Cryptography Portfolio

Mitchell Anderson

Table of Contents

1. Code Documentation

a. Running the Program

The program was coded using Python 3.7, so it is recommended to run it using this version of Python. To run the program, you must have Python downloaded and installed.

Once you have Python installed, simply navigate to the directory with the code and use the `py` (or possibly `python` depending on your OS) command. For example, to run the affine cipher program, you would enter: `py affine_cipher.py` and then the program will be run. None of the programs require command line input, although `frequency_calc.py` can optionally take input from the command line.

b. Function Documentation

The documentation for each of the functions can be found in the program files themselves. I have written Pydocs for each of the functions in each of the files. The programs are split into separate files which must be run individually. The functionality of each file is self-explanatory based on the name (though you can read the code and documentation if you need more clarity).

2. Algorithm Descriptions

a. Affine Cipher

An affine cipher is a basic cipher using multiplication and a shift to encrypt messages. It uses the following formula to encrypt a message:

$$\alpha x + \beta = c$$

Where x is the value of a letter of the plaintext, c is the corresponding ciphertext letter value, and α and β are the keys. The formula is evaluated mod 26 if you are using only letters.

An example of the cipher is as follows. Plaintext = “hello”, alpha = 3, beta = 11. If a =0, b = 1, and so on, then h = 7. So, to encrypt h, we would calculate the following:

$$3 * 7 + 11 = 32 = 6\ (mod\ 26)$$

The value 6 corresponds to the letter g, so h encrypts to g. Performing the same operation with the rest of the letters yields a ciphertext of gxssb. To decrypt, you must calculate the inverse of alpha (described in the mod inverse section) then use the formula:

$$\alpha^{-1}(c - \beta) = x$$

In this case, $\alpha^{-1} = 9$ so $9(6 - 11) = -45 = -19 = 7\ (mod\ 26)$ which is the value for h. It is important to not your alpha must be coprime with your mod for decryption to work properly. If you are using mod 26, then the following alpha values are acceptable: 1, 3, 5, 7, 9, 11, 15, 17, 19, 21, 23, 25.

b. Vigenère Cipher

The Vigenère cipher is another relatively basic cipher system. This system requires a key word. For example, let's use the word "lies" for our key. This key is then placed in line with the plaintext and repeated as many times as needed to match the length of the plaintext. This is shown below (plaintext is "schooliscool"):

```
s c h o o l i s c o o l
l i e s l i e s l i e s
```

In this case, the key lined up nicely, but if it doesn't the key can simply be cut off at the end of the plaintext. To encrypt, simply add the values of each vertical pair of letters to receive the corresponding ciphertext. In this case, this leads to the ciphertext of: "dklgztmknwsd".

In order to decrypt, the exact same procedure is performed except you subtract the key from the ciphertext, rather than add. This will yield the original plaintext.

c. Frequency Calculation

This algorithm is pretty straightforward. There's really not much of an algorithm to it at all. The program that I wrote simply reads in a file, iterates through it, and counts the number of occurrences of each character in the file.

In terms of how it is implemented more specifically, I perform the following steps. I iterate through the file, character-by-character, and check each character against the keys a Python dictionary. If the character already exists in the dictionary, I increment the corresponding value by one. If the character is not yet in the dictionary, I add it as a key and set the corresponding value to one. In this way, it only has a count for characters which appear in a text. (i.e. if a text has no letter b's in it, then b is never stored in the dictionary, rather than storing it with a value of 0).

d. Affine Cipher Attack

The most effective affine cipher attack is a known plaintext attack. This attack works most of the time as long as you have at least two letters of the plaintext and the corresponding letters of the ciphertext. Using these values, you can create a system of equations to attempt to solve for alpha and beta. For example, suppose you have the plaintext "if"

corresponding to ciphertext “pq”. This means “i” produces “p” and “f” produces “q”. Using this information, you can create the following two equations:

$$8\alpha + \beta = 15$$
$$5\alpha + \beta = 16$$

By subtracting these two equations, you obtain the following result:

$$3\alpha = -1 = 25\ (mod\ 26)$$
$$\alpha = \frac{25}{3} = 17\ (mod\ 26)$$

Thus, we have found that alpha equals 17.

