|  |  |  |
| --- | --- | --- |
| **ipn** | **INSTITUTO POLITÉCNICO NACIONAL**  **ESCUELA SUPERIOR DE CÓMPUTO** |  |

**Cryptography**

**Hill Cipher OM**

Abstract

There are some classical ciphers, one of these, is called Affine. The importance, is to keep your information safe and make it unreadable for unauthorized people, that’s why, this cipher could encrypt and decrypt a message ‘m’ in C language.

**By:**

**Romero Gamarra Joel Mauricio**

Professor:

MSc. NIDIA ASUNCIÓN CORTEZ DUARTE

October 2017

**Imagen que contiene texto, pizarra

Descripción generada con confianza muy alta**

To validate this report, it is necessary to include the corresponding seal

**Index**

Content

[Introduction: 1](#_Toc476313175)

[Literature review: 1](#_Toc476313176)

[Software (libraries, packages, tools): 2](#_Software_(libraries,_packages,)

[Procedure: 3](#_Toc476313178)

[Results 3](#_Toc476313179)

[Discussion: 7](#_Discussion:)

[Conclusions: 7](#_Conclusions:)

[References: 8](#_References:)

[Code 9](#_Code)

# Introduction:

Hill is a block cipher, it means it could encrypt blocks of several letters (or in this case, pixels). A change of one character in the plaintext, change potentially all the characters in the corresponding ciphered text block. Hill Cipher was invented in 1929 by Lester Hill, using for the first-time algebraic methods in cryptography.1

First, we need to understand that, this cipher use matrix operations. The following formula represents HILL Cipher (Encryption):

C = p · K mod n

Where:

* C = Ciphered text
* p = Plaintext
* K = Key
* n = Size of the alphabet

Hill, is just a matrix multiplication, depending on what we want to do, that’s why K is the encryption key and K-1 is the decryption key, finally, we apply modular division to have the result between 0 to (n – 1).

In this case, we are encrypting and decrypting images (formed by pixels), and we know that a pixel is formed by 3 elements: Blue, Green and Red. These, can be between 0 to 255, so, our alphabet is 256.

# Literature review:

I will start explaining each of the Modes of Operation implemented in this practice, the first one is **Electronic Codebook (ECB)** which is represented on Figure 1.

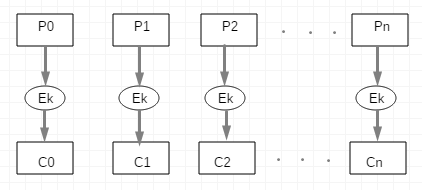


Figure 1. Electronic Codebook Diagram (Encryption)

From Figure 1, we can note the encryption process and then, we can obtain the formula:

C0 = Ek (P0) C1 = Ek (P1) C2 = Ek (P2) … **Cn = Ek (Pn)**

Now, to decrypt the image, we can apply “Dk” both on the left and right sides of the formula:

Cn = Ek (Pn) Dk (Cn) = Dk (Ek (Pn)) **Pn = Dk (Cn)**

Where:

* Pi = Plaintext = Pixels of the original image
* Ci = Ciphers = Pixels of the modified image
* Ek = Encryption Key (Matrix K)
* Dk = Decryption Key (Matrix K-1)

Figure 2, shows the decryption process for **Electronic Codebook Mode (ECB)**.

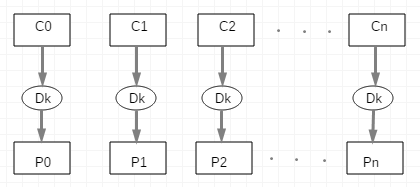


Figure 2. Electronic Codebook Diagram (Decryption)

Something interest of this mode of operation is, it is not safe, we can verify this by looking at Figure 1 and Figure 2, for example, if a Cipher (Ci) is damaged, the others are not, in an image, just a pixel won’t affect too much, because you can appreciate almost all the image in good conditions. If you want to encrypt a message, just a letter will be affected, and the others not. That’s why, this mode of operation is not common neither in images, nor in messages. Computationally, we can parallelize both processes, encryption and decryption because we don’t need to wait for the previous calculation of the pixel to calculate the next.

Now, I continue explaining another Mode of Operation, **Cipher Block Chaining (CBC)**, this mode of operation, must has an initialization vector (IV), also called C0 to start encrypting the information block by block. Also, we add a bit – level operation to encrypt the data, a XOR, converting the value of each BGR value (pixel) to its binary representation, and applying it.1

The result we obtain with XOR, enter to Hill function to multiply by the encryption key and finally obtaining the cipher, then, use the cipher to make the next XOR with the next pixel we reed from the original image like you can see at Figure 3:

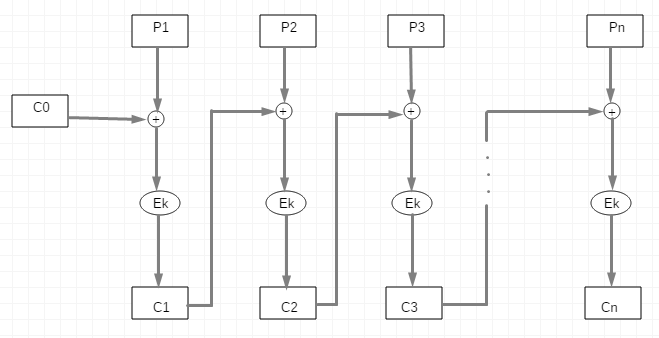


Figure 3. Cipher Block Chaining Diagram (Encryption)

