Universidad Autónoma Metropolitana
Unidad Azcapotzalco
División de Ciencias Básicas e Ingeniería
Licenciatura en Ingeniería en Computación

Implementación de algoritmos para tratar el problema del logaritmo discreto en campos con p elementos

Modalidad: Proyecto Tecnológico

Trimestre 2018 Invierno

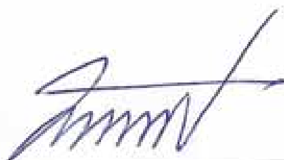
Aramast Avedis Beredgiklian Galván
Matrcula: 2133002107

Víctor Cuauhtémoc García Hernández
Profesor Asociado
Ciencias Básicas

Yo Víctor Cuauhtémoc García Hernández, declaro que aprobé el contenido del presente Reporte de Proyecto de Integración y doy mi autorización para su publicación en la Biblioteca Digital, así como en el Repositorio Institucional de la UAM Azcapotzalco.

Víctor Cuauhtémoc García Hernández

Yo, Aramast Avedis Beredgiklian Galván, doy mi autorización a la Coordinación de Servicios de Información de la Universidad Autónoma Metropolitana, Unidad Azcapotzalco, para publicar el presente documento en la Biblioteca Digital, así como en el Repositorio Institucional de UAM Azcapotzalco.



Aramast Avedis Beredgiklian Galván

Resumen

En este trabajo se realiza el diseño, la implementación, la comparación y análisis de resultados de los algoritmos *Silver-Pohlig-Hellman* y *Rho Pollard* en aritmética modular, los cuales tratan el problema del logaritmo discreto. Este trabajo esta conformado por dos secciones, en la primera sección se describe el marco teórico, la descripción de los algoritmos antes mencionados y algunas herramientas matemáticas que son parte de los algoritmos. Seguido a esto, se presenta el desarrollo de cada algoritmo, su implementación variando los parámetros y los tiempos de ejecución obtenidos de cada variación. Por último se presenta el análisis de los resultados y las conclusiones.

Índice General

1 Introducción **1**
- 1.1 Antecedentes . . . 2
- 1.2 Justificación . . . 3

2 Objetivos **5**
- 2.1 General . . . 5
- 2.2 Específicos . . . 5

3 Marco Teórico **6**
- 3.1 Problema del Logaritmo discreto . . . 6
- 3.2 Raíces de la Unidad . . . 6
- 3.3 Teorema Chino del Residuo . . . 7
- 3.4 Algoritmo Silver-Pohlig-Hellman . . . 8
- 3.5 Algoritmo Rho Pollard . . . 9

4 Desarrollo del Proyecto **11**
- 4.1 Problemas enfrentados durante el proyecto . . . 11
- 4.2 Diseño Algoritmo Silver-Pohlig-Hellman . . . 11
  - 4.2.1 División Extendida . . . 12
  - 4.2.2 Exponenciación Rápida . . . 13
  - 4.2.3 Cálculo de las r's y búsqueda de los x's . . . 14
  - 4.2.4 Configuración de parámetros y Escritura de Resultados . . . 15
  - 4.2.5 Silver-Pohlig-Hellman . . . 16
- 4.3 Diseño Algoritmo Rho Pollard . . . 18
  - 4.3.1 División Extendida . . . 18
  - 4.3.2 Exponenciación Rápida . . . 19
  - 4.3.3 Calculo de las betas, x & y . . . 20
  - 4.3.4 Búsqueda de betas iguales . . . 21
  - 4.3.5 Configuración de parámetros y Escritura de Resultados . . . 21
  - 4.3.6 Rho Pollard . . . 22

5 Resultados **25**
- 5.1 Tablas de párametros . . . 25
  - 5.1.1 Silver-Pohlig-Hellman . . . 26
  - 5.1.2 Rho Pollard . . . 27

5.2 Tablas y gráficas de resultados 28
5.2.1 Silver-Pohlig-Hellman 28
5.2.2 Rho Pollard 30

6 Conclusiones **32**
6.1 Rho Pollard vs Silver Pohlig Hellman 32

Apéndices **34**

A Algoritmo Silver-Pohlig-Hellman **34**

B Algoritmo Rho Pollard **39**

Capítulo 1

Introducción

El *problema del logaritmo discreto* es un tema clásico en Ciencias de la Computación de particular importancia en la criptografía. El término fue adoptado por su semejanza con el cálculo de logaritmos en variable real, sin embargo, aunque intuitivamente no lo parezca, el problema del logartimo discreto requiere otra clase de ideas para su estudio [1], [2].

En la forma más básica, el problema del logaritmo discreto se enuncia en estructuras aritméticas conocidas como campos con q elementos, denotados por $\mathbb{F}_q$, donde q es un número primo. Los campos $\mathbb{F}_p$ poseen operaciones suma y producto, se denota por $\mathbb{F}_q^* = \mathbb{F}_q \backslash \{0\}$ al grupo abeliano con respecto a la multiplicación [1], [3]. Se dice que dos enteros m, n son congruentes módulo q, si q divide a $m - n$ y se escribe como

$$m \equiv n \pmod{q}.$$

Se sabe que grupo multiplicativo $\mathbb{F}_q^*$ admite un generador g, véase [4]. Es decir, para cualquier entero λ que no sea múltiplo de q existe un entero x_n tal que:

$$g^{x_n} \equiv \lambda \pmod{q}.$$

Dicho x es conocido como el índice de λ según g módulo q y se denota por $\mathrm{Ind}(a)$. En general, calcular $\mathrm{Ind}(\lambda)$ es un problema abierto computacionalmente [2].

En Ciencias de la Computación, el problema del logaritmo discreto consiste en el diseño de algoritmos para determinar x_n de manera eficiente. A la fecha se desconoce si existe un algoritmo que resuelva el problema del logaritmo discreto en tiempo polinomial [1]. Sin embargo, dos de los algoritmos más exitosos son el algoritmo *Rho de Pollard* y el de *Silver-Pohlig-Hellman* [1].

Este proyecto consta de dos etapas, la primera se enfoca en el desarrollo del marco teórico para construir y hacer el análisis de complejidad de los algoritmos Rho de Pollard y Silver-Pohlig-Hellman [1], [5]. En la segunda etapa se realizará la implementación de los algoritmos Rho de Pollard y Silver-Pohlig-Hellman para tratar el problema del logaritmo

discreto. Además se presentará un análisis de los tiempos de ejecución para determinar la eficiencia de cada algoritmo.

1.1 Antecedentes

Proyectos Terminales

1. Diseño, implementación y comparación de los métodos de encriptación RSA en aritmética modular y Massey–Omura en curvas elípticas [6].

   En este proyecto se realizó la implementación de dos sistemas de encriptación, en particular Massey–Omura el cual fue implementado en curvas elípticas. A diferencia con esta propuesta que realizará la implementación de dos algoritmos que tratan el problema del logaritmo discreto. En el cual esta basada la hipótesis de Diffie–Hellman, y en esta a su vez, se basa gran parte de la seguridad de el sistema de encriptación Massey–Omura.

2. Diseño, implementación y comparación del método de encriptación ElGamal vía aritmética modular y curvas elípticas [7].

   En este proyecto se implementaron dos sistemas de encriptación, además se realizó compararon los tiempos de ejecución de los algoritmos implementados. A diferencia con esta propuesta que realizará la implementación de dos algoritmos que tratan el problema del logaritmo discreto. En el cual esta basada la hipótesis de Diffie–Hellman, y en esta a su vez, se basa gran parte de la seguridad de los sistemas de encriptación implementados.

3. Implementación en Lenguaje C de Algoritmos de Test de Primalidad y Comparación [8]

   En este proyecto se implementaron distintos algoritmos de test de primalidad para números aleatorios, grandes, se realizó una simulación de los mismos y un análisis de los resultados. A diferencia con esta propuesta la cual implementará dos algoritmos que tratan el problema del logaritmo discreto. En el cual se sabe que si los números primos son demasiado grandes podrían no encontrar una solución.

Articulos

4. Monte Carlo methods for index computations (mod p) [9].

   En este artículo de investigación se define el marco teórico, en el cual explica la funcionalidad del algoritmo Rho de Pollard, además de establecer la complejidad computacional del mismo. A diferencia con esta propuesta, que realizará dos algoritmos que tratan este problema del logaritmo discreto, uno de ellos el algoritmo Rho de Pollard.

5. An improved algorithm for computing logarithms over $\mathbb{GF}(q)$and its cryptographic significance [10].

En este artículo se habla del problema del logaritmo discreto y de que no es computacionalmente soluble, se propone el algoritmo Silver-Pohlig-Hellman para tratar el problema del logaritmo discreto y se define su complejidad computacional. A diferencia con esta propuesta que implementará el algoritmo Silver-Pohlig-Hellman.