If alpha had been equal to multiple values, then we would simply pick the value that is a valid alpha value (it must be coprime with the mod). In this instance, we only got 17, which is a valid value of alpha. All we need to do to find beta is to plug alpha back in to one of the equations above.

$$8\alpha + \beta = 15$$
$$\beta = 15 - 8\alpha$$
$$\beta = 15 - 8 * 17 = -121 = -17 = 9\ (mod\ 26)$$

So, we have discovered that the key values for this affine cipher are alpha = 17 and beta = 9.

e. Vigenère Cipher Attack

The first step of a Vigenère cipher attack is to determine the key length. This can be decently accurately guessed using the following method. Take the ciphertext string and copy it. Shift the copied string once to the right and compare the two strings with each other (the original and the shifted ciphertexts). If our ciphertext is “hghhg” then the result should look like this:

h g h h g

  h g h h g

    *

You can see that there is one column where both characters match, marked with the asterisk. In this case, we have one such occurrence. We then shift the copied ciphertext string to the right once more and count the occurrences again. We repeat this procedure until the strings can no longer be compared. Here are the results of each shift:

Shift	Occurrences
1	1
2	1

3	2
4	0

As you can see, the greatest number of occurrences happen at shift 3. Therefore, the key length is most likely three.

Now that we've determined the key length, we can determine the key itself. The following is the method which I have used in my code.

First, take the ciphertext and trim it down to only contain every n^{th} character, where n is the key length. In the example used above, that means we would look at only the 1^{st}, 4^{th}, 7^{th}, etc. characters. We count the frequencies of all 26 letters and store the results in a list. In the example above, that sums up to only two h's.

$$frequencies = [0,0, \dots ,2, \dots ,0]$$

We then divide each value by the total number of letters counted. In this case just the two h's, so have the following result:

$$freqDivided = [0,0 \dots ,1, \dots ,0]$$

Typically, you will end with an array with many fractions in it, but the procedure remains the same. Now, you will need one more list, which contains the frequencies of English letters:

$$engFreq = [0.082, 0.015, 0.028, 0.043, 0.127, 0.022, 0.020, 0.061, 0.070, 0.002, 0.008, 0.040, 0.024, 0.067, 0.075, 0.019, 0.001, 0.060, 0.063, 0.091, 0.028, 0.010, 0.023, 0.001, 0.020, 0.001]$$

You must now compute freqDivided * engFreq (dot product) 26 times. Each time it is computed, rotate engFreq once to the right. Keep track of which iteration each dot product is on. The iteration of the largest dot product correlates to the value of the character of the key. For instance, if the largest dot product is the fifth one, then the value of one of the key characters would be 5, which would correlate to "f".

When you calculate this result for the first iteration of character frequencies (as in the 1^{st}, 4^{th}, etc. characters of the ciphertext), then the determined key character would be the first character of the key. This entire process is repeated through the entire the length of the key (in this case, the second iteration would be 2^{nd}, 5^{th}, etc. characters; this would yield the second character of the key). Thus, this process will yield the key. Feel free to play around with the code to help understand the process a bit better.

f. Linear Feedback Shift Register Sequence

An LFSR sequence is a method to generate a key using a basic recurrence relation. For example, we can use the following recurrence relation:

$$x_{m+3} = x_{m+2} + x_m$$

We will also need some initial values for the recurrence relation (specifically, we'll need three because that is the largest variable shift in the recurrence relation – from x_{m+3}):

$$011$$

Given these initial values and recurrence relation, we can generate any length key we want. Say we want a key to encrypt the plaintext:

$$1010101$$

We'll need a key of length three to encrypt this with an LFSR sequence. So, we'll generate the key using the following recurrence relation. The way it works is simple. The relation above essentially states the following:

$$4th\ bit = 3rd\ bit + 1st\ bit$$

So, from our initial values, the 4th bit would be equal to the 3rd bit (1) plus the 1st bit (0), which equals 1. In this way we can expand the key to match the length of the plaintext as follows:

$$0111010$$

Then, to encrypt the plaintext, we simply add the key to the plaintext bit-by-bit:

$$\begin{array}{r} 1010101 \\ \underline{0111010} \\ 1101111 \end{array}$$

Thus, the ciphertext is 1101111. Decryption uses the exact same method, where adding the ciphertext to the key yields the plaintext.

g. GCD (Greatest Common Denominator)

To calculate the GCD of two numbers, you can use the simple Euclidean algorithm. To calculate the GCD you would use the following steps which follow this simple rule:

$$\text{Remainder} -> \text{Divisor} -> \text{Dividend}$$

With an equation on the form of:

$$Dividend = Quotient * Divisor + Remainder$$

Example: gcd(27, 33):

$$\begin{aligned} 33 &= 1 * 27 + 6 \\ 27 &= 4 * 6 + 3 \\ 6 &= 2 * 3 + 0 \end{aligned}$$

This means gcd(27, 33) = 3 because 3 is the first divisor that perfectly divides its dividend.

