P2GP03

Natalia Leyenda Lodares
Daniela Blanco del Real

Cambios o mejoras implementadas en “modular.py”:

1. ***es_primo(n) y lista_primos(a,b)***
   El único error en estas dos funciones que podían afectar a esta práctica es que no reconocía a 2 como primo, hemos implementado en es_primo(n) una condición que dice que, si n es 2, que devuelva True. Respecto al segundo error en la práctica anterior, que devolvía True/False en vez de Sí/No, no afecta a esta práctica.
2. ***factorizar(n)***
   En la práctica anterior la función fallaba si el primo más grande que divide a n estaba al cuadrado (por ejemplo, al factorizar 4, en el que no se reconocía que 4=2^2 y se tomaba como primo). Para ello, el único cambio realizado en esta función es en la condición del while poner un menor o igual en vez de un menor estricto. Y en cuanto a que se imprimía el diccionario entre {}, no concierne a esta práctica.
   También hemos añadido un absoluto de la n al principio de la función para que pueda factorizar número negativos.
3. ***mcd(a,b) y bezout(a,b)***
   En la primera función, no funcionaba si uno de los dos números era 0, para solucionarlo hemos añadido una condición que indica que, si uno de los dos es 0, que devuelva el máximo de entre los dos (viniendo a ser, el número que no es 0).

   Por otro lado, Bezout no funcionaba con número negativos y se habían introducido parámetros de más en la función. Para el segundo error, hemos creado una función extra llamada

   bezout_recursivo(a:int,b:int,l1:List[int],2:List[int],mcd:int,a0:int,b0:int) -> Tuple[int,int,int]

   en la cual se realizan todos los cálculos necesarios para realizar Bezout. Esta función es llamada por la función *bezout(a,b)*. Para el primer error hemos creado 4 condicionales:

   1. a > b and b < 0 and a > 0
   2. b > a and a < 0 and b > 0
   3. a > b
   4. b > a

   Para la explicación tengamos en cuenta que la función busca una x e y tal que:

   $$ax + by = 1$$

   Las condiciones 1 y 2 cuando uno sea negativo y otro positivo, 3 y 4 cuando sean los dos positivos o los dos negativos. En los casos 1 y 4, se actualizan las coordenadas asociadas a la y y en los casos 2 y 3, actualiza las asociadas a la x.

4. ***potencia_mod_p(base,exp,p)***
   El único error en esta función es que no se trataban bien los exponentes negativos, por lo que en vez de hacer exp%p (lo que hacía antes), hemos aplicado el Teorema de Euler.

En primer lugar, calculamos la $\phi(p)$ con la función *euler(n)* y después hemos hecho el exponente módulo $\phi(p)$.

5. ***inversa_mod_p(n,p)***
   Nuestra anterior versión de la función no calculaba correctamente inversas de números negativos. Pero ese error era referente a la función de *bezout(a,b)*, la cual ya está arreglada.
6. ***euler(n)***
   Los errores referentes a esta función eran causados por los errores de factorizar, con lo cual, no hemos cambiado nada en esta función.

Descripción del funcionamiento de "rsa.py":

1. ***generar_claves(min_primo, max_primo)***
   El usuario introduce como input dos números. Generamos una lista con los números primos entre estos dos números y elegimos dos primos al azar: *p* y *q.* Multiplicando p_1 y p_2 obtenemos *n.* Para calcular la $\phi(n)$, sabemos que:

$$\phi(n) = \phi(pq) = \phi(p)\phi(q).$$

   A continuación, buscamos un número, *d*, tal que:

$$1 < d < \phi(n)$$

   y,

$$mcd\big(d, \phi(n)\big) = 1$$

   Es decir, que d y $\phi(n)$ sean coprimos, lo que comprobamos con la función de "modular.py", *coprimos*.
   Calculamos una *e* tal que:

$$e \cdot d = 1 \ (mod\ \phi(n))$$

   Es decir, calculamos la inversa de *d* módulo $\phi(n)$ llamando a la función *inversa_mod_p* de *"modular.py".* De esta forma se generan la clave pública *(n, e)* y la clave privada *d.*

2. ***aplicar_padding(m, digitos_padding)***
   Aplicamos padding al mensaje antes de ser encriptado para generar *ruido* y dificultar los ataques de texto plano. Para aplicar el padding pasamos el mensaje a string, al que añadimos cifras aleatorias al final. Por se convierte a entero para poder realizar el cifrado.