Aplicaciones

6. Calculadora de logaritmos discretos [11].

   El objetivo principal de este calculador de logaritmos discretos online es implementar el algoritmo Silver-Pohlig-Hellman para la resolución del logaritmos discretos. Esta calculador en línea implementa el algoritmo Silver-Pohlig-Hellman para tratar logaritmos discretos. La principal diferencia esta en que la propuesta además de implementar el algoritmo Silver-Pohlig-Hellman implementará el algoritmo Rho de Pollard.

1.2 Justificación

Sea p es un número primo fijo y g un generador de $\mathbb{F}_p^*$. Entonces para todo x el cálculo

$$g^x \equiv \lambda \pmod{p},$$

se puede efectuar en tiempo polinomial. En el otro sentido, si se conoce n y la tarea es determinar ω, entonces hablamos del problema del logaritmo discreto. Por otra parte, si se escogen dos enteros α y β *privados*, que no son múltiplos de $p-1$, entonces se puede calcular de manera eficiente

$$a \equiv g^{\alpha} \pmod{p} \quad \text{y} \quad b \equiv g^{\beta} \pmod{p},$$

y asuma que las potencias a y b se hacen *públicas*. La *hipótesis de Diffie–Hellman* establece que es imposible determinar $g^{\alpha\beta} \pmod{p}$ en tiempo polinomial a partir de las potencias a, b, véase [1]. Observe que si el problema del logaritmo discreto se resuelve en tiempo polinomial, entonces se determina a los enteros α y β a partir de a y $b \pmod{p}$. En particular, se podría calcular computacionalmente

$$g^{\alpha\beta} \equiv a^{\beta} \equiv b^{\alpha} \pmod{p},$$

y de esta forma fallaría la hipótesis de Diffie–Hellman. En el otro sentido, se cree que si se resuelve la hipótesis de Diffie-Hellman, entonces el problema del logaritmo discreto sería soluble. Este hecho no ha sido demostrado pero se cree que la hipótesis de Diffie-Hellman y la intratabilidad del problema del logaritmo discreto son equivalentes [1], [2].

El problema del logaritmo discreto tiene un papel fundamental en importantes sistemas de encriptación tales como ElGamal y Massey-Omura, por mencionar algunos [1]. Gran parte de la seguridad de ElGamal se basa en la *hipótesis de Diffie–Hellman*[12] y la seguridad de Massey-Omura depende de la intratabilidad del problema del logaritmo discreto.

Aunque no se sabe si el problema del logaritmo discreto se puede resolver en tiempo polinomial, se han dedicado esfuerzos en el desarrollo de algoritmos para resolverlo. Algunos de los algoritmos clásicos y su complejidad son:

1. El algoritmo natural, complejidad de orden p.
2. Algoritmo Rho de Pollard, complejidad de orden $\sqrt{p}$, [1].
3. Algoritmo Shanks, complejidad de orden $\sqrt{p}\ \log p$, [5].
4. Algoritmo del Canguro de Pollard, complejidad de orden $\sqrt{p}$, [5].
5. Algoritmo Silver-Pohlig-Hellman. La complejidad depende de la estructura de $p-1$ y en el peor escenario es $\sqrt{p}$, [5].

El problema del logaritmo discreto y la hipótesis de Diffie–Hellman se enuncian también en estructuras aritméticas más generales, por ejemplo en cualquier campo finito y en curvas elípticas sobre campos finitos. Más aún, se han desarrollado criptosistemas cuya seguridad también depende del problema del logaritmo discreto y la hipótesis de Diffie–Hellman en estas estructuras aritméticas.

Existen herramientas de software matemático libre como Octave, se encuentran disponibles para diversos sistemas operativos y están destinados principalmente a cálculos númericos, se sabe que estas herramientas calculan logaritmos en los números reales pero no cuentan con el cálculo de logaritmos discretos en campos finitos. Por lo que este proyecto ofrece una solución a este problema ya que permitirá calcular logaritmos discretos a través de los algoritmos Rho de Pollard y Silver-Pohlig-Hellman, saber cual es el tiempo en que tarda cada algoritmo además de contar con los códigos que implementarán cada algoritmo lo cual no es ofrecido por ningún software matemático.

Capítulo 2

Objetivos

2.1 General

Implementar dos algoritmos que tratan el problema del logaritmo discreto en campos con p elementos, mediante el lenguaje de programación Python.

2.2 Específicos

1. Realizar un módulo que configure los parámetros para cada algoritmo.
2. Realizar un módulo de validación de parámetros mediante el algoritmo de la división.
3. Realizar un módulo que implemente el algoritmo Rho de Pollard en $\mathbb{F}_p$.
4. Realizar un módulo que implemente el algoritmo Silver-Pohlig-Hellman en $\mathbb{F}_p$.
5. Realizar un módulo que determine la eficiencia de cada algoritmo.
6. Realizar un módulo que compare los tiempos de ejecución de los algoritmos implementados.

Capítulo 3

Marco Teórico

3.1 Problema del Logaritmo discreto

Sea p un número primo, g una raíz primitiva en $\mathbb{F}_p*$ y λ una clase residual en $\mathbb{F}_p*$. Existe $x \in \mathbb{F}_p*$ tal que.

$$g^x \equiv \lambda \pmod{p}$$

x es conocido como el logaritmo discreto

El problema del logaritmo discreto, se basa en calcular de manera eficiente dicha x, teóricamente esta comprobado que este problema tiene solución. Computacionalmente se desconoce si existe un algoritmo que resuelva el problema del logaritmo discreto de manera eficiente, es decir, en tiempo polinomial, a la fecha se han diseñado algoritmos que tratan el problema del logaritmo discreto.

3.2 Raíces de la Unidad

Sea $n \geq 2$ un entero, las raíces enésimas de la unidad módulo p son las soluciones a la ecuación en $\mathbb{F}_p$.

$$x^n \equiv 1 \pmod{q}. \tag{3.1}$$

Las soluciones a la ecuación son conocidas como raíces enésimas de la unidad.

Si g es raíz primitiva, para cada $p|q-1$.

$$\frac{q-1}{p} \in \mathbb{Z} \qquad g^{\frac{q-1}{p}} \in \mathbb{F}_p$$

es raíz p-ésima de la unidad.

Por el pequeño teorema de Fermat, tenemos:

$$(g^{\frac{q-1}{p}})^p \equiv g^{q-1} \equiv 1 \pmod q$$

sea

$$r_j \equiv (g^{\frac{q-1}{p}})^j \pmod q \; ; \qquad\qquad 0 \geq j \geq \alpha - 1$$

se tiene

$$(r_j)^p \equiv (g^{\frac{q-1}{p}})^{jp} \equiv (g^{q-1})^j \equiv 1 \pmod q$$

El sistema r_j esta conformado de $\alpha - 1$ raíces p-ésimas de la unidad **No necesariamente distintas**

3.3 Teorema Chino del Residuo

El Teorema Chino del Residuo, ofrece una solución única a una serie de congruencias simultáneas **lineales**.

Sean $p_1, ..., p_n$ números enteros positivos primos entre sí, es decir

$$gcd(p_i, p_j) = 1 \; para \; todo \; i \neq j$$

Y $a_1, ..., a_n$ números enteros, existe una x que resuelve el sistema de congruencias simultáneas

$$x \equiv a_1 \pmod{p_1}, \; x \equiv a_2 \pmod{p_2}, \; ... \; , \; x \equiv a_n \pmod{p_n}$$

Sea

$$p = \prod_{i=1}^{n} p_i, \; M_i = \frac{m}{m_i}, \quad 1 \leq i \leq n$$

Observe que la solución a la congruencia simultánea es única módulo p, dado que p_i es coprimo por parejas, se tiene

$$gcd(p_i, M_i) = 1, \quad 1 \leq i \leq n$$

Es decir que existe un inverso multiplicativo para ese par de numeros, del algoritmo de la división extendida de Euclides obtenemos $y_i, \;\; 1 \leq i \leq n$ tal que

$$y_i M_i \equiv 1 \pmod{p_i}, \quad 1 \leq i \leq n$$

Se construye a x de la siguiente forma

$$x = (\sum_{i=1}^{n} a_i y_i M_i) \pmod{p-1}$$

la cual es una solución de la congruencia simultánea.

3.4 Algoritmo Silver-Pohlig-Hellman

Asuma que se conoce la descomposición canónica de $q-1$ como producto de primos.

$$q - 1 = p_1^{\alpha_1} + ... + p_t^{\alpha_t} \qquad\qquad \alpha_i > 0$$

El algoritmo *Silver–Pohlig–Hellman* encuentra $x \pmod{p^\alpha}$ para cada $p|q-1$ tal que $p^{\alpha+1}||q-1$. Esto significa que $p|q-1$, $p^{\alpha+1} \not| q-1$.

Se calculan las raices p-ésimas de la unidad para cada primo p divisor de $q-1$.

$$r_{p,j} = g^{\frac{j(q-1)}{p}} \; para \; j = 0, ..., \alpha - 1$$

Para cada primo $p|q-1$, se desea encontrar x tal que $x \equiv X_p \pmod{p^\alpha}$ y aplicar el *Teorema Chino del Residuo* para encontrar la solución al problema del logaritmo discreto.

