Yamin Mousselli
CS 4235/6035: Introduction to Information Security

Project 3 Writeup

Task 1
In order to decrypt a message, we can use (m is equivalent to (c^d) mod N). I believe it's stated in both project 3 and PublicKeyCryptography Slides. We use pow because it's more efficient (have not done the time complexity analysis) than decrypting the message via the old-fashioned way. I found this out through https://stackoverflow.com/questions/32738637/calculate-mod-using-pow-function-python

Task 2
Weak passwords are susceptible to brute-force attacks including dictionary attacks, and can be determined relatively effortlessly using lookup tables, reverse lookup tables, and rainbow tables.

The reason why look lookup tables and rainbow tables work is because the same password will map to the same hash. Salts were introduced to combat this and a salt is appended or prepended to a password before it is hashed. Salts will essentially randomize each hash so that no two passwords will have the same hash. Since the attacker will not know the salt(s), it will be extremely difficult for him/her to build their lookup or rainbow tables. Everything mentioned before is true if and only if a random salt is generated per password (i.e., for each password including changed passwords) and every salt's length is appropriately long.

Since the most common passwords and salts (another password) are stored in *top_passwords.txt*, we will iterate through this list, one by one, and prepend a salt to a password. We will take the SHA256 hash of this result and encode it in hex because all the password hashes in *hashes4studenttask2.json* are in hex. We will then compare the password and salt we hashed, *hashed_password*, with the SHA256 hash of a password we received, *password_hash.* If they are the same, then we have cracked the password. If we do not find a match after we have tried every possible combination, then the password or salt does not exist within *top_passwords.txt* and we cannot crack the password. If this is the case, then we return our default values letting us know this.

Yes, the hash of the salted password is still vulnerable because we are not generating appropriately long and random salts. As stated before, salts are only effective if a different one is used per password (i.e., unique for all passwords). We can mitigate many attacks by using an appropriately long and random salt. Additionally, we can enhance security by not using common passwords and by enforcing a strong password policy. Example: the policy would require using a password or passphrase comprised of a minimum of 10 characters that must include at least one digit (0-9), one symbol, an uppercase character, a lowercase character, cannot be user's name or username, cannot be anything related to the user that would uniquely identify him/her (e.g., profile information such as birth date, address, secret question answers, etc), and would require the user to change their password every 6 months.

I read the following:

- https://crackstation.net/hashing-security.htm
- https://stackoverflow.com/questions/1645161/salt-generation-and-open-source-software/1645190#1645190

Task 3

Asymmetric encryption is based on modular arithmetic. It is important to note that only numbers relatively prime (i.e., their gcd = 1) to N have a multiplicative inverse. If the numbers are not relatively prime, then they are co-prime (aka mutually prime) because there exists multiple common factors (i.e., their gcd is not **only** 1). Given N, we can factor it into (p,q) and compute the private key, *d,* by the following:

- d = e^(-1) mod totient(N)
- e^(-1) is modular inverse of e mod phi(N)
- totient(N): phi(N) = (p-1)*(q-1)
- We can then use the Extended Euclidean Algorithm.

We can use the Euclidean Algorithm to calculate the gcd of two numbers. The last non-zero remainder is the gcd. We can use the Extended Euclidean Algorithm to write the gcd as a linear combination of the original two numbers that you're finding the gcd of. This is because the gcd of two numbers will be equal to a linear combination of he original two numbers you're finding the gcd for. This is true for every pair of positive integers you can find. Use the Extended Euclidean Algorithm to find the weights in the linear combination. I created two subroutines, *modular_inverse()* and *egcd_0()* to find the modular inverse by using the Extended Euclidean Algorithm and I utilized this website for my implementation of these subroutines:
https://en.wikibooks.org/wiki/Algorithm_Implementation/Mathematics/Extended_Euclidean_algorithm#Recursive_algorithm_2

We know N should have been generated by multiplying two prime numbers that are relatively prime to each other. In other words, the prime factorization (breaking N into the product of a bunch of prime numbers) should only consist of p and q. That is, if you find p, you can also find q by dividing n by p (also stated in task 4 paper). We must remember that for any two numbers (either integer or float) in python x and y, x/y will *always* be a float. x//y will be an integer and so we use the latter to force python3 to do division in integer mode. We only need to find one factor besides 1 and n. We can search for all the numbers beginning at 2 up to the square root of n so that we limit the number of values we have to look at to find a factor (we're only looking at primes). I used this information to implement *get_factors*. On another note, there's a faster way to compute the prime factorization than the method described above. We can use Pollard's Rho Algorithm to compute the factors more efficiently (have not done time complexity analysis).

