UCLV
Universidad Central
"Marta Abreu" de Las Villas

MFC
Facultad de Matemática
Física y Computación

Departamento de Matemática

TRABAJO DE DIPLOMA

Título: Generación de Parámetros de Seguridad para la Criptografía Asimétrica.

Autor: Yamil Ernesto Morfa Avalos

Tutores: Dr. C. Lucia Argüelles Cortés

MSc. Gonzalo Palencia Fernández

Santa Clara

UCLV
Universidad Central
"Marta Abreu" de Las Villas

MFC
Facultad de Matemática
Física y Computación

Math department

DIPLOMA THESIS

Title: Generation of Security Parameters for Cryptography Asymmetric.

Author: Yamil Ernesto Morfa Avalos

Thesis Director: Dr. C. Lucía Argüelles Cortés

MSc. Gonzalo Palencia Fernández

Santa Clara, July 2019

Para cualquier información contacte con:
Dirección de Información Científico Técnica. Universidad Central “Marta Abreu” de Las Villas. Carretera a Camajuaní. Km 5½. Santa Clara. Villa Clara. Cuba.
CP. 54 830
Teléfonos.: +53 01 42281503-1419

… A mi familia que siempre me ha apoyado, a mis amigos que tantos buenos momentos me han dado, a mis profesores por una magnifica formación….

Resumen:

En la Criptografía Asimétrica o Criptografía de Clave Pública son usadas diferentes técnicas basadas en un mismo principio, es necesario que el proceso de cifrado sea “fácil” de realizar o computacionalmente tratable y además que el proceso de descifrado sea “difícil” de realizar o computacionalmente intratable. Tras haber realizado un estudio y análisis de las técnicas criptográficas más representativas de la criptografía simétrica, ha resaltado la relevancia que juegan los números primos y todas las propiedades que estos ofrecen. Para el proceso de cifrado de todos los algoritmos criptográficos tratados es necesaria la selección de números primos que cumplen ciertas propiedades, a los cuales se les denomina números primos fuertes o parámetros de seguridad, es por esto que se ha hecho un estudio bibliográfico de este tema en específico, a través de dicho estudio se ha observado que los algoritmos actualmente usados para obtener estos parámetros presentan algunas “debilidades” ; esto viene dado por el hecho de que dichos algoritmos son probabilísticos y este tipo de algoritmos tiene ciertas características que pueden tanto perjudicar como beneficiar el proceso de cifrado. El propósito del autor es proponer un algoritmo original que resuelva este problema y además sea un algoritmo determinístico con una complejidad computacional similar a la de los algoritmos probabilísticos antes mencionados. Dado que la sucesión de todos los números primos mayores que 3 está contenida en la sucesión de todos los números impares mayores que 1 y no múltiplos de 3, entonces se ha hallado una interesante forma de suprimir los números no primos mediante el uso de ecuaciones diofánticas. Este resultado sirve como un nuevo test para determinar la primalidad de un número, test que a su vez es usado en la modificación del Algoritmo de Gordon para generar números primos fuertes. Al Algoritmo de Gordon Modificado, algoritmo determinista, se la ha realizado un análisis de su complejidad. Dicho análisis ha mostrado que el Algoritmo Modificado es fiable y que se comporta de manera eficiente para números de tamaño hasta $50\ bits$. Además se ha recomendado como trabajo futuro el mejoramiento del método de solución de las ecuaciones diofánticas usadas.

Summary:

In Asymmetric Cryptography or Public Key Cryptography are used different techniques based on the same principle, it is necessary that the encryption process is "easy" to perform or computationally treatable and also that the decryption process is "difficult" to perform or computationally intractable. After having carried out a study and analysis of the most representative cryptographic techniques of symmetric cryptography, it was highlighted the relevance of prime numbers and all the properties they offer. For the encryption process of all the treated cryptographic algorithms it is necessary to select prime numbers that fulfill certain properties which are called strong prime numbers or security parameters, that is why a bibliographic study of this specific topic has been done, through this study it has been observed that the algorithms currently used to obtain these parameters present some "weaknesses". This is due to the fact that these algorithms are probabilistic and this type of algorithms has certain characteristics that can both harm and benefit the encryption process. The purpose of the author is to propose an original algorithm that solves this problem and is also a deterministic algorithm with a computational complexity similar to the aforementioned probabilistic algorithms. Since the succession of all prime numbers greater than 3 is contained in the sequence of all odd numbers greater than 1 and not multiples of 3, then an interesting way of suppressing non-prime numbers has been found by using diophantine equations . This result serves as a new test to determine the prime condition of a number, a test that in turn is used in the modification of the Gordon Algorithm to generate strong prime numbers. To the Modified Gordon Algorithm, a deterministic algorithm, an analysis of its complexity has been carried out. This analysis has shown that the Modified algorithm is reliable and behaves efficiently for numbers up to 50 bits in size. In addition, the improvement of the solution method of the used diophantine equations has been recommended as future work.

Introducción:

Los números primos han sido motivación de innumerables investigaciones a lo largo de la historia, desde los pitagóricos y Euclides quienes develaron algunas de sus propiedades, sería muy difícil citar a todos los matemáticos que han hecho aportes significativos a la teoría de números primos. Sin embargo, su relevancia se ha visto incrementada notablemente, no tanto por la riqueza de sus propiedades, sino por sus aplicaciones en el contexto de la teoría de la información, (Criptología, Teoría de Códigos, Comprensión de la Información). Este trabajo ocupa solo el rol que tienen los números primos en la rama Criptología y más específicamente en la criptografía.

Criptografía: La criptografía se dedica al desarrollo de técnicas que transforman un texto inteligible en otro (llamado criptograma) cuyo contenido de información (semántica) es igual al del texto inteligible, pero que sólo pueden interpretar personas autorizadas (Pardo, 2016).

El contexto de la Criptografía moderna es también el contexto de la Teoría de Números, de los números primos y del diseño y análisis de algoritmos de factorización de números enteros. Desde el año 1978, a partir de la aparición del sistema criptográfico RSA (por sus autores [**RSA, 78**] su utilización en sistemas de telecomunicación y, especialmente, en Internet se ha extendido al punto de ser la base de las comunicaciones seguras por internet (Criptografía de mensajes y correos electrónicos, el sistema de acceso seguro SSH, etc.). A medida que la tecnología avanza y con ella las necesidades de una mejor protección de la información se hace inminente la necesidad de la constante investigación en la Criptografía Moderna y Teoría de Números Primos.

En la década de 1970, la aparición de un nuevo tipo de sistemas criptográficos con características radicalmente distintas a todos los conocidos hasta entonces supuso una auténtica revolución, marcando un antes y un después en la ciencia de la Criptografía. Así nació, casi de un día para otro, la Criptografía

moderna, sin cuya existencia el mundo de las comunicaciones, tal y como lo conocemos hoy día, no sería posible (o, al menos, tendría muy graves limitaciones). Por el contrario, el interés en lo que hasta entonces había sido la Criptografía (que en adelante pasaría a ser denominada Criptografía clásica) decreció considerablemente, aunque no hasta el punto de desaparecer, de hecho en la actualidad los sistemas comerciales de uso común son un híbrido de los sistemas modernos y clásicos. Whitfield Diffie y Martin Hellman, responsables del artículo “New Directions of Cryptography” , fueron los primeros en dar el primer paso significativo a lo que hoy llamamos Criptografía Asimétrica o Criptografía Moderna (sección 2.2), Anteriormente el proceso de cifrado y descifrado de cierto mensaje se realizaba usando una única clave de seguridad que rea intercambiada por medios seguros entre las partes que se comunicaban, la idea de Diffie-Hellman era realizar este intercambio de claves públicamente y aun así de manera segura. Esta idea revolucionó la criptografía clásica y sobre ésta se construyen los algoritmos criptográficos tan utilizados hoy en día tales como el RSA.

La criptografía moderna está respaldada por una extensa teoría matemática: Algebra Moderna, Teoría de Números, Teoría de la Complejidad, pero todos sus protocolos y métodos siguen un principio, y es que el proceso de cifrado de cierto mensaje, que se realiza utilizando una clave que es de conocimiento público a la cual se le denomina llave pública, sea fácil de realizar y que a un “intruso” que intercepte el mensaje cifrado le sea difícil descifrarlo a partir de la llave pública. Es en la generación de estas llaves públicas en donde intervienen los números primos y sus propiedades, para la generación de las mismas es una precondición el poder generar números primos con ciertas propiedades, llamados números primos fuertes o parámetros de seguridad. Los algoritmos actuales usados para esta cuestión son probabilísticos (sección 2.8) por lo que es posible que un número compuesto pase por falso primo comprometiendo así la seguridad del criptosistema que lo use para la generación de su llave pública, pero;

¿Será posible encontrar un algoritmo determinista, o sea que su solución sea fiable, que tenga una complejidad (sección 1.4) similar a los algoritmos deterministas?

La intención del autor con este trabajo es dar respuesta a esta interrogante, es por esto que el objetivo principal de este trabajo es:

- Proponer un algoritmo determinístico para la generación de números primos fuertes o parámetros de seguridad para la criptografía asimétrica.

Para esto se dará cumplimiento a los siguientes objetivos específicos:

- Realizar un estudio de las técnicas más relevantes de la criptografía asimétrica.
- Resaltar la importancia de la correcta elección de los parámetros de seguridad.
- Realizar un estudio de los algoritmos más importantes para la generación de números primos y de números primos fuertes
- Explicar los resultados teóricos novedosos que dan pie a la creación del algoritmo.
- Proponer un algoritmo determinístico para chequear la primalidad de un número.
- Proponer un algoritmo para la generación de números primos fuertes.
- Analizar su fiabilidad y eficiencia.
- Implementar usando el lenguaje de programación Python (sección 3.1) el algoritmo propuesto.
- Mostrar algunas corridas comparando su eficiencia con el algoritmo usado actualmente.

Para dar cumplimiento a dichos objetivos, el autor se ha valido de diferentes métodos de la Metodología de la Investigación.

Métodos del nivel teórico:

- **Análisis-síntesis**

  Al realizar la revisión bibliográfica, para reconocer las características principales de las técnicas de criptografía asimétrica y los algoritmos más importantes para la generación de números primos.

- **Inducción-deducción**

  Posibilitó realizar inferencias y deducciones de los principales sustentos teóricos que fundamentan la investigación, permitiendo la creación de nuevas ideas que condujeron a la solución de la problemática presentada.

- **Histórico-lógico**

  Se utilizó para ver la evolución del tema, principalmente en el proceso de evolución de la criptografía y el uso de los números primos en ella, viendo así la necesidad de la constante investigación y desarrollo de nuevas teorías en esta rama.

Métodos del nivel empírico

- **Análisis documental**
  Este método fue considerado para la revisión bibliográfica de un grupo relevante de documentos relacionados con el tema, el cual incluye artículos, libros, tesis, datos de la práctica y sitios de Internet.

- **Análisis grafico**

  Este método fue considerado, para la realización de una comparación del resultado original, obtenido por el autor, con los resultados clásicos.

Capítulo 1 Fundamentos Matemáticos

Este capítulo está constituido por una colección de elementos de la matemática que provee al lector de una noción básica de las notaciones importantes, métodos, operaciones algebraicas y los conceptos para el posterior desarrollo de este trabajo.

1.1 Álgebra Abstracta y Álgebra modular

En este epígrafe se introducen algunas estructuras algebraicas que son los conceptos centrales del algebra abstracta y además se brindan los elementos y operaciones en los que se fundamenta la criptografía moderna. [Mao, 2003]

1.1.1 Grupos

Definición 1.1.1.1 Grupo: Un grupo $(G,\circ)$ es un conjunto G y una operación $\circ$ que satisface los siguientes axiomas:

1. $\forall a, b \in G$ se tiene que $a \circ b \in G$
2. $\forall a, b, c \in G$ se tiene que $a \circ (b \circ c) = (a \circ b) \circ c$
3. $\exists\, e \in G$ único que llamaremos identidad de manera que $\forall a \in G\ a \circ e = e \circ a = a$
4. $\forall a \in G\ \exists a^{-1} \in G$ que llamaremos inverso de a de manera que $a \circ a^{-1} = a^{-1} \circ a = e$

Definición 1.1.1.2 Grupos Finitos y Grupos infinitos: Un grupo $(G,\circ)$ se denomina grupo finito si el número de elementos del conjunto G es finito, de lo contrario se dice que el grupo es infinito.

Definición 1.1.1.3 Grupo Abeliano: Un grupo $(G,\circ)$ se denomina grupo abeliano si $\forall a,b \in G$ se tiene que $a \circ b = b \circ a$, en otras palabras un grupo abeliano es un grupo conmutativo.

Se debe aclarar que en algunas ocasiones se usa la notación $a^i \in G$, esto es una representación abreviada de $\underbrace{a \circ a \circ \ldots . \circ a \circ a}_{i \text{ veces}}$ puesto que la "operación" entre un entero i y el elemento a no es una operación de grupo.

Definición 1.1.1.4 Subgrupo: Se le denomina subgrupo a $(H,\circ)$ del grupo $(G,\circ)$ si $H \subseteq G$ no vacío y $(H,\circ)$ satisface los axiomas de grupo bajo la misma operación que $(G,\circ)$

Definición 1.1.1.5 Grupo Cíclico: Un grupo $(G,\circ)$ es cíclico si $\exists\, a \in G$ tal que $\forall b \in G \;\; \exists i \geq 0$ entero de manera que $b = a^i$. El elemento a es llamado generador del grupo. Cuando un grupo es generado por un elemento$a$, se puede denotar $G = \langle a \rangle$

1.1.2 Anillos

Definición 1.1.2.1 Anillo: Un anillo $(R,+,*)$ y dos operaciones, una llamada aditiva y denotada por $+$, y otra llamada multiplicativa y denotada por $*$ sobre R, que satisface los siguientes axiomas:

1. $(R,+)$ es un grupo abeliano, al elemento identidad e de dicho grupo se le denota por 0_R.
2. La operación $*$ es asociativa, o sea, cumple que: $a * (b * c) = (a * b) * c$ para todo elementos $a,b,c \in R$
3. Existe un elemento identidad para la operación $*$ denotado por $1_R \neq 0_R$ de manera que $a * 1_R = 1_R * a = a$ para todo elemento $a \in R$

4. La operación $*$ cumple la propiedad distributiva con respecto a $+$, o sea que: $a * (b + c) = (a * b) + (a * c)$ para todo elementos $a, b, c \in R$

Un anillo se dice que es conmutativo si cumple que: $a * b = b * a$

Definición 1.1.2.2: Elemento Invertible: Un elemento $a \in R$ se denomina invertible si existe un elemento $b \in R$ de manera que $a * b = 1_R$

1.1.3 Campos

Definición 1.1.3.1 Campo: Se denomina campo a un anillo $(R, +, *)$ conmutativo donde todos los elementos distintos al elemento identidad de $+$ (0_R) tienen un inverso multiplicativo.