Si para cada primo $p|q-1$, se encuentra X_p tal que

$$x \equiv X_p \pmod{p^\alpha}$$

y se escribe

$$X_p \equiv X_0 + X_1 p + X_2 p^2 + ... + X_{\alpha-1} p^{\alpha-1} \pmod{p^\alpha}$$

Se desea encontrar $X_0, X_1, ..., X_{\alpha-1}$. Considere $\lambda \in \mathbb{F}_p*$, es raíz primitiva módulo q. Para calcular X_0 sea

$$(\lambda^{\frac{q-1}{p}})^p \equiv \lambda^{q-1} \equiv 1 \pmod{q}$$

observe que

$$\lambda^{\frac{q-1}{p}} \equiv g^{x \; \frac{q-1}{p}} \equiv g^{X_0 + X_1 p + ... + X_{\alpha-1} p^{\alpha-1} \frac{q-1}{p}} \equiv g^{X_0 (\frac{q-1}{p})} \in g^{(\frac{q-1}{p})} = r_{p,j}$$

Así

$$\lambda^{\frac{q-1}{p}} \in r_{p,j} \qquad X_0 = j_0 \qquad tal \; que \; \lambda^{\frac{q-1}{p}} = r_{p,j_0}$$

De manera general, se puede calcular la raíz p-ésima $\pmod{q}$ conocida como λ_j, de la siguiente manera.

$$\lambda_j \equiv \frac{\lambda}{g^{X_0 + X_1 p + ... + X_{j-1} p^{j-1}}} \equiv g^{x_j p^j + ... + x_{\alpha-1} p^{\alpha-1}}$$

Así

$$\lambda_j = r_{p,j} \qquad X_j = j$$

El Teorema Chino del Residuo garantiza que

$$x \equiv \sum_{p|q-1} X_p M_p m_p \pmod{q-1}$$

es el logaritmo discreto $\pmod{q-1}$ donde

$$M_p = \frac{q-1}{p}, \qquad M_p m_p \equiv 1 \pmod{p^\alpha}$$

3.5 Algoritmo Rho Pollard

El algoritmo Rho de Pollard, es de orden $\sqrt{p}$ y requiere de un almacenamiento constante de elementos. Este algoritmo resuelve el problema del logaritmo discreto.

Se necesita dividir a $\mathbb{F}_p*$ en 3 subconjuntos g_0, g_1, g_2 tal que

$$g_0 \cup g_1 \cup g_2 = \mathbb{F}_p*$$

se define $f : G \to G$ como

$$f(\beta) = \begin{cases} g\beta & si \quad \beta \in g_0 \\ \beta^2 & si \quad \beta \in g_1 \\ \lambda\beta & si \quad \beta \in g_2 \end{cases}$$

Para iniciar el algoritmo se debe buscar un numero aleatorio x_0 que se encuentre en el conjunto $\{1, ..., q\}$ y calcular el elemento $\beta_0 = g^{x_0}$. Despues se calcula (β_i) tal que

$$\beta_{i+1} = f(\beta_i)$$

La secuencia anterior puede escribirse como

$$\beta_i = g^{x_i}\lambda^{y_i}, \ i \geq 0$$

Donde, x_0 es un numero inicial aleatorio, $y_0 = 0$, la secuencia de las x_i y y_i se calculan de la siguiente forma

$$x_{i+1} = \begin{cases} x_i + 1 & si \quad \beta_i \in g_0 \\ 2x_i & si \quad \beta_i \in g_1 \\ x_i & si \quad \beta_i \in g_2 \end{cases}$$

y

$$y_{i+1} = \begin{cases} y_i & si \quad \beta_i \in g_0 \\ 2y_i & si \quad \beta_i \in g_1 \\ y_i + 1 & si \quad \beta_i \in g_2 \end{cases}$$

Dadas las propiedades de los campos finitos, dos elementos de la secuencia (β_i) deberán ser iguales, es decir, existe $i \geq 0$ y $k \geq 1$ tal que $\beta_{i+1} = \beta_i$, esto implica

$$g^{x_i}\lambda^{y_i} = g^{x_{i+k}}\lambda^{y_{i+k}}$$

por lo tanto

$$g^{x_i - x_{i+k}} = \lambda^{y_{i+k} - y_i}$$

Por lo tanto el logaritmo x de λ base g satisface

$$(x_i - x_{i+k}) \equiv x(y_{i+k} - y_i) \pmod{q-1}$$

La solución es única $\pmod{q-1}$ si $y_{i+k} - y_i$ es primo relativo con $q-1$ ($GCD(y_{i+k} - y_i, q-1) = 1$), si la solución no es única entonces $GCD(y_{i+k} - y_i, q-1) > 1$, se tiene $d = GCD(y_{i+k} - y_i, q-1)$ tal que

$$\frac{x_i - x_{i+k}}{d} \equiv \frac{y_{i+k} - y_i}{d} \pmod{\frac{q-1}{d}}$$

La solución a esta nueva congruencia es única $\pmod{\frac{q-1}{d}}$, por lo tanto el logaritmo discreto es un valor en $x = (\frac{y_{i+k}-y_i}{d})^* + \omega * \frac{q-1}{d}, 0 \leq \omega < d$. Si no se encuentra el logaritmo discreto, se deberá aplicar el algoritmo nuevamente con un x_0 diferente.

Capítulo 4

Desarrollo del Proyecto

El proyecto, fue dividido en 3 partes, en la primera se realizó el diseño del pseudocódigo para el algoritmo Silver-Pohlog-Hellman, en la segunda parte se realizó el diseño del pseudocódigo para el algoritmo Rho Pollard. Y en la tercera parte, se realizó un análisis de los resultados.

4.1 Problemas enfrentados durante el proyecto

El principal problema al que nos enfrentamos, se presento al realizar el algoritmo Rho Pollard, ya que este requiere memoria $\sqrt{p}$, por otro lado cada que se calculaba β nueva, se debía hacer una comparación con todas las anteriores, es decir cada búsqueda era de orden n ya que se debía buscar en todo el arreglo una coincidencia.

Para resolver este par de problemas, el número de betas calculadas correspondería a potencias de 2, tal que

$$\#\ de\ \beta's\ a\ calcular \ = \ 2^i, \quad 0 \leq i \leq j$$

De este modo, se podría reducir el número de comparaciones gracias a que estamos en clases residuales, existirá $\beta_i = \beta_{i+j}$ y se puede desechar betas anteriores, de tal forma que el almacenamiento y las comparaciones se reducen significativamente.

4.2 Diseño Algoritmo Silver-Pohlig-Hellman

Este algoritmo consta de 5 módulos, los cuales se presentarán a continuación.

4.2.1 División Extendida

Este módulo es una modificación del algoritmo de Euclides, dado que permite conocer el máximo común divisor (**MCD**)de dos enteros además permite expresar al **MCD** como una combinación lineal. Lo cual permite encontrar inversos multiplicativos. Los pseudocódigos de este módulo se presentan a continuación.

Algoritmo 1 Función DivExtends(m,n,flag)

Entrada: Tres enteros m, n y flag (bandera).
Salida: Máximo Común Divisor de m y n.

$m \leftarrow m$
$n \leftarrow n$
if $m = 0$ and $n = 0$ **entonces**
  return No hay GCD
else if $m = 0$ **entonces**
  $gcd \leftarrow n$
else if $n = 0$ **entonces**
  $gcd \leftarrow m$
else if $m \neq 0$ and $n \neq 0$ **entonces**
  $gcd \leftarrow$ mcd(m,n,flag)
end if
return gcd

Algoritmo 2 Función mcd(m,n,flag)

Entrada: Tres enteros m, n y flag (bandera).
Salida: Máximo Común Divisor de m y n, inverso multiplicativo dependiendo las entradas.

$i = 0$
$r_0 = m$
$r_1 = n$
$r_2 = i$
$T \leftarrow \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$
while $r_2 \neq 0$
  $q \leftarrow \frac{r_0}{r_1}$
  $E \leftarrow \begin{bmatrix} q & 1 \\ 1 & 0 \end{bmatrix}$
  $T \leftarrow T \bullet E$
  $x \leftarrow T[2, 1]$
  $y \leftarrow T[1, 1]$
  $r_2 \leftarrow (-1)^i * x * m + (-1)^{i+1} * y * n$
  $r_0 \leftarrow r_1$
  $r_1 \leftarrow r_2$
  $i + +$
end while
$GCD \leftarrow \lfloor r_0 \rfloor$
return GCD

4.2.2 Exponenciación Rápida

Este módulo se encarga de realizar el calculo de potencias muy grandes, ya que reduce el número de operaciones que debe hacer la computadora para este calculo, haciendo una reducción modular.

