Instituto Politécnico Nacional

Escuela Superior de Cómputo

Session 2

Cryptography

Group: 3CM6

Students:
Nicolás Sayago Abigail
Naranjo Ferrara Guillermo

Teacher:
Díaz Santiago Sandra

February 20, 2019

Contents

1 Topics

1.1 Hill Cipher

This cipher was invented in 1929 by Lester S.Hill. Let m be a positive integer, and define $\mathcal{P} = \mathcal{C} = (\mathbb{Z}_{26})^m$. The idea is to take m linear combinations of the m alphabetic characters in one plaintext element.

Cryptosystem 1.5: *Hill Cipher*

Let $m \geq 2$ be an integer. Let $\mathcal{P} = \mathcal{C} = (\mathbb{Z}_{26})^m$ and let

$$\mathcal{K} = \{m \times m \text{ invertible matrices over } \mathbb{Z}_{26}\}.$$

For a key K, we define

$$e_K(x) = xK$$

and

$$d_K(y) = yK^{-1},$$

where all operations are performed in $\mathbb{Z}_{26}$.

1.1.1 Example

We shall encrypt the plaintext message **retreat now** using the keyphrase back up and a 3 x 3 matrix. The first step is choose a key, we use:

$$\begin{pmatrix} 1 & 0 & 2 \\ 10 & 20 & 15 \\ 0 & 1 & 2 \end{pmatrix}$$

Now we split the plaintext into trigraphs (we are using a 3 x 3 matrix so we need groups of 3 letters), and convert these into column vectors. However, since the plaintext does not go perfectly into the column vectors, we need to use some nulls to make the plaintext the right length. We then convert these into numeric column vectors.

$$\begin{pmatrix} r \\ e \\ t \end{pmatrix} \begin{pmatrix} r \\ e \\ a \end{pmatrix} \begin{pmatrix} t \\ n \\ o \end{pmatrix} \begin{pmatrix} w \\ x \\ x \end{pmatrix}$$

The plaintext split into trigraphs and written in column vectors. Note the nulls added to make it the right length.

$$\begin{pmatrix}17\\4\\19\end{pmatrix}\ \begin{pmatrix}17\\4\\0\end{pmatrix}\ \begin{pmatrix}19\\13\\14\end{pmatrix}\ \begin{pmatrix}22\\23\\23\end{pmatrix}$$

Now we perform matrix multiplication, multiplying the key matrix by each column vector in turn. To perform matrix multiplication we combine the top row of the key matrix with the column vector to get the top element of the resulting column vector. We then combine the middle row of the key matrix with the column vector to get the middle element of the resulting column vector and similarly for the bottom row. The way we "combine" the six numbers to get a single number is that we multiply the first element of the key matrix row by the top element of the column vector, multiply the second element of the key matrix row by the middle element of the column vector, and multiply the third element of the key matrix row by the bottom element of the column vector. We then add together these three answers.

We perform all the matrix multiplcations, and take the column vectors modulo 26. Then we convert them back into letters to produce the ciphertext.

$$\begin{pmatrix}1&0&2\\10&20&15\\0&1&2\end{pmatrix}\begin{pmatrix}17\\4\\19\end{pmatrix}=\begin{pmatrix}55\\535\\42\end{pmatrix}\ mod\ 26=\begin{pmatrix}3\\15\\16\end{pmatrix}=\begin{pmatrix}D\\P\\Q\end{pmatrix}$$

We repeat the same procedure for the following vectors. This gives us a final ciphertext of:

DPQRQ EVKPQ LR

1.2 Permutation

The idea of a permutation cipher is to keep the plaintext characters unchanged, but to alter their positions by rearranging them using a permutation. The **Permutation Cipher** (also known as the Transposition Cipher) is defined formally as Cryptosystem.

A **permutation** of a finite set X is a bijective function $\pi : X \to X$. In other words, the function π is one-to-one (injective) and onto (surjective).

Cryptosystem 1.6: *Permutation Cipher*

Let m be a positive integer. Let $\mathcal{P} = \mathcal{C} = (\mathbb{Z}_{26})^m$ and let $\mathcal{K}$ consist of all permutations of $\{1, \ldots, m\}$. For a key (i.e., a permutation) π, we define

$$e_\pi(x_1, \ldots, x_m) = (x_{\pi(1)}, \ldots, x_{\pi(m)})$$

and

$$d_\pi(y_1, \ldots, y_m) = (y_{\pi^{-1}(1)}, \ldots, y_{\pi^{-1}(m)}),$$

where π^{-1} is the inverse permutation to π.

