Lecture 16

More Recursion

Announcements for This Lecture

Prelim 1

- Prelim 1 back **today!**
  - Pick up in Lab Section
  - Solution posted in CMS
  - **Mean**: 80, **Median**: 83
- What are letter grades?
  - A bit too early to tell
  - **A**: Could be a consultant
  - **B**: Could take 2110
  - **C**: Good enough to pass

Assignments and Labs

- Need to be working on A4
  - Instructions are posted
  - Just reading it takes a while
  - Slightly longer than A3
  - Problems are harder
- **Lab Today**: lots of practice!
  - 4 functions are mandatory
  - Lots of optional ones to do
  - Exam questions on Prelim 2

Recall: Reversing a String

Using Recursion

```python
def reverse(s):
    """Returns: reverse of s

    Precondition: s a string"""
    # s is empty
    if s == '':
        return s

    # s has at least one char
    # (reverse of s[1:])+s[0]
    return reverse(s[1:])+s[0]
```

Using a For-Loop

```python
def reverse(s):
    """Returns: reverse of s

    Precondition: s a string"""
    # create an accumulator
    copy == '

    # accumulate copy in reverse
    for x in s:
        copy = x+copy
    return copy
```

Recall: Reversing a String

Using Recursion

Using a For-Loop

```
def reverse(s):
    """Returns: reverse of s

    Precondition: s a string"""
    # s is
    if
```

```
def reverse(s):
    """Returns: reve
```

Why bother with recursion at all?

```
    # s has at least one char
    # (reverse of s[1:])+s[0]
    return reverse(s[1:])+s[0]
```

```
    # accumulate copy in reverse
    for x in s:
        copy = x+copy
    return copy
```

Recall: Iteration

1. Process each item in a sequence
   - Compute aggregate statistics for such as the mean, median, stand
   - Send everyone in a Facebook group an appointment time

   ```
   for x in sequence:
       process x
   ```

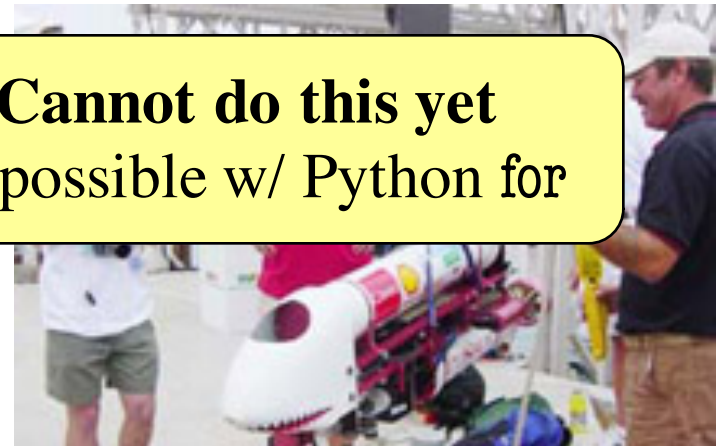2. Perform *n* trials or get *n* samples.
   - **OLD A4**: draw a triangle six time
   - Run a protein-folding simulation

   ```
   for x in range(n):
       do next thing
   ```

3. Do something an unknown number of times
   - CUAUV team, vehicle keeps moving until reached its goal

   Cannot do this yet
   Impossible w/ Python for

Recursion and Iteration

- Recursion *theoretically equivalent* to iteration
  - Anything can do in one, can do in other
  - But what is easy in one may be hard in other
  - **When is using recursion better?**
- Recursion is more flexible in breaking up data
  - Iteration typically scans data left-to-right
  - Recursion works with other "slicings"
- Recursion has interesting advanced applications
  - See some of these in **Assignment 4**

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:**

have to be the same

AMANAPLANACANALPANAMA

has to be a palindrome

- **Precise Specification:**

```python
def ispalindrome(s):

    """Returns: True if s is a palindrome"""
```

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
  Definition

- **Recursive Function:**

```python
def ispalindrome(s):
    """Returns: True if s is a palindrome"""
    if len(s) < 2:
        return True          Base case

    // { s has at least two characters }    Recursive case
    return s[0] == s[-1] and ispalindrome(s[1:-1])
```

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 Function:**

```python
def ispalindrome(s):
    """Returns: True if s is a palindrome"""
    if len(s) < 2:
        return True                    Base case

    // { s has at least two characters }   Recursive case
    return s[0] == s[-1] and ispalindrome(s[1:-1])
```

Example: More Palindromes

```python
def ispalindrome2(s):
    """Returns: True if s is a palindrome
    Case of characters is ignored."""
    if len(s) < 2:
        return True

    // { s has at least two characters }
    return ( equals_ignore_case(s[0],s[-1])
            and ispalindrome2(s[1:-1]) )
```

Example: More Palindromes

```python
def ispalindrome2(s):
    """Returns: True if s is a palindrome
    Case of characters is ignored."""
    if len(s) < 2:
        return True

    // { s has at least two characters }
    return ( equals_ignore_case(s[0],s[-1])
             and ispalindrome2(s[1:-1]) )
```

