Instituto Politécnico Nacional

Escuela Superior de Cómputo

Session 1: Substitution cipher

Cryptography

Group: 3CM6

Students:
Nicolás Sayago Abigail
Naranjo Ferrara Guillermo

Teacher:
Díaz Santiago Sandra

February 6, 2019

Contents

1 Topics

1.1 Shift Cipher

The Shift Cipher is a symmetric cipher algorithm that use the technique of substitution; it means that, for both sender and receiver of the message there's a single key; and the encryption of the message will consist in the replacement of the symbols in the plain text for others, giving us the ciphertext.

The Shift Cipher is based on modular arithmetic.It consist to assign a particular integer number to each one of the symbols that compose your alphabet. It is usually defined over $\mathbb{Z}_{26}$ since there are 26 letters in the English alphabet, but it actually could be defined over any $\mathbb{Z}_m$

In the case of $\mathbb{Z}_{26}$:

$$0 \leq K \leq 25$$

$$e(x) = (x + K) mod 26$$

$$d(y) = (y - k) mod 26$$

So the encryption adds the key to the value giving to the character, and obtains modulo 26, so the character corresponding to the new integer value obtained will be part of the ciphertext.

For the descipher it is only necessary to substract the key to the value of the character in the ciphertext and apply to it modulo 26.

1.2 Affine Cipher

The Affine Cipher is a special case of the Substitution Cipher. In Affine Cipher, we restrict the encryption functions to functions of the form

$$e(x) = (ax + b) mod 26$$

a , $b \epsilon \mathbb{Z}_{26}$. These functions are called affine functions, hence the name Affine Cipher. (Observe that when $a = 1$, we have a Shift Cipher.)

In order that decruption is possible, it is necessary to ask when an affine function is injective. In other words, for any $y \epsilon \mathbb{Z}_{26}$, we want the congruence

$$ax + b \equiv y - b(mod 26)$$

Now, as y varies over $\mathbb{Z}_{26}$, so, too, does $y - b$ ary over $\mathbb{Z}_{26}$. Hence, it suffices to study the congruence

$$ax \equiv y(mod 26)(y \epsilon \mathbb{Z}_{26}).$$

We claim that this congruence has a unique solution for every y if and only if gcd(a, 26) = 1 (where the gcd function denotes the greatest common divisor of its arguments). First, suppose

that gcd(a, 26) = $d > 1$. Then the congruence $ax \equiv 0(\text{mod } 26)$ has (at least) two distinct solutions in $\mathbb{Z}_{26}$), namely $x = 0$ and $x = 26/d$. In this case $e(x) = ax + b)mod26$ is not an injective function and hence not a valid encryption function.

1.2.1 Definition

Let $P = C = \mathbb{Z}_{26}$ and let

$$K = (a, b)\epsilon\mathbb{Z}_{26} \text{ X } \mathbb{Z}_{26} : \text{gdc}(a, 26) = 1$$

For $K = (a, b)\epsilon K$ define

2 Programming Exercises

2.1 Exercise 1

2.1.1 Instructions

Consider that we are using the set of printable characters in ASCII as the alphabet to write plaintext. Modify your implementation to the affine cipher to consider this alphabet. Consider the following requirements.

- ✓ The values for a and b must be chosen by the user. Your program must check that a is a valid, using your implementation of the extended Euclidean algorithm.
- ✓ Your program must receive the plaintext in a file of any size (at least 5Kb) and the ciphertext must be store in a file with the same name that the file of the plaintext, but extension afn.
- ✓ Your program must be able to encrypt and decrypt. Also the descryption must work with files. Consider that you must run your program once to encrypt and you must re run it to decrypt.

2.1.2 How do we code?

In general, our program receives the keys and the text, sends those parameters to the corresponding function, (there is a function to encrypt and another to decipher). First,, when we receive a, we validate that this number. For it, we send the number to the function called **gcdRecursive**. If the result is 1, that means it a is a valid number. Here we explain each of the functions:

2.1.3 GCD Implementation

```
int gcdRecursive(int a, int b) {
   if(b == 0)
      return a;
   else {
      return gcdRecursive(b, a % b);
   }
}
```