As stated in the awesome paper we have to read for task 4, "Anyone who knows the factorization of N can efficiently compute the private key for any public key (e, N) using d = e^(-1) mod (p-1)*(q-1). In contrast to factoring, the gcd of two integers can be computed efficiently with Euclid's algorithm, especially with a small key size of only 64 bits.

Task 4

In task 4, we were told the "random" number generator used for key generation was vulnerable. The research paper explains very well why random number generators used in RSA can be vulnerable and how an attacker can deduce the prime factorization needed to get the private key(s). Randomness is very important in the field of cryptography and "… security often depends on the keys being chosen uniformly at random (Heninger, Durumeric, Wustow, Halderman, 2012)". Entropy essentially denotes bits of randomness and it is of utmost importance to have reasonably strong entropy for cryptographic algorithms, especially during key generation. Section 2.1 describes how an attacker can efficiently compute an RSA private key. Section 4.2 also talks about the consequences of sharing a prime factor, *p or q*, with another key, due to a lack of entropy. It is very important that systems not generate keys on boot up time, and that any private keys generated by any system are manually regenerated after bootup and some uptime because there is a major lack of entropy within a system's input pool (/dev/random or /dev/urandom). Repeating keys and default keys are a threat to any cryptosystem and they should be avoided at all costs. Having the system's RNG generate private keys increases the probability of collisions and doesn't offer high entropy because the number of possible primes produced is very low and therefore N can easily be factored. The fact that our key size is only 64 bits does not help and only makes the attacker's job easier. The security community and mathematics suggests the product of p and q should be reasonably long to make the key computationally infeasible to crack. This is all explained in the paper, and explained in much greater and accurate detail.