Precise Specification

Example: More Palindromes

```
def ispalindrome2(s):
    """Returns: True if s is a palindrome
    Case of characters is ignored."""
    if len(s) < 2:
        return True

    // { s has at least two characters }
    return ( equals_ignore_case(s[0],s[-1])
             and ispalindrome2(s[1:-1]) )

def equals_ignore_case (a, b):
    """Returns: True if a and b are same ignoring case"""
    return a.upper() == b.upper()
```

Precise Specification

Example: More Palindromes

```python
def ispalindrome3(s):
    """Returns: True if s is a palindrome
    Case of characters and non-letters ignored."""
    return ispalindrome2(depunct(s))


def depunct(s):
    """Returns: s with non-letters removed"""
    if s == '':
        return s
    # use string.letters to isolate letters
    if s[0] in string.letters:
        return s[0]+depunct(s[1:])
    return depunct(s[1:])
```

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

Recursion is form of Divide and Conquer

Goal: Solve problem P on a piece of data

data

Recursion is form of Divide and Conquer

Goal: Solve problem P on a piece of data

data

Idea: Split data into two parts and solve problem

data 1	data 2

Solve Problem P   Solve Problem P

Recursion is form of Divide and Conquer

Goal: Solve problem P on a piece of data

data

Idea: Split data into two parts and solve problem

| **data 1** | **data 2** |
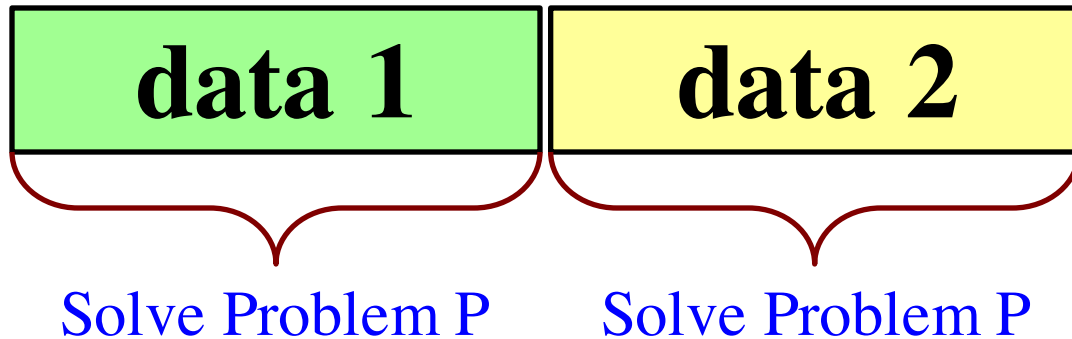
Solve Problem P   Solve Problem P
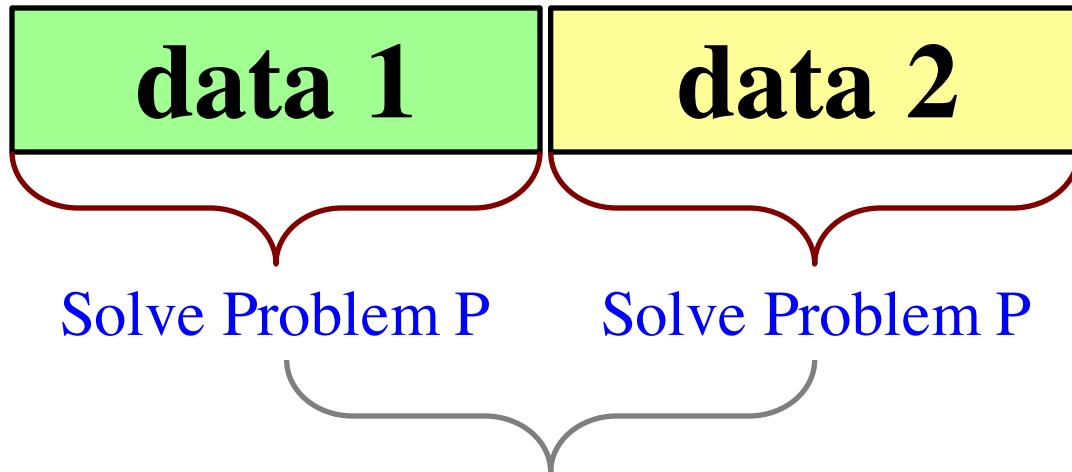
Combine Answer!

Recursion is form of Divide and Conquer

Goal: Solve problem P on a piece of data

data

Idea: Split data into two parts and solve problem

data 1 **data 2**
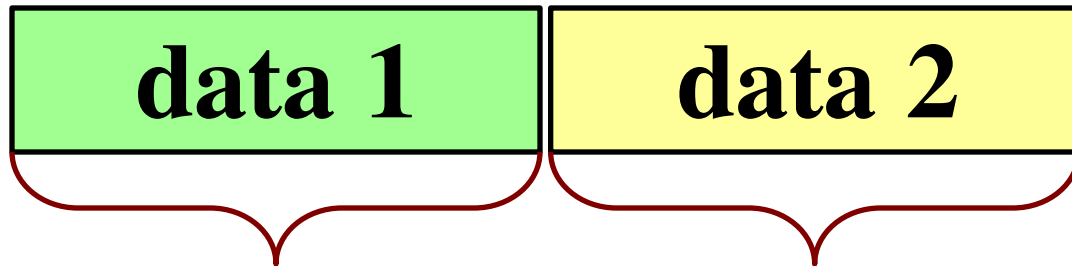
Solve Problem P Solve Problem P

