April 2022

A Deep Dive Into Modern Encryption

Ishaan Singh

1 Introduction

Encryption is defined as the process by which information is rewritten in order to hide its true contents. For many years, I have enjoyed learning more about computers and the software that runs within them, partaking in a multitude of competitive programming events. Recently, I took it upon myself to learn the intricacies of string hashing (the process of turning a sequence of characters into an integer value) which has a deep basis in modular arithmetic. Having enjoyed learning about this topic as well as being told of the importance of security/privacy in the modern world, I decided that encryption algorithms would be the next step up from basic string hashing and would be beneficial to learn. The aim of this paper is to gain a deeper understanding of encryption, its weaknesses/security issues, why/how various ciphers work, and different types of ciphers.

2 Substitution Ciphers

A simple example of a substitution cipher is the ROT13 cipher which is a variant of the caesar cipher. The Caesar cipher is a cipher that rotates each letter in the alphabet such that they map to a different letter x letters away (Shimeall Spring, n.d.). For example, a caesar cipher with a shift of 1 follows the mapping of:

$A \rightarrow B$, $B \rightarrow C$, $C \rightarrow D$, .. $Z \rightarrow A$

Similarly, a ROT13 cipher is a caesar shift of 13. ROT13 is used frequently as its inverse function is the same as the function itself. Shifting the alphabet back 13 letters results in the same mapping as shifting the alphabet forward 13 letters. This is due to the fact that the alphabet is 26 letters long and 26/2 = 13. In other words, if x is the plaintext that is to be ciphered, then ROT13(x) is the cipher, and ROT13(ROT13(x)) = x.

2.1 Matrix Multiplication

A key component of the hill cipher is matrix multiplication. A matrix is defined as "a rectangular array of quantities or expressions in rows and columns that is treated as a single entity and manipulated according to particular rules". Let's

try to find the result of the following expression:

$$\begin{bmatrix} 6 & 9 & 10 \\ 3 & 7 & 8 \end{bmatrix} \cdot \begin{bmatrix} 4 & 2 \\ 13 & 17 \\ 3 & 4 \end{bmatrix} = \begin{bmatrix} w & x \\ y & z \end{bmatrix}$$

There are a few guidelines/rules when it comes to matrix multiplication. Firstly, order matters in matrix multiplication. Let's consider the product of two matrices, A and B where A is the first term in the expression. Matrix $A \cdot B$ is not necessarily equivalent to $B \cdot A$. Additionally, the number of rows in A must equal the number of columns in B and the number of columns in A must equal the number of rows in B. The resulting matrix has the same number of rows as in A and the same number of columns as in B. Therefore, the product of the expression above should have 2 rows and 2 columns.

The value of w in the resulting matrix is equivalent to the dot product of matrix A's first row (as a vector) and matrix B's first column (as a vector):

$$w = \begin{bmatrix} 6 & 9 & 10 \end{bmatrix} \cdot \begin{bmatrix} 4 \\ 13 \\ 3 \end{bmatrix} = 6 \cdot 4 + 9 \cdot 13 + 10 \cdot 3 = 24 + 117 + 30 = 171$$

Similarly, x is equivalent to the dot product of matrix A's first row (as a vector) and matrix B's second column (as a vector):

$$x = \begin{bmatrix} 6 & 9 & 10 \end{bmatrix} \cdot \begin{bmatrix} 2 \\ 17 \\ 4 \end{bmatrix} = 6 \cdot 2 + 9 \cdot 17 + 10 \cdot 4 = 12 + 153 + 40 = 205$$

When computing the second row, simply use the second row of matrix A instead of the first row.

2.2 Hill Cipher - Encryption

Alike the ROT13 cipher in that they are both substitution ciphers, the hill cipher, developed by Lester S. Hill in 1929. To encrypt a plaintext message using the hill cipher, start by separating the message into groups of equal size. As an example, let's encrypt the message, "first example" with groupings of length 2. Along with the message, we have to choose a key, in the hill cipher, the key is a n x n matrix (where n is the length of the groupings, in this case, 2) (Mustafeez, n.d.). For this example, the key, K, that will be used is:

