21 June Lab Stuff

The trouble with quotes on the internet is that you can never know if they are genuine. – Abraham Lincoln

1 RSA

Let us remember how the RSA encryption programme works:

Alice wants people to send her secret messages. She constructs a public key as follows: choose two primes p and q, compute $n = pq$, find e relatively prime to $\varphi(pq)$ and compute $d = e^{-1} \bmod \varphi(pq)$. The public key is (n, e), and d is kept private for decryption purposes.

- Write a function `keygen(p,q)` that takes in two primes and outputs `[n,e,d]` as above. You should be able to do most of this by piecing together functions you have already written. The tricky bit is generating e; this needs to be relatively prime to $\varphi(pq)$ so you can't just choose anything. You can use your GCD function to test whether any particular e works, but you'll need some way to selecting different e's until you find one that works.

Now you are Bob and you want to send Alice a message using her public key $(1405661, 125)$. To do so you write your message as a number m and compute $c = m^{125} \bmod 1405661$. The number c is the encrypted message you send to Alice.

- Write a function `RSA(message,n,e,blocksize)` that first blocks up the message as per last time, then performs RSA on each of the pieces. For example, using the given public key, I could run my function `RSA('SEASONSINTHEABYSS',1405661,125,4)` and it does the following:

  ```
  'SEASONSINTHEABYSS'

  'SEASONSINTHEABYSSXXX'

  [SEAS, ONSI, NTHE, ABYS, SXXX]

  [319090, 255328, 241518, 1318, 332537]

  [319090^125 mod 1405661, 255328^125 mod 1405661, 241518^125 mod 1405661, 1318^125 mod 1405661, 3325

  [559261, 596833, 232812, 1169293, 473729]
  ```

Now you are Alice again and you want to decypt the message that Bob has sent to you. This is accomplished by raising all of the numbers you received to the secret power `d` that you computed with your key, reducing mod `n` and changing things back to letters.

- Write a function `unRSA(message,n,d,blocksize)` that undoes the above process. The code for this can look almost exactly the same as for `RSA`; the difference is that `RSA` changes words to numbers at the beginning, while `unRSA` changes numbers to words at the end.

If you get your code working, go to `http://summermathprogram2016.blogspot.com/` to find an RSA public key I've posted. Use this to post a message in the comments section and I'll decrypt it (make sure you tell me the blocksize!). If you post your own public key in the comments section I'll send you a message that you can decrypt.

2 Factoring method

We have seen that factoring $n = pq$ can be accomplised relatively easily if either $p-1$ or $q-1$ has prime factorization involving only small primes. Roughly, if $p-1$ has small prime factors, then $p-1$ will divide $k!$ for some k that is not too big. We write $k! = (p-1)m$ and get

$$a^{k!} = a^{m(p-1)} = (a^m)^{p-1} \equiv_p 1$$

The last equvalence comes from Fermat's little theorem; *everything* to the $p-1$ power is equivalent to 1 modulo p.

The equation $a^{k!} \equiv_p 1$ means that p evenly divides $a^{k!} - 1$. If q does not evenly divide $a^{k!} - 1$ then $\gcd(a^{k!} - 1, pq) = p$ and we have successfully factored the number.

This leads to the following algorithm that attempts to factor a number $n = pq$:

- Choose a.
- If $\gcd(a, n) > 1$ then you have found a factor.
- Compute $a^{2!} \bmod n$. If $1 < \gcd(a^{2!} - 1, n) < n$ then you have found a factor.
- Compute $a^{3!} \bmod n$. If $1 < \gcd(a^{3!} - 1, n) < n$ then you have found a factor.

  $\vdots$

- Compute $a^{k!} \bmod n$. If $1 < \gcd(a^{k!} - 1, n) < n$ then you have found a factor.

  $\vdots$

It may be that some choices of a work better than others. How does one know when to stop the above process? By Euler's generalization of Fermat's little theorem, we know that so long as a is relatively prime to n then $a^{\varphi(n)} \equiv_n 1$. This means that once k is large enough so that $k!$ contains all of the prime factors of $\varphi(n)$, then $a^{k!}$ will be equivalent to 1 modulo n for all larger k.

So, the short way to answer the question of when to stop is, if you get $a^{k!} \equiv_n 1$ for some k and you haven't found a divisor yet, stop because a to all higher factorials will be 1 also.

- Write a function `factortest(n,a)` that tries to find a factor of `n` using an initial guess `a`, as per the algorithm above. This is most easily accomplished using a `while` loop.

If you think you have your code working, use it to factor `254781501911375497200433027277863476921 1` into two primes. You should be able to use any small number for `a` (I tried 2,3,4 and they all worked). You may have to let your program run for a minute or two. Consider that since each of the prime factors that divide this number are on the order of 10^{19}, if one tried to find the factors by starting at 2 and dividing by every number it would take more than 350,000,000,000 years if you could check one-billion numbers per second. With this in mind, a little under two minutes is pretty good!