Algoritmo 3 Función Fastexp(g,p,n)

Entrada: Tres enteros g, p y n
Salida: $g^n \pmod p$

```
if g = 0 entonces
    powa ← 0
else if n = 0 and g ≠ 0 entonces
    powa ← 1
else if g = 1 entonces
    powa ← 1
else
    powa ← 1
    i ← 1
end if
while n > 0
    δ ← n%2
    powa ← (powa * g^δ)%p
    n ← ⌊(n−δ)/2⌋
    g ← (g * g)%p
    i + +
end while
return powa
```

4.2.3 Cálculo de las r's y búsqueda de los x's

El módulo *calculo_y_busqueda* tiene dos tareas, en la primera se encarga de de calculas las raíces p- ésimas de la unidad. La segunda tarea

Algoritmo 4 Función calculo_busqueda(g,lamb,q,p,alpha)

```
j ← 0
rpj ← [ ]
qmenosuno ← q − 1
xp ← [ ]
for i = 0 hasta alpha
  xp[i] ← −1
end for
while j ≤ (p − 1)
  n ← ⌊qmenosuno / p⌋ * j
  aux_result ← fastexp(g, q, n)
  j + +
  rpj ← aux_result
end while
j ← 0
while j ≤ (alpha − 1)
  p_j ← p^(j+1)
  x ← fastexp(lamb, q, ⌊qmenosuno / p_j⌋)
  for i = 0 hasta longitud(rpj)
    if x = rpj[i] entonces
      xp[j] ← i
      break
    end if
  end for
  z ← (xp[j] * p^j)%q
  auxg ← fastexp(g, q, z)
  g_inv_z ← DivExtends(auxg, q, 1)
  lamb ← (lamb * g_inv_z)%q
  j + +
end while
return xp
```

4.2.4 Configuración de parámetros y Escritura de Resultados

Una de las funciones de este módulo, es leer de un archivo los parámetros, separarlos y darle un formato de tipo entero para que los cálculos se efectúen de manera correcta. Por otro lado se encarga de verificar si la congruencia tiene solución, de ser así, escribirá en un archivo la solución y el tiempo de ejecución para cada conjunto de parámetros.

Algoritmo 5 Función escribir_resultados(g,q,x,lamb,tiempo_inicial)

$guardado$ = **false**
if fastexp(g,q,x = $lamb$) **entonces**
  Abrir Archivo
  $tiempo_final \leftarrow NOW$
  $tiempo_ejecucion \leftarrow tiempo_final - tiempo_inicial$
  Escribir solución a la ecuación y tiempo de ejecución
  $guardado$ = **true**
else
  $guardado$ = **false**
end if
return $guardado$

Algoritmo 6 Función configuracion_parametros(linea)

$valores = linea.split('\#')$
$lamb \leftarrow valores[0]$
$g \leftarrow valores[1]$
$q \leftarrow valores[2]$
$facp \leftarrow valores[3]$
$primes \leftarrow valores[4]$
$pows \leftarrow valores[5]$
$fprimo \leftarrow primes.split(',')$
$alpha \leftarrow pows.split(',')$
return $lamb, g, q, facp, fprimo, alpha$

4.2.5 Silver-Pohlig-Hellman

Este es el módulo principal ya que manda llamar a los módulos anteriores, además de usar el algoritmo del Teorema Chino del Residuo para encontrar la solución al sistema de congruencias generado por los parámetros.

Algoritmo 7 Función Silver_Pohlig_Hellman()

```
Abrir el archivo que contiene los párametros.
for cada línea del archivo
  rpj ← [ ]
  tiempo_inicial ← NOW
  li ← [ ]
  pi ← [ ]
  Mi ← [ ]
  mi ← [ ]
  lamb, g, q, facp, fprimo, alpha ← configuracion_parametros(linea)
  for x = 0 hasta facp
    primo ← ⌊fprimo[x]⌋
    potencia ← ⌊alpha[x]⌋
    a ← calculo_busqueda(g, lamb, q, primo, potencia)
    aux ← a[0]
    for x = 1 hasta longitud(a)
      aux_x ← ⌊a[x] * primo^x⌋
      aux ← aux + aux_x
    end for
    li ← aux
    pi ← primo^potencia
  end for
  qmenosuno ← 1
  for xx = 0 hasta facp
    qmenosuno ← qmenosuno * pi[xx]
  end for
  for xxx = 0 hasta facp
    auxMi ← ⌊qmenosuno / pi[xxx]⌋
    Mi ← auxMi
  end for
  for j = 0 hasta facp
    if pi[j] < Mi[j] entonces
      Mi ← DivExtends(pi, Mi, 0)
    else
      Mi ← DivExtends(Mi, pi, 1)
    end if
  end for
  for i = 0 hasta facp
    aux_candidato ← ⌊li[i] * mi[i] * Mi[i]⌋
    x_candidato ← x_candidato + aux
  end for
  x ← x_candidato%qmenosuno
  if !escribir_resultados(g, q, x, lamb, tiempo_inicial) entonces
    print  Error, no hay logaritmo discreto
  end if
end for
```

4.3 Diseño Algoritmo Rho Pollard

El algoritmo Rho de Pollard, consta de 5 módulos, al igual que en Silver-Pohlig-Hellman utiliza el algoritmo de la división extendida con algunas modificaciónes, la exponenciación rápida y los módulos de escritura y condiguración de parámetros con sus respectivos cambios, además de hacer uso de otros módulos, estos se presentan a continuación.

4.3.1 División Extendida

Este módulo es una modificación del algoritmo de Euclides, dado que permite conocer el máximo común divisor (**MCD**)de dos enteros además permite encontrar inversos multiplicativos. Los pseudocódigos de este módulo se presentan a continuación.

Algoritmo 8 Función DivExtends(m,n,flag)

Entrada: Tres enteros m, n y flag (bandera).
Salida: Máximo Común Divisor de m y n.
  $m \leftarrow m$
  $n \leftarrow n$
  if $m = 0$ and $n = 0$ **entonces**
    return No hay GCD
  else if $m = 0$ **entonces**
    $gcd \leftarrow n$
  else if $n = 0$ **entonces**
    $gcd \leftarrow m$
  else if $m \neq 0$ and $n \neq 0$ **entonces**
    $inverso, gcd \leftarrow$ mcd(m,n,flag)
  end if
  return $inverso, gcd$

Algoritmo 9 Función mcd(m,n,flag)

Entrada: Tres enteros m, n y flag (bandera).
Salida: Máximo Común Divisor de m y n, inverso multiplicativo dependiendo las entradas.

$i = 0$
$r_0 = m$
$r_1 = n$
$r_2 = i$
$T \leftarrow \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$
while $r_2 \neq 0$
  $q \leftarrow \frac{r_0}{r_1}$
  $E \leftarrow \begin{bmatrix} q & 1 \\ 1 & 0 \end{bmatrix}$
  $T \leftarrow T \bullet E$
  $x \leftarrow T[2, 1]$
  $y \leftarrow T[1, 1]$
  $r_2 \leftarrow (-1)^i * x * m + (-1)^{i+1} * y * n$
  $r_0 \leftarrow r_1$
  $r_1 \leftarrow r_2$
  $i + +$
end while
$potencia1 \leftarrow (-1)^i$
$potencia2 \leftarrow (-1)^{i+1}$
$X \leftarrow T[0, 1]$
$Y \leftarrow T[1, 1]$
if $flag = 0$ **entonces**
  $GCD \leftarrow r_0$
  $inverso \leftarrow (yi * potencia1)\%m$
else if $flag = 1$ **entonces**
  $GCD \leftarrow r_0$
  $inverso \leftarrow (xi * potencia2)\%n$
end if
return $inverso, GCD$

4.3.2 Exponenciación Rápida

Este módulo se encarga de realizar el calculo de potencias muy grandes, ya que reduce el número de operaciones que debe hacer la computadora para este calculo, haciendo una reducción modular.

Algoritmo 10 Función Fastexp(g,p,n)

Entrada: Tres enteros g, p y n
Salida: $g^n \pmod p$

```
if g = 0 entonces
    powa ← 0
else if n = 0 and g ≠ 0 entonces
    powa ← 1
else if g = 1 entonces
    powa ← 1
else
    powa ← 1
    i ← 1
end if
while n > 0
    δ ← n%2
    powa ← (powa * g^δ)%p
    n ← ⌊(n−δ)/2⌋
    g ← (g * g)%p
    i++
end while
return powa
```

4.3.3 Calculo de las betas, x & y

La función de este módulo es calcular β_{i+1}, x_{i+1}, y_{i+1} dadas β_i, y_i y x_i, el pseudocódigo se presenta a continuación.

