UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

Posgrado en Ciencia e Ingeniería de la Computación

ALGORITMOS DE DESCOMPOSICIÓN DE NÚMEROS ENTEROS EN FACTORES EN LA CRIPTOGRAFÍA MODERNA

T E S I S

QUE PARA OBTENER EL GRADO DE:

MAESTRO EN CIENCIAS (COMPUTACIÓN)

P R E S E N T A:

DAVID TINOCO VARELA

DIRECTOR DE TESIS: VLADISLAV KHARTCHENKO

México D.F. a **2009**

Resumen

El alcance de este trabajo de tesis es presentar algunas singularidades de los algoritmos de factorización de números enteros, sin entrar en conceptos cuánticos. Hasta el momento, este problema es un problema abierto dentro de las matemáticas, particularmente teoría de números, y el cómputo matemático. Los algoritmos de descomposición hasta hoy existentes, no son capaces de llevar a cabo la factorización de enteros en su forma más general.

El conocimiento y entendimiento de estos algoritmos nos permiten tener una idea más clara de como llevar a cabo una encriptación de datos de forma segura. Basándonos en como funciona un algoritmo de descomposición y bajo que circunstancias actúa, podemos definir el tipo de claves que pueden resistir a un ataque de desencriptación. Esto es de suma importancia en el desarrollo de las comunicaciones, en los sistemas económicos y bancarios y en general, toda acción que requiera de una encriptación de datos segura.

Un factor importante en esta rama, son los algoritmos de *pruebas de primalidad*, por lo que también estudiamos algunos de ellos y presentamos sus características especificas.

También proponemos una forma distinta, catalogada dentro de los algoritmos de propósito especifico, de buscar factores primos en números grandes.

Agradecimientos

En el transcurso de mis estudios ha habido muchas personas a las que les quisiera agradecer su ayuda y apoyo, sin los cuales no hubiera sido posible la realización de este trabajo.

A mis padres, que siempre muestran su apoyo hacia mi.

Muy especialmente a mi tía Lupita, que yo se que sin su apoyo esto no hubiera sido posible.

A Xochitl que siempre ha estado ahí presente para darme ánimos y apoyarme.

A todos mis profesores, que son parte de mi desarrollo, muy en particular a aquellos profesores, que, estando dentro y fuera del aula me mostraron su apoyo y me brindaron su experiencia y ayuda en muchas cuestiones relativas a mi trabajo de tesis, no voy a mencionar los nombres de esas personas, pero, créanme que les agradezco bastante.

A José Luis, compañero de posgrado, que siempre me brindo un poco de su tiempo y experiencia, principalmente en las cuestiones paralelas, para resolver algunas dudas y errores que tuve.

A Mayra, compañera de posgrado, que de igual manera, siempre accedió a brindarme su ayuda cuando yo lo requería.

A todas las secretarias tanto de IIMAS como de FES-C, que han estado en la mejor disposición para atender mis dudas o necesidades de la mejor manera posible.

Y principalmente y de forma muy específica quisiera darle las gracias a mi tutor, el Dr. Vladislav Khartchenko, por haberme aceptado como su alumno, por haberme propuesto este tema de tesis, por su tutoría y por todo lo que ha hecho por mí.

Hago una mención especial y respetuosa al Dr. Vladimir Tchijov (q.e.p.d), ya que en su momento también estuvo dispuesto a trabajar conmigo y brindarme su apoyo.

A la UNAM, mi casa de estudios y la cual me ha dado tanto.

A CONACyT por la beca otorgada para la realización de mis estudios.

Al IIMAS y a FES-C, por las instalaciones y facilidades brindadas.

Se que faltan personas por mencionar, pero, no por eso no les estoy agradecido, para todas esas personas que han estado ahí de una u otra forma, gracias.

Índice general

Índice de cuadros

Capítulo 1

Introducción

La criptografía es el arte de ocultar datos que no deben ser vistos salvo por las personas a las que están dirigidos. Existen desde la antigüedad sistemas criptográficos tales como la escítala, utilizada en el siglo V A.C. por los antiguos pueblos griegos o el cifrador de *Polybios*, usado en el siglo II A.C., por mencionar solo algunos[1].

Se dice que la criptografía moderna nació junto con el surgimiento de las computadoras, al permitirnos realizar mayor cantidad de cálculos y de mayor complejidad de una manera rápida y sencilla. Pero existen 2 trabajos fundamentales que hoy en día son considerados como los pilares de la llamada criptografía moderna, el primero de ellos, *Communication Theory of Secrecy Systems*[3], publicado en 1949 por *Claude Shannon*, donde describe un sistema que resiste a cualquier atacante, incluso con poder de cómputo y tiempo infinito, donde la longitud de la clave K tiene que ser, al menos, tan larga como la longitud del texto plano. Y el segundo, publicado en el año de 1976 por *Whitfield Diffie* y *Martin E. Hellman* titulado *New Directions in Cryptography*[4], este trabajo introdujo el revolucionario concepto de la *Criptografía de clave pública* y también nos brinda un nuevo e ingenioso método de intercambio de clave, su seguridad está basada en la intratabilidad del problema del *logaritmo discreto*.

