Cryptography, Sem: 2020-2, 3CM9, Laboratorio 1, 03/Febrero/2020

Laboratorio 1: Classical Cryptography

Torres Olivera Karla Paola

Escuela Superior de Cómputo
Instituto Politécnico Nacional, México
kpto997@gmail.com

Resumen: El presente trabajo consiste en la descripción e implementación en C++ de algoritmos de cifrado pertenecientes a criptografía clásica (Vigenère Cipher y Affine Cipher), que a su vez es parte de la criptografía simétrica o de clave secreta. Para esto, se hará uso de aritmética modular.
Palabras Clave: criptografía, cifrado, decifrado, aritmética modular, key.

1 Introducción

Los esquemas criptográficos simétricos también se conocen como esquemas o algoritmos de clave simétrica, clave secreta y clave única. La criptografía simétrica se introduce mejor con un problema fácil de entender: Hay dos usuarios, mejor conocidos como Alice y Bob, que desean comunicarse a través de un canal inseguro. Por ejemplo, Internet, un tramo de aire en el caso de teléfono, o cualquier otro medio de comunicación. El problema empieza cuando, Oscar, mejor llamado como *adversario* tiene acceso no autorizado al canal.

En esta situación, la criptografía simétrica ofrece una solución poderosa: Alice cifra su mensaje x usando un algoritmo simétrico, produciendo el texto cifrado y. Bob lo recibe y lo decifra con la misma llave. El decifrado, es por lo tanto, el proceso inverso de cifrado.

Toda la criptografía desde la antigüedad hasta 1976 se basó exclusivamente en métodos simétricos. Los cifrados simétricos todavía se usan ampliamente, especialmente para el cifrado de datos y la verificación de integridad de los mensajes [1].

Tomando en cuenta lo anterior es importante conocer algoritmos de clave secreta, así como sus características, con la finalidad de poder determinar cual utilizar dependiendo de las necesidades que se tengan. Para esto se desarrollarán ejercicios teóricos haciendo uso de la aritmética modular antes de realizar las implementaciones correspondientes.

2 Conceptos Básicos

2.1 Criptografía clásica

La criptografía clásica consta de problemas y herramientas que incluyen cifrado, distribución de llaves, funciones unidireccionales, entre otras. En esta, existen dos tipos de sistemas criptográficos: los cifrados por sustitución y los cifrados por transposición. En estos últimos, las letras del texto claro se codifican (reordenan) sistemáticamente para que el texto se vuelva intangible. Por ejemplo, la palabra ”software” puede codificarse para leerse como ”fosawter”, es decir, las letras se intercambian entre sí. Mientras que en los sistemas de sustitución las letras en el texto claro se reemplazan sistemáticamente por otras pertenecientes al alfabeto definido previamente por las personas que se desean comunicar [2].

2.1.1 Cifrados por sustitución

Como se mencionó anteriormente en los cifrados clásicos por sustitución las letras del texto claro se reemplazan por otras letras o números o símbolos. En [2] podemos encontrar algunos ejemplos de cifrados pertenecientes a este tipo, los cuales son:

1. Caesar Cipher
2. Mono-Alphabetic Cipher
3. Hill Cipher
4. Play-Fair Cipher
5. Vigenère Cipher
6. One-Time Pad

2.2 Aritmética Modular

Casi todos los algoritmos criptográficos ya sean cifrados simétricos o asimétricos, se basan en la aritmética dentro de un número finito de elementos. La definición de operación módulo encontrada en [1] se muestra en Tabla 1.

Operación Módulo
Sean a, r, m $\in \mathbb{Z}$ *(donde* $\mathbb{Z}$ *es un conjunto de todos los enteros) y m >0. Decimos*

$$a \equiv r \bmod m$$

si m divide a-r.
m es llamado el modulo y r el residuo.

Tabla 1: Definición Operación Módulo

En criptografía el uso de la aritmética modular es esencial debido a que el cálculo de logaritmos discretos y raíces cuadradas *mod m* pueden ser problemas computacionalmente difíciles; además, la aritmética modular utiliza cálculos que se realizan cómodamente en ordenadores, ya que restringe tanto el rango de valores intermedios calculados como el resultado final [3].

