UNIVERSIDAD MAYOR DE SAN ANDRÉS FACULTAD
DE CIENCIAS PURAS Y NATURALES
CARRERA DE INFORMÁTICA

TESIS DE GRADO
COMPARACIÓN Y REFINAMIENTO DE ALGORITMOS DE FACTORIZACIÓN DE NÚMEROS ENTEROS

Tesis de Grado para obtener el Título de Licenciatura en Informática Mención Ciencias de la Computación

POR: ROLANDO TROCHE VENEGAS
TUTOR: M.SC. JORGE HUMBERTO TERÁN POMIER

LA PAZ - BOLIVIA

2024

DEDICATORIA

Dedicado a mi familia, por creer en mí;

a mis profesores, por iluminar el camino;

AGRADECIMIENTOS

Quiero expresar mi más sincero agradecimiento a todos aquellos que me han brindado su apoyo y guía durante la elaboración de esta tesis:

A mis padres, por su incondicional apoyo a lo largo de mi carrera y por ser mi fuente constante de inspiración y motivación.

A mi tutor, el M. Sc. Jorge Teran Pomier, por su invaluable apoyo, confianza y orientación durante todo el proceso de elaboración de esta tesis.

Al Club de Programación Competitiva U.M.S.A., por todo lo aprendido y las experiencias compartidas, las cuales han enriquecido mi formación y desarrollo profesional.

A mis amigos, "Los Osos", por ser los mejores compañeros que pude tener y por su compañía durante todo este camino en la universidad.

Finalmente, al lector, gracias por darle vida y sentido a este trabajo con su interés y atención.

rtrochevenegas@gmail.com

RESUMEN

El teorema fundamental de la aritmética establece que todo número puede expresarse como un producto de números primos o ser un número primo. En matemáticas y, más recientemente, en ciencias de la computación, un problema de gran importancia es encontrar el conjunto de números primos que, al multiplicarse, nos den el número original. Esta propiedad fundamental no solo es interesante desde un punto de vista teórico, sino que también tiene aplicaciones prácticas significativas en diversas áreas de la ciencia y la tecnología.

Aunque desde la antigüedad se conocen algoritmos para realizar esta tarea, hoy en día cobra una relevancia particular debido a que gran parte de la seguridad informática se basa en la dificultad de factorizar números muy grandes. A pesar de que existen algoritmos eficientes y con complejidades polinomiales para esta tarea, el enorme tamaño de los números utilizados en la criptografía moderna hace que la factorización siga siendo un desafío considerable. Esta dificultad es precisamente la que garantiza la seguridad de muchos sistemas criptográficos, como RSA, que dependen de la factorización de números compuestos muy grandes.

En esta tesis se realizará un análisis y comparación de varios algoritmos y métodos de factorización para determinar cuáles son los más adecuados según el tipo de número y el contexto en el que se deben usar. Además, se abordará el refinamiento de estos métodos durante su implementación en un lenguaje de programación moderno. Se considerarán varios enfoques, desde los más clásicos como el método de la división por prueba hasta los más avanzados como el algoritmo de factorización de Lenstra basado en curvas elípticas y el método general de números.

La elección del algoritmo adecuado no solo depende del tamaño del número a factorizar, sino también de otros factores como la disponibilidad de recursos computacionales y el tiempo disponible para el proceso de factorización. Por ejemplo, para números relativamente pequeños, los métodos clásicos pueden ser suficientemente rápidos y eficientes. Sin

embargo, para números extremadamente grandes, se requieren métodos más avanzados y sofisticados que aprovechen propiedades matemáticas más profundas y que puedan ser paralelizados en sistemas de computación distribuida.

Además de la comparación de algoritmos, esta tesis también explorará la implementación práctica de estos métodos. Se evaluará el rendimiento en un lenguaje de programación moderno. Se realizarán experimentos para medir el tiempo de ejecución y la eficiencia de los algoritmos implementados en Python.

Palabras clave: Factorización, números primos, teorema fundamental de la aritmética, factores primos.

ABSTRACT

The fundamental theorem of arithmetic states that every number can be expressed as a product of prime numbers or be a prime number. In mathematics and, more recently, in computer science, a problem of great importance is finding the set of prime numbers that, when multiplied, give us the original number. This fundamental property is not only interesting from a theoretical point of view, but also has significant practical applications in various areas of science and technology.

Although algorithms for this task have been known since ancient times, it is particularly relevant today because much of computer security is based on the difficulty of factoring very large numbers. Although efficient algorithms with polynomial complexity exist for this task, the enormous size of the numbers used in modern cryptography means that factorization remains a considerable challenge. This difficulty is precisely what guarantees the security of many cryptographic systems, such as RSA, which depend on the factorization of very large composite numbers.

This thesis will analyze and compare several factorization algorithms and methods to determine which ones are the most suitable for the type of number and the context in which they are to be used. In addition, the refinement of these methods during their implementation in a modern programming language will be addressed. Several approaches will be considered, from the most classical ones such as the trial division method to the most advanced ones such as the Lenstra factorization algorithm based on elliptic curves and the general number method.

The choice of the appropriate algorithm not only depends on the size of the number to be factored, but also on other factors such as the availability of computational resources and the time available for the factorization process. For example, for relatively small numbers, classical methods can be sufficiently fast and efficient. However, for extremely large numbers, more advanced and sophisticated methods are required that take advantage of deeper

mathematical properties and that can be parallelized in distributed computing systems.

In addition to the comparison of algorithms, this thesis will also explore the practical implementation of these methods. Performance in a modern programming language will be evaluated. Experiments will be performed to measure the execution time and efficiency of the algorithms implemented in Python.

Keywords: Factorization, prime numbers, fundamental theorem of arithmetic, prime factors.

ÍNDICE

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

ÍNDICE DE ALGORITMOS

CAPÍTULO 1 ANTECEDENTES GENERALES O MARCO REFERENCIAL

1.1. INTRODUCCIÓN

Factorizar números enteros es importante, hoy más que nunca la factorización de números enteros juega un papel crucial en la vida de todas las personas. Los métodos de encriptación actuales se basan, en gran medida, en la complejidad y el tiempo que toma factorizar números grandes.

También se debe notar la definición de número grande, si se le pregunta a un niño que es un número grande este puede decirnos que es el 100 o hasta el 1000 y si se le preguntara a una persona adulta esta podría decirnos 1 000 000 o 100 000 000, pero esto no es nada si se piensa en los problemas que hoy en día lidian los matemáticos.

Factorizar un número se refiere a encontrar todos los factores primos por los que está compuesto dicho número. El teorema fundamental de la aritmética nos dice que para todo número entero positivo mayor a 1 es un numero primo o bien un único producto de números primos (Euclides, 300 AC).

$$n = \prod_{i=1}^{k} p_i^{a_i}$$

Este es el objetivo que buscamos, encontrar esos factores para números, ahora sí, grandes.

Los algoritmos de encriptación actuales, como RSA, usan números de, por ejemplo, 250 dígitos (829 bits) que fue el último en ser factorizado febrero de 2020. Estos son los números grandes que nos interesan. Los números RSA por ejemplo son números con exactamente dos factores primos, estos números primos, de al menos, la mitad de cantidad de dígitos que el resultado.

Hoy en día tenemos diferentes métodos de factorización como los algoritmos simples de factorización, fracciones continuas, curva elíptica, algoritmos de cribado y hasta nuevos algoritmos basados en programación cuántica.

En este trabajo se estudiará los diferentes métodos de factorización de números y su implementación con diferentes enfoques y refinamientos de implementación que el autor propondrá para ciertos métodos. Con esto se podrá revisar los alcances de los algoritmos de encriptación actuales y a donde se puede llegar en base a estos y realizar una comparación de los mismo, mostrando que algoritmo es el adecuado dada ciertas condiciones.

1.2. PROBLEMA

1.2.1. ANTECEDENTES

Los algoritmos de factorización de números enteros han sido y son un area de interés en el campo de la teoría de números, criptografía y ciencias de la computación, los primeros algoritmos para realizar esta tarea sn tan antiguos como la matemática misma y con el pasar de los años se fueron agregando nuevos y mejores algoritmos y métodos para factorizar números enteros.

Las siguientes investigaciones proporcionan un marco importante para el estudio de los algoritmos de factorización de números y han influido significativamente en el desarrollo de este campo:

- **Título:** "A monte carlo method for factorization"
  Autor: J. M. Pollard
  Año: 1975
  Institución: BIT Numerical Mathematics
  Resumen: Se describe brevemente un nuevo método de factorización que involucra ideas probabilísticas y se sugiere que este método debería considerarse como una alternativa viable a los métodos de factorización tradicionales. (Pollard, 1975).

- **Título:** "Theorems on factorization and primality testing"
  Autor: J. M. Pollard

Año: 1974

Institución: Mathematical Proceedings of the Cambridge Philosophical Society

Resumen: Este artículo trata del problema de obtener estimaciones teóricas para el número de operaciones aritméticas necesarias para factorizar un entero grande n o comprobar su primalidad. (Pollard, 1974).

- **Título:** "A one line factoring algorithm"

  Autor: W. B. Hart

  Año: 2012

  Institución: Journal of the Australian Mathematical Society

  Resumen: Describimos una variante del algoritmo de factorización de Fermat que es competitiva con SQUFOF en la práctica, pero tiene una complejidad de tiempo de ejecución heurística $O(n^{1/3})$ como algoritmo de factorización general. También describimos una clase dispersa de números enteros para los que el algoritmo es particularmente eficaz. Ofrecemos comparaciones de velocidad entre una implementación optimizada del algoritmo descrito y la variedad optimizada de algoritmos de factorización en el paquete de álgebra computacional Pari/GP. (Hart, 2012).

- **Título:** "A method of factoring and the factorization of F7"

  Autor: M. A. Morrison and J. Brillhar

  Año: 1975

  Institución: Mathematics of Computation,

  Resumen: Se analiza el método de fracciones continuas para factorizar números enteros, introducido por D. H. Lehmer y R. E. Powers, junto con su implementación informática. La potencia del método se demuestra con la factorización del séptimo número de Fermât F_7 y otros grandes números de interés. (Morrison & Brillhar, 1975).

1.2.2. PLANTEAMIENTO DEL PROBLEMA

Con el avance de las computadoras y el mayor poder de cómputo, toda la seguridad basada en factorización y factores primos corre riesgo de quedar deprecada algún día.

Por esto es necesario tener un compendio de gran parte de los métodos y algoritmos de factorización existentes y sus limitaciones, con el poder de cómputo actual. Por otro lado, hoy en día con la ayuda de los nuevos y mejores procesadores muchos algoritmos pueden ser paralelizados lo que reduciría su tiempo de ejecución.

1.2.3. FORMULACIÓN DEL PROBLEMA

¿Qué método de factorización de números enteros es mejor de acuerdo a las características de los números?

1.3. OBJETIVOS

1.3.1. OBJETIVO GENERAL

Estudiar y realizar una evaluación de números enteros con diferentes características y aplicando sobre ellos algoritmos de factorización de números enteros, comparando el tiempo de ejecución, espacio en memoria y factores primos encontrados.

1.3.2. OBJETIVOS ESPECÍFICOS

- Revisar el estado del arte referido a algoritmos de factorización de números enteros.
- Programar algoritmos de factorización de números enteros.
- Refinamiento de métodos de factorización de números enteros.

- Evaluar el desempeño de los programas con base en tiempo de ejecución, espacio en memoria y factores primos encontrados.

1.4. HIPÓTESIS

Para números donde la diferencia entre factores primos sea pequeña el algoritmo de Fermat es el mejor en cuanto a tiempo de ejecución, para números en general y de uso cotidiano el que presenta un mejor tiempo promedio es Pollard Rho y los métodos de cribado.

1.5. JUSTIFICACIÓN

1.5.1. JUSTIFICACIÓN ECONÓMICA

Con el constante avance de la tecnología y la ciencia en diferentes campos hace que cada día se necesite de mejores equipos, maquinaria, software, hardware entre otros, lo cual implica costos gigantescos, por lo cual hacer una redefinición en las herramientas teóricas resulta ser un gran ahorro y reducción de gastos para las diferentes investigaciones realizadas por universidades, instituciones del estado, comunidades científicas.

Además de una reducción de tiempos, el cual es representado a su vez en una reducción de costos, mejorando así la accesibilidad a nuevos campos de investigación.

1.5.2. JUSTIFICACIÓN SOCIAL

La sociedad en su conjunto se beneficia indirectamente ya que la presente investigación está enfocada al área teórica pero enteramente ligada a futuros campos de investigación y aplicación para mejorar el estilo de vida, hambre de conocimientos y experimentación por parte de la población en general.

1.5.3. JUSTIFICACIÓN CIENTÍFICA

La presente investigación brinda al campo científico tecnológico un complejo análisis de diferentes algoritmos de factorización, lo cual conlleva a tener nuevos y/o actualizaciones de los mismos generando avances en diferentes campos no solo del área sino también de otros campos afines, generando propuestas o aplicaciones de la presente investigación.

Por lo tanto, al emprender la investigación de los algoritmos de factorización de números enteros grandes y su evaluación, profundiza el conocimiento, aportando así a futuras investigaciones, ya que se está trabajando en un área en desarrollo. Así también como bases prácticas y teóricas para la aplicación de dichos algoritmos en áreas como el análisis complejo de números primos, representación del conocimiento, teoría de números, criptografía, combinatoria, entre otros.

1.6. ALCANCES Y LIMITES

1.6.1. ALCANCES

- Se implementaran los métodos teoricos descritos en un lenguage de programacion moderno.
- Se realizará un análisis teórico mediante una función que limitará el tiempo de cálculo del algoritmo y una prueba experimental de dónde se recogerán estadísticas de tiempo consumido por los diferentes algoritmos.
- Se refinara los métodos a nivel de implementacion.
- Se comparara los resultados de los diferentes tiempos de ejecucion y memoria de cada algoritmo

1.6.2. LIMITES

- No se tomara en cuenta algoritmos cuanticos.
- No se tomara en cuenta algoritmos hibridos.
- No se demostrara la correctitud de los métodos de manera teorica.

1.7. METODOLOGÍA

Para el desarrollo del presente trabajo se utilizará el método lógico inductivo ya que partiremos de casos particulares de factorización de números enteros para posteriormente llegar a una conclusión respecto a cada uno de los algoritmos que serán comparados y refinados.

CAPÍTULO 2 MARCO TEÓRICO

En el desarrollo de este capítulo se plantea la teoría relacionada con divisibilidad, números primos, factorización, congruencia, diferentes definiciones en teoría de números que serán fundamentales para el desarrollo de los posteriores algoritmos.

2.1. NÚMEROS ENTEROS

La teoría de números se ocupa de las propiedades de los números naturales $1, 2, 3, 4, \ldots$, también llamados enteros positivos. Estos números, juntos con los enteros negativos y cero, forman el conjunto de números enteros (Niven, Zuckerman & Montgomery, 1991).

Para el desarrollo de las siguientes definiciones y algoritmos usaremos la notación de conjuntos para referirnos a los números naturales $\mathbb{N} = 1, 2, 3, 4, \ldots$ y números enteros $\mathbb{Z} = \ldots, -2, -1, 0, 1, 2, \ldots$ y usualmente si se habla de "número" nos referiremos a un número entero.

