ITERATION AND RECURSION 3

COMPUTER SCIENCE 61A

June 26, 2012

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 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:

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

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)
```

1.1 Questions

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):
```

Solution:

```
    def fn(y):
        return y*y*y - x
    return newtons_method(fn)
```

2. Newtons 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):
```

Solution:

```
        dtv = derivative(fn, guess)
        if abs(dtv) < min_size:
            return None
        return guess - fn(guess) / derivative(fn, guess)
```

```
    def newtons_done(guess):
```

Solution:

```
        ALLOWED_ERROR_MARGIN= 0.0000001
```

```
        if guess == None:
            return True
        y = fn(guess)
        return abs(y) <= ALLOWED_ERROR_MARGIN
```

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

3. In Newton's method, the function `iter_improve` may cycle between two different guesses. For instance, consider finding the zero of $f(x) = x^3 - 2x + 2$ using exact derivatives. A starting guess of 1 updates to 0, which updates to 1 again. The cyclic pattern 1, 0, 1 is called a *length-two* cycle, because it contains only two distinct elements before repeating.

   Write a new version of `iter_improve` that returns as soon as a length-two cycle of guesses occurs, but otherwise behaves the same as the implementation from class.

   ```
   def iter_improve(update, done, guess=1, max_updates=1000):
   ```

Solution:

```
k = 0
previous_guess = None
while not done(guess) and k < max_updates:
    next_guess = update(guess)
    if previous_guess == next_guess:
        return False
    previous_guess, guess = guess, next_guess
    k += 1
return guess
```

2 Recursion

We say a procedure is *recursive* if it calls itself in its body. Below is an example of a recursive procedure to find the factorial of a positive integer n:

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

```
    return n * factorial(n-1)
```

Upon first glance, it seem like this might not work, since we haven't finished defining factorial, but we're already calling it. However, note that we do have one case that is simple: when `n` is either `0` or `1`. This case, called our *base case*, gives us a starting point for our definition. Now we can compute factorial of 1 in terms of factorial of 0, and the factorial of 2 in terms of the factorial of 1, and the factorial of 3, ... well, you get the idea.

Three common steps in a recursive definition:

1. *Figure out your base case*: Ask yourself, "what is the simplest argument someone could possibly give me? The answer should be extremely simple, and is often given simply by definition. For example, the factorial of 0 is 1, by definition, or the first two Fibonacci numbers are 0 and 1.
2. *Make a recursive call with a slightly simpler argument*: Simplify your problem somehow, and assume that a recursive call for this new problem will simply just work. This is sometimes referred to as a "leap of faith" – as you do more recursion problems, you will get more used to this idea. For the defintion of `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. Write a countdown using recursion.

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

Solution:

```
    if n <= 0:
        return
    print(n)
    countdown(n - 1)
```

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

Solution: Move the `print` statement to after the recursive call.

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):
```

Solution:

```
    if power == 0:
        return 1
    elif power < 0:
        return expt(base, power + 1) / base
    else:
        return expt(base, power - 1) * base
```

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):
```

Solution:

```
    if (n <= 1):
        return 0
    elif (isprime(n)):
        return n + sum_primes_up_to(n - 1)
    else:
        return sum_primes_up_to(n - 1)

    OR

    if (n <= 1):
        return 0
    else:
        num_if_prime = n if isprime(n) else 0
        return sum_primes_up_to(n - 1) + num_if_prime
```

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):
```

Solution:

```
    if n <= 0:
        return 0
    elif pred(n):
        return n + sum_filter_up_to(n - 1, pred)
    else:
        return sum_filter_up_to(n - 1, pred)
```

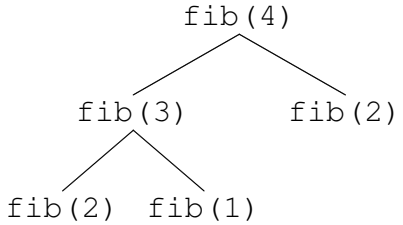
3 Tree Recursion

Say we had a procedure that requires more than one possibility to be computed in order to get an answer. A simple example is a function that computes Fibonacci numbers:

```
def fib(n):
    if n == 1:
        return 0
    elif n == 2:
        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.

We often call this type of recursion, where we use more than one recursive call to find an answer *tree recursion*. We call it 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):
   ```

   Solution:

   ```
   if n == 1:
       return 1
   elif n == 2:
       return 2
   return count_stair_ways(n-1) + count_stair_ways(n-2)
   ```

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):
   ```

Solution:

```
if column == 0:
    return 1
elif row == 0:
    return 0
else:
    return pascal(row - 1, column) +
           pascal(row - 1, column - 1)
```

3. Let's say that 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
    """
```

Solution:

```
if sum == n1 or sum == n2:
    return True
if sum < min(n1, n2):
    return False
return hasSum(sum - n1, n1, n2) or
       hasSum(sum - n2, n1, n2)
```

4 Iteration vs. Recursion

We have written `factorial` recursively. Let us 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
```

Notice how, recursively, 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 what the code is doing.

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