

**Ingeniería Matemática e Inteligencia Artificial**

**Memoria – Práctica 2**

Criptografía

Asignatura de Matemática Discreta

GP04

Alejandro Martínez de Guinea García - 202113492

Javier Prieto Domínguez - 202103783

**Índice**

[**1.** **Introducción** 2](#_Toc117840549)

[**2.** **CRIPTOCHAT** 3](#_Toc117840550)

[2.1. INTERFAZ 3](#_Toc117840551)

[2.2. CHECK\_INTEGER 4](#_Toc117840552)

[2.3. HANDLER\_SIGNAL 4](#_Toc117840553)

[**3.** **MODULAR.PY** 5](#_Toc117840554)

[**4.** **RSA.PY** 6](#_Toc117840555)

[4.5. ATAQUE\_TEXTO\_PLANO 7](#_Toc117840556)

[**5.** **TEXTO PLANO RECUPERADO** 8](#_Toc117840557)

[**6.** **BIBLIOGRAFÍA** 9](#_Toc117840558)

# **Introducción**

En nuestra segunda práctica de Matemática Discreta se nos pedía hacer un programa que permitiese encriptar y desencriptar mensajes dadas unas claves públicas y privadas. Para ello, se pedía la implementación de funciones que permitan generar claves, cifrar y descifrar mensajes y, romper claves.

Al comenzar a programar cada función primero hacíamos un algoritmo básico, intentando ceñirnos a los conocimientos aprendidos en clase. Posteriormente a eso, tratamos de optimizar dichas funciones de cara a poder realizar operaciones más largas y complejas. Para ello, hemos mejorado también algunas de las funciones de la librería “modular.py” de la primera práctica.

# **CRIPTOCHAT**

## INTERFAZ

Al correr criptochat.py se muestra un menú que permite realizar las diferentes funciones:

* Generar nuevas claves: el usuario debe introducir dos límites entre los cuales se buscarán los dos números primos que formarán la de la clave pública (, ).
* Registrar unas claves pública y privada para el usuario: debe introducir las claves que usará para descifrar los mensajes.
* Registrar una clave pública del destinatario: el usuario debe introducir la clave pública del destinatario al que va a enviar el mensaje para poder cifrarlo.
* Cifrar mensaje: el usuario introducirá el mensaje que quiere enviar y este se cifrará con la clave pública registrada del destinatario.
* Descifrar mensaje: el usuario introducirá el mensaje recibido para descifrarlo con la clave privada previamente generada/registrada.
* Cambiar número de cifras de padding: el usuario tiene la opción de introducir dígitos padding para hacer que su cifrado sea más sofisticado y difícil de romper.
* Salir del programa

Una vez el usuario elige la opción que desea realizar, esta se ejecuta y posteriormente se vuelve a mostrar el menú para elegir de nuevo otra opción.

Ahora pasamos a explicar como se ejecuta cada opción.

Generar claves

En esta opción se le pide al usuario que introduzca dos números enteros a través de inputs, primero el límite inferior y luego el superior. La comprobación de las entradas se ejecuta en una función explicada más adelante llamada “check\_integer”. En caso de ser incorrecta la entrada (es decir, que no sea un número entero), se mostrará el mensaje “*Por favor, introduzca un número entero correcto.”*, y se le pedirá de nuevo el input. Una vez los números son válidos, se pasa a generar las claves. En caso de error durante este proceso, se mostrará el mensaje en pantalla *“No se han podido generar las claves.”*, y se volverá al menú. En caso contrario, se le informa al usuario que se han generado las claves correctamente y se le muestran la clave pública y privada.

Registrar claves (tanto públicas como privadas, es decir, opciones 2 y 3)

Estas opciones permiten al usuario introducir las claves manualmente. De la misma manera que en la anterior, se le pide que introduzca los números a través de inputs y se hace la comprobación con “check\_integer”. De nuevo, en caso de ser incorrecta la entrada, se le volverá a pedir al usuario que lo introduzca. Una vez registradas las claves, se guardan en sus respectivas variables (*“clave\_privada”* para las claves del usuario y *“clave\_publica”* para la clave del destinatario) y se da paso al menú.

