NEWTON'S METHOD, AND RECURSION 2a

COMPUTER SCIENCE 61A

July 2, 2013

1 Newton's Method

Newton's method is an algorithm that is widely used to compute the zeros of functions. It can be used to approximate a root of any continuous, differentiable function.

Intuitively, Newton's method works based on two observations:

- At a point $P = (x, f(x))$, a root of the function f is in the same direction relative to P as the root of the linear function L that not only passes through P, but also has the same slope as f at that point.
- Over any *very small* region, we can approximate f as a linear function. This is one of the fundamental principles of calculus.

Starting at an initial guess $(x_0, f(x_o))$, we estimate the function f as a linear function L, solve for the zero $(x', 0)$ of L, and then use the point $(x', f(x'))$ as the new guess for the root of f. We repeat this process until we have determined that $(x', f('x))$ is a zero of f.

Mathematically, we can derive the update equation by using two different ways to write the slope of L:

Let x be our current guess for the root, and x^* be the point we want to update our guess to. Let L be the linear function tangent to f at $(x, f(x))$.

Remember that x^* is the root of L. So, we know two points L passes through, namely $(x, f(x))$ and $(x^*, 0)$.

We can write the slope of L as

$$L'(x) = \frac{0 - f(x)}{x^* - x} = \frac{-f(x)}{x^* - x} \tag{1}$$

We also know that L is tangent to f as x, so:

$$L'(x) = f'(x) \tag{2}$$

We can equate these to get our update equation:

$$\frac{-f(x)}{x^* - x} = f'(x) \Rightarrow x^* = x - \frac{f(x)}{f'(x)} \tag{3}$$

We know $f(x)$, and from calculus, for some very small ε:

$$f'(x) = \frac{f(x+\varepsilon) - f(x)}{(x+\varepsilon) - x} = \frac{f(x+\varepsilon) - f(x)}{\varepsilon} \tag{4}$$

From the above, we get this algorithm:

```
def approx_deriv(fn, x, dx=0.00001):
    return (fn(x+dx)-fn(x))/dx

def newtons_method(fn, guess=1, max_iterations=100):
    ALLOWED_ERROR_MARGIN = 0.0000001
    i = 1
    while abs(fn(guess)) > ALLOWED_ERROR_MARGIN and i <= max_iterations:
        guess = guess - fn(guess) / approx_deriv(fn, guess)
        i += 1
    return guess
```

We can generalize this idea into a framework known as *iterative improvement*. Basically, you start out by guessing a value, and then continuously update the guess until it is a reasonable approximation of the value we are looking for. Here is an implementation for `iter_improve`. The `update` function takes the current guess, and returns an updated guess. The `isdone` function also takes the current guess, and returns `True` if and only if the current guess is "good enough", according to some set criterion.

```
def iter_improve(update, isdone, guess=1, max_iterations=100):
    i = 1
    while not isdone(guess) and i <= max_iterations:
        guess = update(guess)
        i += 1
    return guess

def newtons_method2(fn, guess=1, max_iterations=100):
    def newtons_update(guess):
        return guess - fn(guess) / derivative(fn, guess)
    def newtons_isdone(guess):
        ALLOWED_ERROR_MARGIN= 0.0000001
        return abs(fn(guess)) <= ALLOWED_ERROR_MARGIN
    return iter_improve(newtons_update,
                        newtons_isdone,
                        max_iterations)
```

1. Write a function `cube_root` that computes the cube root of the input number `x`. (*Hint*: Use `newtons_method` with a function that is zero at the cube root of the input.)

   ```
   def cube_root(x):
   ```

2. Newton's method converges very slowly (or not at all) if the algorithm happens to land on a point where the derivative is very small. Modify the implementation that uses `iter_improve` to return `None` if the derivative is under some threshold, say 0.001.

   ```
   def newtons_method2(fn, guess=1, max_iterations=100):
       def newtons_update(guess, min_size=0.001):




       def newtons_done(guess):




       return iter_improve(newtons_update, newtons_done, guess,
           max_iterations)
   ```

2 Recursion

A function is *recursive* if it calls itself. Below is recursive `factorial` function.

```
def factorial(n):
    if n == 0 or n == 1:
        return 1
    else:
        return n * factorial(n-1)
```

It seems like this won't work – we haven't finished defining `factorial`, yet we're already calling it. However, we do have one *base case*: when `n` is `0` or `1`. Now we can compute `factorial(1)` in terms of `factorial(0)`, and `factorial(2)` in terms of `factorial(1)`, and `factorial(3)` – well, you get the idea.