$$\begin{bmatrix} 4 & 3 \\ 13 & 17 \end{bmatrix}$$

Now, we consider the first grouping of 2 characters in the plaintext message which is "fi". We then convert this grouping into a vector in which the first value is "f"'s position in the alphabet (f is the 6th letter) and the second value being i's position in the alphabet (i is the 9th letter). Note, we subtract one from all letter's positions (the letter a is mapped to 0). We then multiply the key matrix with this vector taken modulo 26 (meaning each integer is now its remainder after dividing by 26):

$$\begin{bmatrix} 4 & 3 \\ 13 & 17 \end{bmatrix} \cdot \begin{bmatrix} 5 \\ 8 \end{bmatrix} = \begin{bmatrix} 44 \\ 201 \end{bmatrix} = \begin{bmatrix} 18 \\ 19 \end{bmatrix}$$

We are left with a vector which maps back to a two character text: "st" (as "s" maps to 18 and "t" maps to 19). We repeat the following process for the rest of the letters in the plaintext and receive the cipher: "stshk donpved".

2.3 Hill Cipher - Decryption

Consider our process to encrypt the plaintext using hill's cipher: $P \cdot K \pmod{26}$ where P is the plaintext and K is the key. So, in order to retrieve the plaintext, we can multiply the cipher (which is $P \cdot K$) by the modular inverse of K (where the modulo is 26). This results in $P \cdot K \cdot K^{-1} = P$, fully deciphering the plaintext message. In modular arithmetic, multiplying a matrix by its modular inverse results in the identity matrix. When the identity matrix is multiplied by a

matrix M, the result is M. Therefore, in order to decrypt this cipher, we need to find the modular inverse of K. For a 2x2 matrix, the identity matrix appears as follows:

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

Let's generalize the matrix K to (where a, b, c, and $d \in Z^+$) :

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix}$$

One way in which we can find the modular inverse of a matrix is via (if the matrix K is a 2x2) the formula:

$$\begin{bmatrix} d & -b \\ -c & a \end{bmatrix} \cdot (ad - bc)^{-1}$$

The second term is taken by its modular inverse with the modulo being 26.

This formula satisfies the claim that the product of the modular inverse of M and $M =$ is the identity matrix. To demonstrate this, let's start by multiplying K by the first term in the formula above:

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \cdot \begin{bmatrix} d & -b \\ -c & a \end{bmatrix} = \begin{bmatrix} ad - bc & bd - bd \\ -ac + ac & ad - bc \end{bmatrix} = \begin{bmatrix} ad - bc & 0 \\ 0 & ad - bc \end{bmatrix}$$

Now, we multiply the matrix above by the second term. In modular arithmetic, when an integer is multiplied by its modular inverse, the result is 1. Therefore:

$$\begin{bmatrix} ad - bc & 0 \\ 0 & ad - bc \end{bmatrix} \cdot (ad - bc)^{-1} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

The final product is the identity matrix, proving that our formula correctly finds the modular inverse of the matrix. After multiplying, the cipher matrix by the key matrix's inverse using the formula above, we are left with the product of P and the identity matrix which is simply, P.

2.4 Hill Cipher - Attacks

The Hill Cipher is completely decryptable when you have n sets of length n strings in which you know the cipher and plaintext values (where n signifies the length/width of the square matrix K). For example, if we were trying to decrypt the following cipher: "p ltpf nplk vrfqbb" and we are aware that the plaintext value for the first four letters are: "i lov" (spaces are excluded), if the matrix is a 2x2, we can solve for the matrix K. This will allow us to decrypt the rest of the plaintext message.

Let's generalize the matrix K to (where a, b, c, and $d \in Z^+$) :

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix}$$

Now, let's write out the encryption process in which we are encrypting "i l" and setting the cipher value to "p l" as those are the first two values of the plaintext and ciphertext, respectively.

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \cdot \begin{bmatrix} 8 \\ 11 \end{bmatrix} = \begin{bmatrix} 15 \\ 11 \end{bmatrix}$$

We perform the same process for the last two known letters of the plaintext and ciphertext: "ov" (plaintext) and "tp" (ciphertext)

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \cdot \begin{bmatrix} 14 \\ 21 \end{bmatrix} = \begin{bmatrix} 19 \\ 15 \end{bmatrix}$$

