Lecture 12

More Recursion

Announcements for This Lecture

Recursion

- Read: 15.1, p. 415
- PLive, activity 15-2.1
- Work on many exercises
  - Today's (& Wed) lab
- Remember you need
  - Good function specification
  - Base case(s) are correct
  - Progress toward termination
  - Recursive case(s) are correct

Prelim 1

- Thursday 7:30-9pm
  - Abel–Price (Upson B17)
  - Rabbit–Teo (Upson 111)
  - Ting–Zytariuk (Upson 109)
- Graded late Thursday
  - Will have grade Fri morn
  - In time for drop day
- Make-up, Friday 4:30
  - For preapproved students

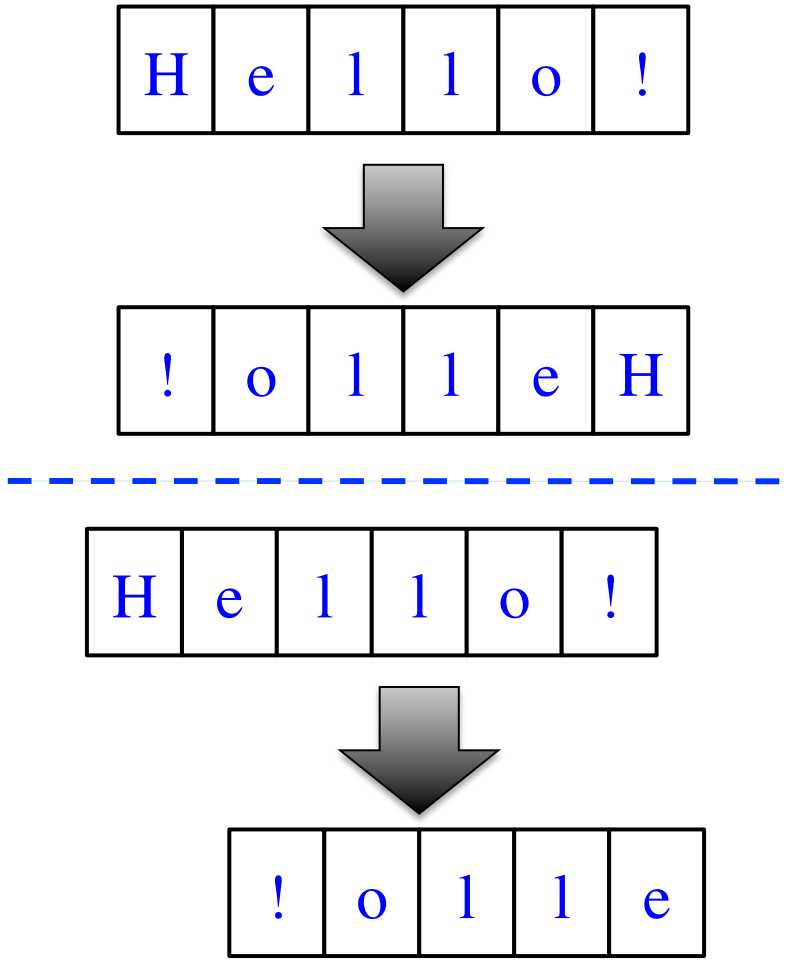
Recursion

- **Recursive Definition**:
  A definition that is defined in terms of itself
- **Recursive Method**:
  A method that calls itself (directly or indirectly)

- Powerful programming tool
  - Want to solve a difficult problem
  - Solve a simpler problem instead
  - **Goal of Recursion:**
    Solve original problem with help of simpler solution