There are *three* common steps in a recursive definition:

1. *Figure out your base case*: Ask yourself, "what is the simplest argument I could possibly get?" The answer should be simple, and is often given by definition. For example, `factorial(0)` is 1, by definition, or the first two Fibonacci numbers are 0 and 1.
2. *Make a recursive call with a simpler argument*: Simplify your problem, and assume that a recursive call for this new problem will simply work. This is called the "leap of faith" – as you use more recursion, you will get more used to this idea. For `factorial`, we make the recursive call `factorial(n-1)` – this is the recursive breakdown.
3. *Use your recursive call to solve the full problem*: Remember that we are assuming your recursive call just works. With the result of the recursive call, how can you solve the original problem you were asked? For `factorial`, we just multiply $(n-1)!$ by n.

2.1 Cool Questions!

1. Print out a countdown using recursion.

```
def countdown(n):
    """
    >>> countdown(3)
    3
    2
    1
    """
```

2. Is there an easy way to change `countdown` to count up instead?

3. Write a procedure `expt(base, power)`, which implements the exponent function. For example, `expt(3, 2)` returns 9, and `expt(2, 3)` returns 8. Use recursion.

```
def expt(base, power):
```

4. Write `sum_primes_up_to(n)`, which sums up every prime up to and including `n`. Assume you have an `isprime()` predicate.

```
def sum_primes_up_to(n):
```

5. Now write `sum_filter_up_to(n, pred)`, which is a general version that adds all integers `1` through `n` that satisfy the argument `pred`.

   ```
   def sum_filter_up_to(n, pred):
   ```

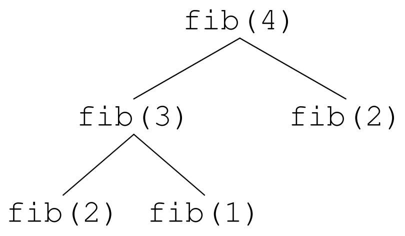
3 Tree Recursion

Consider a function that requires more than one recursive call. A simple example is a function that computes Fibonacci numbers:

```
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n - 1) + fib(n - 2)
```

This is where recursion really begins to shine: it allows us to explore two different calculations at the same time. In this case, we are exploring two different possibilities (or paths): the $n - 1$ case and the $n - 2$ case. With the power of recursion, exploring all possibilities like this is very straightforward. You simply try everything using recursive calls for each case, then combine the answers you get back.

This type of recursion is called *tree recursion*, because the different branches of computation that form from this recursion end up looking like an upside-down tree:



We could, in theory, use loops to write the same procedure. However, problems that are naturally solved using tree recursive procedures are generally difficult to write iteratively, and require the use of additional data structures to hold information. As a general rule of thumb, whenever you need to try multiple possibilities at the same time, you should consider using tree recursion.

3.1 Exercises

1. I want to go up a flight of stairs that has n steps. I can either take 1 or 2 steps each time. How many different ways can I go up this flight of stairs? Write a function `count_stair_ways` that solves this problem for me.

   ```
   def count_stair_ways(n):
   ```

2. Pascal's triangle is a useful recursive definition that tells us the coefficients in the expansion of the polynomial $(x+a)^n$. Each element in the triangle has a coordinate, given by the row it is on and its position in the row (which you could call its column). Every number in Pascal's triangle is defined as the sum of the item above it and the item that is directly to the upper left of it. If there is a position that does not have an entry, we treat it as if we had a 0 there. Below are the first few rows of the triangle:

   ```
   Item:     0    1    2    3    4    5
   Row 0:    1
   Row 1:    1    1
   Row 2:    1    2    1
   Row 3:    1    3    3    1
   Row 4:    1    4    6    4    1
   Row 5:    1    5    10   10   5    1
   ...
   ```

   Define the procedure `pascal(row, column)` which takes a row and a column, and finds the value at that position in the triangle. Don't use the closed-form solution, if you know it.

   ```
   def pascal(row, column):
   ```

3. The TAs want to print handouts for their students. However, for some unfathomable reason, both the printers are broken; the first printer only prints multiples of `n1`, and the second printer only prints multiples of `n2`. Help the TAs figure out whether or not it is possible to print an exact number of handouts!

```
def hasSum(sum, n1, n2):
    """
    >>> hasSum(1, 3, 5)
    False
    >>> hasSum(5, 3, 5) # 1(5) + 0(3) = 5
    True
    >>> hasSum(11, 3, 5) # 2(3) + 1(5) = 11
    True
    """
```

4 Mutual Recursion

Sometimes it isn't enough to have one function call itself; sometimes functions recursively call one another! Here is an example:

```
def even(n):
    if n == 0:
        return True
    return odd(n - 1)

def odd(n):
    if n == 0:
        return False
    return even(n - 1)
```

Given a positive integer, the function `even` will return a boolean value representing whether or not the integer is even. However, notice that the recursive call is to `odd`, not itself. We call this mutual recursion because because `even` and `odd` are defined in terms of one another, and as a result alternatively call one another to arrive at the answer.

