RECURSION (CONTINUED)

Lecture 9

CS2110 – Spring 2016

Overview references to sections in text

- Note: We've covered everything in JavaSummary.pptx!
- 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

Solutions to exception-handling problem set are on lecture-notes page of course website.
Look at row for recitation 3

Today, we continue discussing recursion

About prelim 1

1. 5:30-7PM. Kennedy 116: last name begins with A..Lib.

2. 7:30-9:PM. Kennedy 116: last name begins with Lie..Z.

3. Gates 405 if authorized to have quiet room or more time. Complete P1Conflict on the CMS. Issue with this, see point 5.

4. Conflict with scheduled time but can make other one: Complete P1Conflict. Go to other prelim.

5. All other conflicts (e.g. won't be in town). Email Megan Gatch mlg34@cornell.edu. State name, netid, conflict clearly and thoroughly. Don't complete P1Conflict. We'll get back to you.

Complete P1Conflict (if you have to) by end of 9 March.

Hoare triple for if-statement

What do we need to know this is true? {Q} if (B) S {R}

Many of you wrote this:

But what if B is false? Doesn't R still have to be true after execution of the if-statement?

If {Q && B} S {R} and Q && !B => R

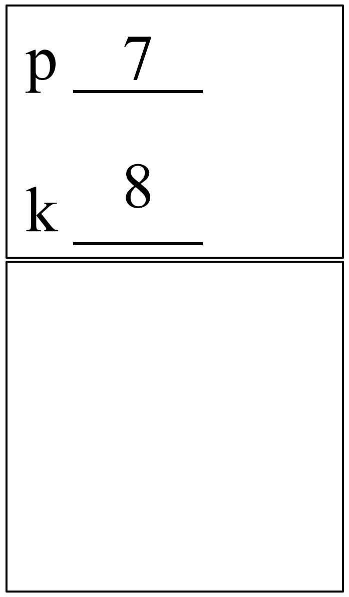
then {Q} if (B) S {R}

Summary of method call execution:

- 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)



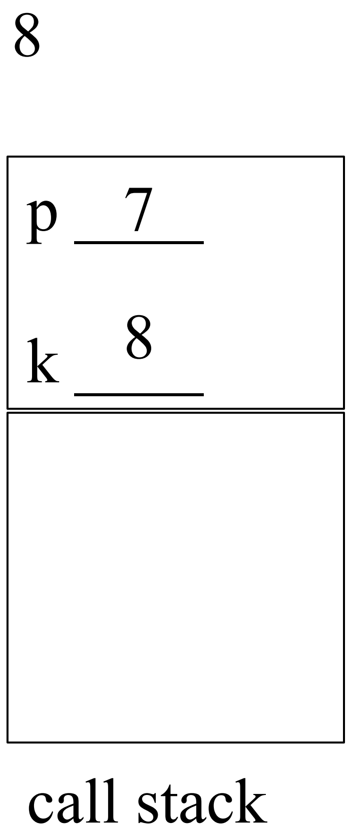
call stack

Summary of method call execution:

- 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 k;
}
```

m(5+2)

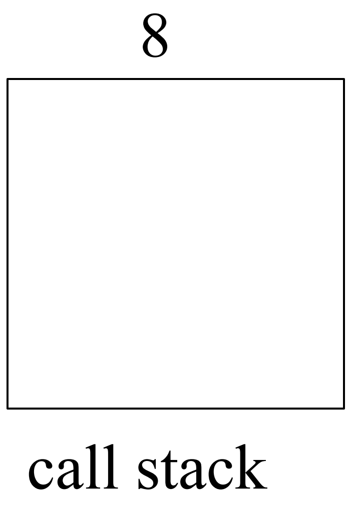


Summary of method call execution:

- 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 k;
}
```

m(5+2)



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

The Fibonacci Function

Mathematical definition:

$$\left.\begin{array}{l} fib(0) = 0 \\ fib(1) = 1 \end{array}\right\} \text{two base cases!}$$

$fib(n) = fib(n - 1) + fib(n - 2), \quad n \geq 2$

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-2) + fib(n-1);
}
```

Fibonacci (Leonardo Pisano) 1170-1240?

Statue in Pisa, Italy
Giovanni Paganucci
1863

Example: Count the e's in a string

```
/** =  number of times c occurs in s */
public static int countEm(char c, String s) {
   if (s.length() == 0) return 0;

   // { s has at least 1 character }
   if (s.charAt(0) != c)
        return countEm(c, s.substring(1));