2.2. DIVISIBILIDAD

Un entero a se dice que es divisible por otro entero b, no 0, si es que existe un tercer entero c tal que $a = b \cdot c$ Si a y b son positivos, c es necesariamente positivo. Se expresa el echo de que a es divisible por b, o b es un divisor de a, con $b \mid a$ Por lo tanto $1 \mid a$, $a \mid a$; y que $b \mid 0$ para cualquier b excepto 0. También se puede decir que $b \nmid a$ para expresar lo contrario a $b \mid a$. Otras propiedades que podemos notar son:

- $b \mid a \wedge c \mid b \implies c \mid a$
- $b \mid a \implies b \cdot c \mid a \cdot c$
- Si $c \neq 0$ y $c \mid a \wedge c \mid b \implies c \mid m \cdot a + n \cdot b$ para todos los enteros m y n

$$b \mid a \wedge c \mid b \implies c \mid a$$

$$b \mid a \implies b \cdot c \mid a \cdot c$$

Si $c \neq 0$ y $c \mid a \wedge c \mid b \implies c \mid m \cdot a + n \cdot b$ para todos los enteros m y n (Hardy & Wright, 1975).

Definimos "$n \mod m$" como el resto cuando el entero n es divido por el entero positivo m. Siempre se tiene que $0 \leq n \mod m < m$. Si al menos uno de los enteros n, m no es cero, se define el máximo común divisor (*greatest common divisor*) de n y m, como $gcd(m, n)$, al mayor entero que divida a ambos m y n. Esta claro que $gcd(m, n) \geq 1$ y que $gcd(m, n) = gcd(n, m)$. Se dice que los enteros m, n son primos relativos o coprimos si $gcd(m, n) = 1$ (S. S. Wagstaff, 2013).

El siguiente enunciado es util para los algoritmos que usaremos más adelante, si m es un entero positivo y n, q, r son enteros tal que $n = m \cdot q + r$, entonces $gcd(n, m) = gcd(m, r)$ y su demostración es la siguiente:

Si $a = gcd(n, m)$ y $b = gcd(m, r)$. Dado que a divide a ambos n y m, también debe dividir a $r = n - m \cdot q$. Esto muestra que a es un común divisor de m y r, entonces este debe ser $\leq b$, su máximo común divisor. Igualmente, dado que b divide a ambos m y r, este debe dividir a n, entonces $b \leq a = gcd(n, m)$. Por lo tanto $a = b$.

2.3. NOTACIÓN BIG-O

Para las funciones dadas $g(n)$, denotaremos por $O(g(n))$ (pronunciado como "big-oh de g de n" o solamente "oh de g de n") al conjunto de funciones

$$O(g(n)) = \{f(n) : \text{si existe una constante positiva } c \text{ y tal que } 0 \leq f(n) \leq c \cdot g(n)\}.$$

Usaremos la notación Big O para dar un limite superior a una función, dentro de un factor constante (Cormen, Leiserson, Rivest & Stein, 2009)

La notación Big O es una forma de describir el comportamiento de un algoritmo en términos de la cantidad de tiempo o espacio que necesita a medida que el tamaño de la entrada crece. Se utiliza principalmente en el análisis de algoritmos para medir su eficiencia y compararlos. La notación Big O se centra en el crecimiento asintótico del tiempo o espacio requerido, lo que significa que describe cómo se comporta el algoritmo cuando la entrada se vuelve muy grande.

La notación Big O se centra en el peor caso, lo que proporciona una garantía sobre el comportamiento del algoritmo independientemente de las condiciones específicas de la entrada

2.4. ALGORITMO DE EUCLIDES

El algoritmo de Euclides es el algoritmo eficiente más antiguo que se tiene, se lo usa desde hace 2500 años, descrito por primera vez por Euclides en su obra Elementos.

Para computar $gcd(m, n)$, con $m \geq n > 0$, el algoritmo repetidamente remplaza el par (m, n) por el par $(n, m \mod n)$ hasta $n = 0$, en ese punto m es el máximo común divisor que se buscaba.

Algoritmo 1: Algoritmo de Euclides

Entrada: Enteros $m >= n > 0$;

while $n > 0$ **do**

$r \leftarrow m$ mod n;

$m \leftarrow n$;

$n \leftarrow r$;

end

Salida: $gcd(m, n) =$ el valor final de m.

2.5. ALGORITMO EXTENDIDO DE EUCLIDES

Si m y n son enteros y al menos uno de ellos no es 0, entonces existen enteros x y y tales que $m \cdot x + n \cdot x = gcd(m, n)$ Esto es importante para calcular el inverso modular m y para calcular estos valores x, y se puede usar el algoritmo extendido de Euclides. Para el siguiente algoritmo se usara tripletas de enteros, como $u = (u_0, u_1, u_2)$, las cuales son sumadas y multiplicadas por enteros usando las reglas de suma vectorial y multiplicación escalar.

Algoritmo 2: Algoritmo extendido de Euclides

Entrada: Enteros $m >= n > 0$;

$\vec{u} \leftarrow (m, 1, 0)$;

$\vec{v} \leftarrow (n, 0, 1)$;

while $v_0 > 0$ **do**

  $q \leftarrow \lfloor u_0/v_0 \rfloor$;

  $\vec{w} \leftarrow \vec{u} - q\vec{v}$;

  $\vec{u} \leftarrow \vec{v}$;

  $\vec{v} \leftarrow \vec{w}$;

end

Salida: $(gcd(m, n), x, y) =$ el valor final de $\vec{u}$.

Los números x y y no son únicos, pero el algoritmo retorna los x y y con valor absoluto más pequeños. El algoritmo funciona porque cada tripleta (a, b, c) satisface $a = b \cdot m + c \cdot n$ durante el algoritmo.

2.6. NÚMEROS PRIMOS

Un entero $p > 1$ es llamado número primo, o simplemente primo, si no existe un divisor d de p que satisfaga $1 < d < p$. Si un entero $a > 1$ no es un primo, es llamado un número compuesto.

2.7. TEOREMA FUNDAMENTAL DE LA ARITMÉTICA

Todos los enteros mayores a 1 se pueden escribir como un producto de primos, este producto es único aparte del orden de los factores primos, y a este producto se llamara la factorización de dicho numero.

$$n = p_1 \times p_2 \times p_3 \times p_4$$

Si se tiene e copias de un primo p_i en el producto, este se puede reescribir como p_i^e, de esta forma se obtiene la forma canónica de la factorización de un entero n como:

$$n = \prod_{i=1}^{k} p_i^{e_i}$$

donde $p_1, p_2, \ldots, p_k$ son los distintos primos que dividen a n y $e_i \geq 1$ es el número de veces que p_i divide a n. Si n es primo, solo existe un "factor", el mismo primo. También esta permitido $n = 1$ con el "producto vació" para esta forma canónica.

2.8. CONGRUENCIA

Si m es un entero positivo y, a y b son enteros, se puede decir a que a es congruente con b módulo m y escribir $a = b(\mod m)$ si m divide a $a - b$. Si $m \nmid (a - b)$ se escribe $a \neq b(\mod m)$. El entero m es llamado el módulo. Cuando $a = b(\mod m)$, cada uno a y b son llamados residuos del otro (modulo m). Congruencia modulo m es una relación de equivalencia, lo que significa que si a, b y c son enteros, entonces:

- $a = a \mod m$
- Si $a = b \mod m$, entonces $b = a \mod m$
- Si $a = b \mod m$ y $b = c \mod m$ entonces $a = c \mod m$

La congruencia $a = b \mod m$ es equivalente a decir que existe un entero k tal que $a = b + k \cdot m$.

2.9. DIVISIONES SUCESIVAS

Si se quiere encontrar los factores primos de n lo que se hace es tener una lista de todos los números primos menores a n, luego se prueba dividir entre cada primo p_i, si $p_i \mid n$ entonces se vuelve a hacer el proceso desde ese primo, pero ahora con n/p. Si se llega a $\sqrt{n}$ y no se ha encontrado ningún primo que divida a n, entonces se declara a n como primo. (Knuth & Pardo, 1975)

2.10. MÉTODO DE DIFERENCIA DE CUADRADOS DE FERMAT

Para factorizar un número impar n, Fermat trato de expresar n como una diferencia de dos cuadrados, $x^2 - y^2$ con el par x, y diferente de $\frac{n+1}{2}$, $\frac{n-1}{2}$. Este par entrega $x + y = n$ y $x - y = 1$. Cualquier otra representación de n como $x^2 - y^2$ entrega una factorización no trivial $n = (x - y) \cdot (x + y)$. (Fermat, 1894)

Algoritmo 3: Método de diferencia de cuadrados de Fermat

Entrada: Enteros un entero compuesto impar n ;

$x \leftarrow \sqrt{n}$;

$t \leftarrow 2 \cdot x + 1$;

$r \leftarrow x^2 - n$;

while r *no sea una raiz cuadrada* **do**

$r \leftarrow r + t$;

$t \leftarrow t + 2$;

end

$x \leftarrow \frac{(t-1)}{2}$;

$y \leftarrow \sqrt{r}$;

Salida: Los factores $x - y$ y $x + y$ de n.

2.11. ALGORITMO DE FACTORIZACIÓN EN UNA LÍNEA DE HART

Hart en 2012 invento una variación del Método de Factorización de Fermat, que es mucha más corto, simple de programar. El da un argumento heurístico de que factoriza en $O(n^{\frac{1}{3+\varepsilon}})$ pasos.

El algoritmo de Hart comienza verificando si n es una raíz. Si n no es una raíz, entonces hace divisiones sucesivas, pero se detiene cuando p alcanza $n^{\frac{1}{3}}$. En caso que n no ha sido factorizado todavía, realiza los siguientes pasos. Para $i = 1, 2, 3, \ldots$ prueba cualquier $\lceil\sqrt{n_i}\rceil^2 \mod n$ si es raíz. Si este número es igual a t^2 entonces, es un factor de n. (Hart, 2012)

Algoritmo 4: Algoritmo de factorización en una línea de Hart

Entrada: Un entero positivo n y un limite L ;

$x \leftarrow \sqrt{n}$;

$t \leftarrow 2 \cdot x + 1$;

$r \leftarrow x^2 - n$;

for $i = 1$ *hasta* L **do**

  $s \leftarrow \lceil n_i \rceil$;

  $m \leftarrow s^2 \mod n$;

  if m *es raiz* **then**

    $t \leftarrow \sqrt{m}$;

    break

  end

end

Salida: $gcd(s - t, n)$ es un factor de n.

Este algoritmo es especialmente rápido para enteros de la forma $(c^a + d) \cdot (c^b + e)$ donde c, $|d|$, $|e|$ y $|a - b|$ son enteros positivos pequeños.

2.12. VARIACIÓN DE FERMAT DE LEHMERS

En 1985, Lawrence propuso una manera de factorizar n cuando se cree que $n = pq$ con $p \leq q$, donde la proporción p/q es aproximadamente a/b y a y b son coprimos pequeños. Cuando $a = b = 1$, este algoritmo es lo mismo que el de Fermat. Asumiendo que $gcd(a \cdot b, n) = 1$.

Suponemos primero que ambos a y b son impares. Escriba $x = \lceil \sqrt{a \cdot b \cdot n} \rceil$. Se prueban los enteros $(x + i)^2 - a \cdot b \cdot n$, $i = 0, 1, 2, \ldots$ si son una raíz, al igual que en el algoritmo de Fermat. Se supone que j es el primer valor de i para el cual este número es una raíz,

entonces $(x+j)^2 - a \cdot b \cdot n = y^2$. Entonces:

$$a \cdot b \cdot n = (x+j)^2 - y^2 = (x+j+y) \cdot (x+j-y).$$

Se remueve los factores de $a \cdot b$ de los dos factores del trinomio para obtener los factores de n. Esto es, $gcd(x+j+y, n)$ y $gcd(x+j-y, n)$ serán los factores de n.

Cuando una de los dos a o b es par y el otro es impar, los cálculos son un poco más complicados porque se debe lidiar con mitades. Lehman evito este problema multiplicando a, b y los otros números en el algoritmo por 2. (Lawrence, 1895)

2.13. MÉTODO DE POLLARD RHO

Este algoritmo de factorización fue inventado por John Pollard el año 1975. Este no utiliza mucho espacio de memoria, y el tiempo esperado de ejecución es proporcional a la raíz del factor primo más pequeño del número compuesto a ser factorizado. Está basada en la combinación de 2 ideas, que también son útiles para muchos otros métodos de factorización. La primera idea es la bien conocida Paradoja del cumpleaños: un grupo de al menos 23 personas seleccionadas aleatoriamente contiene 2 personas con el mismo cumpleaños en más del $50\,\%$ de los casos. Más generalmente: si los números son elegidos de manera aleatoria en un conjunto de p números, la probabilidad de elegir el mismo número dos veces excede el $50\,\%$ después de $1{,}177 \cdot \sqrt{p}$ números elegidos. El primer duplicado se espera que aparezca después de que $c \cdot \sqrt{p}$ números hayan sido seleccionados, para alguna pequeña constante c.

La segunda idea es la siguiente: si p es algún divisor desconocido de n y las variables x, y son 2 enteros que se piensa son idénticas modulo p, en otras palabras $x \cong y \mod p$, entonces este puede ser verificado calculando $gcd(x-y, n)$; más importante, este cálculo puede revelar una factorización de n, a menos que x, y también sean idénticos modulo n.

Estas ideas pueden ser combinadas en un algoritmo de factorización de la siguiente manera.

Generar una secuencia en $0, 1, 2, \ldots, n-1$ seleccionando x_0 y definiendo a x_{i+1} como el resto no negativo más pequeño de $x_i^2 + 1 \mod n$, ya que p divide a n los restos no negativos más pequeños $x_i \mod p$ y $x - j \mod p$ son iguales si y solo si x_i y x_j son idénticos modulo p, ya que $x \mod p$ se comporta más o menos como un entero aleatorio en $0, 1, 2, \ldots, p-1$ podemos esperar factorizar n calculando $gcd(|x_i - x_j|, n)$ para $i \neq j$ después de que al menos $c \cdot \sqrt{p}$ elementos de la secuencia han sido calculados. (Pollard, 1975)

Algoritmo 5: Método de factorización de Pollard Rho

Entrada: Un entero positivo n;

$b \leftarrow$ un número aleatorio b en $1 \leq b \leq n-3$;

$s \leftarrow$ un número aleatorio s en $0 \leq s \leq n-1$;

$A \leftarrow s$;

$B \leftarrow s$;

Defina una función $f(x) \leftarrow (x^2 + b) \mod n$;

$g \leftarrow 1$;

while $g = 1$ **do**

    $A \leftarrow f(A)$;

    $B \leftarrow f(f(B))$;

    $g \leftarrow gcd(A - B, n)$;

end

if $g < n$ **then**

    g es un factor propio de n;

else

    Rendirse o intentar con nuevos valores para s y/o b;

Salida: Un factor propio g de N.

El factor g de n no esta garantizado que sea primo. Uno siempre debería probar la primalidad de g cuando el algoritmo finalice.

