Proyecto 1
Criptografía
Grupo 3
Prof.ª: Ing. Magdalena Reyes Granados

Integrantes:
González Sánchez Rodrigo
Landázuri Brambila Álvaro Ulises
Lona Reveles Iván

15 de octubre del 2021

Índice

1. Introducción

La información tiene demasiado valor para todas las personas y debido a esto el ser humano se ha visto en la necesidad de idear formas de cómo protegerla, por lo que han surgido una gran diversidad de técnicas y algoritmos que la protegen cifrando y descifrando nuestros mensajes para hacerlos lo menos vulnerables posibles y mantener su confidencialidad. Evidentemente, han surgido algoritmos más complejos que otros y muchos han sido sustituidos por otros más avanzados, puesto que conforme estos avanzan también lo hacen las técnicas de las personas que buscan corromper nuestra seguridad, sin embargo, en este proyecto nos centramos en el desarrollo de los siguientes 3 algoritmos:

- *Vernam*
- *Columnas*
- *Afín*

2. Justificación

El presente trabajo se desarrolló con el objetivo de llevar los conocimientos adquiridos de forma teórica a la práctica mediante la programación de los tres algoritmos que se abordan. Mediante cada algoritmo se busca realizar tanto el cifrado como el descifrado del contenido de un archivo para lo cual se ha creado un ejecutable donde se puede apreciar la ejecución de esto en conjunto.

Cabe mencionar que el lenguaje que se decidió utilizar fue Python, puesto que se adaptaba mejor a los conocimientos de los integrantes y se cree que su versatilidad lo hace eficiente para desarrollar este tipo de aplicaciones.

3. Desarrollo

Se utilizó un paradigma de programación orientada a objetos para desarrollar la aplicación. La funcionalidad se distribuyó en tres clases principales: Vernam, Columnas y Afín. Todas las clases cuentan con métodos para cifrar y descifrar, mientras que la configuración se realiza de dos maneras:

- Usando el constructor de la clase
- Empleando los *getters* de cada clase

3.1. Vernam

Para lograr cifrar un mensaje de acuerdo al algoritmo Vernam se requieren seguir los siguientes pasos:

1. Obtener los códigos ASCII para cada carácter del mensaje en claro Mcla.

2. Obtener los códigos ASCII para cada carácter de la llave K.

3. Realizar la operación XOR (*or* exclusiva) entre los bits obtenidos en **1** y **2**. Se obtiene el criptograma en ASCII como resultado.

Por otro lado, para descifrar un mensaje es prácticamente el mismo proceso, usando el criptograma en vez del mensaje en claro como insumo:

1. Obtener los códigos ASCII para cada carácter del criptograma Cripto.

2. Obtener los códigos ASCII para cada carácter de la llave K.

3. Realizar la operación XOR (*or* exclusiva) entre los bits obtenidos en **1** y **2**. Se obtiene el mensaje en claro en formato ASCII como resultado.

A partir de este diseño de interfaz es que se planteó la elaboración de la aplicación real.

3.2. Columnas

Para el cifrado por columnas se necesita el mensaje en claro y una llave, la llave debe estar formada únicamente por letras y el tamaño de esta definirá el numero de columnas que usaremos. Acomodaremos el mensaje una letra por columna, escribiendo por filas, ninguna fila puede quedar incompleta así que lo que haga falta lo rellenaremos con 'x', hay que mencionar que el nombre de cada columna estará dado por la letra de la palabra llave, así que reorganizaremos la tabla ordenando las columnas por el alfabeto. Finalmente de la misma forma que ingresamos el mensaje, ahora lo retiramos de la tabla y ese será nuestro mensaje cifrado.

3.3. Afín

4. Implementación

Como se mencionó con anterioridad, se utilizó el lenguaje de programación *Python* para implementar la aplicación.

La estructura de archivos final es la siguiente:

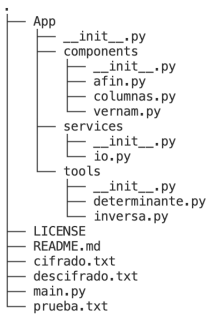

Figura 1: Estructura de archivos

4.1. Vernam

4.2. Columnas

```
def cifrar(self, cadena, mcla):
    long_cadena=len(cadena)
    long_mcla=len(mcla)
    Lista_aux=[]
    aux=int(long_mcla/long_cadena)+1
    for i in range(aux):
        Lista_aux.append([])
    for i in range(aux):
        for j in range(long_cadena):
            if long_mcla>(i*long_cadena)+j:
                Lista_aux[i].append(mcla[(i*long_cadena)+j])
            else:
                Lista_aux[i].append('x')
    print(Lista_aux)
    cripto=''
    lista_trans_cif=transpuesta(Lista_aux)
    print(lista_trans_cif)
    cadena_sort = sorted(cadena)
    lista_trans_cif2=[]
    for i in range(long_cadena):
        indice=cadena.index(cadena_sort[i])
        lista_trans_cif2.append(lista_trans_cif[indice])
    print(lista_trans_cif2)
    lista_cif=transpuesta(lista_trans_cif2)
    print(lista_cif)
    cripto=reconstruccion(lista_cif)
    print(cripto)
    return cripto
```

