Makiya Crochiere

Cryptosystems

Summary

This is a supporting document that is used to elaborate on the cryptosystems that have been covered within the cryptography class. The main sections are classical cryptosystems, basic number theory, data encryption standard, and the RSA algorithm. Within each section there will be breakdowns on all pertinent information.

Classical Cryptosystems

Classical cryptosystems, in today's standards, are extremely easy to break, but provide a good starting point for new developers and show some aspects of how cryptography has evolved over time. The primary ones that were covered were both the Affine Cipher and the Vigenere Cipher; however, there were many other ones and both the ADFGX and Block Ciphers will be touched upon.

Running the Code

Within the same file as this document there will be additional files that contain the source code for the following ciphers. Within each folder there will be a file with the name of main.py. You should be able to run main.py using a python compiler however PyCharm was used for the creation of these programs. Within PyCharm there should be an option to run main.py in the top right of the application and it should look like a green play button. After the program is running you should be able to go through the console for user inputs and to input the information needed for the program to run.

Frequency Calculation

The frequency calculation is a simple program that walks through the provided text and counts the number of each individual character. This can be used as a basis for an attack. Using a Frequency Calculation you can determine what kind of cipher was used. This is because if the cipher has the same frequency as the normal english language it is very likely that the cipher is some kind of substitution cipher. Using the frequency calculation you may be able to map two or more cipher letters to their plain text letters and with that information solve a system of equations to figure out the encryption equations.

The limitation of a frequency calculation is the length of the cipher text. If the cipher text is short then the calculation would not have enough information to come up with a solid estimate for each letter. In this case, the longer the ciphertext the better the result would be.

Affine Cipher

The Affine Cipher uses an equation that looks like E(x) = (ax + b) (mod m), where E(x) is the encrypted letter, a is the multiplier key value, x is the plain text, b is the offset, and where m is the length of the character set. In this iteration within Affine Cipher folder m will be 26, for the letters within the english alphabet, and the user will provide both a and b for the key values. For decryption there is a very similar equation that generally looks like P(x) = (a^-1)(x - b) (mod m), where a^-1 is the modular multiplicative inverse of a.

Encode

Within the program the plain text isn't just simply run through the equation. To make life easier the plain text is first sifted through and any unwanted characters such as periods, commas, and any others are removed. Once the unwanted characters are removed the plain text is then converted to its numerical equivalent so that it can easily run through the equation. After the numerical string has been encoded it is transferred back into its alphabetical equivalent by using ascii numbers to get the correct character.

Decode

The decode is similar to the encode where the alphabetical characters are sorted through and are converted to and from the numerical equivalent to get the resulting plain text. The only major difference is within the equation and with having the modular multiplicative inverse of a instead of just a. There is a section within BASIC NUMBER THEORY that will go over modular inverse in more detail.

Attacks

There are two major attacks within the program. First of which is the brute force attack. The brute force isn't hard since there is a relatively small number of possible combinations of a and b. The basic algorithm is to walk through every combination of a and b, look at the print outs and look for one that is a readable message.

The other attack that was used was by using a frequency analysis to figure out popular letters and finding their cipher text equivalent. For example you know cipher text e goes to plain text i and cipher text k goes to plain text a you can then solve a system of equations to find the key values for both a and b. For this program the user will provide the set of letters and the cipher text since if the cipher text is small the frequency calculation won't yield any real noticeable results.

Multi-alphabet cipher (Vigenere Cipher)

The Vigenere Cipher is a fairly old cipher that uses a keyword inorder to encrypt its message. Within this program there are two functions such as filter_plain_text and build key which are used to support the encode and decode, but have no major significance. The filter_plain_text function makes ture all the text is in uppercase and removes any unwanted characters. Build_key is used to build a string with the length of the text you are either encrypting or decrypting and adding the key on to itself until it matches the same length.

Endcode

The encode function is fairly simple. Since we built the key to the same length as the message we walk through the length of the message and add key[i] to text[i] then take the mod of 26 to get the encrypted cipher text.

