C.R.J.!
This is the last CS 5 lecture you’ll ever "need"!

On Warner Brothers’ insistence, we affirm that this ‘C’ does not stand for ‘Chamber’ and ‘S’ does not stand for ‘Secrets.’

* Caution: do not take this statement too literally or it is possible find yourself in twice as many CS 5 lectures as you need!
Recursion example: \( \text{numis}(s) \)

Total # of i's in
'i<3five'

\# of i's in 'i'
S[0]

\# of i's in '<3five'
S[i: ]
Recursion example: \( \text{numis}(s) \)

- total # of i's in 'alien'
- is
- # of i's in 'a'
- +
- # of i's in 'lien'
Recursion example: \( \text{numis}(s) \)

- total # of i's in \('\text{aliiien}'\)
- is
- # of i's in \('a'\)
- +
- # of i's in \('\text{liiiien}'\)
Recursion example: $\text{numis}(s)$

Analysis...

<table>
<thead>
<tr>
<th>total # of i's in 'aliiiiien'</th>
</tr>
</thead>
<tbody>
<tr>
<td>is</td>
</tr>
<tr>
<td># of i's in 'a'</td>
</tr>
<tr>
<td>+</td>
</tr>
<tr>
<td># of i's in 'liiiiiien'</td>
</tr>
</tbody>
</table>

...via self-similarity!
This is the last CS 5 lecture you’ll ever "need"!*  

Hw #1 due this Monday, 9/17, at 11:59 pm

Recursion  
As close as CS gets to magic

HMC’s legal counsel requires us to include these footnotes...

☞ On Warner Brothers' insistence, we affirm that this 'C' does not stand for 'Chamber' and 'S' does not stand for 'Secrets.'

* Caution: do not take this statement too literally or it is possible find yourself in twice as many CS 5 lectures as you need!
Career Fair + CS5!
if you attended lab and submit pr1+pr2:
you get full credit for hw1pr1 and hw1pr2

else:
you should complete the two lab problems, pr1 + pr2

either way: submit pr1 + pr2
complete and submit hw1pr3

Extra Credit: Pig Latin / CodingBat
This week's reading **on data**...

The End of Theory: The Data Deluge Makes the Scientific Method Obsolete

By Chris Anderson 06.23.08

*Illustration: Marien Banijes*

**THE PETABYTE AGE:**
Sensors everywhere. Infinite storage. Clouds of processors. Our ability to capture, warehouse, and understand massive amounts of data is changing science, medicine, business, and technology. As our collection of facts and figures grows, so will the opportunity to find answers to fundamental questions. Because in the era of big data, more isn’t just more. More is different.

"All models are wrong, but some are useful."

So proclaimed statistician George Box 30 years ago, and he was right. But what choice did we have? Only models, from cosmological equations to theories of human behavior, seemed to be able to consistently, if imperfectly, explain the world around us. Until now. Today companies like Google, which have grown up in an era of massively abundant data, don’t have to settle for wrong models. Indeed, they don’t have to settle for models at all.
Computation's Dual Identity

But what does the stuff on this side look like?
Computation's Dual Identity

accessed through \textit{functions}…

\textbf{Functions!}

It's no coincidence this starts with \textit{fun}!
C.R.J.!
Functioning across disciplines

**procedure**

```python
def g(x):
    return x**100
```

**structure**

\[ g(x) = x^{100} \]

**CS's googolizer**

defined by *what it does*

+ what follows behaviorally

**Math's googolizer**

defined by *what it is*

+ what follows logically
Giving names to data **helps f'ns**

def flipside(s):
    """ flipside(s): swaps s's sides!
    input s: a string
    """
    x = len(s)//2
    return s[x:] + s[:x]

This idea is the key to **your** happiness!
Giving names to data helps f'ns

```python
def flipside(s):
    '''flipside(s): swaps s's sides!
    input s: a string
    '''
    x = len(s) // 2
    return s[x:] + s[:x]
```

This idea is the key to your happiness!
I'm happy about this, too!

OK: we humans work better with named variables.

But -- why would even computers "prefer" the top version, too?

