1 Recursively defined functions

Recall that $n!$ is defined to be the product of all numbers between 1 and n, inclusive. That is, $n! = n \cdot (n-1) \cdot (n-2) \cdot ... \cdot 3 \cdot 2 \cdot 1$. This gives a sequence of numbers $1!, 2!, 3!, 4!, ... = 1, 2, 6, 24, ...$

We can also define the factorial function *recursively*. What it means to define a sequence recursively is that you give the first number in the sequence and rule for how to get to the next number in the sequence from any known number. In this spirit, we can define the sequence of factorials by the following two pieces of information:

- $1! = 1$
- $n! = n \cdot (n-1)!$

From these two pieces of information we can figure out what the factorial of any number is, working up from 1. For example, this information does not tell me directly what 4! is, but it does tell me that it is 4 times 3!. We don't know what 3! is either, but we do know that it is 3 times 2!. We don't know what 2! is, but we do know that it is 2 times 1!. Since we know that 1! is 1, it follows that 2! is 2, that 3! is 6 and finally that 4! is 24.

Let's see how this idea transfers to code.

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

Notice that his contains exactly the two pieces of information that define factorial: input 1 gives output 1, and any input greater than 1 gives itself times the factorial of one number one lower.

We could also define multiplication recursively via the following information:

- Any number x multiplied by 1 is itself ; $x \cdot 1 = x$
- Any number x multiplied by any other number n is x more than x multiplied by the number one lower ; $x \cdot n = x + x \cdot (n-1)$

This thought process is realized as code as follows:

```
def multiply(x,n):
    if n==1:
        return x
    else:
        return x + multiply(x,n-1)
```

Similarly, we could define a function that tells how many times one number b goes into a number a without going over. The thought process is, the number of times that b goes into a is equal to the number of times that we can subtract b from a without getting a negative. So we count:

```
def quotient(a,b):
    if a<b:
        return 0
    else:
        return 1 + quotient(a-b,b)
```

Let's next see a more involved example of how this can be used. Suppose that we want to find all of the prime factors of a number, including how many times each prime factor occurrs. We could think of doing this via the following process:

- If the given number is 1 then we don't do anything.
- If the given number is greater than 1, then we start at 2 and try dividing the given number by every number.
- Once we find a divisor we print it, divide the given number by it, and repeat the process on the smaller quotient.

This line of thinking translates into the following code,

```
def printprimes(n):
    if n == 1:
        print 'That is all'
    else:
        possiblefactor = 2
        while 1==1:
            if n % possiblefactor == 0:
                print possiblefactor
                return printprimes(n/possiblefactor)
            else:
                possiblefactor = possiblefactor + 1
```

Note that this code is not very efficient; it does not find prime factors well but merely exists as an example of a recursive definition.

1. Write a function `rGCD(a,b)` that computes the greates common divisor of two numbers a and b. The fastest way to do this is to use the following recursive scheme:

   - If the smaller number b divides the bigger number a, then the gcd is the smaller number b.
   - If the dividing a by b gives a remainder r, then the gcd of a and b is the same as the gcd of b and r.

   I'll get you started by giving you the code to ensure that a is always the larger of the two numbers. Fill in the $\cdots$ with the above recursive scheme.

   ```
   def rGCD(a,b):
       if a<b:
           a,b = b,a
       ...
   ```

2. (**Challenge**) Write a function `bezout(a,b)` that computes s and t in the equation $sa + tb = d$ where d is the greatest common divisor of a and b. Use this function to write another function `modInv(a,n)` that computes the inverse of a mod n, assuming a and n are coprime.

2 Public Key Encryption

In order to run the public key encrytion programmes we are going to be studying for the rest of camp, we are going to want functions that do the following things:

- Generate large prime numbers.
- Find generators for large prime mods.
- Find the inverse of a given number in a given mod.
- Compute the prime factorization of a number.

Feel free to work first on the one of these that you find most interesting/approachable. In fact, it is probably good to have different people in the same team work on different things.

What follows is some insight into each task to give you some idea of how to get started.

Generate large prime numbers:

The most obvious thing to do is perhaps to define a function `primes(n)` that returns all prime numbers less than n. You might want to start by defining a function `isprime(n)` that returns True or False depending on if n is prime or not and then use this inside of your `primes(n)` function. When checking if a number is prime or not, you only need to check that it is not divisible by anything below its square root; this observation can make your function run way faster.

An alternative (and very good) method of generating primes is the Seive of Eratosthenes. Look this up on Google and try implementing it in Sage; this will work much faster than the above mentioned method.

Find generators for large prime mods:

Recall that a number x is a generator mod p if the powers $x, x^2, x^3, ..., x^{p-1}$ are all different and hit every value mod p, except for zero. To write a function `findgenerator(p)` it may be easiest to write a functon `isgenerator(x,n)` that returns True or False depending on whether x is a generator mod n. How does one check if x is a generator mod p? One way is to make a list, add powers of x to this list until one repeats, and if the list is not $p-1$ entries long when this happens then x is not a generator.

Find the inverse of a given number in a given mod:

We want a function `modInv(a,n)` that returns the number b so that $ab \equiv_n 1$. The most obvious way to do this is to multiply a by everything, starting at 2 and working up, until you get 1. This works but will not be very fast for large numbers. The better way to do this is to write the `bezout(a,b)` function mentioned on the previous page.

Compute the prime factorization of a number:

We want a function `pFactors(n)` that returns (not just prints!) a list of the prime factorization for n, counting repeated factors. For example, `pFactors(24)` should return `[2,2,2,3]`. One way to do this is to first write a function `primes(n)` that generates a list of the prime numbers between 2 and n and then have `pFactors(n)` go through this list, checking if each number divides the given number. Alternatively, you can try to modify the idea in the recursive example given earlier so that it returns its output as a list instead of printing individual numbers.