Práctica 3
Cifrado Afín y Hill

Códigos y Criptografía
Curso 2023/24
Ingeniería Informática UCO

Alumno:	Cáceres Gaitán, Pablo
Email:	i12cacgp@uco.es

Índice

1. Introducción

En esta práctica se implementan dos métodos de cifrado: Afín y Hill. Además, se incluyen funciones para el calculo con aritmética modular y matrices en el fichero *funciones.py*.

Dado que la base teórica ha sido estudiada en clase no será incluida en este documento.

2. Ejemplo de ejecución

2.1. Modo Debug

Para facilitar el desarrollo se incluyó un modo de depuración que puede ser accedido mediante la flag *-v* al ejecutar cada script, para los ejemplos asumiremos como desactivado dicho modo.

Un ejemplo para activar el modo de depuración para el cifrado afín:

```
$ python3 afin.py -v
```

2.2. Cifrado Afín

Comenzamos lanzando el programa *afin.py* con *python3*:

```
$ python3 afin.py
```

Deberíamos ver el menú principal:

```
--- CIFRADO AFIN ---
1. Ingresar k y d
2. Generar k y d aleatorios
3. Cifrar mensaje
4. Descifrar mensaje
5. Salir
Elige una opción:
```

2.2.1. Elección de la clave

En este punto, podemos elegir una clave (k y d) propia o podemos dejar que el programa elija aleatoriamente. Para esta demostración elegiremos la opción aleatoria:

```
--- CIFRADO AFIN ---
1. Ingresar k y d
2. Generar k y d aleatorios
3. Cifrar mensaje
4. Descifrar mensaje
5. Salir
Elige una opción: 2
Valores generados: k = 19, d = 2
```

2.2.2. Cifrado

Tras generar la clave, podemos utilizarla para cifrar, para ello debemos elegir la opción 3:

```
--- CIFRADO AFIN ---
1. Ingresar k y d
2. Generar k y d aleatorios
3. Cifrar mensaje
4. Descifrar mensaje
5. Salir
Elige una opción: 3
Introduce el mensaje a cifrar: Esto es una prueba
Mensaje cifrado: 24201225##2420##040602##170104242102
```

La clave se guarda internamente, por lo que sólo tendremos que introducir el mensaje.

2.2.3. Descifrado

Finalmente, escogeremos la opción 4, a la que le daremos el mensaje cifrado (tal cual nos lo da la función de cifrado), ya que la clave se guarda internamente:

```
--- CIFRADO AFIN ---
1. Ingresar k y d
2. Generar k y d aleatorios
3. Cifrar mensaje
4. Descifrar mensaje
5. Salir
Elige una opción: 4
Introduce el mensaje cifrado a descifrar: 24201225##2420##040602##170104242102
Mensaje descifrado: esto#es#una#prueba
```

Como se esperaba, nos devuelve el texto en claro. Los espacios o posibles caracteres especiales se convierten a #.

2.3. Cifrado Hill

Para trabajar con cifrado Hill lanzaremos el programa *hill.py*:

```
$ python3 hill.py
```

Se nos mostrará el menú principal:

```
===== Cifrado Hill ======
1. Establecer clave
2. Cifrar mensaje
3. Descifrar mensaje
4. Salir
Elige una opción:
```

2.3.1. Elección de la clave

Al igual que antes, al seleccionar la opción 1 podremos introducir nuestra propia matriz ó generarla aleatoriamente. Para la demostración, la generaremos de forma aleatoria:

```
===== Cifrado Hill - Generar Clave ======
1. Aleatoria
2. Introducir manualmente
3. Volver
[*] Elige una opción: 1
[*] Elige el tamaño de la matriz: 2
[+] Clave: [[21, 16], [22, 7]]
```

2.3.2. Cifrado

Como antes, la clave se guardará internamente, así que para cifrar sólo debemos escoger la opción 2 e introducir el mensaje:

```
===== Cifrado Hill ======
1. Establecer clave
2. Cifrar mensaje
3. Descifrar mensaje
4. Salir
Elige una opción: 2
Introduce el mensaje a cifrar: esto es una prueba
[+] Mensaje cifrado: vzcdkbashsaaurzjvw
```

2.3.3. Descifrado

De igual forma, para descifrar sólo debemos escoger la opción 3 e introducir el mensaje cifrado:

```
===== Cifrado Hill ======
1. Establecer clave
2. Cifrar mensaje
3. Descifrar mensaje
4. Salir
Elige una opción: 3
[*] Introduce el mensaje cifrado a descifrar: vzcdkbashsaaurzjvw
[+] Mensaje descifrado: estoaesaunaaprueba
```

En este caso, los caracteres especiales como los espacios se convierten en a (deberían ser $\#$, pero no se ha podido implementar por falta de tiempo).