Como se puede notar, asumiendo que f es una función aleatoria, la complejidad del Método de Pollard Rho es $O(\sqrt{p})$ pasos, donde p es el factor más pequeño de n. Dado que $p \leq n$, la complejidad sera $O(n^{1/4})$

2.14. MÉTODO p-1 DE POLLARD

El método $p-1$ esta basado en el pequeño teorema de Fermat, que dice que $a^{p-1} \equiv 1(\mod p)$ cuando p es un primo que no divide a a. Por lo tanto, $a^L \equiv 1(\mod p)$ para cualquier múltiplo L de $p-1$. Si tambien $p|n$, entonces p divide a $gcd(a^L-1, n)$, Obviamente no podemos calcular $a^L \mod p$ porque p es un factor primo desconocido de n. De todas maneras, podemos computar $a^L \mod n$. La idea de Pollard es permitir que L tenga muchos divisores de la forma $p-1$ y asi probar muchos potenciales factores primos p de n de una sola vez.

Si $p-1$ esta acotado en B, esto es, que el factor primo $p-1$ más grande de $p-1$ es $\leq B$, entonces $p-1$ dividira a L si L es el producto de todos los primos $\leq B$, cada uno repetido un número apropiado de veces. Si un primo $q \leq B$ divide a $p-1$, entonces q no puede dividir a $p-1$ mas de $\log_q p - 1 = (\log p / \log q) - 1$ veces. Este numero es un limite superior en el "numero apropiado de veces" que q divide a L. De todas formas, primes muy grandes raramente dividen grandes numeros enteros aleatorios más de una vez. Un compromiso razonable para L es elegir un limite B, el cual nos diga cuanto trabajo se esta dispuesto a realizar en un esfuerzo de factorizar n, y definer L como el múltiplo común más pequeño de los enteros positivos hasta B, Se puede mostar que $L = \prod q^e$, donde q recorre todos los primos que son $\leq B$. Tupicamente, B esta en los millones y L is enorme. No hay necesidad de computar L. Dado que q^e es formado, uno computa $a = a^{q^e} \mod n$. (Pollard, 1974)

Algoritmo 6: Método de factorización $p-1$ de Pollard

Entrada: Un entero positivo n y un limite B;

Encontrar todos los primos $p_1 = 2, p_2 = 3, p_3, \ldots, p_k \leq B$;

$a \leftarrow 2$;

for $i \leftarrow 1$ *hasta* k **do**

  $e \leftarrow \lfloor (\log B / \log p_i) \rfloor$;

  $f \leftarrow p_i^e$;

  $a \leftarrow a^f \mod n$;

end

$g \leftarrow gcd(a-1, N)$;

if $1 < g < n$ **then**

  g divide a n;

else

  Rendirse;

Salida: Un factor propio g de N.

El algoritmo tiene una segunda parte in la cual uno elige un segundo limite $B_2 > B$ and busca un factor p de N para el cual el factor primo más grande de $p-1$ es $\leq B_2$ y el segundo factor primo mas grade es $\leq B$. Al final de la primera parte, algoritmo x, a tiene el valor $2^L \pmod n$. Sean $q_1 < q_2 < \ldots < q_t$ los primos entre B y B_2. La idea es computar sucesivamente $w^{L \cdot q_i} \pmod n$ y luego $gcd(2^{L \cdot q_i} - 1, n)$ para $1 \leq i \leq k$. La primera potencia $2^{L \cdot q_i} \pmod n$ es computada directamente como $a^{q_i} \pmod n$. Las diferencias $q_{i+1} - q_i$ son números pares y mucho mas pequeños que q_i por si mismos. Podemos precomputar $2^{L \cdot q_{i+1}} \pmod n$ para $d = 2, 4, \ldots$ hasta unos cuantos cientos. Para encontrar $2^{L \cdot q_{i+1}} \pmod n$ desde $2^{L \cdot q_i} \pmod n$, multiplicamos el último por $2^{L \cdot d} \pmod n$, donde $d = q_{i+1} - q_i$. El costo amortizado de computar $2^{L \cdot q_i} \pmod n$ para $2 \leq i \leq k$ es una simple multiplicación modulo n.

2.15. FRACCIONES CONTINUAS

Una fracción continua simple es una expresión de la forma:

$$x = q_0 + \cfrac{1}{q_1 + \cfrac{1}{q_2 + \cfrac{1}{q_3 + \cdots}}}$$

donde denotamos por $x = [q_0; q_1; q_2, q_3, \dots]$. Los números q_i deben ser enteros para todos los i y también positivos cuando $i > 0$. Una fracción continua simple puede ser finita.

$$x = q_0 + \cfrac{1}{q_1 + \cfrac{1}{q_2 + \cfrac{1}{q_3 + \cdots + \cfrac{1}{q_k}}}}$$

que se escribe como $[q_0, q_1, q_2, q_3, \dots, q_k]$. Los números $q_1, q_2, \dots$ son llamados los cocientes parciales de cada fracción continua.

Todos los números reales x tienen una expansión continua simple que puede ser computada por el siguiente algoritmo. Separamos x entre su parte entera q_0 y su parte fraccional, el nuevo valor de x. El bucle principal alterna recíprocamente con esta operación de separación, formando la secuencia de cocientes parciales q_i de x.

Algoritmo 7: Algoritmo de fracciones continuas

Entrada: Un número real x;

$i \leftarrow 0$;

$q_0 \leftarrow \lfloor x \rfloor$;

$x \leftarrow x - q_0$;

while $x > 0$ **do**

  $i \leftarrow i + 1$;

  $q_o \leftarrow \lfloor 1/x \rfloor$;

  $x \leftarrow x - q_i$;

end

Salida: $[q_0, q_1, q_2, \dots]$ es la fracción continua para x.

Computar una fracción continua de esta manera a través de aritmética de punto flotante requiere una gran precisión para encontrar más de unas cuantas primeras q_i.

Se puede demostrar que este algoritmo termina en un número finito de pasos si y solo si x es un número racional.

2.16. ALGORITMO DE FACTORIZACIÓN DE FRACCIONES CONTINUAS

Este algoritmo fue el primer algoritmo de factorización de números con una complejidad subexponencial.

Morrison y Brillhart repitieron el algoritmo de Lehmer y Powers y lo programaron en una computadora. La computadora hizo el tedioso cálculo de las secuencias de fracciones continuas y factorizó las Q. Pero, ¿cómo ”inspeccionó” la computadora la Q factorizada para formar un cuadrado? La brillante idea de Morrison y Brillhart fue utilizar la eliminación gaussiana en vectores de exponentes módulo 2, una tarea fácil de programar, para formar cuadrados. (Morrison & Brillhar, 1975)

Al igual que el algoritmo de Lehmer y Powers, el algoritmo de factorización de fraccio-

nes continuas (CFRAC) de Morrison y Brillhart usa el hecho de que, dado que los Q_i son pequeños (cerca de $\sqrt{N}$), es más probable que sean más suaves que los números cercanos a $N/2$. El algoritmo usa la expansión de fracción continua para $\sqrt{N}$ para generar las secuencias Q_i y $A_i \mod N$ e intenta factorizar cada Q_i por Divisiones sucesivas. Una innovación de Morrison y Brillhart fue restringir los primos en la Divisiones sucesivas a aquellos por debajo de un límite B, llamado la base de factores. Es decir, si un Q_i tiene factores primos mayores que B, entonces ese Q_i se descarta.

El CFRAC guarda el Q_i, acotado en B, junto con el $A_i - 1$ correspondiente, lo que representa la relación $A_{i-1}^2 \equiv (-1)^i \cdot Q_i (\mod N)$. Cuando se han recopilado suficientes relaciones, se utiliza la eliminación gaussiana para encontrar dependencias lineales (módulo 2) entre los vectores exponenciales de las relaciones. Hay suficientes relaciones cuando hay más de ellas que primos en la base de factores. Cada dependencia lineal produce una congruencia $x^2 \equiv y^2 (\mod N)$ y una posibilidad de factorizar N.

Hay una segunda restricción más allá de $p \leq B$ en los primos en la base de factores. Supongamos que el primo p divide a Q_i. La ecuación $A_{i-1}^2 - N \cdot B_{i-1}^2 = (-1)^i \cdot Q_i$ muestra que $(A_{i-1}/B_{i-1})^2 \equiv N (\mod p)$, por lo que N es un residuo cuadrático módulo p. Por lo tanto, la base de factores debería contener solo primos p para los cuales N es un residuo cuadrático, es decir, aproximadamente la mitad de de los primos hasta B.

Algoritmo 8: Algoritmo de factorización de enteros de fracciones continuas

Entrada: Un número entero $n > 1$ para factorizar;

Se elige una cota superior B para la base de factores ;

$p_0 \leftarrow -1$;

Sean $p_1, \ldots, p_k$ los primos $\leq B$ con $(N/p_i) = +1$;

$r \leftarrow 0$;

$i \leftarrow 0$;

while $R < K + 10$ **do**

  if $i = 0$ **then**

    $P_i \leftarrow 0$;

    $Q_i \leftarrow 1$;

    $q_i \leftarrow \lfloor \sqrt{N} \rfloor$;

  else if $i = 1$ **then**

    $P_i \leftarrow q_0$;

    $Q_i \leftarrow N - q_0^2$;

  else if $i \geq 2$ **then**

    $P_i \leftarrow q_{i-1} \cdot Q_{i-1} - P_{i-1}$;

    $Q_i \leftarrow Q_{i-2} + (P_{i-1} - P_i) \cdot q_{i-1}$;

  if $i > 0$ **then**

    $q_i \leftarrow \lfloor \frac{\sqrt{N}+P_i}{Q_i} \rfloor$;

  end

  Se intenta factorizar Q_i usando solo los primos en la base de factores;

  Si se tiene exito, guardar i, Q_i y añadir 1 a R;

  $i \leftarrow i + 1$;

end

Salida: $[q_0, q_1, q_2, \ldots]$ es la fracción continua para x.

2.17. FACTORIZACIÓN CON CURVAS ELÍPTICAS

En 1985, H. W. Lenstra, Jr. inventó un algoritmo de factorización que utiliza curvas elípticas. Se llama el método de curva elíptica o ECM. Sea R_p el grupo multiplicativo de números enteros módulo un primo p. Consiste en los números enteros $1, 2, \ldots, p-1$; la operación de grupo es la multiplicación módulo p. Recordemos que el método de factorización $p-1$ de Pollard, realiza un cálculo ($a^L \mod N$, donde L es el producto de las potencias de los primos por debajo de algún límite B) en los números enteros módulo N que oculta un cálculo ($a^L \mod p$) en R_p.

El factor p de N se descubre cuando el tamaño $p-1$ del grupo R_p divide a L. El método $p-1$ para encontrar p cuando $p-1$ tiene un divisor primo mayor que B. Lenstra reemplazó R_p con un grupo de curvas elípticas $E_{a,b}$ módulo p. Por el teorema de Hasse, los dos grupos tienen aproximadamente el mismo tamaño, es decir, aproximadamente p. El algoritmo de Lenstra descubre p cuando $M_{p,a,b}$ divide a L. No logra encontrar p cuando $M_{p,a,b}$ tiene un factor primo mayor que B. Solo hay un grupo R_p, pero muchos grupos de curvas elípticas $E_{a,b}$ módulo p. Si el tamaño de R_p tiene un factor primo $> B$, estamos atascados. Pero si el tamaño de $E_{a,b}$ módulo p tiene un factor primo $> B$, simplemente cambiamos a y b y probamos con otra curva elíptica. Cada curva da una probabilidad independiente de encontrar el factor primo p. (Lenstra, 1987)

Si comparamos con el algoritmo con el método $p-1$ de Pollard. Los dos algoritmos funcionan exactamente de la misma manera, excepto que el método $p-1$ de Pollard eleva a a la potencia p_i^e, mientras que el método de la curva elíptica multiplica P por p_i^e. El primer algoritmo calcula explícitamente un máximo común divisor con N, mientras que el segundo algoritmo oculta esta operación en el cálculo de la pendiente de la suma de puntos de la curva elíptica.

Algoritmo 9: Algoritmo simple de factorización con curvas elípticas

Entrada: Un número entero n para factorizar y una cota B;

Encontrar los primos $p_1 = 2, p_2 = 3 \ldots, p_k \leq B$;

Elegir una curva elíptica aleatoria $E_{a,b} \mod N$ y un punto aleatorio $P! = \inf$ en la curva;

$g \leftarrow gcd(4a^3 + 27b^2, N)$;

if $g = N$ **then**

  Elegir una nueva curva y un nuevo punto;

end

if $g > 1$ **then**

  g es un factor de N;

end

for $i \leftarrow 1$ *hasta* k **do**

  $e \leftarrow \lfloor (\log B)/\log p_i \rfloor$;

  $P \leftarrow (p_i^e)P$ o bien encuentra un factor g de N;

end

Rendirse o intentar con otra curva elíptica aleatoria;

Salida: Un factor propio p de N

2.18. MÉTODOS DE CRIBADO

2.18.1. CRIBA DE ERATOSTENES

La criba de Eratóstenes encuentra los primos menores que un límite J. Comienza escribiendo los números $1, 2, \ldots, J$. Tacha el número 1, que no es primo. Después, sea p el primer número no tachado. Entonces p es primo, así que no se lo tacha, pero tacha cada p-esimo número (incluidos los que ya están tachados) comenzando con $2p$. Es decir, tacha todos los múltiplos de p mayores que p. Repite esta operación, reemplazando p por el

siguiente número que aún no se haya tachado, siempre que $p \leq \sqrt{J}$. Todos los números tachados son compuestos (o 1) y todos los números no tachados son primos. Este algoritmo funciona porque todo número compuesto $\leq J$ tiene un factor primo $p \leq \sqrt{J}$ y, por lo tanto, sería tachado como múltiplo de p. (de Cirene, 200 a.C.)

Algoritmo 10: Criba de Eratóstenes

Entrada: Un número entero $J > 1$;

$P[1] \leftarrow 0$;

for $i \leftarrow 2$ *hasta* J **do**

  $P[i] \leftarrow 1$;

end

while $p \leq \sqrt{J}$ **do**

  $i \leftarrow p + p$;

  while $i \leq J$ **do**

    $P[i] \leftarrow 0$;

    $i \leftarrow i + p$;

  end

  $i \leftarrow p + 1$;

  while $i \leq \sqrt{J}$ *y* $P[i] = 0$ **do**

    $i \leftarrow i + 1$;

  end

  $p \leftarrow i$;

end

Salida: El arreglo $P[]$ con la lista de primos $\leq J$.

Cuando el algoritmo termina, el valor de $P[i]$ es 1 si i es primo y 0 si i es 1 o compuesto.

La siguiente variación busca todos los números enteros en un intervalo no divisible por ningún primo en un conjunto finito de primos. Primero escribe los números en el intervalo. Luego, para cada primo p en el conjunto, tacha todos los múltiplos de p en el intervalo. El

conjunto de números que no está tachado es la respuesta.

Algoritmo 11: Criba de Eratóstenes Modificada

Entrada: Enteros $J > I > 1$ y un conjunto finito de primos P;

for $i \leftarrow I$ *hasta* J **do**

  $A[i] \leftarrow 1$;

end

foreach $p \in P$ **do**

  $i \leftarrow$ el multiplo más pequeño de p que sea $\geq I$;

  while $i \leq J$ **do**

    $A[i] \leftarrow 0$;

    $i \leftarrow i + p$;

  end

end

Salida: El arreglo $A[]$ con la lista de números entre I y J libres de factores de P.