Decode

The decode has almost the exact same structure as the encode with one difference. Instead of adding the two we subtract the key value from the text value and take the modulus of the result to get the decrypted message.

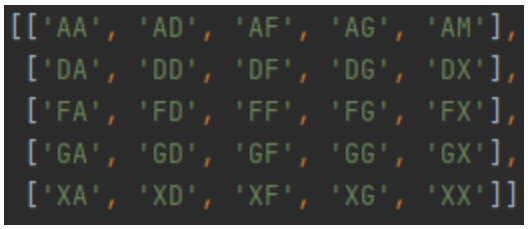
Attack

One of the ways for the Vigenere Cipher to be attacked would be to use the shift method. This is where one can write down the cipher message multiple times below each other shifted to the right once each time. The top characters are then compared to each shifted text below itself and the number of matching characters in each row are recorded. Once the values are recorded, determine which rows have the highest number of coincidences. Determine the number of rows between each of the highest number rows in order to determine the key length. After determining the length between each of the rows, use that number and get every nth character in the message where n is that length determined before. With these letters add all similar letters together then find the frequency of each letter. Compare the frequency of each letter with its equivalent in the English language to get the highest value by shifting the values found compared to the English values left each time. Whichever shift has the highest value is the correct alignment to the english alphabet then that shift value is the first number for the key. This is then repeated for each index value then the key is equal the each value found for each index.

ADFGX Cipher

The ADFGX cipher was created curing the midst of WWI by the german army in order to easily send messages securely over transmit messages over Morse code. This cipher uses two 5x5 matrices and a keyword in order to generate the ciphertext. The first matrix is a key matrix that can be assigned by a user, however for simplicity both it

and the keyword will be predetermined for the program included with this document. The other matrix consists of all the possible combinations of ADFGX. Here are some pictures of both matrices.



Where the first image is the key matrix and the second image is the combination matrix. One slight drawback is that there are 26 characters in the English alphabet with only 25 sections in the matrix. To remedy this j is read in as i and the user will have to determine what makes the most sense in the decrypted message.

Encode

The encoding for the program included is broken down into the following. First the message is read in by each character and is then found on the key table and gets its equivalent on the ADFGX matrix. After the string is converted to the new text it is then fed into a new matrix the size of a keyword. The keyword will then be sorted alphabetically which will then sort the columns that the text was fed into. Once the columns are swapped the text is then read vertically out creating the new cipher text.

Decode

Decode is basically the reverse of encode but the first thing required is to build the sorted version of the matrix from the sorted version of the key. This means the program will first determine each row length then will sort the rows based on the key. The program will then feed in the ciphertext into the sorted matrix and then unsort the matrix. The new cipher text is then mapped backwards through the ADFGX and then to the key matrix. This then outputs the plain text.

Block Cipher

The hill cipher is the block cipher used within the program. The block cipher uses matrix manipulation to both encrypt and decrypt messages. This program uses a nine character keyword to generate the key matrix. However some matrices do not have a det in order to find the inverse to decode so if a keyword creates a matrix without a det then an error message is output to the console. In order to prevent this issue a set key will be used within the program. The message to be encrypted will be broken down into three character blocks hence why it is called a block cipher.

Encode

The encoding function is simple. It takes the message character value vector and multiplies it by the key matrix. The values simply have to be modded by 26 and then returned.

Decode

Decrypting is a little more involved. The first thing required to do is to find the det of the key matrix. If the det is not zero and valid the program will find the inverse of the determinate. The adjugate matrix is then created and multiplied by the inverse of the determinate to get the new key to decrypt the message. The values are then sent to encryption since it's the same thing just with the inverse key matrix.