Para 1978 *Rivest, Shamir* y *Adleman* dieron a conocer en *A Method for Obtaining Digital Signatures and Public-Key Cryptosystems*[5], la primera encriptación práctica de clave pública, ahora conocida como criptosistema RSA, este criptosistema está basado en el problema matemático y computacional de factorizar números con factores primos grandes.

Este sistema no sólo brinda gran seguridad de encriptación, sino que, además, es implementado de una forma sencilla y elegante.

Basándonos en este hecho sabemos que la evaluación de algoritmos de factorización de enteros, desde un unto de vista teórico y práctico, es de gran importancia para cualquier persona interesada en la seguridad basada en la factorización de los criptosistemas de clave públic [6], como RSA.

[1] Para un conocimiento mas profundo acerca de todos estos sistemas criptográficos podemos consultar [2].

Hasta el momento de escribir esta tesis no existe ningún algoritmo capaz de poder romper el criptosistema RSA rápidamente en su forma más general, sin embargo, el equipo de investigadores en este campo conformado por *F. Bahr, M. Boehm, J. Franke* y *T. Kleinjung* lograron la factorización del número RSA-200 de 663 bits, el 9 de Mayo del 2005. Este número fue:

27997833911221327870829467638722601621070446786955428537560009929326128400107
60934567105295536085606182235191095136578863710595448200657677509858055761357
90987349501441788631789462951872378692218239 83

Obteniendo los factores primos

35324619344027701212726049781984643686711974001976250364930346877612125367942
3200058547956528088349

79258699544783330333470858414800596877379758573642199607343303414557678728181
52135381409304740185467.

1.1. Resultados

A lo largo del trabajo de tesis se han obtenido muchos datos interesantes, concernientes al mundo de la criptografía moderna, algunos de ellos ya conocidos, pero replanteados con equipos de computo más poderosos y por lo tanto, más veloces en el momento de llevar a cabo la ejecución de los algoritmos, tal es el caso de los test de primalidad, en donde hemos completado un análisis de dichos test y podemos constatar los tiempos de ejecución que lleva cada uno de ellos en equipos de computo actuales, donde contamos, además, con grandes cantidades de memoria volátil y memoria física. Esto nos da una nueva visión de uso de cada uno de esos algoritmos, ya que, de alguna manera podemos trabajar datos con los cuales, anteriormente, no era posible trabajar, o en su defecto requerían demasiado tiempo de ejecución.

Durante este proceso también incursionamos en la búsqueda de números primos grandes, llegando a crear listas enormes de dichos primos, algunos de ellos los mostramos en el apéndice B, un caso especifíco dentro de los primos generados es el número

45300972779019928016266609685804591429494237375618231522249561547492982731849671213488739104205506377933810
76965922758876283988090956930621907221036060031138923318529745279937373480134255903605335649369905324018219
73200823964863650099993899415973572385914626556574966398977305341665574654612643323457481634241059860464381
36800031367652587906770895387754910932827693753003157973144541389753614662268505559721637527943836329833645
09066503692353763075566199591799053281335265832103075317101096177397597368518505252007763982814892890306731
10739957370280705280438573489806816438512700796763605366571613431405129065039540129368756666456676969668732
81917585567800961648145617463921022073641361594492746583987019922437641375365544727463538644807738138897790
14512152970645659664713212683316264233546149896091612946645424469229591082579560842193707370038919305279077
64149439647264314297192413525631755662446790330163502745299953571151726689511875663219604012137713678442002
84782843612036483519255340232684558891274103576516312872716791107026447322747963096152699325409925038556496
42468719030219796981227766877180619618412630254519262645884368081588801481710377041845875681474871630483 25