Using matrix multiplication, we generate the following equations from these two equations (Expression 1):

$$8a + 11b = 15$$

$$8c + 11d = 11$$

(Expression 2):

$$14a + 21b = 19$$

$$14c + 21d = 15$$

Note that all of these equations are taken modulo 26 due to the nature of the hill cipher as described in the previous

sections. We can now solve for the values a, b, c, d as we have formed a system of equations (Christensen). Since we have 4 variables and 4 unique equations, this system is solvable for all 4 variables. Let's start by solving for a and b:

$8a + 11b = 15$	multiply by 7
$14a + 21b = 19$	multiply by 4
$56a + 77b = 105$	
$(-)56a + 84b = 76$	Subtract bottom equation from top
$-7b = 29$	-7b = 19b as this expression is taken modulo 26
$19b = 29$	29 is equivalent to 133 (mod 26)
$19b = 105$	Divide by 19
$b = 7$	Solved for b, now we solve for a
$8a + 11(7) = 15$	Compute product of 11(7)
$8a + 77 = 15$	Subtract 77
$8a = -61$	-62 = 16(modulo 26)
$8a = 16$	Divide by 8
$a = 2$	

Now that we have solved for a and b we use the same process and solve for b and c giving us the key matrix:

$$\begin{bmatrix} 2 & 7 \\ 1 & 5 \end{bmatrix}$$

Using the decryption described in the previous section, we can calculate the rest of the plaintext message: “i love hill cipher”.

2.5 Hill’s cipher - Attacks vs. Brute Force

To test the efficiency of the approach used in the previous section, I wrote a python “stress tester” program that randomly generated plaintext messages as well as 2x2 matrices to be used as a key.

The brute force approach to solve a given cipher was to try all 26^4 possible values for the key matrix and then use this key to decrypt the cipher, generating 26^4 possible sequences of which only one is the correct plaintext message. For practical purposes, this would only work if the “attacker” had some knowledge of the original message. For example, if it followed proper grammar rules, then the attacker can run each generated sequence through a grammar checking library to reduce the size of the sequences he has to manually check significantly. If the original message can be complete gibberish, there is no way of knowing whether or not you have deciphered the text correctly using the brute force method. However, the purpose of this program is to demonstrate the significant time difference between a brute force attack and the attack discussed in the previous section. After generating one possible key, the program deciphers the cipher using the key, and checks whether or not the deciphered message is equivalent to the original message. If they are equivalent, the program quits automatically as it has successfully found the key matrix. If they are different, the program continues, generating and testing more key matrices. The time taken by this method is then recorded.

The program would then call another function, representing the attack described in the previous section in which the attacker has some information about the plaintext message. The “attacker” (the linear equation solving function) was given the cipher as well as the first four values of the plaintext message. It then completes the few system of equations necessary to solve for the key matrix (described in subsection 2.4). The time taken to decipher the message with this approach was also recorded.

This program was run 10 times. In each iteration the brute force method and the linear equation attack method were given the same cipher to decrypt. Listed below are the results and the total average time for each approach:

Deciphering Approach Comparison (in seconds)	
Brute Force Approach	System of Linear Equations Attack
4.06	$1.51 \cdot 10^{-7}$
3.72	$1.53 \cdot 10^{-7}$
3.29	$1.38 \cdot 10^{-7}$
0.17	$1.24 \cdot 10^{-7}$
3.45	$1.36 \cdot 10^{-7}$
1.77	$1.51 \cdot 10^{-7}$
0.98	$1.39 \cdot 10^{-7}$
1.71	$1.47 \cdot 10^{-7}$
4.02	$1.24 \cdot 10^{-7}$
4.09	$1.78 \cdot 10^{-7}$
Average Time (in seconds)	
2.72	$1.44 \cdot 10^{-7}$