2.2.1 Propiedades de la Aritmética Modular

Para entender cómo funcionan los algoritmos de cifrado que hacen uso de la aritmética modular es importante conocer las propiedades aritméticas para la suma y el producto de congruencias, las cuales se muestran en la Tabla 2.

Propiedad	Expresión
Conmutatividad	(a+b) mod n = (b+a) mod n
	(a·b) mod n = (b·a) mod n
Asociatividad	[a+(b+c)] mod n = [(a+b)+c] mod n
	[a·(b·c)] mod n = [(a·b)·c] mod n
Cerradura	a+b $\in \mathbb{Z}_n$
	a·b $\in \mathbb{Z}_n$
Neutro aditivo	(a+0) mod n = a mod n
Neutro multiplicativo	(a·1) mod n = a mod n
Inverso aditivo	(a+(-a)) mod n = 0
Inverso multiplicativo	$(a \cdot a^{-1})$ mod n = 1

Tabla 2: Propiedades de la Aritmética Modular

El inverso multiplicativo existe solo para algunos elementos, pero no para todos y generalmente se emplea el algoritmo de Euclides extendido para obtener su valor. Sin embargo, hay una manera fácil de saber si un elemento dado tiene o no inverso:

Un elemento $a \in \mathbb{Z}$ tiene inverso multiplicativo a^{-1} si y solo si $gcd(a, m) = 1$, donde gcd es el máximo común divisor, es decir, el entero más grande que divide ambos números (a y m). Esta propiedad es de gran importancia en la teoría de números, se dice que: si $gcd(a, m) = 1$, entonces a y m son *relativamente primos o coprimos* [1].

2.2.2 Algoritmo de Euclides extendido

El algoritmo de Euclides nos ayuda a encontrar el *gcd* de dos enteros r_0 y r_1. Sin embargo, existe una extensión de este que nos permite calular inversos modulares, lo cual es de gran importancia en la criptografía. Además de calcular el *gcd*, el algoritmo de Euclides extendido (EEA) calcula una combinación lineal de la forma:

$$gcd(r_0, r_1) = s \cdot r_0 + t \cdot r_1$$

Donde s y t son coeficientes enteros. Esta ecuación a menudo se denomina como *Diophantine equation* [1].

La idea para la implementación de este algoritmo es ejecutar el de Euclides estándar, pero expresando en cada iteración el residuo actual r_i como una combinación lineal de la forma:

$$r_i = s_i \cdot r_0 + t_i \cdot r_1 \tag{2.1}$$

Con esto, en la última iteración se obtendría la siguiente ecuación:

$$r_l = gcd(r_0, r_1) = s_l \cdot r_0 + t_l \cdot r_1 = s \cdot r_0 + t \cdot r_1$$

Esto significa que el último coeficiente s_l es el coeficiente s en la ecuación (2.1) que se busca y también $t_l = t$ [1].

Lo anterior fue una breve introducción al algoritmo de Euclides extendido con la finalidad de saber que datos podemos obtener haciendo uso de este. Sin embargo, para un mejor entendimiento en [1] se presentan ejercicios así como su pseudocódigo.

Para la elaboración de esta práctica se implementará este algoritmo, debido a que será una pieza clave para Affine Cipher el cual se verá más adelante. El pseudocódigo final se muestra en Algorithm 1.