3. ***eliminar_padding(m, dígitos_padding)***
   Esta función es llamada por la función *descifrar_rsa(),* para eliminar el padding de la cadena con el mensaje descifrado y su funcionamiento es análogo a *aplicar_padding().*

4. ***cifrar_rsa(m, n, e, dígitos_padding)***
   Si hay padding a añadir *cifrar_rsa()* llama a la función *aplicar_padding().* Después se realiza el cifrado:

$$c \equiv m^e \ (mod\ n)$$

El nuevo mensaje cifrado, c, se calcula como la potencia en módulo *n* del mensaje original, *m*, elevado a *e,* parte de la clave pública del destinatario. Esta potencia la calculamos con la función *potencia_mod_p()* de *modular.py,* en la que aplicamos el método de la exponenciación binaria y el Pequeño Teorema de Fermat para reducir los tiempos de ejecución y mejorar el rendimiento.

5. ***descifrar_rsa(c ,n, d, dígitos_padding)***
   Esta función *obtiene* el mensaje descifrado, *m,* usamos también la función *potencia_mod_p()* de *modular.py*. Calculamos:

$$m \equiv c^d \ (mod\ n)$$

   Haciendo uso de la clave privada, *d*, ya que:

$$e \cdot d \equiv 1 \ (mod\ n)$$

$$m \equiv m^{e \cdot d} (mod\ n) \equiv c^d \ (mod\ n)$$

   Tras el descifrado se elimina el padding, si es que lo hay.

6. ***cifrar_cadena_rsa(s, n, e, dígitos_padding)***
   Esta función cifra cadenas llamando con un bucle a la función *cifrar_rsa()* y guardando los valores de cada carácter cifrado en una lista. Antes del cifrado se obtiene el *Unicode* de cada carácter.

7. ***descifrar_cadena_rsa(cList, n, d, dígitos_padding)***
   La función recorre la lista de caracteres cifrados y los descifra uno a uno usando la función *descifrar_rsa().* A partir de los códigos Unicode obtenidos se convierte a los caracteres correspondientes que se añaden a la cadena del mensaje.

8. ***romper_clave(n, e)***
   Esta función pretende obtener la clave privada, *d,* a partir de las claves públicas, *n* y *e.* En primer lugar, se calcula la $\phi(n)$ usando la función de *euler()* de "*modular.py"*. Una vez obtenida se calcula la clave privada, *d*, como la inversa módulo $\phi(n)$ de *e.*

9. ***ataque_texto_plano(cList, n, e)***
   Una vez obtenida la clave privada, *d*, con la función *romper_clave(),* se descifra el mensaje con la función *descifrar_cadena_rsa(),* igual que antes.

¿Es capaz el código de generar claves grandes (más de 20 dígitos) en un tiempo razonable?

El código permite generar claves de más de 20 dígitos, pero el tiempo que tarda en hacer no depende tanto del número de dígitos de las claves, sino de la diferencia entre los dos números introducidos, ya que cuanto más grande es la diferencia de estos dos números, más grande será la lista de primos a encontrar. A continuación, se muestran un ejemplo de generación de claves de más de 20 dígitos.

```
Introduzca un número: 8364829202
Introduzca otro número: 8369865400
n:  70035043926230440283
e:  63314237925569560507
Ha tardado 553.652526140213s
```

Si en el ejemplo con una diferencia de 5036198 ha tardado entre 9 y 10 minutos en generar una clave de 21 dígitos, cuanto mayor diferencia más tardará y cuando menos diferencia, menos tardará

Texto plano recuperado a partir del texto cifrado recibido en la práctica presencial.

Hemos implementado un archivo extra llamado ataque_texto_plano.py que realiza esta funcionalidad. En este creamos tres variables con las claves públicas dada en el archivo "Criptograma X" de Moodle y un string con todos los caracteres del mensaje encriptados. A este string le aplicamos un string y con un bucle pasamos posición a posición para convertir cada número a entero. Por último, llamamos a la función de ataque de texto plano en *rsa* e imprimimos la solución. A continuación, se muestra la salida por pantalla obtenida.

```
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, salpi
cón las más noches, duelos y quebrantos los sábados, lantejas los viernes,
 algún palomino de añadidura los domingos, consumían las tres partes de su
 hacienda.
Ha tardado 891.6892642974854s
```

Ataques a RSA

Mediante el Algoritmo de Fermat hemos aplicado una nueva estrategia de ataque rsa. Según Fermat, si $n = pq$

$$(p+q)^2 - 4n = (p+q)^2$$

El método, se trata de un método de factorización, y consiste en encontrar una x e y tal que cumplan $4n = x^2 - y^2$. Por lo tanto, implementamos un bucle que va probando $x = \lceil 2\sqrt{n} \rceil$, $\lceil 2\sqrt{n} \rceil + 1$, $\lceil 2\sqrt{n} \rceil + 2$, … así hasta que $x^2 - 4n$ sea un cuadrado perfecto. Cuando ya se hallan la x e y:

$$p = \frac{1}{2}(x-y),\ q = \frac{1}{2}(x+y)$$

Hemos añadido un archivo más a la entrega llamado ataque_rsa.py en el que se muestra el código.