Definición 1.1.3.2 Característica de un Campo: La característica de un campo se define como el menor entero positivo m tal que $\underbrace{1_R + 1_R + \cdots + 1_R}_{m \text{ veces}} = 0_R$, si no existe dicho entero, entonces se dice que la característica del Campo es 0

1.1.4 Algebra modular y $\mathbb{Z}_n$

Definición 1.1.4.1 Congruencia Módulo n: Sean a, b enteros y n un entero positivo, se dice que a es congruente a b módulo n y se denota $a \equiv b \bmod(n)$ si n divide exactamente a $(a - b)$.

La definición **1.1.4.1** es equivalente a la siguiente: $a \equiv b \bmod(n) \Leftrightarrow$ Se obtiene el mismo resto al dividir a y b por n

La Relación Congruencia módulo n cumple las siguientes propiedades:

1. $a \equiv a \bmod(n)$. A esta propiedad se le llama propiedad Reflexiva
2. Si $a \equiv b \bmod(n)$ entonces $b \equiv a \bmod(n)$. A esta propiedad se le llama propiedad Simétrica.
3. Si $a \equiv b \bmod(n)$ y $b \equiv c \bmod(n)$ entonces $a \equiv c \bmod(n)$. A esta propiedad se le llama propiedad Transitiva.
4. Si $a \equiv a_1 \bmod(n)$ y $b \equiv b_1 \bmod(n)$ entonces $a + b \equiv a_1 + b_1 \bmod(n)$ y además $a * b \equiv a_1 * a_2$

Se dice clase de equivalencia de un entero a al conjunto de todos los enteros congruentes a a módulo n. Siguiendo las propiedades $1, 2, 3$ (Reflexiva, Simétrica y Transitiva), para un entero positivo fijo n, la relación congruencia módulo n particiona los enteros ($\mathbb{Z}$) en n clases de equivalencia.

Definición 1.1.4.2 $\mathbb{Z}_n$**:** Los enteros módulo n, denotados por $\mathbb{Z}_n$, son el conjunto de enteros $\{0, 1, 2, \dots, n-1\}$. Cada entero de $\mathbb{Z}_n$ representa una partición de $\mathbb{Z}$, donde cada partición representada por un elemento de $\{0, 1, 2, \dots, n-1\}$ está compuesta por todos los enteros que son congruentes al elemento modulo n

Las operaciones adición, sustracción, y multiplicación en $\mathbb{Z}_n$ son realizadas módulo n. Es un hecho que $(\mathbb{Z}_n, + \bmod(n))$ es un Grupo Abeliano.

Definición 1.1.4.3: Sea $a \in \mathbb{Z}_n$, el inverso multiplicativo de $a \bmod(n)$ es un entero $x \in \mathbb{Z}_n$ de manera que $a * x \equiv 1 \bmod(n)$

El conjunto $\mathbb{Z}_n$ con las operaciones suma y multiplicación módulo n, o sea, $(\mathbb{Z}_n, + \bmod(n), * \bmod(n))$ es un Anillo Conmutativo. Además es un hecho que $(\mathbb{Z}_n, + \bmod(n), * \bmod(n))$ es un Campo si y solo si n es un número primo. También se conoce que si n es primo entonces $\mathbb{Z}_n$ tiene característica n.

Definición 1.1.4.4 $\mathbb{Z}_p^*$**:** El Grupo multiplicativo de $\mathbb{Z}_n$ es $\mathbb{Z}_p^* = \{a \in \mathbb{Z}_n; mcd(a, n) = 1\}$. En particular si p es un número primo, entonces $\mathbb{Z}_p^* = \{a \in \mathbb{Z}_n; 1 \leq a \leq n-1\}$

Definición 1.1.4.5: Sea $a \in \mathbb{Z}_n^*$. Se define como orden de a, se denota como $ord(a)$, al menor entero positivo t de manera que $a^t \equiv 1 \bmod(n)$.

Definición 1.1.4.6: Sea $a \in \mathbb{Z}_n^*$. Si $ord(a) = \phi(n)$ entonces a es llamado generador o elemento primitivo de $\mathbb{Z}_n^*$. Además, si $\mathbb{Z}_n^*$ tiene un generador, entonces $\mathbb{Z}_n^*$ es un grupo cíclico con la operación multiplicación módulo n. $(\phi(n))$ se refiere a la función Phi de Euler, que se definirá más adelante.

Ejemplo: Sea $p = 97$, $\mathbb{Z}_p{}^*$ es un grupo cíclico de orden $n = 96$. Puesto que: Sea $a = 5 \in \mathbb{Z}_{97}^*$, el menor entero t tal que $5^t \equiv 1 \bmod(97)$ es $t = 96 = \phi(97) = 97 - 1$ por ser 97 primo.

1.2 Teoría de Números

La teoría de números es la rama de las matemáticas que estudia las propiedades de los números, en particular los enteros, pero más en general, estudia las propiedades de los elementos de dominios enteros (anillos conmutativos con elemento unitario y cancelación) así como diversos problemas derivados de su estudio. Contiene una cantidad considerable de problemas que podrían ser comprendidos por "no matemáticos". A continuación se darán algunas definiciones básicas de la teoría de números de gran utilidad para el posterior entendimiento del trabajo.

Definición 1.2.1 Divisibilidad: Sea $a \in \mathbb{Z}$, se dice que es divisible o exactamente divisible por un número $b \neq 0$ y se denota por $a|b$ si $\exists\, q \in \mathbb{Z}$ de manera que $a = bq$

Definición 1.2.2 Máximo Común Divisor: Se define el máximo común divisor entre dos o más números enteros, al mayor número entero que divide exactamente a todos.

Definición 1.2.3 Número Primo: Un número primo es, por definición, un entero positivo mayor que 1 que es divisible, solamente, por sí mismo y la unidad.

Definición 1.2.4 Coprimos: Dos números enteros a y b son números primos entre sí (o *coprimos*, o *primos relativos*) si no tienen ningún factor primo en común, o, dicho de otra manera, si no tienen otro divisor común más que 1

1.2.4 Pequeño Teorema de Fermat:

El pequeño teorema de Fermat es uno de los teoremas clásicos de teoría de números relacionado con la divisibilidad. Se formula de la siguiente manera:

Si p es un número primo, entonces $\forall\, \mathrm{a} \in \mathbb{N},\ \mathrm{a}^{\mathrm{p}} \equiv \mathrm{a}\ \mathrm{mod}(\mathrm{p})$, o lo que es equivalente $\forall\, \mathrm{a} \in \mathbb{N},\ \mathrm{a}^{\mathrm{p}-1} \equiv 1\ \mathrm{mod}(\mathrm{p})$. La demostración de este teorema se puede consultar en (Garret, 1997)

1.2.5 La Función $(\boldsymbol{\phi}(\mathbf{n}))$ Phi de Euler

Si n es un número entero, la cantidad de enteros entre 1 y n que son primos relativos con n se denota como $\phi(\mathrm{n})$, formalmente se puede definir como $\phi(\mathrm{m}) = |\{\mathrm{n} \in \mathbb{N}; \mathrm{n} \leq \mathrm{m} \wedge \mathrm{mcd}(\mathrm{m}, \mathrm{n}) = 1\}|$ donde $|\,.\,|$ significa cantidad de elementos.

Algunas Propiedades de la función phi de Euler

- $\phi(1) = 1$ Se define de esta manera.
- $\phi(\mathrm{p}) = \mathrm{p} - 1$ siempre que p sea un número primo.
- $\phi(\mathrm{p}^{\mathrm{k}}) = (\mathrm{p} - 1)\mathrm{p}^{\mathrm{k}-1}$ siempre que p sea un número primo $\forall\, \mathrm{k} \in \mathbb{N}$.
- $\phi(\mathrm{mn}) = \phi(\mathrm{m})\phi(\mathrm{n})$ siempre que m, n sean primos relativos entre sí.

1.2.6 Símbolo de Jacobi

Definición 1.2.6.1 Residuo cuadrático: Se denomina residuo cuadrático módulo n a cualquier entero r coprimo con n para el cual tenga solución la ecuación de congruencia $x^2 \equiv r\ mod(n)$

Definición 1.2.6.2 Símbolo de Legendre: El símbolo de Legendre es una función multiplicativa utilizada en teoría de números que toma como argumentos un entero a y un primo p y devuelve uno de los valores $1, -1, o\ 0$ dependiendo de si $a|p$ o a es o no residuo cuadrático módulo p, es decir si la congruencia $x^2 \equiv a\ mod(p)$ tiene solución. Se denota como $\left(\frac{a}{p}\right)$.

Definición 1.2.6.3 Símbolo de Jacobi: En esencia se puede considerar como una generalización del símbolo de Legendre para valores impares de n que no necesariamente han de ser primos.

$$\left(\frac{a}{n}\right) = \begin{cases} 0\ si\ n|a \\ 1\ si\ \exists\ x;\ x^2 \equiv a\ mod(n) \\ -1\ si\ \nexists\ x;\ x^2 \equiv a\ mod(n) \end{cases}$$

1.2.7 Distribución de los números primos

Los números primos son infinitos, esto es un hecho demostrado desde el año 300 a.n.e, cuando Euclides en su libro Los Elementos da a conocer una ingeniosa demostración por reducción al absurdo (se cree que esta es la primera vez en la historia que se usa esta técnica tan conocida por los

matemáticos), entonces cabe preguntarse cómo se distribuyen los números primos en el conjunto de los números naturales, es decir, cuán frecuentes son y donde se espera encontrar el n-esimo número primo. Para dar respuesta a estas interrogantes se introduce $\pi(n)$ la función enumerativa de los números primos o función de distribución

Teorema 1.2.7.1: Teorema de los Números Primos: Sea la función $\pi(n)$, función de distribución de los números primos, sea $g(n) = \frac{n}{\ln(n)}$, entonces $\lim_n \frac{\pi(n)}{g(n)} = 1$, o sea,

$$\pi(n) \sim \frac{n}{\ln(n)}$$

Este resultado que data del siglo XVIII, ha sido notablemente mejorado con las oportunidades que ofrecen los ordenadores de manera que hoy en día resulta fácil conocer cuántos números primos hay en cierto rango.

1.3 Ecuaciones diofánticas.

Se llama ecuación diofántica a cualquier ecuación algebraica, de dos o más incógnitas, cuyos coeficientes recorren el conjunto de los números enteros, de las que se buscan soluciones enteras, esto es, que pertenezcan al conjunto de los números enteros. Formalmente una ecuación $f(x_1, x_2, \dots, x_n) = b$ es llamada ecuación diofántica si y solo si $f(x_1, x_2, \dots, x_n)$ es un polinomio en $x_1, x_2, \dots, x_n$ con coeficientes enteros, b constante tal que $b \in \mathbb{Z}$ y solo son de interés las soluciones enteras. (Bronshtein, Semendyayev, et al, 2005)

Sobre el tema de ecuaciones diofánticas existe una extensa bibliografía, en relación a sus diversos tipos y métodos de solución, en el presente trabajo solamente se trata un tipo de ecuación diofántica: Ecuaciones Diofánticas Cuadráticas.

Definición 1.3.1 Ecuación diofántica cuadrática: La ecuación diofántica cuadrática tiene la forma $ax^2 + bxy + cy^2 + dx + ey + f = 0$.

Esta ecuación representa un cono en el plano cartesiano, se desea reducirla hasta obtener una forma canónica. Se introduce el discriminante de la ecuación $\Delta = b^2 - 4ac$.

- Si $\Delta < 0$ el cono define una elipse.
- Si $\Delta = 0$ el cono define una parábola.
- Si $\Delta > 0$ el cono define una hipérbola.

Es un hecho que las ecuaciones de este tipo son reducibles a un tipo de ecuación diofántica cuadrática llamada Ecuación del Tipo Pell (Andreescu, 2015)

1.3.2 Ecuaciones diofánticas de Pell

Euler atribuyó los primeros estudios serios de las soluciones no triviales de la ecuación de la forma $u^2 - Dv^2 = 1$ a John Pell. Aunque se descubrió que Pell nunca había considerado el estudio de estas ecuaciones, y a quien se le debía asignar este mérito era a Fermat, pasaron a la historia como las ecuaciones de Pell.

Teorema 1.3.2.1: Si D es un entero positivo, no es un cuadrado perfecto entonces la ecuación $u^2 - Dv^2 = 1$ tiene infinitas soluciones naturales y la forma general de la solución está dada por $(u_n, v_n)\ con\ n \geq 0$ donde $u_{n+1} = u_1u_n + Dv_1v_n$, $v_{n+1} = v_1u_n + u_1v_n$ donde (u_1, v_1) es la solución trivial.

1.4 Complejidad Algorítmica

Definición 1.4.1 Algoritmo: Un algoritmo es un conjunto de pasos bien definido que toma una entrada y produce una salida.

Algoritmos relacionados a problemas computacionales: Un problema computacional es una deseada relación entre una entrada y una salida. Un algoritmo resuelve un problema computacional si logra producir la relación deseada.

Complejidad en tiempo y espacio. La notación de la O grande.

