Lecture 16

More Recursion
Announcements for This Lecture

Prelim 1

• Prelim 1 back today!
  § Pick up in Gates 216
  § Solution posted in CMS
  § Mean: 73, Median: 76
  § Twinsies was the issue
• What are letter grades?
  § A: 85+ (consultant level)
  § B: 60-79 (major level)
  § C: 35-55 (should be S/U)

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!
  § First 4 functions mandatory
  § Many optional ones in PDF
  § Exam questions on Prelim 2

10/17/17

More Recursion
Recall: Divide and Conquer

Goal: Solve problem P on a piece of data

Idea: Split data into two parts and solve problem

Combine Answer!
def reverse(s):
    """Returns: reverse of s
    Precondition: s a string"""
    # 1. Handle small data
    if len(s) <= 1:
        return s
    # 2. Break into two parts
    # 3. Combine the result
def reverse(s):
    """Returns: reverse of s
    Precondition: s a string""
    # 1. Handle small data
    if len(s) <= 1:
        return s

    # 2. Break into two parts
    left = s[0]
    right = reverse(s[1:]):

    # 3. Combine the result
Example: Reversing a String

```python
def reverse(s):
    """Returns: reverse of s
Precondition: s a string"
    # 1. Handle small data
    if len(s) <= 1:
        return s
    # 2. Break into two parts
    left = s[0]
    right = reverse(s[1:])
    # 3. Combine the result
    return right + left
```

10/17/17  More Recursion 6
def reverse(s):
    """Returns: reverse of s
    Precondition: s a string""
    # 1. Handle small data
    if len(s) <= 1:
        return s
    # 2. Break into two parts
    left = s[0]
    right = reverse(s[1:])
    # 3. Combine the result
    return right+left
Example: Reversing a String

```python
def reverse(s):
    # Returns: reverse of s
    # Precondition: s a string
    # 1. Handle small data
    if len(s) <= 1:
        return s
    # 2. Break into two parts
    left = s[0]
    right = reverse(s[1:])
    # 3. Combine the result
    return right + left
```

Base Case

Remove recursive call

Recursive Case
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?

```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

5  341,267
```
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

5 341267

Approach 2

5341 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
```

Approach 2

```
5341   267
```

commafy

```
5,341
```

Always? When?
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
```

Approach 2

```
5341
267
```

Always? When?
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

\[ \begin{align*}
5 & \rightarrow 341267 \\
5 & , 341,267 \\
\end{align*} \]

Approach 2

\[ \begin{align*}
5341 & \rightarrow 267 \\
5,341 & , 267 \\
\end{align*} \]
def commafy(s):
    """Returns: string with commas every 3 digits
    e.g. commafy('5341267') = '5,341,267'
    Precondition: s represents a non-negative int"""
    # 1. Handle small data.
    if len(s) <= 3:
        return s

    # 2. Break into two parts
    left = commafy(s[:-3])
    right = s[-3:]  # Small part on RIGHT

    # 3. Combine the result
    return left + ',' + right
def exp(b, c)
    """Returns: \(b^c\)
    Precondition: b a float, \(c \geq 0\) an int""

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

\[b^c = (b \times b)^{c/2} \text{ if } c \text{ even}\]
Raising a Number to an Exponent

Approach 1

```python
def exp(b, c):
    """Returns: \( b^c \)
    Precond: \( b \) a float, \( c \geq 0 \) an int""
    # \( b^0 \) is 1
    if c == 0:
        return 1
    # \( b^c = b(b^{c-1}) \)
    left = b
    right = exp(b, c-1)
    return left*right
```

Approach 2

```python
def exp(b, c):
    """Returns: \( b^c \)
    Precond: \( b \) a float, \( c \geq 0 \) 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-1)//2)
```
Raising a Number to an Exponent