Algorithm 1 Algoritmo de Euclides extendido

Entrada: Dos enteros positivos p y a con $p > a$

Salida : $\gcd(p, a)$ así como s y t tal que $\gcd(p, a) = s \cdot p + t \cdot a$

```
s_0 ← 1
s_1 ← 0
t_0 ← 0
t_1 ← 1
if a == 0 then
    return 0, 1, 0
end
while a ≠ 0 do
    r ← p mod a
    q ← (p − r)/a
    s ← s_0 − q · s_1
    t ← t_0 − q · t_1
    p ← a
    a ← r
    s_0 ← s_1
    s_1 ← s
    t_0 ← t_1
    t_1 ← t
end
return p, s_0, t_0
```

2.3 Vigenère Cipher

El sistema de cifrado de Vigenère (en honor al criptógrafo fránces del mismo nombre) es un sistema polialfabético[1] o de sustitución múltiple. Este tipo de criptosistemas aparecieron para sustituir a los monoalfabéticos o de sustitución simple, basados en el Caesar, debido a que presentaban ciertas debilidades frente al ataque de los criptoanalistas en relación a la **frecuencia de aparición de elementos del alfabeto**.

La llave del cifrado Vigenère es una palabra de k letras, tal que, $k \geq 1$, del alfabeto $\mathbb{Z}_{26}$ ={A, B, C, ..., Z}; esta palabra es un elemento del producto cartesiano $\mathbb{Z}_{26} \cdot \mathbb{Z}_{26} \cdot \mathbb{Z}_{26}$ (k veces).

De esta forma, el mensaje a cifrar en texto claro se descompone en bloques de k elementos - letras - y aplica sucesivamente la llave a cada uno de estos.

Este método se consideraba invulnerable hasta que en el siglo XIX consiguieron decifrar algunos mensajes codificados con este sistema, mediante el estudio de la repetición de bloques de letras.Una mejora de este algoritmo fue introducida por el sistema de Vernam el cual, utiliza una llave aleatoria cuya longitud es igual que a la del mensaje [3].

Tomando en cuenta lo anterior una definición más formal de este algoritmo de cifrado se muestra en la Tabla 3.

Definición Vigenère Cipher
Sea
$P = p_0, p_1, ..., p_{n-1}$ *secuencia de letras del texto claro,*
$K = k_0, k_1, ..., k_{m-1}$ *una llave que consiste en una secuencia de letras tal que,* $m < n$

Cifrado: $C_i = (p_i + k_{i \ mod \ m}) \bmod 26$
Decifrado: $p_i = (C_i - k_{i \ mod \ m}) \bmod 26$

Tabla 3: Definición Vigenère Cipher

Con la finalidad de entender mejor cómo funciona, se realizó un ejercicio el cual se puede observar en la Tabla 4 especificando los pasos que se llevaron a cabo. Es importante recordar que el número asignado a cada una de las letras depende de su localización en el alfabeto (A-Z).

key:	*deceptive*
plaintext:	*wearediscovered*
cipherttext:	ZICVTWQNGRZGVTW

[1]Son aquellos que cifran letras en función de su posición en el texto claro.

M:	w	e	a	r	e	d	i	s	c	o	v	e	r	e	d
	22	4	0	17	4	3	8	18	2	14	21	4	17	4	3
K:	3	4	2	4	15	19	8	21	4	3	4	2	4	15	19
M + K:	25	8	2	21	19	22	16	39	6	17	25	6	21	19	22
(M + K) mod 26:	25	8	2	21	19	22	16	13	6	17	25	6	21	19	22
C:	Z	I	C	V	T	W	Q	N	G	R	Z	G	V	T	W

Tabla 4: Ejemplo Vigenère Cipher

2.4 Affine Cipher

Affine Cipher no comparte la propiedad con Shift Cipher que si se conoce la correspondencia entre una letra del texto claro y una del cifrado, el resto de las correspondencias siguen.

Este algoritmo de cifrado a diferencia de otros, cuenta con una llave compuesta por dos números. Cifra multiplicando el mensaje original por una parte de esta, seguido de la adición de la otra parte de la llave. La definición de Affine Cipher encontrada en [1] se muestra en la Tabla 5.

