CS 134 Lecture 20: More Recursion
Announcements & Logistics

• **HW 6** on GLOW due Mon at 10pm
  • Good practice for short-code questions on exam
  • Practice on pencil and paper first

• Lab 7, 8, and 9 are **partner labs**
  • Pair programming is an important skill as well as a vehicle for learning

• Colloquium Today: Tim Randolph ’18
  • Theoretical computer science talk on the Subset Sum problem (a problem you may use a “brute-force” approach to solve recursively in a future assignment!)

Do You Have Any Questions?
Last Time

• Introduction to recursion
  • Recursion as a new problem solving paradigm
  • Recursion as an alternative to iteration
  • Recursive solution to a familiar problem (count elements in a list)
A recursive function is a function that calls itself.

A recursive approach to problem solving has two main parts:

- **Base case(s).** When the problem is so small, we can solve it directly, without having to reduce it any further.
- **Recursive step.** Does the following things:
  - Performs an action that contributes to the solution.
  - Reduces the problem to a smaller version of the same problem, and calls the function on this smaller subproblem.

The recursive step is a form of "wishful thinking" (also called the inductive hypothesis).
Understanding Recursive Functions

- Let's review a simple recursive function that gives us some intermediate feedback through `print` statements:
  - we'll write a recursive function to print integers from \( n \) down to 1
  - Recursive definition of countdown:
    - **Base case:** \( n = 1, \text{print}(n) \)
    - **Recursive rule:** `print(n), call count_down(n-1)`
      - **Perform one step**
      - **Reduce the problem (or make the problem “smaller”)**
Understanding Recursive Functions

• Recursive definition of countdown:

  • **Base case:** $n = 1$, print($n$)
  • **Recursive rule:** print($n$), count_down($n-1$)

```python
def count_down(n):
    '''Prints numbers from n down to 1'''
    # Base case: n == 1
    if n == 1:
        print(n)
    # Recursive case: n > 1:
    else:
        print(n)
        count_down(n-1)
```

```python
>>> result = count_down(5)
5
4
3
2
1
```
Understanding Recursive Functions

• Recursive functions seem to be able to reproduce looping behavior without writing any loops at all

• To understand what happens behind the scenes when a function calls itself, let’s review what happens when a function calls another function

• Conceptually we understand function calls through the function frame model
Review:
Function Frame Model
• Consider a simple function `square`
• What happens when `square(5)` is invoked?

```python
def square(x):
    return x**x
```
Review:
Function Frame Model

```python
>>> square(5)
```

```
25
```

```
square(5)
```

```
x  5
```

```
return x * x
```

```
Summary:
Function Frame Model

- When we **return** from a function frame "control flow" goes back to where the function call was made
- Function frame (and the local variables inside it) **are destroyed after the return**
- If a function does not have an explicit return statement, it returns **None** after all statements in the body are executed
Review:
Function Frame Model

• How about functions that call other functions?
  
```python
def sum_square(a, b):
    return square(a) + square(b)
```

• What happens when we call `sum_square(5, 3)`?
```python
def sum_square(a, b):
    return square(a) + square(b)
```

```python
>>> sum_square(5, 3)
```

```
square(5)
```

```python
x
```

```python
return x * x
```
def sum_square(a, b):
    return square(a) + square(b)

>>> sum_square(5, 3)
25
def sum_square(a, b):
    return square(a) + square(b)

>>> sum_square(5, 3)

5

a

3

b

return square(a) + square(b)

25

square(5)

x

5

x * x

square(3)

x

3

x * x

return x * x
def sum_square(a, b):
    return square(a) + square(b)

>>> sum_square(5,3)

25

25 + 9

square(5) + square(3)

5

3

return x * x

return x * x
def sum_square(a, b):
    return square(a) + square(b)

>>> sum(5, 3)
34
Function Frame Model to Understand count_down
```python
def count_down(n):
    '''Prints ints from n down to 1'''
    if n == 1:
        print(n)
    else:
        print(n)
        count_down(n-1)