Mutual recursion is especially useful for when you need to deal with different data structures that interact with one another. We'll see this later in the semester with Trees.

4.1 Exercises

1. Let's answer the age-old question of "What came first, the chicken or the egg?" Each function takes an argument (pun unintended!), which is a boolean value that represents an argument for whether or not the corresponding element came first. To mirror the debate, let's do this:

```
def chicken(argument=True):
    print("Chickens!")
    return egg(argument)
def egg(argument=True):
    print("Eggs!")
    return chicken(argument)
```

We could argue that chickens came first by calling `chicken()`, but then the argument would never be resolved. That's unfortunate! Therefore, let's also add `generations`, which is a variable that represents which generation we are on. For example, if you argue for the 5th generation chicken, you're arguing for not the 4th generation egg. The 0th generation determines the winner.

```
def chicken(argument, generations=0):
    """ Argue whether or not the chicken came first.
    >>> chicken(True)
    [Insert reason chickens came first.]
    >>> chicken(False) # is equivalent to egg(True)
    [Insert reason eggs came first.]
    >>> chicken(False, 1) # is equivalent to egg(True, 0)
    [Insert reason eggs came first.]
    """
```

```
def egg(argument, generations=0):
    "Argue whether or not the egg came first."""
```

5 Iteration vs. Recursion

We've written `factorial` recursively. Let's compare the iterative and recursive versions:

```
def factorial_recursive(n):
    if n <= 0:
        return 1
    else:
        return n * factorial_recursive(n-1)

def factorial_iterative(n):
    total = 1
    while n > 0:
        total = total * n
        n = n - 1
    return total
```

Notice that the recursive test corresponds to the iterative test. While the recursive function "works" until n is less than or equal to 0, the iterative "works" while n is greater than 0. They are essentially the same.

Let's also compare `fibonacci`.

```
def fib_r(n):
    if n == 1:
        return 0
    elif n == 2:
        return 1
    else:
        return fib_r(n - 1) + fib_r(n - 2)

def fib_i(n):
    curr, next = 0, 1
    while n > 1:
        curr, next = next, curr + next
        n = n - 1
    return curr
```

For the recursive version, we copied the definition of the Fibonacci sequence straight into code! The nth fibonacci number is literally the sum of the two before it. Iteratively, you need to keep track of more numbers and have a better understanding of the code.

Sometimes code is easier to write iteratively, sometimes code is easier to write recursively. Have fun experimenting with both!

6 Extra Practice Problems

1. Recall the hailstone function from homework 1. You pick a positive integer n as the start. If n is even, divide it by 2. If n is odd, multiply it by 3 and add 1. Repeat this process until n is 1. Write a recursive version of hailstone that prints out the values of the sequence and returns the number of steps.

```
def hailstone(n):
```

2. In homework 2 you encountered the repeated function, which takes arguments f and n and returns a function equivalent to the nth repeated application of f. This time, we want to write repeated recursively. You'll want to use compose1, given below for your convenience:

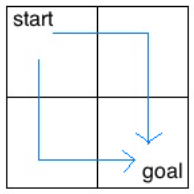
```
def compose1(f, g):
    """Return a function h, such that h(x) = f(g(x))."""
    def h(x):
        return f(g(x))
    return h
```

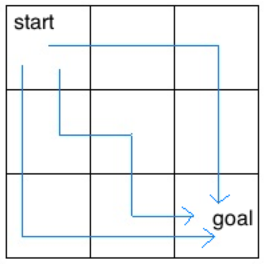
3. The greatest common divisor of two positive integers a and b is the largest integer which evenly divides both numbers (with no remainder). Euclid, a Greek mathematician in 300 BC, realized that the greatest common divisor of a and b is the smaller value if it evenly divides the larger value or the same as the greatest common divisor of the smaller value and the remainder of the larger value divided by the smaller value. So if a is greater than b and a is not divisible by b then:

```
gcd(a, b) ==    gcd(b, a % b)
```

   Write the gcd function using Euclid's algorithm. Try coming up with a recursive and an iterative solution.

4. Consider an insect in a MxN grid. The insect starts at the top left corner, (0,0), and wants to end up at the bottom right corner, (M-1,N-1). The insect is only capable of moving right or down. Write a function `count_paths` that takes a grid length and width and returns the number of different paths the insect can take from the start to the goal. [There is an analytic solution to this problem, but try to answer it procedurally using recursion].





   For example, the 2x2 grid has a total of two ways for the insect to move from the start to the goal. For the 3x3 grid, the insect has 6 different paths (only 3 are shown above).

```
def paths(x, y):
```