Definición Affine Cipher
Sea $x,\ y,\ a,\ b \in \mathbb{Z}_{26}$

Cifrado: $e_k(x) = y \equiv a \cdot x + b \bmod 26$
Decifrado: $d_k(y) = x \equiv a^{-1} \cdot (y - b) \bmod 26$

con llave: k=(a,b), con la restricción: gcd(a,26)=1

Tabla 5: Definición Affine Cipher

El decifrado se deriva fácilmente de la función de cifrado:

$$a \cdot x + b \equiv y \bmod 26$$
$$a \cdot x \equiv (y - b) \bmod 26$$
$$x \equiv a^{-1} \cdot (y - b) \bmod 26$$

La restricción $\gcd(a, 26) = 1$ proviene del hecho de que el parámetro a de la llave necesita tener inverso multiplicativo para el decifrado. Como se mencionó en la sección 2.2.1 para que esto se cumpla a y el módulo deben ser relativamente primos. Por lo tanto debe estar en el conjunto [1]:

$$a \in \{1, 3, 5, 7, 9, 11, 15, 17, 19, 21, 23, 25\}$$

Con base en lo anterior se dará un ejemplo del funcionamiento de este algoritmo de cifrado, el cual se visualiza en la Tabla 6.

key:	$E_{(3,2)}(m_i)$
plaintext:	*coronavirus*
cipherttext:	ISBSPCNABKE

M:	C	O	R	O	N	A	V	I	R	U	S
	2	14	17	14	13	0	21	8	17	20	18
$a \cdot m_i + b$:	8	44	53	44	41	2	65	26	53	62	56
$(a \cdot m_i + b)$ mod 26:	8	18	1	18	15	2	13	0	1	10	4
C:	I	S	B	S	P	C	N	A	B	K	E

Tabla 6: Ejemplo Affine Cipher

Por otro lado, cuando se desee decifrar el mensaje se hará uso del algoritmo de Euclides extendido, explicado en la sección 2.2.2 para calcular el inverso de a. A su vez, este también nos ayuda a determinar si el valor a de una llave candidata es válido o no, es decir, si $\gcd(a, \mathbb{Z}_n) = 1$.

3 Experimentación y Resultados

3.1 Punto 1. Vigenère Cipher (cifrado)

Algunos puntos importantes a considerar antes de presentar el código fuente son las variables globales utilizadas: se hizo uso de un `map<char,int>` con la finalidad de almacenar el alfabeto ingresado por el usuario con su valor númerico correspondiente, para esto, se puso como llave cada una de las letras. Además, se declaro un `string` que almacena el nombre del archivo hasta el *".txt"* con el objetivo de poder concatenarle o quitarle las terminaciones de cifrado correspondientes.

Las variables anteriores se declararon globales debido a que la mayoría de los módulos de código accedían a estas.

Para la implementación del algoritmo de Vigenère se tomó en cuenta la definición presentada en la sección 2.3. Se diseñó una función que recibe como parámetros tres apuntadores: uno al mensaje claro, otro a la llave y el último al alfabeto. El código fuente de dicha función se muestra en el Listing 1.

```
void encryptVigenere( string *message, string *key, string *alphabet ){

    int numCharacter, modMessageandKey;
    char messageCipher[ (*message).size() ];

    for( register int i = 0; i < (*message).size(); i++ ){
        numCharacter = ( tableAlphabet.find( (*message)[i] )->second + tableAlphabet.find( (*key)[i%(*key).size()] )->second ) % (*alphabet).size();
        messageCipher[i] = (*alphabet)[numCharacter];
    }

    messageCipher[ (*message).size() + 1 ] = '\0';

    cout<< messageCipher;

    saveFile( messageCipher,".vig", "Encrypt" );

}
```

Listing 1: Vigenère Cipher (cifrado)

En caso de que el usuario seleccione la opción de cifrar un mensaje haciendo uso del algoritmo de Vigenère se le preguntará primeramente si desea generar una llave aleatoria o no. Posteriormente, requerirá que ingrese el nombre de cada uno de los archivos *.txt* que contienen los datos pedidos. Como resultado de lo anterior se creará un archivo nuevo con el mismo nombre del texto claro añadiendo la terminación *.vig*. Para una mejor visualización se muestra un ejemplo en las Figuras 1 y 2.

