COMPUTER EXPERIMENTATION #6 – MATH 5405
SPRING 2016

DUE: THURSDAY MARCH 10TH

We will do an implementation of RSA. We will also do an implementation of a quadratic sieve (at least a poor one). Finally, you should put together various tools you've written for factoring and practice factoring some large numbers.

RSA

Shocking as it is, we have never implemented the extended Euclidean algorithm. In particular, we don't already have functions that can compute the inverse of a modulo n. Let's do that first. Here's how I made my function, let's just use it (or you can write your own). You can download it at

`http://www.math.utah.edu/~schwede/math5405/invMod.py`

Now create your own RSA encryption and decryption functions. Your encryption function should take the public key (m, e) and the plaintext x and then output $x^e \bmod m$.

```
def RSAencrypt(x, m, e):
    ...
```

You also need a decryption function. If you'd like, your decryption function should take primes p, q so that $m = pq$ and e. It should also take the ciphertext y.

```
def RSAdecrypt(y, p,q,e):
    #remember, we first need to find d = e^{-1} mod \phi(m)
    ...
```

Make a couple large primes, choose a decent sized e, and then try encrypting and decrypting a message. Make sure your functions give the write input and outputs.

Quadratic Sieve

Next let's make a simple quadratic sieve function. In particular, let's make a really simple function that just checks if $x^2 \bmod n$ is a square for various small values of n.

The first issue we run into is that the built in square root function `math.sqrt` does not work well on numbers this big. This is because $\sqrt{n}$ is represented as a decimal but it only keeps track of a few decimals, far fewer than we will need. Hence we will need a better square root function. There are at least two ways to do this. Use `decimal` to tell it to represent things to a higher precision or just compute the square root yourself.

Indeed, a little googling will lead you to the following code snippet.

```
def isqrt(n):
    x = n
    y = (x + 1) // 2
    while y < x:
        x = y
        y = (x + n // x) // 2
    return x
```

which is just Newton's method. See

`http://stackoverflow.com/questions/15390807/integer-square-root-in-python`

for more discussion.

Here's how I started my poor quadratic sieve function.

```
def poorQuadSieve(n,maxi=-1):
    #this looks for numbers x such that x^2 mod n is a perfect square...
    sqrtN = isqrt(n)
    i = 1
    factoredFlag = False
    while (factoredFlag == False):
        x = sqrtN + i
        ...
        i = i+1
```

For extra credit (up to 30 bonus points on the midterm, Due March 22nd): Make your quadratic sieve work the right way (with the linear dependence as we discussed). Note, I didn't include this as part of the actual assignment because the standard off the shelf python linear algebra packages (for instance scipy) do not have row reduction mod p built into them. However, you can always write this yourself, it is particularly easy mod 2, and if you do write it, you get lots of extra credit. The next page has some information on how you can get started on this.

Make your own public key

Find primes p and q, each less than 20 digits, so that the $p-1$, $p+1$ and quadratic sieve methods you have written cannot factor $n = p \cdot q$ "quickly". Note that n must be less than 40 digits long. Be ready for other groups to try to factor your number next week! Also practice doing RSA encryption and decryption with your number.

Extra credit details

Let's talk about getting started on the extra credit.

First you will need a list of column primes. In other words, primes such that n is a square mod that prime. To do this, you need to have a function that creates a list of primes. There are two good ways to do this.

(1) Write your own Sieve of Eratosthenes function. Roughly speaking, this creates a list of primes. The idea is to start with a small list of primes `[2]` and then check whether 3,5,7,9,11, etc. are prime by seeing if any of the previous primes we found divide them. (Better yet, you can start with a smaller list of primes [2,3] and then check whether numbers of the form $6 \cdot k \pm 1$ are prime).

(2) Use `sympy` (from lab computers, you can just call `import sympy`, for your own computer you might have to install it). Then you can run things like

```
[i for i in sympy.primerange(1,2000)]
```

Now you need to take your list of primes and throw away those for which n is not a square mod p. I first made a legendreSymbol function like this.

```
def legendreSymbol(a,p):
    if (p < 2):
        print 'Expected a prime p > 1'
        return 0
    if (p == 2):
        return a%p
    else:
        k = pow(a,(p-1)/2,p)
        ...
```

Then in my list `quadraticSieve` function I would run something like

```
def quadSieve(n, maxPrime):
    tempPrimeList = sieveEratosthenes(maxPrime)
    #now we throw out all the primes with bad legendre symbol
    factorBase = [] #this is the list of primes we want
    for i in range [0, len(tempPrimeList)]:
        if (legendreSymbol(n,tempPrimeList[i]) == 1):
            factorBase.append(tempPrimeList[i])
```

The next part is tricky. You want to let `sqrtN = isqrt(n)` as before. Then consider `j` $=$ `sqrtN` $+$ `i` and see if any of these are squares mod n. For speed reasons, I would compute j^2 mod n, and then use your previous factoring code to factor it (you will want to optimize your previous factoring code so that it only goes up to $\sqrt{m}$ if you didn't already do this). Indeed, you actually probably want to make this faster. You only care about this factorization if all of the factors are in your `factorBase`.

Ok, now that you have your list of factors, you need to fill in a row of a matrix based on whether you have an even or odd number of any given prime factors.

Remember, each column of this matrix will correspond to a prime in your list `factorBase`. Remember, you want a 0 if you have an even number of that prime or a 1 if you have an odd number of that prime in your factorization. If your number has factors not in your `factorBase` you will throw that j out and not even make a matrix row for it. You will also probably want to at least store your j for the future (because you are not keeping all of them). I would recommend using the matrix class from `sympy` mentioned above to store this.

Eventually, your matrix should have more rows than columns and a linear dependence relation between the rows must exist. You need to find this. I don't think `sympy` will do this for you. You will either need to write your own row reduction or do some cleverness with `sympy`. You can also try some searching for relations naively. If you only get some naive searching for relations (ie, any two rows are equal), this can still get you some partial credit.

Once you have a relation, proceed like we have done before.