Criptografía

Grupo 10

Obligatorio 2

Integrantes

Nombre	CI	Correo
Lucía Santos	4.720.691-6	lulisantosm1009@gmail.com
Germán Ouviña	4.823.566-1	german.ouvina@fing.edu.uy

Índice

1. Introducción

El siguiente documento es un informe de las actividades realizadas en el contexto del **Obligatorio 2** del curso de Criptografía de Fing, UdelaR, edición 2019. Dicho trabajo tiene como objetivo el analizar los fundamentos del criptosistema **RSA**, centrandose en propiedades de los **números primos**.

Con el fin de facilitar la lectura de este informe y dada la naturaleza de investigación del trabajo, cada ejercicio contará con una sección **preliminares**, donde se trataran los objetivos del ejercicio y los pasos seguidos al momento de encararlo, una sección **problema**, donde se adjuntan los datos generados durante el ejercicio y los conceptos teóricos trabajados durante el mismo y por último una sección **conclusiones**, agregando las conclusiones generadas.

A su vez, se adjutará como pie de página cuando sea pertinente, comentarios u observaciones en carácter informal, con el objetivo de transmitir las nociones percibidas (correctas o no) al momento de investigar, sin intervenir con el marco teórico.

Como detalle final, se agrega una sección de **conclusiones generales** sobre el trabajo realizado. Por otra parte, el código de los scripts es adjuntado junto al informe en la entrega.

2. Desarrollo

2.1. Ejercicio 1 - Teorema de Números Primos

2.1.1. Preliminares

El objetivo del siguiente ejercicio es estudiar el comportamiento de la función $\pi(x)$ también denominada como **función de conteo de primos**. Para ello, siguiendo los puntos de la consigna, se encaró el problema con los siguientes pasos:

1. Estudiar $\pi(x)$, su aproximación y sus cotas para valores representativos de x.
2. Verificar que el *Teorema de Números Primos 3.21* se cumple mediante métodos gráficos y algorítmicos.
3. Graficar $\pi(x)$ y sus cotas, todo dividido por su aproximación. Observar su comportamiento y extraer conclusiones.

2.1.2. Problema

2.1.2.1 Verificación Empírica de Teorema de Números Primos

El *Teorema de Números Primos* establecido en la sección 3.21 del libro *CryptoSchool* determina que la **función de conteo de primos** $\pi(x)$ cumple las siguientes propiedades:

$$\begin{cases} \pi(x) \approx \frac{x}{ln(x)} \\ \pi(x) > \frac{x}{ln(x)}(1 + \frac{1}{2ln(x)}) \\ \pi(x) < \frac{x}{ln(x)}(1 + \frac{3}{2ln(x)}) \end{cases} : x \geq 59$$

En otras palabras, $\pi(x)$ se aproxima a la función $\frac{x}{ln(x)}$ y está acotada superior e inferiormente. Antes de poder asegurar que el teorema efectivamente se cumple, se compararon los valores representativos pedidos en la consigna. A continuación se adjunta dicha tabla comparativa [1]:

x	$\pi(x)$	$\frac{x}{ln(x)}(1 + \frac{1}{2ln(x)})$	$\frac{x}{ln(x)}(1 + \frac{3}{2ln(x)})$	$\frac{x}{ln(x)}$	$\frac{\pi(x)}{x/ln(x)}$
1000	168	176	155	144	1.17
2000	303	315	280	263	1.15
3000	430	444	398	375	1.15
4000	550	569	511	482	1.14
5000	669	690	621	587	1.14
6000	783	808	729	690	1.13
7000	900	924	835	791	1.14
8000	1007	1038	939	890	1.13
9000	1117	1151	1042	988	1.13
10000	1229	1262	1144	1086	1.13

[1] Todos los valores se encuentran redondeados hacia abajo

Se pueden realizar múltiples observaciones rápidas, como que $\pi(x)$ se encuentra efectivamente acotado para esos valores y que $\pi(x)$ y $\frac{x}{ln(x)}$ tienen un comportamiento aproximado, ya que su división se mantiene alrededor de **1.15**. En la sección de **conclusiones** se expanden dichas observaciones.