El hecho de que no se conozca un algoritmo eficiente para resolver un problema no quiere decir que éste no exista, y por eso es importante la Teoría de Algoritmos para la Criptografía. Si, por ejemplo, se lograra descubrir un método eficiente capaz de resolver logaritmos discretos o capaces de descomponer en factores primos números compuestos (secciones 2.2 y 2.3), algunos de los algoritmos asimétricos más populares en la actualidad dejarán de ser seguros. De hecho, la continua reducción del tiempo de ejecución necesario para resolver ciertos problemas, propiciada por la aparición de algoritmos más eficientes, junto con el avance de las prestaciones del hardware disponible, obliga con relativa frecuencia a actualizar las previsiones sobre la seguridad de muchos sistemas criptográficos.

Cuando un programa se ejecuta en una computadora, dos de las más importantes consideraciones que deben tenerse son: cuánto tiempo le tomará y cuánta memoria ocupará. Hay otras cuestiones como si el programa funciona, pero las dos consideraciones de tiempo de cómputo y espacio de memoria son dominantes. Por ejemplo, en grandes computadoras, la atención a procesos de los usuarios cambia dependiendo del tiempo en que un proceso debe ejecutarse y cuánta memoria utiliza. Aun en computadoras pequeñas, se desea que un programa ejecute rápidamente y no exceda la cantidad de memoria disponible.

En la mayoría de los casos carece de interés calcular el tiempo de ejecución concreto de un algoritmo en una computadora, e incluso algunas veces simplemente resulta imposible. En su lugar se ha empleado una notación de tipo asintótico, que permitirá acotar dicha magnitud. Normalmente se considera

el tiempo de ejecución del algoritmo como una función $f(n)$ en función del tamaño n de la entrada de dicho algoritmo (Lucena, 2001).

Definición 1.4.2 Límite superior asintótico: Dada la función $f(n)$ entonces: $f(n) = O(g(n))$ si existe una contante c y un tamaño n_0 de manera que $0 \leq f(n) \leq cg(n)$ para todo $n \geq n_0$

Definición 1.4.3 Límite inferior asintótico: Dada la función $f(n)$ entonces: $f(n) = \Omega(g(n))$ si existe una contante c y un tamaño n_0 de manera que $0 \leq cg(n) \leq f(n)$ para todo $n \geq n_0$

Definición 1.4.4 Límite exacto asintótico: Dada la función $f(n)$, entonces $f(n) = \Theta(g(n))$ si existen dos números positivos c_1, c_2 y un tamaño n_0 de manera que $c_1 g(n) \leq f(n) \leq c_2 g(n)$ para todo $n \geq n_0$

Intuitivamente, $f(n) = O(g(n))$ significa que $f(n)$ crece asintóticamente no más rápido que $g(n)$ multiplicada por una constante. Análogamente $f(n) = \Omega(g(n))$ quiere decir que $f(n)$ crece asintóticamente al menos tan rápido como $g(n)$ multiplicada por una constante. Se definirán ahora algunas propiedades sobre la notación que se acaban de introducir.

a) $f(n) = O(g(n)) \Leftrightarrow g(n) = \Omega(f(n))$

b) $f(n) = \Theta(g(n)) \Leftrightarrow f(n) = O(g(n)) \wedge f(n) = \Omega(g(n))$.

c) $f(n) = O(h(n)) \wedge g(n) = O(h(n)) \Longrightarrow (f + g)(n) = O(h(n))$

d) $f(n) = O(h(n)) \wedge g(n) = O(l(n)) \Longrightarrow (f(n) * g(n)) = O(h(n) * l(n))$

e) $f(n) = O(f(n))$.

f) $f(n) = O(g(n)) \wedge g(n) = O(h(n)) \Longrightarrow (f)(n) = O(h(n))$

Operaciones elementales

Se puede considerar una operación elemental como aquella que se ejecuta siempre en tiempo constante. Evidentemente, en función de las características concretas de la computadora que se esté manejando, habría operaciones que

podrán considerarse elementales o no. Por ejemplo, en una computadora que pueda operar únicamente con números de $16\,\text{bits}$, no podría considerarse elemental una operación con números de 32 bits. En general, el tamaño de la entrada a un algoritmo se mide en bits, y se consideran en principio elementales únicamente las operaciones a nivel de bit. Por ejemplo sean a y b dos números enteros positivos, ambos menores o iguales que n. Se necesita, pues, aproximadamente $\log_2 \text{n}\ \text{bits}$ para representarlos, nótese que en este caso, $\log_2 \text{n}$ es el tamaño de la entrada. Según este criterio, las operaciones aritméticas, llevadas a cabo mediante los algoritmos tradicionales, presentan los siguientes órdenes de complejidad:

Suma $(a+b)$: $O(log_2 a + log_2 b) = O(log_2 n)$
Resta $(a-b)$: $O(log_2 a + log_2 b) = O(log_2 n)$
Multiplicación $(a*b)$: $O(log_2 a * log_2 b) = O((log_2 n)^2)$
División $\left(\frac{a}{b}\right)$: $O(log_2 a * log_2 b) = O((log_2 n)^2)$

Como a lo largo de este trabajo son muy utilizadas las operaciones modulares, vale destacar que éstas también son elementales.

Suma modular $(a + b\ mod(n))$: $O(log\ n)$
Resta modular $(a - b\ mod(n))$: $O(log\ n)$
Multiplicación modular $(a * b\ mod(n))$: $O((log\ n)^2)$
Inverso modular a^{-1}: $O((log\ n)^2)$
Exponenciación modular $a^k\ mod(n), k < n$: $O((log\ n)^3)$

Algoritmos Polinomiales, Exponenciales y Subexponenciales

Diremos que un algoritmo es polinomial si su peor caso de ejecución es de orden $\text{O}(\text{n}^\text{k})$ donde n es el tamaño de la entrada y k una constante. Adicionalmente, cualquier algoritmo que no pueda ser acotado por una función polinomial, se conoce como exponencial. En general, los algoritmos polinomiales se consideran eficientes, mientras que los exponenciales se

consideran ineficientes. Un algoritmo se denomina subexponencial si en el peor de los casos, la función de ejecución es de la forma $e^{O(n)}$

Tabla 3.4.1 Algunos Órdenes de complejidad más conocidos

N	$O(\log_2 n)$	$O(n^2)$	$O(n \log_2 n)$	$O(n^2)$	$O(2^n)$	$O(n!)$
10	3 µs	10 µs	30 µs	0.1 ms	1 ms	4 s
25	5 µs	25 µs	0.1 ms	0.6 ms	33 s	10^{11} años
50	6 µs	50 µs	0.3 ms	2.5 ms	36 años	...
100	7 µs	100 µs	0.7 ms	10 ms	10^{17} años	...
1000	10 µs	1 ms	10 ms	1 s	...	...
10000	13 µs	10 ms	0.1 s	100 s	...	...
100000	17 µs	100 ms	1.7 s	3 horas	...	...
1000000	20 µs	1 s	20 s	12 días	...	...

Nota: Lo que se denota como μs son microsegundos $1\mu s = 10^{-6}s$

Conclusiones del capítulo

En este capítulo se ha realizado un breve recorrido por las teorías matemáticas básicas que respaldan las técnicas de criptografía que se exponen en capítulos posteriores, así como un resumen de la teoría de complejidad necesaria para realizar el análisis de complejidad del algoritmo propuesto. Definiciones y propiedades del Algebra Abstracta, Algebra modular y Teoría de Números permiten formalizar matemáticamente cada paso realizado en cada una de las técnicas mencionadas en el capítulos dos, un ejemplo sería, es necesario para realizar el protocolo de cambio de clave Diffie-Hellman (sección 2.1) generar el grupo cíclico $\mathbb{Z}_p^*$ tal que p sea un número primo, otro ejemplo, los test de primalidad usados para la generación de parámetros de seguridad (sección 2.6) están basados en el Pequeño Teorema de Fermat y usan elementos como el símbolo de Jacobi, congruencia módulo n, y otros tratados en este capítulo. En la sección 2.7 se realiza la propuesta de un Test de primalidad original, el cual chequea la primalidad de un número mediante la resolución de ecuaciones diofánticas cuadráticas. La resolución de estas ecuaciones

diofánticas, o criterios sobre su solubilidad, es de vital importancia puesto que la decisión de si un número es primo, o no, viene dada por la condición de si dichas ecuaciones tienen soluciones positivas, o no. Tras la presentación del mencionado test y la creación de un algoritmo para la generación de números primos fuertes mediante el uso del test, es necesario analizar las conjeturas realizadas por el autor sobre la eficiencia de estos algoritmos (sección 2.7), aquí interviene el análisis de complejidad, sumamente importante para la verificación o negación de las conjeturas.

Capítulo 2 Propuesta de Algoritmo para Generar Números primos Fuertes

Técnicas de cifrado Asimétrico.

Cifrados clásicos, como el cifrado de César, dependen de mantener todo el proceso de cifrado en secreto. En cambio en los cifrados modernos, como DES o AES, los detalles algorítmicos se realizan públicamente, demostrando así que la seguridad de estos sistemas criptográficos radica en la elección de la clave secreta de cifrado. En este capítulo se dará una breve introducción sobre los métodos y protocolos básicos de la criptografía moderna de clave pública. Se debe aclarar que las nociones básicas que se muestran en este capítulo son etiquetadas como “Cifrados de libro de texto”, pues dichos algoritmos son fácilmente encontrados en la mayoría de los libros sobre Criptografía o temas similares, es por esto que estos no son propicios para su uso real, los algoritmos y protocolos que están siendo usados generalmente son de propiedad confidencial, sin embargo el basamento teórico de los mismos radica en estos “Cifrados de libro de texto”.

En los cuatro primeros epígrafes de este capítulo se muestran las nociones básicas de las técnicas de cifrado público más relevantes: el protocolo de intercambio de clave Diffie-Hellman y los criptosistemas RSA, Rabin y ElGamal y consecuentemente se introducen los problemas del Logaritmo Discreto y Factorización de Enteros, que es donde radica la complejidad de las técnicas. Del análisis que se realiza es importante notar que la búsqueda de los parámetros para generar la clave pública es un prerrequisito de suma importancia. Ejemplos específicos serian: el requerimiento de un número primo p para definir el grupo $\mathbb{Z}_p^*$ en el algoritmo de Diffie-Hellman y en el Criptosistema ElGamal, también se requieren los primos p, q para el cálculo $N = pq$ en los Criptosistemas RSA y Rabin. De aquí que el resto de los epígrafes se destinen a la problemática de generación de parámetros de clave pública mediante la valoración de varios tipos de test, tanto probabilísticos como deterministas.

2.1 Protocolo de intercambio de clave Diffie-Hellman

En Criptosistemas simétricos o de clave privada es necesario transferir una clave secreta a ambas partes que se comunican antes de que estas puedan realizar una comunicación segura. Antes del nacimiento de la criptografía de clave pública, el intercambio de clave secreta representaba un gran problema pues era necesario un canal seguro para realizar el intercambio. Uno de los avances que la criptografía de clave pública brinda a sistemas criptográficos de clave privada son los protocolos de cambio de clave, mediante los que se puede realizar un intercambio de clave secreta sin la necesidad de un canal seguro. El primero de estos protocolos fue propuesto por Whitfield Diffie y Martin Hellman en 1976 y es conocido como el Protocolo de intercambio de clave Diffie-Hellman.

Para comenzar ambas partes que se comunican acuerdan un número primo p y un generador arbitrario g del grupo cíclico $\mathbb{Z}_p^*$

Protocolo 2.1 Protocolo de Intercambio de clave Diffie-Hellman

Entrada Común: (p, g): p es un número primo suficientemente grande, g un generador del grupo cíclico $\mathbb{Z}_p^*$

Salida: k, un elemento de $\mathbb{Z}_p^*$ compartido entre las partes que se comunican (A y B).

1. A selecciona un entero $a \in [1, p-1]$; calcula $g_a \leftarrow g^a \, mod(p)$ y se lo envía a B;
2. B selecciona un entero $b \in [1, p-1]$; calcula $g_b \leftarrow g^b \, mod(p)$ y se lo envía a A;
3. A calcula $k \leftarrow {g_b}^a \, mod(p)$;
4. B calcula $k \leftarrow {g_a}^b \, mod(p)$

Se comprueba que la parte A ha obtenido $k = g^{ba}\, mod(p)$ y la parte B ha obtenido $k = g^{ab}\, mod(p)$, como $ab \equiv ba\, mod(p-1)$ ambas partes han obtenido el mismo valor.

2.2 El Problema de Diffie-Hellman y el Problema del Logaritmo Discreto

La seguridad del protocolo Diffie-Hellman de intercambio de clave radica en la complejidad que supone el cálculo computacional del elemento g^{ab} dados los elementos g_a y g_b. Este problema es llamado el Problema de Diffie-Hellman (CDH por las siglas en inglés Computational Diffie-Hellman Problem) y conduce a la definición siguiente (Mao, 2003).

Definición 2.2.1 CDH

El proceso CDH requiere:

Entrada: (p, g): p es un número primo suficientemente grande, g un generador del grupo cíclico $\mathbb{Z}_p^*$, $g^a\, mod(p)$, $g^b\, mod(p)$

Salida: $g^{ab}\, mod(p)$.

La complejidad de encontrar este elemento $g^{ab}\, mod(p)$ está en determinar uno de los elementos a o b dado $g^a\, mod(p)$ o $g^b\, mod(p)$. Este problema es el llamado Problema del Logaritmo Discreto (DLP por sus siglas en inglés). La noción requerida se caracteriza por la definición siguiente (Menezes, 1997).

Definición 2.2.2 Logaritmo Discreto

Sea G un grupo cíclico de orden n, α un generador de G y $\beta \in G$ el logaritmo discreto de β en base α, denominado $\log_\alpha \beta$, es un entero único $x \in [0, n-1]$ de manera que $\beta = \alpha^x$. (Menezes, 1997)

Ejemplo: Retomando el ejemplo de la definición 1.1.4.6**:** Sea $p = 97$, $\mathbb{Z}_p{}^*$ es un grupo cíclico de orden $n = 96$. Un generador del grupo es $\alpha = 5$. Como $5^{32} \equiv 35\, mod(97)$, entonces $\log_5 35 = 32$ en $\mathbb{Z}_{97}{}^*$.