   // { first character of s is c}
   return 1 + countEm (c, s.substring(1));
}
```

substring s[1..], i.e. s[1], …, s(s.length()-1)

- countEm('e', "it is easy to see that this has many e's") = 4
- countEm('e', "Mississippi") = 0

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:

- $a^0 = 1$
- 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 a, double 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 a, double n) {
  if (n == 0) return 1;
  if (n%2 == 0) return power(b*a, b/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

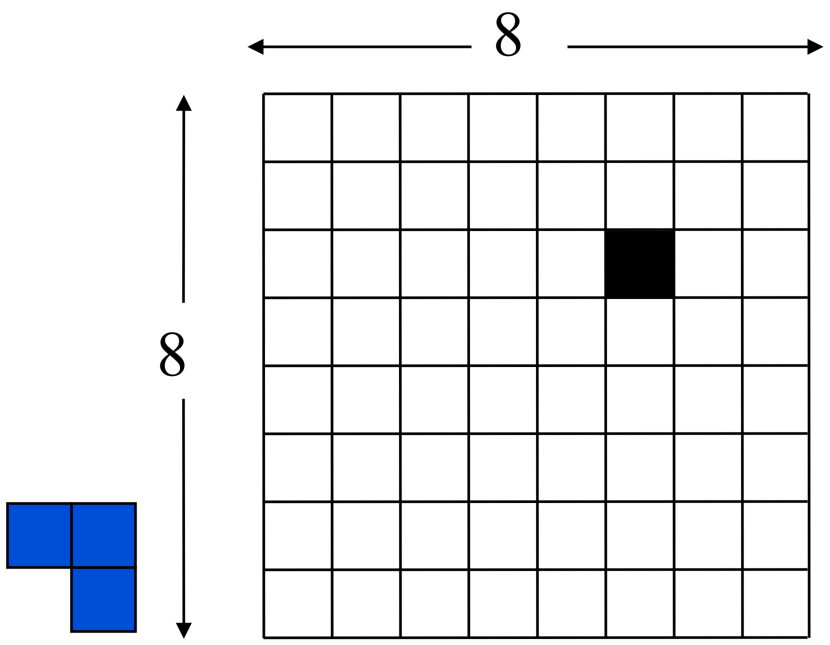
Tiling Elaine's kitchen

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.

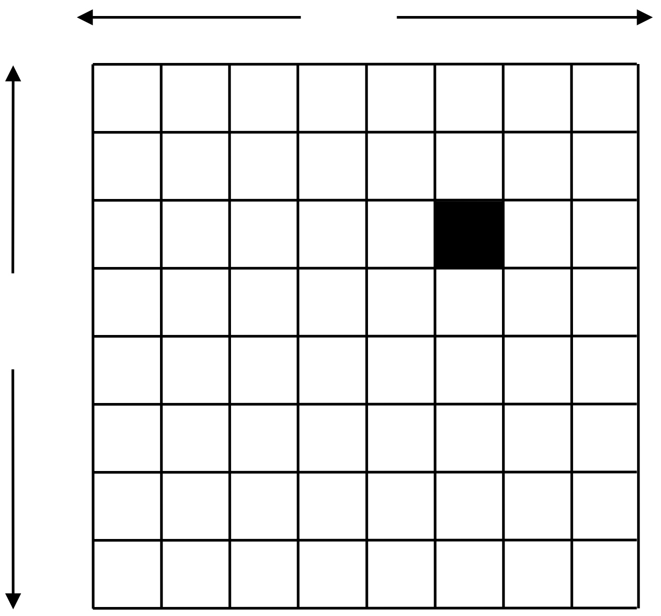
```
/** tile a 2^3 by 2^3 kitchen with 1
    square filled. */
public static void tile(int n)
```

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

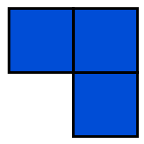


Tiling Elaine's kitchen

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

    if (n == 0) return;



}
```