2.1.2.2 Verificación Exacta de Teorema de Números Primos

Habiendo comprobado empíricamente el comportamiento de las funciones a estudiar, se plantearon dos métodos distintos para verificar que el teorema se cumple para $x \leq 10000$:

1. **Verificación algorítmica:** Se desarrolló un script denominado *pi_verification.py* que comprueba para todo $x \leq 10000$ por fuerza bruta (es decir, prueba iterativamente) que el valor de $\pi(x)$ se encuentre entre el valor de las dos cotas para x. Dicho script utiliza la implementación de $\pi(x)$ denominada *primepi*, proveniente de la biblioteca *sympy*. Dicha implementación se asume correcta.
2. **Verificación gráfica:** Se desarrolló un script denominado *pi_plots.py* que genera gráficas comparativas entre $\pi(x)$, $\frac{x}{ln(x)}$ y las cotas mencionadas en el teorema, para valores de $x \leq 10000$. Al igual que en el script anterior, se utiliza la implementación de $\pi(x)$ denominada *primepi*, proveniente de la biblioteca *sympy*.

Una vez desarrollados ambos scripts, se comprobó que efectivamente $\pi(x)$ se encuentra acotada, cumpliendo el teorema. En la siguiente sección se adjuntan las gráficas generadas, así como las observaciones resultantes de las mismas.

2.1.2.3 Gráficas y comparaciones

Se generaron las siguientes gráficas:

1. Comparación entre $\pi(x)$ y $\frac{x}{ln(x)}$:
   Efectivamente, ambas funciones se aproximan, sobretodo al principio. Cabe destacar que $\pi(x)$ ¿$\frac{x}{ln(x)}$.
2. Comparación entre $\pi(x)$ y sus cotas
   Efectivamente, $\pi(x)$ se encuentra acotada superior e inferiormente en todo el intervalo.
3. Comparación entre $\frac{\pi(x)}{x/ln(x)}$ y sus cotas
   Es interesante ver que si bien la aproximación entre $\pi(x)$ y $\frac{x}{ln(x)}$ oscila entre valores cercanos a 1.15 y 1.25, se encuentra perfectamente acotada por las cotas anteriormente definidas.

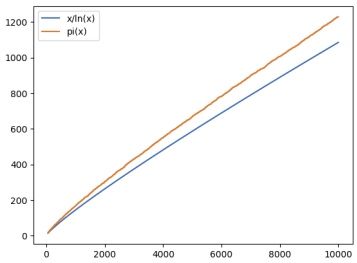

Figura 2.1: Comparación entre $\pi(x)$ y $\frac{x}{ln(x)}$

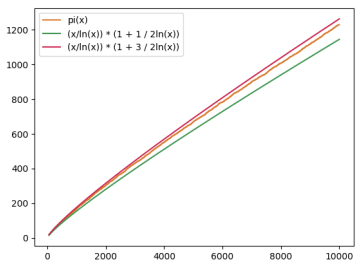

Figura 2.2: Comparación entre $\pi(x)$ y sus cotas

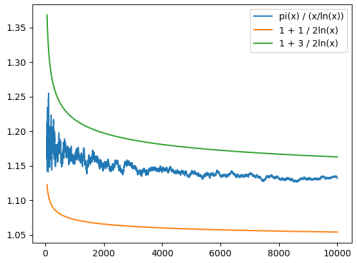

Figura 2.3: Comparación entre $\frac{\pi(x)}{x/ln(x)}$ y sus cotas

2.1.3. Conclusiones

Analizando los datos obtenidos a lo largo del ejercicio, se puede determinar que el comportamiento de $\pi(x)$ no es errático, pudiendo aproximarse a $\frac{x}{ln(x)}$. De todas formas, dicha aproximación dista de ser perfecta, como se comprueba en la última gráfica. Si se necesita una condición más fuerte, se cuenta con una cota superior y otra inferior.

Teniendo en cuenta lo anterior, dada la cantidad de primos desconocidos que todavía existen entre los enteros de mayor orden, el conocimiento de la aproximación y de las cotas puede ser de gran utilidad al intentar estudiar la función $\pi(x)$ en intervalos donde la cantidad de primos existente es desconocida.