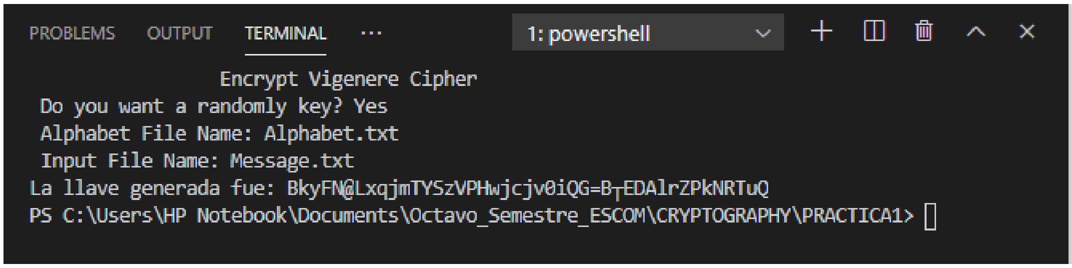

Figura 1: Programa en ejecución

Figura 2: Resultado: Archivo cifrado

3.2 Punto 2. Vigenère Cipher (decifrado)

Una vez teniendo la implementación del cifrado, realizar el decifrado es más sencillo. Debido a que como se vió en la parte teórica es hacer lo inverso. Sin embargo, un aspecto que se tomo a consideración es que existe el caso en que nos resulte un número negativo, por lo que, se implemento una función adicional para invertir el número siguiendo la fórmula vista en clase. Tomando en cuenta esto, el código fuente se visualiza en el Listing 2.

Los parámetros que recibe esta función son iguales a la de cifrado con la diferencia de que ahora *message* hace referencia al mensaje cifrado generado anteriormente.

```
void decryptVigenere( string *message, string *key, string *alphabet ){

    int numCharacter, modMessageandKey;
    char decryptMessage[ (*message).size() ];

    for( register int i = 0; i < (*message).size(); i++ ){
        numCharacter = ( tableAlphabet.find( (*message)[i] )->second - tableAlphabet.find( (*key)[i%(*key).size()] )->second );
        if( numCharacter < 0 )
            negativeNumbMod( &numCharacter, (*alphabet).size() );
        else
            numCharacter = numCharacter % (*alphabet).size();
        decryptMessage[i] = (*alphabet)[numCharacter];
    }

    decryptMessage[ (*message).size() + 1 ] = '\0';

    cout<< decryptMessage;

    saveFile( decryptMessage, ".vig", "Decrypt" );

}
```

Listing 2: Vigenère Cipher (decifrado)

El menú que aparece al seleccionar esta opción es muy parecido al del caso anterior, sin embargo, ahora le pregunta al usuario si desea o no generar una llave aleatoria, es decir, este archivo se vuelve un requisito indispensable. En esta función también se genera un archivo de texto pero ahora al nombre original se le agrega al principio la palabra *Decrypt* el cual contiene el texto claro. En las Figuras 3 y 4 se muestra un ejemplo del programa en ejecución.