Cifrar y descifrar mensaje (opciones 4 y 5)

En estas opciones, se le pide al usuario que introduzca un mensaje y este se cifra o descifra según sea conveniente. Para el cifrado, la función usa los datos de la variable “clave\_publica”, y transforma el mensaje para posteriormente ser mostrado en pantalla. Para el descifrado, la función usa los datos de la variable “clave\_privada” que permiten convertir el mensaje cifrado a mensaje original. En caso de existir algún error durante este proceso, se muestra en pantalla el siguiente mensaje: *“Error al descifrar el mensaje.”*. Esto puede deberse a varias razones, por ejemplo, el mal registro de las claves (en caso de haberlas registrado manualmente), las cuales pueden ser erróneas ya que no cumplen las propiedades que deben cumplir, o por ejemplo, la mala introducción del mensaje en el input (escribir alguna letra, espacio extra, etc.).

Cambiar padding

Al igual que con las 3 primeras opciones, en esta opción se le pide al usuario que introduzca un número entero a través de un input que determinará el número de cifras de padding añadidas al final de cada mensaje para cifrar. De nuevo, se hace la comprobación con “check\_integer”, y se pide de nuevo el input en caso de error. Una vez registrado dicho número, se guarda en la variable “padding”.

## CHECK\_INTEGER

Esta función se encarga de comprobar si una entrada de input (string) es un número entero. Para ello, asumiendo que la entrada es un número, se convierte dicho string en entero y en flotante y se comparan para ver si son iguales. En caso de serlo, la entrada es correcta. Sin embargo, si no lo son significa que la entrada es un número flotante, por lo que no es válido.

Para controlar que la entrada es un número, se pone un *try* de manera que al parsear a entero el string, no dé error el programa, sino que pida otro número en su lugar.

## HANDLER\_SIGNAL

Esta función se encarga de controlar la interrupción intencionada por teclado *“Ctrl+C”*. Al ejecutar dicha acción, el programa se interrumpe mostrando un error por pantalla. En este caso, evitamos que se muestre dicho error, y en su lugar, mostramos el mensaje *“Saliendo por interrupción…”*.

# **MODULAR.PY**

En la librería modular.py hemos realizado cambios para asegurarnos que ninguna función da error.

En primer lugar, hemos modificado la función factorizar para que solo ejecutase el algoritmo de Rho Pollard en caso de que el número a factorizar fuese mayor que 10000. De esta forma, nos hemos asegurado de que ejemplos como “factorizar(4)” no nos daba error.

También hemos arreglado la función “es\_primo”. Antes era determinista con el algoritmo de Miller-Rabin hasta 64 bits. Como para esta práctica hemos necesitado calcular primos más grandes, hemos asumido como cierta la hipótesis de Riemann de manera que pudiésemos ejecutar una versión determinista para cualquier número , y tomando como base los primos hasta comprobamos la primalidad de un número.

# **RSA.PY**

* 1. GENERAR\_CLAVES

Inicialmente este algoritmo tomaba los límites entre los que los primos podían estar comprendidos y ejecutaba el comando de la librería modular “lista\_primos”, de la que cogíamos 2 números primos para calcular la clave pública y privada. Rápidamente vimos que no era óptima y decidimos cambiar la manera en la que cogíamos 2 primos.

Creamos una nueva función en rsa.py llamada “generar\_primos” que no devuelve una lista de todos los primos, ya que es una operación que necesita mucha memoria. Devuelve solamente 2 primos dentro del intervalo para calcular con ellos claves. Coge un número aleatorio dentro del intervalo y comprueba si es primo o no. Si no lo es le suma 1 y vuelve a comprobarlo. Así hasta encontrarlo. Si llega al máximo del intervalo, empieza desde el límite inferior. Si llega hasta el número aleatorio inicial, devuelve que no existen primos dentro del intervalo. Hacemos el mismo proceso para el segundo primo, comprobando también que no sea el mismo número que el primer primo.

Últimamente, comprobamos que el seleccionado es comprimo con phi(n). De no ser así, aumentamos en dos, hasta que sea coprimo.

