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

Solutions to recitation problem sets. See Piazza Supplemental Study Material

TA midsemester evaluation coming! PLEASE help us and complete the evaluations! A chance to help TAs and YOU this semester

Prelim a week from Thursday. Look here for information:

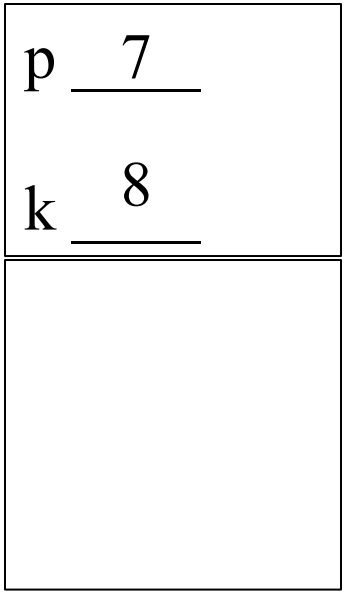
http://www.cs.cornell.edu/courses/CS2110/2016fa/exams.html

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

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

deduplicate


```
/** Return shortest substring x of s such that s = x + x + ··· + x
public static String deduplicate(String s) {}
```

Many solutions far too complicated! We show how to do this simply, using recursion –next lecture, using iteration.

deduplicate

This is a linear search in 1..s.length() for smallest something. It's guaranteed to succeed because x = s is a solution, but not necessarily the smallest.

```
/** Return shortest substring x of s such that s = x + x + ··· + x

public static String deduplicate(String s) {
    for (int k= 1; k < s.length(); k= k+1) {
        String x= s.substring(0, k);



    }

}
```

deduplicate

STAY AWAY FROM NESTED LOOPS. Put in comment saying what to do.

```
/** Return shortest substring x of s such that s = x + x + ··· + x
public static String deduplicate(String s) {

    for (int k= 1; k < s.length(); k= k+1) {
        String x= s.substring(0, k);
        // Return true if s is  x + x + … + x

    }

    return s;
}
```

deduplicate

STAY AWAY FROM NESTED LOOPS. Write a method to return true if s is x + x + … + x

```
/** = "s is formed by catenating x   1 or more times."
   *     Precondition: x.length() > 0. */
 public static boolean sIsCatX(String s, String x) {



}
```

deduplicate

STAY AWAY FROM NESTED LOOPS. Write a method to return true if s is x + x + … + x

```
/** = "s is formed by catenating x   1 or more times."
  *     Precondition: x.length() > 0. */
 public static boolean sIsCatX(String s, String x) {
        int n=  x.length();
        if (s.length() < x.length()) return false;
        if (!x.equals(s.substring(0, n))) return false;
        return s.length() == n  ||  sIsCatX(s.substring(n), x);
}
```

deduplicate

STAY AWAY FROM NESTED LOOPS. Write a method to return true if s is x + x + … + x

```
/** Return shortest substring x of s such that s = x + x + ··· + x
public static String deduplicate(String s) {

    for (int k= 1; k < s.length(); k= k+1) {
        String x= s.substring(0, k);
        // Return true is s is  x + x + … + x
        if (sIsCatX(s, x)) return x;
    }

    return s;
}
```

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

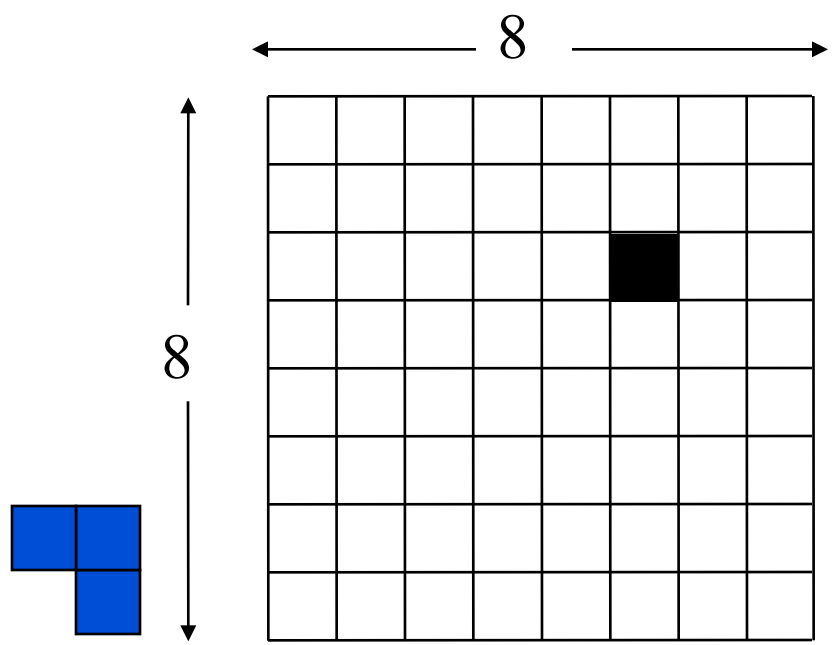
Tiling Elaine's kitchen

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.

