Congruencia cuadrática y su impacto matemático e informático

(Mayo de 2021)

Dylan Samuel Cantillo Arrieta, dilanc@uninorte.edu.co
CIU#: 200085471, Cel: 3215043512

Resumen

Se realiza una investigación sobre la resolución de los problemas de la forma $\boldsymbol{x^2 \equiv a\ (mod\ p)}$ más conocido como congruencia cuadrática, para valores de $\boldsymbol{p} \in \mathbb{N}$, no solo para $\boldsymbol{p}$ primos, y sobre su impacto en la sociedad, usos e importancia tanto en las matemáticas como en la informática. Basándose en el símbolo de Legendre y el criterio de Euler, tomando como base los apuntes de Fermat sobre los números primos. Además, se presenta una comparación y prueba de hipótesis de la comparación con un algoritmo iterativo sobre el tiempo de demora de cada algoritmo. También, se calculan las funciones de tiempo y complejidad con respecto a algoritmos concretos utilizados para comprobar la existencia del x que cumple las condiciones. Mostrando la importancia y estrecha relación con los problemas NP-Hard.

Índice de Términos: algoritmo, aritmética modular, congruencia cuadrática, java, reciprocidad cuadrática, residuo cuadrático.

Abstract

Research is carried out on the resolution of the problems of the form x^2≡a (mod p) better known as quadratic congruence, for values of p∈N, not only for p cousins, and on their impact on society, uses and importance in both mathematics and computer science. Based on the Legendre symbol and Euler's criterion, based on Fermat's notes on the prime numbers. In addition, a comparison and hypothesis testing of the comparison with an iterative algorithm for the delay time of each algorithm is presented. Time and complexity functions are also calculated against specific algorithms used to check the existence of the x that meets the conditions. Showing the importance and close relationship with the NP-Hard problems.

Term Index: algorithm, java, modular arithmetic, quadratic congruence, quadratic reciprocity, quadratic residue.

I. Antecedentes

La incomprensión indirecta de las congruencias cuadráticas y números primos comenzó desde 1640 con Pierre de Fermat [1] con la conjetura de que todos los números de la forma

$$F_m = 2^{2^m} + 1, \qquad m = 0, 1, 2, \ldots$$

eran primos, sin embargo, luego se comprobó que esto era falso. Después, de varios años, siendo los números de Fermat un gran problema matemático, Thomas Clausen se comunica con Gauss (1855) [2] de que el F_6 era producto de dos números primo, lo que le dio paso a François Édouard Anatole Lucas en 1878 a formular el siguiente teorema, si un primo p divide a F_m para $m > 1$, entonces existe $k \in \mathbb{N}$ tal que

$$p = k2^{m+2} + 1$$

planteando años después la operación en forma de congruencia

$$3^{(F_m-1)/2} \equiv -1\ (mod\ F_m)$$

practicando estos términos principalmente en la geometría de polígonos. Mientras transcurrieron los años se planteó el teorema Euler-Fermat [3], donde si la función de Euler $\phi(n) \in \mathbb{N}$ más pequeño tal que se cumple

$$a^{\phi(n)} \equiv 1\ (mod\ n)$$

entonces, a es la raíz primaria modulo n, por consecuente si $a, n \in \mathbb{N}$, con $n > 1$ y $mdc(a, n) = 1$

$$x^2 \equiv a\ (mod\ n)$$

Las congruencias cuadráticas han sido y son un gran punto de inicio para la búsqueda de los factores primos y principalmente de los números primos, que hoy en día sigue siendo un gran problema por resolver de las matemáticas y computación.

El propósito de este artículo es informar, explicar y formular mejores formar para la resolución de las congruencias cuadráticas, calculando su tiempo de corrida computacionalmente, destacando la importancia de la relación de las congruencias cuadráticas y los números primos. Se buscará la construcción de un algoritmo para calcular la existencia de x que cumpla la congruencia cuadrática, calculando su tiempo de corrida y comparándolo con los algoritmos de Tonelli-Shanks y Cipolla-Lehmer que solucionan la incógnita de congruencia cuadrática con un n primo.

Cabe aclarar que este artículo se centra no solo para un n primo, sino para cualquier $n \in \mathbb{N}$, donde se llevaran los n no primos a primos por sus factores primos. Este problema pertenece a los NP-Hard [4] por la razón que puede ser resuelto mediante un algoritmo no determinístico, con un costo en tiempo polinomial [5]. Debido a que, en cierto punto, el algoritmo tiene diferentes caminos a seguir, y es capaz de escoger el mejor camino para solucionar el problema. También, teniendo en cuenta que está directamente relacionado con los números primos donde se consideran undecidables [6] porque no hay un algoritmo que siempre resuelva este problema, sin tener en cuenta de cuánto tiempo y espacio se dispone.

II. Estado del arte

En la historia se ha tratado de dar respuesta desde temprano a este tiempo de problemas, como se comentó anteriormente, pero verdaderamente Fermat se interesó en la resolución de estos problemas por la obra de 13 libros de Arithmetica de Diophantus of Alexandria (los libros I a III en griego, los libros IV a VII en árabe [7] y, presumiblemente, los libros VIII a X en griego) esto cerca del siglo XV. Años después fueron desarrollándose posibles soluciones y algoritmos para la resolución de no solo el residuo cuadrático sino también para la aritmética modular.

La ley de reciprocidad cuadrática planteada por Leonhard Euler en 1783 [8] y descubierta por Carl Friedrich Gauss en 1795, junto con el criterio de Euler con una complejidad de $O((\log p)^3)$[9], son la base de los algoritmos computacionales de las congruencias cuadráticas.

En 1891 se desarrolló el Tonelli's algorithm creado por Alberto Tonelli con complejidad de $O((log\ p)^4)$[10], en el siglo siguiente en 1907 se desarrolló el Cipolla's algorithm creado por Michele Cipolla con complejidad de $O((log\ p)^3)$[10], luego en 1917 se desarrolló el Pocklington's algorithm creado por Henry Cabourn Pocklington con complejidad de $O(log^3(N)\ log\ log\ N\ log\ log\ log\ N)$ [11] y finalmente en 1970 se creó el algoritmo Berlekamp–Rabin algorithm por Elwyn Berlekamp con complejidad de $O\left(n^2\ log(p)\right)$[12].