1.2.1 Example

Suppose $m = 6$ and the key is the following permutation π:

x	1	2	3	4	5	6
$\pi(x)$	3	5	1	6	4	2

We can see that the first row of the above diagram lists the values of x, $1 \leq x \leq 6$, and the second row list the corresponding values of $\pi(x)$.

The inverse permutation π^{-1} can be constructed by interchanging the two rows, and rearranging the columns so that the first row is in increasing order.

x	1	2	3	4	5	6
$\pi^{-1}(x)$	3	6	1	5	2	4

Suppose we are given the plaintext:

shesellsseashellbytheseashore

We first partition the plaintext into groups of six letters:

shesel	lsseas	hellsb	ythese	ashore

Now each gropu of six letters is rearranged according to the permutation π, yielding the following:

eeslsh	salses	lshble	hsyeet	hraeos

So, the ciphertext is:

EESLSHSALSESLSHBLEHSYEETHRAEOS

2 Programming Exercises

2.1 Exercise 1

2.1.1 Key Generator

To generate the key we use three functions.

The first one called llenadoMatriz gets the matrix and fills it with random numbers and once filled, sends it to a .txt extension file.

✓Code

```
void llenadoMatriz(int **K) {
        srand(time(NULL));
        int num = 0, cont, cont2;
        FILE *f;
        // Create file to save key
        f = fopen("C:/Users/Dell/Documents/Crypto/Session2/key.txt", "w");
        for(cont = 0; cont < 3; cont++){
                fprintf(f, "(\t");
                for(cont2 = 0; cont2 < 3; cont2++){
                        // Generated element in the matrix (random) and
                         ↪  added to it
                        num = rand() % 10;
                        K[cont][cont2] = num;
                        fprintf(f, "%d\t", K[cont][cont2]);
                }
                fprintf(f, ")\n");
        }
        fclose(f);
}
```

The second function gets a matrix and calculates its determinant, based in the equation given in the the book "Cryptography. Theory and Practice". It is

$$a_{11}a_{22}a_{33} + a_{12}a_{23}a_{31} + a_{13}a_{21}a_{32} - (a_{11}a_{23}a_{32} + a_{12}a_{21}a_{33} + a_{13}a_{22}a_{31})$$

Finally returns the corresponding int value of the determinant.

✓Code

```
int matrixDeterminant(int **K) {
        int det = K[0][0] * K[1][1] * K[2][2] + K[0][1] * K[1][2] * K[2][0]
         ↪  + K[0][2] * K[1][0] * K[2][1] -
```

```
                                        (K[0][0] * K[1][2] * K[2][1] + K[0][1] *
                                        ↪  K[1][0] * K[2][2] + K[0][2] * K[1][1] *
                                        ↪  K[2][0]);
        return det;
}
```

In the last function we create the matrix that will be our key. Then we use a do-while structure to call the other two functions. First we call llenadoMatriz and after fill the matrix we obtain its determinant calling matrixDeterminant. In every iteration we call the function gcdRecursive to know if the gcd between the matrix determinant and our n (value of the modulo with we're working. In this case is 95) is equals to 1. In case it is true, the function will return the matrix filled that will be our key. Else, the process will be repeated till we find a valid key.

✓Code

```
int **generateKey() {
        int det=0, n=95;
        int **K = (int**)malloc(sizeof(int*)*3);
        //Se crea la matriz K
        for(int i = 0; i < 3; i++)
                K[i] = (int*)malloc(sizeof(int)*3);
        // Keys are generated until one is valid
        do{
                llenadoMatriz(K);
                det = matrixDeterminant(K);
        }while(gcdRecursive(det, n) != 1);
        return K;
}
```

2.1.2 Plaintext Encryption

This funciont receives our key matrix and the path where is located the plaintext.

The first thing we do is open the file that will be encrypted and get all the text, which we store in the string msj.

After that we proceed with de cipher of the plaintext. First we use a for structuer that gets the next 3 characters in our message that will be encrypted, and stores them temporarily in an array. Inside this cycle we use other two for cycles to multiply every one of the values in the array with its correspondent possition in each one of the columns of the matrix. So every time we multiply the array with a column, we obtain the new value of one character which will be concatenated with the string that contains our ciphertext.

This process is repeated till we reach the end of the message. Then we return the ciphertext.