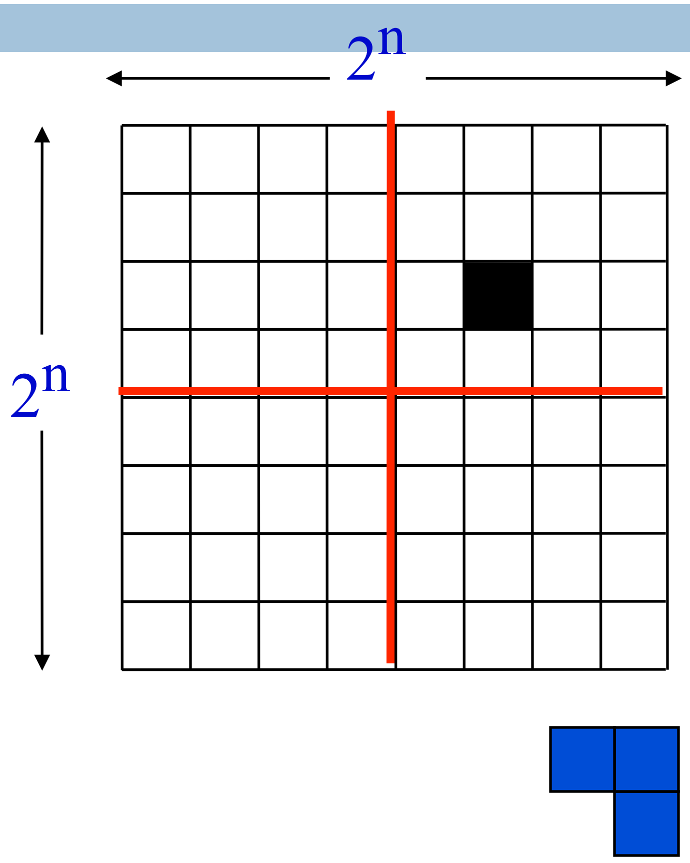
Base case?



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

Tiling Elaine's kitchen

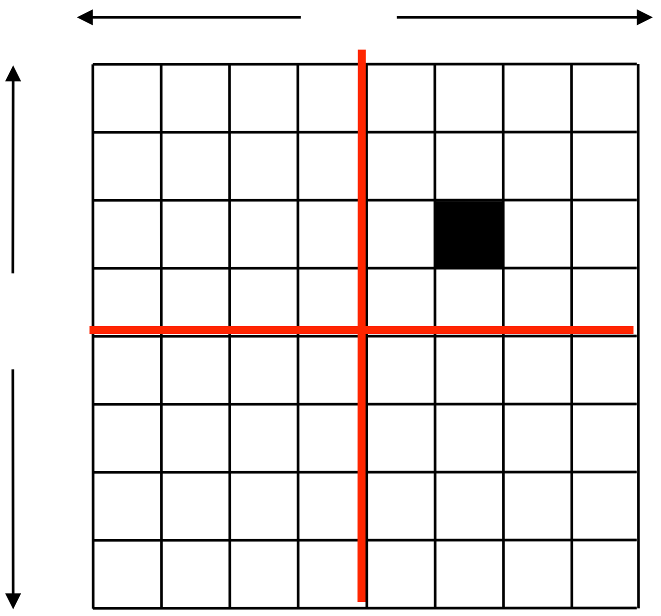
```
/** tile a 2^n by 2^n kitchen with 1
    square filled. */
public static void tile(int n) {

    if (n == 0) return;




}
```

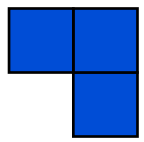


n > 0. What can we do to get kitchens of size 2^{n-1} by 2^{n-1}

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;



 }
```

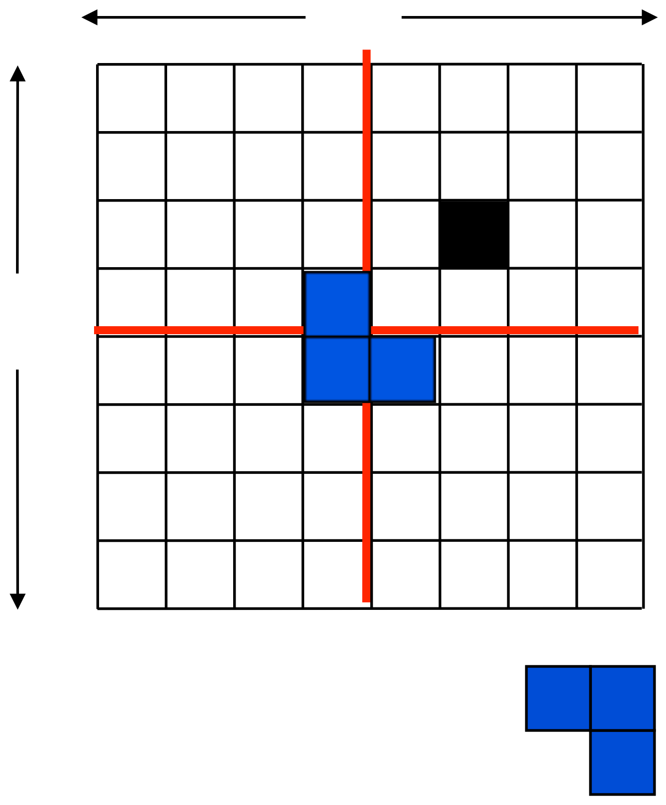




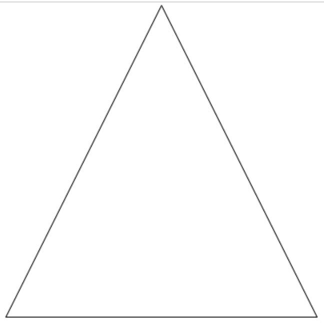
We can tile the upper-right 2^{n-1} by 2^{n-1} kitchen recursively.
But we can't tile the other three because they don't have a filled square.
What can we do? Remember, the idea is to tile the kitchen!

Tiling Elaine's kitchen

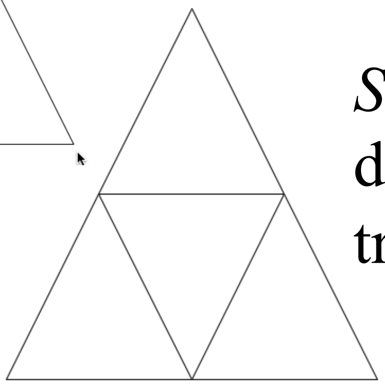
```
/** tile a 2^n by 2^n kitchen with 1
      square filled. */
public static void tile(int n) {
    if (n == 0) return;
    Place one tile so that each kitchen
    has one square filled;