Al pasar de los años, se han desarrollado diferentes algoritmos para diferentes funciones y circunstancias, a pesar de la demora, unos más que otros, es por ciertas razones que requieres. Contando con una gran importancia, principalmente en la criptografía [13], al ser un problema muy difícil por la factorización de los números primos, también en las pruebas de primalidad [14] y la factorización de enteros [15], por la ley de reciprocidad. Logrando notar que la mayoría de sus usos e importancia radica en su relación con los números primos y la factorización de estos.

III. Conceptos matemática

Introduciendo los conceptos básicos, lo primero es mencionar que hasta el día de hoy no existe alguna fórmula para hallar los números primos, así que por esa razón no se mencionara al respecto.

A. Congruencia

Sea $m \in \mathbb{N}$ y $a, b \in \mathbb{Z}$. Se dice que a y b son congruentes modulo m

$$a \equiv b\ (mod\ m)$$

sí $m|(a-b)$, es decir si $a - b = km$, para $k \in \mathbb{Z}$.

B. Congruencia con ecuación lineal

Las ecuaciones de primer grado están dadas por

$$ax + b, \qquad a \neq 0$$

Entonces, la ecuación de primer grado general para las congruencias es

$$ax \equiv b\ (mod\ m)$$

en el caso de $a = 1$, siempre existe solución. El m más utilizado es $m = 10^n, con\ n \in \mathbb{N}$, que es para encontrar el número de dígitos de un x.

C. *Criterio de Euler*

Sea p > 2 un número primo y a un número entero coprimo con p. Entonces a es un residuo cuadrático módulo p si y solo si

$$a^{(p-1)/2} \equiv 1\ (mod\ p)$$

Como corolario de este teorema se obtiene que si a no es un residuo cuadrático módulo p entonces

$$a^{(p-1)/2} \equiv -1\ (mod\ p)$$

D. *Simbología de Legendre*

Dado un entero a y un primo impar p, el símbolo de Legendre, denotado $\left(\frac{a}{p}\right)$, de define

$$\left(\frac{a}{p}\right) = \begin{cases} 0, & si\ p\ divide\ a \\ 1, & si\ a\ es\ residuo\ cuadratico\ de\ p \\ -1, & si\ a\ no\ es\ residuo\ cuadratico\ de\ p \end{cases}$$

IV. Planteamiento

Partiendo de las ecuaciones lineales de primer grado donde sea $a, b, m \in \mathbb{N}$, ¿Existirá un $x < m$, tal que $ax \equiv b\ (mod\ m)$?, ahora bien, si aumentamos el grado de la ecuación, obtendríamos una cuadrática de grado dos donde sea $a, b, c, m \in \mathbb{N}$, entonces ¿Existirá un $x < m$, tal que $ax^2 + bx - c \equiv 0\ (mod$ m)? Ahora, considerando el caso donde $b = 0\ y\ a = 1$, tenemos $x^2 \equiv c\ (mod$ m), donde $x < m$.

Teniendo en cuenta lo anterior, el planteamiento de nuestra congruencia cuadrática sería el siguiente: Sea $a, m \in \mathbb{N}$ ¿Existe un $x \in \mathbb{N}$, tal que $x < m$, donde $x^2 \equiv a\ (mod$ m)?

V. Primer algoritmo

Se planteo un primer algoritmo iterativo para grandes valores de entrada donde se va probando uno por uno los posibles valores de $x < m$, comprobando su existencia, donde

$$x = \sqrt{mk + a}, \qquad k \in \mathbb{N}$$

$$x^2 \equiv a\ (mod\ m)$$

de esta manera, comprobando cada uno de los puntos posibles.

Algoritmo Congruencia Cuadrática 1

```
Enteros: a, m
Cadena: respuesta
x ← 1
potencia ← x^2
Mientras_que (x < m y potencia % m ≠ a) haga
    x ← x + 1
    potencia ← x^2
Fin_Mientras_que
Entregue (potencia % m = a)
```

El algoritmo realiza iteraciones hasta que $x = m\ o$ hasta encontrar un x que cumpla la condición de residuo, devolviendo verdadero si existe $x < m$ que $x^2 \equiv a\ (mod\ m)$, sin embargo, si no hay posibles respuesta devolverá falso.

Se procede a hallar su respectiva función del tiempo donde podemos ver claramente que únicamente depende de la variable max por lo tanto es univariada. Consideraremos $n = m$.

Teniendo en cuenta que trabajamos con un ciclo "Mientras_que", sea $T_p^q(n)$ el tiempo promedio de demora del algoritmo en función al valor límite de x y una probabilidad q.

Definimos dos instancias de (I) de comportamiento

I_i = caso en el que potencia % $m = a$, esta en la i-esima posición.

I_m = caso en que la potencia % $m \neq a$.

El tiempo de demora de las instancias es

$t(I_i) =$ x

$t(I_m) =$ n

Sea q la probabilidad que potencia % $m = a$ exista

$P(I_i) =$ q/n

$P(I_m) = 1 -$ q

Luego,

$$T_p^q(n) = P(I_i) \cdot t(I_i) + P(I_n) \cdot t(I_n)$$

$$T_p^q(n) = \sum_{x=1}^{x=n-1} i \cdot \left(\frac{q}{n}\right) + (1 - q) \cdot n$$

$$T_p^q(n) = \left(\frac{q}{n}\right) \cdot \left(\frac{\mathrm{n}\,(n-1)}{2}\right) + (1-q) \cdot n$$

$$T_p^q(n) = q \cdot \left(\frac{n-1}{2}\right) + (1-q) \cdot n$$

Por consecuente, si potencia $\% m = a$ (x existe), entonces $q = 1$

$$T_p^q(n) = \frac{n-1}{2}$$

De igual manera, si potencia $\% m \neq a$ (x no existe), entonces $q = 0$

$$T_p^q(n) = n$$

Así podemos concluir que el Algoritmo Congruencia Cuadrática 1 cuenta con una complejidad polinomial univariada de $O(n)$.