✓Code

```
string cipher(int **K, char *path) {
        fstream f;
        int car[3], new_val[3];
        int cont, ck = 0, aux = 0, cont2, cont3 = 0;
        string aux_msj, msj, cip_msj;
        // Read file
        f.open(path);
        if(f.is_open()) {
                while(!f.eof()) {
                        getline(f, aux_msj);
                        msj += aux_msj;
                }
                f.close();
        }
        // CHIPER : Cycle obtained by the group of 3 characters
        for(cont3 = 0; cont3 < msj.size(); cont3+=3){
                car[0] = (int)msj[cont3] - 32;
                car[1] = (int)msj[cont3 + 1] - 32;
                car[2] = (int)msj[cont3 + 2] - 32;

                // Multiplication of the matrices to obtain the new values
                for(cont = 0; cont < 3; cont++){
                        for(cont2 = 0; cont2 < 3; cont2++)
                        // Obtain Aij
                                aux=aux+(car[cont2] * K[cont2][cont]);
                        new_val[cont] = aux % 95;
                        aux = 0;
                        cip_msj += (new_val[cont] + 32);
                }
        }
        return cip_msj;
}
```

2.1.3 Inverse key

This function is used to get the inverse key so we can decrypt the ciphertext. It receives the key and returns the its inverse.

The first thing we do is to obtain the key determinant and calculate its inverse. Then we create the matrix that will stores our inverse key.

So to obtain the inverse matrix we need to previously obtain the adjoint matrix. But to do it we need to obtain the cofactor of each one of the elements in the matrix, and that's what

the four nested for's do.

The first two cycles are to run trough all the matrix, and the other two cycles are to obtain the cofactor of each one of the elements. So considering the cofactors method in linear algebra, we know that in a 3x3 matrix, to obtain the cofactor of an elemnt we take the four elements that are neither row nor column of the element we're analysing (who will be our cofactor matrix) and we calculate it's determinant, the resulting number will be the cofactor of the element we're currently analysing. To do this we compare if the counters of the two nestles for's are equals or not to the counters of the other two for's (always considering that the counters are comparing row with row and column with column). If both column and row comparisons are different we add the element in the current possition of the matrix to the cofactor matrix (which for facility we handled it as an unidimensional array). Once do that, we obtain the cofactor using the equation

$$cofactor = cofac[0]xcofac[3] - cofac[1]xcofac[2]$$

And to know the corresponding sign ('+' or '-') to the current possition in the matrix we only check if the sum of the counters that run trough all the matrix is a pair number or not.

After all of that we calculate $cofactormodn$ to assure that all of the elements belong to $\mathbb{Z}_n$.

The resultant matrix of the last step is almost the adjoint matri, except that it is not transposed. Thing we're going to do later.

Then we multiply all the elements in the matrix by the inverse of the determinant of the key, that was calculated at the beginning and one more time we calculate $\mathbb{Z}_n$ for all the elements.

Finally we only fill the matrix of the inverse key, but instead of doing it by row, we do it by column, doing this way, the step that was missing to obtain the adjoint matrix.

The function returns the matrix with the inverse key.

✓Code

```
int **inverseKey(int **K){
        int cofac[4];
        int cont, cont2, cont3, cont4, cont_cofac, det = 0, inv_det = 0,
         ↪  cofactor = 0, n = 95;
        // Obtaining the determinant and its inverse
        det = matrixDeterminant(K);
        inv_det = inverse(n, det);
        // Create inverse matrix
        int **K_inv = (int**)malloc(sizeof(int*) * 3);
        for(int i = 0; i < 3; i++)
                K_inv[i] = (int*)malloc(sizeof(int) * 3);
        // Create attached matrix
        // First two cycles to traverse all the elements of the matrix
```

```
        for(cont = 0; cont < 3; cont++){
                for(cont2 = 0; cont2 < 3; cont2++){
                        // Last two cycles to obtain cofactor of the element
                         ↪  analyzed
                        cont_cofac = 0;
                        for(cont3 = 0; cont3 < 3; cont3++){
                                for(cont4 = 0; cont4 < 3; cont4++){
                                        if(cont != cont3 && cont2 != cont4){
                                                cofac[cont_cofac] =
                                                 ↪  K[cont3][cont4];
                                                cont_cofac++;
                                        }
                                }
                        }
                        // Get cofactor
                        cofactor = cofac[0] * cofac [3] - cofac[1] *
                         ↪  cofac[2];
                        // It is evaluated if i + j of the position of the
                         ↪  matrix is even or odd
                        if ((cont + cont2) % 2 != 0)
                                cofactor = cofactor * (-1);
                        // Get mod of cofactor
                        if(cofactor < 0)
                                cofactor = n + (cofactor % n);
                        else
                                cofactor = cofactor % n;
                        // The cofactor element is multiplied by the inverse
                         ↪  of the determinant and the module is obtained
                        cofactor = (cofactor * inv_det) % n;
                        /* The matrix of cofactors is transposed to obtain
                         ↪  attached
                           (which will be the inverse of the key by the
 ↪  previous operation) */
                        K_inv[cont2][cont] = cofactor;
                }
        }
        return K_inv;
}
```