As exhibited in the table, the average time for the linear equations attack was over 10^7 times faster than the brute force approach. This demonstrates the power of even having a little knowledge of the plaintext message when deciphering the hill cipher. Below is a snippet of the main code utilized to generate the right side of the table above:

```
def linearEquationAttack(plaintext, cipher):
    a1 = [ord(plaintext[0])-97,ord(plaintext[1])-97,ord(cipher[0])-97]
    a2 = [ord(plaintext[0])-97,ord(plaintext[1])-97,ord(cipher[1])-97]
    a3 = [ord(plaintext[2])-97,ord(plaintext[3])-97,ord(cipher[2])-97]
    a4 = [ord(plaintext[2])-97,ord(plaintext[3])-97,ord(cipher[3])-97]
    key = linearEquationSolver(a1,a3)
    k1 = linearEquationSolver(a2,a4)
    key.append(k1[0])
    key.append(k1[1])
    return decryptor(cipher, key)

def linearEquationSolver(eq, eq1):
    ogeq = eq[:]
    lcm = compute_lcm(eq[0], eq1[0])
    if(eq[0] != 0):
        eq[1]*=lcm/eq[0]
        eq[2]*=lcm/eq[0]
    if(eq[1] != 0):
        eq1[1]*=lcm/eq1[0]
        eq1[2]*=lcm/eq1[0]
    ft = (eq1[1]-eq[1])%26
    sc = (eq1[2]-eq[2])%26
    val = 0
    for i in range(0, 26):
        if((sc+i*26)%ft == 0):
            val = i
            break
    ret = [((sc+i*26)/ft)%26]

    ft = ogeq[0]
    sc = ogeq[2]-ogeq[1]*ret[0]
    for i in range(0, 26):
        if((sc+i*26)%ft == 0):
            val = i
            break
    ret.append(((sc + i * 26) / ft)%26)
    ret = [int(ret[1]),int(ret[0])]
    return ret
```

The method, "linearEquationAttack", takes in two arguements, the first four letters of the plaintext message, as well as the cipher. The method then creates the 4 linear equations using the procedure described in section 2.4 and calls "linearEquationSolver" which takes in 2 linear equations and solves for the 2 variables (which is half of the key). Therefore, "linearEquationSolver" is called twice, one for each half of the key.

How does "linearEquationSolver" solve these two equations? Let's take a look at the procedure:

1. Calculate the least common multiple of the first term in both equations
2. Multiply the first equation by an integer such that the first term is now equivalent to the least common multiple calculated in the previous step. Do the same for the second equation.
3. We can now subtract the second equation from the first and will be left with one equation and one variable.
4. The equation has been taken modulo 26 so the next step is to ensure that the coefficient of the remaining variable is the smallest positive value that is in its equivalence class. Two integers, x and y are in the same equivalence class, modulo 26, if $26|y-x$. We do the same for the right side of the equation (which contains some known integer).
5. The equation is currently in the format $z \cdot y = w$ where z and w are known positive integers. However, we don't know if $z|w$, therefore, we cannot simply divide both sides by z giving us y. Instead, the code searches for a positive integer in w's equivalence class that satisfies the condition $z|w$ by continuously adding 26 to the current value of w and checking to see if the condition is satisfied.
6. After a suitable value of w is found, both sides of the equation are divided by z leaving us with the solution for y.
7. We then substitute y back into the equation and solve for x by subtracting $z*y$ from w. A larger snippet of the code is available in the appendix.

3 RSA encryption

3.1 Background - Type of encryption

RSA encryption, named after its creators, Ron Rivest, Adi Shamir, and Leonard Adleman, is a cryptosystem that was published in 1977. It uses an asymmetric key model. This means that there is a public key used to encrypt the plaintext message and a separate private key used to decrypt the cipher. The idea behind this form of encryption is to ensure

confidentiality. Once the public key is published, anyone may send encrypted messages to the recipient, however, intercepted messages cannot be decrypted easily. The only way to decipher the encrypted message is with the private key owned only by the intended recipient owns.

3.2 Encryption/Decryption Process

Both the public and private key in RSA encryption is made up of a pair of integers. The process for encrypting a message with RSA is the same as decrypting a message with RSA (Khan Academy, Cryptography — computer science — computing). The only difference is that decryption uses the private key whereas encryption uses the public key. Given the public key: (7,33), let's encrypt the letter "e". We need to use an integer value for plaintext message, so let's use the integer 5 as the letter "e" is in the third position of the alphabet. The process for encryption with our key is defined as: $5^7 \ (mod\ 33)$. This is equivalent to 14 which can be represented as the letter "n".

If we now have the private key: $(3,33)$ and the cipher being 14, we can decipher the original plaintext message by doing: $14^3 \ (mod\ 33) = 5$. 5 represents the letter "e" so we have successfully encrypted and decrypted the plaintext message.

