The Hill Cipher

1. Characters vs numbers

The Hill Cipher uses a mathematical technique that needs *numbers* instead of *characters*. This means that we must first convert the plaintext **`Message`** into an equivalent array of numbers between 0 and 25.

This is done in the straightforward way (as also in the other techniques, where *numbers* have to be substituted for *characters*, like in our implementation of the Caesar Cipher.)

```
for( i=0; i< Message.length(); i++ )
{
 _M[i]= (int) Message.charAt(i)-97;
}
```

This assumes that the input plaintext is in *all lowercase characters*.

When we want to read the encrypted *ciphertext* and also the *decrypted* plaintext, we will have to convert an integer array into a character array. It will also be straightforward, as below.

For getting the ciphertext:

```
for( i=0; i<Message.length(); i++ )
{ Cipher[i]= (char)(_C[i]+97); }
```

After decryption, when we want to read the plaintext (for verification, perhaps),

```
for( i=0; i<Message.length(); i++ )
{ Decrypt[i]= (char)(_D[i]+97); }
```

The rest is mathematics.

2. Encryption

The encryption of the **`Message`** is performed by <u>taking two characters at a time</u>. This can also be done using *three* characters at a time, but we will take two since the code is exceedingly simple in this case.

The two characters at a time are processed in a way that resembles *matrix multiplication modulo 26*. This corresponds to the procedure below.

The *matrix* here is the 2X2 “key” matrix called **`Key[][]`**. It is used as:

```
/*Encryption.
  Take two characters at a time.
  Assume the plaintext has an even number
of characters. */
  for( i=0; i<Message.length(); i+=2 )
  {
   _C[i]= ( Key[0][0]*_M[i] +
Key[0][1]*_M[i+1] )%26;
   _C[i+1]= ( Key[1][0]*_M[i] +
Key[1][1]*_M[i+1] )%26;
  }
```

3. Decryption

The decryption is a little more work, but it is simple, once we write down the maths.

The main point lies in *taking the inverse of a matrix*. In order to decrypt the ciphertext (in its numerical form **`_C[]`**), we must carry out the same procedure of i) taking two characters at a time, and ii) performing matrix multiplication *modulo 26*. The only difference is, we will <u>not</u>

use the same 2X2 matrix **Key[][]** here, but instead its *inverse*.

The formula for the inverse of a 2X2 matrix is very simple:

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix}^{-1} = \begin{pmatrix} d & -b \\ -c & a \end{pmatrix} \times (\, ad - bc \,)^{-1}$$

But we are dealing with matrices *modulo 26* (actually, the technical term is *matrices over a finite field*), which means that we cannot calculate $(\, ad - bc \,)^{-1}$the usual way. We must take the *"multiplicative inverse"* of $(\, ad - bc \,)$ modulo 26. This too is found using a simple procedure.

So, first, we must find the multiplicative inverse of $(\, ad - bc \,)$, as below. (Remember that, in our program, a is **Key[0][0]**, b is **Key[0][1]**, c is **Key[1][0]** and d is **Key[1][1]**.)

```
/* Call this quantity (ad-bc) "det". */
    det=
 Key[0][0]*Key[1][1]-
 Key[0][1]*Key[1][0];


for( i=0; i<26; i++ )
{
 if( (i*det)%26 == 1 ) Mul_inv= i;
}
```

Now the value of $(ad - bc)^{-1}$modulo 26 has been found and is stored in the variable **Mul_inv**.

Let us next save the inverse matrix as below:

```
  /*Using the formula for inverse of a 2X2 matrix, modulo 26.*/
  Key_Inv[0][0]= (Key[1][1]*Mul_inv);
  Key_Inv[0][1]= (-1*Key[0][1]*Mul_inv);
  Key_Inv[1][0]= (-1*Key[1][0]*Mul_inv);
  Key_Inv[1][1]= (Key[0][0]*Mul_inv);
```

And finally, the decryption.

We can perform the decryption in the same way as encryption, <u>except</u> we now use the matrix **Key_Inv[][]** instead of the matrix **Key[][]**, and also, we are processing the *ciphertext* **_C[]** instead of the message **_M[]**.

```
/*Decryption. Just like in encryption: Take two characters at a time. */
  for( i=0; i<Message.length(); i+=2 )
  {
   _D[i]= ( Key_Inv[0][0]*_C[i] + Key_Inv[0][1]*_C[i+1] )%26;
```

```
   _D[i+1]= ( Key_Inv[1][0]*_C[i] +
Key_Inv[1][1]*_C[i+1] )%26;
  }
```

Done.

