# Classic Crypto and Crypto Math

Dana Thomson

September 2021

**Description:**
This is the first part of the cryptography portfolio. This document covers the classic crypto and crypto math sections of the portfolio. The classic crypto section delves into affine ciphers, vigenere ciphers, affine attacks, vigenere attack, hill ciphers, and one time pass ciphers. Each of the cipher type can both encrypt or decrypt a given text. The crypto math section delves into useful algorithms for cryptography. This includes algorithms that are commonly included in math libraries and ones which are not. This document will serve as a user guide and walkthrough.

**Algorithms and Libraries used:**
I used 2 python libraries random and numpy. Random is used for the random.randint function. This function is used for pseudo random number generation. The numpy library is used for dot and matmul functions. The dot product function is used in finding the vigenere key and the matmul is used for hill cipher.

**Function and Program Structure:**
My system has 3 files, main.py, classiccrypto.py, and cryptomath.py. Main contains the user menu and input. It is a starting point for the system. Classiccrypto contains functions for affine encryption/decryption, vigenere encryption/decryption, hill encryption/decryption, one time pass encryption/decryption, frequency analysis, affine brute force attack, affine known plain text attack, and vigenere attack. The Cryptomath file contains function for gcd, egcd, and mod inverse. The cryptomath functions are used throughout the classiccrypto functions as helper functions. I will describe the algorithms in more detail now.

**Affine:** When the user wants to run an affine cipher they have the option to encrypt, decrypt, a brute force attack, or a given plain text attack. The basis of an affine cipher is to use a linear one-to-one function to encrypt the plain text. To decrypt the modular inverse must be found of the alpha value used to encrypt. The modular inverse function is written in the cryptomath file.

$$x \mapsto \alpha x + \beta \pmod{26}.$$

The encryption and decryption functions ask the user for the alpha and beta used for encryption and the text. The encryption function will then take the plain text and alter it using the one-to-one function. The decryption function takes the modular inverse of alpha and then uses the one-to-one function to alter the cipher text back to plain text. I have coded two types of attacks for text encrypted by an affine cipher. The first is a brute force attack, this attack tries all 312 possible alpha/beta combinations and prints a section of decrypted text to a file. The user then can look through the file to find the readable decrypted text and associated alpha/beta values. The second attack type is know plain text, this requires the user to enter two characters of cipher text with their associated two characters of plain text. Then the code uses these two values to find the alpha and beta values using basic algebra.

**Vigenere:** The Vigenere cipher is a shift cipher based on a key. One of the strengths of this cipher is that it doesn't require a one-to-one function. Since the text is shifted based on the key there isn't a set unique cipher text value for each unique plain text value. Another strength of the Vigenere cipher is that the key length is unknown to an attacker, making it harder to decrypt. The encryption and decryption methods ask the user to supply the text and the key. Then to encrypt the key value at a the current index is added to the text value mod 26. To decrypt the key value is subtracted instead of added.

| | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| (plaintext) | *h* | *e* | *r* | *e* | *i* | *s* | *h* | *o* | *w* | *i* | *t* | *w* | *o* | *r* | *k* | *s* |
| (key) | 21 | 4 | 2 | 19 | 14 | 17 | 21 | 4 | 2 | 19 | 14 | 17 | 21 | 4 | 2 | 19 |
| (ciphertext) | *C* | *I* | *T* | *X* | *W* | *J* | *C* | *S* | *Y* | *B* | *H* | *N* | *J* | *V* | *M* | *L* |