3.3 Generating the keys

To generate the keys, we start by choosing two primes, p and q. As an example, let's use $p = 3$ and $q = 11$. The second integer component in both the public and private key is the product of these two integers, $p \cdot q = 3 \cdot 11 = 33$. Let this value be known as n. We then find $\phi(n)$, the function, ϕ being known as "phi". The "phi" function calculates the number of integers co-prime to n that are also less than n. Two integers are co-prime to each other if they don't share any factors other than 1. $\phi(33)$ is calculated by considering 1,2,4,5,7,8,10,13, and so on.

3.3.1 Proving $\phi(n) = (p-1) \cdot (q-1)$

Due to the property that p and q are prime, $\phi(n) = (p-1) \cdot (q-1)$. This is because, the prime factorization of n is $p \cdot q$. Therefore, any integer that does not have a factor of p or q, is co-prime to n. There are $pq-1$ integers less than pq, so we consider the range $0 - pq - 1$. Within this range, there are $p-1$ multiples of q and $q-1$ multiples of p, all of which are not co-prime to n. There is no overlap between these two sets of integers as the lowest common multiple of p and q is pq as they are prime. Since these are the only multiples of p and q in the valid range:

$$(pq-1)-(p-1)-(q-1)$$

$$= pq-1-p+1-q+1$$

$$= pq-p-q+1$$

$$= (p-1)(q-1)$$

3.3.2 Generation of Keys continued

Now that we have derived $\phi(n)$, we know that $\phi(33) = 10 \cdot 2 = 20$. This value represents the upper bound for another variable e that we will now generate. In order to find a suitable value for e, we need to follow 2 rules: $1 < e < \phi(n)$ and e must be co-prime with n and $\phi(n)$. In our example, one suitable e value is 7. We have now completed the public key which is in the format: (e,n).

The last value we have to derive to complete the private key is d. We must choose d such that $d \cdot e \ (mod \ \phi(n)) = 1$. For our example, d must follow $d \cdot 7 \ (mod \ 20) = 1$. One suitable d value is 3 as $3 \cdot 7 \ (mod \ 20) = 21 \ (mod \ 20) = 1 \ (mod \ 20)$. The private key is in the format: (d,n).

We have now completed both keys: $(7,33)$ (public) and $(3,33)$ (private).

3.4 Proof of correctness

RSA encryption: $m^e \ (mod \ n) = c$
RSA decryption: $c^d \ (mod \ n) = m$

In order to prove that the process described in the sections above correctly forms a public and private key and can be used as a valid encryption method, we need to show that the above conditions are satisfied in conjunction. Need to prove:

$(m^e)^d \ (mod \ n) = m$

Demonstrating this would prove RSA encryption as it would prove that, after encrypted, and then decrypted, the

original plaintext message returns to itself.

Recall that $e \cdot d = 1 \ (mod \ \phi(n))$. Therefore, $e \cdot d = \phi(n) \cdot k + 1$ where $k \in Z^+$, giving us:

$$m^{\phi(n)k+1} \ (mod \ n) = m \cdot m^{\phi(n)k} \ (mod \ n)$$

Euler's theorem (Euler's Totient Function and Euler's Theorem, n.d.) states that $a^{\phi(n)} = 1 \ (\text{mod } n)$. Therefore, we are left with:

$$m \cdot 1^k \ (mod \ n) = m$$

We have reached the RHS from the LHS, successfully proving that RSA is a valid method of encryption.

4 Conclusion

In this paper, hill's cipher and RSA encryption were both tackled in detail. I have gained a much stronger knowledge on encryption as well as modular arithmetic and matrices. I have not used matrices in the past so this was a great introduction to them. Additionally, I proved various theorems throughout the paper which helped develop a better sense of how to approach proofs and the different ways to prove a theorem. I also used programming to demonstrate ideas learned throughout the paper (such as the advantage of having a few plaintext letters when decrypting a cipher). Mainly, I used programming to exhibit the significance of even a little bit of knowledge of the plaintext message when it comes to encryption and in the future, it would be interesting to look into attacks for RSA encryption. RSA encryption however, is far securer and is able to prevent more basic/standard attacks. For the future, I would like to analyze other types of ciphers such as transposition ciphers and their advantages/disadvantages when compared to the substitution ciphers described in this paper. It would also be quite interesting to dive deeper into modular arithmetic, which was used significantly throughout the paper but was not a focal point. Additionally, I did not completely explore the applications of these encryption techniques to the world of competitive programming, the motivation for this research. However, the knowledge I gained from the modular arithmetic side is extremely helpful as it is used constantly in hashing and other math problems.