4. Input/Output

The small point of the input/output can be tackled as follows.

i) The Input of the (plaintext) message:
(The user must take care to enter <u>only strings with even-number of characters</u> like *good*, *crypto*, *hill*, *cipher* etc.)

```
Message= br.readLine();
```

ii) The printing of the ciphertext:

```
for( i=0; i<Message.length(); i++ )
{ System.out.print( Cipher[i] ); }
```

iii) The input of the key matrix:
(The user <u>cannot</u> enter any matrix he/she wishes to. The matrix chosen may <u>not be invertible</u> modulo 26. For the sake of reference, you may use the matrix

entered in the sample run (Appendix B, see below), which is: $\begin{pmatrix} 3 & 0 \\ 0 & 1 \end{pmatrix}$.)

```
  Key[0][0]=
Integer.parseInt(br.readLine());
  Key[0][1]=
Integer.parseInt(br.readLine());
  Key[1][0]=
Integer.parseInt(br.readLine());
  Key[1][1]=
Integer.parseInt(br.readLine());
```

iv) The printing of the decrypted text:

```
for( i=0; i<Message.length(); i++ )
{ System.out.print( Decrypt[i] ); }
```

APPENDIX.

A.Full Program

Below is the full, and working, program.

Again, this program is actually quite small, but appears large only because of the typing style.

```
import java.io.*;


public class HC
{
 public static void main( String[] A )
throws Exception, IOException
  {


   //Beginning:


   int i, det, Mul_inv= 0;
   String Message;
   char[] Cipher= new char[1000], Decrypt=
new char[1000];
   int[] _C= new int[1000], _D= new
int[1000], _M= new int[1000];
```

```
  int[][] Key= new int[2][2], Key_Inv= new
int[2][2];



  BufferedReader br= new BufferedReader(
new InputStreamReader( System.in ) );

  System.out.println( "Enter plaintext:" );
  Message= br.readLine();

  System.out.println( "Enter key matrix:"
);
  Key[0][0]=
Integer.parseInt(br.readLine());
  Key[0][1]=
Integer.parseInt(br.readLine());
  Key[1][0]=
Integer.parseInt(br.readLine());
  Key[1][1]=
Integer.parseInt(br.readLine());

  /*Convert Message into integer array:*/
  for( i=0; i< Message.length(); i++ )
```

```
  {
   _M[i]= (int) Message.charAt(i)-97;
  }

//------------------------------------------
------------------

  /*Encryption.
  Take two characters at a time.
  Assume the plaintext has an even number
of characters. */
  for( i=0; i<Message.length(); i+=2 )
  {
   _C[i]= ( Key[0][0]*_M[i] +
Key[0][1]*_M[i+1] )%26;
   _C[i+1]= ( Key[1][0]*_M[i] +
Key[1][1]*_M[i+1] )%26;
  }

  /*Convert number version into string of
characters.*/
  for( i=0; i<Message.length(); i++ )
  { Cipher[i]= (char)(_C[i]+97); }
```

```
  /*Print ciphertext.*/
  System.out.println( "\nCiphertext:" );
  for( i=0; i<Message.length(); i++ )
  { System.out.print( Cipher[i] ); }

//----------------------------------------------------------------

  /*Decryption.*/

  /*Finding multiplicative inverse.*/
  /* Call this quantity (ad-bc) “det”. */
  det=
Key[0][0]*Key[1][1]-Key[0][1]*Key[1][0];

  for( i=0; i<26; i++ )
  {
   if( (i*det)%26 == 1 ) Mul_inv= i;
  }
```

```
/*Using the formula for inverse of a 2X2 matrix, modulo 26.*/
Key_Inv[0][0]= (Key[1][1]*Mul_inv);
Key_Inv[0][1]= (-1*Key[0][1]*Mul_inv);
Key_Inv[1][0]= (-1*Key[1][0]*Mul_inv);
Key_Inv[1][1]= (Key[0][0]*Mul_inv);

/*And finally,..
Just like in encryption: Take two characters at a time. */
for( i=0; i<Message.length(); i+=2 )
{
 _D[i]= ( Key_Inv[0][0]*_C[i] + Key_Inv[0][1]*_C[i+1] )%26;
 _D[i+1]= ( Key_Inv[1][0]*_C[i] + Key_Inv[1][1]*_C[i+1] )%26;
}

/*Convert number version into string of characters.*/
for( i=0; i<Message.length(); i++ )
{ Decrypt[i]= (char)(_D[i]+97); }
```

```
  /*Print decrypted text.*/
  System.out.println( "\nDecrypted text:"
);
  for( i=0; i<Message.length(); i++ )
  { System.out.print( Decrypt[i] ); }


 }
}
```

B.Sample output

Here is a sample run of the program.

```
C:\Users\15it070\Downloads>java HC
Enter plaintext:
good
Enter key matrix:
3
0
0
1

Ciphertext:
soqd
Decrypted text:
good
```