Recursion

Clicker Question

What will this print?

```
def blah(x):
    return x+1
def main():
    z = blah(3)
    print(z)
main()
```

A) 3 B) 4 C) 5 D) It causes an error

So how about this?

```
def foo(s):
    if len(s) == 1:
        return 1
    else:
        return foo(s[1:])+1
def main():
    print( foo( "a" ) )
main()
```

A) 0 B) 1 C) 2 D) It causes an error

In fact, if s is any string of length 1, then foo(s) is 1.

Now remember that if s is a string, s[1:] is the portion of s after the first letter. If s == "Bob", then s[1:] is "ob". And if s is "ab" then s[1:] is just "b".

So what will this print?

```
def foo(s):
    if len(s) == 1:
        return 1
    else:
        return foo(s[1:]) + 1
def main():
    print( foo( "ab" ) )
main()
```

A) 0   B) 1   C) 3   D) It causes and error.

It is easy to see that if s is a string of length 2 then s[1:] is a string of length 1, so foo(s[1:]) is 1 and foo(s) is foo(s[1::])+1, which is 2.

In fact, if is is any string then foo(s) is the length of s. That isn't so surprising, since foo() uses the len( ) function for strings.

But consider this function, which is just like foo:

```
def stringLength(s):
    if s == "":
        return 0
    else:
        return stringLength(s[1:])+1
```

This finds the length of any string and it doesn't use the len( ) function.

Functions that call themselves are said to be *recursive*. This is a very important programming technique, which sometimes is the only way to solve a problem. There is an art to writing recursive functions; it requires thinking about programming in somewhat different terms than we have up to now.

With loops we usually think mechanically -- here is what we do to this variable, here is where we compute this, and so forth. With recursion we need to think more holistically -- here is what the function does.

The reason function stringLength( ) works is that it finds the length of a string in terms of the length of a shorter string, which depends on the length of an even shorter string, and so forth, and this eventually leads to applying stringLength() to a string that is so short that we know how to find its length without doing any work.

The key insight into recursion is that all recursive function must have arguments for which they can return without recursing and the recursive call must move the arguments closer to the non-recursive cases.

It isn't often that you get to use the same word 5 times in one sentence. ....

Recursive functions almost always start with a test to see if the argument is one of the non-recursive cases.

Here is another example that also works with strings. Let's find a recursive function that reverses a string. As with stringLength( ), we are going to recurse on shorter strings. The easy ways to shorten string s are s[1:] (all of s except its first letter) and s[0:-1] (all of s except its last letter).

Let's use s[1:]. We will compute the reversal of s in terms of the reversal of s[1:] and the first letter of s, s[0].

Remember that we need to start with a test for the non-recursive case. We are going to keep pulling letters off of s until we get to strings so simple that they are easy to reverse. Certainly the empty string is easy to reverse; since there is nothing there, it is its own reversal. For that matter strings of length 1 are their own reversals. This means our reverse( ) function can start

```
def reverse(s):
    if len(s) <= 1:
        return s
    else:
        .....
```

For the recursive case we need to say how to put together reverse(s[1:]) and s[0]. Suppose s is "Oberlin". Then s[1:] is "berlin" and reverse(s[1:]) is "nilreb". How do we put this together with s[0] to get "nilrebO"??

A) s[0] + reverse(s[1:])
B) reverse(s[1:])+s[0]
C) reverse(s[0]+s[1:])
D) reverse(s[1:]+s[0])

Altogether, here is our reverse( ) function

```
def reverse(s):
    if len(s) <= 1:
        return s
    else:
        return reverse(s[1:]) + s[0]
```

Here's another example. I want to write a function to compute factorials. Of course factorial(5) is the product 1x2x3x4x5. To do this recursively we need some way to relate factorials to other factorials. Here are two formulas

1. factorial(n) = n*factorial(n-1)
   For example, factorial(5) = 5x(4x3x2x1)
   = 5*factorial(4)
2. factorial(n)= factorial(n+1)/(n+1)
   For example, factorial(5)=(6x5x4x3x2x1)/6
   = factorial(6)/6

We need non-recursive cases -- numbers for which we know the factorial without doing any work. Mathematicians usually define the factorials of both 0 and 1 to be 1. So the non-recursive cases are small arguments; we need to move larger arguments towards these. This means our first formula for factorials, finding factorial(n) in terms of factorial(n-1), makes sense.

All of this thinking gives us the following function:

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

If we use the other recursive formula for factorials we get

```
def badFactorial(n):
    if n <= 1:
        return 1
    else:
        return badFactorial(n+1)//(n+1)
```

With this function recursions for arguments greater than 1 never terminate: to compute badFactorial(2) you need to know badFactorial(3) which requires badFactorial(4) and badFactorial(5) and so forth.

Which calculates the value of a to the nth power?

```
def A(a, n):
    if n==0:
        return 1
    else:
        return a*A(a, n-1)
```

```
def C(a, n):
    if n==0:
        return 1
    else:
        return n*C(a, n-1)
```

```
def B(a, n):
    if n==0:
        return 0
    else:
        return n*B(a, n-1)
```

```
def D(a, n):
    if a==0:
        return 1
    else:
        return a*D(a, n-1)
```

Let's write isIn(x, s), which returns True if x is in the string s.

Our recursive idea is that x is in s if either x is s[0] or x is in s[1:]. This is recursing on smaller strings. For our "base" (or non-recursive) case, we want a string so small that we know the answer without looking. This happens when s is the empty string; s=="".

Altogether this gives us

```
def isIn(x, s):
    if s=="":
        return False
    else:
        return (x==s[0]) or isIn(x, s[1:])
```

Clicker Question

What will this print?

```
def f( s ):
    if  s == "":
        return 0
    else:
        return f(s[1:])
print(  f( "Marvin Krislov" )  )
```

A) 0
B) 6
C) 14
D) It will run forever

And Another

The Fibonacci recursion is pretty obvious:

Fib(n) = Fib(n-1) + Fib(n-2)

But what is the base case?

A) n == 0
B) n == 1
C) n == 1 or n == 0
D) This can't be written recursively

The Towers of Hanoi puzzle is an old game based on the mythical story that somewhere in Hanoi there is a temple with 3 giant towers containing 64 golden disks. Each disk is a different size. Monks in the temple are busy moving disks from one tower to another. The rules are simple:

- Only 1 disk can be moved at a time
- No disk can be put on top of a smaller disk.

The monks started with all 64 disks on one tower. When they have finished moving all 64 disks to the next tower, the world will end.

The programming version of this is: Find the sequence of moves that will move n disks from one tower to the next.