Combine Answer!
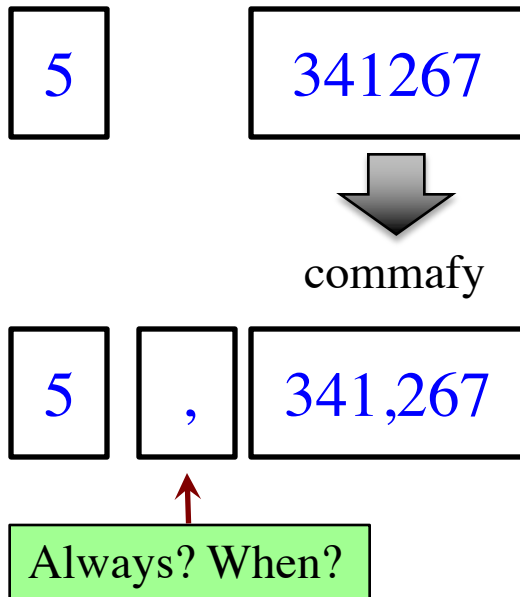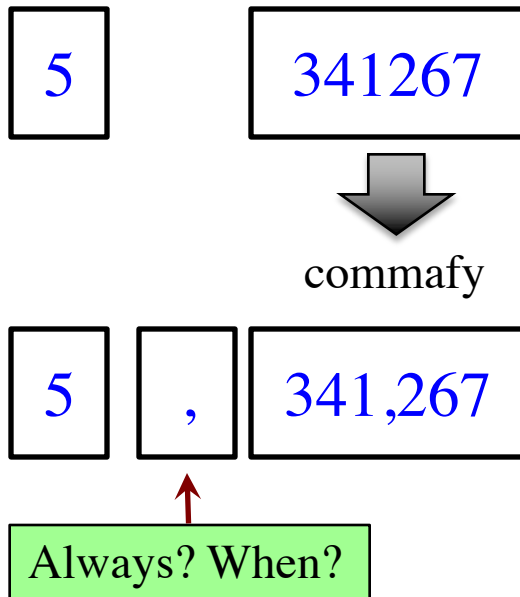
Where work is all done

How to Break Up a Recursive Function?

```
def commafy(s):
    """Returns: string with commas every 3 digits
    e.g. commafy('5341267') = '5,341,267'
    Precondition: s represents a non-negative int"""
```

Approach 1

| 5 | | 341267 |

How to Break Up a Recursive Function?

```python
def commafy(s):
    """Returns: string with commas every 3 digits
    e.g. commafy('5341267') = '5,341,267'
    Precondition: s represents a non-negative int"""
```
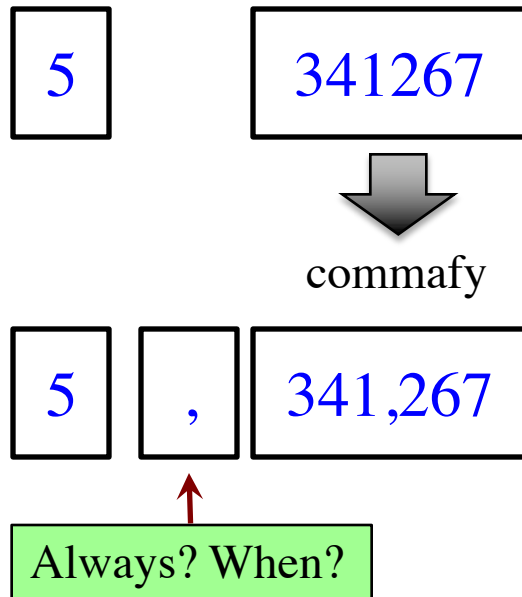
Approach 1

5   341267
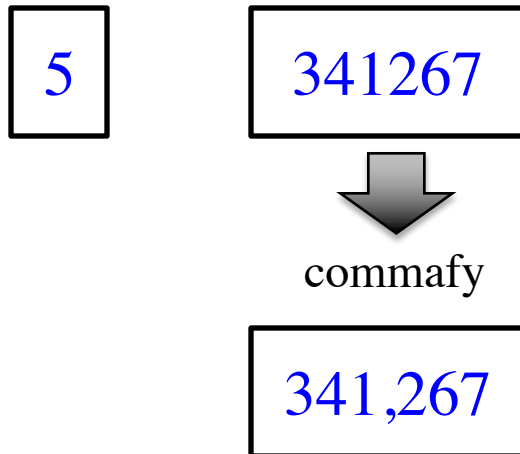
commafy

341,267

How to Break Up a Recursive Function?

```
def commafy(s):
    """Returns: string with commas every 3 digits
    e.g. commafy('5341267') = '5,341,267'
    Precondition: s represents a non-negative int"""
```