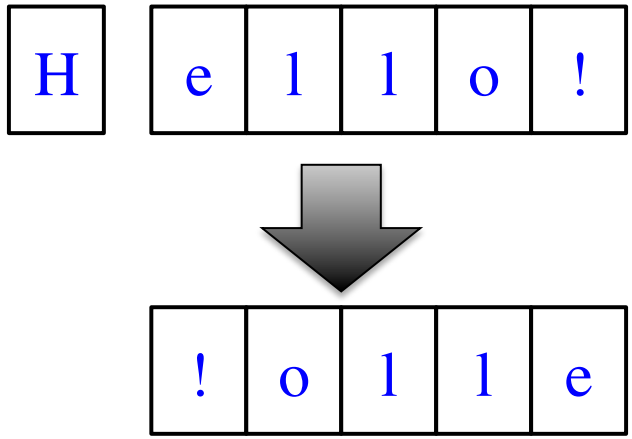
Example: Reversing a String

- **Precise Specification**:
  - Yield: reverse of String s
- Solving with recursion
  - Suppose we could reverse a smaller string (e.g. less one character)
  - Can we use that solution to reverse whole string?
- Often easy to understand first without Java
  - Then sit down and code



Example: Reversing a String

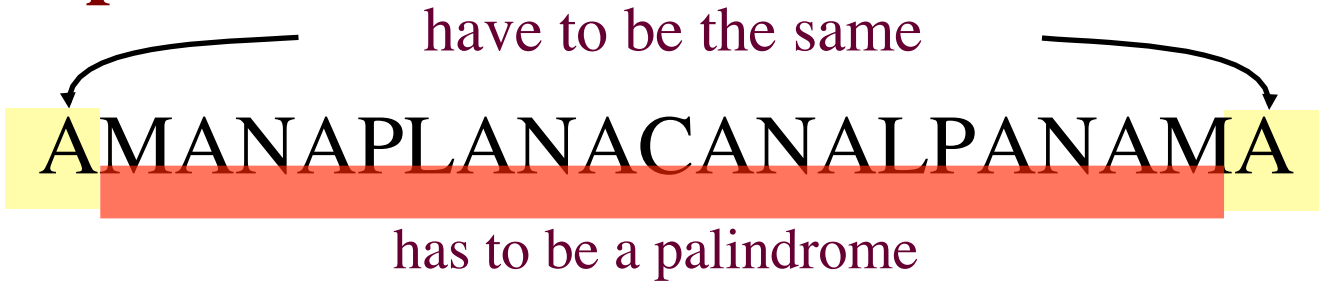
```
/** Yields: reverse of string s */
public static String reverse(String s) {
  if (s.length() == 0) {
    return s;
  }

  // {s is not empty}
  // (reverse of s[1..])+s[0]
  return reverse(s.substring(1)) +
         s.charAt(0);
}
```



✔ 1. Precise specification?
✔ 2. Base case: correct?
✔ 3. Recursive case: progress to termination?
✔ 4. Recursive case: correct?

Example: Palindromes

- String with ≥ 2 characters is a palindrome if:
  - its first and last characters are equal, and
  - the rest of the characters form a palindrome
- **Example:**



- **Precise Specification:**

```
/** Yields: "s is a palindrome" */
public static boolean isPalindrome(String s)
```

Example: Palindromes

- String with ≥ 2 characters is a palindrome if:
  - its first and last characters are equal, and
  - the rest of the characters form a palindrome
- **Recursive Method:**

Recursive Definition

```
/** Yields: "s is a palindrome" */
public static boolean isPalindrome(String s) {
    if (s.length() <= 1) { return true; }
    // { s has at least two characters }
    return s.charAt(0) == s.charAt(s.length()–1) &&
           isPalindrome(s.substring(1, s.length()–1));
}
```

Base case

Recursive case

Example: Palindromes

- String with ≥ 2 characters is a palindrome if:
  - its first and last characters are e
  - the rest of the characters form
- **Recursive Method:**

1. Precise specification?
2. Base case: correct?
3. Recursive case: progress to termination?
4. Recursive case: correct?

```
/** Yields: "s is a palindrome" */
public static boolean isPalindrome(String s) {
    if (s.length() <= 1) { return true; }
    // { s has at least two characters }
    return s.charAt(0) == s.charAt(s.length()–1) &&
           isPalindrome(s.substring(1, s.length()–1));
}
```

