Lecture 14

Recursion
## Announcements for Today

### Prelim 1
- Tonight at 5:15 OR 7:30
  - A–D (5:15, Uris G01)
  - E–K (5:15, Statler)
  - L–P (7:30, Uris G01)
  - Q–Z (7:30, Statler)
- Graded by noon on Sun
  - Scores will be in CMS
  - In time for drop date

### Other Announcements
- Reading: 5.8 – 5.10
- Assignment 3 now graded
  - **Mean** 93.4, **Median** 98
  - **Time**: 7 hrs, **StdDev**: 3.5 hrs
  - But only 535 responses
- Assignment 4 posted Friday
  - Parts 1-3: Can do already
  - Part 4: material from today
  - Due two weeks from today

10/11/18  Recursion
Recursion

- **Recursive Definition:**
  A definition that is defined in terms of itself

- **Recursive Function:**
  A function that calls itself (directly or indirectly)

**PIP** stands for “**PIP** Installs Packages”
A Mathematical Example: Factorial

- Non-recursive definition:
  \[ n! = n \times (n-1) \times \ldots \times 2 \times 1 = n (n-1) \times \ldots \times 2 \times 1 \]

- Recursive definition:
  \[ n! = n (n-1)! \quad \text{for } n \geq 0 \]
  \[ 0! = 1 \]

What happens if there is no base case?
Factorial as a Recursive Function

```python
def factorial(n):
    '''Returns: factorial of n.
    Pre: n ≥ 0 an int'''
    if n == 0:
        return 1
    return n * factorial(n - 1)
```

- \( n! = n \times (n-1)! \)
- \( 0! = 1 \)

What happens if there is no base case?
Example: Fibonacci Sequence

- Sequence of numbers: $1, 1, 2, 3, 5, 8, 13, \ldots$
  \[ a_0 \ a_1 \ a_2 \ a_3 \ a_4 \ a_5 \ a_6 \]
  - Get the next number by adding previous two
  - What is $a_8$?

A: $a_8 = 21$
B: $a_8 = 29$
C: $a_8 = 34$
D: None of these.
Example: Fibonacci Sequence

- Sequence of numbers: 1, 1, 2, 3, 5, 8, 13, ...

\[a_0 \quad a_1 \quad a_2 \quad a_3 \quad a_4 \quad a_5 \quad a_6\]

- Get the next number by adding previous two

- What is \(a_8\)?

<table>
<thead>
<tr>
<th>Option</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>(a_8 = 21)</td>
</tr>
<tr>
<td>B</td>
<td>(a_8 = 29)</td>
</tr>
<tr>
<td>C</td>
<td>(a_8 = 34) correct</td>
</tr>
<tr>
<td>D</td>
<td>None of these.</td>
</tr>
</tbody>
</table>

A: \(a_8 = 21\)
B: \(a_8 = 29\)
C: \(a_8 = 34\)  correct
D: None of these.
Example: Fibonacci Sequence

• Sequence of numbers: 1, 1, 2, 3, 5, 8, 13, ...
  \[ a_0 \ a_1 \ a_2 \ a_3 \ a_4 \ a_5 \ a_6 \]
  - Get the next number by adding previous two
  - What is \( a_8 \)?

• Recursive definition:
  - \( a_n = a_{n-1} + a_{n-2} \)  \[ \text{Recursive Case} \]
  - \( a_0 = 1 \) \[ \text{Base Case} \]
  - \( a_1 = 1 \) \[ \text{(another) Base Case} \]

Why did we need two base cases this time?
Fibonacci as a Recursive Function

```python
def fibonacci(n):
    """Returns: Fibonacci no. \( a_n \)
    Precondition: \( n \geq 0 \) an int""
    if n <= 1:
        return 1
    return fibonacci(n-1) + fibonacci(n-2)
```

Note difference with base case conditional.
Fibonacci as a Recursive Function

```python
def fibonacci(n):
    """Returns: Fibonacci no. \( a_n \)
    Precondition: n \( \geq 0 \) an int""
    if n <= 1:
        return 1
    return (fibonacci(n-1)+
            fibonacci(n-2))
```

- Function that calls itself
  - Each call is new frame
  - Frames require memory
  - \( \infty \) calls = \( \infty \) memory
Fibonacci: # of Frames vs. # of Calls

- Fibonacci is very inefficient.
  - $\text{fib}(n)$ has a stack that is always $\leq n$
  - But $\text{fib}(n)$ makes a lot of redundant calls
Fibonacci: # of Frames vs. # of Calls

- Fibonacci is very inefficient.
  - $\text{fib}(n)$ has a stack that is always $\leq n$
  - But $\text{fib}(n)$ makes a lot of redundant calls