2.1.4 Ciphertext Decryption

This is the same algorithm used for the encryption, but instead of multiplying the groups of three characters by the key, we do it by the inverse key, so we can obtain de plaintext, that will be returned by the function.

✓Code

```
string decipher(int **K, char *path) {
        fstream f;
        int car[3], new_val[3], cofac[4];
        int cont, aux = 0, cont2, cont3, n = 95;
        string aux_msj, msj, decip_msj;

        // Obtain inverse K
        int **K_inv = inverseKey(K);

        // Get cipher text
        f.open(path);
        if(f.is_open()) {
                while(!f.eof()) {
                        getline(f, aux_msj);
                        msj += aux_msj;
                }
                f.close();
        }
        aux = 0;
        // DESCIPHER: Cycle obtained by the group of 3 characters
        for(cont3 = 0; cont3 < msj.size(); cont3+=3) {
                car[0] = (int)msj[cont3] - 32;
                car[1] = (int)msj[cont3 + 1] - 32;
                car[2] = (int)msj[cont3 + 2] - 32;
                // Multiplication of the matrices to obtain the new values
                for(cont = 0; cont < 3; cont++) {
                        for(cont2 = 0; cont2 < 3; cont2++)
                                // Get Aij
                                aux=aux+(car[cont2] * K_inv[cont2][cont]);
                        new_val[cont] = aux % n;
                        aux = 0;
                        decip_msj += (new_val[cont] + 32);
                }
        }
        return decip_msj;
}
```

2.2 Exercise 2

2.2.1 Plaintext Encryption

In this subsection, we will explain the main code. To be able to use this encryption, we need the size of the key, that is **m**. The user give us **m** by the command line and the plain text by a file. We expect an entry like the following:

- 7
- fileName.in

We use **getline()** for each entry and we received an string, so we convert **m** to a integer using **atoi()**. With an integer **m**, we can generated a valid key with the function **generateKey()**.

The next step is make blocks of size **m**, but we know that generally we have incomplete blocks, we need to know when is that case, so we validated it with the gcd, if the gcd is equal to 1, we need to fill spaces, and we made it with the function **fillSpace()**. After that, we have complete blocks to use.

Finally, we send the blocks to be cipher in the function **permutationCipher()**.

✓Code

```
int main(int argc, char const *argv[]) {
        int i, j, auxK, m, size, block, space, end;
        string tam, cadena;
    char *path;
    fstream f;
        vector<int> vec;

        // Read m
    getline(cin, tam);

        // Read plain text
        string dir;
        getline(cin, dir);
        path = const_cast<char*>(dir.c_str());
        f.open(path);
        if(f.is_open()){
                while(!f.eof()){
                        getline(f, cadena);
                        msj += cadena;
                }
                f.close();
        }
```

```
        // Convert m to integer
        m = atoi(tam.c_str());

        // Generate a valid Key
        generateKey(m);

        // Use gcd to know if we need spaces
        size = msj.length();
        if((gcdRecursive(m, size) == 1) && (m > 1)) {
                fillSpace(size, m);
        }

        // Cipher
         string cip = " ";
         i = 0;
         while(i < msj.length()) {
                 // Obtain a substring with size of m
                cip = msj.substr(i, m);
                // Send substring to cipher
                permutationCipher(cip, m);
                i += m;
         }
        return 0;
}
```

2.2.2 Generate key

We use the random function by c++, first we generate a secret number between 1 and m. We have a vector called **key**, if the number generated exist in key, we can not put it, in other case we can use the number. Thus, we are sure that the number is not repeated in the vector key.