Example: More Palindromes

```
/** Yields: "s is a palindrome". Case of characters is ignored. */
public static boolean isPalindrome2(String s) {
    if (s.length() <= 1) { return true; }

    // { s has at least two characters }
    return equalsIgnoreCase(s.charAt(0),s.charAt(s.length()–1) &&
           isPalindrome(s.substring(1, s.length()–1));
}

/** Yields: "c and d are same ignoring case" */
public static boolean equalsIgnoreCase(char a, char b) {
    return Character.toUpperCase(a) == Character.toUpperCase(b);
}
```

Precise Specification

Example: More Palindromes

```
/** Yields: "s is a palindrome".
 *  Case of characters and non-letters ignored.  */
public static boolean isPalindrome3(String s) {
    return isPalindrome2(depunct(s));
}

/** Yields: s with non-letters removed */
public static String depunct(String s) {
    if (s.length() == 0)  { return s; }
    // {s is not empty}
    if (!Character.isLetter(s.charAt(0))) { return depunct(s.substring(1)); }
    // {s is not empty and s[0] is a letter}
    return s.charAt(0) + depunct (s.substring(1));
}
```

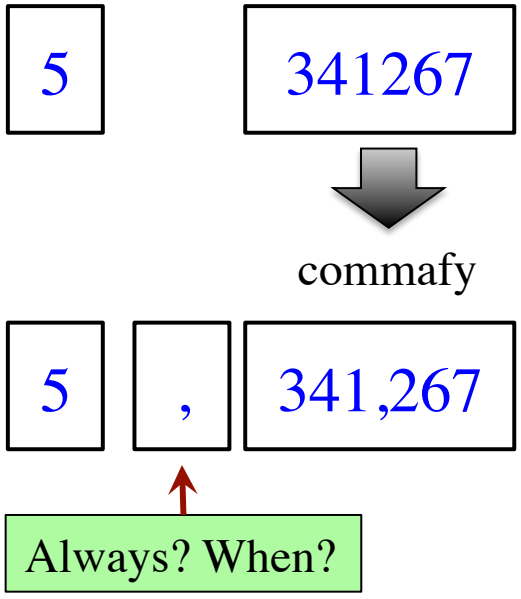
Use helper methods!

- Often easy to break a problem into two
- Can use recursion more than once to solve

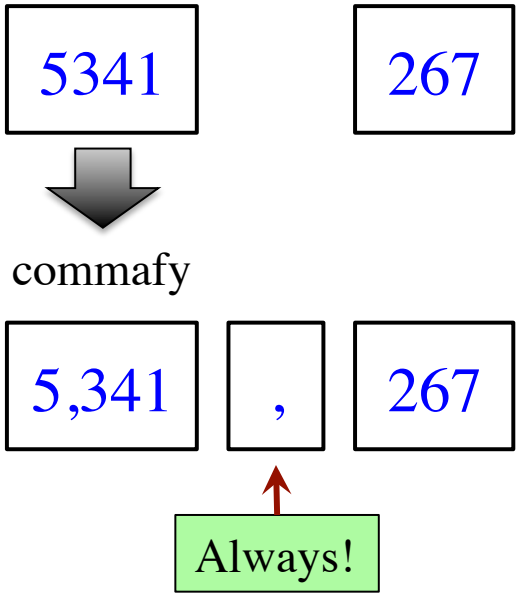
How to Break Up a Recursive Method?

```
/** Yields: String with commas every 3 digits
 *   Precondition: s represents a non-negative int
 *   e.g. commafy("5341267") = "5,341,267" */
public static String commafy(String s)
```

Approach 1



Approach 2



How to Break Up a Recursive Solution?

```
/** Yields: String with commas every 3 digits
 *    Precondition: s represents a non-negative int
 *    e.g. commafy("5341267") = "5,341,267" */
public static String commafy(String s) {
   // No commas if too few digits.
   if (s.length() <= 3) { return s; }

   // Add the comma before last 3 digits
   return commafy(s.substring(0,s.length()-3)) + "," +
          s.substring(s.length()-3);
}
```

Base case

Recursive case

How to Break Up a Recursive Method?