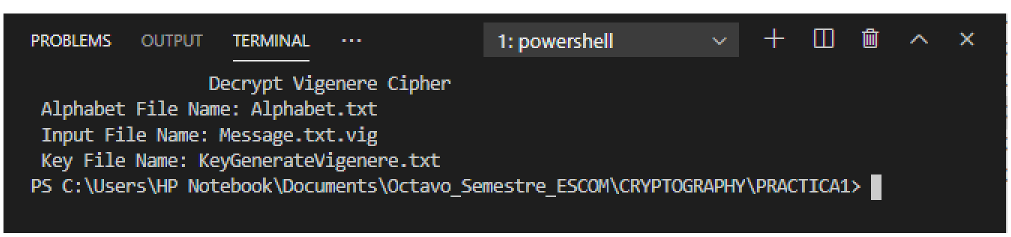

Figura 3: Programa en ejecución

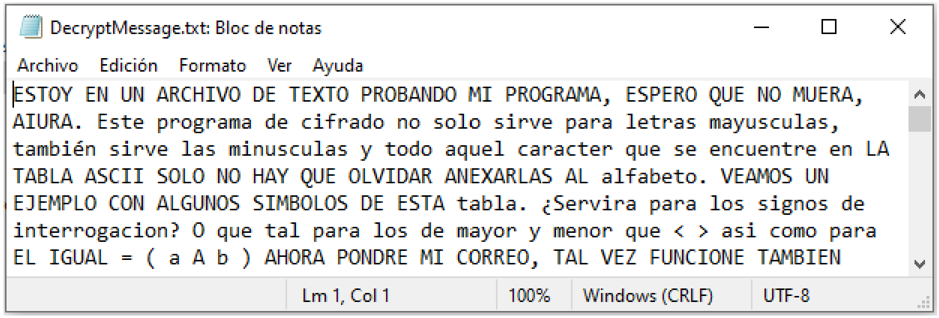

Figura 4: Resultado: Archivo decifrado

3.3 Punto 3. Verificación de llave para Affine Cipher

Recordemos que para determinar si una llave es válida o no para el algoritmo de Affine Cipher se debe cumplir que $\gcd(a, n) = 1$. Para la implementación de este punto se hizo uso de una función de C++ la cual retorna el valor del gcd. Sin embargo, para el punto 4 se programo el algoritmo de Euclides extendido que también ayudaría a resolver este punto. El código fuente se muestra en el Listing 3.

```cpp
int verifyAffineKey( int sizeAlphabet , int a ){
    if( __gcd( a, sizeAlphabet ) == 1 )
        return 1;
    else
        return 0;
}
```

Listing 3: Función gcd

En caso de que la funcion retorne 1 quiere decir que es una llave válida, en caso contrario es inválida. Los parámetros que recibe es n correspondiente al tamaño del alfabeto y a correspondiente al primer valor de la llave. El usuario también puede ingresar el archivo *.txt* del alfabeto en caso de que no sepa su tamaño. En las Figuras 5 y 6 se visualiza el programa en ejecución.