Cuando el algoritmo termina, $A[i] = 0$ si algún primo $p \in P$ divide a i y $A[i] = 1$ si ningún primo en P divide a i.

La siguiente variación factoriza los números enteros entre I y J. Cada número entero i en este intervalo está representado por una lista $L[i]$, inicialmente vacía, que contendrá los factores primos de i cuando el algoritmo termine. El algoritmo primero encuentra los factores primos $\leq \sqrt{J}$ de cada i con una criba. El primer bucle "while" coloca un p en $L[i]$ siempre que $p|i$. Luego, el segundo bucle "while" coloca un p más en $L[i]$ siempre que $p^2|i$, un p más en $L[i]$ siempre que $p^3|i$, etc., hasta que se hayan añadido un total de j $p's$ a $L[i]$ si $p^j|i$. Luego, un segundo bucle "for" divide i por cada factor primo encontrado por la criba. Si el cofactor restante excede 1, entonces este cofactor es un último factor primo de i, por lo que se agrega a la lista.

Algoritmo 12: Factorización con la Criba de Eratostenes

Entrada: Enteros $J > I > 1$;

for $i \leftarrow I$ *hasta* J **do**

  $L[i] \leftarrow []$;

end

foreach *primo* $p \leq \sqrt{J}$ **do**

  $i \leftarrow$ el múltiplo más pequeño de p con $i \geq I$;

  while $i \leq J$ **do**

    Añadimos p a $L[i]$;

    $i \leftarrow i + p$;

  end

  $a \leftarrow 2$;

  while $p^a \leq \sqrt{J}$ **do**

    $i \leftarrow$ el múltiplo más pequeño de p^a con $i \geq I$;

    while $i \leq J$ **do**

      Añadimos p a $L[i]$;

      $i \leftarrow i + p^a$;

    end

    $a \leftarrow a + 1$;

  end

end

for $i \leftarrow I$ *hasta* J **do**

  $j \leftarrow i$;

  foreach *primo* p *en* $L[i]$ **do**

    $j \leftarrow j/p$;

  end

  if $j > 1$ **then**

    Añadimos j a $L[i]$;

  end

end

Salida: Para $I \leq i \leq J$, $L[i]$ tiene los factores de i.

La última variación factoriza los números en el rango de un polinomio $f(x)$ con coeficientes enteros, pero solo encuentra los factores primos de cada $f(x)$ que se encuentran en un conjunto finito P de primos. El polinomio $f(x)$ se supone fijo y no es parte de la entrada. Este algoritmo de criba es el corazón de los algoritmos de factorización de criba de cuerpos numéricos y cuadráticos. Este algoritmo funciona igual que el anterior, excepto que $L[i]$ contiene los factores primos de $f(i)$ en lugar de los de i. El número de raíces de $f(x) \equiv 0(\mod p^a)$ no es mayor que el grado de f.

Algoritmo 13: Criba para Factorizar el rango de un polinomio

Entrada: Enteros $J > I > 1$ y un conjunto finito P de primos;

for $i \leftarrow I$ *hasta* J **do**

  $L[i] \leftarrow []$;

end

foreach $p \in P$ **do**

  Encontrar las raices $r_1, \ldots, r_d$ de $f(x) \equiv 0(\mod p)$;

  for $j \leftarrow 1$ *hasta* d **do**

    $i \leftarrow$ el menor entero $\geq I$ y $\equiv r_j(\mod p)$;

    while $i \leq J$ **do**

      Añadimos p a $L[i]$;

      $i \leftarrow i + p$;

    end

    $a \leftarrow 2$;

    while $p^a \leq \sqrt{J}$ **do**

      Elevar r_j a una raíz r de $f(x) \equiv 0(\mod p^a)$;

      $i \leftarrow$ el menor entero $\geq I$ y $\equiv r(\mod p^a)$;

      while $i \leq J$ **do**

        Añadimos p a $L[i]$;

        $i \leftarrow i + p^a$;

      end

      $a \leftarrow a + 1$;

    end

  end

end

Salida: Para $I \leq i \leq J$, $L[i]$ tiene los factores en P de $f(i)$.

2.18.2. CRIBA CUADRÁTICA

El algoritmo de factorización de criba cuadrática es similar al algoritmo de factorización de fracciones continuas. La diferencia radica en el método de producir relaciones $x^2 \equiv q(\mod N)$ con q factorizada completamente. El CFRAC forma x y q a partir de la expansión fraccionaria continua de $\sqrt{N}$ y factoriza q por divisiones sucesivas, que es lenta. Es probable que los residuos cuadráticos q en el CFRAC sean suaves porque son $< 2\sqrt{N}$. El algoritmo de factorización de criba cuadrática (QS) fue inventado por Pomerance, pero muchas de sus ideas se remontan a Kraitchik. Produce x y q utilizando un polinomio cuadrático $q = f(x)$ y factoriza q con una criba, un proceso mucho más rápido que las divisiones sucesivas. El polinomio cuadrático $f(x)$ se elige de modo que q sea lo más pequeño posible. Esto significa que la mayoría de ellos superarán $2\sqrt{N}$, pero no por un factor grande, de modo que es casi tan probable que sean suaves como q en el CFRAC.

Sean $f(x) = x^2 - N$ y $s = \lceil \sqrt{N} \rceil$ Consideremos los números $f(s), f(s+1), f(s+2), \ldots$

La criba cuadrática factoriza algunos de estos números mediante la criba para factorizar el rango de un polinomio. Si hay K primos $p \leq B$ y podemos encontrar números $R > K$ acotados en B $f(x)$, entonces tendremos R relaciones que involucran K primos y el álgebra lineal nos dará al menos $R - K$ congruencias $x^2 \equiv y^2(\mod N)$, cada una de las cuales tiene una probabilidad de al menos $1/2$ de factorizar N.

Cribamos utilizando la criba para factorizar el rango de un polinomio para encontrar los números acotados en B entre $f(s), f(s+1), f(s+2), \ldots$ La base factorial P consiste en los primos $p < B$ (para los cuales el símbolo de Legendre $(N/p) = -1$). Escribimos los números $f(s+i)$ para i en un intervalo $a \leq i < b$ de longitud conveniente, digamos unos pocos millones. El primer intervalo tendrá $a = s$. Los intervalos subsiguientes comenzarán con a igual al punto final b del intervalo anterior. Para cada primo $p < B$, eliminamos todos los factores de p de aquellas $f(s+i)$ que p divide. Como $f(x) = x^2 - N$, p divide a $f(x)$ precisamente cuando $x^2 \equiv N(\mod p)$. Las soluciones x para esta congruencia

se encuentran en la unión de dos progresiones aritméticas con diferencia común p. Si las raíces de $x^2 \equiv N(\mod p)$ son x_1 y x_2, entonces las progresiones aritméticas comienzan con los primeros números $\equiv x_1$ y $x_2(\mod p)$ que son $\geq a$. El factor primo p se elimina de cada $f(s+i)$ que divide.

El número de operaciones de criba para un número primo p es aproximadamente $\frac{2}{p}(b-a)$ porque exactamente dos de cada p números se dividen por p. La complejidad de la criba es $\sum_{p<B, p \text{ primo}} \frac{2}{p}(b-a)$. Se puede demostrar que esta suma es $O((b-a)\ln\ln B)$. El coste amortizado de cribar un valor i es entonces $\ln\ln B$. (Pomerance, 1982)

2.18.3. CRIBA DEL CUERPO DE NÚMEROS

Pollard fue el primero en sugerir elevar el grado del polinomio en la criba cuadrática de 2 a un valor superior, pero sólo para números con forma especial. Él factorizó el número de Fermat F_7 (que había sido factorizado anteriormente por Morrison y Brillhart) utilizando el polinomio cúbico $2x^3+2$ en una pequeña computadora. Manasse y los hermanos Lenstra pronto extendieron las ideas de Pollard a polinomios de grado superior, todavía sólo para números de la forma $r^e - s$. Su objetivo era factorizar F_9, el número de Fermat más pequeño sin factor primo conocido. Esperaban utilizar la forma especial de F_9 para hacer que los números que debían suavizarse fueran más pequeños que los necesarios para la criba cuadrática. Después de factorizar F_9 en 1990, ellos y otros extendieron la criba del cuerpo numérico a números generales.

Recordemos que la criba cuadrática produce muchas relaciones $x_i^2 \equiv q_i(\mod N)$ con q_i factorizado completamente. Cuando tenemos suficientes relaciones, hacemos coincidir los factores primos del q_i y creamos un subconjunto del q_i cuyo producto es cuadrado. De esta manera, encontramos congruencias $x^2 \equiv y^2(\mod N)$ que pueden factorizar N.

Busquemos relaciones $r_i \equiv q_i(\mod N)$ en las que tanto r_i como q_i hayan sido factorizadas completamente, como en las cribas dobles y lineales. Usaremos álgebra lineal para

hacer coincidir los factores primos de r_i y los factores primos de q_i y seleccionaremos un subconjunto de las relaciones para las que tanto el producto de r_i como el producto de q_i sean cuadrados. Esta es una buena idea, pero demasiado lenta para ser práctica. La principal dificultad es que al menos uno de $|r_i|$, $|q_i|$ debe ser mayor que $N/2$, por lo que tiene una baja probabilidad de ser suave. (Pomerance, 1994)

La Criba del cuerpo de números hace que la idea sea rápida y práctica al permitir que los números de un lado de cada relación sean enteros algebraicos de un campo numérico algebraico. La idea es hacer coincidir los factores irreducibles de modo que cada uno aparezca un número par de veces y el producto de los enteros algebraicos en el subconjunto seleccionado de las relaciones pueda ser un cuadrado en el campo numérico algebraico.

En resumen, la criba del cuerpo de números tiene los siguientes pasos:

1. Seleccionamos un polinomio $f(x) \in \zeta[x]$ y un entero m con $f(m) \equiv 0(\mod N)$.

2. Cribar los números $a - bm$ y $N(a - b\alpha)$; guardamos (a, b) cuando ambos $a - bm$ y $N(a - b\alpha)$ están acotados.

3. Filtramos la relación para remover duplicados y aquellos que contengan un primo que no aparezca en otra relación.

4. Usamos algebra lineal modulo 2 para encontrar los conjuntos S como en las formulas:

$$\prod_{(a,b)\in S} (a - bm) \text{ es un cuadrado en } \mathbb{Z}$$

$$\prod_{(a,b)\in S} (a - b\alpha) \text{ es un cuadrado en } \mathbb{Z}[\alpha]$$

5. Encontramos las raíces y y γ de los cuadrados de las formulas del paso anterior

6. Sea $x = h(\gamma)$; intentamos factorizar N a través de $gcd(N, x \pm y)$

CAPÍTULO 3 MARCO APLICATIVO

Para el presente capitulo es necesaria la lectura de los capítulos anteriores por los conceptos que se manejan los cuales son nombrados y fueron introducidos anteriormente.

3.1. ALGORITMOS DE FACTORIZACIÓN

Esta sección detalla la implementación de los algoritmos mostrados en el capitulo 2, se implementaran en Python sin el uso de librerías especializadas.

3.1.1. ALGORITMO DE EUCLIDES

El algoritmo de Euclides, lo usamos para encontrar el Maximo Comun Divisor (gcd) de dos números

3.1.1.1. IMPLEMENTACIÓN

```
1 def gcd(a, b):
2     while b != 0:
3         a, b = b, a % b
4     return a
```

3.1.1.2. EJEMPLO NÚMERICO

Encontrar el Máximo Común Divisor de 252 y 105

$$\begin{aligned} gcd(252, 105) &= gcd(105, 252\,\%105) \\ &= gcd(105, 42) \\ gcd(105, 42) &= gcd(42, 105\,\%42) \\ &= gcd(42, 21) \\ gcd(42, 21) &= gcd(21, 42\,\%21) \\ &= gcd(21, 0) \\ gcd(42, 21) = 21 &\Rightarrow gcd(252, 105) = 21 \end{aligned}$$

3.1.1.3. EJEMPLO DE EJECUCIÓN

```
(myenv) ┌─ rolando@rolando [09:06:40]
└─> $  python3 euclides.py
252 105
gcd(252, 105) = 21
```

Figura 1: Ejecución de prueba del algoritmo de Euclides

3.1.2. ALGORITMO EXTENDIDO DE EUCLIDES

El algoritmo extendido de Euclides, no solo encontrara los valores del Máximo Común Divisor (gcd) de dos números, sino también encontrara los valores de los coeficientes de Bézout, que son dos números tales que:

$$ax + by = gcd(a, b)$$

3.1.2.1. IMPLEMENTACIÓN

```
def gcd_extendido(a, b):
if a == 0:
    return b, 0, 1
else:
    gcd, x1, y1 = gcd_extendido(b % a, a)
    x = y1 - (b // a) * x1
    y = x1
    return gcd, x, y
```

3.1.2.2. EJEMPLO NÚMERICO

Aplicamos el algoritmo de Euclides:

$$252 = 2 \cdot 105 + 42$$

$$105 = 2 \cdot 42 + 21$$

$$42 = 2 \cdot 21 + 0$$

Despejamos los residuos en los pasos del algoritmo de Euclides:

$$42 = 252 - 2 \cdot 105$$

$$21 = 105 - 2 \cdot 42$$

Remplazamos 42 en la segunda ecuación:

$$21 = 105 - 2 \cdot (252 - 2 \cdot 105)$$

$$21 = 105 - 2 \cdot 252 + 4 \cdot 105$$

$$21 = 2 \cdot 252 + 5 \cdot 105$$

$$\Rightarrow x = -2, y = 5$$

3.1.2.3. EJEMPLO DE EJECUCIÓN

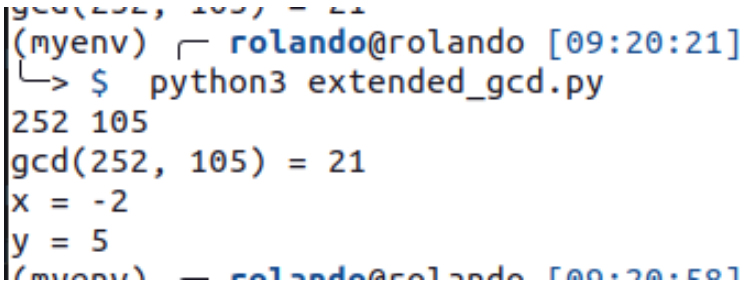

Figura 2: Ejecución de prueba del algoritmo extendido de Euclides

3.1.3. ALGORITMO DE EXPONENCIACIÓN MODULAR

Con este algoritmo encontraremos de manera eficiente $f(x, a, m) = x^a \,\%m$

3.1.3.1. IMPLEMENTACIÓN

```
1 def mod_exp(base, exponente, modulo):
2     resultado = 1
3     base = base % modulo
4 
5     while exponente > 0:
6         if (exponente % 2) == 1:
7             resultado = (resultado * base) % modulo
8         exponente = exponente >> 1
```

```
        base = (base * base) % modulo

    return resultado
```