Basic Number Theory

The following are the functions used in Crypt_Math.py and are the basic and most used mathematical functions within the cryptography programs.

my_gcd(a, b)

The my_gcd function is used to find the greatest common divisor of two numbers. The first part of the function will swap whichever number is greater to allow the math to be easier. The base case will return the greatest common denominator and the function calls itself recursively by making ‘a’ a mod b.

my_extended_gcd(a, b)

The my_extended_gcd function is very similar to the my_gcd function however it also is used to calculate the x and y. The x and y are used for the equation ax+by = gcd(a,b). However, the x and y can also be used for many other functions such as my_mod_inverse.

my_mod_inverse(a, m)

My_mod_inverse is a function used frequently in cryptography. The version within Crypt_math first uses my_extended_gcd to get the information required to determine if there is an inverse. If the gcd is -1 there is no inverse however if it is not -1 it uses the x element and the modulo number to determine the inverse.

my_sqr_roots_find(a, n)

This function is used to find the square roots of a particular number mod m. First it will check to see if a is within the modulus of m and will then walk through all the numbers between 1 and m and check if the number i is a square root of a through mod m by setting i*i mod m equal to a. If ‘i’ is a root it is added to a list of square roots. However, if no roots are found, it returns a message saying such.

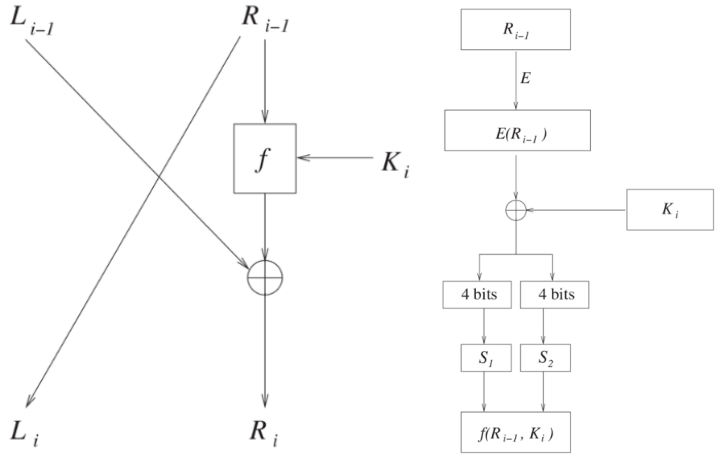
Data Encryption Standard

DES is a block cipher created by a mixture of government powers and IBM in the 1970s. People grew concerned with the NSAs involvement and determined the weakness of the DES was through its S-Boxes and led to a possible backdoor for the parties that developed it.

Simplified DES-type Algorithm, with four rounds and two S-boxes

Since DES can be a bit complicated with all the S-Boxes and different permutations; a good way for one to get their feet wet is with the Simplified DES. This version removes most of the permutations through tables and only has two S-Boxes instead of eight.

The algorithm for creating the components and encrypting / decrypting the message blocks is the following. First, the message must be split into 12 bit blocks and fed into the encryption or decryption. For encryption, once the message is passed in it will be split into both a left and right six bit message and walks through an algorithm. In the algorithm the right bit is expanded by passing through an expanding table to allow it to XORed with the first eight bits of the key for its current round. After the XOR, the result is then fed into the S-Box and splint into two where the first bit in each half represents the row and the remaining three represent the column value for each S-Box. The resultant from the S-Box output will then be XORed with the initial left bit from the message then that value becomes the new right side and the old right becomes the new left side of the message. The key for each of these rounds is generated by using left shifts for each round. Here is a diagram of the proces.



Where the image on the left represents a round and the one on the right represents the f(R) segment in each round. The decryption for the simple DES is very similar however

the key rounds go backwards and the left side of the message goes through all of the f(L) permutations instead.

The Differential Cryptanalysis for Three rounds

The primary goal of the Differential Cryptanalysis was to try to determine the key used for encryption by comparing pairs of plaintext. For the cryptanalysis within the simple DES the program requires two sets of values where the right six bits are the same in all values. To make the process easier the program also requests a key to do encryption however if the user has the resultant messages from after encrypting those can be put in instead.