Path to end = the call stack
Recursion vs Iteration

- **Recursion** is *provably equivalent* to **iteration**
  - Iteration includes **for-loop** and **while-loop** (later)
  - Anything can do in one, can do in the other
- But some things are easier with recursion
  - And some things are easier with iteration
- Will **not** teach you when to choose recursion
  - This is a topic for more advanced classes
- We just want you to **understand the technique**
Recursion is best for Divide and Conquer

Goal: Solve problem P on a piece of data
Recursion is best for **Divide and Conquer**

**Goal:** Solve problem $P$ on a piece of data

**Idea:** Split data into two parts and solve problem $P$
**Recursion is best for Divide and Conquer**

**Goal**: Solve problem P on a piece of data

**Idea**: Split data into two parts and solve problem

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

**Combine Answer!**
Divide and Conquer Example

Count the number of 'e's in a string:

```
  p e n n e
```

Two 'e's

```
  p e  +  n n e
```

One 'e'

One 'e'
Divide and Conquer Example

Count the number of 'e's in a string:

```
penne
```

Two 'e's

```
p  
```

Zero 'e's

```
enne
```

Two 'e's
Divide and Conquer Example

Count the number of 'e's in a string:

Will talk about *how* to break-up later

Zero 'e's

Two 'e's
Three Steps for Divide and Conquer

1. **Decide what to do on “small” data**
   - Some data cannot be broken up
   - Have to compute this answer directly

2. **Decide how to break up your data**
   - Both “halves” should be smaller than whole
   - Often no wrong way to do this (next lecture)

3. **Decide how to combine your answers**
   - Assume the smaller answers are correct
   - Combining them should give bigger answer
def num_es(s):
    """Returns: # of 'e's in s""
    # 1. Handle small data
    if s == '':
        return 0
    elif len(s) == 1:
        return 1 if s[0] == 'e' else 0

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

    # 3. Combine the result
    return left + right

    """Short-cut"" for
    if s[0] == 'e':
        return 1
    else:
        return 0

s[0] s[1:]
\[p\] e n n e
0 + 2
Divide and Conquer Example

```python
def num_es(s):
    """Returns: # of 'e's in s""

    # 1. Handle small data
    if s == "":
        return 0
    elif len(s) == 1:
        return 1 if s[0] == 'e' else 0

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

    # 3. Combine the result
    return left + right

"""Short-cut"" for
if s[0] == 'e':
    return 1
else:
    return 0
```

### Example
- **s[0]**: `p e n n e`
- **s[1:]**: `e n n e`
- **Final Result**: `0 + 2`

**Recursion**
def num_es(s):
    """Returns: # of 'e's in s""
    # 1. Handle small data
    if s == '':
        return 0
    elif len(s) == 1:
        return 1 if s[0] == 'e' else 0
    # 2. Break into two parts
    left = num_es(s[0])
    right = num_es(s[1:]):
    # 3. Combine the result
    return left + right

    """Short-cut""" for
    if s[0] == 'e':
        return 1
    else:
        return 0
def num_es(s):
    """Returns: # of 'e's in s""
    # 1. Handle small data
    if s == '':
        return 0
    elif len(s) == 1:
        return 1 if s[0] == 'e' else 0
    # 2. Break into two parts
    left = num_es(s[0])
    right = num_es(s[1:])
    # 3. Combine the result
    return left+right

    """Short-cut"" for
    if s[0] == 'e':
        return 1
    else:
        return 0

s[0]   s[1:]
  p  e  n  n  e

0  +  2
def num_es(s):
    """Returns: # of 'e's in s"""
    # 1. Handle small data
    if s == '':
        return 0
    elif len(s) == 1:
        return 1 if s[0] == 'e' else 0
    # 2. Break into two parts
    left = num_es(s[0])
    right = num_es(s[1:])
    # 3. Combine the result
    return left + right
def deblank(s):
    """Returns: s but with its blanks removed"""

1. Decide what to do on “small” data
   - If it is the empty string, nothing to do
     
     ```python
     if s == ":
         return s
     ```

   - If it is a single character, delete it if a blank
     
     ```python
     if s == ": # There is a space here
         return " # Empty string
     else:
         return s
     ```
Exercise: Remove Blanks from a String

```python
def deblank(s):
    """Returns: s but with its blanks removed"""

2. Decide how to break it up

    left = deblank(s[0])  # A string with no blanks
    right = deblank(s[1:])  # A string with no blanks

3. Decide how to combine the answer

    return left + right  # String concatenation
```

10/11/18
Recursion
```python
def deblank(s):
    """Returns: s w/o blanks"""
    if s == '':
        return s
    elif len(s) == 1:
        return '' if s[0] == ' ' else s
    left = deblank(s[0])
    right = deblank(s[1:])
    return left + right
```