Definición 2.2.3 DLP

Dado un número primo p, g un generador del grupo cíclico $\mathbb{Z}_p^*$, $\beta \in \mathbb{Z}_p^*$ encontrar un entero $x \in [0, p-1]$ de manera que $\beta \equiv g^x\, mod(p)$.

Obsérvese que la dificultad de implementar CDH depende de la correcta elección del parámetro p

2.3 El Problema de RSA y El Problema de la Factorización de Enteros

La seguridad del criptosistema RSA radica en la complejidad de cómputo del cálculo de la raíz e-esima del mensaje cifrado $c\ mod(N)$. Este problema es el llamado Problema de RSA (RSAP por sus siglas en inglés). La concepción del RSAP está dada por la definición siguiente.(Mao, 2003)

Definición 2.3.1 RSAP

Entrada: $N = pq$, con p, q números primos; $e \in [1, \varphi(N)]$ de manera que $mcd\big(e, \varphi(N)\big) = 1$; $c \in \mathbb{Z}_N^*$

Salida: un entero único $m \in \mathbb{Z}_N^*$; $m^c \equiv c\, mod(N)$

La dificultad del RSAP depende a su vez de la dificultad de que dado un número entero compuesto N encontrar su descomposición en factores primos. Este es el llamado Problema de la Factorización de Enteros (IFP por sus siglas en inglés) que se define seguidamente Referencia (Mao, 2003)

Definición 2.3.2 IFP

El problema de la factorización de enteros (IFP): Dado un número entero compuesto e impar N con al menos dos factores primos distintos encontrar un p primo de manera que p divida exactamente a N

2.4 El Criptosistema de Rabin

Michael Oser Rabin desarrolló un criptosistema de clave pública, basado en la dificultad computacional de encontrar la raíz cuadrada módulo un entero compuesto. El trabajo de Rabin tiene una gran importancia teórica; provee del primer criptosistema de seguridad probable: la seguridad del Criptosistema de Rabin es exactamente la dificultad del Problema de Factorización de Enteros anteriormente tratado.

Algoritmo 2.4 Criptosistema de Rabin

Generación de Claves:

1. Se escogen p, q números primos arbitrarios
2. Calcular $N = pq$
3. Se escoge un entero arbitrario $b \in [1, N-1]$
4. Se hace público el par (N, b) y se mantiene (p, q) como clave privada

Cifrado:

Para enviar un mensaje confidencial m a la parte A, el emisor B obtiene el mensaje cifrado c de manera siguiente: $c \leftarrow m(m+b)\ mod(N)$

Descifrado:

Para obtener el mansaje original m, A calcula las raíces de la ecuación

cuadrática $m^2 + bm - c \equiv 0\ mod(N)$ para $m < N$

2.5 Criptosistema ElGamal

El procedimiento de cifrado y descifrado ElGamal se refiere a un esquema de cifrado basado en el problema matemático del logaritmo discreto. Es un algoritmo de criptografía asimétrica basado en la idea de Diffie-Hellman y que funciona de una forma parecida a este algoritmo discreto. El algoritmo de ElGamal puede ser utilizado tanto para generar firmas digitales como para cifrar o descifrar. Fue descrito por Taher ElGamal en 1984. La seguridad del algoritmo se basa en la suposición de que la función utilizada es una función trampa de un sólo sentido debido a la dificultad de calcular un logaritmo discreto. El procedimiento de cifrado (y descifrado) está basado en cálculos sobre un grupo cíclico cualquiera G, lo que lleva a que la seguridad del mismo dependa de la dificultad de calcular logaritmos discretos en G.

Algoritmo 2.5 El Criptosistema ElGamal

Generación de Claves:

1. Se selecciona un número primo arbitrario p;
2. Se busca un generador g del grupo cíclico $\mathbb{Z}_p^*$
3. Se selecciona un entero $x \in [1, p-1]$ como clave privada
4. Se calcula la clave publica $y \leftarrow g^x\ mod(p)$
5. Se hace público la terna (p, g, y) y se mantiene x como clave privada

Cifrado:

Para enviar un mensaje confidencial m a la parte A, el emisor B selecciona un entero $k \in [1, p-1]$ y obtiene el par $(c_1 \leftarrow g^k\ mod(p),\ c_2 \leftarrow y^k m\ mod(p))$

Descifrado:

Para obtener el mansaje original m, A calcula $m \leftarrow {}^{c_2}/_{{c_1}^x} \ mod(p)$

2.6 Generación de Parámetros de Clave Pública.

Generalmente, en criptografía, se le denomina parámetro de seguridad de clave pública, o parámetros de clave pública a ciertos números que cumplen algunas propiedades especiales sobre los cuales radica la complejidad de romper los protocolos y criptosistemas. A lo largo de este capítulo se han nombrado algunas de estas propiedades, como que el parámetro debe ser un número primo, además se ha trabajado con el término número primo grande, pero ¿Qué es un número primo grande? Debido a que la idea de un número grande es un poco ambigua, en este trabajo cuando hablemos de número primo grande nos referiremos a la siguiente definición.

Definición 2.6.1 Números Primos Grandes: se considera que un número es grande si tiene longitud al menos de 512 bits (155 dígitos), a causa de que los procesadores actuales manejan solo números de 32 bits, se tienen que diseñar programas para poder efectuar las operaciones sobre este tipo de números.

El método más natural para generar un número primo grande seria generar un número impar arbitrario del tamaño apropiado y chequear si este número es primo, entonces nos enfrentamos ahora a un nuevo problema ¿cómo certificar que un número es primo? Si se le pregunta a un escolar lo más probable es que conteste que miremos la tabla de número primos que tiene en su libro de textos (si es muy vago. . . o muy listo). O bien podría contestar que probemos a dividir por todos los números menores que él (ahora bien, si es realmente listo dirá que solo hace falta hasta su raíz cuadrada). Estas ideas simples fueron realmente los primeros test de primalidad. El primero es conocido como la criba de Eratóstenes. Este algoritmo tiene la particularidad de que no sólo certifica si un número es primo o no, sino que da todos los primos menores que él. El algoritmo funciona, pues estamos comprobando si es o no múltiplo de algún número menor que él, lo cual es equivalente a la definición de número primo.

Este método es óptimo cuando se necesita construir la tabla con todos los números primos hasta n, pero es tremendamente ineficiente para el actual propósito, pues se trata de un algoritmo exponencial tanto en tiempo como en espacio. Sorprendentemente, no se tiene constancia del método trivial hasta el siglo XII cuando Leonardo de Pisa (más conocido como Fibonacci) introduce el método de las divisiones sucesivas.

Algoritmo 2.6 División Sucesiva
Entrada: $n \in \mathbb{Z}$ **Salida:** Una respuesta "Primo" o "Compuesto" a la pregunta ¿Es n primo? 1. Mientras que $i < \sqrt{n}$ hacer: Si $n \equiv 0\ mod(i)$ entonces devolver "Compuesto" 2. Devolver "Primo"

2.6.1 Test de Fermat

El Teorema de Fermat asegura que si p es primo, a un entero; $1 \leq a \leq p-1$, entonces $a^{p-1} \equiv 1\ mod(p)$. Por lo que para probar que un posible primo p es compuesto, basta con encontrar algún entero $1 \leq a \leq p-1$; $a^{p-1} \not\equiv 1\ mod(p)$. Este teorema motiva la obtención del resultado siguiente. Si el algoritmo 2.5.1 devuelve "Compuesto", se puede afirmar con seguridad que n es compuesto, sin embargo, si dicho algoritmo devuelve "Primo" esto no es prueba de que el número n es de hecho primo. Los números de Carmichael, por ejemplo, son números enteros compuestos n que cumplen que $a^{n-1} \equiv 1\ mod(n)$ para todo entero a que satisfaga que $mcd(a,n) = 1$. Al aplicar el algoritmo 2.5.1 a un número de Carmichael, aun cuando el parámetro de seguridad t sea un número grande, este nos devolverá una falsa primalidad.

Algoritmo 2.6.1 Test de Primalidad de Fermat
Entrada: Un entero impar $n \geq 3\ y\ un\ parametro\ de\ seguridad$ t

Salida: Una respuesta “Primo” o “Compuesto” a la pregunta ¿Es n primo?

1. Para i entero que recorre desde 1 hasta t hace:
   - **1.1.** Elegir un número entero aleatorio $2 \leq a \leq n-1$
   - **1.2.** Calcular $r \leftarrow a^{n-1}\ mod(n)$
   - **1.3.** Si $r \neq 1$ devolver “Compuesto”
2. Devolver “Primo”

El algoritmo tiene complejidad $O((\log n)^3$, (Borges, 2005). La probabilidad de error descenderá dependiendo del número de veces que se repita el proceso, pero en el caso de encontrar un número de Carmichael seguirá fallando. Las deficiencias del Test de Fermat son corregidas en los test de Solovay-Strassen y Miller-Rabin. Este primer ejemplo sirve de introducción a lo que será común a lo largo de este capítulo, los tests de primalidad probabilistas. Este tipo de algoritmos son muy rápidos a costa de poder fallar en ciertas ocasiones.

2.6.2 Test de Solovay-Strassen

El test probabilístico de primalidad Solovay-Strassen el primer test usado en criptosistemas de clave pública como RSA. Actualmente no es usado pues el Test de Miller-Rabin es más eficiente.

El Criterio de Euler asegura que si p es un número primo impar entonces $a^{p-1}/_2 \equiv \left(\frac{a}{p}\right)\ mod(p)$. La notación $\left(\frac{a}{p}\right)$ se refiere al Símbolo de Jacobi (véase definición)

Algoritmo 2.6.2 Test de Solovay-Strassen

Entrada: Un entero impar $n \geq 3\ y\ un\ parametro\ de\ seguridad$ t

Salida: Una respuesta “Primo” o “Compuesto” a la pregunta ¿Es n primo?

1. Para i entero que recorre desde 1 hasta t hace:

1.1. Elegir un número entero aleatorio $2 \leq a \leq n-1$

1.2. Calcular $r \leftarrow a^{\frac{n-1}{2}} \, mod(n)$

1.3. Si $r \neq 1$ y $r \neq n-1$ devolver "Compuesto"

1.4. Calcular el Símbolo de Jacobi $s \leftarrow (\frac{a}{n})$

1.5. Si $r \not\equiv s \, mod(n)$ devolver "Compuesto"

2. Devolver "Primo"

Observamos que el algoritmo realiza $O(tq)$ operaciones, donde q es el coste de calcular el símbolo de Jacobi. H. Cohen demuestra en (Cohen, 1993) que el símbolo de Jacobi se puede calcular en no más de $O((\log n)^2)$ con $m = max\{a, n\}$ con lo cual da lugar a un algoritmo de orden $O(k(\log n)^2) = O((\log n)^2)$. El mayor problema de este algoritmo radica en la dificultad para implementar el cálculo del símbolo de Jacobi. Sin embargo, años más tarde aparecería otra caracterización de los números primos que posibilitaría un test sustancialmente mejor que éste. La idea del test se le debe a Miller y la posterior mejora a Rabin.

2.6.3 Test de Miller-Rabin

El test más implantado en la actualidad es el Miller-Rabin (también conocido como *test fuerte del pseudoprimo*). El test de Miller-Rabin (MR) está basado en el siguiente hecho:

Si se tiene un número primo p y $p-1 = 2^s r$ donde r es impar se cumple que $\forall \, a \in \mathbb{N}$; $mcd(a,p) = 1$ entonces o bien $a^r \equiv 1 \, mod(n)$ o bien $\exists \, j \in [0, s-1]$; $a^{2^j r} \equiv -1 \, mod(p)$.

Algoritmo 2.6.3 Test de Miller-Rabin

Entrada: Un entero impar $n \geq 3 \; y \; un \; parametro \; de \; seguridad$ t

Salida: Una respuesta "Primo" o "Compuesto" a la pregunta ¿Es n primo?

1. Escribir $n-1=2^s r$ donde r es impar
2. Para i entero que recorre desde 1 hasta t hace:
   - **2.1.** Elegir un número entero aleatorio $2 \leq a \leq n-1$
   - **2.2.** Calcular $y = a^r \ mod(n)$
   - **2.3.** Si $r \neq 1$ y $r \neq n-1$ hacer lo siguiente:
     - **2.3.1.** $j \leftarrow 1$
     - **2.3.2.** Mientras $j \leq s-1$ y $y \neq n-1$ Calcular $y \leftarrow y^2 \ mod(n)$
       - **2.3.2.1.** Si $y = 1$ devolver "Compuesto"
       - 2.3.2.2. $j \leftarrow j+1$
     - 2.3.3. Si $y \neq n-1$ devolver "Compuesto"
3. Devolver "Primo"

La probabilidad de que n compuesto pase como primo es de $\frac{1}{4^t}$ para parámetros de seguridad grandes esta probabilidad es significativamente más pequeña que la del algoritmo 2.5.2, además la complejidad computacional de los mismos son iguales haciendo este test mucho más eficiente que el de Solovay-Strassen.