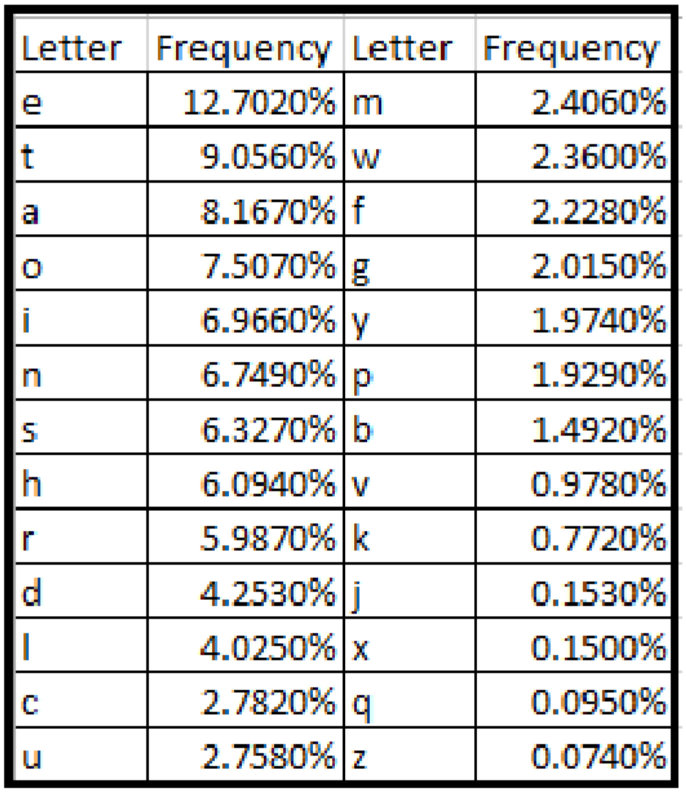
The attack on a Vigenere cipher is dependent on finding the key length. To do this the encrypted text is compared against a shifted version of the encrypted text looking for coincidences. A coincidence occurs when the characters at index i are the same in the original and shifted text. The key length is most likely the number of shifts that gives the highest number of coincidences. In our algorithm the key length is not updated if the new max value is a multiple of the old, for instance 5 and 15. This is due to the cyclic nature of apply the key and the smallest multiple is most likely the true key. Once the key length is found the key value is found through frequency analysis. The cipher text is split into index values based on the key length and a frequency analysis using dot products is done for each letter of the alphabet, the known frequency table is shifted every time to have a new leading letter. By doing this the letter that produced the highest dot product value is most likely the letter of the key at that index. This is repeated for the length of the key to find the complete key value.

**Hill Cipher:** The hill cipher is a fairly simple cipher which uses an invertable matrix to encrypt and decrypt text. To encrypt the text is split into vectors of the same dimension as the matrix, then the vector is multiplied by the matrix mod 26. The result of this multiplication is the cipher text. To decrypt the same process is done with the inverse of the encryption matrix.

$$(0,\ 1,\ 2)\begin{pmatrix}1 & 2 & 3\\ 4 & 5 & 6\\ 11 & 9 & 8\end{pmatrix} \equiv (0,\ 23,\ 22) \pmod{26}.$$

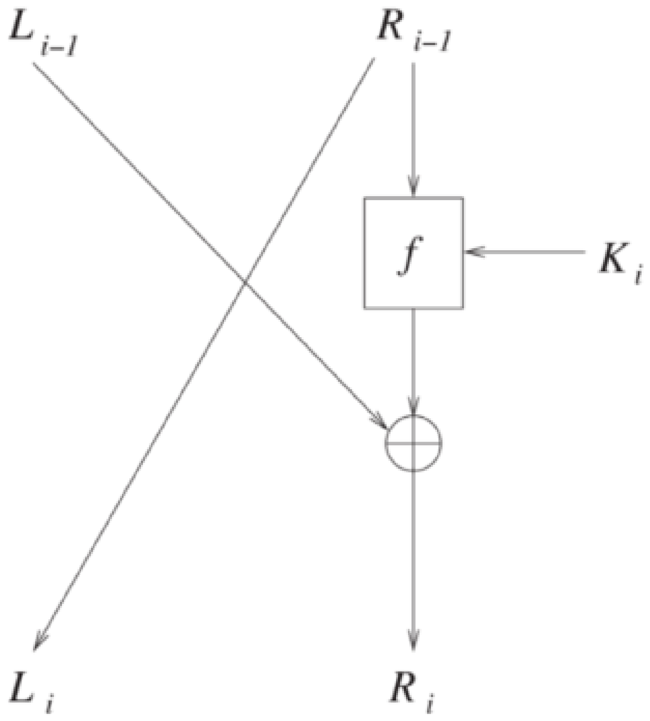
**One Time Pad:** one_time_pad: One time pad is a type of block cipher like the hill cipher. The one time pad though uses a key value to add to the message but the key values are never repeated. This makes it so that even if an attacker where to find the key from a message, that key couldn't decrypt any other messages. My solution for one time pad is implemented using random numbers generated from a user given seed. This allows us to be able to encrypt and decrypt if we know the seed value. The encryption works by adding the random value to the plain text value and taking the result mod 26. The decryption works the same way except subtracting the random value instead of adding.