Once the cipher version of each message is found the output values for the S-Boxes can be derived by XORing R4' and L1'. These values will be used later when creating possible key value pairs. The next task is to determine the input values for the s-boxes. These are found by expanding the L4 and L4* and XORing them to find the expanded L4'. The first four bits of L4' are the inputs for the first s-box and the last four bits of L4' are the inputs for the second s-box.

Now that both the inputs and outputs are found for the s-boxes the next step is to generate the pairs that could possibly be XORed to get the values found from before. Pairs are generated for both s-boxes and then used to determine the key value. However, before that can be done another word must be encrypted and walked through so that the pairs can be generated for that as well. Once the pairs for both words are generated the pairs can then be compared to see which ones are the same in each word.

Once the correct set of pairs are found, eight of the nine bits for the key have been found. Since it is a binary key it can be simple to just guess the final bit and run a word through the algorithm. If the word matches the desired output then the bit is correct and if not all that is needed is to switch that guessed bit to the opposite value and try again. Then the key has been successfully found.

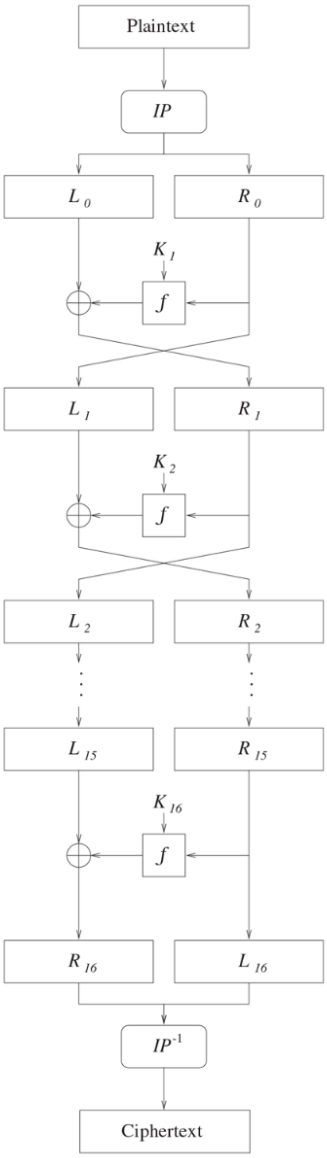
64-bit DES

The regular 64-bit DES is similar to the simple DES but uses more s-boxes, different permutation steps, and has an initial permutation.The version created for this assignment can take any ascii input and encrypt and decrypt the characters. For simplicity a key has been hard coded into the program but can be commented out if the user desires their own key. The user can also choose if they would like to encrypt or decrypt messages which both have the same hard coded key unless commented out.

After submitting a message to encrypt or decrypt the program will convert it to its binary equivalents and then will split the message into 64 bit chunks and process 64 bits of the message at a time. It will also generate all the key rounds before encrypting or

decrypting the program to reduce the computing time greatly. The program will then either encrypt or decrypt the message depending on the user input. The encryption and decryption algorithms are very similar with the decryption basically just being the inverse of the encryption algorithm.

First the 64-bit chunk will go through the initial permutation table. After the initial permutation the message is then split into the left and right half similarly to that in the simple DES. The right bits are then expanded and XORed with the key of the current round of which there are sixteen. The new right bits are then sent through the eight s-boxes to get the original size of 32 bits back by going through the s-box permutation table. This set of right bits are then XORed with the left bits to form the next right set of bits for the next round while the ould set of right boots become the new set of left bits. This process is repeated for the sixteen rounds as depicted in the figure to the right.



After the rounds are complete the left and right sets are combined and are sent through the inverse permutation table. This whole process is repeated until the whole message is encrypted or decrypted resulting in the completion of the DES algorithm.

The RSA algorithm

The RSA algorithm is still widely used today and is known as a very secure method of encryption as long as the primes used are extremely large numbers. Then main algorithm consists of the user choosing two secret prime numbers p and q which are then used to determine n by p*q = n. The user must also choose an e value that satisfies the argument gcd(e, (p-1)(q-1)) = 1, and then the user can compute a d value where d*e = 1 (mod (p-1)(q-1)). Once the user has these values the user can make n and e public while keeping p, q, and d secret. Then anyone can send the user a message by taking m^e (mod n) = c and sending c to the user. The user can then take c and decrypt it by doing c^d (mod n) = m.