Approach 1

How to Break Up a Recursive Function?

```
def commafy(s):
    """Returns: string with commas every 3 digits
    e.g. commafy('5341267') = '5,341,267'
    Precondition: s represents a non-negative int"""
```
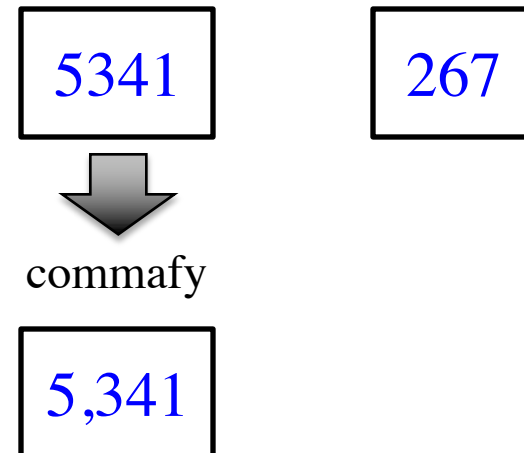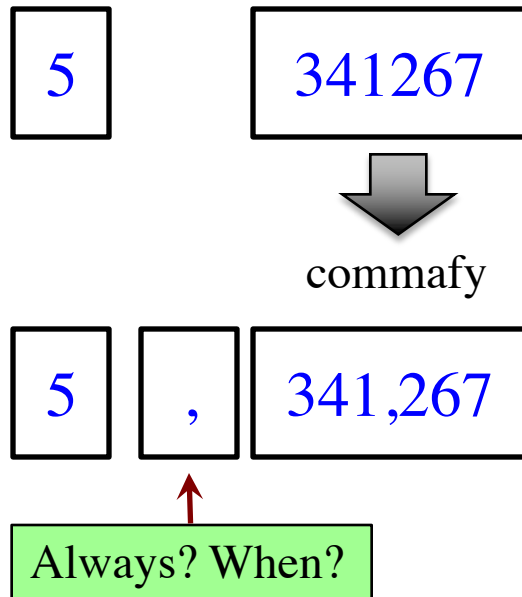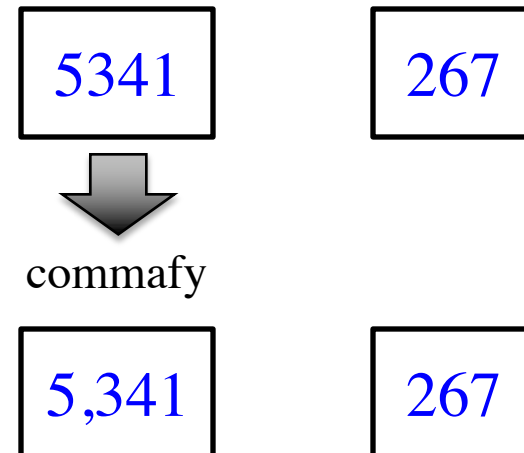
Approach 1
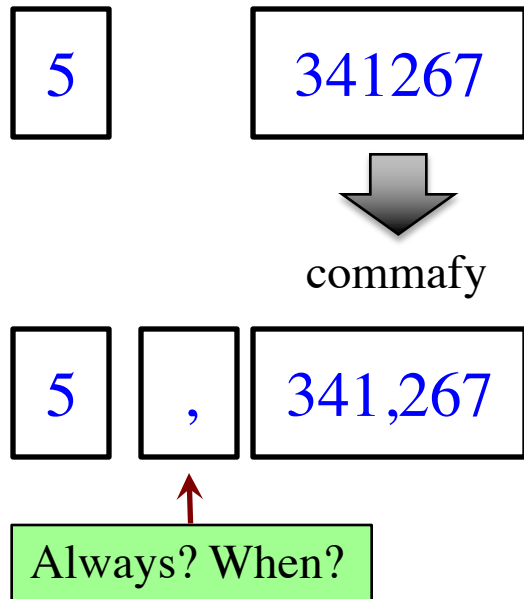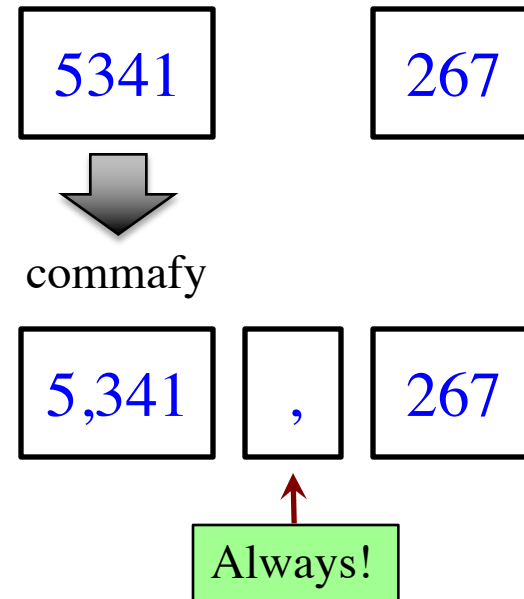

commafy

Always? When?

How to Break Up a Recursive Function?

```
def commafy(s):
    """Returns: string with commas every 3 digits
    e.g. commafy('5341267') = '5,341,267'
    Precondition: s represents a non-negative int"""
```

Approach 1

| 5 | 341267 |

commafy

| 5 | , | 341,267 |

Always? When?