3.1.3.2. EJEMPLO NÚMERICO

Encontrar el resultado para $25^9 \,\%7$

$$
\begin{aligned}
x &= 25^9 \,\%7 \\
x &= (25^4 \,\%7)^2 (\times 25 \cdot 1) \,\%7 \\
x &= ((25^2 \,\%7)^2)^2 \times (25 \,\%7) \\
x &= ((25 \cdot 25 \,\%7)^2)^2 \times (25 \,\%7) \\
x &= (((625 \,\%7)^2)^2 \times 4) \,\%7 \\
x &= (((2)^2)^2 \,\%7 \times 4) mod 7 \\
x &= ((4^2) \,\%7 \times 4) \,\%7 \\
x &= (16 \,\%7 \times 4) \,\%7 \\
x &= (2 \times 4) \,\%7 \\
x &= 8 \,\%7 \\
x &= 1
\end{aligned}
$$

3.1.3.3. EJEMPLO DE EJECUCIÓN

```
y = 5
(myenv) ┌─ rolando@rolando [09:20:58]
└─> $  python3 mod_exp.py
25 9 7
25^9 % 7 = 1
```

Figura 3: Ejecución de prueba del algoritmo de exponenciación modular

3.1.4. ALGORITMO DE DIVISIONES SUCESIVAS

3.1.4.1. IMPLEMENTACIÓN

```
def divisiones_sucesivas(n):
    factores = []
    while n % 2 == 0:
        factores.append(2)
        n //= 2
    for i in range(3, int(n**0.5) + 1, 2):
        while n % i == 0:
            factores.append(i)
            n //= i
    if n > 2:
        factores.append(n)

    return factores
```

3.1.4.2. EJEMPLO NÚMERICO

Encontrar los factores primos de 1759875, se prueba diviendo por todos los primos que sean p_i tal que $p_i \leq \sqrt{(n)}$ y sean divisores de n.

$$1759875/3 = 586625$$
$$586625/5 = 117325$$
$$117325/5 = 23465$$
$$23465/5 = 4693$$
$$4693/13 = 361$$
$$361/19 = 19$$
$$19/19 = 1$$
$$\Rightarrow factores(1759875) = [3, 5^3, 13, 19^2]$$

3.1.4.3. EJEMPLO DE EJECUCIÓN

```
(myenv) ┌─ rolando@rolando [10:08:33]
└─> $  python3 trial_division.py
1759875
1759875 = [3, 5, 5, 5, 13, 19, 19]
0.08893013000488281
```

Figura 4: Ejecución de prueba del algoritmo de factorización por divisiones sucesivas

3.1.5. MÉTODO DE DIFERENCIA DE CUADRADOS DE FERMAT

3.1.5.1. IMPLEMENTACIÓN

```
1 import math
2
3 def factorizacion_fermat(n):
```

```
    if n % 2 == 0:
        return (2, n // 2)

    a = math.ceil(math.sqrt(n))
    b2 = a * a - n

    while not es_cuadrado_perfecto(b2):
        a += 1
        b2 = a * a - n

    b = int(math.sqrt(b2))
    return (a - b, a + b)

def es_cuadrado_perfecto(x):
    s = int(math.isqrt(x))
    return s * s == x
```

3.1.5.2. EJEMPLO NÚMERICO

Encontrar los factores primos de 5959

$$a = \lceil \sqrt{5959} \rceil$$

$$a = 78$$

$$b^2 = a^2 - n$$

$$b^2 = 78^2 - 5959$$

$$b^2 = 6084 - 5959$$

$$b^2 = 125$$

Verificamos si b^2 es cuadrado perfecto:

$$\sqrt{125} \approx 11{,}18$$

Como no es cuadrado perfecto se incrementa a y se repite:

$$\begin{aligned} a &= 79 \\ b^2 &= 79^2 - 5959 \\ b^2 &= 282 \\ \sqrt{282} &\approx 16{,}79 \\ a &= 80 \\ b^2 &= 80^2 - 5959 \\ b^2 &= 441 \\ \sqrt{441} &= 21 \end{aligned}$$

Como 441 es cuadrado perfecto se han encontrado $a = 80$ y $b = 21$, y su factorización es:

$$\begin{aligned} n &= (a+b)(a-b) \\ n &= (80+21)(80-21) \\ x &= (101)(59) \end{aligned}$$

3.1.5.3. EJEMPLO DE EJECUCIÓN

```
(myenv) ┌─ rolando@rolando [10:32:33]
└─> $  python3 fermat_factor.py
5959
5959 = [59, 101]
```

Figura 5: Ejecución de prueba del algoritmo de Fermat

3.1.6. ALGORITMO DE FACTORIZACIÓN EN UNA LINEA DE HART

3.1.6.1. IMPLEMENTACIÓN

```
1  import math
2
3  def factorizacion_hart(n, l):
4      s = 1
5      t = 1
6      for i in range(1, l):
7          s = math.ceil(math.sqrt(n*i))
8          m = mod_exp(s, 2, n)
9          t = math.isqrt(m)
10         if t*t == m:
11             break
12     return gcd(s-t, n)
```

3.1.6.2. EJEMPLO NÚMERICO

Factorizar 13290059 con el algoritmo de factorización en una linea de Hart

$$
\begin{aligned}
s &= \lceil \sqrt{13290059 \cdot 1} \rceil \\
s &= 3646 \\
m &= 3646^2 \,\%13290059 \\
m &= 3257 \\
\sqrt{m} &\approx 57{,}07 \\
&\dots \\
s &= \lceil \sqrt{13290059 \cdot 165} \rceil \\
s &= 46828 \\
m &= 46828^2 \,\%13290059 \\
m &= 1849 \\
t &= \sqrt{m} \\
t &= \sqrt{1849} \\
t &= 43 \\
gcd(s-t, n) &= gcd(46828 - 43, 13290059) \\
&= 3119
\end{aligned}
$$

3.1.6.3. EJEMPLO DE EJECUCIÓN

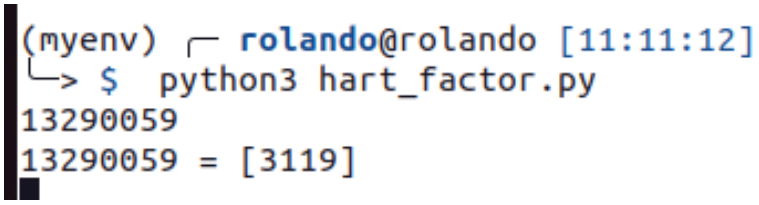

Figura 6: Ejecución de prueba del algoritmo de factorización en una linea de Hart

3.1.7. MÉTODO DE POLLARD RHO

3.1.7.1. IMPLEMENTACIÓN

```
def pollard_rho(n):
    if n % 2 == 0:
        return 2

    x = random.randint(2, n - 1)
    y = x
    c = random.randint(1, n - 1)
    d = 1

    while d == 1:
        x = (mod_exp(x, 2, n) + c + n) % n
        y = (mod_exp(y, 2, n) + c + n) % n
        y = (mod_exp(y, 2, n) + c + n) % n
        d = gcd(abs(x - y), n)

    if d == n:
        return pollard_rho(n)

    return d
```

3.1.7.2. EJEMPLO NÚMERICO

Factorizar 8051 usando el algoritmo Pollard-Rho

$$x = 2$$

$$y = 2$$

$$c = 1$$

$$d = 1$$

$$x = ((x^2)\,\%n + c + n)\,\%n$$

$$x_0 = (2^2\,\%8051 + 1 + 8051)\,\%8051$$

$$x_0 = 5$$

$$y_0 = (2^2\,\%8051 + 1 + 8051)\,\%8051$$

$$y_0 = 5$$

$$y_1 = (5^2\,\%8051 + 1 + 8051)\,\%8051$$

$$y_1 = 26$$

$$d = gcd(|x - y|, n)$$

$$d = gcd(|5 - 26|, 8051)$$

$$d = 1$$

$$x_1 = (5^2 \,\%8051 + 1 + 8051) \,\%8051$$

$$x_1 = 26$$

$$y_2 = (26^2 \,\%8051 + 1 + 8051) \,\%8051$$

$$y_2 = 677$$

$$y_3 = (677^2 \,\%8051 + 1 + 8051) \,\%8051$$

$$y_3 = 7474$$

$$d = gcd(|26 - 7474|, 8051)$$

$$d = 1$$

$$x_2 = (26^2 \,\%8051 + 1 + 8051) \,\%8051$$

$$x_2 = 677$$

$$y_2 = (7474^2 \,\%8051 + 1 + 8051) \,\%8051$$

$$y_2 = 2839$$

$$y_3 = (2839^2 \,\%8051 + 1 + 8051) \,\%8051$$

$$y_3 = 871$$

$$d = gcd(|677 - 871|, 8051)$$

$$d = 97$$

3.1.7.3. EJEMPLO DE EJECUCIÓN

```
(myenv) ┌─ rolando@rolando [11:43:40]
└─> $  python3 pollard_rho2.py
8051
8051 = [83, 97]
```

Figura 7: Ejecución de prueba del algoritmo Pollard-Rho

3.1.8. ALGORITMO DE FACTORIZACIÓN DE ENTEROS DE FRACCIONES CONTINUAS

3.1.8.1. IMPLEMENTACIÓN

```
import math

def fraccion_continua_sqrt(n):
    a0 = int(math.isqrt(n))
    if a0 * a0 == n:
        return [a0]

    cf = []
    m = 0
    d = 1
    a = a0
    cf.append(a)

    while a != 2 * a0:
        m = d * a - m
        d = (n - m * m) // d
        a = (a0 + m) // d
        cf.append(a)
    return cf

```

```python
def convergentes(cf):
    h1, h2 = 1, 0
    k1, k2 = 0, 1
    convergentes_list = []

    for i in range(len(cf)):
        h = cf[i] * h1 + h2
        k = cf[i] * k1 + k2
        convergentes_list.append((h, k))
        h2, h1 = h1, h
        k2, k1 = k1, k
    return convergentes_list

def factorizar_fracciones_continuas(n):
    cf = fraccion_continua_sqrt(n)
    convs = convergentes(cf)

    for h, k in convs:
        if k == 0:
            continue
        x = h
        y = (x * x - n) // k

        if y >= 0 and int(math.isqrt(y)) ** 2 == y:
            factor = gcd(x + int(math.isqrt(y)), n)
            if 1 < factor < n:
                return factor

    return None
```

3.1.8.2. EJEMPLO DE EJECUCIÓN

```
(myenv) ┌─ rolando@rolando [12:06:01]  [~/Documents/tesis/tesis_factorizacio
/src] [master ✗]
└─> $  python3 continued_fractions.py
8051
Terminos de la fraccion continua: [89, 1, 2, 1, 2, 89, 2, 1, 2, 1, 178]
Convergentes: [(89, 1), (90, 1), (269, 3), (359, 4), (987, 11), (88202, 983)
 (177391, 1977), (265593, 2960), (708577, 7897), (974170, 10857), (174110837
 1940443)]
8051 = [97]
```

Figura 8: Ejecución de prueba del algoritmo de factorización con fracciones continuas

3.1.9. ALGORITMO DE FACTORIZACIÓN CON CURVAS ELÍPTICAS

3.1.9.1. IMPLEMENTACIÓN

```
1  def inverso_modular(a, p):
2      g, x, _ = gcd_extendido(a, p)
3      return x % p
4
5  def adicion_curva_eliptica(P, Q, a, p):
6      if P == Q:
7          lam = (3 * P[0] * P[0] + a) * inverso_modular(2 * P[1], p)
       % p
8      else:
9          lam = (Q[1] - P[1]) * inverso_modular(Q[0] - P[0], p) % p
10
11     x3 = (lam * lam - P[0] - Q[0]) % p
12     y3 = (lam * (P[0] - x3) - P[1]) % p
13
14     return (x3, y3)
15
16 def multiplicacion_curva_eliptica(k, P, a, p):
17     R = (0, 0)
18     Q = P
19     while k > 0:
20         if k % 2 == 1:
```

```
            R = adicion_curva_eliptica(R, Q, a, p)
        Q = adicion_curva_eliptica(Q, Q, a, p)
        k //= 2
    return R

def ecm(n, B1=10000, B2=100000):
    while True:
        x = random.randint(1, n - 1)
        y = random.randint(1, n - 1)
        a = random.randint(1, n - 1)
        b = (y * y - x * x * x - a * x) % n

        if (4 * a * a * a + 27 * b * b) % n == 0:
            continue

        P = (x, y)
        for k in range(2, B1):
            P = multiplicacion_curva_eliptica(k, P, a, n)
            g = gcd(P[1], n)
            if 1 < g < n:
                return g


        for k in range(B1, B2):
            P = multiplicacion_curva_eliptica(k, P, a, n)
            g = gcd(P[1], n)
            if 1 < g < n:
                return g
```

3.1.9.2. EJEMPLO DE EJECUCIÓN

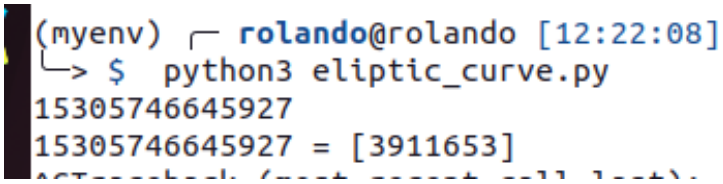

Figura 9: Ejecución de prueba del algoritmo de factorización con curvas Elipticas (ECM)

3.1.10. CRIBA DE ERATOSTENES

3.1.10.1. IMPLEMENTACIÓN

```
def sieve_of_eratosthenes(n):
    primes = [True] * (n + 1)
    p = 2
    while p * p <= n:
        if primes[p]:
            for i in range(p * p, n + 1, p):
                primes[i] = False
        p += 1
    return [p for p in range(2, n + 1) if primes[p]]
```

3.1.10.2. EJEMPLO NÚMERICO

En la Figura 11 se puede ver el resultado despues de ejecutar la criba de Eratostenes hasta 120, y a la derecha los primos encontrados