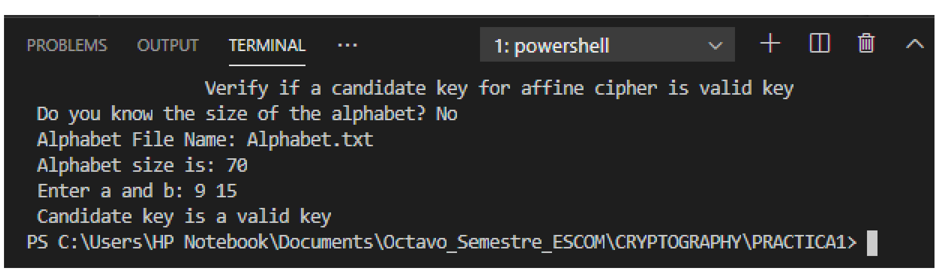

Figura 5: Programa en ejecución

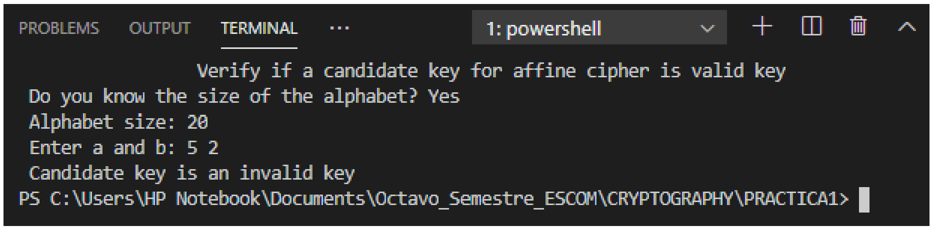

Figura 6: Programa en ejecución

3.4 Punto 4. Calculo de a^{-1} mod n

En la sección 2.2.2 se mostró el pseudocódigo para la implementación del algoritmo de Euclides extendido, recordando que este nos ayuda a calcular el inverso de un número mod n. Por lo que, el código en C++ se muestra en el Listing 4.

```
void extendedEuclid( int valueofA, int valueofN, int *s, int *t, int *gcdResult ){

    int s0 = 1, s1 = 0, t0 = 0, t1 = 1, residue, quotient, sizeAlphabet = valueofN;

    if( valueofA == 0 ){
        *gcdResult = 0;
        *s = 1;
        *t = 0;
        return;
    }

    while( valueofA != 0 ){

        residue = valueofN % valueofA;
        quotient = ( valueofN - residue ) / valueofA;
        *s = s0 - ( quotient * s1 );
        *t = t0 - ( quotient * t1 );
        valueofN = valueofA;
        valueofA = residue;
        s0 = s1;
        s1 = *s;
        t0 = t1;
        t1 = *t;

    }

    if( s0 < 0 )
        negativeNumbMod( &s0, sizeAlphabet );

    if( t0 < 0 )
        negativeNumbMod( &t0, sizeAlphabet );
```

```

    *gcdResult = valueofN;
    *s = s0;
    *t = t0;
    return;

}
```

Listing 4: Algoritmo de Euclides extendido

Es importante recordar que en este caso estamos dando por hecho que la llave ingresada por el usuario es válida, en caso de que se desee modificar esto se debe verificar que el valor de $gcd(a, n) = 1$. En la Figura 7 se observa el programa en ejecución.

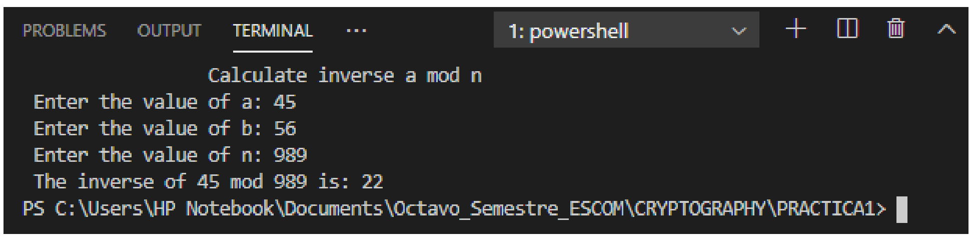

Figura 7: Programa en ejecución

3.5 Punto 5. Affine Cipher (cifrado)

Al igual que en Vigenère Cipher la implementación de este algoritmo se llevo a cabo tomando en cuenta la definición vista en la sección 2.4. El código fuente se muestra en el Listing 5.

```
void encryptAffine( string *message, string *key, string *alphabet ){

    char messageCipher[ (*message).size() ];
    int valueAandMi, finalValue;

    for( register int i = 0; i < (*message).size(); i++ ){
        valueAandMi = tableAlphabet.find( (*key)[0] )->second * tableAlphabet.find( (*message)[i] )->second;
        finalValue = ( valueAandMi + tableAlphabet.find( (*key)[1] )->second ) % (*alphabet).size();
        messageCipher[i] = (*alphabet)[ finalValue ];
    }

    messageCipher[ (*message).size() ] = '\0';

    cout<< messageCipher;

    saveFile( messageCipher, ".aff", "Encrypt" );

}
```

Listing 5: Affine Cipher (cifrado)

Esta función recibe como parámetros tres apuntadores pertenecientes al mensaje que se desea cifrar, la llave y el alfabeto respectivamente. Existe la posibilidad de que el usuario no cuente con el segundo parámetro por lo que le programa es capaz de generar una de manera aleatoria. El programa en ejecución se muestra en las Figuras 8 y 9.

El archivo final se guardará con el mismo nombre del texto claro añadiendo la terminación *.aff* para que sepamos que se trata de un cifrado haciendo uso de Affine Cipher.