From Figure 3, we can note the encryption process and then, we can obtain the formula:

C1 = Ek (P1 C0) C2 = Ek (P2 C1) C3 = Ek (P3 C2) … **Cn = Ek (Pn Cn-1)**

Now, for the opposite process, we need to clear **Pi:**

Cn = Ek (Pn Cn-1) Dk (Cn) = Dk (Ek (Pn Cn-1)) Dk (Cn) Cn-1 = Pn Cn-1 Cn-1

Finally:

**Pn = Dk (Cn) Cn-1**

Where:

* = XOR

Figure 4, shows the decryption process for **Cipher Block Chaining Mode (CBC)** according to the formula we obtained mathematically.

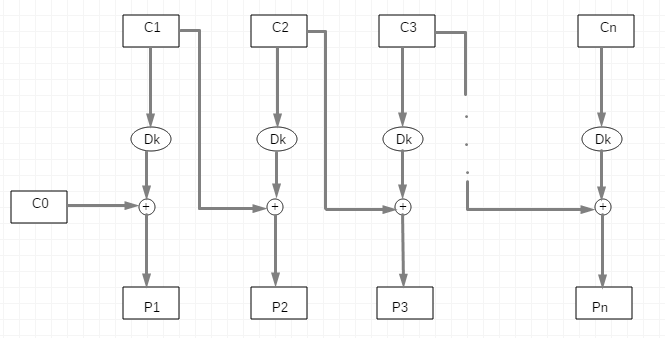


Figure 4. Cipher Block Chaining Diagram (Decryption)

In CBC Mode, we can appreciate in Figure 3 (Encryption Process), that we can’t parallelize because, to obtain the next cipher, we use the previous one, that’s why, we need to calculate the ciphered image pixel by pixel.

Otherwise, in decryption process if we have all the ciphered pixels, we can parallelize the process to make it faster, because, to obtain the next original pixel, we just need the cipher (and we have all of them), not the previous original pixel, that’s why for this process we could use threads to increase the velocity of the image processing.

The next Mode of Operation I will talk about, is called **Output Feedback Mode (OFB)**. As the previous mode (CBC), this mode of operation uses an Initialization Vector we will call C0 to start encrypting / decrypting the image introduced by the user. Also, it needs a bit – level operation (XOR), Figure 5 shows in detail in which vectors we need to apply XOR Operation.

Imagen que contiene texto, interior, pared, pizarra

Descripción generada con confianza muy alta

Figure 5. Output Feedback Diagram (Encryption)

From Figure 5, we can note the encryption process and then, we can obtain the formula:

C1 = Ek (C0) P1 C2 = Ek (Ek (C0)) P2  C3 = Ek (Ek (Ek (C0))) P3

Finally:

**Cn = Ekn (C0) Pn**

Now, for the opposite process, we need to clear **Pi:**

Cn = Ekn (C0) Pn Cn Ekn (C0) = Ekn (C0) Pn Ekn (C0)

Cn Ekn (C0) = Pn

Finally:

**Pn =** Cn Ekn (C0)

Now, we obtain the decryption process mathematically, is important to observe that for **decryption**, we don’t use **Dk**, we use **Ek** again and always applied n times (number of pixels) to C0 (The initialization vector introduced by the user). This mode of operation is not safe and I will explain why in a little while. But first, let’s see on Figure 6 the decryption process for OFB Mode.

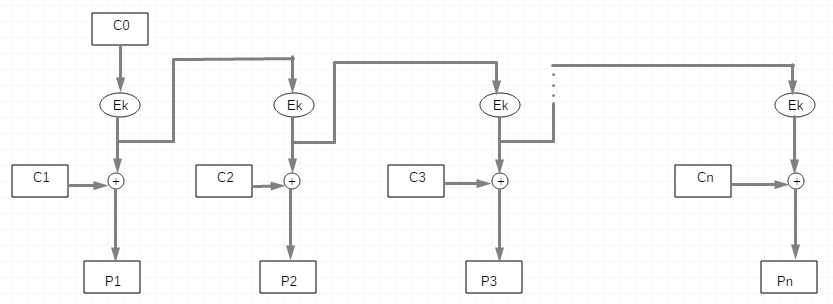


Figure 6. Output Feedback Diagram (Decryption)

The last mode of operation implemented in this practice, is called **Cipher Feedback Mode (CFB)**, this cipher needs an initialization vector to start the processes such as the others (except for Electronic Codebook), the bit – level XOR Operation is presented again to encrypt and decrypt the images.

In figure 7, the decryption process of **CFB** is showed.