Aargh!

```python
def flipside(s):
    x = len(s)//2
    return s[x:] + s[:x]
```

```python
def flipside(s):
    return s[len(s)//2:] + s[:len(s)//2]
```
def flipside(s):
    """ flipside(s): swaps s's sides!
        input s: a string
    """
    x = len(s)/2
    return s[x:] + s[:x]

# Tests!
assert flipside('homework') == 'workhome'
assert flipside('poptart') == 'tartpop'

print(" petscar ~", flipside('carpets'))
print("    cs5! ~", flipside('5!cs'))

We provide tests (for now...)
def convertFromSeconds(s):
    # total seconds
    """ convertFromSeconds(s): Converts an integer # of seconds into a list of [days, hours, minutes, seconds]
    input s: an int
    """

    days = s // (24*60*60)  # total days
    s = s % (24*60*60)      # remainder s
    hours = s // (60*60)    # total hours
    s = s % (60*60)        # remainder s
    minutes = s // 60       # total minutes
    s = s % 60             # remainder s

    return [days, hours, minutes, s]
def convertFromSeconds(s):
    """ convertFromSeconds(s): Converts an integer number of seconds into a list of [days, hours, minutes, seconds]
    input s: an int
    """
    days = s // (24*60*60)  # total days
    s = s % (24*60*60)  # remainder s
    hours = s // (60*60)  # total hours
    s = s % (60*60)  # remainder s
    minutes = s // 60  # total minutes
    s = s % 60  # remainder s
    return [days, hours, minutes, s]
return vs. print

```python
def dbl(x):
    '''dubs x?'''
    return 2*x

dbl(20)

ans = dbl(20)
```

```python
def dblPR(x):
    '''dubs x?'''
    print(2*x)

dblPR(20)

ans = dblPR(20)
```

What's the difference ?!
```python
def dbl(x):
    """ dbls x? """
    return 2*x

def dblPR(x):
    """ dbls x? """
    print(2*x)

ans = dbl(20) + 2
    this is a value for further use!

ans = dblPR(20)+2
    this turns lightbulbs on!

print changes pixels on the screen...

return yields the function call's value ...
```

... which the shell then prints!
how software passes information from function to function...

changes the pixels (little lightbulbs) on your screen
return > print

how software *passes information* from function to function...

changes the pixels (little *lightbulbs*) on
def demo(x):
    y = x/3
    z = g(y)
    return z + y + x

def g(x):
    result = 4*x + 2
    return result

def f(x):
    if x == 0:
        return 12
    else:
        return f(x-1) + 10*x

What is \texttt{demo(15)} here?

What is \texttt{f(2)} here?
How functions work...

```python
def demo(x):
    y = x/3
    z = g(y)
    return z + y + x

def g(x):
    result = 4*x + 2
    return result
```

"the stack"

they stack.
def demo(x):
    y = x/3
    z = g(y)
    return z + y + x

def g(x):
    result = 4*x + 2
    return result

How functions work...

call: demo(15)

local variables:
    x = 15
    y = 5
    z = ?????

"the stack"

they stack.
def demo(x):
    y = x/3
    z = g(y)
    return z + y + x

def g(x):
    result = 4*x + 2
    return result

How functions work...

call: demo(15)  
local variables:  
x = 15  
y = 5  
z = ?????  

returns 22

call: g(5)  
local variables:  
x = 5
result = 22
returns 22

they stack.
def demo(x):
    y = x/3
    z = g(y)
    return z + y + x

def g(x):
    result = 4*x + 2
    return result

How functions work...

They stack.
def demo(x):
    y = x/3
    z = g(y)
    return z + y + x

def g(x):
    result = 4*x + 2
    return result
def demo(x):
    y = x/3
    z = g(y)
    return z + y + x

def g(x):
    result = 4*x + 2
    return result

How functions work...

15

call: demo(15)  
local variables: x = 15  
y = 5  
z = 22  
return 42
How functions work...

```
def demo(x):
    y = x/3
    z = g(y)
    return z + y + x

def g(x):
    result = 4*x + 2
    return result
```

afterwards, the stack is empty..., but ready if another function is called

they stack.
def f(x):
    if x == 0:
        return 12
    else:
        return f(x-1) + 10*x

what's f(2)?
def f(x):
    if x == 0:
        return 12
    else:
        return f(x-1) + 10*x

How functions work...

"the stack"

stack frame:
call: f(2)
local variables:
  x = 2
  need f(1)
def f(x):
    if x == 0:
        return 12
    else:
        return f(x-1) + 10*x

How functions work...