2.2. Ejercicio 2 - Buscar un Número Primo

2.2.1. Preliminares

El objetivo del siguiente ejercicio es implementar el *Test de Fermat* y correrlo para un número grande establecido (más concretamente, $B = 19960910000000$) y sus siguientes números. Para ello, siguiendo los puntos de la consigna, se encaró el problema con los siguientes pasos:

1. Utilizar distintos métodos para encontrar $k : \exists p \in (B, B + k)$ siendo p primo.
2. Optimizar el *Test de Fermat* para reducir la cantidad de corridas necesarias en un factor de $\frac{1}{2}$ o más.
3. Implementar el *Test de Fermat* optimizado.
4. Buscar **testigos de Fermat** para ciertos valores y encontrar un número *posiblemente primo* mayor a B.

2.2.2. Problema

2.2.2.1 Estimación de números a examinar

Como se vio en la sección anterior, la función $\pi(x)$ podría resultar particularmente útil. De hecho, bastaría con encontrar $k : \pi(B + k) - \pi(B) >= 1$.

En la práctica y dado el tamaño de B, es difícil, por lo que se debe recurrir a otros métodos. Para tener una idea de cual es el k, se puede utilizar la aproximación $\pi(x) \approx \frac{x}{ln(x)}$, buscando $k : \frac{(B+k)}{ln(B+k)} - \frac{(B)}{ln(B)} >= 1$. El problema con esto es que, como ya se determinó, $\pi(x) > \frac{x}{ln(x)}$ y la aproximación se separa al crecer x.

Para asegurarlo con exactitud, se pueden utilizar las cotas de $\pi(x)$. Si el valor de la cota inferior para $B + k$ es superior al valor de la cota superior para B, entonces se puede asegurar que existe al menos un primo entre B y $B + k$.

Para $B = 19960910000000$, se encontró que $k = 663675692565$ cumple con el postulado anterior. Se utilizó un script denominado *check_k.py* que prueba por fuerza bruta a partir de un k pasado por parámetro. Dada la diferencia entre las cotas superior e inferior para B, se determinó que el valor k iba a rondar por 10^11, por lo que se utilizó un k inicial cercano.

Cabe destacar que utilizando un algoritmo de búsqueda con bipartición, se podría encontrar mucho más rápido el valor exacto de k.

2.2.2.2 Optimización de Test de Fermat

Dado que el objetivo es correr el *Test de Fermat* k veces para el intervalo $(B, B + k)$, se evaluarán muchos números compuestos para los cuales no es necesario correr el test: los números pares.

Tomando esto en cuenta, si se agrega una comprobación sencilla de que el parámetro N a probar no sea par, la cantidad de corridas se reduciría a la mitad sin problema.

Si quisiese reducirse más aún, podrían realizarse otras comprobaciones como por ejemplo, que la suma de dígitos del parámetro N no sea múltiplo de 3 (en caso de serlo, N sería múltiplo de 3 y por lo tanto debería ser descartado), o que el parámetro $N \not\equiv 5 \pmod{19}$ (en caso de serlo, N sería múltiplo de 5). De esta forma, se reducen notablemente las corridas necesarias, optimizando el tiempo de ejecución.

2.2.2.3 Testigos de Fermat

Una vez implementado el *Test de Fermat* en el script *fermat_test.py*, se determinó que el primer número compuesto siguiente a B es $B+1 = 19960910000001$. El testigo encontrado fue 7361782047475. No obstante, siempre que existe un testigo de Fermat existen muchos, por lo que al correr el script varias veces, se obtienen distintos testigos.

Además de esto, se corrió una variante del script que permite encontrar el primer número **posiblemente primo** siguiente a B. Corriendo para distintos parámetros $k = 10, 100, 1000, 10000, 100000, 1000000$ del *Test de Fermat* se determinó que el siguiente **posible primo** a B es $B_p = 19960910000027$. Los primeros 10 números a encontrados que cumplen $a^{B_p} \equiv 1 (\text{mod } B_p)$ fueron: 13818850694700 - 15520516008267 - 18226268629593 - 19505519012377 - 19621430597373 - 10722973921764 - 5307054094290 - 2936681375254 - 8009142805623 - 6767440914124.