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



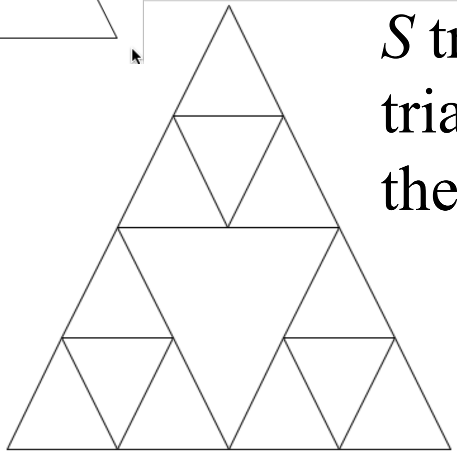
Sierpinski triangles



S triangle of depth 0



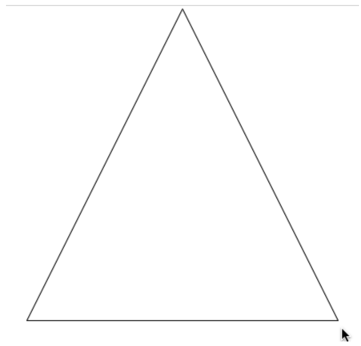
S triangle of depth 1: 3 S triangles of depth 0 drawn at the 3 vertices of the triangle



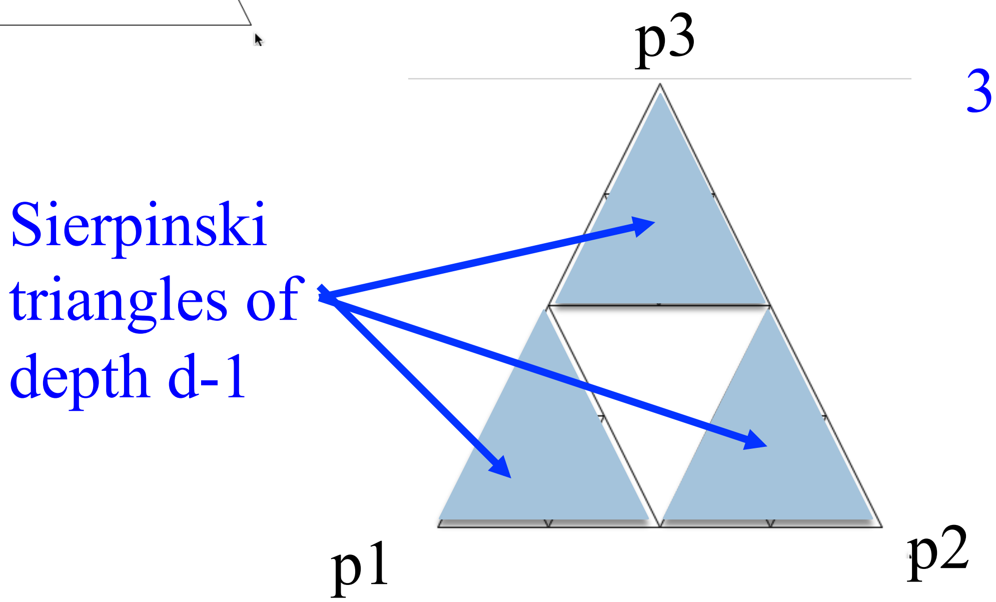
S triangle of depth 2: 3 S triangles of depth 1 drawn at the 3 vertices of the triangle

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



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