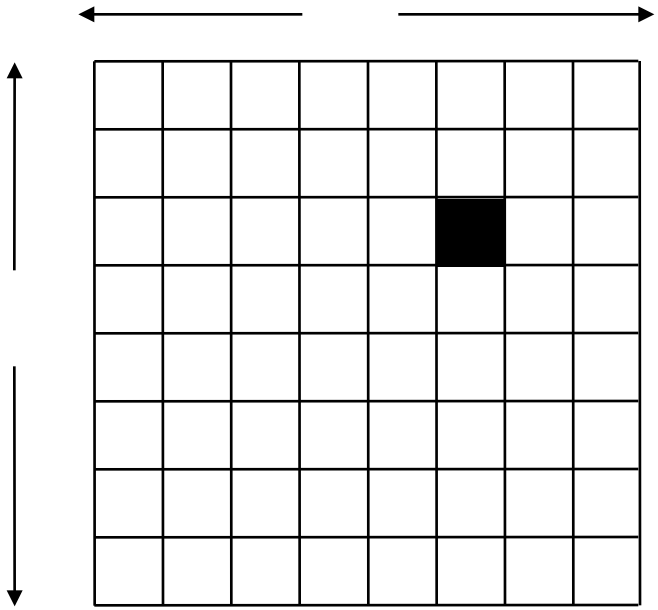
```
/** tile a 2^3 by 2^3 kitchen with 1
    square filled. */
public static void tile(int n)
```

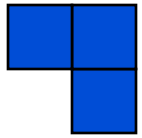
We abstract away keeping track of where the filled square is, etc.



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;


}
```



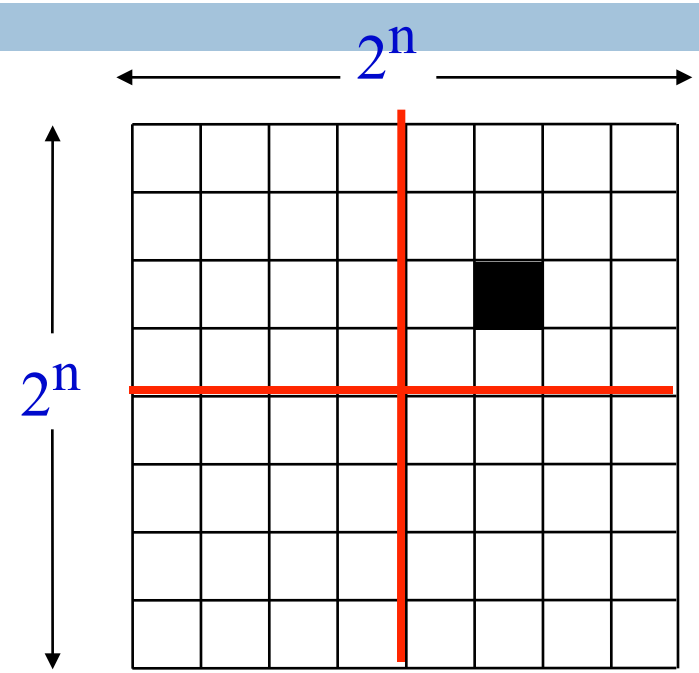


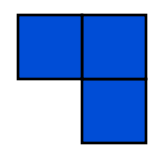
Base case?

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;


}
```

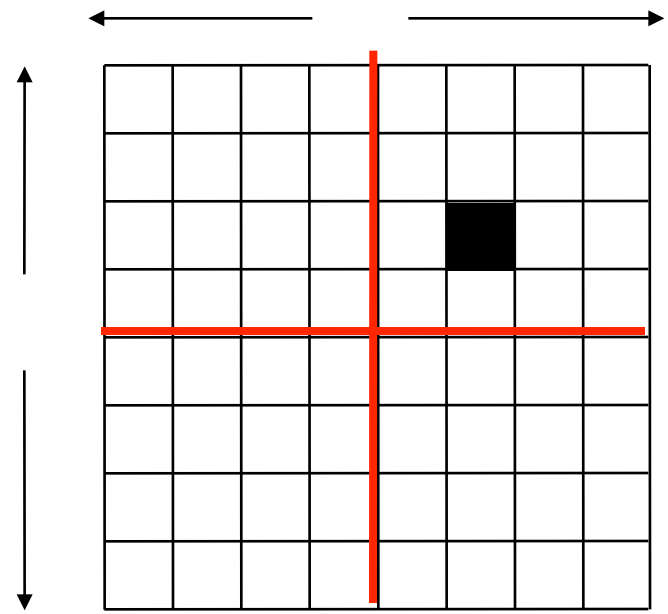




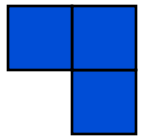
n > 0. What can we do to get kitchens of size 2^{n-1} by 2^{n-1}

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.