Approach 2

| 5341 | 267 |

How to Break Up a Recursive Function?

```
def commafy(s):
    """Returns: string with commas every 3 digits
    e.g. commafy('5341267') = '5,341,267'
    Precondition: s represents a non-negative int"""
```

Approach 1

| 5 | 341267 |

↓ commafy

| 5 | , | 341,267 |

↑

Always? When?

Approach 2

| 5341 | 267 |

↓ commafy

| 5,341 |

More Recursion

How to Break Up a Recursive Function?

```
def commafy(s):
```
   """Returns: string with commas every 3 digits
   e.g. commafy('5341267') = '5,341,267'
   Precondition: s represents a non-negative int"""

Approach 1   ## Approach 2

| 5 | 341267 | | 5341 | 267 |

commafy   commafy

| 5 | , | 341,267 | | 5,341 | 267 |

Always? When?

How to Break Up a Recursive Function?

```
def commafy(s):
    """Returns: string with commas every 3 digits
    e.g. commafy('5341267') = '5,341,267'
    Precondition: s represents a non-negative int"""
```

Approach 1

5	341267

↓ commafy

5	,	341,267

Always? When?

Approach 2

5341	267

↓ commafy

5,341	,	267

Always!

More Recursion

How to Break Up a Recursive Function?

```python
def commafy(s):
    """Returns: string with commas every 3 digits
    e.g. commafy('5341267') = '5,341,267'
    Precondition: s represents a non-negative int"""
    # No commas if too few digits.
    if len(s) <= 3:
        return s
```

Base case

```python
    # Add the comma before last 3 digits
    return commafy(s[:-3]) + ',' + s[-3:]
```

Recursive case

How to Break Up a Recursive Function?

```
def exp(b, c)
    """Returns: bᶜ
    Precondition: b a float, c ≥ 0 an int"""
```

Approach 1

$$12^{256} = 12 \times \boxed{12^{255}}$$

Recursive

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

Approach 2

$$12^{256} = \boxed{12^{128}} \times \boxed{12^{128}}$$

Recursive   **Recursive**

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

Raising a Number to an Exponent

Approach 1

```
def exp(b, c)
    """Returns: bᶜ
    Precondition: b a float,
                c ≥ 0 an int"""
    # b⁰ is 1
    if c == 0:
        return 1

    # bᶜ = b(bᶜ)
    return b*exp(b,c-1)
```

Approach 2

```
def exp(b, c)
    """Returns: bᶜ
    Precondition: b a float,
                c ≥ 0 an int"""
    if c == 0:
        return 1
    # c > 0
    if c % 2 == 0:
        return exp(b*b,c/2)

    return b*exp(b*b,(c-1)/2)
```

Raising a Number to an Exponent

```
def exp(b, c)
    """Returns: b^c
    Precondition: b a float,
                  c ≥ 0 an int"""

    # 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*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!

Recursion and Objects

- Class Person (`person.py`)
  - Objects have 3 attributes
  - `name`: String
  - `mom`: Person (or `None`)
  - `dad`: Person (or `None`)
- Represents the "family tree"
  - Goes as far back as known
  - Attributes `mom` and `dad` are `None` if not known
- **Constructor**: Person(n,m,d)
  - Or Person(n) if no mom, dad
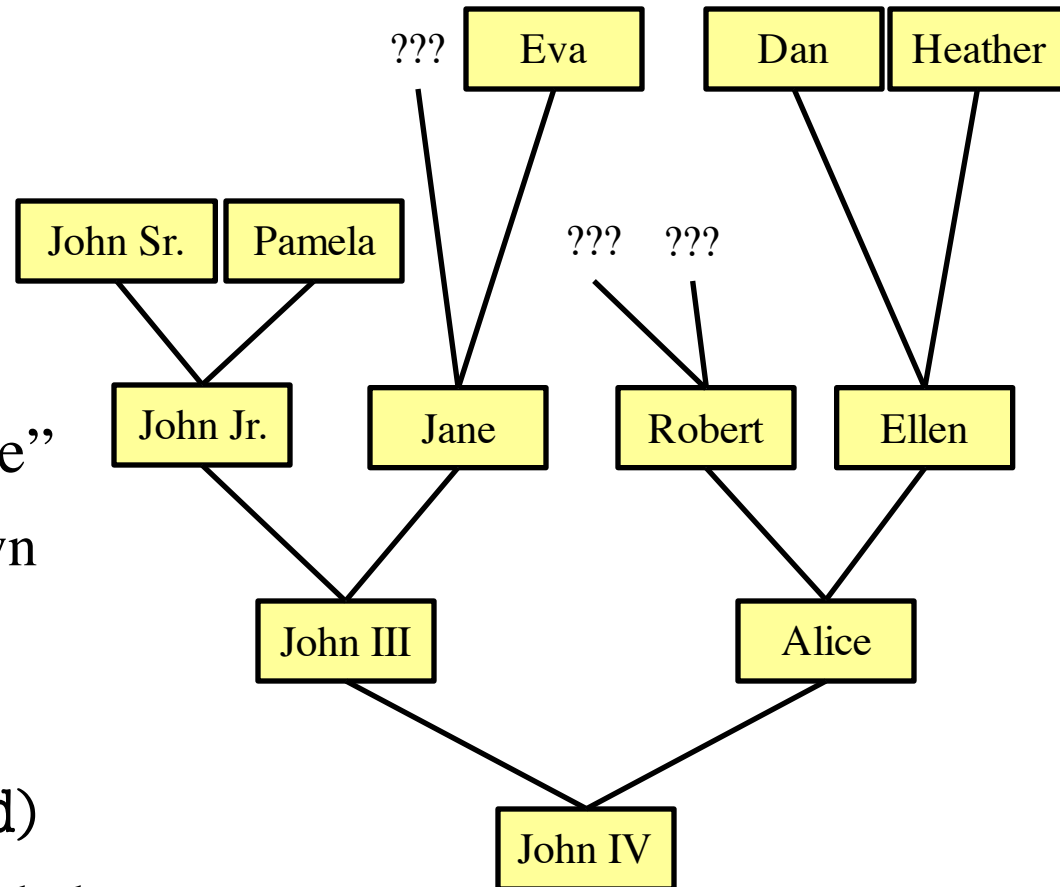
Recursion and Objects