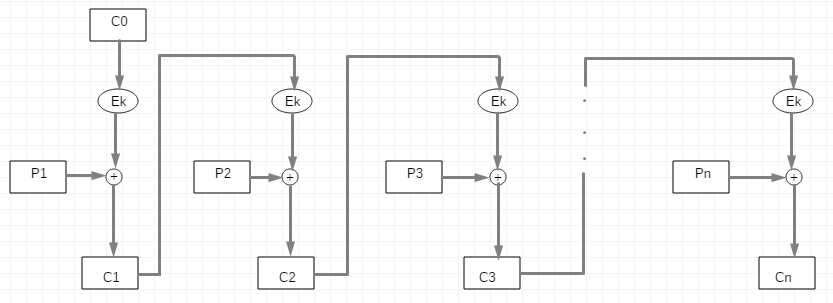


Figure 7. Cipher Feedback Diagram (Encryption)

From Figure 7, we can note the encryption process and then, we can obtain the formula:

C1 = Ek (C0) P1 C2 = Ek (C1) P2 C3 = Ek (C2) P3 … **Cn = Ek (Cn-1) Pn**

Now, for the opposite process, we need to clear **Pi:**

Cn = Ek (Cn-1) Pn Cn Ek (Cn-1) = Ek (Cn-1) Pn  Ek (Cn-1)Cn Ek (Cn-1) = Pn

Finally:

**Pn = Cn Ek (Cn-1)**

As you can see, this time we don’t use the Decryption Matrix again (just as OFB). Figure 8 shows the decryption process we obtained previously Mathematically.

|  |  |  |
| --- | --- | --- |
|  | **Heart** | **Landscape** |
| **Plaintext** |  |  |
| **Electronic Codebook** |  |  |
| **Cipher Block Chaining** |  |  |
| **Output Feedback** |  |  |
| **Cipher Feedback** |  |  |

Table 1. Comparation of implemented Modes of Operation in 2 images

AQUI EXPLICAS LA TABLA E WE

Either alpha or beta, they must have a value between 1 and alphabet size (in this case, English alphabet has 26 letters). In addition, alpha and the alphabet size must be prime to each other to obtain a great result encrypting the message, this means, that the greatest common divisor (GCD) between alpha and 26 should be 1 to guarantee they’re prime each other.

Affine cipher consists first, at giving to each letter of the alphabet a value, for example a = 0, b = 1, c = 2, …, z = 25. Then, the formula for encrypt a message with Affine cipher, is the following:

) mod 26

It means, that each letter of the message, we need to multiply by alpha’s values, then add beta’s values and finally, applying module alphabet’s size (in this case is 26), it gives us the encrypted message (in the successive, with capital letters) and we can decrypt it by founding the inverse additive and the inverse multiplicative for alpha and beta. The formula for decrypt a message is the following:

Inverse additive is too simple, we only need to know that 26 mod 26 = 0, so, what we need to do next is founding a number that in addition to β is 26. However, the multiplicative inverse, is a little more complicated to calculate, but not too much, we only need to remember that α·α-1 mod 26 = 1.

One of the most useful algorithms in this practice is Euclides’s algorithm2, that consists on founding de greatest common divisor between 2 numbers in a fast way. Computationally, we can apply modular division between both numbers, and then use a temporal variable to save the previous number, breaking the cycle when the result of the modular division is zero and then the result will be the previous number before that zero. It is too useful, due to its spatial complexity and temporal complexity, reducing execution time, used memory and useless operations, giving the result faster and permitting creating new applications for the algorithm based on it.2

# Software (libraries, packages, tools):

**Libraries:3**

* Stdio.h: Used for the following functions:
  + int printf(const char \*format, ...)
  + int scanf(const char \*format, ...)
  + FILE \*fopen(const char \*filename, const char \*mode)
  + int fgetc(FILE \*stream)
  + int feof(FILE \*stream)
  + int fprintf(FILE \*stream, const char \*format, ...)
  + int fclose(FILE \*stream)
* Stdlib.h: Used for the following functions:
  + void \*malloc(size\_t size)
  + void free(void \*ptr)
  + void exit(int status)
  + int system(const char \*string)
* String.h: Used for the following functions:
  + size\_t strlen(const char \*str)
* Functions.h (Own): Used the following functions, making use of the above functions:
  + void encrypt (int alpha, int beta)
  + char \* readMessage ()
  + void writeCiphertext (char \* ciphertext)
  + void decrypt (int alpha, int beta)
  + char \* readCiphertext ()
  + int multiplicativeInverse (int alpha)
  + void menu ()
  + void validateNumbers (int alpha, int beta)
  + int gcd (int alpha, int alphabet) Euclides’s algorithm

**Tools:**

* Sublime text 3
* Bizagi Modeler

# Procedure:

Figure 8. Affine Cipher Flowchart

Affine cipher works the following way:

* Assign a value to each letter. a = 0, b = 1, c = 2, d = 3, e = 4, …, y = 24, z = 25.
* Multiply each value by alpha (given by the user), for example, alpha = 5. c = 2(3) = 6.
* Add to the new value, the value of beta, for example, beta = 14. c = 6 + 14 = 20.
* Apply modular division to thee new value with alphabet’s size. c = 20%26 = 20.
* The new letter is given by this last operation, in this example with those parameters, the letter encrypted that corresponds to the original letter ‘c’ is letter ‘U’ in capital letter only to distinguish the encrypted and original messages.