Algoritmo 11 Función calcula_beta(betai,g,q,lamb,xi,yi)

```
if betai%3 = 0 entonces
    xi ← (xi + 1)%(q − 1)
    yi ← yi%(q − 1)
    betai ← (fastexp(g, q, xi) * fastexp(lamb, q, yi))%q
else if betai%3 = 1 entonces
    xi ← (2 * xi)%(q − 1)
    yi ← (2 * yi)%(q − 1)
    betai ← (fastexp(g, q, xi) * fastexp(lamb, q, yi))%q
else if betai%3 = 2 entonces
    xi ← (xi)%(q − 1)
    yi ← (yi + 1)%(q − 1)
    betai ← (fastexp(g, q, xi) * fastexp(lamb, q, yi))%q
end if
return betai, xi, yi
```

4.3.4 Búsqueda de betas iguales

Debido a que se esta trabajando en clases residuales, este módulo se encarga de buscar los índices donde 2 betas sean iguales.

Algoritmo 12 Función busca_betas_iguales(r,inicio,B)

$valor_fijo \leftarrow B[r-1]$
for $x = 0$ hasta $r-1$
  if $B[x] = valor_fijo$ **entonces**
    return $x, r-1,$ **false**
  end if
end for
return $x, r-1,$ **true**

4.3.5 Configuración de parámetros y Escritura de Resultados

Las funciones de este módulo son, leer de un archivo los parámetros, separarlos y darle un formato de tipo entero para que los cálculos se efectúen de manera correcta. Por otro lado se encarga de verificar si la congruencia tiene solución, de ser así, escribirá en un archivo la solución y el tiempo de ejecución para cada conjunto de parámetros.

Algoritmo 13 Función escribir_resultados(g,q,x,lamb,tiempo_inicial)

$guardado =$ **false**
if fastexp(g,q,x = $lamb$) **entonces**
  Abrir Archivo
  $tiempo_final \leftarrow NOW$
  $tiempo_ejecucion \leftarrow tiempo_final - tiempo_inicial$
  Escribir solución a la ecuación y tiempo de ejecución
  $guardado =$ **true**
else
  $guardado =$ **false**
end if
return $guardado$

Algoritmo 14 Función configuracion_parametros(linea)

$valores = linea.split('\#')$
$lamb \leftarrow valores[0]$
$g \leftarrow \lfloor valores[1] \rfloor$
$q \leftarrow \lfloor valores[2] \rfloor$
return $lamb, g, q$

4.3.6 Rho Pollard

Este es el módulo principal, llama a los módulos anteriores y calcula las posibles soluciones a través de un par de betas, si no hay solución, vuelve a iterar el algoritmo con un valor de x_0 distinto.

Algoritmo 15 Función Rho_Pollard()

```
Abrir el archivo que contiene los párametros.
for cada línea del archivo
    termindo ← true
    tiempo_inicial ← NOW
    while terminado
        B ← [ ]
        X ← [ ]
        Y ← [ ]
        yi ← 0
        lam, g, q ← configuracion_parametros(linea)
        xi ← RANDOM
        betai ← fastexp(g, q, xi)
        B ← betai
        X ← xi
        Y ← yi
        r ← 1
        inicio ← 0
        betx ← 2
        solucion ← true
        while solucion
            betai, xi, yi ← calcula_beta(B[r − 1], g, q, lamb, X[r − 1], Y[r − 1]
            B ← betai
            X ← xi
            Y ← yi
            if longitud(B) = betx entonces
                pi, si, solucion ← busca_betas_iguales(betx, inicio, B)
                inicio ← betx
                betx ← betx ∗ 2
            end if
            r + +
        end while
        resta_Y ← |Y[si] − Y[pi]|
        resta_X ← |X[pi] − X[si]|
        inverso, gcd ← DivExtends(resta_Y, (q − 1), 1)
        nuevomodulo ← ⌊(q−1)/gcd⌋
        nuevaY ← ⌊resta_X/gcd⌋
        nuevaX ← ⌊resta_Y/gcd⌋
        inverso, nuevo_gcd ← DivExtends(nuevaY, nuevomodulo, 1)
        n ← (auxr ∗ nuevax)%nuevomodulo
        for x = 1 hasta gcd + 1
            logaritmo_candidato ← (n + (x ∗ nuevomodulo))%q
            if escribir_resultados(g, q, nld, lamb, tiempo_inicial) entonces
                terminado ← true
                break
            end if
        end for
    end while
end for
```

1

Capítulo 5

Resultados

5.1 Tablas de párametros

Para realizar las comparaciones, se escogieron 30 números primos (p) entre 10 y 13 cifras, una raíz primitiva (g), una clase residual (λ) y la descomposición de $(p-1)$, a continuación se presentan los datos en las tablas siguientes.

5.1.1 Silver-Pohlig-Hellman

Clase Residual (λ)	Raíz Primitiva (g)	p	Descomposición de $p-1$
654	2	1896548141	$2^2, 5, 6329, 14983$
242	3	1987459589	$2^2, 37, 1009, 13309$
546	13	2269786747	$2, 3, 2663, 142057$
474	6	2549879581	$2^2, 3, 5, 31, 167, 8209$
564	13	2836549477	$2^2, 3^3, 31, 847237$
646	5	1654875463	$2, 3, 2789, 98893$
735	12	2987441081	$2^3, 5, 13, 5745079$
257	13	2038072901	$2^2, 5^2, 853, 23893$
862	11	22801763489	$2^5, 7^2, 47, 309403$
372	3	22801762013	$2^2, 131, 2851, 15263$
227	7	24192095747	$2, 7, 13, 271, 490493$
575	15	31856365477	$2^2, 3^3, 13^2, 331, 5273$
142	10	32856366619	$2, 3^2, 1663, 1097627$
547	11	33654875831	$2, 5, 41, 617, 133039$
682	3	34687412789	$2^2, 7, 83, 14925737$
427	7	46874522419	$2, 3, 29, 6029, 44683$
234	5	56624590507	$2, 3, 11, 37, 43, 73, 83, 89$
487	5	94875643487	$2, 13, 59, 83, 701, 1063$

Clase Residual (λ)	Raíz Primitiva (g)	p	Descomposición de $p-1$
548	7	179354493841	$2^4, 3, 5, 23251, 32141$
785	2	146978636219	$2, 809, 3331, 27271$
964	11	896346551473	$2^4, 3, 19, 29473, 33347$
348	5	217896435013	$2^2, 3, 23, 787, 827, 1213$
165	14	489876518113	$2^5, 3, 29, 61, 107, 26959$
492	3	546974353001	$2^3, 5^3, 113, 4840481$
648	13	899613246449	$2^4, 7, 881, 9117209$
389	10	100747316713	$2^3, 3, 7^2, 149, 574963$
697	5	2038073003	$2, 7, 1361, 106963$
135	2	3121238429	$2^2, 7, 11, 421, 24071$
364	7	2468775769	$2^3, 3, 17, 31, 47, 4153$
831	12	22801763119	$2, 3, 823, 1583, 2917$

Tabla 5.1: Parámetros del algoritmo Silver-Pohlig-Hellman

5.1.2 Rho Pollard

Clase Residual (λ)	Raíz Primitiva (g)	p
654	2	1896548141
242	3	1987459589
546	13	2269786747
474	6	2549879581
564	13	2836549477
646	5	1654875463
735	12	2987441081
257	13	2038072901
862	11	22801763489
372	3	22801762013
227	7	24192095747
575	15	31856365477
142	10	32856366619
547	11	33654875831
682	3	34687412789
427	7	46874522419
234	5	56624590507
487	5	94875643487
548	7	179354493841
785	2	146978636219
964	11	896346551473
348	5	217896435013

Clase Residual (λ)	Raíz Primitiva (g)	p
165	14	489876518113
492	3	546974353001
648	13	899613246449
389	10	100747316713
697	5	2038073003
135	2	3121238429
364	7	2468775769
831	12	22801763119