```
def num_ancestors(p):
    """Returns: num of known ancestors
    Pre: p is a Person"""
    # Base case
    # No mom or dad (no ancestors)


    # Recursive step
    # Has mom or dad
    # Count ancestors of each one
    # (plus mom, dad themselves)
    # Add them together
```

Recursion and Objects

```python
def num_ancestors(p):
    """Returns: num of known ancestors
    Pre: p is a Person"""
    # Base case
    if p.mom == None and p.dad == None:
        return 0

    # Recursive step
    moms = 0
    if not p.mom == None:
        moms = 1+num_ancestors(p.mom)
    dads = 0
    if not p.dad == None:
        dads = 1+num_ancestors(p.dad)
    return moms+dads
```
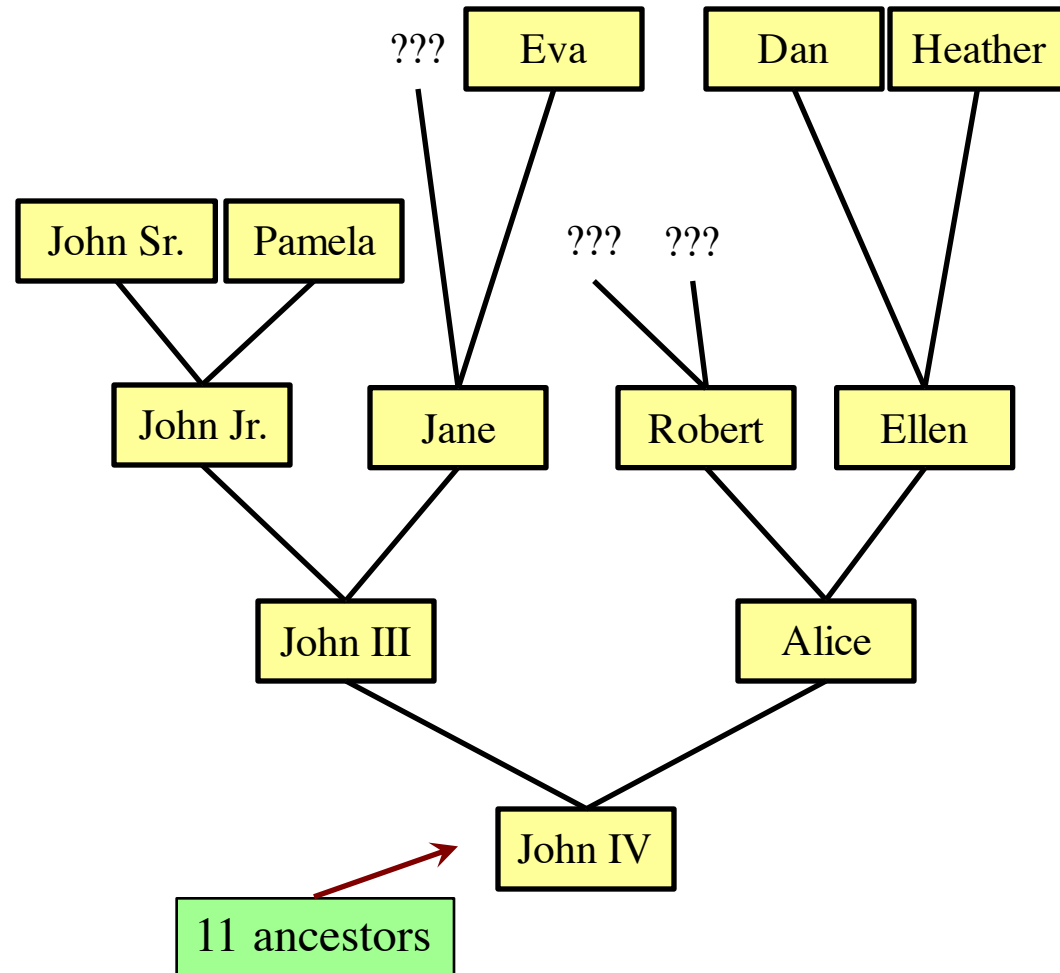
Space Filling Curves

Challenge

. . . .

. . . .

. . . .

. . . .

. . . .

. . . .

. . . .

(·) . . . . . . (·)

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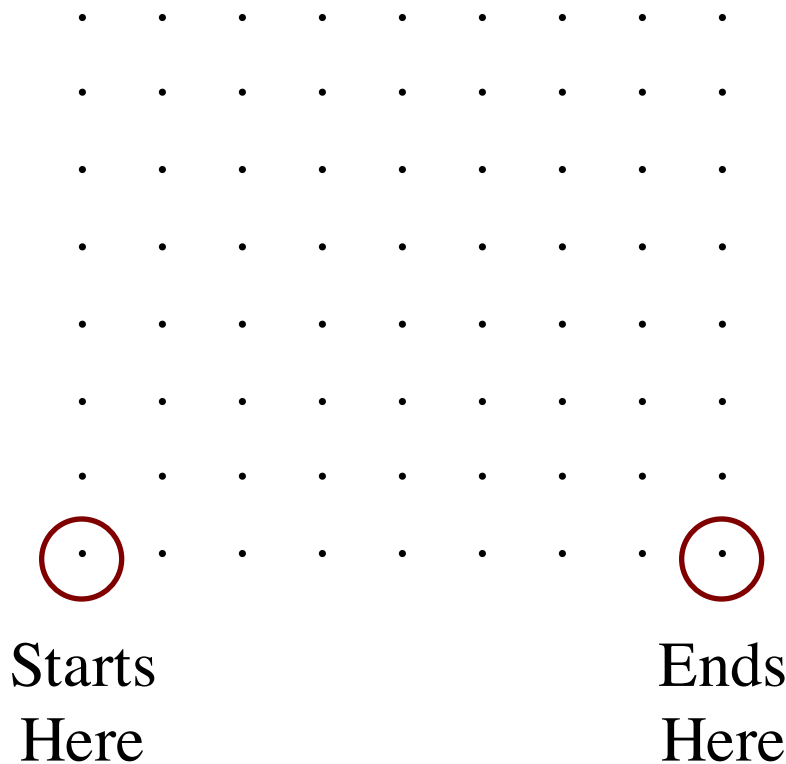
- Draw a curve that
  - Starts in the left corner
  - Ends in the right corner
  - Touches every grid point
  - Does not touch or cross itself anywhere
- Useful for analysis of 2-dimensional data

Hilbert's Space Filling Curve

2^n

2^n
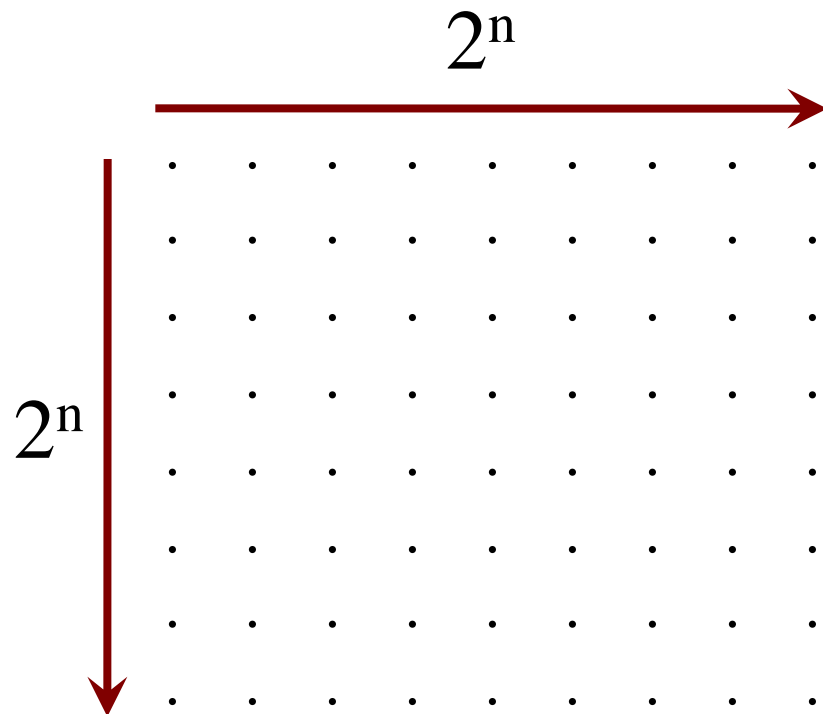
Hilbert(1):

Hilbert(2):

Hilbert(n):

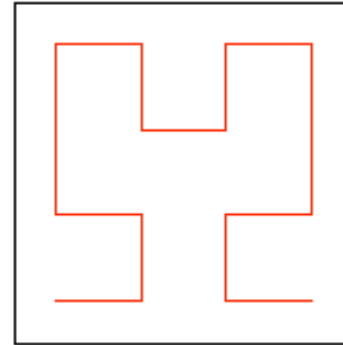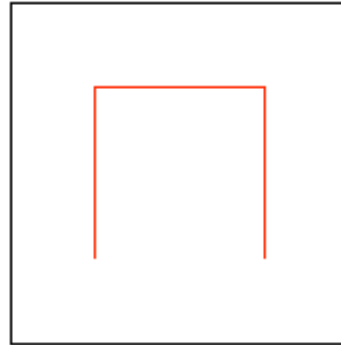| H(n-1) down	H(n-1) down
