HILL CIPHER

STUDENTS: BASTIDA PRADO JAIME ARMANDO

SOLORIO PAREDES DANIEL

PROFESSOR: DÍAZ SANTIAGO SANDRA

SUBJECT: CRYPTOGRAPHY

GROUP: 3CM6

February 19th 2019

1. Hill Cipher

This programm is an implementation of the Hill cipher considering the set of printable ASCII characters.

1.1. Generating a random key

To encipher we need to generate a random matrix key and verify that its determinant is not cero and that $gcd(determinant, 95) = 1$.

After the random key is generated the program calculates the determinant:

```
49              printf("Calculating determinant...\n");
50              determinant = 0;
51              for(x = 0; x < 6; x++)
52              {
53                  if(x == 0)
54                      determinant += key[0][0] * key[1][1] * key[2][2];
55                  else if(x == 1)
56                      determinant += key[0][1] * key[1][2] * key[2][0];
57                  else if(x == 2)
58                      determinant += key[0][2] * key[1][0] * key[2][1];
59                  else if(x == 3)
60                      determinant += -(key[2][0] * key[1][1] * key[0][2]);
61                  else if(x == 4)
62                      determinant += -(key[2][1] * key[1][2] * key[0][0]);
63                  else if(x == 5)
64                      determinant += -(key[2][2] * key[1][0] * key[0][1]);
65              }
66              if(determinant < 0)
67                  determinant = 95 - (-determinant % 95);
68              else
69                  determinant = determinant % 95;
70
71              printf("Determinant = %d\n", determinant);
```

Figura 1: Calculating the determinant

Then it validates $gcd(determinant, 95) = 1$ by applying the Euclid Algorithm.

```
72              printf("Validating gcd(%d, 95) = 1\n", determinant);
73              //Validating "a" using the Euclid gcd algorithm
74              n = 95;
75              remainder = 0;
76              aux = determinant;
77              while(1)
78              {
79                  remainder = aux % n;
80                  aux = n;
81                  n = remainder;
82                  if(remainder == 0)
83                      break;
84                  previousRemainder = remainder;
85              }
86              if(previousRemainder != 1)
87              {
88                  printf("|+|+|+|ERROR: Key not valid. Trying again.\n\n\n");
89                  fclose(write_fp);
90                  goto tryagainkey;
91              }
92              printf("Validated.\n");
```

Figura 2: Euclid Algorithm

1.2. Enciphering

Once we have the key in a file, to encipher the program receives the file with the key and validates it. It does also receive the file with the plaintext. Now, the program does the matrix multiplication to encipher and stores the ciphertext, as can be seen here:

```
137             printf("Enciphering...\n");
138             while((ch = getc(read_fp)) != EOF)
139             {
140                 if(ch > 31 && ch < 127)
141                 {
142                     msg[i++] = ch - 32;
143                     if(i > 2)
144                     {
145                         i = 0;
146                         for(j = 0; j < 3; j++)
147                             putc(((msg[0] * key[j][0] + msg[1] * key[j][1] + msg[2] * key[j][2]) % 95) + 32, write_fp);
148                     }
149                 }
150                 else
151                     putc(ch, write_fp);
152             }
153             printf("Enciphered\n");
```

Figura 3: Encipher

1.3. Deciphering

To decipher we need to provide the program with the key and the ciphertext, both stored in a file. Once the program receives the key it validates the key, reads it and finds the inverse of the determinant by applying the Extended Euclid Algorithm

```
221             printf("Step Three: Finding inverse of determinant using Extended Euclid Algorithm\n");
222             n = 95;
223             index = 0;
224             while(1)
225             {
226                 remainder = n % determinant;
227                 if(remainder == 0)
228                     break;
229
230                 array[index][0] = remainder;
231                 array[index][1] = n;
232                 array[index][2] = determinant;
233                 array[index][3] = - (n - remainder) / determinant;
234
235                 n = determinant;
236                 determinant = remainder;
237                 index++;
238             }
239             array1[0][0] = 1;
240             array1[0][1] = array[0][3];
241             factor1 = array[0][1];
242             factor2 = array[0][2];
243             for(i = 1; i < index; i++)
244             {
245                 product1 = array[i][3] * array1[i - 1][0];
246                 product2 = array[i][3] * array1[i - 1][1];
247
248                 if(array[i][1] == factor1)
249                     product1++;
250                 else if(array[i][1] == factor2)
```