What I did, is re-create the steps explained above, with some functions in C language and then substitute the letter with the new one and write it on a message, with the purpose to obtain the decrypted message (with the correct key) at any moment the user wants, reading the encrypted message and introducing the values for alpha and beta to discover the ciphered text.

# Results

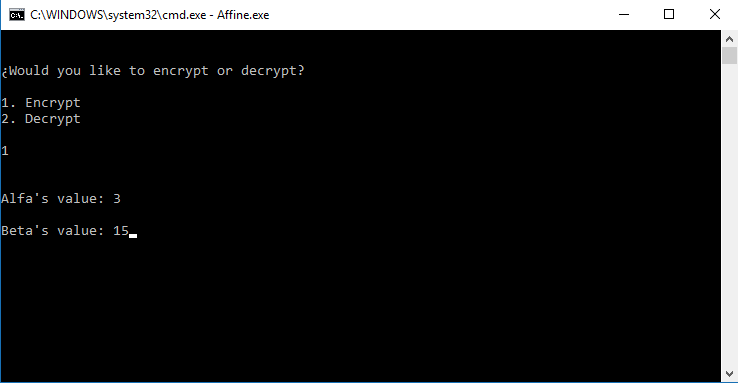


Figure 9. Main menu of the program selecting option 1 (Encrypt).

In figure 2, you could see the main menu, selecting first the option if you want to encrypt or decrypt a message from a file, then, you could select the values that alpha and beta are going to have through the execution of the program to obtain the cipher/decipher message.

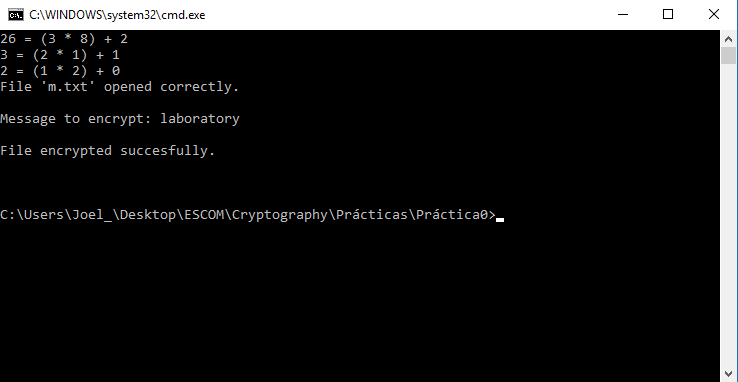


Figure 10. Execution of encryption showing the message, a text if the ciphered was successful and the steps corresponding to the Euclidean algorithm.

After we receive the message that the encryption was successful, we’ll proceed to review the text to check if it’s correctly ciphered (Figure 4) and compared to a web page4 with the same values and the same message (Figure 5).

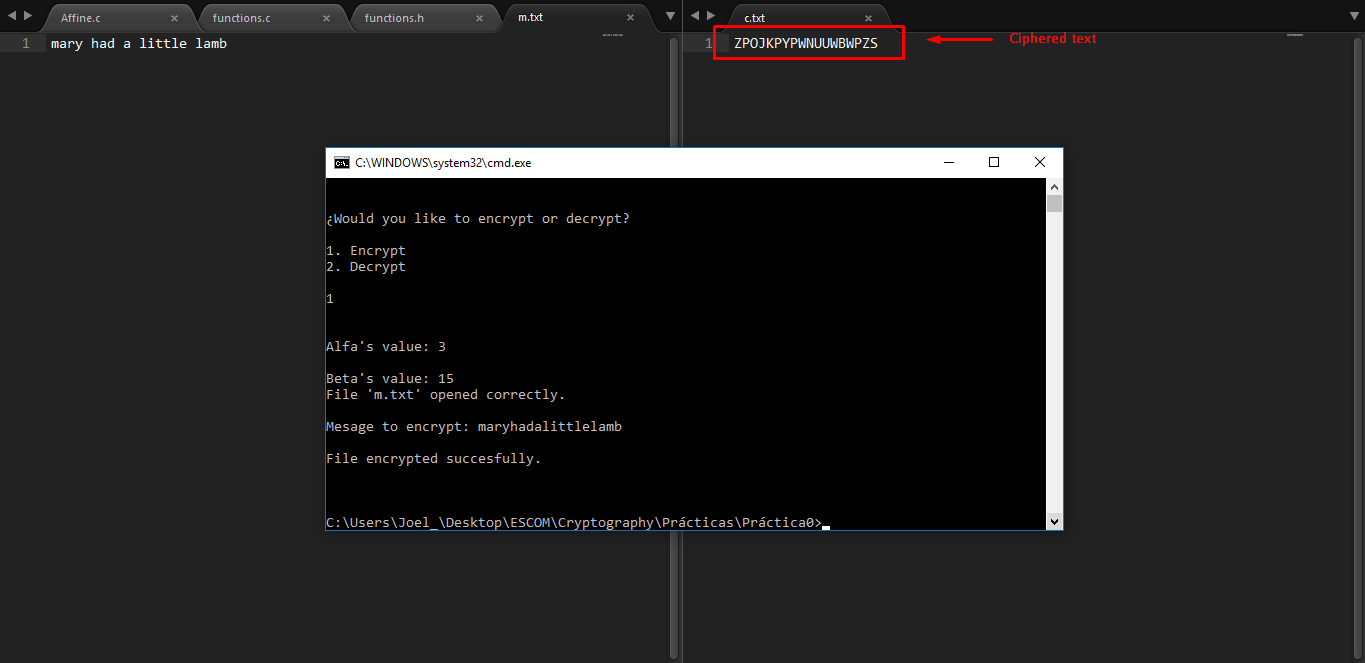


Figure 11. Demonstration of the encrypted message (right) with the original one (left).