5 Bibliography

- Khan Academy. (n.d.). Cryptography — computer science — computing. Khan Academy. Retrieved May 1, 2022, from https://www.khanacademy.org/computing/computer-science/cryptographymodern-crypt
- Christensen, C. (n.d.). Hill Cipher Cryptanalysis. Retrieved May 1, 2022, from https://www.nku.edu/ christensen/Section
- Mustafeez, A. Z. (n.d.). What is the Hill cipher? Educative. Retrieved May 1, 2022, from https://www.educative.io/edpresso/what-is-the-hill-cipher
- Katz, A., Ng, A., amp; Bourg, P. (n.d.). RSA encryption. Brilliant Math amp; Science Wiki. Retrieved May 1, 2022, from https://brilliant.org/wiki/rsa-encryption/
- Shimeall, T. J., amp; Spring, J. M. (n.d.). Substitution cipher. Substitution Cipher - an overview — ScienceDirect Topics. Retrieved May 1, 2022, from https://www.sciencedirect.com/topics/computer-science/substitution-cipher
- Euler's Totient Function and Euler's Theorem. Euler's totient function and Euler's theorem. (n.d.). Retrieved June 5, 2022, from https://www.doc.ic.ac.uk/ mrh/330tutor/ch05s02.html

6 Appendix

- Figure 1: Larger Coding Snippet Processing Hill Cipher Attack vs. Brute Force

```
def stressTester():
    averageBrute = 0
    averageAttack = 0
    for i in range(10):
        key = []
        message = ""
        for i in range(4):
            key.append(random.randint(1,25))
        for i in range(10):
            message+=chr(random.randint(1,25)+97)
        cipher = encryptor(message, key)
        print(message, cipher, key)
        startTime = time.perf_counter()
        bruteForce(0,[],cipher,message)
        endTime = time.perf_counter()
        averageBrute+=endTime-startTime
        startTime = time.perf_counter()
        if(linearEquationAttack(message, cipher) != message):
            print(linearEquationAttack(message, cipher))
        endTime = time.perf_counter()
        averageAttack+=endTime-startTime
    print("Brute Force Average: " + averageBrute)
    print("Attack Time Average: " + averageAttack)

def bruteForce(index, key, cipher, message):
    if(index < 4):
        for i in range(0, 26):
            k = key[:]
            k.append(i)
            ft = bruteForce(index+1, k, cipher, message)
            if(ft != False):
                return True
    else:
        if(decryptor(cipher, key) == message):
            return True
    return False

def linearEquationAttack(plaintext, cipher):
    a1 = [ord(plaintext[0])-97,ord(plaintext[1])-97,ord(cipher[0])-97]
    a2 = [ord(plaintext[0])-97,ord(plaintext[1])-97,ord(cipher[1])-97]
    a3 = [ord(plaintext[2])-97,ord(plaintext[3])-97,ord(cipher[2])-97]
    a4 = [ord(plaintext[2])-97,ord(plaintext[3])-97,ord(cipher[3])-97]
    key = linearEquationSolver(a1,a3)
    k1 = linearEquationSolver(a2,a4)
    key.append(k1[0])
    key.append(k1[1])
    return decryptor(cipher, key)

def linearEquationSolver(eq, eq1):
    ogeq = eq[:]
    lcm = compute_lcm(eq[0], eq1[0])
    if(eq[0] != 0):
        eq[1]*=lcm/eq[0]
        eq[2]*=lcm/eq[0]
    if(eq[1] != 0):
        eq1[1]*=lcm/eq1[0]
        eq1[2]*=lcm/eq1[0]
    ft = (eq1[1]-eq[1])%26
    sc = (eq1[2]-eq[2])%26
    val = 0
    for i in range(0, 26):
        if((sc+i*26)%ft == 0):
            val = i
            break
    ret = [((sc+i*26)/ft)%26]

    ft = ogeq[0]
    sc = ogeq[2]-ogeq[1]*ret[0]
    for i in range(0, 26):
        if((sc+i*26)%ft == 0):
            val = i
            break
    ret.append(((sc + i * 26) / ft)%26)
    ret = [int(ret[1]),int(ret[0])]
    return ret
stressTester()
```