VI. Algoritmo de la Resolución Mejorado

Realizando una gran recopilación de partes importantes de otros algoritmos y apuntes de diferentes sitios web [16] como el teorema de Euler [17] y el símbolo de Legendre [18] y conocimiento previos, se presenta el algoritmos de congruencia cuadrática, planteándolo de la siguiente manera:

Primero se verifica si $p|a$ es decir si exististe un factor común diferente de uno entre p, a. En el caso que $\gcd(a,p) = 1$, se mira si p es par o impar, si es par es porque $p = 2^k$, si p es impar es porque es un numero primo o $p = q^k$ donde q es primo y $k \in \mathbb{N}$. Luego, si p es de la forma $p = q^k$ se aplica la simbología de Legendre, donde si $\left(\frac{a}{p}\right) = 1$ existe una respuesta, por el otro lado, si $p = 2^k$, se verifica su exponente, si $k = 1$ solo existe una respuesta donde x = 1, si $k = 2$ y $a \equiv 1\ (mod\ 4)$, existen dos respuestas $x = \pm 1$ y en el caso de $k \geq 3$ y además $a \equiv 1\ (mod\ 8)$, existen dos posibles respuestas.

Viendo el lado donde $\gcd(a,p) \neq 1$. Sea $\gcd(a,p^n) = p^r$, el caso donde $r \equiv 1\ (mod\ 2)$ no existe solución, pero si $r \equiv 0\ (mod\ 2)$, verificamos si $p = q^k$ con q impar entonces existen dos soluciones, si $p = 2^k$, entonces si $n - r = 2$ entonces existe una solucion, si $n - r = 4$ entonces existen dos respuestas y si $n - r \geq 8$ existen dos posibles respuestas.

Algoritmo Congruencia Cuadrática Mejorado

```
Entero a, p
Boleano respuesta
a ← a % p
respuesta ← falso
Si (a = 0 or a = 1) entonces
    respuesta ← verdadero
Si_no
    gcd = Llamar gcd (a, p)
    Si (gcd = 1) entonces
        Si (p % 2 = 1) entonces
            Si (Llamar legendre(a, Llamar encontrarBase (p)) = 1) entonces
                respuesta ← verdadero
            Fin_Si
        Si_no
            Si (a = 1 and p = 2) entonces
                respuesta ← verdadero
            Si_no
                Si (a = 1 and p = 2) entonces
                    respuesta ← verdadero
                Si_no
                    Si (a % 8 = 1) entonces
                        respuesta ← verdadero
                    Fin_si
                Fin_si
            Fin_si
    Si_no
        r = Llamar encontrarPotencia(gcd)
        n = Llamar encontrarPotencia(p)
        Si (r % 2 = 1) entonces
            respuesta ← verdadero
        Si_no
            Si (p % 2 = 1) entonces
                respuesta ← verdadero
            Si_no
                Si (n – r = 2) entonces
                    respuesta ← verdadero
                Si_no
                    Si (n – r = 4) entonces
                        respuesta ← verdadero
                    Si_no
                        respuesta ← verdadero
                    Fin_Si
                Fin_si
            Fin_si
        Fin_si
    Fin_si
```

```
Fin_si
Entregue (respuesta)
```

El algoritmo busca dentro de un rango reducido, gracias a los dígitos del numero, el exponente i de la base encontrada, que sea igual a el numero.

Algoritmo gcd (Máximo Común Divisor)

```
Entero a, p
Si (p = 0) entonces
    Entregue (a)
Si_no
    Entregue (Llamar (gcd (p, a%p)))
Fin_si
```

Algoritmo recursivo para encontrar el máximo común divisor de dos números.

Algoritmo Encontrar Base

```
Entero numero
i ← 2
Mientras_que (numero % i <> 0) haga
    i ← i + 1
Fin_mientras_que
Entregue (i)
```

Búsqueda del primer numero tal que numero % i = 0, es decir, que lo divida.

Algoritmo Legendre

```
Entero a, p
contador ← 0
Para (i = 1 hasta (p-1)/2 con incremento de 1) haga
    Si (a * i % p < (p-1)/2) entonces
        contador ← contador + 1
    Fin_si
Fin_para
Entregue ((−1)^contador)
```

El algoritmo Legendre Symbol está basado en el criterio de Euler y el algoritmo de Jacobi Symbol. Sin embargo, se optó por hacer una propia versión de este, que se sintió más pertinente para la resolución.

Como se puede observar el algoritmo está conformado por varias condiciones, donde consumiría un mayor tiempo en el caso donde $p \nmid a$ y si p es primo, debido a que debe de encontrar primeramente la base del número p y luego establecer el Legendre de (a, p), por consecuente las acciones de mejora sobre este algoritmo se encuentran en este lugar, donde se podría mejor más exactamente el algoritmo Legendre o Encontrar Base, pero más adelante veremos que el método Encontrar Base es el más rápido del sistema.

Algoritmo Encontrar Potencia

```
Entero numero, base, numDigitos
numDigitos ← lenght (numero)
base ← encontrarBase (numero)
valorInferior ← (numDigitos – 1) / log10 base
valorSuperior ← numDigitos / log10 base
Para (i = valorInferior hasta valorSuperior con incremento de 1) haga
    Si (base^i = numero) entonces
        Entregue (i)
    Fin_si
Fin_para
Entregue (0)
```

Luego, si pensamos de una manera más general, debido a que el algoritmo Congruencia Cuadrática Mejorado solo logra hallar resultados para algún p primo, nosotros antes de esto debemos buscar los factores primos del número p ingresado, entonces la mejor opción es hacer una mejora en el algoritmo de búsqueda de los factores primos.

Algoritmo Factores Primos

```
Entero numero
Cadena respuesta
Para (i = 2 hasta numero con incremento de 1) haga
    cont ← 0
    Mientras_que (numero % i = 0) haga
        numero ← numero / i
        cont = cont + 1
        Si (numero = 1) entonces
            Salir
        Fin_si
    Fin_mientras_que
```

respuesta ← respuesta + i^{cont}
Fin_para
Entregue (respuesta)

Se decidió trabajar el algoritmo con una cadena que luego se puedan obtener los factores con un "Split", ya que inmediatamente se mejora el rendimiento de memoria al no hacer uso de algún tipo de lista o vector.

VII. Cálculo del tiempo

Primeramente, calculando los algoritmos uno por uno, podemos concluir una complejidad de $O(\log p)$ para el Algoritmo gcd (Máximo Común Divisor), gracias al libro Origins of the Analysis of the Euclidean Algorithm [19], luego el Algoritmo Legendre, o también conocido como Algoritmo de Jacobi [20], cuenta con una complejidad de $O\big((\log a)(\log p)\big)$, sin embargo esta no aplica al sistema, debido a las nuevas modificaciones realizadas y la factorización en números primos cuenta con una tiempo de complejidad de $O(\log n)$ [21]. Estos diferentes algoritmos fueron estudiados y desarrollados en su manera óptima para un mejor desarrollo del ejercicio y contar con una complejidad de tiempo menor.