Figura 2: Cifrado por columnas

```
def descifrar(self, key, cripto):
    key_sort = sorted(key)
    long_key=len(key)
    long_cripto=len(cripto)
    Lista_aux=[]
    aux=int(long_cripto/long_key)
    for i in range(aux):
        Lista_aux.append([])
    for i in range(aux):
        for j in range(long_key):
            Lista_aux[i].append(cripto[(i*long_key)+j])
    listatrans=transpuesta(Lista_aux)
    listatrans2=[]
    for i in range(long_key):
        indice=key_sort.index(key[i])
        listatrans2.append(listatrans[indice])
    lista2=transpuesta(listatrans2)
    mcla = reconstruccion(lista2)
    while mcla[len(mcla)-1]=='x':
        print(mcla)
        mcla=mcla[0:len(mcla)-1]
    return mcla
```

Figura 3: Cifrado por columnas

```
def cifrar(self, cadena, mcla):
    long_cadena=len(cadena)
    long_mcla=len(mcla)
    Lista_aux=[]
    aux=int(long_mcla/long_cadena)+1
    for i in range(aux):
        Lista_aux.append([])
    for i in range(aux):
        for j in range(long_cadena):
            if long_mcla>(i*long_cadena)+j:
                Lista_aux[i].append(mcla[(i*long_cadena)+j])
            else:
                Lista_aux[i].append('x')
    print(Lista_aux)
    cripto=''
    lista_trans_cif=transpuesta(Lista_aux)
    print(lista_trans_cif)
    cadena_sort = sorted(cadena)
    lista_trans_cif2=[]
    for i in range(long_cadena):
        indice=cadena.index(cadena_sort[i])
        lista_trans_cif2.append(lista_trans_cif[indice])
    print(lista_trans_cif2)
    lista_cif=transpuesta(lista_trans_cif2)
    print(lista_cif)
    cripto=reconstruccion(lista_cif)
    print(cripto)
    return cripto
```

Figura 4: Cifrado por columnas

4.3. Afín

El constructor de la clase es el siguiente:

```
def __init__(self, k = "contrasena"):
    self._k = k

def get_k(self):
    return self._k

def set_k(self, k):
    self._k = k
```

Figura 5: Constructor de la clase Vernam

A través del constructor se define la llave a emplear por el algoritmo.
El método para cifrar es el siguiente:

```
def cifrar(self, cadena):
    # Se inicializa el criptograma como una cadena vacía
    cripto = ""
    # Se usa un contador para recorrer la llave
    i = 0
    # Se recorre la cadena a cifrar caracter por caracter
    for c in cadena:
        # Se va añadiendo al criptograma el resultado de aplicar
        # la operación XOR (^) entre los códigos ASCII, calculados
        # con ord(), de cada caracter de la cadena y llave. Este resultado
        # se convierte a un caracter antes de agregarse al criptograma.
        cripto += chr(ord(c) ^ ord(self._k[ i % len(self._k)]))
        # Se suma uno al contador que recorre la llave
        i += 1
    # Se regresa el criptograma
    return cripto
```

Figura 6: Cifrado en la clase Vernam

Como se puede apreciar, recibe como parámetro la cadena a cifrar. Obtiene los códigos ASCII de cada carácter del mensaje y llave usando la función *ord()*, emplea el símbolo ˆpara denotar la operación XOR y obtiene el caracter asociado al código ASCII resultante con la función *chr()*.

Por último, para descifrar se emplea el siguiente método:

```
def descifrar(self, cadena):
    # Se inicializa el mensaje en claro como una cadena vacía
    mcla = ""
    # Se usa un contador para recorrer la llave
    i = 0
    # Se recorre la cadena a descifrar caracter por caracter
    for c in cadena:
        # Se va añadiendo al criptograma el resultado de aplicar
        # la operación XOR (^) entre los códigos ASCII, calculados
        # con ord(), de cada caracter de la cadena y llave. Este resultado
        # se convierte a un caracter antes de agregarse al criptograma.
        mcla += chr(ord(c) ^ ord(self._k[ i % len(self._k)]))
        # Se suma uno al contador que recorre la llave
        i += 1
    # Se regresa el mensaje descifrado
    return mcla
```

Figura 7: Descifrado en la clase Vernam

En este método se realiza el mismo procedimiento, pero usando el criptograma en vez del mensaje en claro para aplicar la operación XOR.

5. Conclusiones

Después del desarrollo de este proyecto, creemos que se han conseguido los objetivos que se tenían planteados para el mismo debido a que se ha logrado desarrollar cada uno de los tres algoritmos correspondientes y consideramos que esto significó un gran aprendizaje, puesto que programar cada uno involucró estudiar y analizar cada uno de sus pasos para poder conseguirlo.

Por otro lado, reconocemos que tuvimos ciertas dificultades para programar cada aspecto de los algoritmos, sin embargo, fue satisfactorio poderlos resolver y terminar el proyecto de buena forma.

6. Referencias

- Anónimo. (Sin año). Python Cheat Sheet. Recuperado de https://blog.finxter.com/python-bitwise-xor-operator/
- Elanotta, M. (2021). How to Create an executable from a Python program. Recuperado de http://www.ordenjuridico.gob.mx/Documentos/Federal/wo15528.pdf
- Parewa Labs Pvt. (Sin año). Python Program to Find ASCII Value of Character. Programiz. Recuperado de https://www.programiz.com/python-programming/examples/ascii-character