2.2.3. Conclusiones

Es importante destacar que tanto las aproximaciones hechas al buscar k en la primera parte, como las corridas utilizando el *Test de Fermat*, no son de las técnicas óptimas a la hora de tratar estos problemas.

Primero, si bien la técnica de las cotas **asegura** encontrar un primo para el intervalo $(B, B + k)$, no garantiza que ese sea el menor k posible. De hecho, es seguro que en la mayoría de casos, el menor k posible es mucho menor al k determinado con ese método.

Por otra parte, al utilizar el *Test de Fermat* siempre existe la posibilidad de no realizar las pruebas suficientes. El parámetro k de cuantas corridas del test se hacen sin duda aumenta la seguridad, pero no puede permitir asegurar al 100 % que el número sea primo (si es un compuesto de Carmichael, incluso probando con todo el espacio devolvería posiblemente primo).

Dicho esto, no debe negarse la utilidad de estos métodos, ya que otorgan información valiosa.

2.3. Ejercicio 3 - Mala Elección en RSA

2.3.1. Preliminares

El objetivo del siguiente ejercicio es determinar **por qué** es una mala elección en **RSA** elegir dos números primos cercanos para determinar el factor $N = pq$. Para ello, utilizando la sugerencia de la consigna de basarse en la *Factorización de Fermat*, se encaró el problema con los siguientes pasos:

1. Estudiar la *Factorización de Fermat* y su relación a la factorización del parámetro N en **RSA**.
2. Determinar un método factible computacionalmente para encontrar la descomposición $N = pq$ utilizando la anteriormente mencionada *Factorización de Fermat*.
3. Generar un programa que permita automatizar el algoritmo para cualquier número.
4. Encontrar la factorización $N = pq$ para $N = 23360947609$ utilizando el programa generado y comprobar que sea correcta.

2.3.2. Problema

2.3.2.1 Factorización de Fermat

El *método de factorización de Fermat* se basa en que cualquier **número entero impar** puede representarse como la diferencia entre dos cuadrados, utilizando la siguiente expresión:

$$N = a^2 - b^2 = (a + b)(a - b) : a, b \in \mathbb{Z}$$

La segunda igualdad (la factorización en base a a y b) se desprende de la primera, representando una posible factorización de N (cuando $a + b \neq 1$ o $a - b \neq 1$).

Tomando el hecho de que en **RSA** los factores N se construyen como el producto de dos **números primos** $N = pq$ y utilizando la anterior expresión, se puede establecer la siguiente igualdad:

$$\begin{cases} p = a + b \\ q = a - b \end{cases} : a, b \in \mathbb{Z}$$

Si se despeja a y b en función de p y q, se obtiene la expresión propuesta en la consigna:

$$\begin{cases} a = \frac{p+q}{2} \\ b = \frac{p-q}{2} \end{cases} : p, q \in \mathbb{Z}$$

Toda esta información resultará de gran utilidad en la siguiente sección

2.3.2.2 Algoritmo de Factorización de N

Tomando las anteriores propiedades, se busca construir un algoritmo que para un factor N de **RSA** que cumpla $N = pq : p, q \in \mathbb{Z}$ y p y q primos, encuentre sus valores de forma viable computacionalmente (es decir, corriendo en un tiempo relativamente corto).

En una primera instancia, se puede utilizar el anteriormente mencionado *método de factorización de Fermat* que partiendo de un N impar, devuelve dos números enteros $a, b : N = a^2 - b^2$. El método funciona como sigue:

1. Se toma $a = \lceil \sqrt{N} \rceil$
2. Se despeja $B = N - a^2$
3. Si $B = b^2$, entonces B es un cuadrado perfecto y se encontró a y b
4. Si no, se aumenta $a = a + 1$ y se prueba el paso anterior hasta encontrar a y b

Existen dos consideraciones importantes a tomar:

1. **Correctitud:** El interés de este punto radica en encontrar p y q, no a y b. Tomando eso en cuenta, el método de Fermat por sí sólo no funcionaría. No obstante, al saber que $N = pq$ (al ser parte de una clave pública en **RSA**, esto es seguro), se puede despejar $p = a + b$ y $q = a - b$ como se determinó en la anterior sección, pudiendo efectivamente obtener p y q del método original.
2. **Eficiencia:** Para un N muy grande, se puede iterar hasta $\sqrt{N}$ veces y operando con números de gran tamaño. En el contexto de **RSA**, se utilizan claves de 1024 bits como mínimo, por lo que efectivamente se tratan números de un tamaño importante. Aquí es donde entra en juego la relación entre p y q. **Si son cercanos** como se asume en la consigna, entonces ambos van a encontrarse cerca de $\sqrt{N}$, por lo que el algoritmo necesitará pocas iteraciones para devolver un resultado, siendo computacionalmente factible.

De esta forma, se implementó un script denominado *fermat_factorization.py* que corre el **método de factorización de Fermat** para un número dado y retorna su factorización $N = pq$ en caso de encontrarla.

2.3.2.3 Factorización de $N = 23360947609$

Utilizando el script anteriormente mencionado, se obtuvo los siguientes valores:

$$\begin{cases} p = 153649 \\ q = 152041 \end{cases}$$

Se puede comprobar que, efectivamente $N = 23360947609 = 153649 * 152041$, por lo que el algoritmo funciona correctamente. A su vez, si se utiliza una herramienta de

Es importante destacar que sólo se necesitaron **3 iteraciones** para encontrar los valores: la primera, donde se probó con $a = \lceil \sqrt{N} \rceil = 152843$ y las dos siguientes.

2.3.3. Conclusiones

Además de las tareas de la consigna, se corrió el script para ciertos $N = pq$ tal que p y q no fueran "cercanos". Se utilizaron los siguientes valores. En cada caso fue necesario correr muchas iteraciones (191, 8965, 77975 respectivamente):

- $\begin{cases} N = 5731 \\ p = 521 \\ q = 11 \end{cases}$
- $\begin{cases} N = 323867 \\ p = 19051 \\ q = 17 \end{cases}$

- $\begin{cases} N = 6602927 \\ p = 161047 \\ q = 41 \end{cases}$

Una conclusión importante que se extrae de esto es que, por muy lejanos que sean los números primos que se utilizan, la cantidad de iteraciones es de menor orden que el más grande de los dos, por lo que es crucial que para hacer computacionalmente inviable el encontrar p y q, se utilice un espacio de claves de gran tamaño.

De todas formas, sigue comprobandose que la separación entre p y q también juega un rol importante, ya que para valores de N relativamente pequeños, se necesitaron muchas más iteraciones que para valores de N mayores pero con p y q cercanos.

3. Conclusiones generales

El criptosistema **RSA** se fundamenta en la utilización de **numeros primos**, los cuales son números enteros muy particulares. Existen muchas investigaciones en relación a los números primos a lo largo de la historia, pero siguen sin encontrarse métodos computacionalmente viables para determinar todos los primos existentes a partir de ciertos valores muy grandes.

Teniendo esto en cuenta, la criptografía basada en números primos probablemente no se vea amenazada hasta que las capacidades de computo aumenten considerablemente.

4. Referencias

Debido al continuo uso de conceptos teóricos extraídos de la bibliografía, no se citó cada referencia al momento de utilizarla. No obstante, a continuación se adjunta la lista de fuentes utilizada en el proceso de investigación y construcción del informe.

[1] **CryptoSchool** - Joachim von zur Gathen, 2015
La fuente principal de información, libro del curso.

[2] **Fermat's factoring trick and cryptography** - https://www.johndcook.com/blog/2018/10/28/fermat-factoring
Artículo sobre el método de factorización de Fermat y su utilidad con respecto a una elección de primos cercanos en **RSA**.

[3] **List of primes up to 1000 billion** - http://compoasso.free.fr/primelistweb/page/prime/liste_online_en.php
Herramienta web que permite buscar números primos en una lista de números primos menores a 1.000.000.000.000.