The stack

call: f(2)  
local variables: x = 2   
need f(1)

call: f(1)  
local variables: x = 1   
need f(0)
def f(x):
    if x == 0:
        return 12
    else:
        return f(x-1) + 10*x

0

call: f(2)  
local variables:  
    x = 2  
    need f(1)  

stack frame

1

call: f(1)  
local variables:  
    x = 1  
    need f(0)  

stack frame

0

call: f(0)  
local variables:  
    x = 0  
    returns 12  

returns
```python
def f(x):
    if x == 0:
        return 12
    else:
        return f(x-1) + 10*x
```

How functions work...

0

```
def f(x):
    if x == 0:
        return 12
    else:
        return f(x-1) + 10*x
```

"the stack"

```
call: f(2)  
local variables:  
x = 2
need f(1)
```

```
call: f(1)  
local variables:  
x = 1
need f(0)
```

```
call: f(0)  
local variables:  
x = 0
returns 12
```
def f(x):
    if x == 0:
        return 12
    else:
        return f(x-1) + 10*x

def f(x):
    if x == 0:
        return 12
    else:
        return f(x-1) + 10*x

How do we compute the result?
How functions work...

```
def f(x):
    if x == 0:
        return 12
    else:
        return f(x-1) + 10*x
```

Where does that result go?
def f(x):
    if x == 0:
        return 12
    else:
        return f(x-1) + 10*x

f(2)

x = 2
need f(1)

f(1)

x = 1
f(0) = 12
result = 22

result = 22
How functions work...

```
def f(x):
    if x == 0:
        return 12
    else:
        return f(x-1) + 10*x
```

Call: `f(2)`

Local variables:
- `x = 2`
- `f(1) = 22`
- `result = 2`
def f(x):
    if x == 0:
        return 12
    else:
        return f(x-1) + 10*x

call: f(2)

local variables:
    x = 2
    f(1) = 22
    result = 42

which then gets returned...
How functions work...

def f(x):
    if x == 0:
        return 12
    else:
        return f(x-1) + 10*x

call: f(2)
for x = 2:
  f(2)
  f(1)
result = 42

the result then gets returned...
def f(x):
    if x == 0:
        return 12
    else:
        return f(x-1) + 10*x

How functions work...

again, the stack is empty, but ready if another function is called...
How functions work...

Functions are software's cells ... ... each one is a self-contained computational unit!

def f(x):
    if x == 0:
        return 12
    else:
        return f(x-1) + 10*x

again, the stack is empty, but ready if another function is called...

functions stack.
def f(x):
    if x == 0:
        return 12
    else:
        return f(x-1) + 10*x

Pass these eastward!
... each one is a self-contained computational unit!

functions stack.
sequential self-similar

iteration recursion

problem-solving paradigms
Thinking *sequentially*

factorial

\[
\begin{align*}
\text{math} & \quad 5! & = & \quad 120 \\
\text{cs} & \quad \text{fac}(5) & = & \quad 5 \times 4 \times 3 \times 2 \times 1 \\
\text{fac}(N) & = & N \times (N-1) \times \ldots \times 3 \times 2 \times 1
\end{align*}
\]
Thinking *sequentially*

factorial

\[ 5! = 120 \]

\[ \text{cs} \quad \text{fac}(5) = 5 \times 4 \times 3 \times 2 \times 1 \]

\[ \text{fac}(N) = N \times (N-1) \times \ldots \times 3 \times 2 \times 1 \]
Thinking *recursively*

**factorial**

\[ 5! = 120 \]

\[ \text{fac}(5) = 5 \times 4 \times 3 \times 2 \times 1 \]

\[ \text{fac}(N) = N \times (N-1) \times \ldots \times 3 \times 2 \times 1 \]

**Can we express \text{fac} \ with a smaller version of itself?**
Recursion ~
self-similarity

$$\text{fac}(5) = 5 \times 4 \times 3 \times 2 \times 1$$

$$\text{fac}(5) = 5 \times \text{fac}(4)$$

$$\text{fac}(N) = N \times (N-1) \times \ldots \times 3 \times 2 \times 1$$

$$\text{fac}(N) = N \times \text{fac}(N-1)$$

can we express \text{fac} w/ a smaller version of itself?

We're done!?
Warning: this is legal!

```python
def fac(N):
    return N * fac(N-1)
```

I wonder how this code will STACK up!? 

```python
def facBad(N):
    return N * facBad(N-1)
```
unionAll resulting in StackOverflow

I've made some progress with my own question (how to load a dataframe from a python requests stream that is downloading a csv file?) on StackOverflow, but I'm receiving a StackOverflow error:

```python
import requests
import numpy as np
import pandas as pd
import sys
```
Recursion

the dizzying dangers of having no base case!