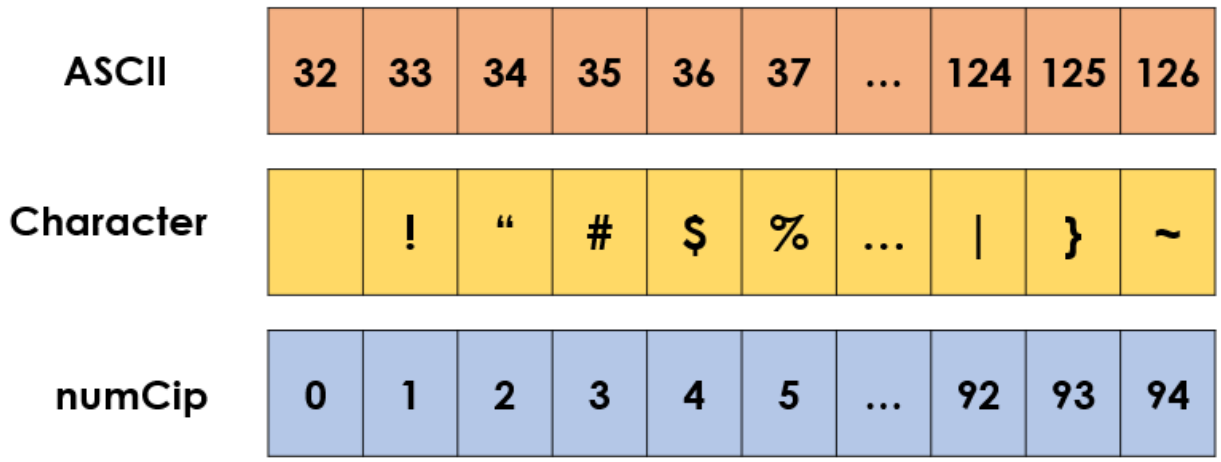
2.1.4 Greats Common Divisor

The algorithm is based on below facts.

If we subtract smaller number from larger (we reduce larger number), GCD doesn't change. So if we keep subtracting repeatedly the larger of two, we end up with GCD. Now instead of subtraction, if we divide smaller number, the algorithm stops when we find remainder 0.

2.1.5 Get Number

We receive a character and obtain the number in the array. To obtain the number of the arrangement, only we reduce 32 because in this number they begin the printable characters of the ASCII.



2.1.6 Get number Implementation

```
int getNumber(char letter) {
   int i = letter - 32;
   return i;
}
```

2.1.7 Affine Cipher

In this function we receive an integer, this number represents the cardinalidad of the alphabet, in this case is 95 cause we are working with the set of printable characters in ASCII, the second argument is the message that we want cipher, at last we receive keys We love using the computer's resources correctly, for that reason we use a programming technique called dynamic programming. We have an array called **numCip** that helps us save the value that is obtained from the character of position i.

First we fill the **numCip** array with -1, if the number in the position i of the **numCip** array is equal to -1, it means that the value we need has not been calculated, so we calculate it.

When we calculated, we use a function called getNumber to obtain the position of the arrangement to which the character belongs. With it, we obtain the number of the ciphered character with the operation:

$$\text{cipherCharacter} = (a * \text{numLetter} + b)$$

It means that we have our ciphered character.

2.1.8 Affine Cipher Implementation

```
1  void affineCipher(int n, string message, int a, int b) {
2     int i, numLetter, numCip[n];
3     char character = ' ';
4     for(i = 0; i < n; i++)
5        numCip[i] = -1;
6     // For each character
7     for(i = 0; i < message.length(); i++) {
8        numLetter = getNumber(message[i]);
9        if(numCip[numLetter] == -1) {
10          numCip[numLetter] = ((a * numLetter) + b) % n;
11       }
12       character = character + numCip[numLetter];
13       cout << character;
14       character = ' ';
15    }
16 }
```

2.1.9 Affine Decipher

In this function we receive an integer, this number represents the cardinalidad of the alphabet, in this case is 95 cause we are working with the set of printable characters in ASCII, the second argument is the message that we want cipher, at last we receive keys We love using the computer's resources correctly, for that reason we use a programming technique called dynamic programming. We have an array called **numCip** that helps us save the value that is obtained from the character of position i.

First we fill the **numCip** array with -1, if the number in the position i of the **numCip** array is equal to -1, it means that the value we need has not been calculated, so we calculate it.

When we calculated, we use a function called getNumber to obtain the position of the arrangement to which the character belongs. We obtain the inverse number. With it, we obtain the number of the ciphered character with the operation:

$$\text{cipherCharacter} = \text{mod}(\text{numLetter} - b * inverse, n)$$

It means that we have our original character.

