COMPUTER EXPERIMENTATION #3 – MATH 5405
SPRING 2016

DUE: TUESDAY FEBRUARY 16TH

Our goal this time is to make better functions for some common mathematical operations, like finding primitive roots.

(1) Begin by making a new file. Let's call it `modArith.py` (ie, from the terminal run `gedit modArith.py &`). The start python in the terminal (or you can use `idle`)

(2) First things first, taking big exponents can be *slow*. About how long does computing

```
>>> (239587**992743)%58234
```

take?

(3) Python already has a function for doing this quickly. Instead try

```
>>> pow(239587, 992743, 58234)
```

Now, write a paragraph explaining how python might be doing this computation so quickly. There are a couple tricks it might be using, remember

$$(a \cdot b \cdot c) \bmod n = ((a \cdot b) \bmod n) \cdot c) \bmod n$$

(4) Back in your file `modAirth.py` make a function which does factoring. In particular, you should return a list of prime factors of an integer. I made mine as a loop but a recursive function would work well too.

Warning: python does not like calling `range` over really big ranges. Don't use it for large primes.

```
def factor(n):
    myList = []
    i = 2
    while (i <= n):
        ...
        i = i+1

    return myList
```

After you do this, see if you can make it faster. Ie, figure out how to loop only through is such that $i^2 = 1$.

(5) Make a new function which checks whether a is a primitive root modulo a prime p. Ie, I started mine as

```
def isPrimitiveRoot(a,p):
```

Remember that in class, we talked about ways to make this fast without checking all powers, definitely use that method.

Make sure to test out your function on some primitive roots (and non-primitive roots) you know.

(6) Now make a new function `findPrimitiveRoot(p)`. This should return a primitive root. Here's how I started mine.

```
def findPrimitiveRoot(p):
     i = 2
     while (i < p):
          ...
          return i
          i = i + 1
```

Again, make sure your function is working appropriately before continuing. (To reload your package, instead of importing again, call `reload(modArith)`.

(7) Now let's make a discrete logarithm function. In other words, you want to solve the equation $a^n \equiv b \bmod p$ for n. There is no trick to this, which is why it is cryptographically secure (remember, a should usually be a primitive root).

```
def discreteLog(a,b,p):
    if (isPrimitiveRoot(a,p) == True):
        i = 1
        while (i < p):
            ...
            i = i + 1
```

(8) Do some tests with some big primes (and primitive roots) to find out how fast this can be done in practice. Here is a list of large primes to play with.

- 60077
- 528763
- 9326077
- 148256399
- 2994528541
- 67942202161
- 2851905084269
- 199962421245781

Remember, for each such prime, start by finding a primitive root a. Then choose a random number $b < p$ and tell the computer to solve $a^b \equiv b; \bmod p$.