CS61A Lecture 6

Recursion

Tom Magrino and Jon Kotker
UC Berkeley EECS
June 26, 2012
How Many Computers to Identify a Cat? 16,000

MOUNTAIN VIEW, Calif. — Inside Google’s secretive X laboratory, known for inventing self-driving cars and augmented reality glasses, a small group of researchers began working several years ago on a simulation of the human brain.
TODAY

• Quick review of Iterative Improvement.
• Defining functions that call themselves.
  — Sometimes more than once.
RECAP: NEWTON’S METHOD

```python
def iter_improve(update, isclose, guess=1):
    while not isclose(guess):
        guess = update(guess)
    return guess

def find_root(f, initial_guess=10):
    return iter_improve(newton_update(f),
                        lambda x: approx_zero(f(x)),
                        initial_guess)
```

RECAP: NEWTON’S METHOD

Incredibly powerful, but does not always work! Certain conditions need to be satisfied: for example, the function needs to be differentiable.

The method can fail in many ways, including:
1. Infinite loop among a set of guesses. (Try $f(x) = x^3 - 2x + 2$.)
2. Guesses may never fall within the tolerance for approximate equality.
3. Guesses converge to the answer very slowly.
**RECAP: NEWTON’S METHOD**

*Iterative Improvement (with one fix)*

We can add a limit on the number of *iterations*.

```python
def iter_improve(update, isclose, guess=1, max_iter=5000):
    counter = 1
    while not isclose(guess) and counter <= max_iter:
        guess = update(guess)
        counter += 1
    return guess
```
**Practice: Newton’s Method**

Using `find_root`, write a function `intersection(f, g)` which takes two functions, `f` and `g`, and finds a point at which the two are equal.

```python
def intersection(f, g):
    return find_root(lambda x: f(x) - g(x))
```
PRACTICE: NEWTON’S METHOD

Using `find_root`, write a function `intersection(f, g)` which takes two functions, `f` and `g`, and finds a point at which the two are equal.

```python
def intersection(f, g):
    return find_root(lambda x: f(x) - g(x))
```
Computing Factorial

The factorial of a positive integer $n$ is:

$$n! = \begin{cases} 1, & n = 0 \text{ or } n = 1 \\ n \times (n - 1) \times \ldots \times 1, & n > 1 \end{cases}$$
The factorial of a positive integer $n$ was:

$$n! = \begin{cases} 1, & n = 0 \text{ or } n = 1 \\ n \times \frac{(n - 1) \times \ldots \times 1}{(n-1)!}, & n > 1 \end{cases}$$
Computing Factorial

The factorial of a positive integer \( n \) was:

\[
n! = \begin{cases} 
1, & n = 0 \text{ or } n = 1 \\
n \times (n - 1)!, & n > 1 
\end{cases}
\]

We generalized our definition. Can we do that with our code?
Computing Factorial

\[ n! = \begin{cases} 1, & n = 0 \text{ or } n = 1 \\ n \times (n - 1)!, & n > 1 \end{cases} \]

def fact(n):
    if n == 1 or n == 0:
        return 1
    total = 1
    while n >= 1:
        total, n = total * n, n - 1
    return total

Can we generalize here like we did with our mathematical expression?
Computing Factorial

\[ n! = \begin{cases} 
1, & n = 0 \text{ or } n = 1 \\
n \times (n - 1)!, & n > 1 
\end{cases} \]

def fact(n):
    if n == 1 or n == 0:
        return 1
    return n * fact(n - 1)

How can fact be defined by calling fact?!?!?!
def fact(n):
    if n == 1 or n == 0:
        return 1
    return n * fact(n - 1)
RECURSIVE FUNCTIONS

A function is a *recursive function* if the body calls the function itself, either directly or indirectly.

Recursive functions typically have 2 main pieces:
1. **Recursive case(s)**, where the function calls itself.
2. **Base case(s)**, where the function does NOT recursively call itself and instead returns a direct answer. This is what ensures that the recursion will eventually stop.

```python
def fact(n):
    if n == 1 or n == 0:
        return 1
    return n * fact(n - 1)
```
RECURSION IN EVERY DAY LIFE: EATING CHOCOLATE

You have a bar of chocolate with n small pieces. How do you eat it?