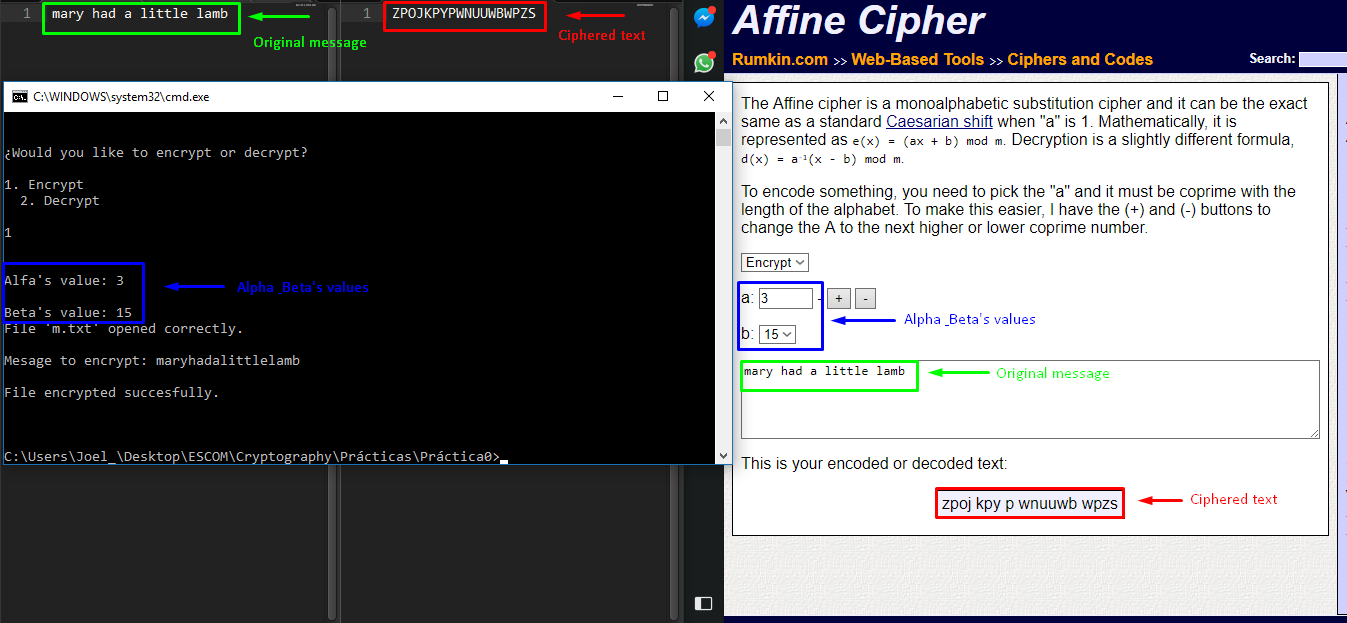


Figure 12. Comparison between a web page and my own program with the same values and messages.

Now, we proceed to verify the decryption option, first with wrong values and then with the correct ones to see if our decryption function described on Literature Review is correctly implemented.

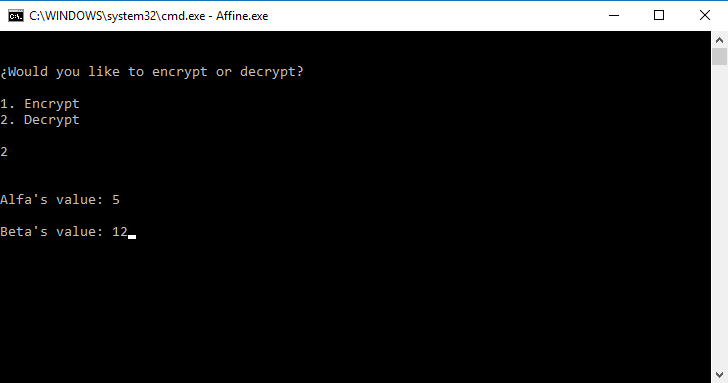


Figure 13. Main menu of the program selecting option 2 (Decrypt).

We can observe, that the value of alpha and beta are wrong (Figure 6), so, what must happen is that the message deciphered is incorrect, because the function won’t be the correct one to decrypt the message correctly.

In the following image (Figure 7), we can see de “decrypted” message, but it’s unreadable because it has no sense, and the message we were expecting is “mary had a little lamb”, like in the original message written in the text file.