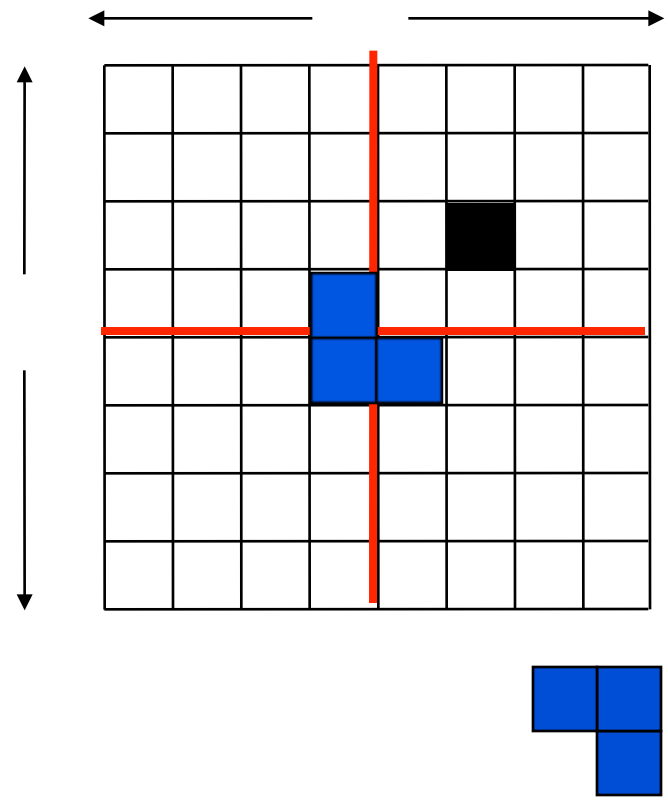


But we can't tile the other three because they don't have a filled square.

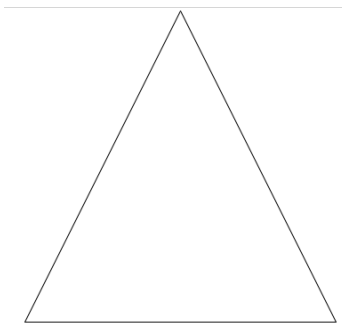
What can we do? Remember, the idea is to tile the kitchen!

Tiling Elaine's kitchen

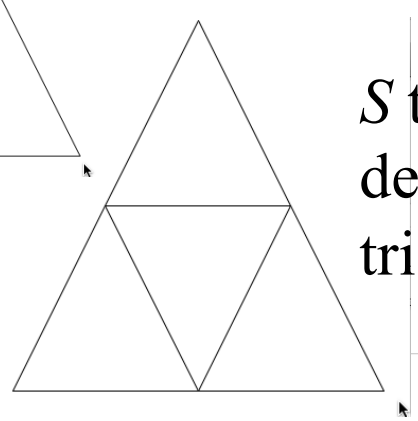

```
/** tile a 2^n by 2^n kitchen with 1
    square filled. */
public static void tile(int n) {
    if (n == 0) return;
    Place one tile so that each kitchen
    has one square filled;

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



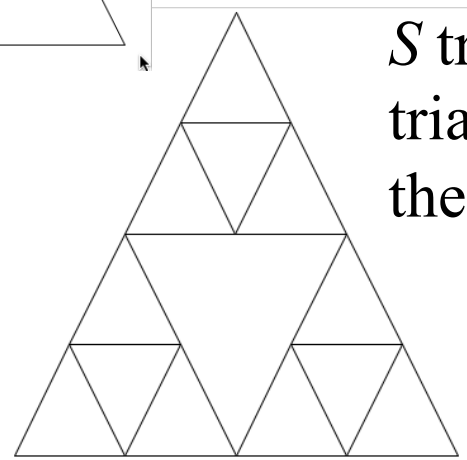
Sierpinski triangles



S triangle of depth 0



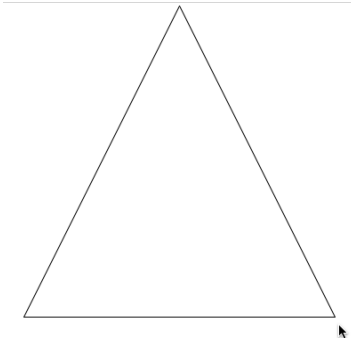
S triangle of depth 1: 3 S triangles of depth 0 drawn at the 3 vertices of the triangle



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

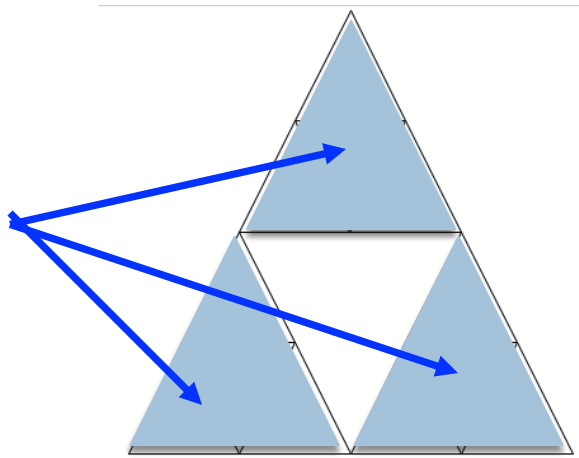
Sierpinski triangles

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



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