>>> val = count_down(5)
5
4
3
2
1

>>> val = count_down(4)
4
3
2
1
```
```python
def count_down(n):
    if n == 1:
        print(n)
    else:
        print(n)
        count_down(n-1)
```

>>> val = count_down(4)
4
3
2
1

Base case reached!
```python
def count_down(n):
    if n == 1:
        print(n)
    else:
        print(n)
        count_down(n-1)

>>> val = count_down(4)
4
3
2
1
Base case reached!
```
```python
def count_down(n):
    if n == 1:
        print(n)
    else:
        print(n)
        count_down(n-1)
```

```python
>>> val = count_down(4)
4
3
2
1
Base case reached!
```
```python
>>> val = count_down(4)
4
3
2
1
```

Base case reached!
```python
>>> val = count_down(4)
4
3
2
1
```

Base case reached!
Recursion **GOTCHAs**!
GOTCHA #1

• If the problem that you are solving recursively **is not getting smaller**, that is, you are not getting closer to the base case --- **infinite recursion**!

• Never reaches the base case

```python
def count_down_gotcha(n):
    '''Prints ints from 1 up to n'''
    if n == 1:  # Base case
        print(n)
    else:       # Recursive case
        print(n)
        count_down_gotcha(n)
```

Subproblem not getting smaller!
GOTCHA #2

- Missing base case/unreachable base case--- another way to cause **infinite recursion**!

```python
def print_halves_gotcha(n):
    """Prints n, n/2, down to ... 1""
    if n >= 0:
        print(n)
        return print_halves_gotcha(n/2)
```

Always true!
"Maximum recursion depth exceeded"

- In practice, the infinite recursion examples will terminate when Python runs out of resources for creating function call frames, leads to a "maximum recursion depth exceeded" error message
Recursion vs. Iteration: sum_list
sum_list

• **Goal:** Write a function to sum up a list of numbers
• Iterative approach? (i.e., using loops?)
Iterative Approach to `sum_list`

- **Goal:** Write a function to sum up a list of numbers
- Iterative approach:

```python
def sum_list_iterative(num_lst):
    sum = 0
    for num in num_lst:
        sum += num
    return sum
```

```python
>>> sum_list_iterative([3, 4, 20, 12, 2, 20])
61
```
sum_list

- **Goal:** Write a function to sum up a list of numbers
- Recursive approach?
Recursive approach to **sum_list**

- **Base case:**
  - `num_lst` is empty, return 0

- **Recursive rule:**
  - Return first element of `num_lst` plus result from calling `sum_list` on rest of the elements of the list.

- Example: Suppose `num_lst = [6, 3, 6, 5]`
  - `sum_list([6, 3, 6, 5]) = 6 + sum_list([3, 6, 5])`
  - `sum_list([3, 6, 5]) = 3 + sum_list([6, 5])`
  - `sum_list([6, 5]) = 6 + sum_list([5])`
  - `sum_list([5]) = 5 + sum_list([])`
  - For the base case we have `sum_list([])` returns 0
Recursive approach to sum_list

• **Base case:**
  • `num_lst` is empty, return 0

• **Recursive rule:**
  • Return first element of `num_lst` plus result from calling `sum_list` on rest of the elements of the list.

• Example: Suppose `num_lst = [6, 3, 6, 5]`
  • `sum_list([6, 3, 6, 5]) = 6 + sum_list([3, 6, 5])`
  • `sum_list([3, 6, 5]) = 3 + sum_list([6, 5])`
  • `sum_list([6, 5]) = 6 + sum_list([5])`
  • `sum_list([5]) = 5 + sum_list([])`
  • For the base case we have `sum_list([])` returns 0
Recursive approach to \texttt{sum\_list}

def \texttt{sum\_list}(\texttt{num\_lst}):
    """Returns sum of given list"""
    \textbf{if} \texttt{num\_lst} \textbf{==} []:
        \textbf{return} 0
    \textbf{else}:
        \textbf{return} \texttt{num\_lst}[0] + \texttt{sum\_list}(\texttt{num\_lst}[1:])

\texttt{>>> sum\_list([3, 4, 20, 12, 2, 20])}
61
Compare **sum_list** approaches

- Compare/Contrast:

```python
def sum_list_iterative(num_lst):
    sum = 0
    for num in num_lst:
        sum += num
    return sum

def sum_list(num_lst):
    if num_lst == []:
        return 0
    else:
        return num_lst[0] + sumList(num_lst[1:])
```
Graphical Recursion
The Turtle Module

- Turtle is a **graphics module** first introduced in the 1960s by computer scientists Seymour Papert, Wally Feurzig, and Cynthia Solomon.
- It uses a programmable cursor — fondly referred to as the “turtle” — to draw on a Cartesian plane (x and y axis.)
**Turtle In Python**

- **turtle** is available as a built-in module in Python. See the [Python turtle module API](#) for details.
- Basic turtle commands:

  Use `from turtle import *` to use these commands

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>fd(dist)</code></td>
<td>turtle moves forward by <code>dist</code></td>
</tr>
<tr>
<td><code>bk(dist)</code></td>
<td>turtle moves backward by <code>dist</code></td>
</tr>
<tr>
<td><code>lt(angle)</code></td>
<td>turtle turns left <code>angle</code> degrees</td>
</tr>
<tr>
<td><code>rt(angle)</code></td>
<td>turtle turns right <code>angle</code> degrees</td>
</tr>
<tr>
<td><code>up()</code></td>
<td>(pen up) turtle raises pen in belly</td>
</tr>
<tr>
<td><code>down()</code></td>
<td>(pen down) turtle lowers pen from belly</td>
</tr>
<tr>
<td><code>shape(shp)</code></td>
<td>sets the turtle's shape to <code>shp</code></td>
</tr>
<tr>
<td><code>speed(spd)</code></td>
<td>sets the turtle's speed 1-10 (slow-fast). 0 skips animation.</td>
</tr>
<tr>
<td><code>home()</code></td>
<td>turtle returns to (0,0) (center of screen)</td>
</tr>
<tr>
<td><code>clear()</code></td>
<td>delete turtle drawings; no change to turtle's state</td>
</tr>
<tr>
<td><code>reset()</code></td>
<td>delete turtle drawings; reset turtle's state</td>
</tr>
<tr>
<td><code>setup(width, height)</code></td>
<td>create a turtle window of given <code>width</code> and <code>height</code></td>
</tr>
</tbody>
</table>
Basic Turtle Movement

• forward(dist) or fd(dist),
  left(angle) or lt(angle),
  right(angle) or rt(angle),
  backward(dist) or bk(dist)

```python
# set up a 400x400 turtle window
setup(400, 400)
reset()

fd(100) # move the turtle forward 100 pixels
lt(90) # turn the turtle 90 degrees to the left
fd(100) # move forward another 100 pixels

# complete a square
lt(90)
fd(100)
lt(90)
lt(90)
fd(100)
done()
```
Drawing Basic Shapes With Turtle

• We can write functions that use turtle commands to draw shapes.
• For example, here's a function that draws a square of the desired size

```python
def draw_square(length):
    # a loop that runs 4 times
    # and draws each side of the square
    for i in range(4):
        fd(length)
        lt(90)
    done()

setup(400, 400)
reset()
draw_square(150)
```
Drawing Basic Shapes With Turtle

- How about drawing polygons?

```python
def draw_polygon(length, num_sides):
    for i in range(num_sides):
        fd(length)
        lt(360/num_sides)
    done()
```

draw_polygon(80, 3)  
draw_polygon(80, 10)
Adding Color!

- What if we wanted to add some color to our shapes?

```python
def draw_polygon_color(length, num_sides, color):
    # set the color we want to fill the shape with
    # color is a string
    fillcolor(color)

    begin_fill()
    for i in range(num_sides):
        fd(length)
        lt(360/num_sides)
    end_fill()
    done()
```

`draw_polygon_color(80, 10, "gold")` `draw_polygon_color(80, 10, "purple")`
Next Time: Recursive Figures With Turtle

- Next time we will explore how to draw recursive pictures with Turtle.