Sin embargo, como se acoplo del Algoritmo de Legendre Symbol, para el correcto funcionamiento con el código que se estableció, calculando su función de tiempo, tenemos

$$T_L(a,p) = t_1 + \sum_{i=1}^{\frac{p-1}{2}} t_2 + t_3$$

$$T_L(a,p) = t_1 + t_2\left(\frac{p-1}{2} - 1 + 1\right) + t_3$$

Si cada t_i consume una unidad de tiempo, entonces el tiempo del algoritmo seria

$$T_L(a,p) = \frac{p-1}{2} + 2$$

Así podemos concluir que el Algoritmo Encontrar Potencia cuenta con una complejidad polinomial univariada de $O(p)$.

Luego, calculando la función de tiempo del algoritmo Encontrar Potencia. Sea $T(n)$ la función de tiempo del algoritmo, siendo n el numero ingresado, teniendo en cuenta que la cantidad de dígitos de un número es aproximadamente $\log n$, entonces

$$T(n) = t_1 + B + t_2 + t_3 + \sum_{i=\frac{(\log n - 1)}{\log_{10} x}}^{\frac{\log n}{\log_{10} x}} t_4 + t_5$$

Como el Algoritmo de Encontrar Base es un caso especial del Algoritmo Factores Primos $O(B) \leq O(\log n)$, entonces

$$T(n) = t_6 + \log n + t_4\left(\frac{\log n}{\log_{10} x} - \frac{\log n - 1}{\log_{10} x}\right) + 1$$

Donde x representa la base obtenida en el Algoritmo Encontrar Base

$$T(n) = t_6 + \log n + \frac{t_4}{\log_{10} x} + 1$$

Si cada t_i consume una unidad de tiempo, entonces el tiempo del algoritmo seria

$$T(n) = \log n + \frac{1}{\log_{10} x} + 1$$

Así podemos concluir que el Algoritmo Encontrar Potencia cuenta con una complejidad polinomial univariada de $O(\log n)$.

Finalmente, después de calcular cada uno de los algoritmos que conforman el algoritmo principal Algoritmo Congruencia Cuadrática Mejorado, podemos calcular su tiempo. Primeramente,

$$T(a,p) = \log p\,, \qquad si\ a = 0\ or\ a = 1$$

Donde solo se realiza la factorización del número p, por esta razón solo cuenta con su complejidad. Luego, contamos con dos casos, el caso donde $p|a$ y donde $p \nmid a$, en los dos casos primeramente se calcula el gcd con complejidad de $O(\log p)$, mirando primero el caso donde $p \nmid a$ y p impar, contamos con

$$T(a,p) = \log p + p\log p = (p+1)\log p$$

Siendo la complejidad de la simbología de Legendre, junto con la complejidad de Encontrar Base $O(p\log p)$, por consecuente la complejidad en el caso de $p \nmid a$ seria $O(p\log p)$, debido a que

$O(\log p) \leq O(p \log p)$. Inmediatamente, mirando el caso donde $p \nmid a$ y p par, contaría con una complejidad de

$$T(a,p) = \log p$$

Debido a que solo realizaría la función de la factorización en números primos. Por consecuente, su complejidad es de $O(\log p)$.

Ahora, mirando el caso donde $p|a$ contamos que se realiza el Algoritmo Encontrar Potencia dos veces, entonces

$$T(a,p) = \log p + \log a + \log p$$
$$T(a,p) = 2\log p + \log a$$

Conociendo que $a < p$ en todas las situaciones debido a que $a = a\,\%\,p$, por consecuente la complejidad en el caso de $p|a$ seria $O(2\log p)$.

Entonces, podemos concluir y confirmar que este algoritmo cuenta con diferentes caminos con diferentes complejidades, logrando notar directamente que se trata de un algoritmo NP-hard, debido a ser no determinístico, donde el algoritmo tiene diferentes caminos a seguir, y es capaz de escoger cual es el mejor camino para solucionar el problema. Viendo que

$$O = \begin{cases} \log p\,, & si\ a = 0\ or\ a = 1 \\ p\log p\,, & si\ p \nmid a\ y\ p\ impar \\ \log p\,, & si\ p \nmid a\ y\ p\ par \\ 2\log p\,, & si\ p|a \end{cases}$$

Presentando así complejidad tanto univariada como multivariada.

VIII. Algoritmo Inicial en Java

```
import java.math.BigInteger;
import java.util.Scanner;

/**
 * @Author: Dylan Cantillo
 */
public class Conguencia_Cuadratica_Inicial {

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        System.out.println("Ingrese a: ");
        BigInteger a = new BigInteger(sc.next());

        System.out.println("Ingrese m: ");
        BigInteger p = new BigInteger(sc.next());

        boolean existencia = congruQuadr(a, p);

        if (!existencia) {
            System.out.println("No existe una
respuesta");
        } else {
            System.out.println("Si existe un x, tal que
x^2 (mod " + p + ") = " + a);
        }
    }

    private static boolean congruQuadr(BigInteger
a, BigInteger m) {
        BigInteger x = BigInteger.ONE;
        BigInteger potencia = x.pow(2);

        while (x.compareTo(m) == -1 &&
potencia.mod(m).compareTo(a) != 0) {
            x = x.add(BigInteger.ONE);
            potencia = x.pow(2);
        }

        return potencia.mod(m).compareTo(a) == 0;
    }
}
```