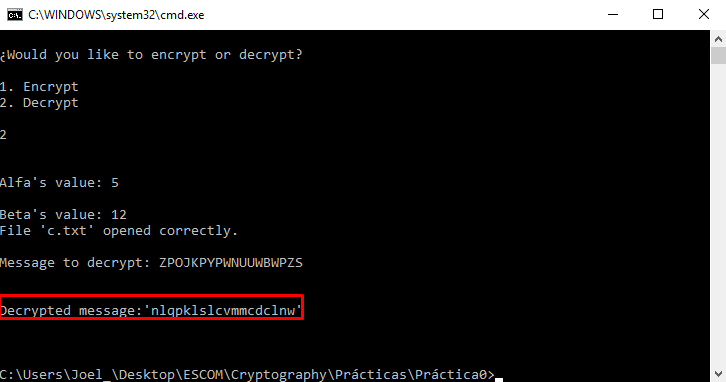


Figure 14. Wrong decrypted message due to alpha's and beta's values.

As we said before, the decrypted message is incorrect and make no sense, now, we’ll try with the correct combination of values and see if it’s correct the algorithm implemented in the “Code” Section.

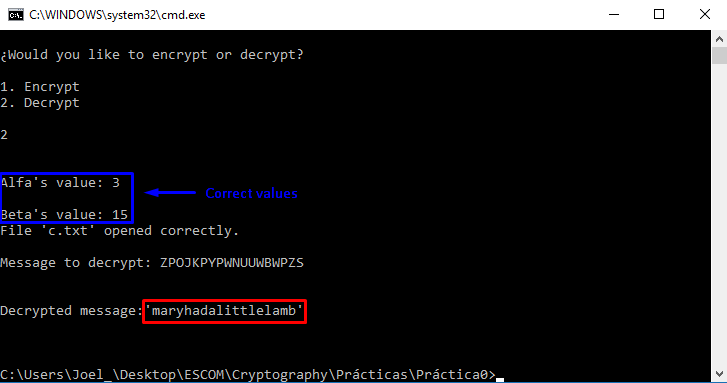


Figure 15. Correct decrypted message with the correct combination of alpha's and beta's values.

In the Figure 8 we can see that the correct combination (**key**) of the values on alpha and beta, gives us a readable and a coherent message, that was the original we tried in the text file showed before (Figure 5).

As you can see, the results are satisfactory even if the values are wrong because the message is protected by the key and the algorithm I implemented on the program, in the next section I will explain some little “errors” in the execution time and what’s the importance of this practice nowadays (as I already said, it’s a classical cipher, too weak in these days).

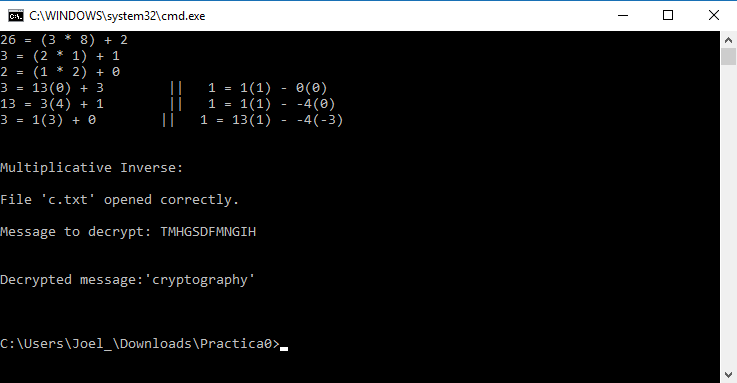
Extended Euclidean algorithm5 is implemented in Figure 9, is used to know the multiplicative inverse of alpha, only if greatest common divisor is 1, because it means that alpha and the alphabet size are prime each other.

Figure 16. Implementation of Extended Euclidean Algorithm to find the multiplicative inverse of alpha

# Discussion:

It’s clear that Affine Cipher could be a great cipher (could think that), because there are 252 possible combinations with alpha and beta, but still being a classical substitution cipher.

However, the importance of encrypting a message is hire the information from a person not allowed to read it, and this (Affine), accomplish the objective, or at least, did it some years ago.

The results obtained in the previous section was correct, because the program read a file with the message you want to encrypt and then ask you for alpha’s and beta’s values to encrypt your message and write it on another text file to decrypt it at any moment. One important application is protecting personnel information like address, salary, credit card, etc.

# Conclusions:

I learned that making a cipher is not too easy (but this practice was kind of), and is important to protect information always, even if you dedicate yourself to something else like singing, dancing, teaching, etc. And one of the needed services is provided by modern cryptography with the new algorithms that have been developed since at least, 40 years ago. For example, you could send your address and/or your phone number by one social network, but what happen if there’s a guy who’s taking information to kidnap your child.

One problem here, is that the program fails when the text to encrypt exceeds 600 characters, and we know that sensitive information has a longer length than that, so is a problem I need to solve immediately to offer my users more security in their personnel information.

Generally, my program could be applied at any alphabet, with some little fixes, but for example I implemented the alphabet size dynamic, so, we could have only a part of it and calculate correctly the ciphered/deciphered text from 2 simple values.

Of course, this program is not perfect, but I tried to optimize the most I could, for example avoiding the pair numbers in the function to obtain the multiplicative inverse because there’s time to the processor we’re wasting, in memory too.