**Frequency Calculation:** This is a utility function that calculates the frequency of each letter in a given text. The function counts the number of times each letter appears and divides it by the total number of letters in the text to provide a decimal value representing the percentage of the letters. A sample of the table is shown below.



| Letter | Frequency | Letter | Frequency |
|---|---|---|---|
| e | 12.7020% | m | 2.4060% |
| t | 9.0560% | w | 2.3600% |
| a | 8.1670% | f | 2.2280% |
| o | 7.5070% | g | 2.0150% |
| i | 6.9660% | y | 1.9740% |
| n | 6.7490% | p | 1.9290% |
| s | 6.3270% | b | 1.4920% |
| h | 6.0940% | v | 0.9780% |
| r | 5.9870% | k | 0.7720% |
| d | 4.2530% | j | 0.1530% |
| l | 4.0250% | x | 0.1500% |
| c | 2.7820% | q | 0.0950% |
| u | 2.7580% | z | 0.0740% |

**Baby DES:** The baby DES is simplified to take in a 12 bit message, do four rounds of the algorithm and return the encrypted message. The algorithm is pictured below. The basis of the algo is to separate the message into left and right 6 bits. Then the right six bits are expanded to 8 bits and xored with a user given key. This value is then xored with the left six bits to create the next set of right six bits. The next set of left six bits are the previous right six bits. The key is 9 bits but only 8 are used in each round. After each round the key is rotated to the left one bit to create a new key. The final L values and R values are combined to form the encrypted 12 bit message. Decryption works in a similar pattern though the xors happen is slightly different orders to undo the pattern.



**Three Round Cryptanlysis:** The three round cryptanalysis is an attack on the DES system to determine the key. The user needs to know 4 initial and 4 end values for a three round simplified des machine, L1R1's and L4R4's. This attack focuses on weeknesses in xor, specifically that if two values are the same the xor is 0. The analysis uses this by having corresponding R1 and R*1 be the same values. By doing so we can learn what the s boxes xored inputs and outputs are. Once these are found an exhaustive search of all input possibilities will find the pairs which satisfy the input and output xors. If this process is repeated for 2 more sets of values then there should only be one overlap between the pairs of possibilities for right and left side of the key. By finding the left-right pair that is repeated in both possibilities 8 bits of the key are then known. Then the user can try filling in the missing bit with and 1 and a 0 on an input message of all zeros and see which version gives the actual encrypted message. By doing this the user now has the key to a simplified DES. **GCD:** This function takes two integers as input. First it checks that the first int, a, is larger than the second int, b. If it isn't then a and b are swapped. Then while b does not equal zero a becomes b and b becomes a mod b. Finally be return a which is the greatest common divisor.

**Extended GCD:** This function takes two integers a and b as input. First it checks that the first int, a, is larger than the second int, b. If it isn't then a and b are swapped. The egcd finds the gcd and the linear combination of a and b which will generate the gcd. To do this while b is not zero a becomes b, b becomes a mod b, linearX is calculated to be x0 minus x1 times the quotient of a and b, and linearY is calculate y0 minus y1 times the quotient. Then x0, y0 are set to x1, y1 and x1, y1 are set to linearX, linearY. Then a, x0, and y0 are returned. These provide the gcd and the linear combination values for a and b.

**Mod Inverse:** This function calls the extended_gcd function if the gcd is one then there is no possible inverse. If it is not one then the inverse is returned as x mod n. Where n is the mod number.

**Primitive Roots:** This function takes a user given prime number and finds the primitive roots of it. A primitive root is a number whose power generates every non-zero class mod p.

**How to compile:**
To run this program simply run the main.py file. Be sure classiccrypto.py and cryptomath.py are in the same directory as main.py. I have been running the program through the command line using the command 'python main.py'.

**Testing and Verification:**
To test this program I have ran each encryption and decryption algorithm on small then large input texts to verify that they are working correctly. For each attack algorithm I ran them multiple times to determine that they would find the correct answers in different situations. For the crypto math functions I tested agaisnt known values provided in the book and online.