La pregunta ahora es, ¿no existen algoritmos que certifiquen primalidad? El problema con los algoritmos llamados Test de verdadera primalidad actuales es su gran complejidad computacional. Es un hecho que la fiabilidad de un algoritmo para la generación de números primos grandes usando un test de verdadera primalidad estaría garantizada, sin embargo tuvieron que pasar 30 años después de la creación del Test de Fermat para que se desarrollara un algoritmo determinista similar que fuera asintóticamente igual de rápido. Es por esta razón que se usan Test Probabilísticos en dichos algoritmos. (Borges, 2005)

Generación de Números Primos Grandes

Algoritmo 2.6.4 Generación de un Número Primo Usando el Test de

Miller-Rabin
Entrada: Un entero positivo k y un parámetro de seguridad t **Salida:** Un número aleatorio de tamaño $k - bit$ probable primo. 1. Generar un entero impar n de tamaño $k - bit$ 2. Usar Algoritmo de División sucesiva para determinar si n es divisible por algún primo $\leq B$. Si lo es ir al paso 1 3. Usar Algoritmo de Miller-Rabin (n, t) (Algoritmo 2.6.3). Si devuelve "Compuesto" ir al paso 1 4. Devolver n

Nota: Se puede observar que en el paso 2 se usa el algoritmo de división sucesiva para determinar si n es divisible por algún primo $\leq B$, pero ¿Qué es este B? O ¿Por qué es necesario este paso? Ese paso garantiza una gran reducción de la complejidad del algoritmo, pues realiza una comprobación del Test División Sucesiva sobre una tabla de números primos $p \in [3, B]$ almacenada previamente con el fin de reducir la cantidad de números a someter al Test de Rabin. La correcta elección de B es realizado empíricamente y estos valores están ligados al tamaño (bit) y al Teorema de Números Primos que habla sobre la distribución de los mismos (sección 1.2).

Números Primos Fuertes

Cuando se habla de generación de parámetros de clave púbica no solo se limita a la obtención de números primos grandes para la posterior generación de las claves públicas, además se pide que este número primo cumpla ciertas condiciones que garantizan una seguridad mayor. Por esta razón se introduce lo que en criptografía se llama número primo fuerte.

Definición 2.6.2 Número Primo Fuerte: Se dice que p es un primo fuerte si: Primero que p es un primo grande, además $p - 1$ tiene un factor primo grande,

que denotaremos r y $p+1$ tiene un factor primo grande, que denotaremos s además $r-1$ tiene también un factor primo grande, que denotaremos t.

Nota: La definición anterior puede verificarse en (Waarts, Braun, 1986), (Gordon, 1985), (Rivest, Silverman, 2007), sin embargo a efectos prácticos en estos artículos se muestran ejemplos de números primos fuertes tales como $11, 17, 29, 37, 41, 59, 67, 107, 197, 239, 251, 499, \ldots$ entonces cabe preguntarse ¿Qué tan importante es la condición de que tanto p como la de r, t, s sean números primos grandes? Se podría modificar la definición anterior estableciendo que tanto p como r, t, s sean de tamaño similar.

Algoritmo para generar números primos fuertes

2.6.5 Algoritmo de Gordon Para la Generación de Números Primos Fuertes
1. Generar dos números primos grandes aleatorios s y t de igual tamaño (bits) 2. Seleccionar un entero i_0. Encontrar el primer número primo en la secuencia $2it+1$ para $i = i_0, i_0+1, i_0+2, \ldots$. Denotar este primo $r \leftarrow 2it+1$ 3. Calcular $p_0 = 2\left(s^{r-2} \, mod(r)\right)s - 1$ 4. Seleccionar un entero j_0. Encontrar el primer primo en la secuencia $p_0 + 2jrs$ para $j = j_0, j_0+1, j_0+2, \ldots$. Denotar este primo $p \leftarrow p_0 + 2jrs$ 5. Devolver p

Justificación: Para ver que el número p es un Número Primo Fuerte obsérvese primero que $s^{r-1} \equiv 1 \, mod\,(r)$, esto es aplicando el Teorema de Fermat (véase 2.6.1); además $p_0 \equiv 1 \, mod(r)$ y $p_0 \equiv -1 \, mod(s)$ por la forma en que se obtiene. Finalmente se cumplen los requisitos de la definición 2.6.2:

(i) $\quad p - 1 = p_0 + 2jrs - 1 \equiv 0 \, mod(r)$

(ii) $p = 1 = p_0 + 2jrs + 1 \equiv 0\, mod(s)$

(iii) $r - 1 = 2it \equiv 0\, mod(t)$

Nótese que la fiabilidad de este algoritmo depende de varios factores: por ejemplo en el paso 1 se usa un algoritmo para generar los primos grandes s y t (Algoritmo 2.6.4), también en los pasos 2 y 4 se utiliza un determinado Test de Primalidad para comprobar que los enteros r y finalmente p son primos (Test de Miller-Rabin). Los Tests de Primalidad y Algoritmos de Generación de Primos usados comúnmente son algoritmos probabilísticos, esto da como resultado que el Algoritmo de Gordon devuelve un número primo fuerte probable.

2.7 Algoritmo Propuesto para generar Números Primos Fuertes

Las variaciones realizadas a este algoritmo parten de la idea del mejoramiento del mismo realizando cambios en el método usado para generar números primos y realizar el test de primalidad usando un algoritmo original que tiene la bondad de ser determinístico, lo que mejoraría la fiabilidad del uso de un algoritmo probabilístico.

Fundamento Teórico del Algoritmo Propuesto

Proposición 1: Sea la sucesión construida de la siguiente forma: $p_n = \begin{cases} 3n+1, & si\ n \in \mathbb{N}\ y\ es\ par \\ 3n+2, & si\ n \in \mathbb{N}\ y\ es\ impar \end{cases}$ y sea I el conjunto de los números impares mayores que 1 que no son divisibles por 3. Entonces $\{p_n\} = I$

Demostración:

En primer lugar se demuestra que $\{p_n\} \subset I$

Caso 1: Sea $n \in \mathbb{N}\ y\ par$ entonces $n = 2k;\ k \in \mathbb{N}\ \ p_n = 3n + 1 = 3(2k) + 1 = 2(3k + 1) - 1 \Longrightarrow p_n\ es\ impar$ luego como $p_n = 3n + 1\ \Longrightarrow p_n \equiv 1\ mod(3)$ lo que es equivalente a que p_n no es divisible por 3. De aquí que en este caso $p_n \epsilon\ I$

Caso 2: Sea $n \in \mathbb{N}\ e\ impar$ entonces $n = 2k - 1;\ k \in \mathbb{N}\ \ p_n = 3n + 2 = 3(2k - 1) + 2 = 2(3k) - 1 \Longrightarrow p_n\ es\ impar$ luego como $p_n = 3n + 2\ \Longrightarrow p_n \equiv 2\ mod(3)$ lo que es equivalente a que p_n no es divisible por 3. Luego en este caso también $p_n \epsilon\ I$

Hasta ahora se ha probado que $\{p_n\} \subset I$

Sea ahora $q \in I\ \Longrightarrow (q = 2k - 1;\ k \in \mathbb{N}\ ) \wedge (q \equiv 1\ mod[3]\ \vee q \equiv 2\ mod[3]\ )$

Caso 1: Sea $(q = 2k - 1;\ k \in \mathbb{N}\ ) \wedge (q \equiv 1\ mod[3]\ ) \Longrightarrow \exists\ m \in \mathbb{N}, tal\ que\ q = 3m + 1$. Se necesita probar que m es par. Si se supone lo contrario, m es impar, entonces $m = 2k - 1;\ k \in \mathbb{N} \Longrightarrow q = 3m + 1 = 3(2k - 1) + 1 = 6k - 2 = 2(3k - 1) \Longrightarrow q\ es\ par$ lo cual es un absurdo pues partimos de la hipótesis de que q es impar.

Caso 2: Sea $(q = 2k - 1;\ k \in \mathbb{N}\ ) \wedge (q \equiv 2\ mod[3]\ ) \Longrightarrow \exists\ m \in \mathbb{N}, tal\ que\ q = 3m + 2$. Se requiere probar que m es impar, Si se supone lo contrario m es par, entonces $m = 2k;\ k \in \mathbb{N} \Longrightarrow q = 3m + 2 = 3(2k) + 2 = 2(3k + 1) \Longrightarrow q\ es\ par$ lo cual es un absurdo pues partimos de la hipótesis de que q es impar.

Se tiene entonces que $I \subset \{p_n\}$ y a su vez $\{p_n\} \subset I_n$ por lo que queda demostrado $\{p_n\} = I$

Se puede observar que $p_n = \begin{cases} 3n + 1,\ \ si\ n \in \mathbb{N}\ y\ es\ par \\ 3n + 2,\ \ si\ n \in \mathbb{N}\ y\ es\ impar \end{cases}$ se puede escribir de forma compacta como $p_n = \frac{6n+3+(-1)^{n+1}}{2}$, de modo que se puede proponer el siguiente teorema. Además como la sucesión de los números primos mayores que 3 está contenida en la sucesión p_n es válido proponer el siguiente teorema:

Teorema 1: Para todo número primo p mayor que 3 existe un natural n, tal que p se puede escribir de forma única como $p = \frac{6n+3+(-1)^{n+1}}{2}$.

Demostración:

La demostración de la existencia está garantizada por la proposición 1 antes demostrada. La unicidad se obtiene de la siguiente consideración:

Si se supone que este natural no es único, o sea $\exists\, n, m$ naturales tal que $m \neq n$ y $p_n = p_m$

Caso 1: Si m, n son ambos pares $p_n = 3n + 1 = 3m + 1 = p_m \implies 3n = 3m \implies n = m$ lo cual es un absurdo pues partimos de la hipótesis de que $m \neq n$.

Caso 2: Si m, n son ambos impares $p_n = 3n + 2 = 3m + 2 = p_m \implies 3n = 3m \implies n = m$ lo cual es un absurdo pues partimos de la hipótesis de que $m \neq n$.

Caso 3: Si m, n son de distinta paridad $p_n = 3n + 1 = 3m + 2 = p_m \implies 3n = 3m + 1 \implies n = \frac{3m+1}{3} = m + \frac{1}{3}$ lo cual es un absurdo pues partimos de la hipótesis de que n, m son naturales.

Por tano necesariamente $m = n$

Se ha demostrado que todo número primo impar puede escribirse de esta forma, pero ¿cómo determinar si un número de esta forma es primo o compuesto?

Sea $p_m = \begin{cases} 3m + 1, & si\ m\ es\ par \\ 3m + 2, & si\ m\ es\ impar \end{cases}$, de manera que p_n no es primo, esto significa que existen p_{n_1} y p_{n_2} también enteros, impares y no múltiplos de 3, tales que $p_m = p_{n_1} * p_{n_2}$, siendo n_1 y n_2 naturales menores que m.

Se deberán analizar los siguientes casos:

1) Si m es par y ambos n_1 y n_2 de la misma paridad, es decir ambos pares o ambos impares.

a) Si m es par y ambos n_1 y n_2 pares entonces se tiene que $3m+1=(3n_1+1)(3n_2+1)=3(3n_1n_2+n_1+n_2)+1$, de donde se obtiene que $m=3n_1n_2+n_1+n_2$. Es decir n_1 y n_2 son solución de la ecuación diofántica $3xy+x+y=m$ (I).

Luego, si la ecuación $3xy+x+y=m$; m par, no tiene soluciones enteras positivas el número p_m, es primo probablemente.

b) Si m es par y ambos n_1 y n_2 impares entonces se tiene que $3m+1=(3n_1+2)(3n_2+2)=3(3n_1n_2+2n_1+2n_2+1)+1$, de donde se obtiene que $m=3n_1n_2+2n_1+2n_2+1$. Es decir n_1 y n_2 son solución de la ecuación diofántica $3xy+2x+2y+1=m$. (II)

Se ha obtenido entonces que si m es par y no existen soluciones enteras positivas de las ecuaciones diofántica (I) y (II) se puede afirmar que el número p_m es primo.

2) Si m es impar n_1 y n_2 de paridad diferente.

a) Si m es impar y n_1 y n_2 par e impar respectivamente entonces se tiene que $3m+2=(3n_1+1)(3n_2+2)=3(3n_1n_2+2n_1+n_2)+2$ de donde se obtiene que $m=3n_1n_2+2n_1+n_2$. . Es decir n_1 y n_2 son solución de la ecuación diofántica $3xy+2x+y=m$.(III)

b) Si m es impar y n_1 y n_2 impar y par respectivamente entonces se tiene que $3m+2=(3n_1+2)(3n_2+1)=3(3n_1n_2+n_1+2n_2)+2$ de donde se obtiene que $m=3n_1n_2+n_1+2n_2$. . Es decir n_1 y n_2 son solución de la ecuación diofántica $3xy+x+2y=m$.(IV)

Las ecuaciones (III) y (IV) son equivalentes, luego se tiene que si m es impar y no existen soluciones enteras positivas de la ecuación diofántica (III) se puede afirmar que el número p_m es primo.

En resumen, se ha obtenido que si p_m con m par es compuesto entonces una y solo una de las ecuaciones diofántica: $3xy+x+y=m$ y $3xy+2x+2y+1=m$ tiene soluciones enteras positivas. Además si p_m con m impar es compuesto entonces la ecuación diofántica: $3xy+2x+y=m$ tiene soluciones enteras positivas.

Nótese que este resultado fundamenta un algoritmo para determinar si un número entero positivo mayor que 3 es primo o no. A continuación se propone dicho algoritmo:

Test de Primalidad Propuesto

Algoritmo 2.7 Test de Primalidad Propuesto

Entrada: Un número natural $p > 3$

Salida: $False$ si n es compuesto y $True$ si n es primo

1. $flag \leftarrow True$ (Se asume que el número p es primo)
2. $m \leftarrow \left\lfloor \frac{p}{3} \right\rfloor$ (parte entera por defecto de $\frac{p}{3}$)
3. **Si** $m \equiv 0\ (mod\ 2)$**,** (Es un número par) **entonces:**
    1. **Si** existen soluciones positivas de solo una de la ecuaciones diofánticas $3xy + x + y = m$ **o** $3xy + 2x + 2y + 1 = m$ **entonces:** $flag \leftarrow False$

    Sí No

    1. **Si** existen soluciones positivas de la ecuación diofántica $3xy + 2x + y = m$

    entonces: $flag \leftarrow False$
4. **Retornar:** $flag$

Generación de Números Primos Grandes

En el epígrafe anterior se ha descrito un algoritmo para generar números primos grandes, mediante el uso del Test de Primalidad de Miller-Rabin (Algoritmo 2.6.4), además en este epígrafe se ha propuesto una novedosa forma de comprobar la primalidad de un número, dado que esta forma constituye un test de primalidad determinístico o test de verdadera primalidad (véase epígrafe 2.6), la primera conjetura del autor será:

Conjetura 1: Al modificar el Algoritmo 2.6.4 cambiando el test de primalidad usado (Miller-Rabin) por el test de primalidad propuesto, este algoritmo será más fiable y eficiente.