h. Extended GCD

This is simply an extended version of the GCD above. It gives you solutions to the values x and y from the following equation (correlating to gcd(a, b) = d):

$$ax + by = d$$

To compute extendedGCD(27, 33), we start with the same steps from above:

$$33 = 1 * 27 + 6$$
$$27 = 4 * 6 + 3$$

But rather than continuing to the final step, we rearrange the first two equations to solve for the remainder:

$$6 = 33 - 1 * 27$$
$$3 = 27 - 4 * 6$$

Then, we will replace the 6 in the second equation with the first equation:

$$3 = 27 - 4 * (33 - 1 * 27)$$
$$3 = 27 - 4 * 33 + 4 * 27$$
$$3 = -4 * 33 + 5 * 27$$

Thus, we have found the values x and y. That is, x = -4 and y = 5.

i. Find Mod Inverse

Now that you know how to perform the extended GCD (assuming you've read the previous section), finding the mod inverse is very easy. The extended GCD finds solutions to the equation: $ax + by = d$ where $gcd(a, b) = d$. Suppose you want to find the inverse of a number, n, under mod m. All you need to do is calculate extendedGCD(n, m) and the value of x will be the inverse of n under mod m.

j. Verify Primitive Roots

Another simple algorithm here. A primitive root is any number whose powers will generate all numbers under a mod. For example, 3 is a primitive root under mod 7 because

$$3^1 = 3 \ (mod\ 7)$$
$$3^2 = 9 = 2 \ (mod\ 7)$$
$$3^3 = 27 = 6 \ (mod\ 7)$$
$$3^4 = 4 \ (mod\ 7)$$
$$3^5 = 5 \ (mod\ 7)$$
$$3^6 = 1 \ (mod\ 7)$$

As you can see, the powers of 3 generate all numbers under mod 7. Therefore, 3 is a primitive root under mod 7.

A test to see if an integer can easily be performed by simply iterating through the powers of a number and ensuring that no numbers are missed under a given mod.

k. Miller-Rabin Primality Test

The Miller-Rabin test can determine if a number is prime relatively quickly with a relatively high level of certainty. If given a number, n, then the Miller-Rabin method goes as follows.

Find the largest power of 2, 2^S, that divides n-1 where $n - 1 = 2^S * m$ for some odd m. Compute $b_0 = 2^m \ (mod\ n)$. If $b_0 = \pm 1 \ (mod\ n)$ then n is probably prime. Otherwise, compute the following:

$$b_1 = {b_0}^2 \ (mod\ n)$$
$$b_2 = {b_1}^2 \ (mod\ n)$$

Repeat this process until you either find some $b = \pm 1$ or you reach b_{k-1}. If any b equals 1, then n is composite. If any b equals -1, then n is probably prime. If you reach b_{k-1} and it is equal to -1, then n is prime, otherwise it is composite.

l. Generate Random Prime

I performed this in a very simple way. I simply used the built-in Python random library to generate random numbers. Then, I used the Miller-Rabin primality test to determine if the number was a prime. If it was not, I'd simply regenerate a number and test it, continuing until I found a prime number.

m. Fermat's Factorization Method

Fermat's factorization is built off the idea that an odd integer n can be represented as

$$n = a^2 - b^2$$

If you rewrite the equation, you get:

$$a^2 - n = b^2$$

The method essentially just tries values for a, in hopes that we find a square. Then, one factor of n would be $a - b$.

n. Pollard Rho Factorization Method

This algorithm uses a polynomial (often $g(x) = x^2 + 1$) to generate a pseudo-random sequence of numbers. You set $a = g(a)$ and $b = g(g(b))$. It works out that if $\gcd(a - b, n) \neq 1$ (where n is the number you wish to factorize), then you have found a factor of n. If it is equal to 1, then you can continue to loop the algorithm. If it is not equal 1, but is also equal to n, then the algorithm has failed and you must try again with new starting values for a and b or with a new polynomial.

o. Pollard p-1 Factorization Method

This algorithm stems off Fermat's little theorem that $a^{k(p-1)} \equiv 1 \ (mod\ p)$. The algorithm works as follows. Let $a = 2$, choose some bound B. Compute $b = a^{B!} \ (mod\ n)$. Then, we set $b_j = {b_{j-1}}^j$ and compute $d = \gcd\ (b - 1, n)$. If $1 < d < n$, then d is a factor of n, otherwise, we continue repeating the above algorithm.

p. Wheel Factorization Method

The wheel factorization method is an improvement upon the trial division method. In this algorithm, you start with a small list of numbers, for example [2, 3, 5]. You test n against this list of numbers, then you generate a new list of numbers that extends your first list. This list of numbers will contain only numbers that are coprime with each of the number previously in the list. Then, n is tested against those number as well. And the process is repeated until a factor of n is found.

In the case of an initial list of [2, 3, 5], the next list generated would be [7, 11, 13, 17, 19, 23, 29, 31], which are each coprime with 2, 3, and 5. In this way, the number of divisions required is reduced compared to the trial division method.