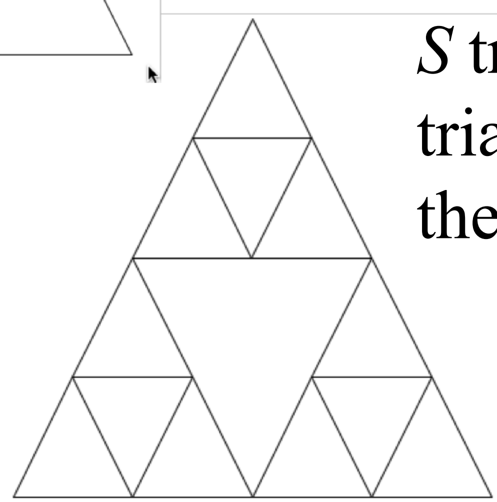
Sierpinski triangles



S triangle of depth 0



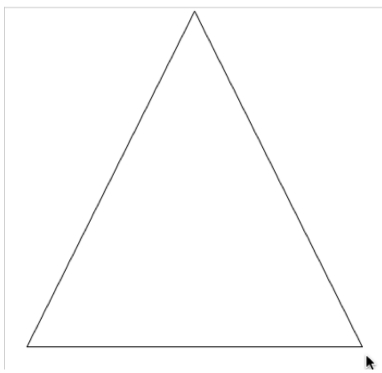
S triangle of depth 1: 3 S triangles of depth 0 drawn at the 3 vertices of the triangle



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

Sierpinski triangles

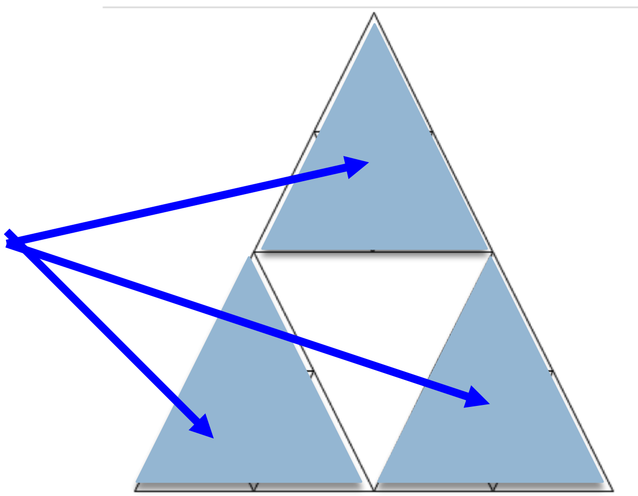
S triangle of depth 0: the triangle



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

p3

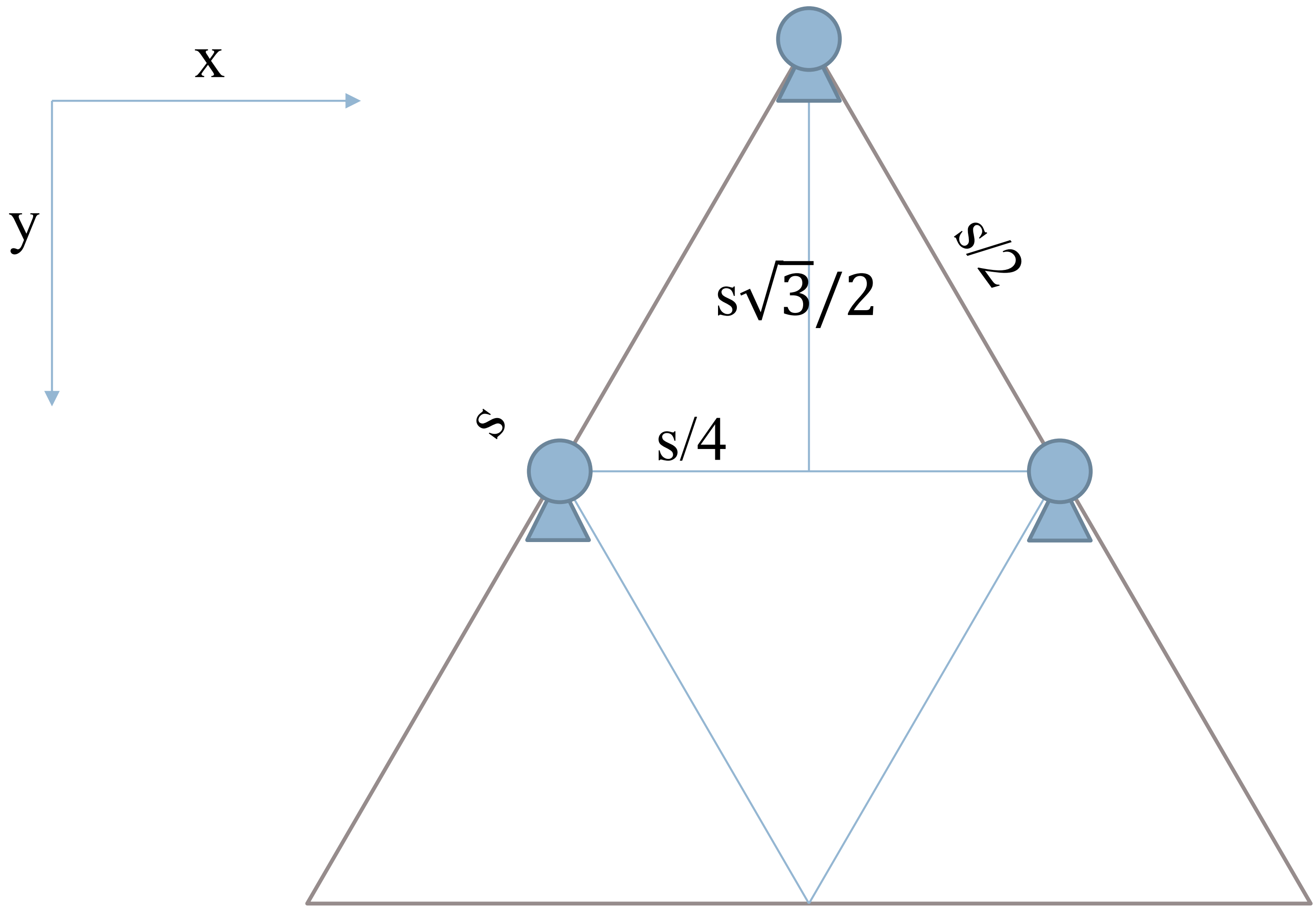
Sierpinski triangles of depth d-1



p1

p2

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