✓Code

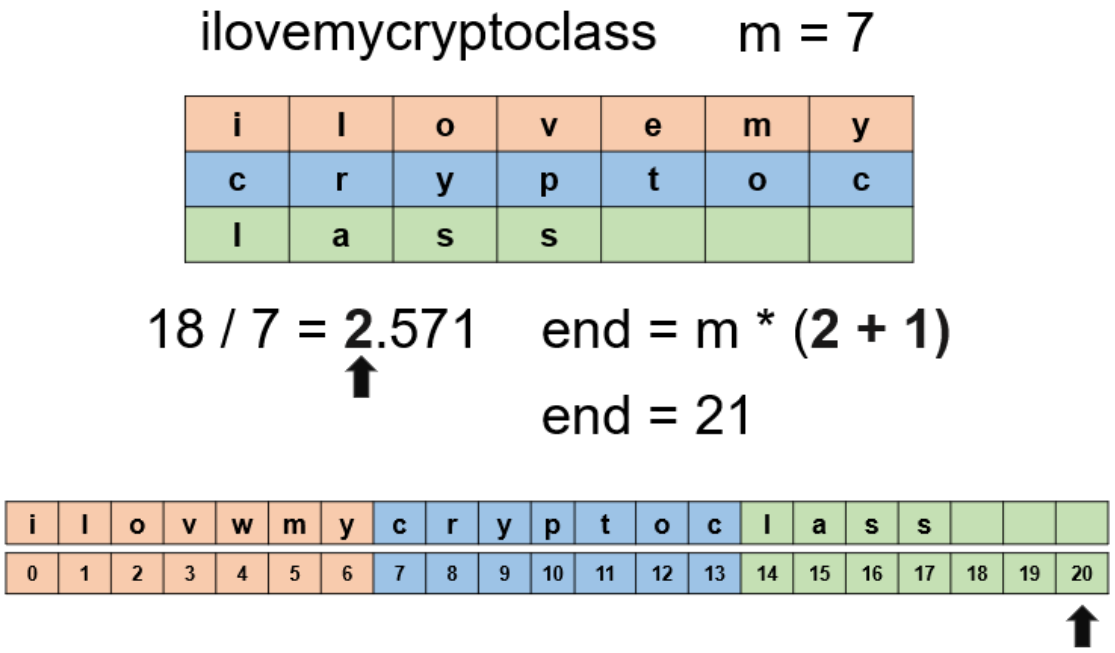
```
void generateKey(int n) {
        int i, iSecret;
        srand(time(NULL));
        // Fill vector Key
        i = 0;
        while(i < n) {
                // Generate secret number between 1 and m:
            iSecret = 1 + rand() % (n);
            if(find(key.begin(), key.end(), iSecret) == key.end()) {
                        key.push_back(iSecret); i++;
```

```
                }
        }
        // Print key
        for(i = 0; i < n; i++)
                cout << key.at(i);
        cout << endl;
}
```

2.2.3 Fill space

Since the plaintext does not go perfectly into the column vectors, we need to use some nulls to make the plaintext the right length, in this case, we use a space character.

We calculated the size of the vector that we need with all the complete blocks (variable **end**).



✓Code

```
void fillSpace(int size, int m) {
        string aux = " ";
        double sized, md, bd;
        int i, block, end;

        // Calculate how many spaces we have
        sized = (double)size;
```

```
        md = (double)m;
        bd = sized / md;
        bd = floor(bd); // Exact block
        block = (int)bd;
        end = m * (block + 1);
        // Fill spaces with " "
        for(i = size; i < end; i++) {
                msj = msj + aux;
        }
}
```

2.2.4 Cipher with permutation

We received a block of the string, we use the key and we changed the position in the correct place.

key

X	1	2	3	4	5	6
$\pi(x)$	3	5	1	6	4	2

✓Code

```
void permutationCipher(string blo, int m) {
        int i, pos;
        string cipher;
        for(i = 0; i < m; i++)
                cipher.push_back(' ');

        for(i = 0; i < blo.length(); i++)
                cipher[i] = blo[key.at(i)-1];

        cout << cipher;
}
```

2.2.5 Ciphertext Decryption

In this subsection, we will explain the main code. To be able to use this encryption, we need a key. The user give us the key and the plain text by a file. We expect an entry like the following:

- key.in
- fileName.in

We use **getline()** for each entry and we received an string, we received the original key but we need the inverse key, we get it with the function **getInverse**.

Finally, we send the blocks to be decipher in the function **permutationDecipher()**.