Tabla 5.2: Parámetros del algoritmo Rho Pollard

5.2 Tablas y gráficas de resultados

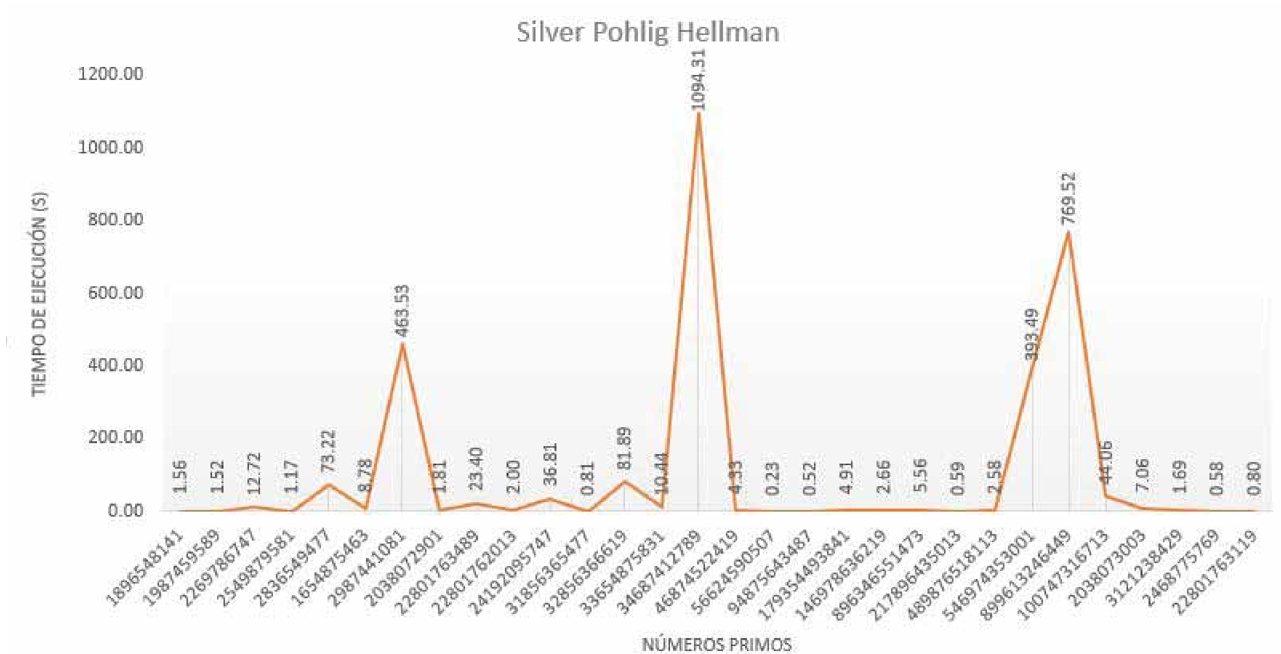
Para cada número primo y sus parámetros, se calculó el tiempo de ejecución, los resultados se muestran en las siguientes tablas, acompañadas de sus correspondientes gráficas.

5.2.1 Silver-Pohlig-Hellman

Número Primo	Tiempo de ejecución
1896548141	1.5625364780426
1987459589	1.51565527915954
2269786747	12.7189044952392
2549879581	1.17188167572021
2836549477	73.21520113945
1654875463	8.78134155273437
2987441081	463.526111364364
2038072901	1.81252217292785
22801763489	23.4005079269409
22801762013	2.00003147125244
24192095747	36.8129565715789
31856365477	0.812496423721313
32856366619	81.8895328044891
33654875831	10.4376490116119
34687412789	1094.31480717658
46874522419	4.32817029953002
56624590507	0.234390497207641
94875643487	0.515638351440429
179354493841	4.90630030632019

Número Primo	Tiempo de ejecución
146978636219	2.65629410743713
896346551473	5.56255054473876
217896435013	0.589229106903076
489876518113	2.57813787460327
546974353001	393.489261865615
899613246449	769.521903038024
100747316713	44.0589423179626
2038073003	7.06258249282836
3121238429	1.68753004074096
2468775769	0.578134775161743
22801763119	0.796858549118041

Tabla 5.3: Resultados de los tiempos de ejecución del algoritmo Silver-Pohlig-Hellman



Gráfica 5.1: Tiempos de ejecución Silver-Pohlig-Hellman

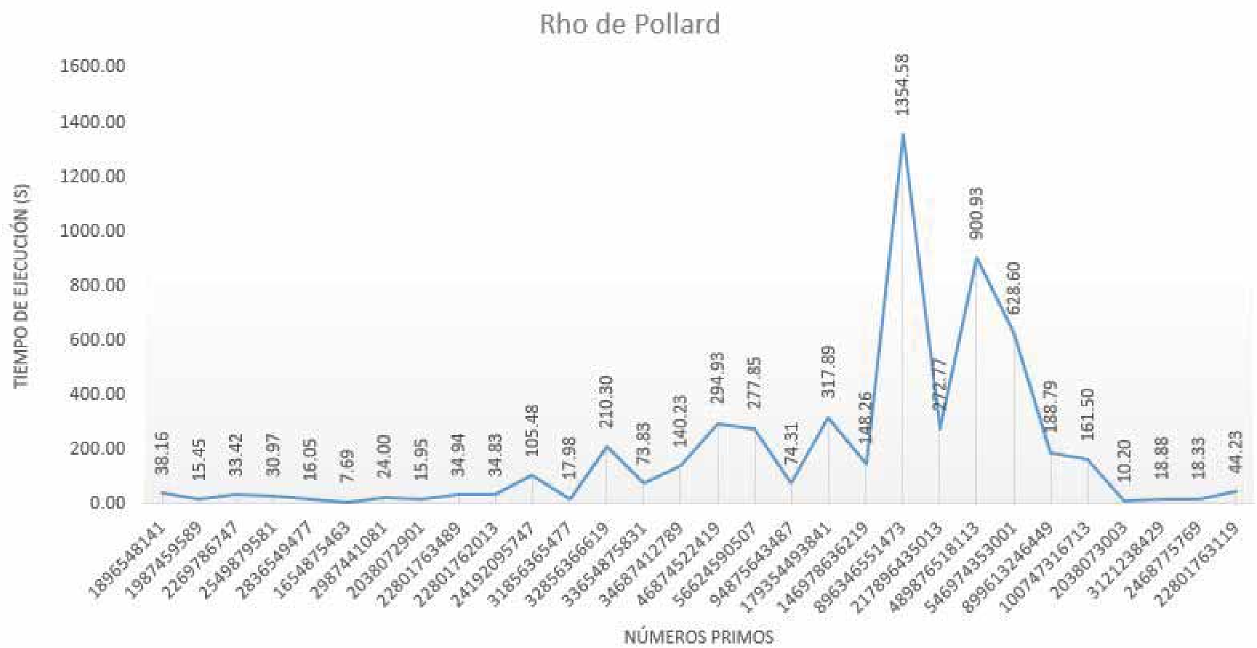
Con base en los datos de la Tabla 5.1 y en la gráfica 5.1 se observa, que el tiempo de ejecución de Silver-Pohlig-Hellman se basa en la descomposición como primos de $p-1$, el tiempo es menor si los divisores primos son pequeos en cuanto a su tamao en cifras, de lo contrario se observa un aumento significativo en el tiempo de ejecución si uno o más divisores primos son de al menos la mitad de cifras.

5.2.2 Rho Pollard

Los números primos, la clase residual y la raíz primitiva que se utilizaron para el algoritmo Rho Pollard, son identicos que Silver-Pohlig-Hellman.

Número Primo	Tiempo de ejecución
1896548141	38.1565976142883
1987459589	15.4532997608184
2269786747	33.4210793972015
2549879581	30.9691359996795
2836549477	16.0470895767211
1654875463	7.68757772445678
2987441081	24.0002975463867
2038072901	15.9533224105834
22801763489	34.9379520416259
22801762013	34.828544139862
24192095747	105.481971979141
31856365477	17.9845941066741
32856366619	210.302386522293
33654875831	73.8290593624114
34687412789	140.233712673187
46874522419	294.925548076629
56624590507	277.850557804107
94875643487	74.3052735328674
179354493841	317.893409967422
146978636219	148.262935638427
896346551473	1354.58217978477
217896435013	272.765139818191
489876518113	900.93481349945
546974353001	628.601037025451
899613246449	188.787221908569
100747316713	161.500504493713
2038073003	10.2032623291015
3121238429	18.8751983642578
2468775769	18.3283059597015
22801763119	44.230586528778

Tabla 5.4: Resultados de los tiempos de ejecución del algoritmo Rho Pollard



Gráfica 5.2: Tiempos de ejecución Rho Pollard

La parte central de que tan grande o pequeo es el tiempo de ejecución de este algoritmo, se basa en encontrar un x_0 óptimo, con el cual el algoritmo pueda llegar a una solución, aunque el algoritmo es de orden $\sqrt{p}$, suele ser rápido una vez que encontró un x_0 óptimo. En la gráfica 5.2 se muestran los tiempos de ejecución para los números primos de la tabla 5.2.