2.1.10 Affine Decipher Implementation

```
void affineDecipher(int n, string cipher, int a, int b) {
   int i, numLetter, numDec[n], inverso;
   char character = ' ';
   for(i = 0; i < n; i++)
      numDec[i] = -1;
   // For each character
   for(i = 0; i <  cipher.length(); i++) {
      numLetter = getNumber(cipher[i]);
      if(numDec[numLetter] == -1) {
         inverso = inverse(n, a);
         numDec[numLetter] = numLetter - b;
         numDec[numLetter] = mod((numLetter - b) * inverso, n);
      }
      character = character + numDec[numLetter];
      cout << character;
      character = ' ';
   }
}
```

2.1.11 Modulo

This function is recursive, we have a base case that is mod(a, b) with a = 0, we know it is 0. If we have a positive a, we return a % b. In the special case if we have a negative a, we return the next operation:

$$b - \text{mod}(a * (-1), b)$$

2.1.12 Modulo Implementation

```
int mod(int a, int b) {
   if (a == 0)
      return 0;
   if(a > 0)
      return a % b;
   else
      return b - mod(a*(-1), b);
}
```

2.1.13 Extended Euclides Algorithm

For the implementation of the Extended Euclides Algorithm we use two functions, the first one is in charge of calculate the equations, and the other one is a recursive function that returns the inverse to the first one.

We defined the function inverse with a return value type int and, it receives two ints, values of $\mathbb{Z}$ and the number we want to get it's inverse.

At the beggining of the function we create an integer matrix where we'll locate the ecuations. It has 4 fields that contains the remainder, the divisor, the dividend and the quotient accordingly. For the calculation of the equations we used a do-while structure that breaks when the value of the variable res, that refers to the reminder, is equals to 0.

With all the equations calculated we verify if the second last reminder is equals to 1, in which case the given number has inverse, otherwise we return a value of -1. After that we recover the number of resulting equations and if there's only one, return the value of the inverse as 1. If there's more than 1 equation we proceed to call the recursive function extendedAlgorithm.

This other function returns an int and receives the matrix of equations and the number of them. It has as base case the fisrt one of the equations, which will return the value in the field corresponding to the quotinet, that we could understand as the number of times that the divisor will increase to reach the value of the dividend. In the second case we obtain the value returned from the base case and multiply it for the quotient of the equation that we are currently evaluating and add 1 to it. This will give us the number of times that the original divisor appears in the second equation. In the case of the third to the nth equation we multiply the quotient of the equation by the returnd value of the last case, and then we add to this value the quotient of the equation before the last one. This will guve us the total number of times that the divisor appears in the currently evaluating equation, and this value will be returned to the first function.

2.1.14 Extended Euclides Algorithm Implementation

```
int inverse(int z, int a) {
   int res = 0, cont = 0, cociente = 0, inverso = 0, val_z_inicial = 0;

   //Matriz donde se guardarán las ecuaciones generadas
   int **ecuaciones = (int**)malloc(sizeof(int*));
   //Obtención de ecuaciones a partir del Algoritmo de Euclides
   val_z_inicial = z;
   do {
      res = z % a;
      cociente = z / a;
      ecuaciones[cont] = (int*)malloc(sizeof(int) * 4);
      ecuaciones[cont][0] = res;
      ecuaciones[cont][1] = a;
      ecuaciones[cont][2] = z;
      ecuaciones[cont][3] = cociente;
      z = a; a = res; cont++;
   }while(res != 0);

   //Sustitución de ecuaciones a partit del Algoritmo Extendido de Euclides
   cont -= 2;
   if(ecuaciones[cont][0] == 1) {
      if(cont == -1)
         inverso = 1;
      else
         inverso = extendedAlgorithm(ecuaciones, cont);
         if((cont % 2) == 0)
            inverso = val_z_inicial - (inverso % val_z_inicial);
      return inverso;
   }
```

```
30      else
31         return -1;
32   }
33
34   //Función recursiva
35   int extendedAlgorithm(int **ecuaciones, int numEc) {
36      //Variable aux recupera el valor acumulado de b's hasta el momento en el caso de la ecuación 1
37      int aux = 0, cont = 0;
38      //Tratándose de la primera ecuación
39      if(numEc == 0)
40         return ecuaciones[numEc][3];
41      //Tratándose de la segunda ecuación
42      else if(numEc == 1) {
43         aux = extendedAlgorithm(ecuaciones, numEc-1);
44         return ecuaciones[numEc][3] = aux * ecuaciones[numEc][3] + 1;
45      }
46      //En el caso de las demás ecuaciones
47      else {
48         aux = extendedAlgorithm(ecuaciones, numEc-1);
49         ecuaciones[numEc][3] = ecuaciones[numEc][3] * ecuaciones[numEc-1][3];
50         ecuaciones[numEc][3] = ecuaciones[numEc][3] + ecuaciones[numEc-2][3];
51         return ecuaciones[numEc][3];
52      }
53   }
```