# References:

**[1]** “Block Ciphers - HILL Cipher”, class notes for Cryptography, Department of Engineering in Computer Systems, Escuela Superior de Cómputo, 2017.

**[2]** Franco Martínez Edgardo Adrián, ‘Complejidad de los Algoritmos’, 2017, [Online]. Available: <http://www.eafranco.com/docencia/analisisdealgoritmos/files/ejercicios/02.pdf>. [Accessed: 18 – August – 2017].

**[3]** Tutorialspoint, ‘The C Standard Library’, 2012, [Online], Available: <https://www.tutorialspoint.com/c_standard_library/>**.** [Accessed: 27 – August – 2017].

**[4]** Akins Tyler, ‘Affine Cipher’, 2008, [Online]. Available: <http://rumkin.com/tools/cipher/affine.php>. [Accessed: 27 – August – 2017].

**[5]** Andrés Esquivel, ‘IMPLEMENTACIÓN DEL ALGORITMO DE EUCLIDES EXTENDIDO EN JAVA’, 2015, [Online]. Available: <http://blog.andresed.me/2015/06/implementacion-de-euclides-extendido-en.html>. [Accessed: 1 – September – 2017].

# Code

**Affine.c**

#include <stdio.h>

#include <stdlib.h>

#include "functions.c"

**int** main (**int** argc, **char** **const** \*argv[])

{

menu ();

**return** 0;

}

**Functions.h**

//Encryption functions

**void** encrypt (**int** alpha, **int** beta);

**char** \* readMessage ();

**void** writeCiphertext (**char** \* ciphertext);

//Decryption functions

**void** decrypt (**int** alpha, **int** beta);

**char** \* readCiphertext ();

**int** alg\_euc\_ext(**int** n1,**int** n2);

//Shared encryption/decryption functions

**void** menu ();

**void** validateNumbers (**int** alpha, **int** beta);

**int** gcd (**int** alpha, **int** alphabet);

**Functions.c**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#define ALPHABET\_SIZE 26

#include "functions.h"

**int** i, value;

//ENCRYPTION FUNCTIONS

**void** encrypt (**int** alpha, **int** beta)

{

**char** \* message, \* ciphertext = (**char** \*) malloc (**sizeof** (**char**));

validateNumbers (alpha, beta);

message = readMessage (); //Receiving the message to encrypt and save it in a dinamic array

printf("\nMesage to encrypt: %s\n\n", message); //Print the message to know it is correct

**for** (i = 0; i < strlen (message); i ++)

{

value = message [i] - 97;

value \*= alpha; //Multiplying each letter by alpha

value += beta; //Adding beta

value %= ALPHABET\_SIZE; //We get the value module alphabet's size

ciphertext [i] = value + 65; //We save each encrypted letter in a dinamic array

}

ciphertext [i] = '\0'; //We add null character to avoid trash on the array

writeCiphertext (ciphertext); //Finally, we write the message in a file

}

**char** \* readMessage ()

{

FILE \* message; //Pointer for read the file

**char** c, \* msgToEncrypt = (**char** \*) malloc (**sizeof** (**char**));

i = 0;

message = fopen ("m.txt", "r"); //Opening the file in reading mode

**if** (message == NULL)

printf("Error while opening file: 'm.txt'.\n");

**else**

printf("File 'm.txt' opened correctly.\n");

c = fgetc (message); //Reading the first character

**while** (c != EOF) //While it's not the end of the file

{

**if** (c != 32 && c != '\n') //If is a space or a line break, we don't add it

{

**if** ((c >= 'a' && c <= 'z'))

msgToEncrypt [i ++] = c; //We save small letters on our dinamic array

**else**

{ //If it's another character, we end the program

printf("Error, the file to encrypt has to has small letters only.\n");

exit (0);

}

}

c = fgetc (message); //Reading the next character

}

msgToEncrypt [i] = '\0'; //We add null character to avoid trash on the array

fclose (message); //We close the file after reading it

**return** msgToEncrypt; //Return the original message to encrypt from file

}

**void** writeCiphertext (**char** \* ciphertext)

{

FILE \* encryptedMessage; //Pointer for writing in the file

encryptedMessage = fopen ("c.txt", "w"); //Opening the file in writing/overwriting mode

**if** (encryptedMessage == NULL)

printf("Error while creating file: 'c.txt'\n");

fprintf(encryptedMessage, "%s", ciphertext); //Writing the encryped message on the file as a string

printf("File encrypted succesfully.\n\n\n");

fclose (encryptedMessage); //We close the file after reading it

}

//DECRYPTION FUNCTIONS

**void** decrypt (**int** alpha, **int** beta)

{

**char** \* ciphertext, \* plainText = (**char** \*) malloc (**sizeof** (**char**));

validateNumbers (alpha, beta);

**int** inverse;

inverse = alg\_euc\_ext(alpha,beta); //Obtaining the multiplicative inverse for alpha

ciphertext = readCiphertext (); //Receiving the message to decrypt and save it in a dinamic array

printf("\nMessage to decrypt: %s\n\n", ciphertext); //Print the message to know it is correct