IX. Algoritmo mejorado en Java

```
import java.math.BigInteger;
import java.util.Scanner;

/**
 * @Author: Dylan Cantillo
 */
public class Congruencia_Cuadratica_Mejorado {

    private static BigInteger expoA, expoP, gcd;
    private static final BigInteger DOS = new
BigInteger(2 + "");
    private static final BigInteger CUATRO = new
BigInteger(4 + "");
```

```
    private static final BigInteger OCHO = new
BigInteger(8 + "");

    public static void main(String[] args) {

        Scanner sc = new Scanner(System.in);

        System.out.println("Ecuación general : x^2
= a (mod p)");
        System.out.println("");
        System.out.println("Ingrese a (residuo)");
        BigInteger a = new BigInteger(sc.next());
        System.out.println("Ingrese p (modular)");
        BigInteger p = new BigInteger(sc.next());
        System.out.println("");

        String existencia = inicial(a, p);
        System.out.println(existencia);
    }

    private static String inicial(BigInteger a,
BigInteger p) {

        String respuesta = "";

        String[] factores = factoresPrimos(p).split("
");

        for (String factor : factores) {
            BigInteger f = new BigInteger(factor);
            if (!congruenciaCuadratica(a, f)) {
                respuesta = "No existe una respuesta";
                break;
            }
        }

        if (respuesta.equals("")) {
            respuesta = "Si existe un x, tal que x^2
(mod " + p + ") = " + a;
        }
        return respuesta;
    }

    private static boolean congruenciaCuadratica
(BigInteger a, BigInteger p) {

        a = a.mod(p);
        if (a.compareTo(BigInteger.ZERO) == 0) {
            return true;
        } else if (a.compareTo(BigInteger.ONE) ==
0) {
            return true;
        } else {
            gcd = maximoComunDivisor(a, p);
            if (gcd.compareTo(BigInteger.ONE) ==
0) {
                if
(p.mod(DOS).compareTo(BigInteger.ONE) ==
0) {
                    if (legendre(a, new
BigInteger(encontrarBase(p) + "")) == 1) {
                        return true;
                    }
                } else {
                    if (a.compareTo(BigInteger.ONE)
== 0 && p.compareTo(DOS) == 0) {
                        return true;
                    } else if
(a.compareTo(BigInteger.ONE) == 0 &&
p.compareTo(CUATRO) == 0) {
                        return true;
                    } else {
                        if
(a.mod(OCHO).compareTo(BigInteger.ONE) ==
0) {
                            return true;
                        }
                    }
                }
            } else {
                expoA = encontrarPotencia(gcd);
                expoP = encontrarPotencia(p);
                if
(expoA.mod(DOS).compareTo(BigInteger.ONE)
== 0) {
                    return false;
                } else {
                    if
(p.mod(DOS).compareTo(BigInteger.ONE) ==
0) {
                        return true;
                    } else if
(expoP.subtract(expoA).compareTo(DOS) == 0)
{
                        return true;
                    } else if
(expoP.subtract(expoA).compareTo(CUATRO)
== 0) {
```

```
                    return true;
                } else {
                    return true;
                }
            }
        }
    }
    return false;
}

private static String factoresPrimos
(BigInteger numero) {
    String resp = "";
    int cont;
    for (BigInteger i = DOS;
i.compareTo(numero) != 1; i =
i.add(BigInteger.ONE)) {
        cont = 0;
        while
(numero.mod(i).compareTo(BigInteger.ZERO)
== 0) {
            numero = numero.divide(i);
            cont++;
            if
(numero.compareTo(BigInteger.ONE) == 0) {
                break;
            }
        }
        if (cont != 0) {
            resp += i.pow(cont) + " ";
        }
    }
    return resp;
}

private static int legendre (BigInteger a,
BigInteger p) {
    BigInteger num =
(p.subtract(BigInteger.ONE)).divide(DOS);

    int contador = 0;
    for (BigInteger i = BigInteger.ONE;
i.compareTo(num) != 1; i =
i.add(BigInteger.ONE)) {
        if
((a.multiply(i)).mod(p).compareTo(num) == 1) {
            contador++;
        }
    }
    return (int) Math.pow(-1, contador);
}

private static BigInteger
maximoComunDivisor (BigInteger a,
BigInteger p) {

    if (p.compareTo(BigInteger.ZERO) == 0) {
        return a;
    } else {
        return maximoComunDivisor(p,
a.mod(p));
    }
}

private static BigInteger encontrarPotencia
(BigInteger numero) {
    int base = 0;
    int numeroDigitos = (numero + "").length();

    base = encontrarBase(numero);

    int valorInferior, valorSuperior;
    valorInferior = (int) ((numeroDigitos - 1) /
Math.log10(base));
    valorSuperior = (int) (numeroDigitos /
Math.log10(base));

    for (int i = valorInferior; i <= valorSuperior;
i++) {
        if (new BigInteger(base +
"").pow(i).compareTo(numero) == 0) {
            return new BigInteger(i + "");
        }
    }
    return BigInteger.ZERO;
}

private static int encontrarBase (BigInteger
numero) {
    BigInteger i = new BigInteger("2");
    while
(numero.mod(i).compareTo(BigInteger.ZERO)
!= 0) {
        i = i.add(BigInteger.ONE);
    }
    return Integer.parseInt(i.toString());
}
}
```

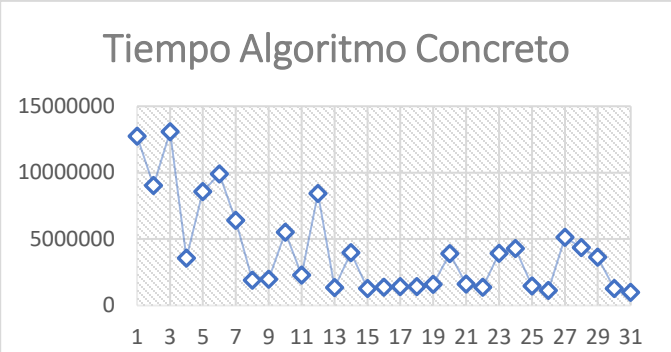
X. COMPARACIÓN ALGORITMOS Y DATOS DE TIEMPO

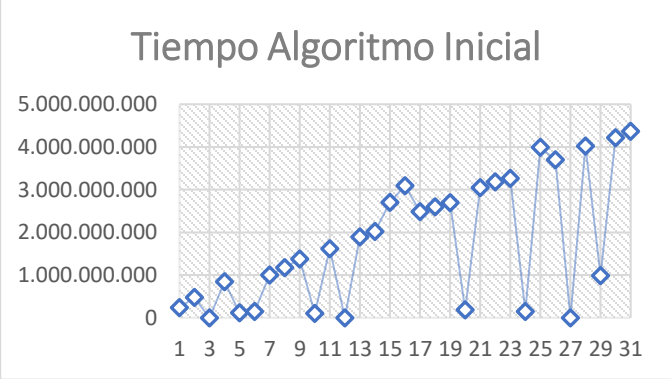
Para la prueba de datos se realizó con una cantidad de $n = 31$ pruebas donde se muestra el tiempo de demora de cada algoritmo, aquí se exponen las tablas de cada algoritmo