This "works" ~ but doesn't work!

```python
def fac(N):
    return fac(N)
```
def fac(N):
    return fac(N)

Recursion: the dizzying dangers of having no base case!

This "works" — but doesn’t work!
def facBad(N):
    return N * facBad(N-1)

calls to facBad will "never" stop: there's no BASE CASE

Make sure you have a base case

a.k.a. "escape hatch"
def fac(N):
    if N == 0:
        return 1
    else:
        return N * fac(N-1)

Thinking recursively...

Base case

Recursive case (too short?)
def fac(N):

    if N == 0:
        return 1

    else:
        return N * fac(N-1)

Base case

Recursive case  
(too short?)

How can this multiply N by something that hasn't happened yet?!!
def fac(N):
    if N <= 1:
        return 1
    else:
        rest = fac(N-1)
        return N*rest

Conceptual

Acting recursively

def fac(N):
    if N <= 1:
        return 1
    else:
        rest = fac(N-1)
        return N*rest

Actual

this recursion happens first!

hooray for variables!
def fac(N):
    if N <= 1:
        return 1.0
    else:
        return N * fac(N-1)

Behind the curtain:  
*how recursion works...*

\[
\begin{align*}
\text{fac}(5) &= 5 \times \text{fac}(4) \\
5 \times \text{fac}(4) &= 4 \times \text{fac}(3) \\
4 \times \text{fac}(3) &= 3 \times \text{fac}(2) \\
3 \times \text{fac}(2) &= 2 \times \text{fac}(1) \\
2 \times \text{fac}(1) &= 1.0
\end{align*}
\]
Behind the curtain: how recursion works...

```python
def fac(N):
    if N <= 1:
        return 1.0
    else:
        return N * fac(N-1)
```

1. **stack frame with N = 1**: $1.0$
2. **stack frame with N = 2**: $2 \times \text{fac}(1)$
3. **stack frame with N = 3**: $3 \times \text{fac}(2)$
4. **stack frame with N = 4**: $4 \times \text{fac}(3)$
5. **stack frame with N = 5**: $5 \times \text{fac}(4)$

---

$\text{fac}(5)$
Behind the curtain: how recursion works...

```python
def fac(N):
    if N <= 1:
        return 1.0
    else:
        return N * fac(N-1)
```

Stack frame with N = 5

- 5 * fac(4)

Stack frame with N = 4

- 4 * fac(3)

Stack frame with N = 3

- 3 * fac(2)

Stack frame with N = 2

- 2 * 1.0
Behind the curtain: 
*how recursion works...*

def fac(N):
    if N <= 1:
        return 1.0
    else:
        return N * fac(N-1)

fac(5)

5 * fac(4)

4 * fac(3)

3 * 2.0
Behind the curtain: *how recursion works...*

```python
def fac(N):
    if N <= 1:
        return 1.0
    else:
        return N * fac(N-1)
```

Stack frame with $N = 5$

5 * fac(4)

Stack frame with $N = 4$

4 * 6.0
Behind the curtain: *how recursion works*...

```python
def fac(N):
    if N <= 1:
        return 1.0
    else:
        return N * fac(N-1)
```

stack frame with $N = 5$

$\text{fac}(5)$

5 * 24.0
Behind the curtain: *how recursion works*...

```python
def fac(N):
    if N <= 1:
        return 1.0
    else:
        return N * fac(N-1)
```

`fac(5)`

120.0  

*complete!*

But is recursion for real?!
Recursion's *conceptual* challenge?

You need to see BOTH the *self-similar pieces* AND the *whole thing* simultaneously!

... because it's completely *self-sufficient*!
Recursion

Base Case

Self-similar design

problem-solving paradigm
Recursion

Base Case

Self-similar design

Next: recursive-function DESIGN →
Base case:
fac(0) should return 1
def fac(x):
    """ factorial! Recursively! """
    if x == 0:
        return 1
    else:
        return x * fac(x - 1)
plusone(5) is $1+1+1+1+1$.

value of $1$ + ___.