/** Yields: b^c
 * Precondition: $c \geq 0$ */
public static double exp(double b, int c)

Approach 1

$12^{256} = 12 \times (12^{255})$

Recursive

$b^c = b \times (b^{c-1})$

Approach 2

$12^{256} = (12^{128}) \times (12^{128})$

Recursive Recursive

$b^c = (b \times b)^{c/2}$ if c even

Raising a Number to an Exponent

Approach 1

```
/** Yields: bc
 *  Precondition: c ≥ 0 */
public static double exp(double b, int c) {
   // b^0 is 1
   if (c == 0) {
      return 1;
   }

   // b^c = b(b^c)
   return b*exp(b,c-1);
}
```

Approach 2

```
/** Yields: bc
 *  Precondition: c ≥ 0 */
public static double exp(double b, int c) {
 // b^0 is 1
   if (c == 0)  { return 1; }

   // c > 0
   if (c % 2 == 0) { return exp(b*b,c/2); }

   return b*exp(b,c/2);
}
```

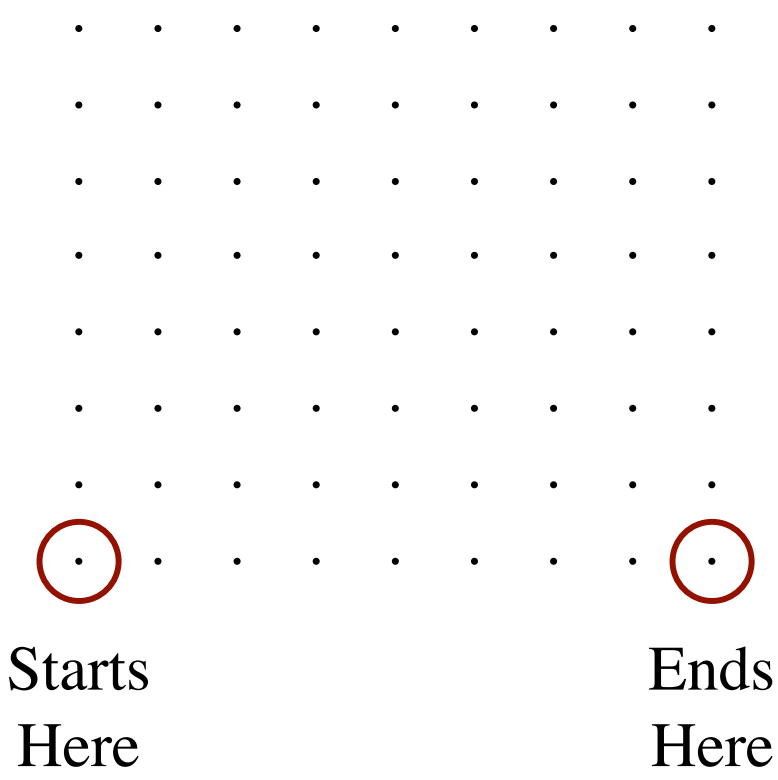
Raising a Number to an Exponent

```
/** Yields: bc
 *  Precondition: c ≥ 0 */
public static double exp(double b, int c) {
   // b^0 is 1
   if (c == 0)  { return 1; }

   // c > 0
   if (c % 2 == 0) {
      return exp(b*b,c/2);
   }

   return b*exp(b,c/2);
}
```

c	# of calls
0	0
1	1
2	2
4	3
8	4
16	5
32	6
2^n	n + 1

32768 is 215
b^{32768} needs only 215 calls!