10/11/18

Recursion
Putting it All Together

```python
def deblank(s):
    """Returns: s w/o blanks"""
    if s == '':
        return s
    elif len(s) == 1:
        return '' if s[0] == ' ' else s
    else:
        left = deblank(s[0])
        right = deblank(s[1:])
        return left + right
```

Base Case

Recursive Case
def deblank(s):
    """Returns: s w/o blanks"""
    if s == 
        return s
    elif len(s) == 1:
        return " if s[0] == 
    left = deblank(s[0])
    right = deblank(s[1:])
    return left+right

Needed second base case to handle s[0]
def deblank(s):
    """Returns: s w/o blanks"""
    if s == '':
        return s
    left = s[0]
    if s[0] == ' ':
        left = ''
    right = deblank(s[1:])
    return left+right

Eliminate the second base by combining

Less recursive calls
Following the Recursion

deblank

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
</table>
Following the Recursion

deblank

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
</table>


deblank

|   | a | b | c |
Following the Recursion

deblank

\[
\begin{array}{c}
\text{deblank} \\
\hspace{1cm} \boxed{\text{a}} \boxed{\text{b}} \boxed{\text{c}} \\
\hspace{1cm} \boxed{\text{a}} \boxed{\text{b}} \boxed{\text{c}} \\
\hspace{1cm} \boxed{\text{a}} \boxed{\text{b}} \boxed{\text{c}}
\end{array}
\]
Following the Recursion

deblank

\[
\begin{array}{ccc}
\text{a} & \text{b} & \text{c} \\
\end{array}
\]

deblank

\[
\begin{array}{ccc}
\text{a} & \text{b} & \text{c} \\
\end{array}
\]

deblank

\[
\begin{array}{ccc}
\text{a} & \text{b} & \text{c} \\
\end{array}
\]

deblank

\[
\begin{array}{ccc}
\text{b} & \text{c} \\
\end{array}
\]

deblank

\[
\begin{array}{ccc}
\text{b} & \text{c} \\
\end{array}
\]
Following the Recursion

debblank

\[
\begin{array}{ccc}
\text{a} & \text{b} & \text{c} \\
\end{array}
\]

debblank

\[
\begin{array}{ccc}
\text{a} & \text{b} & \text{c} \\
\end{array}
\]

debblank

\[
\begin{array}{cc}
\text{b} & \text{c} \\
\end{array}
\]

debblank

\[
\begin{array}{c}
\text{c} \\
\end{array}
\]

10/11/18
Following the Recursion
Following the Recursion

deblank

\[ \text{deblank} \quad \begin{array}{ccc} a & b & c \end{array} \]

\[ \begin{array}{ccc} a & b & c \end{array} \]

\[ a \quad \begin{array}{cc} b & c \end{array} \]

\[ \begin{array}{cc} b & c \end{array} \]

\[ b \quad \begin{array}{c} c \end{array} \]

\[ \begin{array}{c} c \end{array} \]
Following the Recursion
Following the Recursion

deblank

a
b

c

deblank

a
b

c

deblank

b

c

deblank

b

c

deblank

c

deblank

c

✗

10/11/18

Recursion
Following the Recursion

deblank: $\begin{array}{ccc} a & b & c \\ \end{array}$
deblank: $\begin{array}{ccc} a & b & c \\ b & c \\ \end{array}$
deblank: $\begin{array}{cc} b & c \\ \end{array}$
deblanc: $\begin{array}{c} c \\ \end{array}$

10/11/18

Recursion
Following the Recursion

defblank

\[ \text{a} \quad \text{b} \quad \text{c} \]

\[ \text{a} \quad \text{b} \quad \text{c} \]

\[ \text{a} \quad \text{b} \quad \text{c} \]

\[ \text{b} \quad \text{c} \]

\[ \text{b} \quad \text{c} \]

\[ \text{c} \]

\[ \text{c} \]

\[ \text{c} \]

\[ \text{c} \]

\[ \text{c} \]

\[ \text{c} \]

\[ \text{c} \]

\[ \text{c} \]

\[ \text{c} \]

\[ \text{c} \]

\[ \text{c} \]

\[ \text{c} \]
Following the Recursion

10/11/18

Recursion
Following the Recursion

deblank

[Diagram showing recursive process with boxes labeled 'a', 'b', 'c']

10/11/18
def deblank(s):
    """Returns: s w/o blanks"""
    if s == '':
        return s
    left = s[0]
    if s[0] == ' ':
        left = ''
    right = deblank(s[1:])
    return left + right

Real work done here
def deblank(s):
    
    """Returns: s w/o blanks"""

    if s == '':
        return s

    left = s
    if s[0] in string.whitespace:
        left = ''

    right = deblank(s[1:])

    return left+right

Real work done here

Module string has special constants to simplify detection of whitespace and other characters.
Next Time: Breaking Up Recursion