Approach 1

```python
def exp(b, c):
    """Returns: b^c
    Precond: b a float, c ≥ 0 an int""
    # b^0 is 1
    if c == 0:
        return 1
    # b^c = b*(b^(c-1))
    left = b
    right = exp(b, c-1)
    return left*right
```

Approach 2

```python
def exp(b, c):
    """Returns: b^c
    Precond: 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-1)//2)
```
def exp(b, c):
    """Returns: b^c
    Precond: 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-1)//2)

<table>
<thead>
<tr>
<th>c</th>
<th># of calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>16</td>
<td>5</td>
</tr>
<tr>
<td>32</td>
<td>6</td>
</tr>
<tr>
<td>2^n</td>
<td>n + 1</td>
</tr>
</tbody>
</table>

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
def num_ancestors(p):
    """Returns: num of known ancestors
    Pre: p is a Person"""
    # 1. Handle small data.
    # No mom or dad (no ancestors)

    # 2. Break into two parts
    # Has mom or dad
    # Count ancestors of each one
    # (plus mom, dad themselves)

    # 3. Combine the result
def num_ancestors(p):
    ""
    Returns: num of known ancestors
    Pre: p is a Person"
    # 1. Handle small data.
    if p.mom == None and p.dad == None:
        return 0

    # 2. Break into two parts
    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)

    # 3. Combine the result
    return moms+dads
Is All Recursion Divide and Conquer?

- Divide and conquer implies two halves “equal”
  - Performing the same check on each half
  - With some optimization for small halves
- Sometimes we are given a recursive definition
  - Math formula to compute that is recursive
  - String definition to check that is recursive
  - Picture to draw that is recursive
  - Example: \( n! = n \cdot (n-1)! \)
- In that case, we are just implementing definition
Example: Palindromes

- String with \( \geq 2 \) characters is a palindrome if:
  - its first and last characters are equal, and
  - the rest of the characters form a palindrome

- **Example:**

  ![Example string](image)

  has to be the same

  has to be a palindrome

- **Function to Implement:**

  ```python
def ispalindrome(s):
  """Returns: True if s is a palindrome"""
  ```
Example: Palindromes

- String with \( \geq 2 \) characters is a palindrome if:
  - its first and last characters are equal, and
  - the rest of the characters form a palindrome

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

    # Halves not the same; not divide and conquer
    ends = s[0] == s[-1]
    middle = is_palindrome(s[1:-1])
    return ends and middle
```

Recursive case
Recursive Functions and Helpers

def ispalindrome2(s):
    """Returns: True if s is a palindrome
    Case of characters is ignored."""
    if len(s) < 2:
        return True
    ends = equals_ignore_case(s[0], s[-1])
    middle = ispalindrome(s[1:-1])
    return ends and middle
def ispalindromes2(s):
    """Returns: True if s is a palindrome
    Case of characters is ignored."""
    if len(s) < 2:
        return True
    ends = equals_ignore_case(s[0], s[-1])
    middle = ispalindromes(s[1:-1])
    return ends and middle
Recursive Functions and Helpers

```python
def ispalindrome2(s):
    """Returns: True if s is a palindrome
    Case of characters is ignored."""
    if len(s) < 2:
        return True
    # Halves not the same; not divide and conquer
    ends = equals_ignore_case(s[0], s[-1])
    middle = ispalindrome(s[1:-1])
    return ends and middle

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

Use helper functions!
- Pull out anything not part of the recursion
- Keeps your code simple and easy to follow
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
    # Combine left and right
    if s[0] in string.letters:
        return s[0]+depunct(s[1:])
    # Ignore left if it is not a letter
    return depunct(s[1:])
```

Use helper functions!
- Sometimes the helper is a recursive function
- Allows you break up problem in smaller parts
Example: Space Filling Curves

• 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

Starts Here

Ends Here

Challenge
Hilbert’s Space Filling Curve

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 $\infty$, curve fills box
“Turtle” Graphics: Assignment A4

- Turn
- Move
- Draw Line
- Change Color