Recursion II
Announcements

• Practice Midterm is worth extra credit!
  - Practice with the exam tool (exam.cs61a.org)
  - Practice with creating a Zoom recording
  - You will submit the exam online, then submit a link to a recording.
  - Will be worth 2 of 56 points.
  - Should only take you 10 minutes. 😊
• Recursion is on the midterm, but not super advanced recursion.
Learning Objectives

• Write a recursive function
• Understand the base case and a recursive case
• Apply Recursive Functions to lists
Why Recursion?

• Recursive structures exist (sometimes hidden) in nature and therefore in data!
• It’s mentally and sometimes computationally more efficient to process recursive structures using recursion.
• Sometimes, the recursive definition is easier to understand or write, even if it is computationally slower.
The Recursive Process

- Recursive solutions involve two major parts:
  - Base case(s), the problem is simple enough to be solved directly
  - Recursive case(s). A recursive case has three components:
    - Divide the problem into one or more simpler or smaller parts
    - Invoke the function (recursively) on each part, and
    - Combine the solutions of the parts into a solution for the problem.
In words

• The sum of no numbers is zero
• The sum of $1^2$ through $n^2$ is the
  - sum of $1^2$ through $(n-1)^2$
  - plus $n^2$

```python
def sum_of_squares(n):
    if n < 1:
        return 0
    else:
        return sum_of_squares(n-1) + n**2
```
How does it work?

• Each recursive call gets its own local variables  
  - Just like any other function call
• Computes its result (possibly using additional calls)  
  - Just like any other function call
• Returns its result and returns control to its caller  
  - Just like any other function call
• The function that is called happens to be itself  
  - Called on a simpler problem
  - Eventually stops on the simple base case
Recursion With Lists

• Recursion over sequence length, rather than number magnitude

```python
def first(s):
    """Return the first element in a sequence.""
    return s[0]

def rest(s):
    """Return all elements in a sequence after the first""
    return s[1:]

def min_r(s):
    """Return minimum value in a sequence.""
    if len(s) == 1:
        return first(s)
    else:
        return min(first(s), min_r(rest(s)))
```
Recursion With Strings

```python
def reverse(s):
    """
    >>> reverse('hello')
    'olleh'
    >>> reverse(reverse('hello'))
    'hello'
    """
    if not s:
        return ''
    return s[len(s)-1] + reverse(s[:len(s)-1])
```

- Recursion over sequence length, rather than number magnitude
Local variables

- Each call has its own “frame” of local variables
- Let’s see the environment diagrams

```python
def sum_of_squares(n):
    n_squared = n**2
    if n < 1:
        return 0
    else:
        return n_squared + sum_of_squares(n-1)
```

https://goo.gl/CiFaUJ
Environments Example

```python
# Python 3.3

1. def sum_of_squares(n):
   2.     n_squared = n**2
   3.     if n == 1:
   4.         return 1
   5.     else:
   6.         return n_squared + sum_of_squares(n-1)

7. sum_of_squares(3)
```

[Diagram of Python code execution]

**Frames**

- Global frame
  - func sum_of_squares(n) [parent=Global]
  - sum_of_squares

**Objects**

- Global frame
  - sum_of_squares
- f1: sum_of_squares [parent=Global]
  - n: 3
Environments Example

Python 3.3

```python
1. def sum_of_squares(n):
2.     n_squared = n**2
3.     if n == 1:
4.         return 1
5.     else:
6.         return n_squared + sum_of_squares(n-1)
7. sum_of_squares(3)
```

Frames | Objects
---|---
Global frame | func sum_of_squares(n) [parent=Global]

- `sum_of_squares`  

f1: sum_of_squares [parent=Global]  

- `n` 3  
- `n_squared` 9

Python 3.3

```python
1. def sum_of_squares(n):
2.     n_squared = n**2
3.     if n == 1:
4.         return 1
5.     else:
6.         return n_squared + sum_of_squares(n-1)
7. sum_of_squares(3)
```

Frames | Objects
---|---
Global frame | func sum_of_squares(n) [parent=Global]

- `sum_of_squares`  

f1: sum_of_squares [parent=Global]  

- `n` 3  
- `n_squared` 9
Environments Example

```python
1 def sum_of_squares(n):
2     n_squared = n**2
3     if n == 1:
4        return 1
5     else:
6        return n_squared + sum_of_squares(n-1)
7
8 sum_of_squares(3)
```