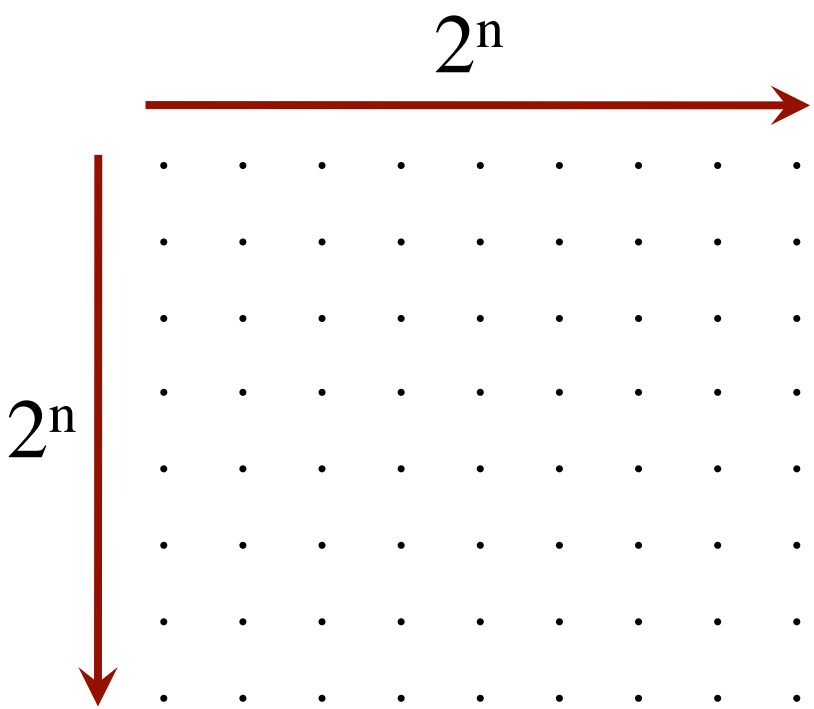
Space Filling Curves



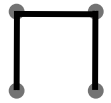
Challenge

- Draw a curve that
  - Starts in the left corner
  - Ends in the right corner
  - Touches every grid point
  - Does not touch or cross itself anywhere
- Useful for analysis of 2-dimensional data

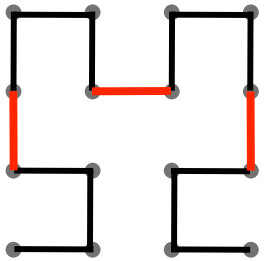
Hilbert's Space Filling Curve



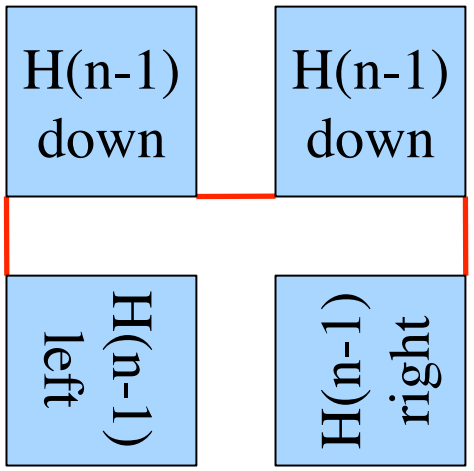
Hilbert(1):



Hilbert(2):