La modificación realizada al Algoritmo 2.6.4 consiste en cambiar el paso 3 de dicho algoritmo, donde se comprueba la primalidad de un número usando el Test de Primalidad de Miller-Rabin, por un paso similar donde se comprueba la primalidad de dicho número usando el test de primalidad propuesto. Esto trae como consecuencia que la entrada de dicho algoritmo, constituida por un entero positivo k y un parámetro de seguridad t, se modifica, ya que no es necesario el parámetro de seguridad t debido a que este parámetro es un requerimiento del Test de Primalidad de Miller-Rabin.

Algoritmo 2.7.1 Generación de un Número Primo Usando el Test de Primalidad Propuesto
Entrada: Un entero positivo k **Salida:** Un número aleatorio de tamaño $k - bit$ probable primo. 1. Generar un entero impar n de tamaño $k - bit$ 2. Usar Algoritmo de División sucesiva para determinar si n es divisible por algún primo $\leq B$. Si lo es ir al paso 1 3. Usar Algoritmo de 2.7 (Test de Primalidad Propuesto) Si devuelve "False" ir al paso 1 4. Devolver n

Se puede observar que se mantiene el paso 2, en el cual se usa el algoritmo de división sucesiva para determinar si n es divisible por algún primo $\leq B$, la necesidad de este paso se explica en el epígrafe anterior.

Generación de Números Primos Fuertes

En el epígrafe anterior se ha descrito un algoritmo para generar números primos fuertes, Algoritmo de Gordon (Algoritmo 2.6.5). Siguiendo la misma línea de ideas del epígrafe anterior se ha obtenido un novedoso test de primalidad, y a partir de este test se ha creado un algoritmo para generar números primos grandes. Esto nos brinda las herramientas necesarias para construir un nuevo algoritmo para la generación de números primos fuertes, las bases teóricas y las técnicas utilizadas en el Algoritmo de Gordon no serán modificadas, pues éstas están bien justificadas y han demostrado una gran eficacia. La modificación que será realizada consiste en cambiar los algoritmos de generación de un número primo grande y test de primalidad usados originalmente por los algoritmos anteriormente propuestos.

Conjetura 2: Al modificar el Algoritmo 2.6.5 (Algoritmo de Gordon para la Generación de Números Primos Fuertes), cambiando los algoritmos de generación de un número primo grande y test de primalidad usados originalmente por los algoritmos anteriormente propuestos, este algoritmo será más fiable y eficiente.

2.7.2 Algoritmo de Gordon Modificado Para la Generación de Números Primos Fuertes

1. Generar dos números primos grandes aleatorios (Algoritmo 2.7.1) s y t de igual tamaño (bit)
2. Seleccionar un entero i_0. Encontrar el primer número primo (Algoritmo 2.7) en la secuencia $2it + 1$ para $i = i_0, i_0 + 1, i_0 + 2, \ldots$ Denotar este primo $r \leftarrow 2it + 1$
3. Calcular $p_0 = 2\left(s^{r-2}\ mod(r)\right)s - 1$
4. Seleccionar un entero j_0. Encontrar el primer primo (Algoritmo 2.7) en la secuencia $p_0 + 2jrs$ para $j = j_0, j_0 + 1, j_0 + 2, \ldots$ Denotar este primo $p \leftarrow p_0 + 2jrs$
5. Devolver p

Como consecuencia del uso de un test de verdadera primalidad, tras modificar el Algoritmo de Gordon se ha obtenido un algoritmo determinístico que genera números primos fuertes.

2.8 Algunas consideraciones en cuanto a los algoritmos probabilísticos

Un algoritmo probabilista (o probabilístico) es un algoritmo que basa su resultado en la toma de algunas decisiones al azar, de tal forma que, en promedio, obtiene una buena solución al problema planteado para cualquier distribución de los datos de entrada. Es decir, al contrario de un algoritmo determinista, a partir de unos mismos datos se puede obtener distintas soluciones y, en algunos casos, soluciones erróneas. Se puede optar por la elección aleatoria si se tiene un problema cuya elección óptima es demasiado costosa frente a la decisión aleatoria. Por otro lado, en muchos casos, es preferible el uso de dichos algoritmos, pues la complejidad de los mismos suele ser mucho menor que un algoritmo determinista que resuelva el mismo problema, además repitiendo la ejecución un número suficiente de veces para el mismo dato, puede aumentar tanto como se quiera el grado de confianza de obtener una solución correcta, esto por supuesto al costo de un aumento de la complejidad.

Conclusiones del capítulo

En este capítulo se ha seguido una línea de ideas con el propósito de sentar las bases para entender la importancia que tiene la generación de parámetros de seguridad para la criptografía asimétrica y se ha descrito paso a paso cómo se construyen dichos parámetros. Primero, realizando un recorrido por las diferentes técnicas de Cifrado Asimétrico, explicando su funcionamiento y haciendo notar en cada caso en dónde radica la seguridad de dichas técnicas, es por esto que se han tratado dos problemas fundamentales de la teoría de números: el problema del logaritmo discreto y el problema de la factorización

de enteros, en estos problemas radica dicha seguridad, pero en ambos, para una mayor seguridad, es preciso hacer una correcta elección de los parámetros. Luego, tratando la cuestión anterior se ha explicado cómo generar números primos grandes, a partir del uso de un test de primalidad y cómo dentro de estos primos grandes encontrar primos fuertes. Finalmente se ha seguido la misma tendencia para presentar un algoritmo destinado a generar números primos fuertes, a partir de un test de primalidad propuesto por el autor. Cabe destacar que el algoritmo original propuesto por el autor tiene la bondad de que se usan algoritmos determinísticos para generar números primos y realizar el test de primalidad, lo que mejora la fiabilidad del uso de un algoritmo probabilístico. La relevancia de este trabajo viene dada al pretender ofrecer un algoritmo determinista, es decir, sin fallos en la respuesta con un costo computacional aceptable. En el próximo capítulo se realizará un análisis de la complejidad algorítmica del algoritmo propuesto con el fin de ganar criterios acerca de la eficiencia mencionada en las conjeturas realizadas.

Capítulo 3 Implementación del algoritmo y análisis de resultados

La primera consideración al plantear la implementación del algoritmo fue el encontrar un lenguaje de programación apropiado. Se consideró para esta tarea el lenguaje de programación Python 3.7.0.

3.1 Características generales del Python

Python es un lenguaje de programación poderoso y fácil de aprender. Cuenta con estructuras de datos eficientes y de alto nivel y un enfoque simple pero eficaz a la programación orientada a objetos. La elegante sintaxis de Python junto con su naturaleza propia, hacen de éste un lenguaje ideal para el desarrollo rápido de aplicaciones en diversas áreas y sobre la mayoría de las plataformas. El intérprete de Python y la extensa biblioteca estándar están a libre disposición en forma binaria y de código fuente para las principales plataformas desde el sitio web de Python, http://www.python.org/, y puede distribuirse libremente. El mismo sitio contiene también distribuciones y enlaces de muchos módulos libres de Python de terceros, programas y herramientas, y documentación adicional. El intérprete de Python puede extenderse fácilmente con nuevas funcionalidades y tipos de datos implementados en C o C++ (u otros lenguajes accesibles desde C). Por otra parte los programas desarrollados en Python no están compilados al del código de máquina, es por esto que dichos programas solo pueden ser computados en un ordenador que tenga el intérprete de Python instalado.

3.2 Implementación del Algoritmo en Python

El Python cuenta con una extensa biblioteca de algoritmos y funciones para el trabajo científico, desde trabajo con listas y arreglos que se pueden encontrar en los cursos más básicos de la programación en Python hasta métodos numéricos para resolver ecuaciones diferenciales y los más actualizados

algoritmos de inteligencia artificial para el reconocimiento de patrones y procesamiento de imágenes. Las bibliotecas usadas por el autor en la implementación han sido Numpy y Sympy, de la primera se usan los métodos de "$randint$" que tiene como entrada dos enteros a, b y devuelve un número entero "aleatorio" $c \in [a, b]$ y la función "$array$" que tiene como entrada una lista y convierte esta en un arreglo (esto último solo se realiza por conveniencia del autor, pues se puede trabajar sobre una lista de igual manera), de la segunda biblioteca se usa la función "$diophantine$" la cual tiene como entrada una ecuación diofántica y devuelve sus soluciones, en caso de no tener soluciones devuelve "$None$".

El primer paso para la implementación del algoritmo es el test de primalidad, siguiendo los fundamentos teóricos planteados en la sección 2.7, para la comprobación de la efectividad de este test, o sea que no de errores, el autor se ha auxiliado del software Wolfram Mathematica (Es posible acceder libremente a la nube Wolfram Alpha) que posee herramientas matemáticas muy fuertes, usando la función $RandomInteger$ para generar una lista de enteros "aleatorios", luego se ha usado $PrimeQ[]$ sobre esta lista para determinar cuáles son primos y cuáles no, se ha pasado el test propuesto por dicha lista y comparado los resultados.

```
In[25]:= L = Table[RandomInteger[{10^2, 10^4}], 100];
          tabla   entero aleatorio
         T = PrimeQ[L];
             ¿primo?
         R = Table[, 100];
             tabla
         For[i = 1, i ≤ 100, i++,
         para cada
           R[[i]] = {L[[i]], T[[i]]}
          ];
         Print[R];
         escribe
```

Figura 3.2.1 Código en el Wolfram

```
57 # Esta es la lista que devolvio el matematica
58 L=L=[9498, 3052, 3013, 9810, 776, 3015, 1854, 4595, 1682, 3890, 9928, \
59 6336, 5635, 2397, 4344, 187, 5896, 5540, 4260, 7923, 9759, 528, 6452, \
60 4590, 1174, 5940, 8804, 3078, 6960, 1981, 6368, 6977, 9343, 1195, \
61 455, 9930, 848, 2115, 3886, 2718, 7276, 6922, 3098, 1344, 5674, 5408, \
62 3176, 8351, 671, 9683, 5001, 3535, 107, 6484, 5335, 8373, 2115, 4451, \
63 8727, 2466, 4580, 3535, 2316, 9573, 493, 9923, 872, 6767, 3827, 7241, \
64 4875, 3455, 5069, 5322, 5788, 1284, 712, 5762, 7300, 6440, 5622, 360, \
65 1330, 2760, 6758, 4326, 2496, 4457, 7570, 4622, 2808, 9145, 5801, \
66 5154, 3704, 4304, 5265, 2486, 5207, 4876]
67
68 l=np.array(L)
69 for i in l:
70     t=morfa_test(i)
71     print(i,t)
72
```

Figura 3.2.2 Código en Python

En el ejemplo expuesto en la tabla siguiente, se muestran 50 números enteros aleatorios que han sido chequeados por los test de primalidad antes mencionados, este ha sido solo un intento de ilustrar la fiabilidad del test propuesto. Una prueba más rigurosa se realizó importando las listas de números y su primalidad ($True$ o $False$), dadas por el Python, en el Mathematica y comparando la igualdad de ambas listas. (Se han realizado corridas de 10^6 enteros en $[10^6, 10^7]$). Todas las pruebas han resultado positivas.

Tabla 3.2.1 Comparación de salidas

Salida de Python	Salida del Mathematica
9498 False	{9498, False},
3052 False	{3052, False},
3013 False	{3013, False},
9810 False	{9810, False},
776 False	{776, False},
3015 False	{3015, False},
1854 False	{1854, False},
4595 False	{4595, False},
1682 False	{1682, False},
3890 False	{3890, False},
9928 False	{9928, False},
6336 False	{6336, False},
5635 False	{5635, False},

2397 False	{2397, False},
4344 False	{4344, False},
187 False	{187, False},
5896 False	{5896, False},
5540 False	{5540, False},
4260 False	{4260, False},
7923 False	{7923, False},
9759 False	{9759, False},
528 False	{528, False},
6452 False	{6452, False},
4590 False	{4590, False},
1174 False	{1174, False},
5940 False	{5940, False},
8804 False	{8804, False},
3078 False	{3078, False},
6960 False	{6960, False},
1981 False	{1981, False},
6368 False	{6368, False},
6977 True	{6977, True},
9343 True	{9343, True},
1195 False	{1195, False},
455 False	{455, False},
9930 False	{9930, False},
848 False	{848, False},
2115 False	{2115, False},
3886 False	{3886, False},
2718 False	{2718, False},
7276 False	{7276, False},
6922 False	{6922, False},
3098 False	{3098, False},
1344 False	{1344, False},
5674 False	{5674, False},
5408 False	{5408, False},

3176 False	{3176, False},
8351 False	{8351, False},
671 False	{671, False},

Para la segunda parte de la implementación, es necesario generar dos números primos de n dígitos, s, t (Algoritmo 2.7.2), para esto se ha implementado la función $primegen$ la cual recibe una entrada n natural y devuelve un número primo aleatorio de n cifras. La función creada por el autor para este fin: $primegen,$ genera números enteros del tamaño adecuado y comprueba su primalidad hasta obtener un número primo de tamaño adecuado. A modo de ilustración del buen funcionamiento del algoritmo, en la tabla siguiente se muestran corridas de números primos generados aleatoriamente con $primege$.

Tabla 3.2.2 Generación de números primos pseudoaleatorios

(5, True)
(59, True)
(457, True)
(9533, True)
(49811, True)
(402587, True)
(2009407, True)
(59278103, True)
(647283019, True)
(9437210521, True)
(47846509819, True)
(399818489947, True)
(3959140205927, True)
(91204622016481, True)
(183557220222463, True)
(5505516356788793, True)
(53800596187819601, True)

(727378535647863551, True)
(4717880619268680971, True)
(54021105470049137083, True)
(423223474934482411481, True)
(1607324312427854403893, True)
(6819661175348097295671 7, True)
(738959479705071528668969, True)
(7322954996363864927545477, True)
(364198401520178396787735 61, True)
(2361415010207485852885642 21, True)
(8640765748107112136393486501, True)
(588767816014912265386791 52091, True)

La última parte del algoritmo consiste en generar un número primo fuerte a partir de los números primos s, t. Puesto que los procedimientos teóricos del algoritmo de Gordon no han sido modificados, solo se ha modificado el test de primalidad usado, es posible afirmar que los números primos que ofrece el algoritmo modificado son en efecto primos fuertes.