	Tiempo Algoritmo Mejorado	Tiempo Algoritmo Inicial
1	12748100	236536600
2	9029300	476093900
3	13072900	94200
4	3567000	845209800
5	8566800	117715500
6	9891300	146457000
7	6400000	999351400
8	1912400	1172381100
9	1980100	1373527100
10	5522200	100100200
11	2286800	1612848700
12	8437900	4000
13	1344600	1890889400
14	3969100	2022233400
15	1266300	2699779400
16	1371900	3093728300
17	1425100	2481009700
18	1432600	2598701600
19	1581100	2697485400
20	3908400	183491300
21	1616500	3045660400
22	1368800	3183397500
23	3921000	3267200500
24	4291900	149217600
25	1473900	3990605800
26	1128600	3699263600
27	5131000	2100
28	4351000	4022420800
29	3635200	984913800
30	1273600	4213818400
31	986900	4365075300

(Tabla de Datos)

Podemos obtener un promedio de tiempo de el Algoritmo Mejorado con $\mu_M = 4157816{,}129$ NanoSegundos y para el Algoritmo Inicial un promedio de tiempo de $\mu_I = 1795781090$ NanoSegundos, con una gran diferencia de $\mu_I - \mu_M = 1791623274$ NanoSegundos.

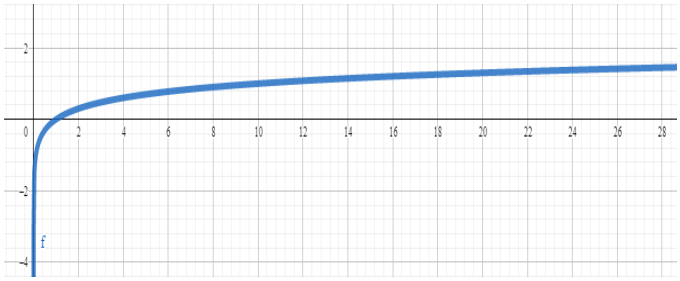




Logrando apreciar como el Algoritmo Mejorado presenta un tiempo aproximadamente decreciente y el Algoritmo Inicial un tiempo creciente lineal, esto debido a sus complejidades, como ya vimos la complejidad del Algoritmo Inicial es de $O(n)$ (con $n = p$) claramente viendo que su tiempo crece si crece p, luego el Algoritmo Mejorado presenta una complejidad dependiendo a su tipo

$$O = \begin{cases} \log p\,, & si\ a = 0\ or\ a = 1 \\ p \log p\,, & si\ p \nmid a\ y\ p\ impar \\ \log p\,, & si\ p \nmid a\ y\ p\ par \\ 2 \log p\,, & si\ p|a \end{cases}$$

entonces tiene una complejidad parecida a la función del logaritmo donde mientras va creciendo p se va aproximando a un contante. Hay que aclarar que los picos que se pueden observar son porque ese índice entro a la categoría de $O(p \log p)$, donde se consume más tiempo.



(Función logaritmo natural)

XI. Prueba de hipótesis

Se cuenta con dos algoritmos (Inicial, Mejorado) que dan solución de la existencia de la congruencia cuadrática. Nos encontramos interesados en estimar la verdadera diferencia entre los tiempo de solución promedio de los algoritmos. Para ellos se realizaron 31 pruebas para cada algoritmo (Tabla de Datos). Haremos la afirmación de que el Algoritmo Inicial es mejor que el Algoritmo Mejorado ¿La afirmación es correcta? Ahora responderemos a la pregunta, considerando un nivel de significancia del $\alpha = 5\%$, esto debido a que el nivel de confianza $1 - \alpha = 95\%$ es más preciso y a su vez garantiza una alta confiabilidad [22].

Teniendo en cuenta la afirmación realizada, tenemos que $\mu_1 < \mu_2$ (1: Inicial, 2: Mejorado), así estableciendo que el algoritmo 1 es mejor. Con base a esta afirmación, planteamos nuestra hipótesis

$$H_0: \mu_1 - \mu_2 \geq 0$$
$$H_1: \mu_1 - \mu_2 < 0$$

Presentadas así, debido a que la hipótesis nula siempre lleva el signo de igualdad, entonces en este caso nuestra afirmación es igual a la hipótesis alternativa.

Extrayendo nuestra información del enunciado y la tabla de datos, tenemos

$$n_1 = 31, \qquad \bar{x}_1 = 1795781090$$
$$S_1^2 = (1487209220)^2$$

$$n_2 = 31, \qquad \bar{x}_2 = 4157816{,}129$$
$$S_2^2 = (3483996{,}047)^2$$

$$\alpha = 0{,}05, \qquad \Delta_0 = 0$$

Sea X_i ={Tiempo que tarda el algoritmo i en resolver la congruencia cuadrada | i ={Inicial (1), Mejorado (2)}}, donde toma valores $x_i > 0$.

De acuerdo con las condiciones, no tenemos seguro si nuestra variable X_i distribuye normal, entonces suponemos que no distribuye normal ($\bar{X}_i \nsim Normal$), luego observando el tamaño de las muestras $n_1 = 31 \geq 30$ y $n_2 = 31 \geq 30$ entonces por el teorema del límite central (TLC) [23] podemos decir que $\bar{X}_i \sim Normal$, luego también sabemos que son muestras aleatorias independientes, finalmente no conocemos las varianzas poblacionales σ_i^2 y tampoco sabemos si su comportamiento es de igualdad o no, entonces procedemos a hallar este resultado.

Primeramente, hacemos una suposición de igualdad donde $\sigma_1^2 = \sigma_2^2$, con base en esta nueva afirmación, planteamos nuestra hipótesis

$$H_0: \sigma_1^2 = \sigma_1^2$$
$$H_1: \sigma_1^2 \neq \sigma_1^2$$

Contamos con los mismos datos numéricos, solo que ahora nuestra diferencia (Δ) se convierte en un cociente, donde

$$v1 = n_1 - 1 = 30, \qquad v2 = n_2 - 1 = 30, \qquad \Delta_0 = 1$$

Como ya sabemos $\bar{X}_i \sim Normal$, además son muestras aleatorias independientes, entonces nos encontramos en el caso de prueba de hipótesis para el cociente de varianza, donde $\bar{X} \sim F$ (Fisher)

$$F_0 = \frac{\sigma_2^2}{\sigma_1^2} \cdot \frac{S_1^2}{S_2^2} = \Delta_0 \cdot \frac{S_1^2}{S_2^2}$$

Tiene una distribución F con v1=n_1-1=10 grados de libertad en el numerador y v2=n_2-1=11 grados de libertad en el denominador.