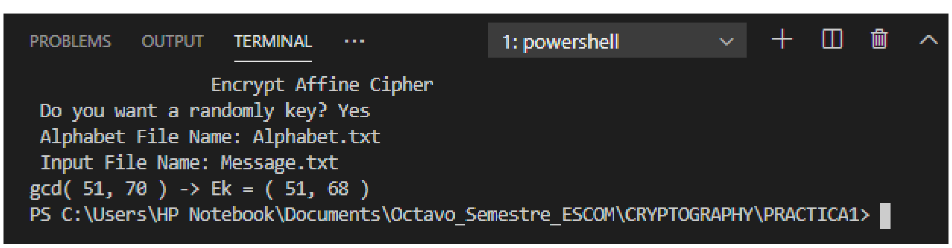

Figura 8: Programa en ejecución

Figura 9: Resultado: Mensaje cifrado

3.6 Punto 6. Affine Cipher (decifrado)

Finalmente tenemos la función que decifra un mensaje haciendo uso del algoritmo Affine Cipher. En este se hizo uso del algoritmo de Euclides extendido visto con anterioridad para calcular el inverso mod n. El código se muestra en el Listing 6

```
void decryptAffine( string *message, string *key, string *
   alphabet ){

    int s, inverseNumber, gcdResult, numCharacter;
    char decryptMessage[ (*message).size() ];

    extendedEuclid( tableAlphabet.find( (*key)[0] )->second, (*
   alphabet).size(), &s, &inverseNumber, &gcdResult );

    for( register int i = 0; i < (*message).size(); i++ ){
        numCharacter = inverseNumber * ( tableAlphabet.find( (*
   message)[i] )->second + (*alphabet).size() - tableAlphabet.
   find( (*key)[1] )->second );
        numCharacter = numCharacter % (*alphabet).size();
        decryptMessage[i] = (*alphabet)[numCharacter];
    }

    decryptMessage[ (*message).size() ] = '\0';

    cout<< decryptMessage;

    saveFile( decryptMessage, ".aff", "Decrypt" );

```

```
}
```

Listing 6: Affine Cipher (decifrado)

El programa en ejecución se muestra en las Figuras 10 y 11.

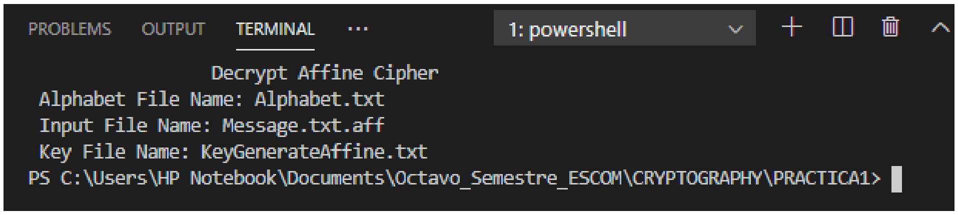

Figura 10: Programa en ejecución

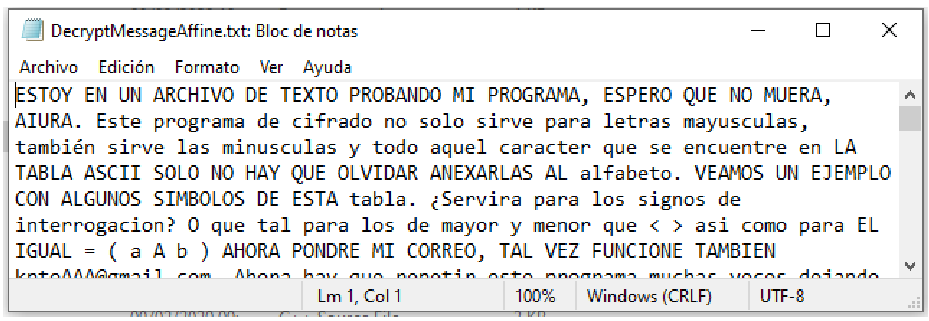

Figura 11: Resultado: Mensaje decifrado

References

[1] C. P. J. Pelzl, *Understanding Cryptography*, 2nd ed. Berlin Heidelberg: Springer-Verlag, 2010.

[2] S. M. Musa, *Network Security and Cryptography.* Sarham M. Musa, 2018.

[3] F. J. M. López, *Informáticos Generalitat Valenciana*, 1st ed. España: Mad, S.L, 2005.