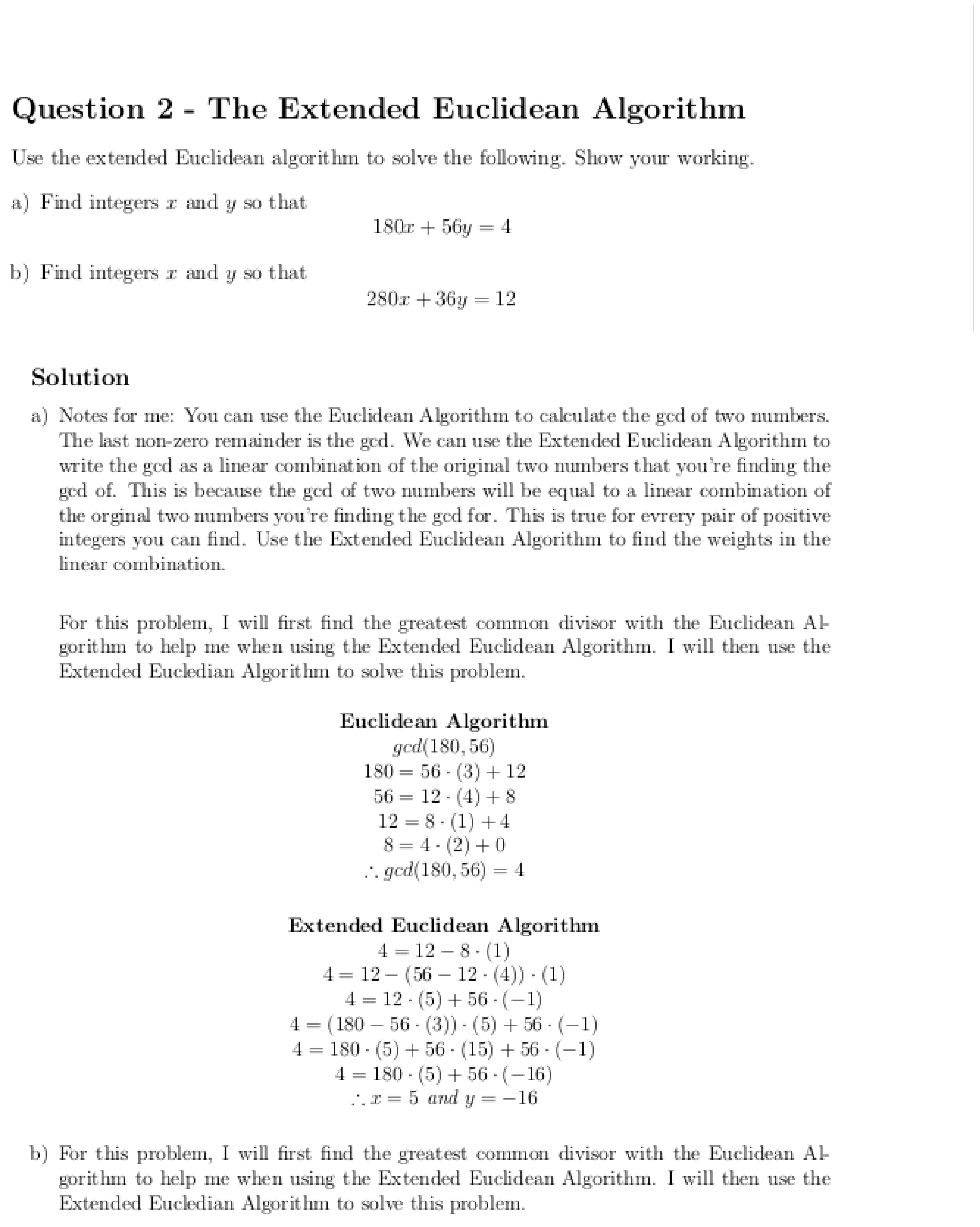
I read the links below and especially Rakesh's question. I looked at his code and basically implemented it to help me find Waldo. I created another helper, *egcd_1(),* which is implemented the same as *egcd_0*() with a minor modification to the return statement. I duplicated my process for obtaining the private key in task 3 with a few modifications. I assigned p to be the gcd of n1 and n2. Next, we can determine q by dividing n1 with the p we calculated above. We can then utilize the totient function to determine phi(N). We do so by multiplying (p-1) and (q-1). Finally, we obtain our private key by solving the modular inverse, and passing in both phi(N) and the exponent. We do this by calling *modular_inverse().* All this is explained in section 2.1 of the paper as well as my answer to task 3.

I read the following:

- https://factorable.net/weakkeys12.conference.pdf
- https://security.stackexchange.com/questions/69250/problem-with-common-factor-in-rsa
- https://security.stackexchange.com/questions/69140/with-rsa-can-one-ensure-that-two-entities-will-not-have-the-same-private-key

Task 5

The broadcast RSA attack is also known as Hastad's broadcast attack. Essentially, this attack will enable us to determine the original message without knowing any of the private keys. We can recover the message because all the keys used the same exponent. We'll want to get m^e mod (n1*n2*n3) by the Chinese Remainder Theorem (CRT). It is important to understand that n1, n2, n3, are pairwise co-prime and their congruences ((c1 = m^e mod n1), (c2 = m^e mod n2), and (c3 = m^e mod n3)) are relatively prime. Since their congruences are relatively prime, we can use CRT. We want to set up a system of equations and solve. I typed two solutions to two CS 3511 homework problems in Latex and it involves the Extended Euclidean Algorithm and CRT. CRT uses the solutions from the previous problem which used the Extended Euclidean Algorithm. These images will suffice as my explanation of CRT.