Frames | Objects
---|---
Global frame | `func sum_of_squares(n) [parent=Global]`

1: `sum_of_squares [parent=Global]`<br>n = 3<br>n_squared = 9

2: `sum_of_squares [parent=Global]`<br>n = 2

Frames | Objects
---|---
Global frame | `func sum_of_squares(n) [parent=Global]`

1: `sum_of_squares [parent=Global]`<br>n = 3<br>n_squared = 9

2: `sum_of_squares [parent=Global]`<br>n = 2<br>n_squared = 4
Environments Example

```python
def sum_of_squares(n):
    n_squared = n**2
    if n == 1:
        return 1
    else:
        return n_squared + sum_of_squares(n-1)
sum_of_squares(3)
```

That has just executed:
```python
line to execute
```
Environments Example

```
Python 3.3
1 def sum_of_squares(n):
2   n_squared = n**2
3   if n == 1:
4       return 1
5   else:
6       return n_squared + sum_of_squares(n-1)
7
8 sum_of_squares(3)
```

```
Global frame
   sum_of_squares | func sum_of_squares(n) [parent=Global]

f1: sum_of_squares [parent=Global]
   n 3
   n_squared 9

f2: sum_of_squares [parent=Global]
   n 2
   n_squared 4

f3: sum_of_squares [parent=Global]
   n 1
   n_squared 1
```

`that has just executed`
`the line to execute`
Environments Example

Python 3.3

```python
1 def sum_of_squares(n):
2     n_squared = n**2
3     if n == 1:
4         return 1
5     else:
6         return n_squared + sum_of_squares(n-1)
7
8 sum_of_squares(3)
```

Frames

- Global frame
  - `func sum_of_squares(n) [parent=Global]`
    - `sum_of_squares`

- `f1: sum_of_squares [parent=Global]`
  - `n = 3`
  - `n_squared = 9`

- `f2: sum_of_squares [parent=Global]`
  - `n = 2`
  - `n_squared = 4`

- `f3: sum_of_squares [parent=Global]`
  - `n = 1`
  - `n_squared = 1`
  - `Return value = 1`
Environments Example

```python
def sum_of_squares(n):
    n_squared = n**2
    if n == 1:
        return 1
    else:
        return n_squared + sum_of_squares(n-1)

sum_of_squares(3)
```

Frames

<table>
<thead>
<tr>
<th>Global frame</th>
</tr>
</thead>
<tbody>
<tr>
<td>sum_of_squares</td>
</tr>
</tbody>
</table>

| f1: sum_of_squares [parent=Global] |
| n 3 |
| n_squared 9 |

| f2: sum_of_squares [parent=Global] |
| n 2 |
| n_squared 4 |
| Return value 5 |

| f3: sum_of_squares [parent=Global] |
| n 1 |
| n_squared 1 |
| Return value 1 |
Environments Example

Python 3.3

```python
1. def sum_of_squares(n):
2.     n_squared = n**2
3.     if n == 1:
4.         return 1
5.     else:
6.         return n_squared + sum_of_squares(n-1)
7. sum_of_squares(3)
```

Frames

Objects

- Global frame
  - `sum_of_squares`

- `f1: sum_of_squares [parent=Global]`
  - `n` 3
  - `n_squared` 9
  - `Return value` 14

- `f2: sum_of_squares [parent=Global]`
  - `n` 2
  - `n_squared` 4
  - `Return value` 5

- `f3: sum_of_squares [parent=Global]`
  - `n` 1
  - `n_squared` 1
  - `Return value` 1
Extra Examples (We’ll cover later, too.)
Tree Recursion: Preview for Post-Midterm

- Break the problem into multiple smaller sub-problems, and Solve them recursively

```python
def split(x, s):
    return [i for i in s if i <= x], [i for i in s if i > x]

def quicksort(s):
    """Sort a sequence - split it by the first element, sort both parts and put them back together.""
    if not s:
        return []
    else:
        pivot = s[0]
        smaller, bigger = split(pivot, s[1:])
        return quicksort(smaller) + [pivot] + quicksort(bigger)

>> qsort([3,3,1,4,5,4,3,2,1,17])
[1, 1, 2, 3, 3, 3, 4, 4, 5, 17]
```
QuickSort Example

[3, 3, 1, 4, 5, 4, 3, 2, 1, 17]

[3, 1, 3, 2, 1]

[1, 3, 2, 1] []

[1] [3, 2]

[] [] [2] [2]

[1] [1] [2, 3]

[1, 1, 2, 3]

[1, 1, 2, 3, 3]

[1, 1, 2, 3, 3, 3, 4, 4, 5, 17]