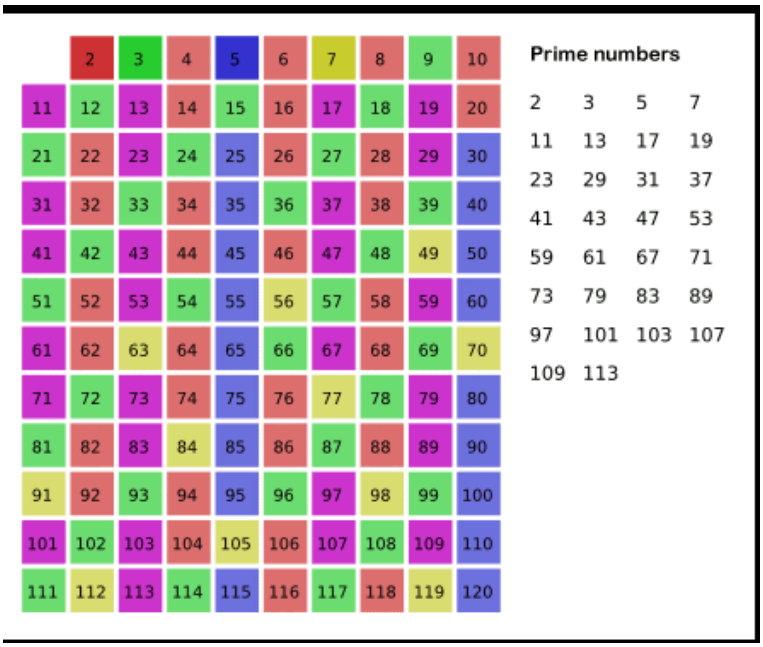

Figura 10: Resultado de la criba de Eratostenes

3.1.11. CRIBA CUADRATICA

3.1.11.1. IMPLEMENTACIÓN

```
def quad_residue(a,n):
    l=1
    q=(n-1)//2
    x = q**l
    if x==0:
        return 1

    a =a%n
    z=1
    while x!= 0:
        if x%2==0:
            a=(a **2) % n
            x//= 2
```

```
        else:
            x-=1
            z=(z*a) % n
    return z

def STonelli(n, p):
    q = p - 1
    s = 0
    while q % 2 == 0:
        q //= 2
        s += 1
    if s == 1:
        r = pow(n, (p + 1) // 4, p)
        return r,p-r
    for z in range(2, p):
        if p - 1 == quad_residue(z, p):
            break
    c = pow(z, q, p)
    r = pow(n, (q + 1) // 2, p)
    t = pow(n, q, p)
    m = s
    t2 = 0
    while (t - 1) % p != 0:
        t2 = (t * t) % p
        for i in range(1, m):
            if (t2 - 1) % p == 0:
                break
            t2 = (t2 * t2) % p
        b = pow(c, 1 << (m - i - 1), p)
        r = (r * b) % p
        c = (b * b) % p
        t = (t * c) % p
        m = i
    return (r,p-r)
```

```
48
49 def is_probable_prime(a):
50     if a == 2:
51         return True
52
53     if a == 1 or a % 2 == 0:
54         return False
55
56     return rabin_miller_primality_test(a, 50)
57
58 def rabin_miller_primality_test(a, iterations):
59     r, s = 0, a - 1
60
61     while s % 2 == 0:
62         r += 1
63         s //= 2
64
65     for _ in range(iterations):
66         n = randint(2, a - 1)
67         x = pow(n, s, a)
68         if x == 1 or x == a - 1:
69             continue
70         for _ in range(r - 1):
71             x = pow(x, 2, a)
72             if x == a - 1:
73                 break
74         else:
75             return False
76     return True
77
78 def prime_gen(n):
79     if n < 2:
80         return []
81
```

```
82      nums = []
83      isPrime = []
84
85      for i in range(0, n+1):
86          nums.append(i)
87          isPrime.append(True)
88
89      isPrime[0]=False
90      isPrime[1]=False
91
92      for j in range(2,int(n/2)):
93          if isPrime[j] == True:
94              for i in range(2*j,n+1,j):
95                  isPrime[i] = False
96
97      primes = []
98      for i in range(0, n+1):
99          if isPrime[i] == True:
100             primes.append(nums[i])
101
102     return primes
103
104 def isqrt(n):
105     x = n
106     y = (x + 1) // 2
107     while y < x:
108         x = y
109         y = (x + n // x) // 2
110     return x
111
112 def find_base(N,B):
113     factor_base = []
114     primes = prime_gen(B)
115     for p in primes:
```

```
116         if quad_residue(N,p) == 1:
117             factor_base.append(p)
118     return factor_base
119 
120 def find_smooth(factor_base,N,I):
121     def sieve_prep(N,sieve_int):
122         sieve_seq = [x**2 - N for x in range(root,root+sieve_int)]
123         return sieve_seq
124 
125     sieve_seq = sieve_prep(N,I)
126     sieve_list = sieve_seq.copy()
127     if factor_base[0] == 2:
128         i = 0
129         while sieve_list[i] % 2 != 0:
130             i += 1
131         for j in range(i,len(sieve_list),2):
132             while sieve_list[j] % 2 == 0:
133                 sieve_list[j] //= 2
134     for p in factor_base[1:]:
135         residues = STonelli(N,p)
136         for r in residues:
137             for i in range((r-root) % p, len(sieve_list), p):
138                 while sieve_list[i] % p == 0:
139                     sieve_list[i] //= p
140     xlist = []
141     smooth_nums = []
142     indices = []
143 
144     for i in range(len(sieve_list)):
145         if len(smooth_nums) >= len(factor_base)+T:
146             break
147         if sieve_list[i] == 1:
148             smooth_nums.append(sieve_seq[i])
149             xlist.append(i+root)
```

```
150             indices.append(i)
151     return(smooth_nums,xlist,indices)
152 
153 def build_matrix(smooth_nums,factor_base):
154     def factor(n,factor_base):
155         factors = []
156         if n < 0:
157             factors.append(-1)
158         for p in factor_base:
159             if p == -1:
160                 pass
161             else:
162                 while n % p == 0:
163                     factors.append(p)
164                     n //= p
165         return factors
166 
167     M = []
168     factor_base.insert(0,-1)
169     for n in smooth_nums:
170         exp_vector = [0]*(len(factor_base))
171         n_factors = factor(n,factor_base)
172         for i in range(len(factor_base)):
173             if factor_base[i] in n_factors:
174                 exp_vector[i] = (exp_vector[i] + n_factors.count(
    factor_base[i])) % 2
175         if 1 not in exp_vector:
176             return True, n
177         else:
178             pass
179 
180         M.append(exp_vector)
181     return(False, transpose(M))
182 
```

```
183 def transpose(matrix):
184     new_matrix = []
185     for i in range(len(matrix[0])):
186         new_row = []
187         for row in matrix:
188             new_row.append(row[i])
189         new_matrix.append(new_row)
190     return(new_matrix)
191 
192 def solve(solution_vec,smooth_nums,xlist,N):
193     solution_nums = [smooth_nums[i] for i in solution_vec]
194     x_nums = [xlist[i] for i in solution_vec]
195     Asquare = 1
196     for n in solution_nums:
197         Asquare *= n
198     b = 1
199     for n in x_nums:
200         b *= n
201     a = isqrt(Asquare)
202     factor = gcd(b-a,N)
203     return factor
204 
205 def solve_row(sol_rows,M,marks,K=0):
206     solution_vec, indices = [],[]
207     free_row = sol_rows[K][0]
208     for i in range(len(free_row)):
209         if free_row[i] == 1:
210             indices.append(i)
211     for r in range(len(M)):
212         for i in indices:
213             if M[r][i] == 1 and marks[r]:
214                 solution_vec.append(r)
215                 break
216 
```

```
217     solution_vec.append(sol_rows[K][1])
218     return(solution_vec)
219 
220 def gauss_elim(M):
221     marks = [False]*len(M[0])
222 
223     for i in range(len(M)):
224         row = M[i]
225         for num in row:
226             if num == 1:
227                 j = row.index(num)
228                 marks[j] = True
229 
230                 for k in chain(range(0,i),range(i+1,len(M))):
231                     if M[k][j] == 1:
232                         for i in range(len(M[k])):
233                             M[k][i] = (M[k][i] + row[i])%2
234                 break
235     M = transpose(M)
236     sol_rows = []
237     for i in range(len(marks)):
238         if marks[i]== False:
239             free_row = [M[i],i]
240             sol_rows.append(free_row)
241     return sol_rows,marks,M
242 
243 def QS(n,B,I):
244     global N
245     global root
246     global T
247     N,root,K,T = n,int(sqrt(n)),0,1
248 
249     if isinstance(sqrt(N),int):
250         return isqrt(N)
```

```
251     factor_base = find_base(N,B)
252     global F
253     F = len(factor_base)
254     smooth_nums,xlist,indices = find_smooth(factor_base, N,I)
255     is_square, t_matrix = build_matrix(smooth_nums,factor_base)
256 
257     if is_square == True:
258         x = smooth_nums.index(t_matrix)
259         factor = gcd(xlist[x]+sqrt(t_matrix),N)
260         return factor, N/factor
261     sol_rows,marks,M = gauss_elim(t_matrix)
262     solution_vec = solve_row(sol_rows,M,marks,0)
263     factor = solve(solution_vec,smooth_nums,xlist,N)
264 
265     for K in range(1,len(sol_rows)):
266         if (factor == 1 or factor == N):
267             solution_vec = solve_row(sol_rows,M,marks,K)
268             factor = solve(solution_vec,smooth_nums,xlist,N)
269         else:
270             return factor, int(N/factor)
```

3.1.11.2. EJEMPLO DE EJECUCIÓN

```
(myenv) ┌─ rolando@rolando [01:17:19]  [~/Documents/tesis/tesis_factor
tzacion/src/QuadraticSieve] [master ✗]
└─> $  python3 main.py
1811706971
89 B-smooth numeros encontrados.
[72254, 327653, 583070, 1008805, 1604918, 2115950, 2286310, 2371493, 2
456678, 2627054, 2797438, 2882633, 2967830, 3138230, 3990350, 4075573,
 4160798, 4331254, 4501718, 4757429, 5865718, 5950985, 6036254, 637735
0, 6547910, 7144933, 7315529, 9021929, 9619358, 9704713, 10472998, 106
43750, 11241445, 11497630, 11753833, 12351710, 13547758, 15342565, 155
99038, 16197545, 16454078, 17052725, 17309318, 17565929, 18763685, 189
34825, 19277129, 20133029, 20646665, 24329830, 24501230, 24586933, 255
29798, 25615525, 26215670, 26387158, 26730158, 29475310, 31192070, 313
63790, 31793125, 31878998, 32308393, 32394278, 34026473, 35401513, 360
03254, 36089225, 36777065, 36949045, 37723054, 38411198, 38927390, 412
51145, 42026054, 42370510, 42542750, 43576358, 44093270, 44179429, 449
54950, 45385865, 46334054, 46678910, 47110025, 48403670, 48748718, 502
15529, 50819678]
1811706971 = (104729, 17299)
```

Figura 11: Resultado de la criba de Cuadratica

CAPÍTULO 4 EVALUACIÓN DE RESULTADOS

4.1. GENERACIÓN DE CASOS DE PRUEBA

Para la elaboración de los casos de prueba se utilizo el siguiente programa en python para lograr general los números compuestos con las características que necesitamos.

```
def primos_aleatorios_menores_a(n):
    primos = list(map(int, primos_list))
    upper = binary_search(n, primos)
    for i in range(6, 23):
        number = 1
        fact = []
        while math.log10(number) < i:
            x = random.randint(0, upper)
            prime = primos[x]
            number = number * prime
            fact.append(prime)
        print(f'{number} = {fact}')


def binary_search(x, array):
    a = 0
    b = len(array)
    while(b-a > 1):
        c = math.floor((b+a)/2)
        if array[c] > x:
            b = c
        else:
            a = c
    return a


```

```
27 def dos_primos_mitad_digitos():
28     primos = list(map(int, primos_list))
29     for i in range(6, 18):
30         lower = binary_search(pow(10, i/2), primos)
31         upper = binary_search(pow(10, (i/2)+1), primos)
32         n1 = random.randint(lower, upper+1)
33         n2 = random.randint(lower, upper + 1)
34         number = primos[n1] * primos[n2]
35         fact = [primos[n1], primos[n2]]
36         print(f'{number} = {fact}')
37 
38 
39 def dos_primos_cercanos():
40     primos = list(map(int, primos_list))
41     for i in range(6, 18):
42         lower = binary_search(pow(10, i/2), primos)
43         upper = binary_search(pow(10, (i/2)+1), primos)
44         n1 = random.randint(lower, upper+1)
45         n2 = n1 + random.randint(0, 100)*(-1**random.randint(0, 10)
    )
46         number = primos[n1] * primos[n2]
47         fact = [primos[n1], primos[n2]]
48         print(f'{number} = {fact}')
```

4.2. RECOLECCIÓN DE DATOS

Para evaluar las propiedades de los algoritmos de factorización de números enteros, se intentara factorizar diferentes números, de diferentes tamaños, y que cumplen ciertas propiedades, utilizando los algoritmos de Divisiones Sucesivas, Algoritmo de Fermat,Algoritmo de Pollard Rho, Método de factorización por curva elíptica y Criba Cuadrática, Se variaron los parámetros de entrada de cada algoritmo para evaluarlos en cuanto a tiempo de ejecución, memoria utilizada y factores primos encontrados.

Tenemos 3 conjuntos te números, en la Tabla 4.1 tenemos el primer conjunto de números los cuales tienen la particularidad de tener varios factores primos pequeños.

Número a Factorizar	Cantidad de Dígitos	Factores Primos
10856663	8	[2267, 4789]
33391459	8	[4327, 7717]
52863301961	11	[2903, 3257, 5591]
60176508803	11	[2399, 4931, 5087]
198506677433	12	[2459, 8839, 9133]
212154123015013	15	[2909, 3221, 4057, 5581]
224446720764659	15	[1019, 3571, 6269, 9839]
178466784060619	15	[577, 3527, 8933, 9817]
191470730623531	15	[1373, 2347, 6361, 9341]
1422516521828333	16	[3797, 4391, 9001, 9479]
1151160785730511661	19	[7, 479, 1069, 5653, 6367, 8923]
188790832382174409613	21	[97, 1949, 2243, 6571, 7057, 9601]
1020515758802666029	19	[607, 2663, 7621, 8861, 9349]
1724030839771633601827	22	[431, 2333, 3359, 6709, 8369, 9091]
5433376091934805169983	22	[599, 4001, 5393, 6067, 7927, 8741]
1524997222903385596412591	25	[887, 1877, 2767, 2969, 3049, 3889, 9403]
3992604690588286549205 3	23	[643, 821, 1213, 1613, 2143, 2797, 6449]

Tabla 4.1: Casos de prueba: factores pequeños

En la Tabla 4.2 tenemos el siguiente conjunto de casos de prueba los cuales son el producto de dos números primos de aproximadamente la mitad de dígitos.

Número a Factorizar	Cantidad de Dígitos	Factores Primos
16236991	8	[3361, 4831]
469526207	9	[18587, 25261]
914465141	9	[20731, 44111]
45774604399	11	[211193, 216743]
138426627499	12	[225343, 614293]
2002311773621	13	[703709, 2845369]
36162913278367	14	[4352419, 8308693]
279747589361921	15	[9353483, 29908387]
3064287451898711	16	[38873827, 78826493]
6587155271801233	16	[64776983, 101689751]
853877380996470053	18	[893082227, 956101639]
831279571604286949	18	[849122999, 978986051]
37351160168752886641	20	[5814185473, 6424143217]

Tabla 4.2: Casos de prueba: Dos primos de la mitad de dígitos

En la Tabla 4.3 se encuentra el ultimo conjunto de casos de prueba los cuales son producto de dos números primos cercanos.

Número a Factorizar	Cantidad de Dígitos	Factores Primos
66436879	8	[8017, 8287]
686703811	9	[25981, 26431]
8101438039	10	[89963, 90053]
85526931211	11	[292183, 292717]
262209534767	12	[511991, 512137]
715281207649	12	[845623, 845863]
1873966393597	13	[1368467, 1369391]
18701749916023	14	[4324261, 4324843]
1757788889012333	16	[41925839, 41926147]
34843998810184129	17	[186665113, 186665833]
874655409151893869	18	[935229067, 935231207]
286073409635796583	18	[534857399, 534859217]

Tabla 4.3: Casos de prueba: Dos primos cercanos

4.3. RESULTADOS

4.3.1. DIVISIONES SUCESIVAS

El algoritmo de divisiones sucesivas tiene una complejidad temporal de $O(\sqrt{n})$ en el peor caso y una complejidad espacial de $O(1)$. Se espera que este algoritmo se comporte bien cuando los factores primos son pequeños y en cuanto el factor más grande crezca este algoritmo dejara de ser adecuado.

