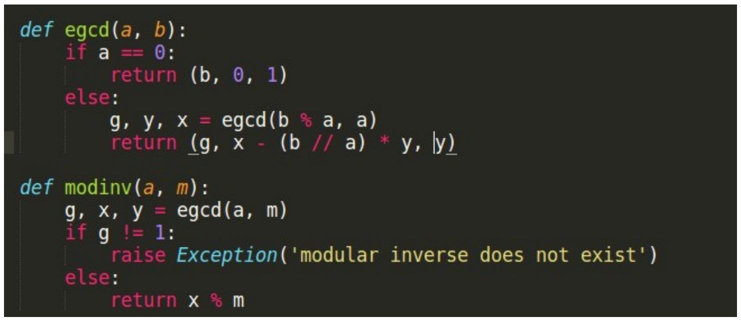
How I got the private key in task2

At first I tried finding a list of 32-bit prime numbers online so I could check if N % prime == 0 for any of them, once I found one prime, finding the other would be easy. Finding a list of primes proved to be too time consuming. Instead, I printed N into the console and put it into WolframAlpha which gave me p and q. These were hard coded in.

Next, I found helper function online that would help me calculate the modular inverse of two numbers. The code was grabbed from http://stackoverflow.com/questions/4798654/modular-multiplicative-inverse-function-in-python and looked like this:



```
def egcd(a, b):
    if a == 0:
        return (b, 0, 1)
    else:
        g, y, x = egcd(b % a, a)
        return (g, x - (b // a) * y, y)

def modinv(a, m):
    g, x, y = egcd(a, m)
    if g != 1:
        raise Exception('modular inverse does not exist')
    else:
        return x % m
```

I knew phi would be (p-1) * (q-1). I calculated phi using the p and q found on WolframAlpha, and used modinv(e, phi) to find d.

My understanding of the weak key problem

The problem posed in the weak key problem was that some RSA keys aren't generated with enough Entropy, which means a lot of them share a p or a q. This is problematic because it is significantly less time consuming to find the greatest common denominator of two numbers than it is to find the factorization of one number. The authors of "Mining Your Ps and Qs: Detection of Widespread Weak Keys in Network Devices" were able to make the process of finding the gcd quicker for multiple numbers, which allowed them to quickly find a "p" for many keys. Once a p is found, finding a "q" is as simple as dividing n/q. This breaks security by making it easy to generate the private key.

How I got the private key in task 3

Since finding the gcd of two numbers is an efficient task, I was able to use pythons built in function for it (which already uses the efficient Euclidean algorithm). My function for knowing if two N's shared a common denominator consisted of returning whether or not the result of calling gcd() on both of them was 1. If it wasn't 1, the two shared a factor.

To get the private key, I would first calculate p by calling gcd(n1, n2). Getting q was than as easy as dividing n1 by p. Once I had p and q, the problem was the same as that in task2 and I followed the same process.