Base case:
plusone(0) should return ___.
plusone(5) is
value of 1+1+1+1+1
value of 1 +
value of 1+1+1+1
plusone(4)

Base case:
plusone(0) should return ___
def plusone(n):
    """
    returns n by adding 1's!
    """
    if n == 0:
        return _______________________
    else:
        return _______________________
def plusone(n):
    """
    returns n by adding 1's!
    """
    if n == 0:
        return 0
    else:
        return 1 + plusone(n-1)
value of $2 \times 2 \times 2 \times 2 \times 2$ is

value of $2$ \* 

Base case: $\text{pow}(2,0)$ should return __?
value of $2 \times 2 \times 2 \times 2 \times 2$ is $\text{pow}(2, 5)$

value of $2 \times 2 \times 2 \times 2 \times 2$ is $\text{pow}(b, p)$

Base case:
$\text{pow}(2, 0)$ should return __?
def pow(b, p):
    
    """b**p, defined recursively!""
    """
    if p == 0:
        return
    else:
        return

Extra! Can we also handle negative powers...?
def pow(b, p):
    """b**p, defined recursively!"""

    if p == 0:
        return 1.0
    elif p < 0:
        return b * pow(b, p - 1)
    else:
        return b * pow(b, p - 1)

Extra! Can we also handle negative powers...?
def pow(b,p):
    
    """
    b**p, defined recursively!
    """

    if p == 0:
        return 1.0

    elif p < 0:
        return 1.0/pow(b,-p)

    else:
        return b*pow(b,p-1)

Extra! Can we also handle negative powers...?
Recursion's advantage: It handles arbitrary structural depth – *all at once + on its own!*

As a hat, I'm recursive, too!

https://www.youtube.com/watch?v=8PhiSSnaUKk @ 1:11
https://www.youtube.com/watch?v=ybXlnVLtNi4   @ 0:08
https://www.youtube.com/watch?v=8PhisSnaUKk  @ 1:11

---

**Pomona Sends Survey To Students To Find Out Why They Don’t Take Surveys**

*Ima Firstyear*

Declining survey response rates at Pomona College prompted the administration to send students a new survey this week, which will assess students’ previous survey experiences and their survey preferences in hopes of explaining—and reversing—the decline.

“We know Pomona students have strong opinions about their education and their campus,” said Vice President and Dean of Students Miriam Feldblum. “But what we find is that when we offer students a chance to express those opinions via a general survey, we don’t get as many responses as we expect. We want to know why, and that’s why we’re sending out this survey.”

Students will be asked to self-identify at the start of the survey as a ‘frequent responder,’ ‘occasional responder’ or ‘forgot the password to my Pomona webmail account three months ago.’ According to Feldblum, these categories will help the administration create new strategies to engage more of the student population in responding to surveys.

The survey also addresses questions of methodology, incentive and access. It asks students to rank their preferences of survey provider, such as SurveyMonkey, Qualtrics and Google Forms, and to name their ideal survey prizes. It also asks students whether they would be more inclined to take school surveys via email, an iPhone app or voting machines in the dining halls complete with ‘I Surveyed!’ stickers.

Erika Bennett PO ’17 said she found some of the questions confusing.

“I had to pick my favorite assessment scale,” she said. “I had to rank ‘Scale of one to five,’ ‘Strongly Disagree to Strongly Agree’ and ‘Sad Face to Happy Face’ from least to most intuitive. But I’m not sure I did it correctly.’’

Bennett added that she did appreciate the chance to critique previous surveys.

“Just last month I took a survey with no progress bar at the bottom of each page,” she said. “I felt lost and confused. I’m glad there’s a real see SURVEY page 2...”
Recursion's advantage: It handles arbitrary structural depth – all at once + on its own!
Design patterns...

Recursion's a design - not a formula, **BUT**, these pieces are common:

\[ s = '\text{alien}' \]

\[ 'a' \quad '\text{liiien}' \]

\textit{in terms of }s, \textit{what are these pieces? (index! slice!)}
Design patterns...

Recursion's a design - not a formula, **BUT**, these pieces are common:

\[ s = 'aliien' \]

\[ 'a' \quad 'liien' \]

\[ s[0] \]

*handle the* **first**

Human!

\[ s[1:] \]

*recurse the* **rest**

Machine!
Design patterns...

Recursion's a design - not a formula, BUT, these pieces are common:

\[ L = [3, 1, 4, 1, 5, 9] \]

\[ L[0] \quad L[1:] \]

(handle the **first**)

(recurse the **rest**)

Human!

Machine!
Design patterns...