Primero se realizara las pruebas con los casos de prueba de la Tabla 4.1

Número a factorizar	Tiempo de ejecución (ms)	Factores primos encontrados
10856663	0.310	[2267, 4789]
33391459	0.464	[4327, 7717]
52863301961	0.386	[2903, 3257, 5591]
60176508803	0.339	[2399, 4931, 5087]
198506677433	0.555	[2459, 8839, 9133]
212154123015013	0.354	[2909, 3221, 4057, 5581]
224446720764659	0.570	[1019, 3571, 6269, 9839]
178466784060619	0.554	[577, 3527, 8933, 9817]
191470730623531	0.585	[1373, 2347, 6361, 9341]
1422516521828333	0.580	[3797, 4391, 9001, 9479]
1151160785730511661	0.545	[7, 479, 1069, 5653, 6367, 8923]
188790832382174409613	0.629	[97, 1949, 2243, 6571, 7057, 9601]
1020515758802666029	0.618	[607, 2663, 7621, 8861, 9349]
1724030839771633601827	0.652	[431, 2333, 3359, 6709, 8369, 9091]
5433376091934805169983	0.565	[599, 4001, 5393, 6067, 7927, 8741]
1524997222903385596412591	0.627	[887, 1877, 2767, 2969, 3049, 3889, 9403]
3992604690588286549205 3	0.434	[643, 821, 1213, 1613, 2143, 2797, 6449]

Tabla 4.4: Resultados del Algoritmo de Divisiones Sucesivas para el conjunto de casos de prueba de la Tabla 4.1

Y ahora se realizara las pruebas con los casos de prueba de la Tabla 4.2

Número a factorizar	Tiempo de ejecución (ms)	Factores primos encontrados
16236991	0.295	[3361, 4831]
469526207	1.523	[18587, 25261]
914465141	2.715	[20731, 44111]
45774604399	14.768	[211193, 216743]
138426627499	37.848	[225343, 614293]
2002311773621	183.913	[703709, 2845369]
36162913278367	582.346	[4352419, 8308693]
279747589361921	2058.641	[9353483, 29908387]
3064287451898711	5634.140	[38873827, 78826493]
6587155271801233	7452.034	[64776983, 101689751]
853877380996470053	80050.156	[893082227, 956101639]
831279571604286949	76501.414	[849122999, 978986051]
37351160168752886641	768585.739	[5814185473, 6424143217]

Tabla 4.5: Resultados del Algoritmo de Divisiones Sucesivas para el conjunto de casos de prueba de la Tabla 4.2

Número a factorizar	Tiempo de ejecución (ms)	Factores primos encontrados
66436879	0.437	[8017, 8287]
686703811	1.394	[25981, 26431]
8101438039	6.293	[89963, 90053]
85526931211	21.026	[292183, 292717]
262209534767	39.504	[511991, 512137]
715281207649	64.788	[845623, 845863]
1873966393597	109.328	[1368467, 1369391]
18701749916023	339.266	[4324261, 4324843]
1757788889012333	3414.500	[41925839, 41926147]
34843998810184129	15096.030	[186665113, 186665833]
874655409151893869	81195.058	[935229067, 935231207]
286073409635796583	43553.943	[534857399, 534859217]

Tabla 4.6: Resultados del Algoritmo de Divisiones Sucesivas para el conjunto de casos de prueba de la Tabla 4.3

Como se puede ver este algoritmo funciona bastante bien cuando se tiene factores primos pequeños pero el tiempo de ejecución crece en gran medida cuando se tienen factores primos muy grandes y también se puede ver que este algoritmo devuelve todos los factores primos de los números a factorizar. Esto se puede notar mejor en el gráfico de la Figura 12

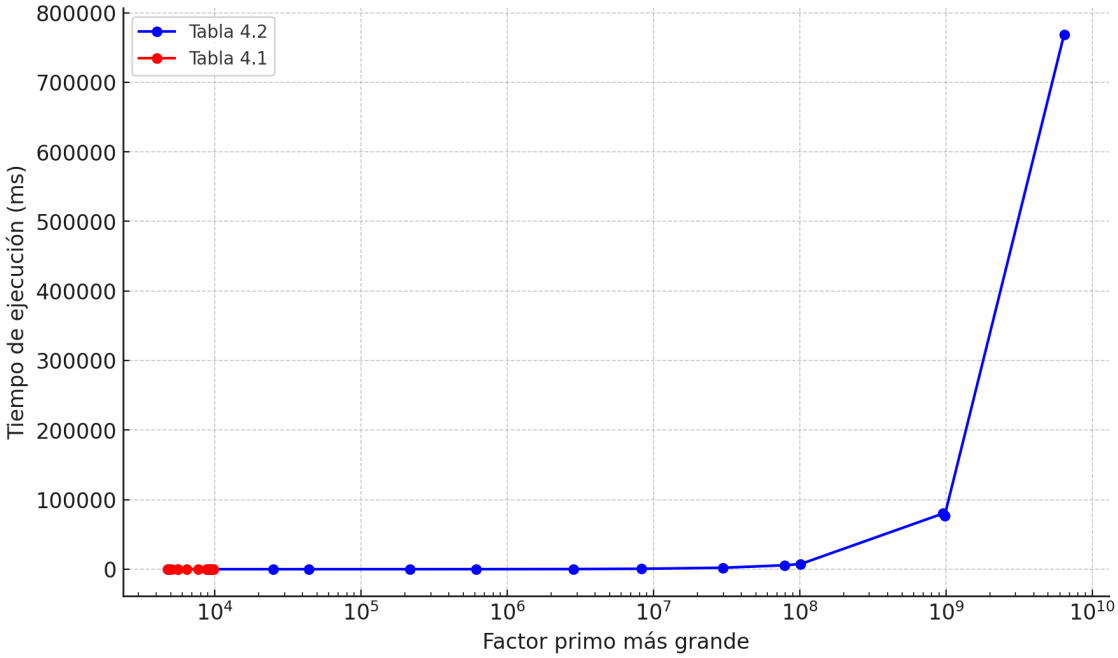

Figura 12: Resultados Algoritmo de Divisiones Sucesivas

4.3.2. MÉTODO DE DIFERENCIA DE CUADRADOS DE FERMAT

El método de diferencia de cuadrados de Fermat tiene una complejidad temporal de $O(\sqrt{n})$ en el peor caso y una complejidad espacial de $O(1)$. Se espera que este algoritmo se desenvuelva bien cuando los factores primos sean cercanos y por la naturaleza de este algoritmo solo devolverá dos factores como resultado.

Por este motivo primero se ejecutara el algoritmo con los casos de prueba de la Tabla 4.2

Número a factorizar	Tiempo de ejecución (ms)	Factores primos encontrados
16236991	0.026	(3361, 4831)
469526207	0.091	(18587, 25261)
914465141	0.602	(20731, 44111)
45774604399	0.007	(211193, 216743)
138426627499	13.287	(225343, 614293)
2002311773621	101.312	(703709, 2845369)
36162913278367	91.956	(4352419, 8308693)
279747589361921	879.645	(9353483, 29908387)
3064287451898711	1034.177	(38873827, 78826493)
6587155271801233	614.972	(64776983, 101689751)
853877380996470053	159.865	(893082227, 956101639)
831279571604286949	688.792	(849122999, 978986051)
37351160168752886641	2511.930	(5814185473, 6424143217)

Tabla 4.7: Resultados del Método de diferencia de cuadrados de Fermat con casos de prueba de la Tabla 4.2

Ahora se ejecutara el algoritmo con los casos de prueba de la Tabla 4.3, y se observara los resultados obtenidos:

Número a factorizar	Tiempo de ejecución (ms)	Factores primos encontrados
66436879	0.009	(8017, 8287)
686703811	0.030	(25981, 26431)
8101438039	0.036	(89963, 90053)
85526931211	0.003	(292183, 292717)
262209534767	0.002	(511991, 512137)
715281207649	0.002	(845623, 845863)
1873966393597	0.002	(1368467, 1369391)
18701749916023	0.002	(4324261, 4324843)
1757788889012333	0.002	(41925839, 41926147)
34843998810184129	0.002	(186665113, 186665833)
874655409151893869	0.002	(935229067, 935231207)
286073409635796583	0.002	(534857399, 534859217)

Tabla 4.8: Resultados del Método de diferencia de cuadrados de Fermat con casos de prueba de la Tabla 4.3

Se puede ver que este algoritmo funciona mejor cuando la distancia entre ambos factores

es pequeña, lo cual puede ser visto de manera gráfica en la Figura 13

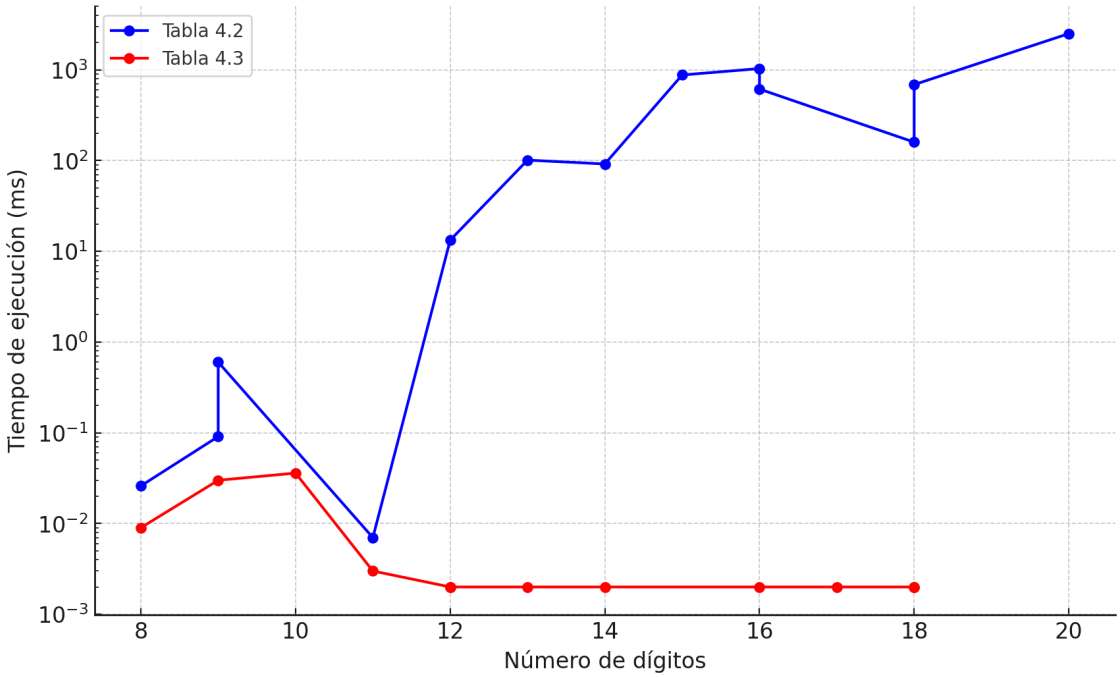

Figura 13: Resultados Método de diferencia de cuadrados de Fermat

4.3.3. ALGORITMO DE POLLARD RHO

El algoritmo de Pollard Rho tiene una complejidad temporal de $O(n^{1/4})$ en el peor caso y una complejidad espacial de $O(1)$. Se espera que este algoritmo de uso general funcione bien para multiples casos.

Comenzamos con los casos de prueba de la Tabla 4.1, que son números que tienen factores primos pequeños.

Número a factorizar	Tiempo de ejecución (ms)	Factores primos encontrados
10856663	0.124	[2267, 4789]
33391459	0.035	[4327, 7717]
52863301961	0.219	[2903, 3257, 5591]
60176508803	0.181	[2399, 4931, 5087]
198506677433	0.242	[2459, 9133, 8839]
212154123015013	0.206	[2909, 4057, 5581, 3221]
224446720764659	0.324	[1019, 9839, 3571, 6269]
178466784060619	0.175	[9817, 8933, 577, 3527]
191470730623531	0.455	[9341, 1373, 2347, 6361]
1422516521828333	0.316	[9479, 3797, 4391, 9001]
1151160785730511661	0.656	[7, 479, 1069, 8923, 5653, 6367]
188790832382174409613	0.484	[97, 7057, 6571, 1949, 2243, 9601]
1020515758802666029	0.442	[607, 7621, 2663, 8861, 9349]
1724030839771633601827	0.481	[431, 2333, 3359, 9091, 8369, 6709]
5433376091934805169983	0.494	[599, 4001, 8741, 7927, 5393, 6067]
1524997222903385596412591	0.506	[887, 2969, 3889, 1877, 2767, 3049, 9403]
3992604690588286549205 3	0.316	[1613, 995873, 643, 5993971, 6449]

Tabla 4.9: Resultados del Método de Pollard Rho con casos de la Tabla 4.1

Ahora usaremos los casos de prueba de la Tabla 4.2:

Número a factorizar	Tiempo de ejecución (ms)	Factores primos encontrados
16236991	0.103	[3361, 4831]
469526207	0.215	[25261, 18587]
914465141	0.357	[20731, 44111]
45774604399	1.069	[211193, 216743]
138426627499	1.070	[225343, 614293]
2002311773621	1.553	[703709, 2845369]
36162913278367	4.246	[4352419, 8308693]
279747589361921	2.086	[9353483, 29908387]
3064287451898711	7.950	[38873827, 78826493]
6587155271801233	21.058	[101689751, 64776983]
853877380996470053	87.079	[893082227, 956101639]
831279571604286949	60.720	[978986051, 849122999]
37351160168752886641	64.655	[6424143217, 5814185473]

Tabla 4.10: Resultados del Método de Pollard Rho con casos de la Tabla 4.2

Y por ultimo ejecutaremos el algoritmo con los casos de prueba de la Tabla 4.3

Número a factorizar	Tiempo de ejecución (ms)	Factores primos encontrados
66436879	0.213	[8287, 8017]
686703811	0.235	[25981, 26431]
8101438039	0.280	[90053, 89963]
85526931211	2.490	[292717, 292183]
262209534767	1.729	[511991, 512137]
715281207649	1.384	[845623, 845863]
1873966393597	1.494	[1369391, 1368467]
18701749916023	0.481	[4324843, 4324261]
1757788889012333	15.424	[41926147, 41925839]
34843998810184129	14.593	[186665113, 186665833]
874655409151893869	62.654	[935231207, 935229067]
286073409635796583	11.130	[534857399, 534859217]

Tabla 4.11: Resultados del Método de Pollard Rho con casos de la Tabla 4.3

En general para cualquier número Pollard-Rho tiene un tiempo de ejecución aceptable, con tiempos que no exceden los $100ms$ para nuestros casos de prueba, como se puede ver en la Figura 14.