Con la nueva función de generar primos, no solo nos genera claves de 20 dígitos rápidamente, si no que nos genera hasta claves de 200 dígitos ( bits aproximadamente) en menos de 20 segundos.

* 1. APLICAR\_PADDING Y ELIMINAR\_PADDING

Estas funciones permiten añadir dígitos padding para cifrar el mensaje y eliminar dichos dígitos a la hora de descifrarlo. Esto se hace para evitar que ataques como el ataque a texto plano descifren fácilmente el mensaje.

Para aplicar el padding, la función recibe un número entero que determina la cantidad de cifras a añadir detrás de cada mensaje. De esta forma, ejecutamos un bucle for de duración , en el que añadimos al final de cada mensaje un número aleatorio entre 0 y 9.

A la hora de eliminar el padding, una vez aplicada la clave privada, eliminamos el número de dígitos establecido y obtenemos el mensaje original.

* 1. CIFRAR\_RSA Y DESCIFRAR\_RSA

Estas funciones se encargan de recibir un mensaje y tranformarlo realizando operaciones matemáticas. Para dichas operaciones necesitamos las claves públicas y privadas correspondientes.

A la hora de cifrar un mensaje requerimos de la clave pública (, ). Tomamos el mensaje y habiendo aplicado el padding correspondiente, realizamos la siguiente operación matemática para cifrar:

Donde es el mensaje cifrado.

Para descifrar, la operación es parecida. Requerimos de la clave pública (, ) y la clave privada . Esta última es la inversa de módulo . De esta forma, la operación que realizamos para descifrar es la opuesta a la que realizamos para cifrar:

Después de ejecutar esto, eliminamos el padding correspondiente y obtenemos el mensaje original.

* 1. ROMPER\_CLAVE

La función romper clave trata de encontrar dicha dada la clave pública (, ). De esta forma, si conseguimos encontrarla, podríamos descifrar cualquier mensaje enviado con dicha clave.

Para ello, debemos factorizar para poder calcular fácilmente . Una vez encontrado esto, simplemente debemos aplicar la propiedad siguiente:

Por lo tanto, debemos calcular la inversa de módulo . La única limitación que tiene este método para romper mensajes es que la función factorizar es muy costosa para números muy grandes, por lo que no es eficiente.

## ATAQUE\_TEXTO\_PLANO

Esta función se encarga de descifrar un texto cifrado sin conocer la clave privada, es decir, es un método alternativo al anterior. Dicho texto cifrado debe no tener padding. El procedimiento es el siguiente:

* Almacenamos en un diccionario todos los caracteres de Unicode junto con su respectivo cifrado.
* Para conseguir dicho cifrado, se pasa cada caracter a su código asociado y se cifra con la clave pública con la que se ha cifrado previamente el mensaje a descifrar.
* Una vez hecho esto, se recorre el mensaje cifrado buscando las coincidencias con los caracteres cifrados del diccionario y se sustituyen.

De esta manera, evitamos tener que factorizar para calcular , y por lo tanto, de no haber padding es una función bastante eficiente.

La única limitación como ya hemos mencionado anteriormente es que el mensaje no debe tener padding, ya que de ser así no habría coincidencias con los cifrados calculados.

# **TEXTO PLANO RECUPERADO**

El texto plano descifrado de la práctica presencial es:

“*En un lugar de la Mancha, de cuyo nombre no quiero acordarme, no ha mucho tiempo que vivía un hidalgo de los de lanza en astillero, adarga antigua, rocín flaco y galgo corredor. Una olla de algo más vaca que carnero, salpicón las más noches, duelos y quebrantos los sábados, lantejas los viernes, algún palomino de añadidura los domingos, consumían las tres partes de su hacienda.*”

Este descifrado tarda aproximadamente segundos.

# **BIBLIOGRAFÍA**

“Prueba De Primalidad De Miller-Rabin.” Frwiki.wiki, <https://es.frwiki.wiki/wiki/Test_de_primalit%C3%A9_de_Miller-Rabin>.

“Ataque De Texto Plano Elegido Introducción y Diferentes Formas.” Hmong, <https://hmong.es/wiki/Chosen_plaintext_attack>.