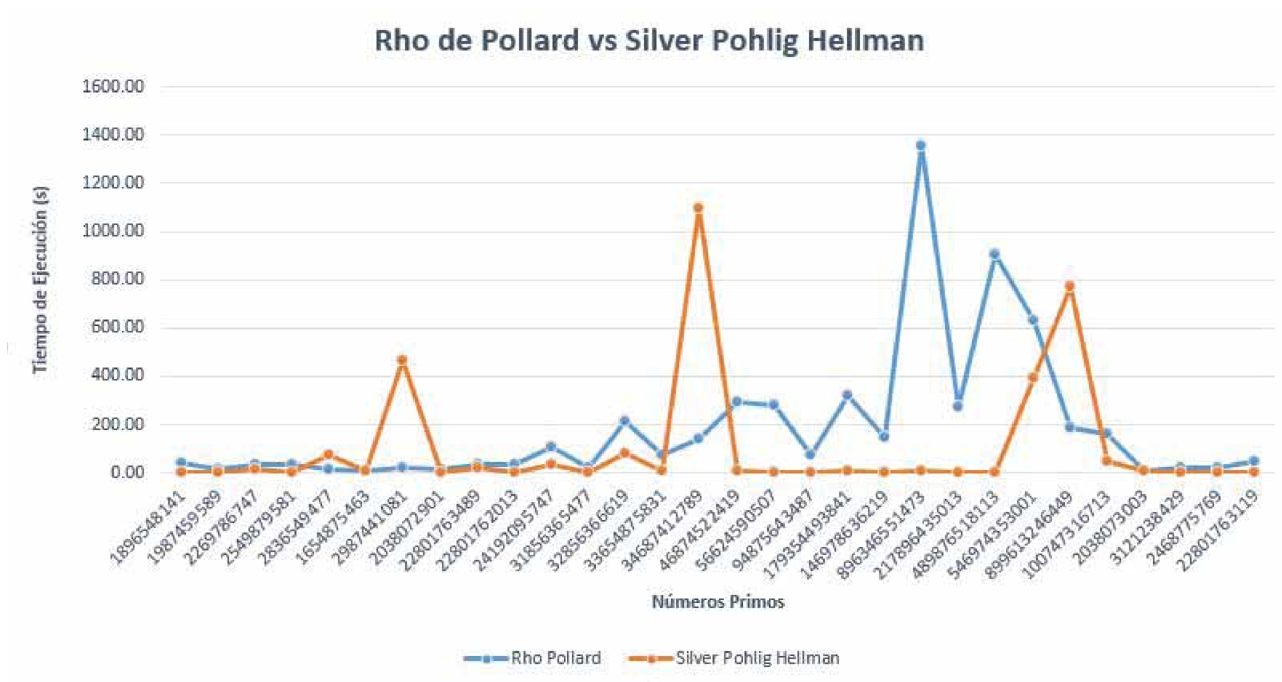
Cabe destacar que este algoritmo requiere un almacenamiento de $\sqrt{p}$, el cual puede ser un problema muy grave ya que la pila podría desbordarse y el algoritmo fallaría. Por otra parte, la búsqueda de betas iguales, llevaría n pasos por iteración. [1]

[1] En el apartado de *Desarrollo del proyecto*, se explica como se resolvieron estos problemas.

Capítulo 6

Conclusiones

6.1 Rho Pollard vs Silver Pohlig Hellman



Gráfica 6.1: Resultados Rho Pollard vs Silver Pohlig Hellman

En la gráfica 6.1 se puede observar que si se conoce la descomposición de $p-1$ en la mayoría de los casos el algoritmo Silver-Pohlig-Hellman es mas rápido solo si los factores de la descomposición de $q-1$ son pequeos en cuanto al número de cifras, este algoritmo es eficiente cuando el número de cifras de sus factores primos es a lo mas la mitad de cifras de

p. Por otro lado se observa que el algoritmo Rho Pollard tarda un poco más, aunque si las cifras de los factores primos de $p-1$ son grandes el tiempo se reduce en comparación con Silver-Pohlig-Hellman.

Al analizar esto, se concluyó que si se conoce la descomposición como primos de $p-1$ y el número de cifras de los duchos factores es pequeo, es recomendable utilizar el algoritmo Silver-Pohlig-Hellman ya que es más eficiente para calcular logaritmos discretos sabiendo lo anterior. De lo contrario, si el número de cifras de los factores primos es grande, es recomendable utilizar el algoritmo Rho Pollard, ya que lo hará de una manera mas eficiente.

A pesar de que originalmente el algoritmo Rho Pollard requiere almacenamiento $\sqrt{p}$ y una búsqueda de orden n para cada cálculo de β, la adecuación que se le hizo a la implementación realizada en este proyecto, reduce ese par de problemas por lo cual los tiempos de ejecución son menores.

Apéndice A

Algoritmo Silver-Pohlig-Hellman

```
"""Algoritmo Silver-Pohlig-Hellman
    Implementado por.
        - Victor Cuauhtemoc Garcia Hernandez
        - Aramast Avedis Beredgiklian Galvan
        - UAM AZCAPOTZALCO"""
from time import *

x_candidato=int(0)
x=int(0)

def mcd(m,n,flag):
    i = int(0)
    r0 = m
    r1 = n
    r2 = int(1)
    T00 = int(1)
    T01 = int(0)
    T10 = int(0)
    T11 = int(1)
    while r2!=0:
        q = int(r0/r1)

        E00 = q
        E01 = int(1)
        E10 = int(1)
        E11 = int(0)

        A00 = int(T00*E00 + T01*E10)
        A01 = int(T00*E01 + T01*E11)
        A10 = int(T10*E00 + T11*E10)
        A11 = int(T10*E01 + T11*E11)

        T00 = int (A00)
        T01 = int (A01)
        T10 = int (A10)
        T11 = int (A11)

```

```
            x= T00
39          y= T10

41          r2 = pow(-1,i)*x*m + pow(-1,i+1)*y*n
            r0 = r1
43          r1 = r2
            i = i+1
45      aux1 = int(pow(-1, i))
        aux2 = int(pow(-1, (i + 1)))
47      xi = int (T01)
        yi = int (T11)
49
        inverse1 = yi*aux1
51      inverse2 = xi*aux2

53      if flag==0:
            regreso = inverse1%m
55      elif flag ==1:
            regreso = inverse2%n
57      return regreso

59  def DivExtends(m,n,flag):
        if n==0 and m ==0:
61          print("No hay GCD")
        elif m==0:
63          print ("El GCD es: ")
            print (abs(n))
65          return int(1)
        elif n==0:
67          print ("El GCD es: ")
            print(abs(m))
69          return int(1)
        elif m!=0 and n!=0:
71          gcd = mcd (m,n,flag)
            return gcd
73
    def fastexp (a,m,n):
75      if a==0:
            powa=0
77          return powa
        elif n==0 and a!=0:
79          powa=1
            return powa
81      elif a==1:
            powa =1
83          return powa
        else:
85          powa=1
            i=1
87          while n>0:
                delta = n%2
89              powa = (powa*pow(a,delta))%m
                n= int((n-delta)/2)
91              a=(a*a)%m
                i=i+1
```

```python
        return powa

def calculo_busqueda(g,lamb,q,p,alpha):
    j=int(0)
    rpj=[]
    qmenosuno = int(q-1)
    xp=[]
    for i in range(alpha): xp.append(-1)
    #C\'alculo de las "r"
    while j <= (p-1):

        n =int(((q-1)/p)*j)
        auxresultado = fastexp(g,q,n)

        j=j+1
        rpj.append(auxresultado)
    j=int(0)
#B\'usqueda de los "x's"
    while j<=(alpha-1):
        p_j = p**(j+1)
        x=fastexp(lamb,q,int(qmenosuno/p_j))
        for i in range(len(rpj)):
            if x == rpj[i]:
                xp[j]=i
                break
        z = (xp[j]*p**j)%q
        auxg = fastexp(g,q,z)
        g_inv_z = DivExtends(auxg,q,1)
        lamb= (lamb*g_inv_z)%q
        j=j+1
    return xp

def escribir_resultados(g,q,x,lamb,tiempo_inicial):
    if fastexp(g,q,x)== lamb:
        f=open('Silver_Pohlig_Hellman.txt','a')
        print("La solucion a la ecuaci\'on")
        f.write("La Soluci\'on a la ecuaci\'on\n")
        print(g,"^",x,"    ",lamb,"mod", q)
        ec=str(str(g)+"^"+str(x)+" = "+str(lamb)+" mod "+str(q)+'\n')
        f.write(str(ec))
        print("x=",x)
        f.write(str("x="+str(x))+'\n')
        tiempo_final =time()
        ejecucion =tiempo_final-tiempo_inicial
        print("Tiempo de Ejecuci\'on",ejecucion)
        f.write(str("Tiempo de Ejecuci\'on "+str(ejecucion)+'\n'))
        f.close()
        return True
    else:
        return False

def configuracion_parametros(linea):
    lamb,g,q,facp,primes,pows = linea.split("#")
    lamb = int(lamb)
    g = int (g)
```

```python
    q = int(q)
    facp=int(facp)
    fprimo=primes.split(",")
    alpha=pows.split(",")
    return lamb,g,q,facp,fprimo,alpha

def Silver_Pohlig_Hellman():
    archivo = open('EntradaSPH.txt')
    for linea in archivo:
        rpj=[]
        tiempo_inicial=time()
        li=[]
        pi=[]
        Mi=[]
        mi=[]
        lamb,g,q,facp,fprimo,alpha = configuracion_parametros(linea)
        for x in range(0,facp):
            primo=int(fprimo[x])
            potencia=int(alpha[x])
            a=calculo_busqueda(g,lamb,q,primo,potencia)
            aux=a[0]
            for x in range(1,len(a)):
                auxx=int((a[x])*(primo**x))
                aux=aux+auxx
            li.append(aux)
            pi.append(primo**potencia)
            print("x     ",aux,"mod ",primo**potencia)

        qmenosuno=int(1)
        for xx in range(0,facp):
            qmenosuno=qmenosuno*pi[xx]

        for xxx in range(0,facp):
            auxMi= int(qmenosuno/pi[xxx])
            Mi.append(auxMi)

        for j in range(0,facp):
            if pi[j]<Mi[j]:
                bandera=int(0)
                auxr =int( DivExtends(pi[j],Mi[j],bandera))
                mi.append(auxr)
            else:
                bandera=int(1)
                auxr =int( DivExtends(Mi[j],pi[j],bandera))
                mi.append(auxr)

        x_candidato=int(0)
        auxx=int(0)

        for i in range (0,facp):
            auxx=int(li[i]*mi[i]*Mi[i])
            x_candidato+=auxx

        x=x_candidato%qmenosuno
```

```
        if not escribir_resultados(g,q,x,lamb,tiempo_inicial):
            print ('Error, no hay logaritmo discreto')
    archivo.close()

Silver_Pohlig_Hellman()
```