Question 2 - The Extended Euclidean Algorithm

Use the extended Euclidean algorithm to solve the following. Show your working.

a) Find integers x and y so that

$$180x + 56y = 4$$

b) Find integers x and y so that

$$280x + 36y = 12$$

Solution

a) Notes for me: You can use the Euclidean Algorithm to calculate the gcd of two numbers. The last non-zero remainder is the gcd. We can use the Extended Euclidean Algorithm to write the gcd as a linear combination of the original two numbers that you're finding the gcd of. This is because the gcd of two numbers will be equal to a linear combination of the orginal two numbers you're finding the gcd for. This is true for evrery pair of positive integers you can find. Use the Extended Euclidean Algorithm to find the weights in the linear combination.

For this problem, I will first find the greatest common divisor with the Euclidean Algorithm to help me when using the Extended Euclidean Algorithm. I will then use the Extended Eucledian Algorithm to solve this problem.

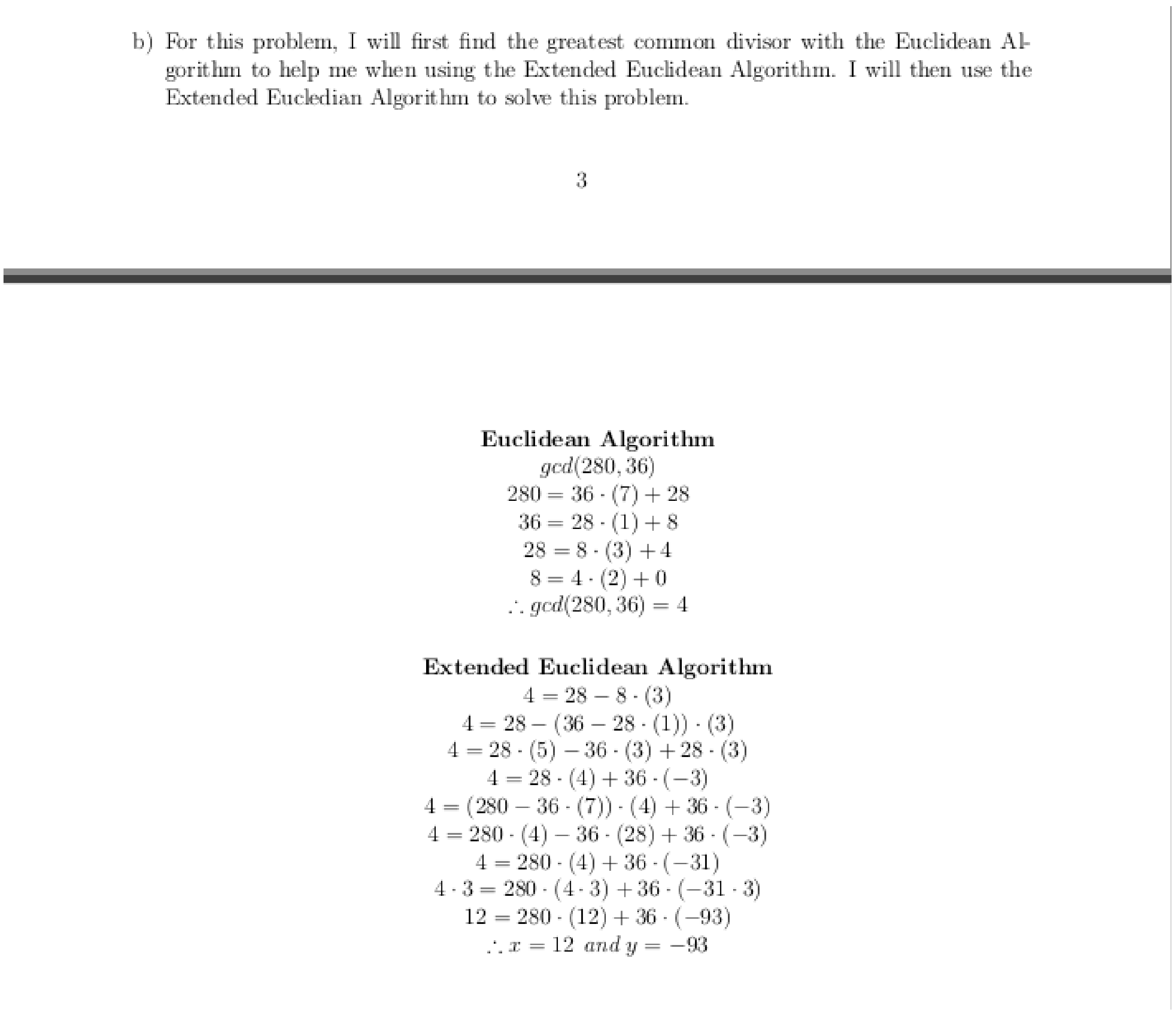
Euclidean Algorithm

$$gcd(180, 56)$$
$$180 = 56 \cdot (3) + 12$$
$$56 = 12 \cdot (4) + 8$$
$$12 = 8 \cdot (1) + 4$$
$$8 = 4 \cdot (2) + 0$$
$$\therefore gcd(180, 56) = 4$$

Extended Euclidean Algorithm

$$4 = 12 - 8 \cdot (1)$$
$$4 = 12 - (56 - 12 \cdot (4)) \cdot (1)$$
$$4 = 12 \cdot (5) + 56 \cdot (-1)$$
$$4 = (180 - 56 \cdot (3)) \cdot (5) + 56 \cdot (-1)$$
$$4 = 180 \cdot (5) + 56 \cdot (15) + 56 \cdot (-1)$$
$$4 = 180 \cdot (5) + 56 \cdot (-16)$$
$$\therefore x = 5 \ and \ y = -16$$