1. You eat 1 piece of chocolate.
2. You eat a bar of n – 1 pieces of chocolate.

What’s your base case?
   – You have no more chocolate.
How would I rewrite the summation function from last week to use recursion?

def summation(n, term):
    if n == 0:
        return 0
    return term(n) + summation(n - 1, term)
PRACTICE: RECURSION

How would I rewrite the summation function from last week to use recursion?

```python
def summation(n, term):
    if n == 0:
        return 0
    return term(n) + summation(n - 1, term)
```
**PRACTICE: RECURSION**

What does the following function calculate?

```python
def fun(a, b):
    if b == 0:
        return 0
    elif b % 2 == 0:
        return fun(a + a, b / 2)
    return fun(a, b - 1) + a
```

It's multiplying `a` times `b`!
**PRACTICE: RECURSION**

What does the following function calculate?

```python
def fun(a, b):
    if b == 0:
        return 0
    elif b % 2 == 0:
        return fun(a + a, b / 2)
    return fun(a, b - 1) + a
```

\[ a \times b \]
Using recursion, write the function \( \log(b, x) \) which finds \( \log_b(x) \), assuming \( x \) is some power of \( b \).

```python
def log(b, x):
    if x == 1:
        return 0
    return 1 + log(b, x / b)
```
PRACTICE: RECURSION

Using recursion, write the function \( \log(b, x) \) which finds \( \log_b(x) \), assuming \( x \) is some power of \( b \).

```python
def log(b, x):
    if x == 1:
        return 0
    return 1 + log(b, x / b)
```
ANNOUNCEMENTS

• Bug-Submit is now available!
• Project 1 is due Friday
• Homework 3 is due Friday
TREE RECURSION

You can have a function defined in terms of itself using more than one recursive call. This is called \textit{tree recursion}.

\[
F_n = \begin{cases} 
0, & n = 0 \\
1, & n = 1 \\
F_{n-1} + F_{n-2}, & n > 1
\end{cases}
\]

\begin{verbatim}
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    return fib(n - 1) + fib(n - 2)
\end{verbatim}
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    return fib(n - 1) + fib(n - 2)

So why is it called *tree recursion*?

```
3
2
1
1
1
0
```
Suppose I want to count all the different routes I could take from (0, 0) to (x, y) on a grid moving only up and right. Write the function paths(x, y) to calculate the number of routes to (x, y).

def paths(x, y):

Practice: Tree Recursion

Suppose I want to count all the different routes I could take from (0, 0) to (x, y) on a grid moving only up and right. Write the function `paths(x, y)` to calculate the number of routes to (x, y).

```python
def paths(x, y):
    if x == 0 or y == 0:
        return 1
    return paths(x - 1, y) + paths(x, y - 1)
```
PRACTICE: TREE RECURSION

Suppose I want to **print** all the different routes I could take from (0, 0) to (x, y) on a grid moving only up and right. Write the function directions(x, y) which prints the each different set of directions using a combination of “UP” and “RIGHT” that one could take. *Hint:* use a helper function that does the recursion and keeps track of the “directions so far.”

def directions(x, y):

**PRACTICE: TREE RECURSION**

Suppose I want to **print** all the different routes I could take from (0, 0) to (x, y) on a grid moving only up and right. Write the function `directions(x, y)` which prints the each different set of directions using a combination of “UP” and “RIGHT” that one could take. *Hint*: use a helper function that does the recursion and keeps track of the “directions so far.”

```python
def directions(x, y):
    def dir_helper(x, y, so_far):
        if x == 0 and y == 0:
            print(so_far)
        elif y > 0:
            dir_helper(x, y - 1, so_far + " UP")
        elif x > 0:
            dir_helper(x - 1, y, so_far + " RIGHT")
        dir_helper(x, y, "")
```

CONCLUSION

• Recursion is a way for functions to be defined using themselves.

• Recursive functions have two parts:
  – Recursive case(s), where the function calls itself.
  – Base case(s), where the function does not call itself (stopping the recursion).

• Tree recursion is used by functions that make more than one recursive call (at the same time) in the definition.