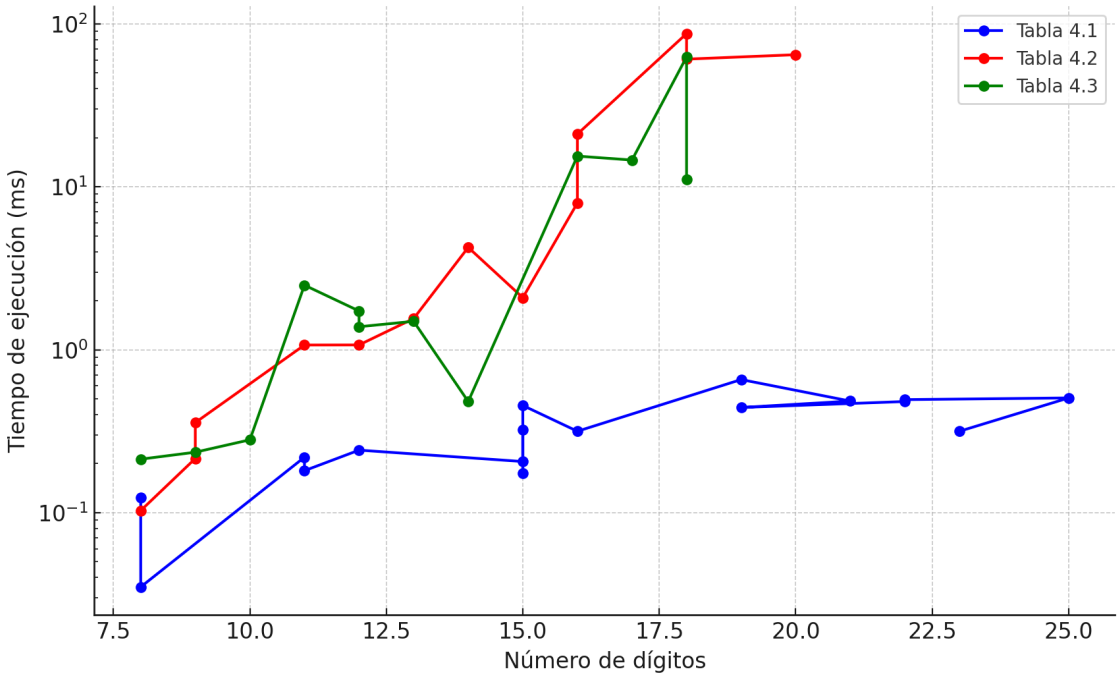

Figura 14: Resultados Algoritmo de Pollard Rho

4.3.4. MÉTODO DE FACTORIZACIÓN CON CURVAS ELIPTICAS

El método de factorización con Curvas Elipticas tiene una complejidad temporal de $O(e^{\sqrt{2 \log p \log \log p}})$ en el peor caso y una complejidad espacial de $O(\log n)$.

Número a factorizar	Tiempo de ejecución (ms)	Factores primos encontrados
10856663	8.789	2267
33391459	41.456	4327
52863301961	15.100	2903
60176508803	134.888	2399
198506677433	274.809	2459
212154123015013	28.153	2909
224446720764659	224.039	1019
178466784060619	2.281	577
191470730623531	61.980	1373
1422516521828333	67.354	4391
1151160785730511661	0.041	7
188790832382174409613	18.549	97
1020515758802666029	1.148	607
1724030839771633601827	9.571	3359
5433376091934805169983	9.329	599
1524997222903385596412591	22.790	2969
39926046905882865492053	17.009	643

Tabla 4.12: Resultados del Método de Factorización con Curvas Elipticas para los casos de prueba de la Tabla 4.1

El método de curvas elípticas es rápido en general encontrando un primer factor, pero a medida que el numero crece se hace mas complicado hacer las cuentas geométricas para encontrar los factores primos.

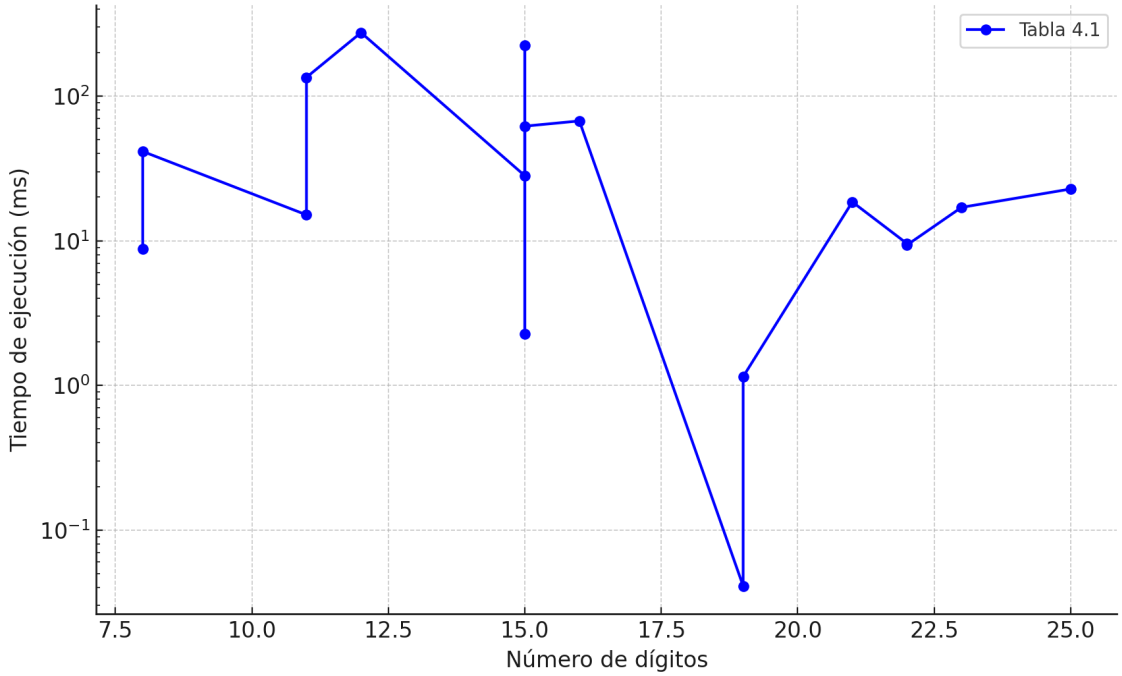

Figura 15: Resultados Método de Factorización con Curvas Elípticas con casos de prueba de la Tabla 4.1

4.3.5. ALGORITMO DE FACTORIZACIÓN DE CRIBA CUADRÁTICA

El método de factorización de Criba Cuadrática tiene una complejidad temporal de $O(e^{(1+o(n))\sqrt{\log n \log\log n}})$ en el peor caso.

Para este algoritmo se usaran los casos de prueba de la Tabla 4.2 y Tabla 4.3

Número a factorizar	Tiempo de ejecución (ms)	Factores primos encontrados
16236991	84.607	[4831.0, 3361.0]
469526207	51.509	[25261.0, 18587.0]
914465141	76.737	[44111, 20731]
45774604399	84.415	[216743.0, 211193.0]
138426627499	97.567	[225343, 614293]
2002311773621	92.401	[703709, 2845369]
36162913278367	99.894	[4352419, 8308693]
279747589361921	103.140	[29908387, 9353483]
3064287451898711	95.126	[38873827, 78826493]
6587155271801233	108.067	[101689751, 64776983]

Tabla 4.13: Resultados de la Factorización por Criba Cuadrática para los casos de prueba de la Tabla 4.2

Número a factorizar	Tiempo de ejecución (ms)	Factores primos encontrados
66436879	69.424	[8287.0, 8017.0]
686703811	70.688	[26431.0, 25981.0]
8101438039	70.603	[90053.0, 89963.0]
85526931211	74.497	[292717.0, 292183.0]
262209534767	57.068	[512137.0, 511991.0]
715281207649	90.008	[845863.0, 845623.0]
1873966393597	82.297	[1369391.0, 1368467.0]
18701749916023	69.058	[4324843.0, 4324261.0]
1757788889012333	68.385	[41926147.0, 41925839.0]

Tabla 4.14: Resultados de la Factorización por Criba Cuadrática para los casos de prueba de la Tabla 4.3

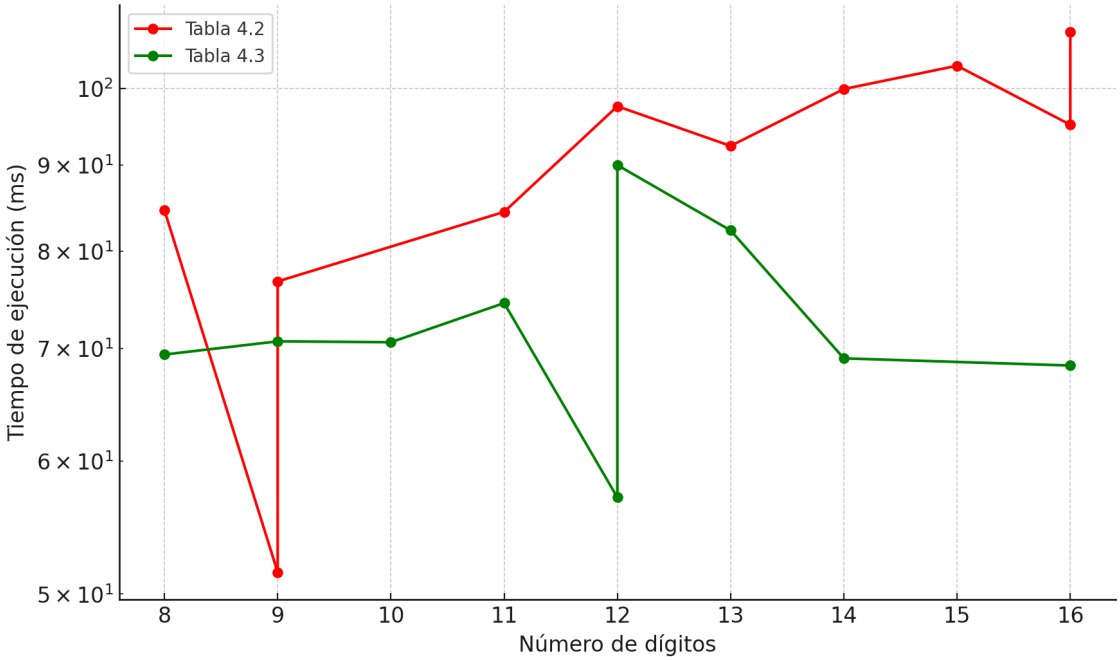

Figura 16: Resultados la Factorización con Criba Cuadrática

4.4. DEMOSTRACIÓN DE LA HIPÓTESIS

Para la demostración de la hipótesis se hará uso de los resultados obtenidos en las comparaciones, observando el tiempo de ejecución, el espacio de memoria utilizado y los factores primos encontrados.

Para la prueba de hipotesis utilizaremos los resultados de los diferentes algoritmos con los casos de prueba de la Tabla 4.3.

Número a factorizar	Divisiones Sucesivas	Fermat	Pollard Rho	Criba Cuadrática
66436879	0.437	0.009	0.213	69.424
686703811	1.394	0.03	0.235	70.688
8101438039	6.293	0.036	0.28	70.603
85526931211	21.026	0.003	2.49	74.497
262209534767	39.504	0.002	1.729	57.068
715281207649	64.788	0.002	1.384	90.008
1873966393597	109.328	0.002	1.494	82.297
18701749916023	339.266	0.002	0.481	69.058
1757788889012333	3414.5	0.002	15.424	68.385
34843998810184129	15096.03	0.002	14.593	103.500
874655409151893869	81195.058	0.002	62.654	84.799
286073409635796583	43553.943	0.002	11.13	88.860

Tabla 4.15: Tiempos de ejecución para diferentes algoritmos con los casos de prueba de la Tabla 4.3

Comparación entre Divisiones Sucesivas y Algoritmo de Fermat:

Estadístico t: $1{,}6454$

Valor p: $0{,}1281$

No se rechaza la hipótesis nula: no hay una diferencia significativa entre los algoritmos.

Comparación entre Divisiones Sucesivas y Algoritmo Pollard-Rho

Estadístico t: $1{,}6452$

Valor p: $0{,}1282$

No se rechaza la hipótesis nula: no hay una diferencia significativa entre los algoritmos.

Comparación entre Divisiones Sucesivas y Criba Cuadratica

Estadístico t: $1{,}6352$

Valor p: $0{,}1303$

No se rechaza la hipótesis nula: no hay una diferencia significativa entre los algoritmos.

Comparación entre Algoritmo de Fermat y Algoritmo Pollard-Rho

Estadístico t: $-1{,}8211$

Valor p: $0{,}0959$

No se rechaza la hipótesis nula: no hay una diferencia significativa entre los algoritmos.

Comparación entre Algoritmo de Fermat y Criba Cuadratica

Estadístico t: $-21{,}0917$

Valor p: $0{,}0000$

Rechazamos la hipótesis nula: hay una diferencia significativa entre los algoritmos.

Comparación entre Algoritmo Pollard-Rho y Criba Cuadratica

Estadístico t: $-12{,}9675$

Valor p: $0{,}0000$

Rechazamos la hipótesis nula: hay una diferencia significativa entre los algoritmos.

CAPÍTULO 5 CONCLUSIONES Y RECOMENDACIONES

5.1. CONCLUSIONES

El algoritmo de Fermat, es el más rápido cuando la diferencia entre los factores primos es pequeña y en este campo es superior a los demás algoritmos.

Si se desea factorizar números que tengan factores primos pequeños, el algoritmo de divisiones sucesivas puede ser utilizado y tiene una complejidad muy baja en cuanto a su implementación.

El algoritmo de Pollard Rho es el algoritmo predilecto para uso general en cuanto se trate de factorizar números de uso cotidiano, por la complejidad que aporta y los factores primos encontrados.

Mientras que si se desea trabajar en el campo de la criptografía o un uso más especializado en el area de Teoría De números el mejor algoritmo en estos casos es la Criba General de Cuerpo de Números o la Criba Cuadrática, estos algoritmos requieren de una comprensión más profunda de la parte teórica y una correcta implementación para su uso adecuado.

5.2. RECOMENDACIONES

Una vez concluido el presente trabajo se sugiere la posibilidad de continuar con algunas de las líneas de investigación desarrolladas e incluso incursionar en otras relacionadas con los temas tratados, se recomienda seguir con las siguientes temáticas.

Continuar con el estudio de los algoritmos cuánticos y algoritmos híbridos para posibles avances en el area de criptografía.

Bibliografía

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms*.

de Cirene, E. (200 a.C.).

Fermat. (1894). *Oeuvres de Fermat*.

Hardy, G. H., & Wright, E. M. (1975). *An Introduction to the Theory of Numbers*.

Hart, W. B. (2012). *A one line factoring algorithm*.

Knuth, D. E., & Pardo, L. T. (1975). *Analysis of a simple factorization algorithm*.

Lawrence, F. (1895). *Factorisation of numbers, Messinger of Math*.

Lenstra, H. (1987). *Factoring integers with elliptic curves*.

Morrison, M. A., & Brillhar, J. (1975). *A method of factoring and the factorization of F7*.

Niven, I., Zuckerman, H. S., & Montgomery, H. L. (1991). *An Introduction to the Theroy of Numbers*.

Pollard, J. M. (1974). *Theorems on factorization and primality testing*.

Pollard, J. M. (1975). *A Monte Carlo method for factorization*.

Pomerance, C. (1982). *Analysis and comparison of some integer factoring algorithms*.

Pomerance, C. (1994). *The number field sieve*.

S. S. Wagstaff, J. (2013). *The Joy of Factoring*.