2.2 Exercise 2

2.2.1 Instructions

Another kind of substitution encryption algorithm is to use a keyword, for example **CIPHER**. This corresponds to the numerical equivalent $K = (2, 8, 15, 7, 4, 17)$. If we are using the English alphabet and the plaintex is **thiscryposystemisnotsecure**, we can convert the plaintext to elements in $\mathbb{Z}_{26}$, write them in groups of six (the length of the keyword) and then, add the keyword modulo 26, as follows:

plaintext:	19	7	8	18	2	17	24	15	19	14	18	24
kerword:	2	8	15	7	4	17	2	8	15	7	4	17
ciphertext:	21	15	23	25	6	8	0	23	8	21	22	15

If we find the alphabetic equivalent of the ciphertext we will have: **VPXZGIAXIVWP**.

- ✓ How do we decrypt in this case?
- ✓ Design a program to ecrypt and decrypt using this encryptation algorithm, but now consider that the alphabet is the set of printable characters in ASCII.
- ✓ The user must be able to choose the keyword.
- ✓ Plaintext and ciphertext must be stored in a textfile (size must be at least 5KB)

2.2.2 How do we code?

Our program receives a file which contains the key word in the first line, and all the text to encrypt or decrypt depending on the case. We get the files of input and output from the comand line. And depending of what we want we will use the functionf of cipher or decipher.

2.2.3 Key Word Cipher

In the main function we declared two string, which will get the key word and the text in the file. Then we have the functions to encrypt and decrypt. Each one of them returns the text, in the fist case the ciphertext and in the other the plaintext, and receive the strings corresponding to the key and the text. We obtain the size of the string that contains the key. After that, starts a for loop that breaks till the end of the message it's reached. In every loop it's obtained the particular value of the character in the message, substracting the value 32 to it so we can work from the values from 0 to 95, which is the range of the printable characters in the ASCII code. The same process is done with the character in the key word, but to know which word turn is we calculate the modulo tamKey (which is the size of the key word) of the value that has our counter.

For the cipher function we calculate the new value on the ciphertext adding the value of the corresponding character of the key word to the value of the character in the message and then calculate modulo 95 ('cause is the total number of printable characters) to finally concatenate the character with our string that contains the ciphertext, and once finished the loop return it.

2.2.4 Key Word Cipher implementation

```
1  string cipher(string key, string msj){
2     //Obtención del mensaje caracter a caracter
3     int cont3, nuevo_val = 0, aux = 0, n = 95, tam_key = 0, val_c = 0;
4     string msj_cip;
5     char aux_msj;
6     //Tamaño de la palabra clave
7     tam_key = key.size();
8     for(cont3 = 0; cont3 < msj.size(); cont3++){
9        val_c = (int)msj[cont3] - 32;
10       aux = (int)key[cont3 % tam_key] - 32;
11       //Se obtiene el nuevo valor para la letra
12       nuevo_val = (val_c + aux) % n;
13       nuevo_val += 32;
14       aux_msj = (char)nuevo_val;
15       msj_cip += aux_msj;
16    }
17    return msj_cip;
18 }
```

2.2.5 Key Word Decipher

For the decipher function we do the same, but for the new value, instead of adding the value of the key to the character, we substract it, and after that we use an if-else structure to evaluate if

the calculation of the new value was possitive or negative. In the first case we calculate modulo 95 of the new value, an in the other case we also calculate modulo 95 of the value but applying the function:

$$-a(mod n) = n - (a(mod n))$$

Finally we concatenate de character to the string that contains our plaintext and return it.

2.2.6 Key Word Decipher implementation

```
1  string descipher(string key, string msj){
2     //Obtención del mensaje caracter a caracter
3     int cont3, nuevo_val = 0, aux = 0, n = 95, tam_key = 0, val_c = 0;
4     string msj_descip;
5     char aux_msj;
6     //Tamaño de la palabra clave
7     tam_key = key.size();
8     for(cont3 = 0; cont3 < msj.size(); cont3++){
9        val_c = (int)msj[cont3] - 32;
10       aux = (int)key[cont3 % tam_key] - 32;
11       //Se obtiene el nuevo valor para la letra
12       nuevo_val = val_c - aux;
13       if(nuevo_val < 0)
14          nuevo_val = n + (nuevo_val % n);
15       else
16          nuevo_val = nuevo_val % n;
17       nuevo_val += 32;
18       aux_msj = (char)nuevo_val;
19       msj_descip += aux_msj;
20    }
21    return msj_descip;
22 }
```