b) For this problem, I will first find the greatest common divisor with the Euclidean Algorithm to help me when using the Extended Euclidean Algorithm. I will then use the Extended Eucledian Algorithm to solve this problem.



b) For this problem, I will first find the greatest common divisor with the Euclidean Algorithm to help me when using the Extended Euclidean Algorithm. I will then use the Extended Eucledian Algorithm to solve this problem.

Euclidean Algorithm

$$gcd(280, 36)$$
$$280 = 36 \cdot (7) + 28$$
$$36 = 28 \cdot (1) + 8$$
$$28 = 8 \cdot (3) + 4$$
$$8 = 4 \cdot (2) + 0$$
$$\therefore gcd(280, 36) = 4$$

Extended Euclidean Algorithm

$$4 = 28 - 8 \cdot (3)$$
$$4 = 28 - (36 - 28 \cdot (1)) \cdot (3)$$
$$4 = 28 \cdot (5) - 36 \cdot (3) + 28 \cdot (3)$$
$$4 = 28 \cdot (4) + 36 \cdot (-3)$$
$$4 = (280 - 36 \cdot (7)) \cdot (4) + 36 \cdot (-3)$$
$$4 = 280 \cdot (4) - 36 \cdot (28) + 36 \cdot (-3)$$
$$4 = 280 \cdot (4) + 36 \cdot (-31)$$
$$4 \cdot 3 = 280 \cdot (4 \cdot 3) + 36 \cdot (-31 \cdot 3)$$
$$12 = 280 \cdot (12) + 36 \cdot (-93)$$
$$\therefore x = 12 \ and \ y = -93$$

Question 3 - Mod Problems Require Mod Solutions

There are some students in a college, whose number is unknown. If we separate them into groups of 5, we have 3 left over; into groups of 36, we have 12 left over; and into groups of 56, we have 4 left over. Find the smallest possible number of students. You may use results from Problem 2, but show any additional working.

Solution

Notes to self: You have to check if the congruences are relatively prime (their gcd = 1) to use the CRT. On the other hand, co-prime, aka mutually prime, is when the common factor is not only 1 (there exists muultiple common factors).