✓Code

```
int main(int argc, char const *argv[]) {
        int i, j, auxK, m, size, block, space, end, num;
        string tam, cadena, aux;
    char *path, *pathAux;
    fstream f, fAux;
        vector<int> vec;

        // Read Key file
        string dir;
        getline(cin, dir);
        path = const_cast<char*>(dir.c_str());
        f.open(path);
        if(f.is_open()) {
                while(!f.eof()) {
                        getline(f, cadena);
                        keyS += cadena;
                }
                f.close();
        }
        // Read cipher text
        string dirAux;
        getline(cin, dirAux);
        pathAux = const_cast<char*>(dirAux.c_str());
        fAux.open(pathAux);
        if(fAux.is_open()) {
                while(!fAux.eof()) {
                        getline(fAux, aux);
                        msj += aux;
                }
                fAux.close();
```

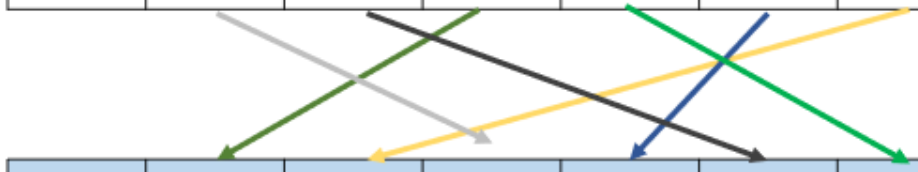
```
        }
        // Get inverse Key
        getInverse(keyS);

        // Decipher
        string cip = " ";
         i = 0;
         while(i < msj.length()) {
                 // Obtain a substring with size of m
                cip = msj.substr(i, key.size());
                // Send substring to cipher
                permutationDecipher(cip, key.size());
                i += key.size();
         }
        return 0;
}
```

2.2.6 Generate inverse key

We received a string **keyS**, that is the original key, with witch the plain text was cipher, so we need to converted it to a inverse key, first we change the string to integer, and then we only change the number as the following example:

X	1	2	3	4	5	6
$\pi(x)$	3	5	1	6	4	2



x	1	2	3	4	5	6
$\pi^{-1}(x)$	3	6	1	5	2	4

✓Code

```
vvoid getInverse(string keyS) {
        int i, aux, a, b;
        string c;
```

```
        char cAux, bAux;
        vector<int> keyAux;

        keyS = keyS + " ";
        for(i = 0; i < keyS.length()-1; i++) {
                a = i + 1;
                cAux = keyS.at(a);
                bAux = keyS.at(i);
                a = cAux;
                b = bAux;
                c = keyS.at(i);
                // Get the number
                if(a == 32) {
                        aux = atoi(c.c_str());
                        keyAux.push_back(aux);
                }
                else if (b != 32) {
                        a = i + 1;
                        c = c + keyS.at(a);
                        aux = atoi(c.c_str());
                        keyAux.push_back(aux);
                        i++;
                }
        }

        for(i = 0; i < keyAux.size(); i++)
                key.push_back(0);

        for(i = 0; i < keyAux.size(); i++) {
                aux = keyAux[i];
                key[aux-1] = i + 1;
        }
        // Print key
        for(i = 0; i < key.size(); i++)
                cout << key.at(i) << " ";
        cout << endl;
}
```

2.2.7 Decipher with permutation

We received a block of the string, we use the key and we changed the position in the correct place.

