COMPUTER EXPERIMENTATION #5 – MATH 5405
SPRING 2016

DUE: THURSDAY, MARCH 3RD

We will implement the $p-1$ and $p+1$ factorization schemes from the text. The $p-1$ scheme is easy. The $p+1$ is a bit more work since we don't yet have a good polynomial class in python, but it isn't all that difficult in the end. We'll just have to think things through carefully.

The $p-1$ method

First let's implement the $p-1$ method. I created a file called `Factor.py` where I stored my functions, but you can make your own if you want, or use an old file. It's up to you.

First let's make a function that takes the number n we want to factor, then does the $p-1$ test based on a user-specified a.

```
def pMinusOne(n, a):
    d = fractions.gcd(a,n)
    if (d > 1):
        return d

    flag = False
    #flag is set to True when we find a factor, but not before
    ai = a
    i = 2
    while (flag == False):
        ...
        i = i+1

    return d
```

Try it where n is the product of a couple 5+ digit primes (you can use Rabin-Miller to find these largish primes).

Next, make a new function that does the test on a series of a values. I set up mine like

```
def finalpMinusOne(n):
    a = 2
    d = n
    while ((d == n) and (a < n)):
        ...
        a = a+1
    return d
```

Again, try it on some products of relatively large primes.

This is the one where we had to deal with things of the form $a + b\sqrt{d} = a + bx$ where a and b are taken modulo n and $x = \sqrt{d}$ is something we picked.

Let's fix a d and represent $a + b\sqrt{d}$ as a list of two numbers `[a,b]`. First we'll create a function which takes two such polynomials and does the computation $(a + b\sqrt{d})(c + e\sqrt{d}) = (ac + bed) + (ae + bc)$.

```
def prodElts(ll1, ll2, d, m):
    a = ll1[0]
    b = ll1[1]
    c = ll2[0]
    e = ll2[1]
    return [ (a*c+b*e*d)%m, (a*e+b*c)%m]
```

Next, we need to be able to compute

$$(a + b\sqrt{d})^m = a' + b'\sqrt{d}$$

in a somewhat efficient way (where the a' and b' are represented modulo n).

Our next function will be one that takes in `[a,b], d, m` and outputs a list [a',b'] as above. An obvious way to write this function is something like this.

```
def powOfAPlusBSqrtDSlow(ll, m, d,n):
    a = ll[0]
    b = ll[1]
    i = 1
    curll = ll
    while (i < m):
        curll = prodElts(curll, ll,d,n)
        i = i+1
    return curll
```

Figure out why this works, and then figure out a way to make a much faster function. I made a much faster function that relied on recursion and the idea that $(a + b)^5 = ((a + b)^2)^2(a + b)$.

```
def powOfAPlusBSqrtD(ll, m, d, n):
    #first grab the two values from my list
    a = ll[0]
    b = ll[1]
    #we do this recursively
    #the zeroth power is 1+0\sqrt{d}
    if (m == 0):
        return [1,0]
    elif (m == 1):
        return ll
    else:
        ...
        #here ll1 is now ll^(m%2) and ll2 is (ll^(m//2))^2
        return prodElts(ll1,ll2,d,n)
```

See if your `powOfAPlusBSqrtD` is much faster than `powOfAPlusBSqrtDSlow` for m large (say around 100,000,000).

Ok, now that our preliminaries are out of the way, we can try implementing the $p+1$ method. Remember, instead of choosing a random a, we now choose a random $z = a + b\sqrt{d}$. I'll first create a function that uses a user-specified z.

```
def pPlusOne(n, z, d):
    normZ = prodElts(z, [z[0], (-1)*z[1]], d, n)

    mygcd = fractions.gcd(normZ[0], n)
    if (mygcd > 1):
        return mygcd

    flag = False
    #flag is set to True when we find a factor, but not before
    zi = z
    i = 2
    while (flag == False):
        ...
    return mygcd
```