During the implementation of RSA the main algorithm was fairly simple to complete and write however the main way to break the message down was to just encrypt each letter separately but anyone could then use a simple substitution method to figure out the message. This program is just proof of concept and should not be used to encrypt any information.

With that disclaimer out of the way the main encryption and decryption algorithms are the same as the ones as described above. The user has the option to either input their own prime numbers and e value or the program can generate them for the user. If the user wants to generate the numbers they will be asked to input the size of the numbers which are in the form of 2^n where n is the number that will be input by the user. The program will then determine all the possible e values that can satisfy the gcd(e, (p-1)(q-1)) = 1. IT will then choose a random e value from the list generated and give that to the user along with a generated d value that is found using mod_inverse of e and (p-1)(q-1). With that the user has all the values necessary to do the RSA algorithm. This program, as stated before, just encrypts each letter separately since there were issues with data being lost early on.

The other opinion the user has when using this program is to be able to factor numbers in one of four ways. The ways of possible factorization are Fermat's, Pollard Rho, Pollard p-1, and Shanks method. Most of the algorithms for these factoring methods were found on their respective wikipedia articles and implemented in the Crypt_Math.py file found in the RSA folder. If the user inputs a prime number this is checked before the program attempts to factor the problem by using the is_prime function. Each of these methods are a different way to factor numbers.

is_prime (n)

The is_prime function is primarily used to determine if a number is prime. It will first check if two is passed in, which is a prime and will return true. If the number is able to divide two or is less than two the function will return false. The function will then take the square root of the number being passed in and add one to it, let's call this n. It will then check all numbers between two and n. If a number is found to divide n it will return false. If a number isn't found then it returns true that the number n is prime.

random_prime(b)

The random_prime function is passed in a variable b. The b value is used to determine the upper and lower bound values for generating the prime where the upper bound is 2^(b+1) - 1 and the lower bound is 2^b - 1. The function will then create a list of primes within the range of numbers. Once the list is generated the function will choose a random prime number from the list and return it.

Fermat's Factorization

Fermat's factorization method is primarily based on the representation of an odd integer as the difference of two squares. Within the function the value is first checked to see if it divides two, and if it does two and the other factor are returned. If not a is found by taking the ceiling square root of the number that was passed into the function. If

a*a== n then a is a factor of n and is returned if not it continues. b1 is then determined by a*a-n which is then used to find b which is the square root of b1. If b*b==b1 the factors are equivalent to a-b and a+b. If b*b!=b1 a is then incremented by one and the process is tried again until a match is found.

Pollard's Rho

Pollard's Rho run time is supposed to be proportional to the square root of the smallest prime factor of the number being passed into it. Like Fermat's and the ones that follow it first checks if it's divisible by two in an attempt to save some time. It will then initialize the values then use the function g(x) = (x^2 -1) mod n to determine x and y. Then the function sees if the gcd(|x-y|, n) is one or not. If it is one the process is repeated with x and y. If g is not one the factors are g and n/g.

Pollard's p-1

Pollard's p-1 is based on the idea to make an exponent a large multiple of p-1, where p is the number being passed in, by making it a number with a lot of prime factors. First the values are initialized and then we make a = a^i mod n. After we see if the gcd(a-1, n) = 1 or not. If it does not equal run the factors are the result and n/result. If it is one i is then incremented by one and the process continues until a factor is found.

Shanks

The main basis for Shank's method is to find integers x and y that satisfy x^2 - y^2 = N where N is the number being sent into the function. The first step is to initialize the values of the arrays the numbers will be stored in. It then follows the main algorithm described in the following picture. As seen in the picture the only time it will return a factor is if p[i] == p[i-1]. The factors would then be equal to gcd(n , p[i]) and n divided by the resultant. If a value is not found i is incremented and the process is tried again.