key

x	1	2	3	4	5	6
$\pi^{-1}(x)$	3	6	1	5	2	4

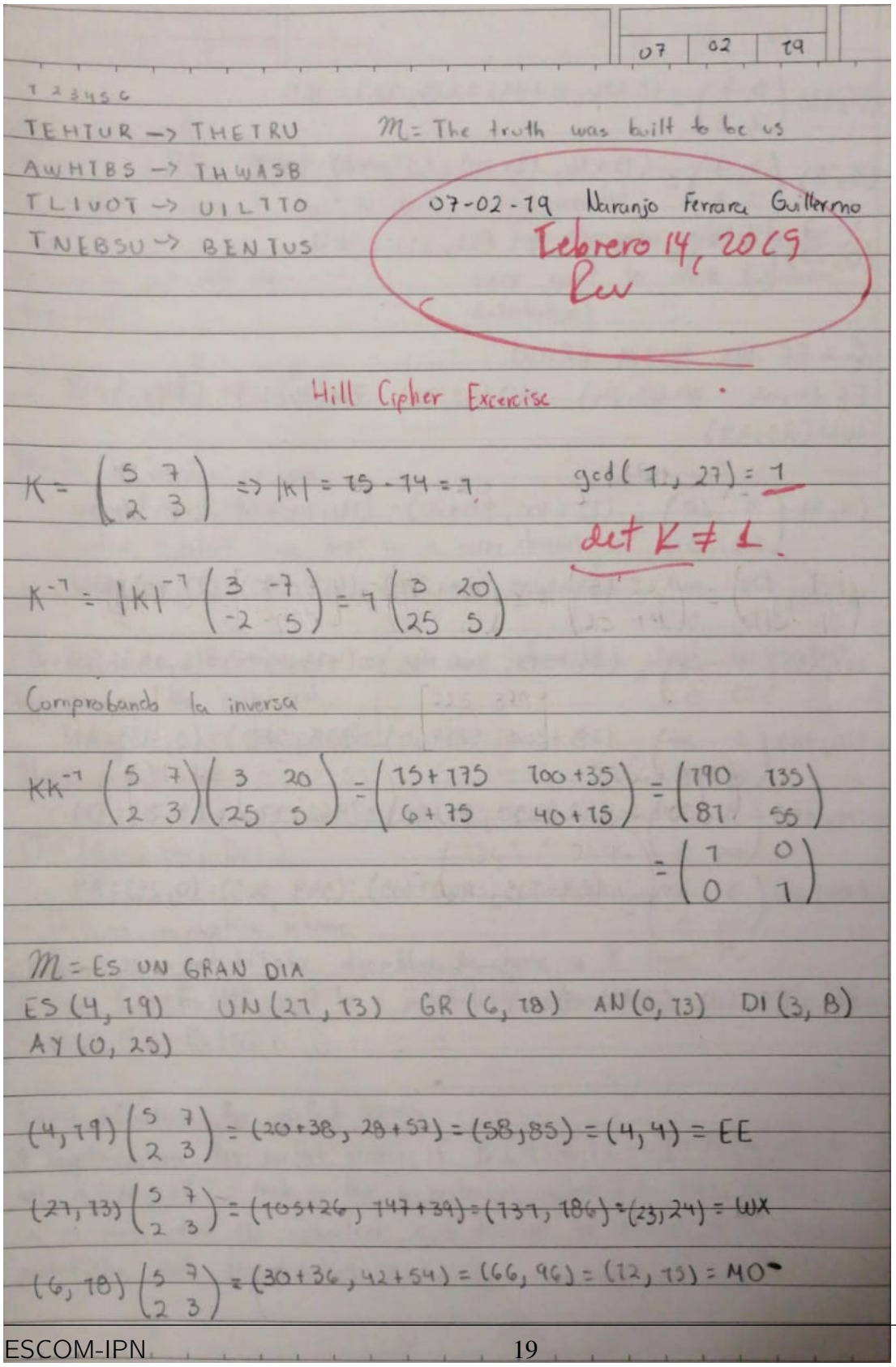
✓**Code**

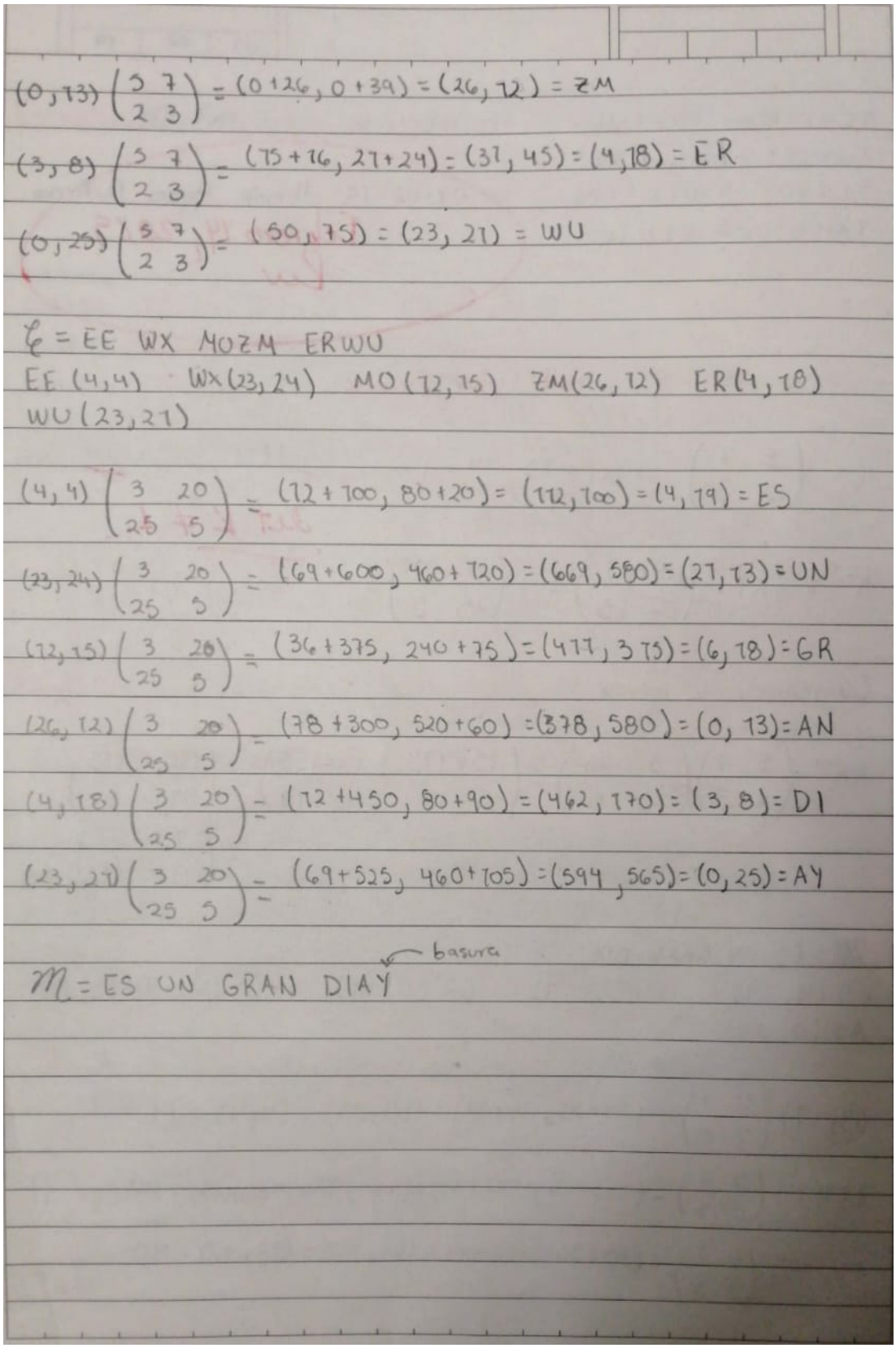
```
void permutationDecipher(string blo, int m) {
        int i, pos;
        string cipher;
        for(i = 0; i < m; i++)
                cipher.push_back(' ');

        for(i = 0; i < blo.length(); i++)
                cipher[i] = blo[key.at(i)-1];
        cout << cipher;
}
```

3 Notebook Exercises

3.1 Naranjo Ferrara Guillermo





$(0,13)\begin{pmatrix}5&7\\2&3\end{pmatrix} = (0+26, 0+39) = (26,12) = ZM$

$(3,8)\begin{pmatrix}5&7\\2&3\end{pmatrix} = (15+16, 21+24) = (31,45) = (4,18) = ER$

$(0,25)\begin{pmatrix}5&7\\2&3\end{pmatrix} = (50,75) = (23,21) = WU$

$\mathcal{C}$ = EE WX MOZM ERWU

EE (4,4) WX(23,24) MO(12,15) ZM(26,12) ER(4,18) WU(23,21)

$(4,4)\begin{pmatrix}3&20\\25&5\end{pmatrix} = (12+100, 80+20) = (112,100) = (4,19) = ES$

$(23,24)\begin{pmatrix}3&20\\25&5\end{pmatrix} = (69+600, 460+120) = (669,580) = (27,13) = UN$

$(12,15)\begin{pmatrix}3&20\\25&5\end{pmatrix} = (36+375, 240+75) = (411,315) = (6,18) = GR$

$(26,12)\begin{pmatrix}3&20\\25&5\end{pmatrix} = (78+300, 520+60) = (378,580) = (0,13) = AN$

$(4,18)\begin{pmatrix}3&20\\25&5\end{pmatrix} = (12+450, 80+90) = (462,170) = (3,8) = DI$

$(23,21)\begin{pmatrix}3&20\\25&5\end{pmatrix} = (69+525, 460+105) = (594,565) = (0,25) = AY$

basura

$\mathcal{M}$ = ES UN GRAN DIAY

3.2 Nicolás Sayago Abigail