Reemplazando nuestros datos numéricos, tenemos

$$F_0 = 1 \cdot \frac{(1487209220)^2}{(3483996{,}047)^2} = 182216{,}93$$

Luego, realizando nuestra toma de decisión con valor P (probabilidad), tenemos

$$VP = P\left(F_{v1,\ v2} > F_0\right)$$

Con un signo >, debido a que la distribución Fisher (F) es asimétrica.

$$VP = P\left(F_{30,\ 30} > 1{,}17\right) \approx 0$$

Donde, nuestra región de rechazo esta dada por

$$RR = [-\infty,\ 0.025] \cup [0.975,\ \infty]$$

Por consecuente, como $VP \approx 0 < 0{,}025$. Concluimos que se rechaza H_0 para un nivel de significancia de 0,05, además como $VP < 0{,}001$ se rechaza H_0 para cualquier nivel de significancia. Entonces, significa que $\sigma_1^2 \neq \sigma_1^2$.

Continuando nuestro ejercicio principal, conociendo que $\sigma_1^2 \neq \sigma_1^2$, entonces nos encontramos en el caso IV de prueba de hipótesis para la diferencia de media, donde $\bar{X} \sim T$ (t-student)

$$T_0 = \frac{(\bar{x}_1 - \bar{x}_2) - (\mu_1 - \mu_2)}{\sqrt{\frac{s_1^2}{n_1} + \frac{s_2^2}{n_2}}} = \frac{(\bar{x}_1 - \bar{x}_2) - \Delta_0}{\sqrt{\frac{s_1^2}{n_1} + \frac{s_2^2}{n_2}}}$$

Teniendo en cuenta que los grados de libertad son

$$v = \frac{\left(\frac{s_1^2}{n_1} + \frac{s_2^2}{n_2}\right)^2}{\frac{\left(\frac{s_1^2}{n_1}\right)^2}{n_1 - 1} + \frac{\left(\frac{s_2^2}{n_2}\right)^2}{n_2 - 1}}$$

Reemplazando nuestros datos numéricos, tenemos que

$$v = \frac{\left(\frac{(1487209220)^2}{31} + \frac{(3483996{,}047)^2}{31}\right)^2}{\frac{\left(\frac{(1487209220)^2}{31}\right)^2}{31 - 1} + \frac{\left(\frac{(3483996{,}047)^2}{31}\right)^2}{31 - 1}} \approx 30$$

$$T_0 = \frac{(1795781090 - 4157816{,}129) - 0}{\sqrt{\frac{(1487209220)^2}{31} + \frac{(3483996{,}047)^2}{31}}} = 6.7074$$

Realizando nuestra toma de decisión con valor P (probabilidad)

$$VP = P(T_v < T_0), \qquad con\ un\ signo\ igual\ que\ H_1$$
$$VP = P(t_{30} < 6.7074) \approx 1$$

Por consecuente, como $VP \approx 1 > 0{,}05 = \alpha$. Concluimos que no se rechaza H_0 con un nivel de significancia de 0,05. Además, como $VP \approx 1 > 0{,}1$ afirmamos que no se rechaza H_0 para cualquier nivel de significancia. Entonces, $\mu_1 - \mu_2 \geq 0$, es decir, $\mu_1 \geq \mu_2$, estableciendo que la afirmación inicial es falsa, por lo tanto el Algoritmo Mejorado es mejor que el algoritmo Inicial.

Logrando concluir y afirma que el Algoritmo Inicial cuenta con un tiempo de demora mayor al tiempo de demora del Algoritmo Mejorado, también muy visible en la diferencia de las medias al ser tan grande.

XII. Conclusiones

La investigación y desarrollo, presentado anteriormente, demuestra que, a pesar de ser un problema ya resuelto, hay diversas maneras y complejidades para la resolución del problema, logrando apreciar con gran claridad la estrecha relación de la congruencia cuadrática con los números primos y además su estrecha relación con los problemas NP-Hard, lográndose apreciar con la complejidad del algoritmo.

$$O = \begin{cases} \log p\,, & si\ a = 0\ or\ a = 1 \\ p \log p\,, & si\ p \nmid a\ y\ p\ impar \\ \log p\,, & si\ p \nmid a\ y\ p\ par \\ 2 \log p\,, & si\ p | a \end{cases}$$

Donde, el algoritmo busca la mejor opción para resolver la ecuación $x^2 \equiv a\ (mod\ p)$, sin ser una opción con menos rendimiento o eficacia que las demás, hacia el caso.

En el artículo realizo primeramente una amplia búsqueda e introducción al gran tema de la teoría de números, más específicamente en la aritmética modular, la cual le dio paso el gran Pierre de Fermat, uno de los grandes matemáticos del siglo XVII y siendo prolongada por Leonhard Euler. Dándole no solo respuesta a la geometría antigua y moderna, sino también estableciendo nuevos conceptos para la limitada vista de los números primos, factorización de enteros, criptografía y hasta en la acústica [24].

En la comparación del Algoritmo Inicial y el Algoritmo Mejorado, se logró apreciar una gran diferencia entre estos tanto en su codificación como en la función de tiempo. El Algoritmo Inicial fue

muy sencillo de codificar sin embargo su complejidad era polinomial univarida $O(p)$ dependiendo del número ingresado, presentando además una gráfica lineal creciente. Por el otro lado, el Algoritmo Mejorado fue más complicado de codificar debido a la complejidad de las funciones matemáticas por la implementación del criterio de Euler y símbolo de Legendre, pero a pesar de esto, presento una gran mejora en la complejidad, lográndose adaptar a la situación de la entrada de datos, con una complejidad de

$$O = \begin{cases} \log p\,, & si\ a = 0\ or\ a = 1 \\ p \log p\,, & si\ p \nmid a\ y\ p\ impar \\ \log p\,, & si\ p \nmid a\ y\ p\ par \\ 2 \log p\,, & si\ p|a \end{cases}$$

donde el peor caso lo encontramos cuando p no divide a y p es impar, es decir, p y a tienen una común divisor igual de 1. Y presentando una gráfica que iba disminuyendo a una constante, debido a su relación con el logaritmo natural.