3.3 Fiabilidad y Eficiencia del Algoritmo.

Cuando se habla de fiabilidad y eficiencia de un algoritmo determinado, es necesario analizar varios factores. El término fiabilidad es descrito en el diccionario de la RAE como "probabilidad de buen funcionamiento de algo". Por tanto, extendiendo el significado a algoritmos, se dice que la fiabilidad de un algoritmo es la probabilidad de que este obtenga una misma respuesta para entradas idénticas, esto es, que al pasarle el mismo valor de entrada varias veces su respuesta sea idéntica. El término eficiencia algorítmica es usado para describir aquellas propiedades de los algoritmos que están relacionadas con la cantidad de recursos (tiempo y espacio) utilizados por el algoritmo.

Tras las comparaciones realizadas, descritas en la sección anterior, y las bien fundamentadas bases teóricas del algoritmo, descritas en la sección 2.7, es acertado decir que el algoritmo funciona correctamente, además por el hecho de ser un algoritmo determinista es sabido que a varias corridas de la misma entrada este devuelve una salida idéntica; es por esto que se puede afirmar que el algoritmo es fiable y que su fiabilidad, o sea la probabilidad de que este obtenga una misma respuesta para entradas idénticas es 1. Esto quiere decir que 100% de probabilidad de la respuesta obtenida es correcta. En cambio, la probabilidad de que el algoritmo de Gordon devuelva un falso número primo fuerte viene dada por la probabilidad de fallar del test de Miller-Rabin, dicha probabilidad es de $\frac{1}{4^t}$, siendo t el parámetro de seguridad del test (sección 2.6.3) para parámetros de seguridad grandes esta probabilidad es significativamente pequeña, lo que hace que el test de Miller-Rabin y por lo tanto el Algoritmo de Gordon sean también fiables, y su fiabilidad sea igual a $1 - \frac{1}{4^t}$, para ilustrar esto véase el siguiente ejemplo: haciendo $t = 10$ la probabilidad de que la respuesta obtenida sea correcta es de 99.9999046325684, teniendo en cuenta que el parámetro t usado para números de 32 y 64 $bits$ es $t \leq 128$ la probabilidad de que la respuesta sea correcta es de $\left(1 - \frac{1}{4^t}\right) * 100 \cong 100$, sin embargo, esta cifra sigue siendo una probabilidad.

Para realizar un análisis del orden de eficiencia del algoritmo propuesto, la notación usualmente empleada es la notación asintótica O (sección 1.4). Las notaciones asintóticas estudian el comportamiento del algoritmo cuando el tamaño de las entradas es lo suficientemente grande, sin tener en cuenta lo que ocurre para entradas pequeñas y obviando factores constantes. Como el orden del Algoritmo de Gordon modificado está acotado por el orden del test de primalidad propuesto, el análisis del mismo se reduce a hacer el análisis de este último.

Sean $g_1(n), g_2(n)$ y $g_3(n)$ las funciones del tiempo de ejecución de las ecuaciones diofánticas (I), (II) y (III) respectivamente (sección 2.7) en función del tamaño n de las entradas. Entonces se tiene el resultado siguiente:

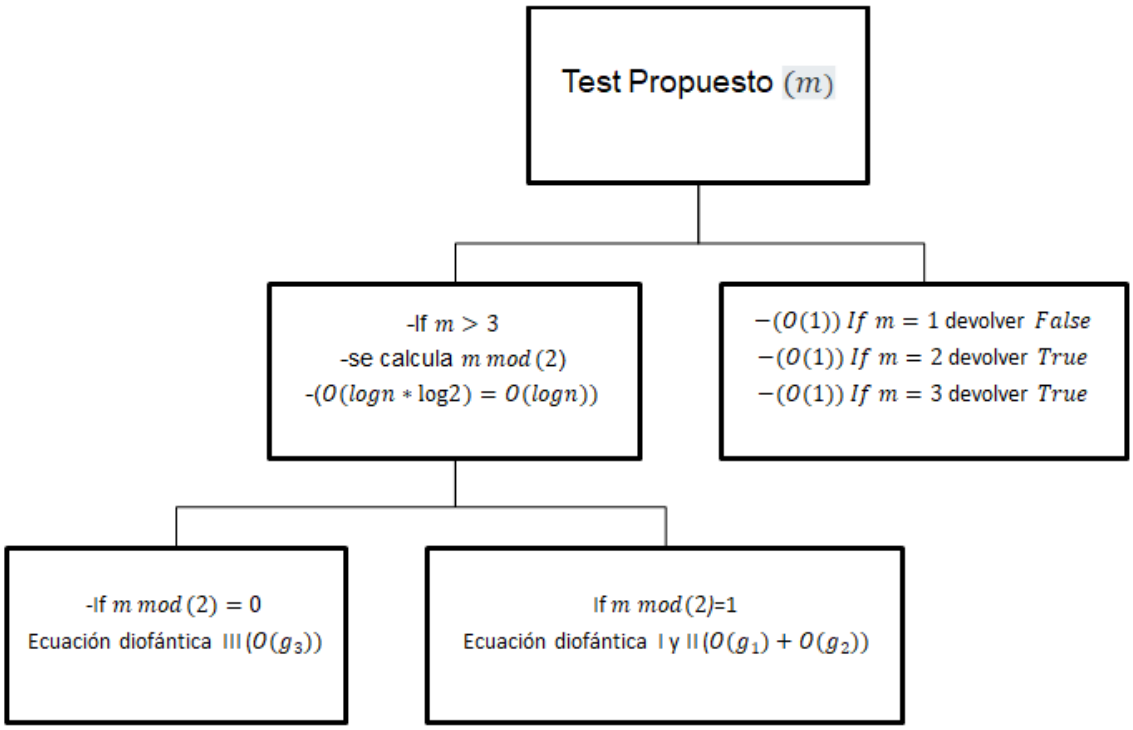


Figura 3.3.1 Esquema de dependencia de la eficiencia

Esto significa que en el peor de los casos el orden de complejidad del algoritmo es $O\big(g_1(n) + g_2(n)\big) + O(\log n) = O(g_1(n) + g_2(n) + \log n)$. Es posible acotar este orden de la siguiente manera: $Min\left(O\big(g_1(n)\big), O\big(g_2(n), O(\log n)\big)\right) \leq O(g_1(n) + g_2(n) + \log n) \leq Max\left(O\big(g_1(n)\big), O\big(g_2(n), O(\log n)\big)\right)$. Las funciones del tiempo de ejecución de las ecuaciones diofánticas, $g_1(n), g_2(n)$ y $g_3(n)$ no son conocidas a la fecha de la realización del presente trabajo por lo que la realización del análisis de eficiencia no será posible, este queda propuesto como futura investigación, sin embargo se realizarán algunas corridas para comparar la eficiencia del algoritmo propuesto con el algoritmo usado hoy en día. Para esta comparación se ha usado una implementación en Python del Test de Miller-Rabin, se ha tomado como parámetro de seguridad del mismo $t = 128$.

Para un mejor entendimiento de los ejemplos por analizar es importante conocer la relación que existe entre el tamaño de cierto número n, en $bits$ y el rango en el que se puede ubicar al número. Sea n un número entero positivo de tamaño k bits, entonces se encuentra en el rango $0 \leq n \leq 2^k - 1$

Tabla 3.3.1 Ejemplos de la relación entre tamaño y rango

Tamaño k en $bits$	Rango
$5\ bits$	$[0, 31]$
$10\ bits$	$[0, 123]$
$15\ bits$	$[0, 32767]$
$20\ bits$	$[0, 1048575]$
$25\ bits$	$[0, 33554431]$

En el primer ejemplo, se ha realizado una corrida, generando 100 números primos fuertes (tómese en consideración la nota que sigue a la Definición 2.6.2) en un rango de 5 a $10\ bits$. Se observa claramente que en este caso el algoritmo del autor es más eficiente que el algoritmo utilizado.

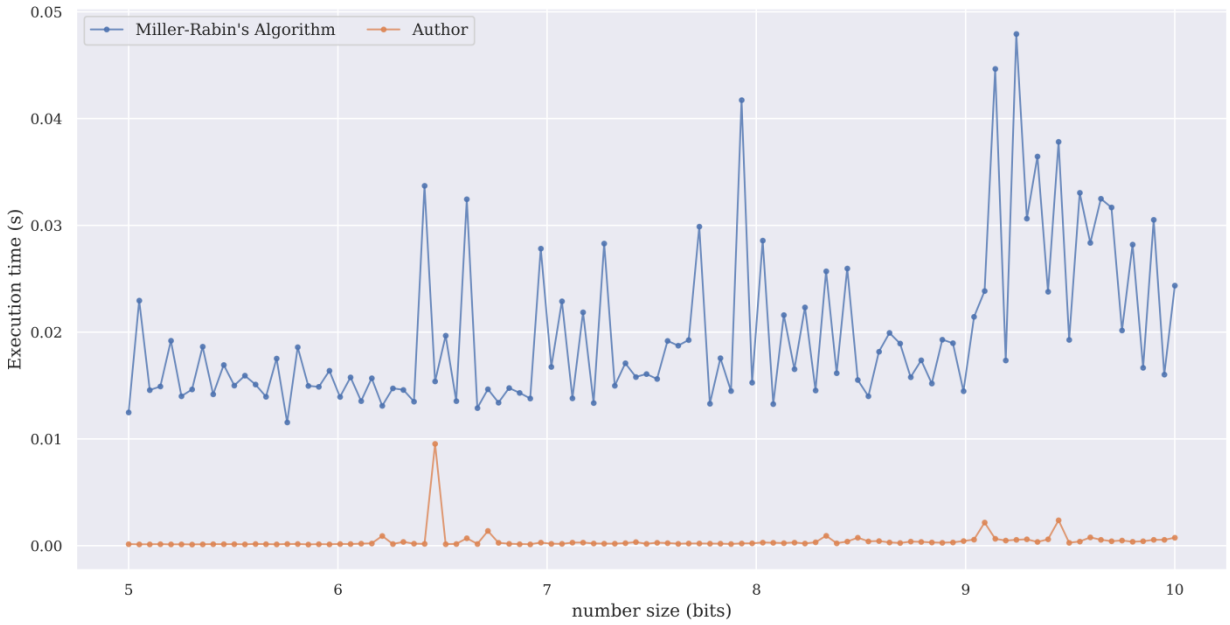


Figura 3.3.1 Comparación en el rango de 5 a 10 bits

Ahora se observará qué sucede para números en un rango mayor. Se generarán 100 números primos fuertes en un rango de 10 a $15\ bits$. Se observa en este caso que la eficiencia del algoritmo del autor (en adelante se denominará algoritmo original) es superior que la del algoritmo clásico.

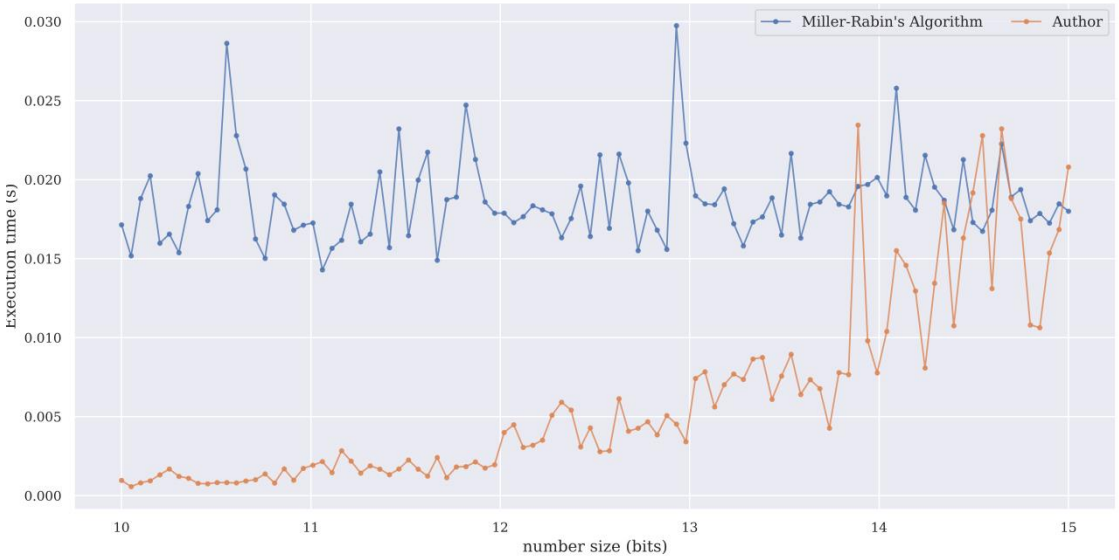


Figura 3.3.2 Comparación en el rango de 10 a 15 bits

Obsérvese que en la figura anterior, el algoritmo del autor tiene un comportamiento creciente mientras que el algoritmo clásico se mantiene estable, con el fin de analizar el comportamiento de ambos algoritmos se han realizado corridas para generar 100 números primos fuertes de 15 a $20\ bits$ y de 20 a $25\ bits$. En ambos casos se observa que el algoritmo de clásico es más eficiente que el algoritmo original, sin embargo es importante notar que el tiempo de corrida de la generación de números primos fuertes de un rango de 15 a $25\ bits$ no supera 0.14 segundos.