I solved this problem using the Chinese Remainder Theorem. From the problem description, we have the following:

$$N \equiv 3 \bmod 5$$
$$N \equiv 12 \bmod 36$$
$$N \equiv 4 \bmod 56$$

In order to determine N, we can first use the Chinese Remainder Theorem to break the problem into three parts for each of the mod spaces, respectively

$$N = ((\tfrac{36 \cdot 56}{4}) \cdot a) + (5 \cdot 56 \cdot b) + (5 \cdot 36 \cdot c)$$

Since 36 *and* 56 are not relatively prime, we divide by 4 because that is their gcd. We need this for when we apply the Extended Euclidean Algorithm. We will solve for each constant by using our results from number two. We will now solve for the constants in each mod space.

Solve for c

$$5 \cdot 36 \cdot c \equiv 4 \bmod 56$$
$$5 \cdot 36 \cdot c + 56 \cdot (y) = 4$$
$$180 \cdot (c) + 56 \cdot (y) = 4$$

Now we can substitute our findings from 2a)

$$180 \cdot (5) + 56 \cdot (-16) = 900 + -896 = 4$$
$$\therefore c = 5$$

Solve for b

$$5 \cdot 56 \cdot b \equiv 12 \bmod 36$$
$$280 \cdot (b) + 36 \cdot (y) = 12$$

Now we can substitute our findings from 2b)

$$280 \cdot (12) + 36 \cdot (-93) = (3360 - 3348) = 12$$
$$\therefore b = 12$$

$$\therefore b = 12$$

Solve for a

$$((\frac{36 \cdot 56}{4}) \cdot a) \equiv 3 \bmod 5$$
$$504 \cdot (a) + 5 \cdot (y) = 3$$
$$504(-1 \cdot 3) + 5 \cdot (100 \cdot 3) = 3$$
$$\therefore a = -3$$

5

Now we can find N. $N = ((\frac{36 \cdot 56}{4}) \cdot -3) + (5 \cdot 56 \cdot 12) + (5 \cdot 36 \cdot 5))$

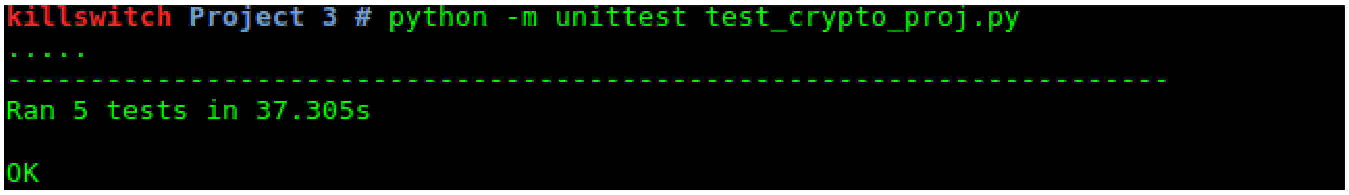
$$N = -1512 + 3360 + 900 = 2748$$

Now we can find the $LCM(5, 14, 9) = 2748$

After that, $N = 2748 \bmod 630 = 228 \therefore N = 228$

I had to rewrite my modular inverse method so many times to get my implementation to run efficiently (sad face). I should have started with the recursive implementation of the extended euclidean algorithm, but lesson learned. I literally recovered the message with the same formulae as described above. There is one exception. After I determined the value of the ciphertext, I took the cubic root of it to decrypt the message. I used binary search to find the cubic root and thus wrote another helper method for that.

A snippet of my terminal after running the unit test is shown below:



```
killswitch Project 3 # python -m unittest test_crypto_proj.py
.....
----------------------------------------------------------------------
Ran 5 tests in 37.305s

OK
```

I read the following:

- https://crypto.stackexchange.com/questions/6713/low-public-exponent-attack-for-rsa
- https://crypto.stackexchange.com/questions/52504/deciphering-the-rsa-encrypted-message-from-three-different-public-keys?fbclid=IwAR1auzY4UOmIUTfg9HzLemqP9qNcnB2QQ1HzqSHysYa4WSsic9i7Itm0CLw
- https://crypto.stackexchange.com/questions/47063/can-you-help-me-understand-the-rsa-broadcast-attack
- https://en.wikipedia.org/wiki/Coppersmith's_attack#H.C3.A5stad.27s_broadcast_attack