Al comparar las gráficas de ambos algoritmos logramos apreciar una gran diferencia de consumo de tiempo y para comprobar si esta suposición era cierta, se realizó una prueba de hipótesis con una confianza del 95%, donde confirmamos el hecho de que el tiempo de demora del Algoritmo Inicial era mayor que el tiempo de demora del Algoritmo Mejorado ($\mu_I > \mu_M$).

De esta manera, queda en presencia el hecho de la importancia de la congruencia cuadrática para un ámbito principalmente matemático, al ser rodeado de los números primos, la mejora de uno conllevara a la mejora del otro. Matemáticos e informáticos han trabajado conjuntos en la resolución de estos problemas que cada que se hacen más complejos, más ventajas le dan a la criptografía de datos, pero cada que logran algún avance en su solución, también se logra un progreso en las matemáticas universales, siendo así un gran complemento de sistemas.

Se invita de esta manera al lector, si quiere conocer más sobre el tema de la congruencia cuadrática, la criptografía y la teoría de números a continuar adquiriendo conocimiento por medio de estos textos que considero muy importantes [25] [26].

XIII. Referencias

[1] M. Křížek, F. Luca and L. Somer, Desde los numéros de Fermat hasta la geometría, 10.2th ed. LA GACETA DE LA RSME, 2007, pp. 471–483.

[2] Gauß, C. F., & Clausen, T. (1855). Thomas Clausen→ Carl Friedrich Gauß, Dorpat, 1855 Jan. 1/13.

[3] Osborn, Roger. «A "Good" Generalization of the Euler-Fermat Theorem». Mathematics Magazine, vol. 47, n.o 1, enero de 1974, p. 28. DOI.org (Crossref), doi:10.2307/2688758.

[4] Brassard, Gilles, y Paul Bratley. Fundamentos de algoritmia. Pearson Prentice Hall, 2008.

[5] Garey, Michael R., y David S. Johnson. Computers and Intractability: A Guide to the Theory of NP-Completeness. 27. print, Freeman, 2009.

[6] Plaisted, David A. «New NP-Hard and NP-Complete Polynomial and Integer Divisibility Problems». Theoretical Computer Science, vol. 31, n.o 1-2, 1984, pp. 125-38. DOI.org (Crossref), doi:10.1016/0304-3975(84)90130-0.

[7] Sesiano, Jacques y Diophantus. Books IV to VII of Diophantus' Arithmetica in the Arabic Translation Attributed to Qusṭā Ibn Lūqā. Springer-Verlag, 1982.

[8] Sandifer, Charles Edward. How Euler did it. Mathematical Association of America, 2007.

[9] Bach, Eric, y Jeffrey Outlaw Shallit. Algorithmic number theory. MIT Press, 1996.

[10] Zhan, Justin, et al. «Towards a Novel Generalized Chinese Remainder Algorithm for Extended Rabin Cryptosystem». IEEE Access, vol. 8, 2020, pp. 26433-44. DOI.org (Crossref), doi:10.1109/ACCESS.2020.2967396.

[11] Chau, H. F., y H. K. Lo. «Primality Test Via Quantum Factorization». International Journal of Modern Physics C, vol. 08, n.o 02, abril de 1997, pp. 131-38. DOI.org (Crossref), doi:10.1142/S0129183197000138.

[12] Naudin, Patrice, y Claude Quitté. «Univariate Polynomial Factorization over Finite Fields». Theoretical Computer Science, vol. 191, n.o 1-2, enero de 1998, pp. 1-36. DOI.org (Crossref), doi:10.1016/S0304-3975(97)80001-1.

[13] Ţiplea, Ferucio Laurenţiu, et al. «On the Distribution of Quadratic Residues and Non-Residues modulo Composite Integers and Applications to Cryptography». Applied Mathematics and Computation, vol. 372, mayo de 2020, p. 124993. DOI.org (Crossref), doi:10.1016/j.amc.2019.124993.

[14] Miller, Gary L. «Riemann's Hypothesis and Tests for Primality». Proceedings of Seventh Annual ACM Symposium on Theory of Computing - STOC '75, ACM Press, 1975, pp. 234-39. DOI.org (Crossref), doi:10.1145/800116.803773.

[15] Hall, Marshall. «Quadratic Residues in Factorization». Bulletin of the American Mathematical Society, vol. 39, n.o 10, octubre de 1933, pp. 758-64. DOI.org (Crossref), doi:10.1090/S0002-9904-1933-05730-0.

[16] BCMATH CONGRUENCE PROGRAMS. http://www.numbertheory.org/php/congruences.html. Accedido 1 de mayo de 2021.

[17] Quadratic reciprocity law in nLab. https://ncatlab.org/nlab/show/quadratic+reciprocity+law. Accedido 1 de mayo de 2021.

[18] Chapman, Robin. Quadratic reciprocity. diciembre de 2003.

[19] Shallit, Jeffrey. «Origins of the Analysis of the Euclidean Algorithm». Historia Mathematica, vol. 21, n.o 4, noviembre de 1994, pp. 401-19. DOI.org (Crossref), doi:10.1006/hmat.1994.1031.

[20] Shallit, Jeffrey, y Jonathan Sorenson. «A Binary Algorithm for the Jacobi Symbol». ACM SIGSAM Bulletin, vol. 27, n.o 1, enero de 1993, pp. 4-11. DOI.org (Crossref), doi:10.1145/152379.152384.

[21] Shor, Peter W. «Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer». SIAM Journal on Computing, vol. 26, n.o 5, octubre de 1997, pp. 1484-509. DOI.org (Crossref), doi:10.1137/S0097539795293172.

[22] Candia B, Roberto, y Gianella Caiozzi A. . «Intervalos de Confianza». Revista médica de Chile, vol. 133, n.o 9, septiembre de 2005, pp. 1111-15. SciELO, doi:10.4067/S0034-98872005000900017.Walpole, Ronald E. Probabilidad y estadística para ingeniería y ciencias. Pearson, 2012.

[23] Walker, R. (1990). The design and application of modular, acoustic diffusing elements. Unknow.

[24] Jara, P., & Rodríguez, M. L. Solving quadratic congruences. by Marius Dragan, Mihály Bencze and Neculai Stanciu 127 Genenralization of Problem EM-55, 105.

[25] Scheidler, R. E. N. A. T. E. (1996). Cryptography in real quadratic congruence function fields. In Proceedings of PRAGOCRYPT (Vol. 96).