- Do one piece of work: \( L[0] \) or \( s[0] \)
- Recurse with the rest: \( L[1:] \) or \( s[1:] \)
- Combine! Make sure all types match...
- Handle base cases, with if ...
numis('xlii') is 
# of i's in 'xlii'

# of i's in 'x' + # of i's in 'lii'

Base case:
numis('') should return ____ ?
def numis(s):
    """ # of i's in s """
    if s == '':
        return 
    elif s[0] == 'i':
        return 
    else:
        return 

numis('xlii')
is
# of i's in 'xlii'
# of i's in 'x' + # of i's in 'lii'
def numis(s):
    ''' # of i's in s
    '''
    if s == '':
        return 0
    elif s[0] == 'i':
        return 1 + numis(s[1:]),
    else:
        return numis(s[1:])

What's really being added here?
Base case:
len('') should return ___?
def len(s):
    
    # returns the length of s
    
    if s == '':
        return __________
    
    else:
        return ________________

Extra! Can we also handle **LISTS**...?
def len(s):
    
    returns the length of s
    
    if s == '' or s == []:
        return 0
    
    else:
        return 1 + len(s[1:])

    one, plus... ... the length of the rest of s
A brief word from our sponsor, Mother Nature...

Like broccoli, recursion is "Good for You"™
Yes... and no. Are these rules for real?
But, do only *plants* get to be recursive?
There still has to be a base case...
or else!
or - one layer up!?
Leap before you look!

Try these four...
Python is...

>>> 'i' in 'team'
False

>>> 'cs' in 'physics'
True

>>> 'i' in 'alien'
True

>>> 3*'i' in 'alien'
False

>>> 42 in [41,42,43]
True

>>> 42 in [[42], '42']
False

I guess Python's the in thing
vwl('eerie')

# of vowels in 'eerie'

is

# of vowels in 'e' + # of vowels in 'erie'

Base case:
vwl('') should return ___?
def vwl(s):
    """ # of vowels in s """
    if s == '':
        return __________
    elif s[0] in 'aeiou':
        return __________
    else:
        return __________
keepvwl('pluto')
keep vowels in 'pluto'

keep vowels in 'p' + keep vowels in 'luto'

Base case:
keepvwl('') should return ___ ?
def keepvwl(s):
    """ returns ONLY the vowels in s! """
    if s == '
        return 
    elif s[0] in 'aeiou'
        return 
    else:
        return 

Base case:
if \( \text{len}(L) == 1 \), what should \( \text{max}(L) \) return?
def max(L):
    """ returns the max of L! """

    if len(L) == 1:
        return __________

    M = _____________ The max of the REST of L

    if _____________:
        return __________
    else:
        return __________
zeroest([\(-7, 5, 9, 2\)])

zeroest of

\([-7, 5, 9, 2]\)

is

either \(-7\)

or the zeroest

of \([5, 9, 2]\)

Base case:
if \(\text{len}(L) == 1\), what should \(\text{zeroest}(L)\) return?
```python
def zeroest(L):
    """ returns L's element nearest 0 """
    if len(L) == 1:
        return
    Z = zeroest(L[1:])
    if abs(L[0]) < abs(Z):
        return
    else:
        return
```

The zeroest of the REST of L
```python
def vwl(s):
    """ # of vowels in s """
    if s == '':
        return 0
    elif s[0] in 'aeiou':
        return 1 + vwl(s[1:])
    else:
        return vwl(s[1:])
```

What's really being added here?

What seven-letter s maximizes vwl(s)?
def keepvwl(s):
    """ returns ONLY the vowels in s!
    """
    if s == '':
        return ''
    elif s[0] in 'aeiou':
        return s[0] + keepvwl(s[1:])
    else:
        return keepvwl(s[1:])

What's really being added here?
def max(L):
    """ returns the max of L! """
    if len(L) == 1:
        return L[0]
    M = max(L[1:])
    if L[0] > M:
        return L[0]
    else:
        return M
def zeroest(L):
    """ returns L's element nearest 0 """

    if len(L) == 1:
        return L[0]

    Z = zeroest(L[1:])  # The zeroest of the REST of L

    if abs(L[0]) < abs(Z):
        return L[0]
    else:
        return Z
The key to understanding recursion is, first, to understand recursion.

- former CS 5 student

Good luck with Homework #1

tutors @ LAC + 4C's  Th/F/Sa/Su/Mon.