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:

### 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

```java
public static String deduplicate(String s)
{
    // 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.
    for (int k= 1; k < s.length(); k= k+1) {
        String x= s.substring(0, k);
    }
}
```
/** Return shortest substring x of s such that s = x + x + ⋯ + x */

public static String deduplicate(String s) {
    // Return true if s is  x + x + ⋯ + x
    return s;
}

/** = "s is formed by catenating x 1 or more times." */

public static boolean sIsCatX(String s, String x) {
    int n = x.length();
    if (s.length() < n) return false;
    if (!x.equals(s.substring(0, n))) return false;
    return s.length() == n || sIsCatX(s.substring(n), x);
}

/** = 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);
}

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^2)^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^2)^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 $n \geq 0$

If $n = 2^k$, $k$ is called the logarithm (to base 2) of $n$: $k = \log n$ or $k = \log(n)$

### Table of log to the base 2

<table>
<thead>
<tr>
<th>$k$</th>
<th>$n = 2^k$</th>
<th>$\log_2 n$ (= $k$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>16</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>32</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>64</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>128</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>256</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>512</td>
<td>9</td>
</tr>
<tr>
<td>10</td>
<td>1024</td>
<td>10</td>
</tr>
<tr>
<td>15</td>
<td>32768</td>
<td>15</td>
</tr>
</tbody>
</table>

### Difference between linear and log solutions?

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

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

**/** $= b**n$. Precondition: $n \geq 0$ */
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);
}

Tiling Elaine’s 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 $2^n$ 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 $2^{n-1}$

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

Tiling Elaine’s kitchen

We generalize to a $2^n$ by $2^n$ 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?
Tiling Elaine’s kitchen

```java
/** 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!

Sierpinski triangles

```plaintext
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
```

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