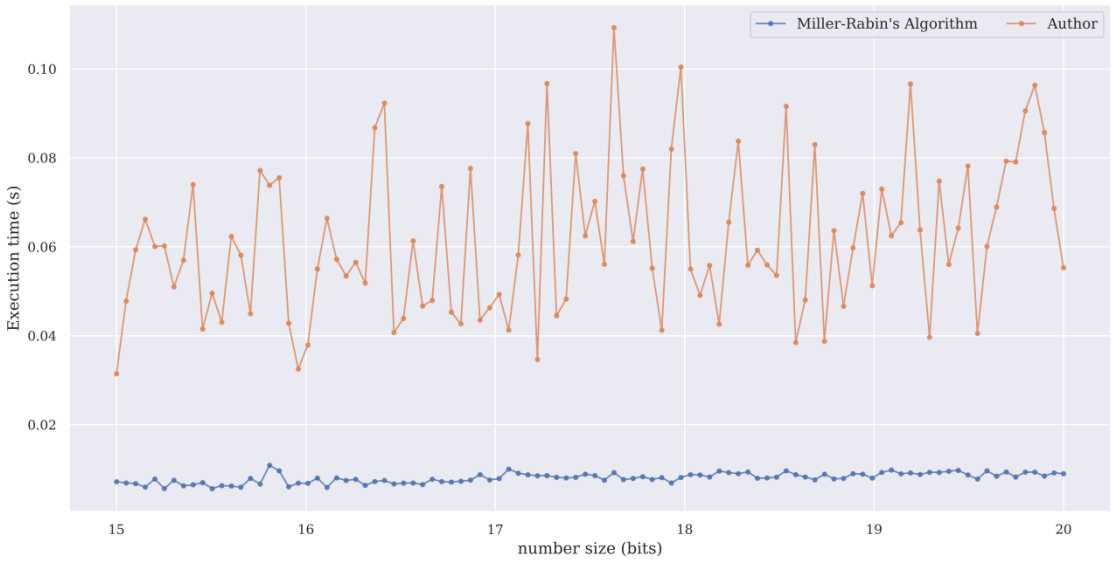
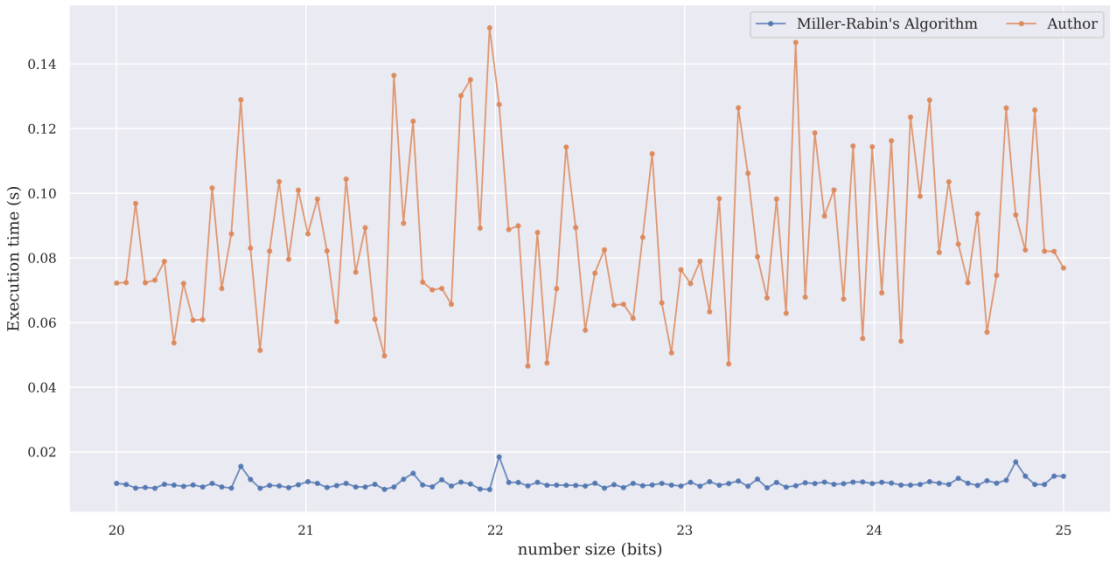


Figura 3.3.3 Comparación en el rango de 15 a 20 bits



Figurad 3.3.4 Comparación en el rango de 20 a 25 bits

El comportamiento para números de 32 a $64\ bits$, se muestra en la siguiente figura.

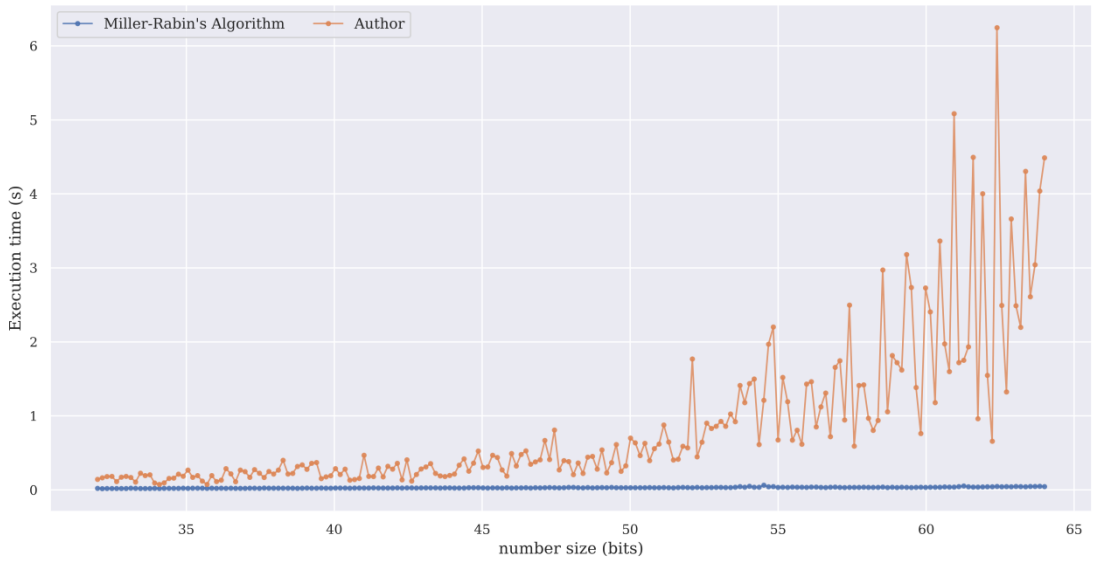


Figura 3.3.5 Comparación en el rango de 32 a 64 bits

Hasta ahora se ha realizado un análisis del comportamiento del tiempo de cómputo del algoritmo original, comparando este con el algoritmo clásico usado hoy en día. Como muestran las figuras mostradas con anterioridad el algoritmo original se comporta de una manera eficiente para entradas de tamaño de hasta $50\ bits$, incluso superando al algoritmo clásico hasta entradas de $15\ bits$, manteniendo un tiempo de computo menor al segundo, sin embargo para números mayores de $50\ bits$ el algoritmo original crece bastante más que el algoritmo clásico, el cual mantiene un comportamiento regular.

3.4 Interpretación de los resultados obtenidos

Tras realizar un análisis de la fiabilidad y eficiencia del algoritmo propuesto se observa que este es fiable, sin embargo se ha notado que su eficiencia depende muy estrechamente de los métodos de solución de las ecuaciones diofánticas. Según el análisis de complejidad realizado en la sección anterior, el orden de complejidad del algoritmo está acotado por el tiempo que toma la resolución de las ecuaciones diofánticas, esto quiere decir que la eficiencia del

algoritmo propuesto depende de la eficiencia del algoritmo que resuelve las ecuaciones diofánticas. Antes se mencionaba que dicha eficiencia es desconocida por el autor, esto se debe a que para la implementación del algoritmo se ha usado una función de $numpy$, biblioteca de python (sección 3.2), y es difícil determinar el método exacto que esta función utiliza para resolver dichas ecuaciones. La función usada $diophantine$ resuelve ecuaciones diofánticas generales, siempre que éstas tengan solución, sin embargo si se logra implementar un método más específico para la resolución de las ecuaciones diofánticas cuadráticas reducibles a Pell, como las usadas en el presente trabajo, la complejidad algorítmica del test de primalidad puede reducirse y por tanto la del algoritmo original.

Conclusiones del capitulo

En este capítulo se ha realizado una implementación del algoritmo de Gordon Modificado propuesto en el capítulo anterior; para esto se ha utilizado una implementación del Test de primalidad propuesto por el autor. Ambas implementaciones se han realizado usando el lenguaje de programación Python por las bondades que este presenta. Con el fin de ganar criterios acerca de la eficiencia mencionada en las conjeturas planteadas sobre los algoritmos antes mencionados, se ha realizado un análisis de la eficiencia y la fiabilidad de los mismos, dicho análisis ha mostrado que el algoritmo propuesto es más fiable que el algoritmo clásico. Los resultados sobre la eficiencia se han obtenido para diversos rangos, se han graficado para una mayor visibilidad y se han interpretado para detectar las causas de comportamientos no esperados en ciertos rangos.

Conclusiones

En el presente trabajo se ha obtenido un importante resultado teórico, una novedosa manera de conocer si un número dado es primo o no, a partir de encontrar o no las soluciones enteras positivas de ciertas ecuaciones diofánticas. De esto se derivan dos resultados prácticos importantes, primero, la implementación de un test de primalidad determinista basado en la teoría antes mencionada; y la implementación de una algoritmo para generar números primos fuertes fundamentado en las bases teóricas desarrollada por John Gordon en 1985 y utilizando el test de primalidad antes mencionado. Estos resultados son de gran utilidad para la criptografía moderna, puesto que los números primos y en especial números primos fuertes son una precondición necesaria para la elaboración de las claves públicas, y algunas de sus propiedades y características, junto a otros aspectos del algebra modular y la teoría de números, son determinantes en su seguridad.

Los resultados principales han sido, en el Capítulo 2, se ha presentado una línea de ideas que nos ayudan en el entendimiento de la construcción del Algoritmo de Gordon, partiendo de un test de primalidad para elaborar un algoritmo que genere números primos y posteriormente un algoritmo que devuelve un número primo fuerte. El autor se ha seguido la misma tendencia para presentar un algoritmo destinado a generar números primos fuertes, a partir de un test de primalidad propuesto. El algoritmo original propuesto por el autor tiene la bondad de que se usan algoritmos determinístico para generar números primos y realizar el test de primalidad, lo que mejora la fiabilidad del uso de un algoritmo probabilístico. En el Capítulo 3 se implementa el algoritmo propuesto usando el lenguaje de programación Python, se ha realizado un análisis de la eficiencia y la fiabilidad de los mismos. Dicho análisis ha mostrado que el algoritmo propuesto es más fiable que el algoritmo clásico, aunque este es también fiable. Los resultados sobre la eficiencia se han obtenido para diversos rangos, se han graficado para una mayor visibilidad y se han interpretado para detectar las causas de comportamientos no esperados en ciertos rangos. Este trabajo además de los resultados teóricos y prácticos obtenido resulta de una buena utilidad metodológica, puesto que se ha

realizado un estudio bibliográfico extenso y se ha resumido los resultados más importantes, tanto de los fundamentos matemáticos de la criptografía moderna, como de sus principales técnicas.

Recomendaciones

Puesto que en el análisis de eficiencia realizado no muestra los resultados deseados y tras la interpretación de las gráficas de la sección 3.3. Se propone como trabajo futuro el análisis de las ecuaciones diofánticas y su método de solución, enfocado solamente a los casos tratados y la posterior implementación de un método para resolver dichas ecuaciones en particular.

Bibliografía

A. Menezes, P. v. O. a. S. V. (1997). Handbook of Applied Cryptography, CRC Press.

Acosta, E. S. (2012). Criptoanálisis más utilizados en la actualidad. Criptografía y teoría de códigos.

Angel, J. d. J. A. (2008). Criptografía para principiantes.

Ayuso, A. (2009). Una simulación simplificada de la criptografía.

Bahit, E. (2012). Curso: Python para principiantes, SafeCreative.

Borges Hernández Cruz Enrique. (2005). Test de primalidad.

Benjamin Baumslag, B. C. (2009). Theory and problems of Group Theory, Schaum's Outline Series.

Cohen. H. (1993) A course in Computational Algebraic Number

García, R. d. M. (2008). Criptografía Clásica y Moderna, Septem Ediciones.

Garret, P. (1998). Intro Abstract Algebra.

Gordon. J. (1985). Strong Primes are Easy to Find

Hector Corrales Sánchez, C. C. R., Alejandro Cuevas Notario (2006). "Criptografía y Métodos de Cifrado."

I. N. Bronshtein, K. A. S., G.Musiol, H.Muehling (2005). Handbook of Mathematics, Spinger.

López, M. J. L. (2001). Criptografía y seguridad en computadoras.

Mao, W. (2003). Modern Cryptography: Theory and Practice, Prentice Hall PTR.

Martín, C. A. R. S. (2016). Introducción a la criptografía. Departamento de Ciencias de la Educación, Universidad del Bio-Bio. Profesor de escuela media

Nápoli, P. d. (2014). Una introducción Matemática la Criptografía.

Olivos, V. C. (2005). Manual de Criptografía. S. d. Perú, Dirección Nacional de Programa de Jóvenes.

Pardo, M. L. (2016). Complejidad Computacional. Universidad de Cantabria.

Pardo, T. J. R. (2008) Introducción a Python.

Rodó, D. M. (2017). El lenguaje Python, Universitat Oberta de de Catalunya.

Rossum, G. v. (2009). El tutorial de Python.

Sánchez, J. M. (2011). "Historias de matemáticas. Riemann y los números primos." Pensamiento Matemático.

Schneider, B. (1996). Applied Cryptography. Protocols, Algorithms, and Source, John Wiley & Sons, Inc.

Sergio Paque Martín, D. A. (2009). Programación Declarativa Avanzada.

Titu Andreescu, D. A., Ion Cucurezeanu (2010). An introduction to Diophantine Equations, Birkhäuser. Springer.

Vera Delgado, R. P. (2006) Introducción a la criptografía: tipos de algoritmos.

Yeste, N. V. (2011). Propiedad de los números primos de Fermat.