Apéndice B

Algoritmo Rho Pollard

```
"""Algoritmo Rho Pollard
    Implementado por.
        - Victor Cuauhtemoc Garcia Hernandez
        - Aramast Avedis Beredgiklian Galvan
        - UAM AZCAPOTZALCO"""

from time import *
import random

pi = int(0)
si = int (0)

def DivExtends(m,n,flag):
    i = int(0)
    r0 = m
    r1 = n
    r2 = int(1)
    T00 = int(1)
    T01 = int(0)
    T10 = int(0)
    T11 = int(1)

    if n==0 and m ==0:
        print("No hay GCD")
    elif m==0:
        print ("El GCD es: ")
        print (abs(n))
        return int(1)
    elif n==0:
        print ("El GCD es: ")
        print(abs(m))
        return int(1)
    elif m!=0 and n!=0:
        m= abs(m)
        n = abs(n)
        while r2!=0:
            q = int(r0/r1)
```

```
            E00 = q
            E01 = int(1)
            E10 = int(1)
            E11 = int(0)

            A00 = int(T00*E00 + T01*E10)
            A01 = int(T00*E01 + T01*E11)
            A10 = int(T10*E00 + T11*E10)
            A11 = int(T10*E01 + T11*E11)

            T00 = int (A00)
            T01 = int (A01)
            T10 = int (A10)
            T11 = int (A11)

            x= T00
            y= T10

            r2 = pow(-1,i)*x*m + pow(-1,i+1)*y*n
            r0 = r1
            r1 = r2
            i = i+1
        aux1 = int(pow(-1, i))
        aux2 = int(pow(-1, (i + 1)))
        xi = int (T01)
        yi = int (T11)

        inverse1 = yi*aux1
        inverse2 = xi*aux2

        if flag==0:
            regreso = inverse1%m
        elif flag ==1:
            regreso = inverse2%n

        return regreso,abs(r0)

def fastexp (a,m,n):
    if a==0:
        powa=0
        return powa
    elif n==0 and a!=0:
        powa=1
        return powa
    elif a==1:
        powa =1
        return powa
    else:
        powa=1
        i=1
        while n>0:
            delta = n%2
            powa = (powa*pow(a,delta))%m
            n= int((n-delta)/2)
```

```
93              a=(a*a)%m
                i=i+1
95          return powa

97  def calcula_beta(betai,g,q,lamb,xi,yi):
        if (betai%3)==0:
99          xi =(xi+1)%(q-1)
            yi = (yi)%(q-1)
101         betai=(fastexp(g,q,xi)*fastexp(lamb,q,yi))%q
            return betai,xi,yi
103     elif (betai%3)==1:
            xi= (2*xi)%(q-1)
105         yi= (2*yi)%(q-1)
            betai=(fastexp(g,q,xi)*fastexp(lamb,q,yi))%q
107         return betai,xi,yi
        elif (betai%3)==2:
109         xi= (xi)%(q-1)
            yi= (yi+1)%(q-1)
111         betai=(fastexp(g,q,xi)*fastexp(lamb,q,yi))%q
            return betai,xi,yi
113
    def escribir_resultados(g,q,nld,lamb,tiempo_inicial):
115     if fastexp(g,q,nld)== lamb:
            f=open('Rho_Pollard.txt','a')
117         print("La solucion a la ecuacion")
            f.write("La Soluci\'on a la ecuaci\'on\n")
119         print(g,"^",nld,"    ",lamb,"mod", q)
            ec=str(str(g)+"^x"+" = "+str(lamb)+" mod "+str(q)+'\n')
121         f.write(str(ec))
            print("x=",nld)
123         f.write(str("x="+str(nld))+'\n')
            tiempo_final =time()
125         ejecucion =tiempo_final-tiempo_inicial
            print("Tiempo de Ejecuci\'on",ejecucion)
127         f.write(str("Tiempo de Ejecuci\'on "+str(ejecucion)+'\n'))
            f.close()
129         return True
        else:
131         return False

133 def busca_betas_iguales(r,inicio,B):
        fijo = B[r-1]
135     for x in range (0,r-1):
            if B[x]== fijo:
137             return x,r-1,False
        return x,r-1,True
139
    def configuracion_parametros(linea):
141     lamb ,g,q = linea.split("#")
        lamb = int(lamb)
143     g = int (g)
        q = int(q)
145     return lamb,g,q

147 def Rho_Pollard():
```

```python
    archivo = open('EntradaRP.txt')
    for linea in archivo:
        terminado = True
        tiempo_inicial = time()
        while terminado:
            B = []
            X = []
            Y = []
            yi = int (0)
            lamb,g,q = configuracion_parametros(linea)
            xi = random.randint(1,(q-1))
            betai = fastexp(g,q,xi)
            B.append(betai)
            X.append(xi)
            Y.append(yi)
            r = 1
            inicio =0
            betx = int(2)
            solucion = True
            while solucion:
                betai,xi,yi= calcula_beta(B[r-1],g,q,lamb,X[r-1],Y[r-1])
                B.append(betai)
                X.append(xi)
                Y.append(yi)
                if len(B) == betx:
                    pi,si,solucion = busca_betas_iguales(betx,inicio,B)
                    inicio = betx
                    betx=betx*2
                r = r+1
            print ('j','                ','Betaj','             ','Xj','                '
    ,'Yj')
            print (pi,'       ',B[pi],'       ',X[pi],'       ',Y[pi])
            print (si,'       ',B[si],'       ',X[si],'       ',Y[si])
            Yfin = abs(Y[si]-Y[pi])
            Xfin = abs(X[pi]-X[si])
            bandera = 1
            auxr,gcd = DivExtends(Yfin,(q-1),bandera)
            nuevomodulo= int((q-1)/gcd)
            nuevax=int(Xfin/gcd)
            nuevay=int(Yfin/gcd)
            auxr, gcd2 = DivExtends(nuevay,nuevomodulo,int(1))
            ld = (auxr*nuevax)% nuevomodulo
            for x in range (1,gcd+1):
                nld = (ld+(x*nuevomodulo))% q
                if escribir_resultados(g,q,nld,lamb,tiempo_inicial):
                    terminado = False
                    break

    archivo.close()

Rho_Pollard()
```

Bibliografía

[1] N. Koblitz, *A Course in Number Theory and Cryptograpghy*, 2nd ed. Springer, 1994.

[2] A. Odlyzko, "Discrete logarithms over finite fields," *CRC Press*, pp. 393–401, 2013.

[3] R. Buchmann, *Introduction to cryptography*, 2nd ed. Springer, 2004.

[4] I. Vinogradov, *An introduction to the theory of numbers.* London & New York: Pergamon Press, 1955.

[5] G. Mullen and D. Panario, *Discrete Mathematics and Its Applications.* CRC Press, 2013.

[6] H. V. Arturo, "Diseño, implementación y comparación de los métodos de encriptación rsa en aritmética modular y massey–omura en curvas elípticas." Proyecto terminal, División de Ciencias Básicas e Ingeniería, Universidad Autónoma Metropolitana Azcapotzalco, México, 2016.

[7] J. A. Hernández Rodríguez, "Diseño, implementación y comparación del método de encriptación elgamal vía aritmética modular y curvas elípticas." Propuesta de proyecto de integración, División de Ciencias Básicas e Ingeniería, Universidad Autónoma Metropolitana Azcapotzalco, México, 2016.

[8] P. F. Flores, "Implementación en lenguaje c de algoritmos de test de primalidad y comparación," Proyecto terminal, División de Ciencias Básicas e Ingeniería, Universidad Autónoma Metropolitana Azcapotzalco, México, 2016.

[9] J. Pollard, "Monte carlo methods for index computations (mod p)," *Math. Comp. 32*, pp. 918–924, 1978.

[10] S. C. Pohlig and M. Hellman, "An improved algorithm for computing logarithms over $\mathbb{GF}(p)$and its cryptographic significance," *IEEE Trans. Inform. Theory IT-24*, pp. 106–110, 1978.

[11] (2016) Calculadora de logaritmos discretos. [Online]. Available: https://www.alpertron.com.ar/LOGDI.HTM

[12] W. Diffie and M. Hellman, "New directions in cryptography," *IEEE Transactions on Information Theory*, pp. 644–654, 1976.