3. Análisis del código fuente

Aquí analizaremos funciones incluidas en el código, aunque no se incluyan, todas las funciones están documentadas internamente.

3.1. Dependencias

En *funciones.py*, podemos ver que se hace uso de *random* y *argparse* como únicas librería externa, para generar números aleatorios y para gestionar los argumentos para el modo de depuración respectivamente. A su vez, *funciones.py* es importada por los dos programas principales:

Listing 1: funciones.py

```
import random
import argparse
```

Listing 2: Programas principales

```
from funciones import *
```

3.2. Funciones Auxiliares

Como hemos mencionado, en *funciones.py* se incluyen las funciones necesarias para implementar los métodos de cifrado.

Además se definen dos macros: *MODULO*, que guarda el módulo en que trabajarán los algoritmos de cifrado; y *debug* que indica si el modo de depuración está activado. Por defecto son 27 y 0 respectivamente.

3.2.1. Aritmética Modular

Entre otras se implementa una función *invmod* que calcula el inverso modular de un número mediante el algoritmo Extendido de Euclides:

Listing 3: Algoritmo Extendido de Euclides

```
def invmod(p, n):
    r = [n, p]
    s = [0, 1]

    count = 0
    while r[-1] != 0 and r[-1] != 1:
        q = r[count] // r[count + 1]
        r.append(r[count] % r[count + 1])
        s.append(s[count] - q * s[count + 1])
        count += 1

    if r[-1] == 0:  # No existe inverso
        if debug: print(f"[INFO] No existe inverso.")
        return None
    else:  # Existe inverso
        inv = s[-1] % n
        if debug: print(f"[DEBUG] El inverso es {inv}")
        return inv
```

3.2.2. Calculo con Matrices

Entre otras, implementamos una función que nos permite calcular el determinante modular de una matriz:

Listing 4: Determinante modular

```
def determinante_modular(matriz, n):
    if len(matriz) == 1:
        return matriz[0][0] % n
    det = 0
    for j in range(len(matriz)):
        menor = matriz_menor(matriz, 0, j)
        signo = (-1) ** j
        det += signo * matriz[0][j] * determinante_modular(menor, n)
    return det % n
```

La función ***InvModMatrix***, utiliza la anterior además de otras para calcular la inversa modular de una matriz:

Listing 5: Función knapsacksol

```
def InvModMatrix(matriz, n):

    if matriz is None or len(matriz) == 0:
        print(f"[ERROR] La matriz no esta definida")
        exit(-1)

    if cuadrada(matriz) == 0:
        print(f"[ERROR] La matriz no es invertible (No cuadrada)")
        exit(-1)

    det = determinante_modular(matriz, n)

    if det == 0:
        print(f"[ERROR] La matriz no es invertible (Det=0)")
        exit(-1)

    if algeucl(det, n)!=1:
        print(f"[ERROR] La matriz no es invertible (GCD(det,n)!=1)")
        exit(-1)

    det_inv = invmod(det, n)
    adjunta = matriz_adjunta(matriz, n)

    inversa_mod = []
    # Iteramos por cada fila de la matriz adjunta
    for i in range(len(adjunta)):
        fila_inversa = []  # Creamos una lista para la fila actual
        for j in range(len(adjunta[i])):

            # Calculamos el valor modular para el elemento actual
            elemento_mod = (det_inv * adjunta[i][j]) % n

            # Anadimos el elemento a la fila
            fila_inversa.append(elemento_mod)
```

```
35          inversa_mod.append(fila_inversa)
36
37      return inversa_mod
```

3.2.3. Conversión de texto

La última función, ***commonfactors***, nos permite saber si algún número tiene factores primos en común con algún elemento de una mochila:

Listing 6: Función commonfactors

```
1  def TexttoNumber(a):
2      result = []
3      for char in a.lower():
4          if 'a' <= char <= 'z':
5              num = ord(char) - ord('a')
6              # Asegurate de que cada numero tiene dos digitos,
7              # agrega un cero si es necesario
8              result.append(f"{num:02}")
9          else: result.append('27')
10     return ''.join(result)
```

La función se asegura de que cada letra se representa con dos dígitos. Para la conversión inversa se realiza el proceso inverso.