¿Pueden estas funciones romper cualquier tipo de cifrado con el esquema establecido en la práctica o hay alguna limitación?

En el ataque texto plano, no hemos encontrado ninguna limitación, más que el hecho de que puede tardar mucho tiempo, dependiendo del número de cifras de la n. Si con 20 cifras tarda unos 15 minutos, se pone una clave con más cifras, tardará aún más.

En el ataque mediante el Algoritmo de Fermat el tiempo que tarda depende la diferencia que haya entre los dos primos, cuanto más cerca estén, menos tardará, pero cuanto más alejados estén, más tardará. Por otro lado, con n pequeñas sí que encuentra los primos correctamente, pero con n grandes, comete fallos. Por ejemplo, si introducimos la n del "Criptograma X", 28282590191348679547, obtenemos lo siguiente por pantalla:

```
p:  5110120897
q:  5534622519
p*q:  28282590191348679543
Ha tardado 5.164473295211792s
```

Como se puede observar, solo falla en la última cifra, pero no sabemos explicar porque, ya que con otros números funciona:

n = 52841

```
p:  53
q:  997
p*q:  52841
Ha tardado 0.0010001659393310547s
```

n = 988027

```
p:  991
q:  997
p*q:  988027
Ha tardado 0.001016378402709961s
```

n = 5959

```
p:  59
q:  101
p*q:  5959
Ha tardado 0.0s
```

Descripción del funcionamiento de la interfaz de "criptochat.py"

En primer lugar, hemos creado un menú que permite al usuario elegir cualquiera de las funcionalidades del programa. El menú se encuentra dentro de un bucle while para que continue ejecutándose hasta que el usuario decida salir del programa, mediante la opción 7.

Para la gestión de errores hemos creado un try que envuelve la selección de opciones y evita que se pueda elegir una opción introduciendo un string o un número < 1 o > 8.

De forma particular, cada opción tiene una gestión de errores que comprueba que haya unas claves público-privadas ("FileNotFoundError") y que los valores introducidos sean válidos ("ValueError"), excepto en la última opción.

Si se elige la **primera opción** (generar un par de claves), pedimos un input de los dos números a partir de los que generar las claves. Se crea un fichero donde se guardan las claves público-privadas para poder usarlas más veces.

La **segunda opción** permite al usuario registrar sus propias claves público-privadas. Comprobamos que los números introducidos sean positivos y, por lo demás, asumimos que el usuario introduce claves público-privadas válidas. Si ya había claves guardadas estas serán

sustituidas. La **tercera opción** tiene un funcionamiento muy similar, pero registrando las claves del destinatario.

La **cuarta opción** permite cifrar un mensaje usando las claves. Las claves se obtienen del fichero y el mensaje a cifrar se introduce con un input. Si no se ha seleccionado un número de dígitos de padding con la sexta opción, no hay padding por defecto.

Si se selecciona la **quinta opción** se puede descifrar un mensaje cifrado, al igual que la opción anterior el mensaje se introduce con un input y las claves se encuentran en el fichero.

En la **sexta opción** se puede seleccionar el número de cifras de padding, comprobando que sean menos cifras que las de las claves y que sea positivo. Antes de aplicarlo se comprueba que haya claves generadas.

Por último, si la **opción seleccionada es la 7**, la ejecución del programa finaliza. Se borran los archivos con las claves generadas durante la ejecución.

Bibliografía

Ataques RSA

Balbás Gutiérrez, David, Julio 2019. "*Ataque al criptosistema RSA*". Facultad de Ciencias de la Universidad de Cantabria

Código en general

Alfaya Sánchez, David, 2022. "*Teoría de Números*". Matemática Discreta. ICAI, Universidad Pontificia de Comillas.

Hernández Encinas, Luis. 2022. "*Matemáticas y Criptología*". ICAI, Universidad Pontificia de Comillas.