Figura 4: Extended Euclid Algorithm

After that, the program finds the adjugate of the key in four steps: doing the transpose, finding the matrix of minors, finding the matrix of cofactors and applying modulo 95 to all entries in the matrix.

```
268                 printf("Step Four: Transpose Matrix\n");
269                 aux = key[0][1];
270                 key[0][1] = key[1][0];
271                 key[1][0] = aux;
272
273                 aux = key[0][2];
274                 key[0][2] = key[2][0];
275                 key[2][0] = aux;
276
277                 aux = key[1][2];
278                 key[1][2] = key[2][1];
279                 key[2][1] = aux;
280
281                 printf("Step Five: Matrix of Minors\n");
282                 for(i = 0; i < 3; i++)
283                 {
284                     for(j = 0; j < 3; j++)
285                     {
286                         //Building minor for a_ij
287                         m = 0;
288                         for(k = 0; k < 3; k++)
289                         {
290                             if(k != i)
291                             {
292                                 n1 = 0;
293                                 for(l = 0; l < 3; l++)
294                                     if(l != j)
295                                     {
296                                         minor[m][n1++] = key[k][l];
297                                     }
298                                 m++;
299                             }
```

Figura 5: Adjugate Matrix

To find the inverse of the key, the program multiplies the inverse of the determinant by the adjugate.

```
305             printf("Step Six: Adjugate, Matrix of Cofactors\n");
306             for(i = 0; i < 3; i++)
307                 for(j = 0; j < 3; j++)
308                     matrix_minors[i][j] *= (int) pow(-1.0, (double) (i + j));
309
310             printf("Step Seven: Applying mod 95 to all entries in the Adjugate Matrix\n");
311             for(i = 0; i < 3; i++)
312                 for(j = 0; j < 3; j++)
313                     if(matrix_minors[i][j] < 0)
314                         matrix_minors[i][j] = 95 - (-matrix_minors[i][j] % 95);
315                     else if(matrix_minors[i][j] > 95)
316                         matrix_minors[i][j] = matrix_minors[i][j] % 95;
317
318             printf("Step Eight and Final: Multiplying each entry by the inverse of the dete
319             for(i = 0; i < 3; i++)
320                 for(j = 0; j < 3; j++)
321                     matrix_minors[i][j] = (matrix_minors[i][j] * detInverse) % 95;
322
```

Figura 6: Inverse Key

Finally, to decipher we do the multiplication of the inverse key by the ciphertext. Notice that since the encipher we omit the set of non-printable ASCII characters.

```
336             printf("Deciphering...\n");
337             while((ch = getc(read_fp)) != EOF)
338             {
339                 if(ch > 31 && ch < 127)
340                 {
341                     msg[i++] = ch - 32;
342                     if(i > 2)
343                     {
344                         i = 0;
345                         for(j = 0; j < 3; j++)
346                             putc(((msg[0] * matrix_minors[j][0] + msg[1] * matrix_minors[j][1]
347                     }
348                 }
349                 else
350                     putc(ch, write_fp);
351             }
352             printf("Deciphered\n");
```

Figura 7: Decipher

2. Permutation Cipher

This program implements the Permutation cipher which is a special case of the Hill cipher.

2.1. Encipher

To encipher the program needs to receive the length of the key, the positions of $\pi(x)$ and the file with the plaintext. Then the program calculates the permutation of a block of 'key_len'characters (i.e. the length of the key) by consulting the array of $\pi(x)$ using an auxiliary array because the original array will be affected by the algorithm:

```
65              i = 0;
66              while((ch = getc(read_fp)) != EOF)
67              {
68                  if(ch > 31 && ch < 127)
69                  {
70                      msg[i++] = ch;
71                      if(i >= k_len)
72                      {
73                          for(i = 0; i < k_len; i++)
74                              aux[i] = msg[i];
75
76                          for(i = 0; i < k_len; i++)
77                              msg[i] = aux[key[i] - 1];
78
79                          for(i = 0; i < k_len; i++)
80                              putc(msg[i], write_fp);
81
82                          i = 0;
83                      }
84                  }
85                  else
86                      putc(ch, write_fp);
87              }
```

Figura 8: Encipher

2.2. Decipher

To decipher we do the same algorithm as with the encipher with the exception of calculating the inverse of the key. This is done by finding the inverse position of the original key array

```
116             //Obtaining inverse of the key
117             for(i = 0; i < k_len; i++)
118                 aux[i] = key[i];
119
120             for(i = 0; i < k_len; i++)
121             {
122                 for(j = 0; j < k_len; j++)
123                 {
124                     if(i == aux[j] - 1)
125                         break;
126                 }
127                 key[i] = j + 1;
128             }
```

Figura 9: Decipher