3.3. Cifrado Afín

En este caso tenemos una función para cifrado y otra para descifrado:

Listing 7: Función de cifrado

```
def Afincypher(text, k, d):

    tocipher = TexttoNumber(text)

    ciphered = ""
    # Procesa la cadena en bloques de 2 caracteres
    for i in range(0, len(tocipher), 2):

        num = int(tocipher[i:i+2])  # Convierte el bloque a numero

        if 0 <= num <= 26:
            num = (num*k + d) % MODULO

            # Aplica el cifrado afin f(x)=kx+d mod MODULO
            ciphered += str(num).zfill(2)

        else: ciphered += '##'
    return ciphered
```

Listing 8: Función de descifrado

```
def Afindecypher(text, k, d):

    inv_k = invmod(k, MODULO)

    deciphered = ""
    # Procesa la cadena en bloques de 2 caracteres
    for i in range(0, len(text), 2):

        if text[i:i+2] == '##': num = 27
        else: num = int(text[i:i+2])  # Convierte el bloque a un numero

        if 0 <= num <= 26:
            num = ((num - d) * inv_k) % MODULO

            # Aplica el descifrado afin f(x)=kx+d mod MODULO
            deciphered += str(num).zfill(2)

        else: deciphered += '##'

    return deciphered
```

3.4. Cifrado Hill

De forma similar, para el cifrado Hill también tenemos una función para cifrado y otra para descifrado:

Listing 9: Función de cifrado

```
def hillcipher(text, key_matrix):

    n = cuadrada(key_matrix)

    if n == 0:
        raise ValueError("La clave debe ser una matriz cuadrada.")

    text_numbers = TexttoNumber(text)

    # Creamos una lista con los numeros
    bloques = [int(text_numbers[i:i + 2])
               for i in range(0, len(text_numbers), 2)]

    while len(bloques) % n != 0:
        text_numbers.append(27)  # Padding con '#'

    encrypted = []
    for i in range(0, len(bloques), n):

        block = bloques[i:i + n]

        # Inicializamos el bloque cifrado como una lista vacia
        result = []
        # Iteramos por cada fila de la matriz clave
        for k in range(n):
            suma = 0
            # Iteramos por cada columna
            for j in range(n):
                # Multiplicamos y acumulamos
                suma += block[j] * key_matrix[k][j]
            suma_mod = suma % MODULO

            # Anadimos el valor cifrado al bloque
            #(asegurando que tenga 2 cifras)
            result.append(str(suma_mod).zfill(2))

        encrypted.extend(result)

    return NumberstoText("".join(encrypted))
```

Listing 10: Función de descifrado

```python
def hilldecipher(encrypted_text, key_matrix):

    inverse_key = InvModMatrix(key_matrix, MODULO)

    n = len(inverse_key)

    # Tratamiento de la entrada
    encrypted_numbers = TexttoNumber(encrypted_text)

    # Creamos una lista con los numeros
    bloques = [int(encrypted_numbers[i:i + 2])
               for i in range(0, len(encrypted_numbers), 2)]

    decrypted = []
    for i in range(0, len(bloques), n):

        block = bloques[i:i + n]

        # Inicializamos el bloque cifrado como una lista vacia
        result = []
        # Iteramos por cada columna de la matriz clave
        for k in range(n):
            # Calculamos la suma de productos para la fila actual
            suma = 0
            for j in range(n):
                suma += block[j] * inverse_key[k][j]

            # Reducimos el resultado modulo MODULO
            suma_mod = suma % MODULO

            # Lo anadimos al mensaje descifrado
            # asegurando que tenga 2 cifras
            result.append(str(suma_mod).zfill(2))

        if debug: print(f"[DEBUG] decrypted block={result}")

        decrypted.extend(result)

    return NumberstoText("".join(decrypted))
```