H(n-1) left	H(n-1) right
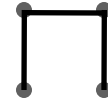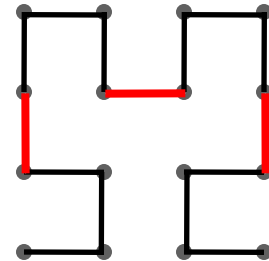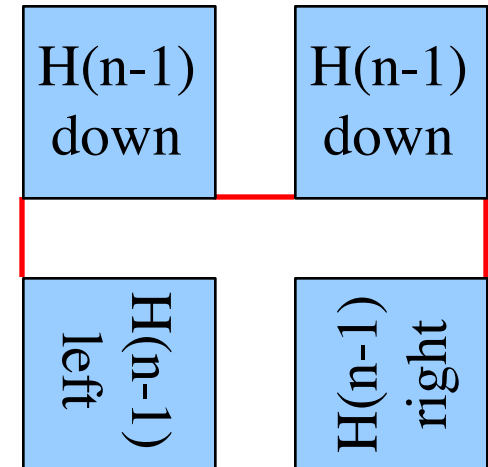
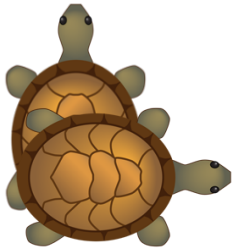# Hilbert's Space Filling Curve

Basic Idea

- Given a box

- Draw $2^n \times 2^n$ grid in box

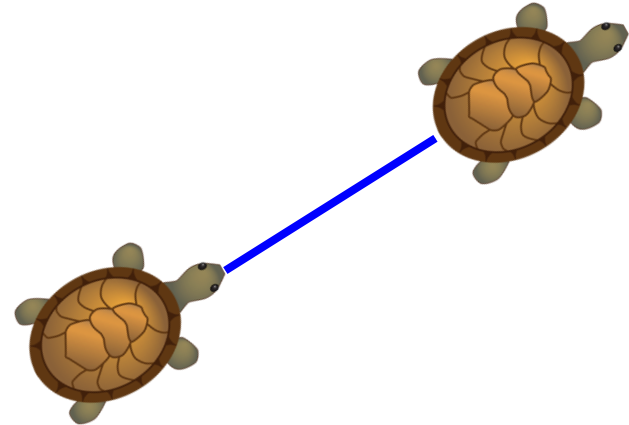- Trace the curve

- As n goes to ∞, curve fills box
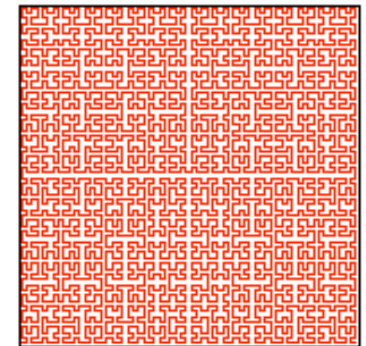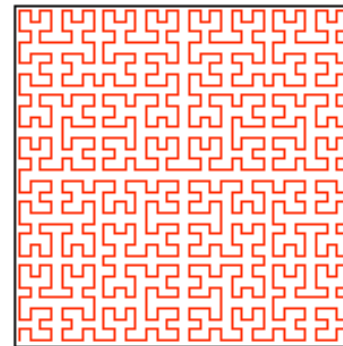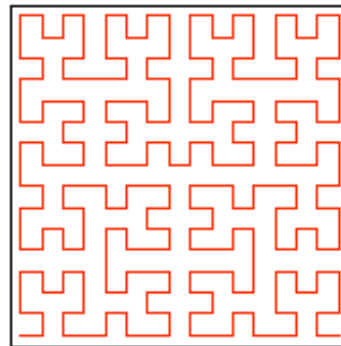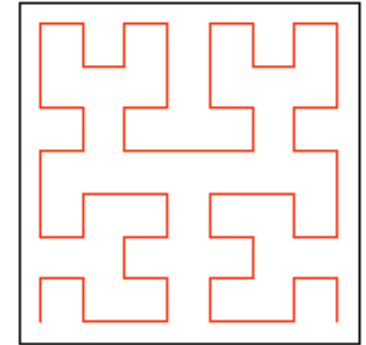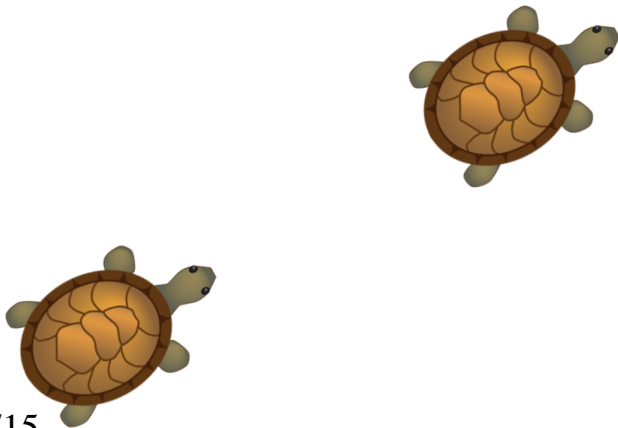
"Turtle" Graphics: Assignment A4

Turn

Draw Line

Move

Change Color