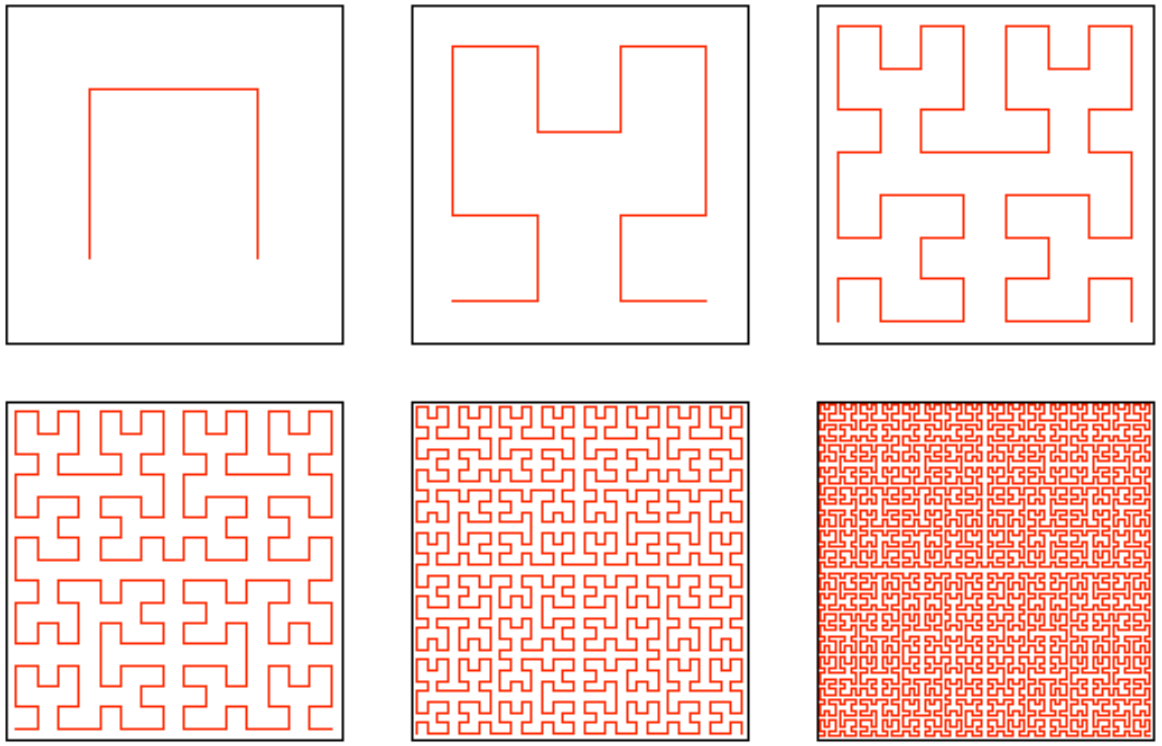
Hilbert(n):



Hilbert's Space Filling Curve

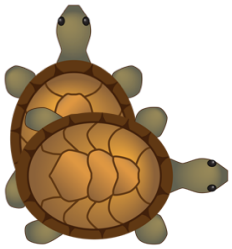
Basic Idea

- Given a box
- Draw $2^n \times 2^n$ grid in box
- Trace the curve
- As n goes to ∞, curve fills box

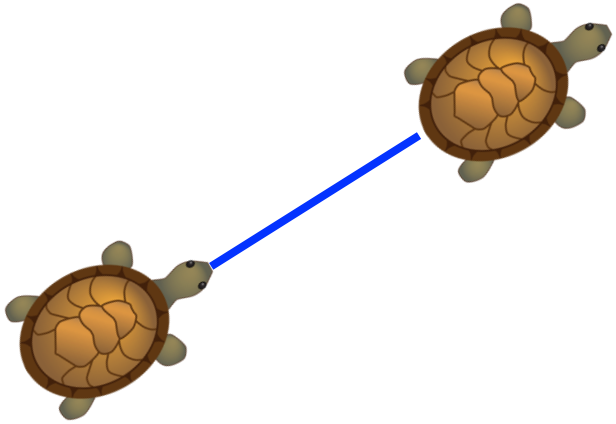


"Turtle" Graphics: Assignment A5

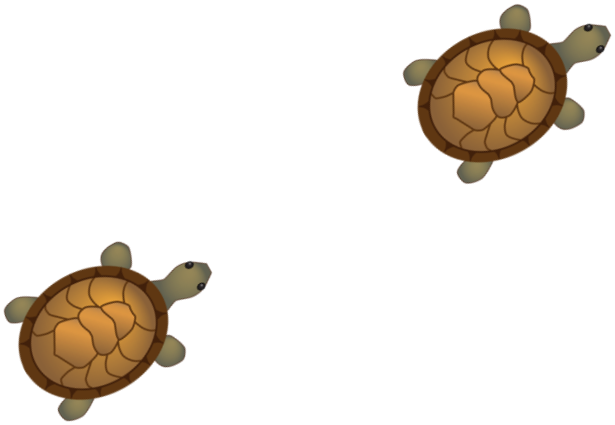
Turn



Draw Line



Move



Change Color