**for** (i = 0; i < strlen (ciphertext); i ++)

{

value = ciphertext [i] - 65;

value \*= inverse; //Multiplying each letter by multiplicative inverse of alpha

value += (ALPHABET\_SIZE - beta) \* inverse; //Adding the aditive inverse of beta

value %= ALPHABET\_SIZE; //We get the value module alphabet's size

plainText [i] = value + 97; //We save each decrypted letter in a dinamic array

}

plainText [i] = '\0'; //We add null character to avoid trash on the array

printf("\nDecrypted message:'%s'\n\n\n", plainText); //Finally, we show the original message to the user

}

**char** \* readCiphertext ()

{

FILE \* message; //Pointer for read the file

**char** c, \* msgToDecrypt = (**char** \*) malloc (**sizeof** (**char**));

i = 0;

message = fopen ("c.txt", "r"); //Opening the file in reading mode

**if** (message == NULL)

printf("Error while opening file: 'c.txt'.\n");

**else**

printf("File 'c.txt' opened correctly.\n");

c = fgetc (message); //Reading the first character

**while** (c != EOF) //While it's not the end of the file

{

**if** (c != 32 && c != '\n') //If is a space or a line break, we don't add it

{

**if** ((c >= 'A' && c <= 'Z'))

msgToDecrypt [i ++] = c; //We save capital letters on our dinamic array

**else**

{ //If it's another character, we end the program

printf("Error, the file to decrypt has been modified.\n");

exit (0);

}

}

c = fgetc (message); //Reading the next character

}

msgToDecrypt [i] = '\0'; //We add null character to avoid trash on the array

fclose (message); //We close the file after reading it

**return** msgToDecrypt; //Return the encrypted message to decrypt from file

}

**int** alg\_euc\_ext (**int** n1,**int** n2)

{

**int** array[3],x=0,y=0,d=0,x2 = 1,x1 = 0,y2 = 0,y1 = 1,q = 0, r = 0;

**int** flag=1;

**int** aux;

**int** in=n1;

**if**(n2==0){

array[0]=n1;

array[1]=1;

array[2]=0;

}

**else**{

**while**(n2>0){

q = (n1/n2);

r = n1 - q\*n2;

x = x2-q\*x1;

y = y2 - q\*y1;

n1 = n2;

n2 = r;

x2 = x1;

x1 = x;

y2 = y1;

y1 = y;

**if**(flag%2 != 0){

printf("%d = %d(%d) + %d || 1 = %d(%d) - %d(%d) \n",n1\*q+r,n1,q,r,x1,y2,x2,y1 );

}**else**{

printf("%d = %d(%d) + %d || 1 = %d(%d) - %d(%d) \n",n1\*q+r,n1,q,r,x2,y1,x1,y2 );

}

flag++;

}

array[0] = n1;

array[1] = x2;

array[2] = y2;

}

aux = multiplicativeInverse(in);

**return** aux;

}

//SHARED ENCRYPTION/DECRYPTION FUNCTIONS

**void** menu ()

{

**int** option, alpha, beta;

system ("cls");

printf("\n\n%cWould you like to encrypt or decrypt?\n\n", 168);

printf("1. Encrypt\n2. Decrypt\n\n");

scanf ("%d", &option);

printf("\n\nAlfa's value: ");

scanf ("%d", &alpha);

printf("\nBeta's value: ");

scanf ("%d", &beta);

system ("cls");

**if** (option == 1)

encrypt (alpha, beta); //If option 1, we encrypt the message

**else** **if** (option == 2)

decrypt (alpha, beta); //If option 2, we decrypt the message

**else**

menu ();

}

**void** validateNumbers (**int** alpha, **int** beta)

{

**if** (alpha == 1)

{

**if** (beta == ALPHABET\_SIZE)

{

printf("Error, the text won't encrypt/decrypt correctly due to alpha's and beta's values.\n\nAlfa: %d \t Beta: %d.\n\n", alpha, beta);

menu ();

}

}**else** **if** (alpha <= 0) //If alpha's value is less than zero, we return to the menu

{

printf("Error, alpha's value must be between 1 and alphabet's size.\n");

menu ();

}

**if** (beta <= 0 || beta > ALPHABET\_SIZE) //If beta's value is bigger than alphabet size, we return to the menu

{

printf("Error, beta's value must be between 1 and alphabet's size.\n");

menu ();

}

**if** (gcd (alpha, ALPHABET\_SIZE) != 1) //We calculate the greatest common divisor

{

printf("Error, the text won't encrypt/decrypt correctly due to alpha's value.\n");

menu ();

}

}

**int** gcd (**int** alpha, **int** alphabet) //Implementation of Euclides algorithm to obtain the greatest common divisor

{

**int** temp, inverse, x;

**if** (alpha > alphabet)

{

temp = alphabet;

alphabet = alpha;

alpha = temp;

}

**if** (alpha != 0) //While alpha's value is bigger than zero

{

printf ("%d = (%d \* %d) + %d\n", alphabet, alpha, alphabet / alpha, alphabet - (alphabet/alpha) \* alpha);

gcd (alpha, alphabet % alpha);

}**else**

{

**return** alphabet;

}

}