$$
\begin{aligned}
&969136912600538387635399234932894227581957916119240520827118880131671824186863186773640065060049732226276302761546689275624989940540501379588628542080816394339982755351267352418517275464052362617874578957472332327\\
&610164065516783172958645158882796219784463785668011731450636337209274720389000256349165751520409848562531326421924544282006744604246452993266012487731049936311382722788881225121464902076843396213236013168965092443843625032915052923404236525111582539142667854720927080713791547177531060321248110419669610528416795861090215\\
&585446364433312408092984047678374181926940996331593360789680813757443437777102315115203895185323498940519259391592715192015691127900747667966650248680999147004309343716042330480633495198482712463615199309614948270840\\
&115534579642239146825214373630467601728697445653604712573255397950976732854638916076677147927437378493829864944136283473256863379510526005661844964701279524900985024846875202300560794338982176370852903463081617308680621331913183086292261542540206256042448569703813033236980971830167310613529530012633085756772230687983100\\
&438556952317686178087462657806753822722338495850628452797043307630586148042417144167433527409780279748449065254664030443893253453452006954111706102324976719625464287160176511035117602735174526633850231618822880839503233589673384195722418548521758090404645985890455734937301355896610962031094448681490930422658379288769080\\
&355049888210865075511464691213556357555392709689621405939363129264405629056724183576358030030746862572205532614875904533357667319710677250359360637215839158944575229851868219400184236476995748748991748700093029817126741995910506557772761448103624081349912839369253401967188735665880089696967012926580508433238780884841894\\
&618095610609235523779279903980848271600423720834326468380970143723200033783185118858076164344616289614282132759523373742883741087114044142808431876237940483602806712473583996029238951371414337353314746059750788539153617949900125675642315568808500906081094752438441936639917853717442569876381169736877028680441999336636243\\
&680076015988188762838437740152405340959596007064749321762575724008743528483673432480219079289070592404435221698347210402359360546306149823531419611678979339514784559728979572842731900803992533843073480581286135248785907382255374207316102097544721498938392467580699239262357301001602602839999469082065179992446475951916729\\
&695947564242807404633645887437283904905402267439308943650837775386842569494538368235180590805343489528954239538065863768659734904261166814980847985455762877185063387847544340572620636283233209073356678952476418607896733114179287864450580108971338700605749172035241698285053323608547448553476434401318307149868035176498758\\
&286327855793067198304154184385754972190910955455010437139895947111565681199652928504791629352309027821695163425328040784909706556090033011994484572190654394636780768885527073275325510890921716077811838052187742581915276808647099318510267157974125517459122377692860820123361979083432911939334794784793101296598989822181995\\
&759018766330393413344421818990722626570092156483676937747344414055538777188710518277951471999540208645379941718720016340630907031560664016922381171397180583449175260642324883852889949949929986825167848406256687487863185937461715716555243447776889893429398732732430843901014539902483878054887941363328234963166283793502283\\
&194496059807178964934625025319288078959973216355556523467775817444201921695558352669750870323664702570237202904969364171939936220354332298005246491489613523580546473577345976882771588208245243534978346072420669605124234181910174921236979611995899711669648623868042274709918578361245649212118504073553044296299264582941132\\
&763031576726439883886659024092452497765466885097441160482660702809232871890236412296470188482305015966187537184729450361428022686305772974952710540598513565062680294294428684653504985071128552588007915409962432525701473061028960225252449729931549454556314910346149768086076871676895810585408913314650275430332718355151366\\
&285659663893621094577446239879703239998569348654189381410477480726320969595376923408128882995222082121903157\\
&319320404780082605965448762604999 9
\end{aligned}
$$

que tiene 5277 dígitos decimales de longitud y tarda 74 segundos aproximadamente al realizarle la prueba de primalidad por medio del algoritmo de Miller-Rabin(Algoritmo 6).

En la tesis retomamos el trabajo de *Williams*[7], ya que presenta ejemplos y resultados más tangibles que otros trabajos consultados, encontrando en primer termino que ahora es posible factorizar números más grandes con los algoritmos $p-1$ (Algoritmo 13) y $p+1$ (Algoritmo 14), debido a las potencias de las computadoras actuales, como por ejemplo, los valores 197002597249 y 47635010587 que son factores que en dicho trabajo se mencionan como valores que son encontrados por $p-1$ y no encontrados por el algoritmo $p+1$, y con los nuevos equipos, ambos algoritmos son capaces de encontrarlos sin ningún problema, o los números 187333846633 y 4866979762781, que el trabajo citado muestra como que son valores encontrados por el algoritmo $p+1$ y que no los puede encontrar el algoritmo $p-1$, y que ahora, así como los anteriores números, son capaces de encontrarlos ambos algoritmos (Todos estos experimentos, originalmente se corrieron en una AMDAHL 470-V7).

También dentro de este trabajo, encontramos un factor que se menciona como primo, 328006342461, pero que en realidad es un número compuesto con una factorización den-

tro de los primos dada como $3 \cdot 7^2 \cdot 17 \cdot 131255039$.

Por otro lado hemos hecho un pequeño análisis del uso del GCD cuando filtramos valores que son divisibles entre números primos pequeños $\in \{2, 3, \ldots, 11\}$, tales resultados nos muestran que al filtrar dichos valores estamos aprovechando más de la mitad del tiempo de ejecución que se emplearía si no se eliminaran esos números, claro, esto es fácil de ver a través del teorema de Mertens, y lo podemos observar con más claridad en el cuadro 5.2, donde podemos verificar que existe solo un 19 % de números que no son divisibles por estos valores primos pequeños.

Llegando al algoritmo que encuentra factores cercanos a las raíces de un número n (Algoritmo 17, ver página 65) y el algoritmo 18 (Ver página 68), hemos encontrado factores primos grandes de valores n, algunos de ellos los podemos observar en el cuadro 5.3, como un ejemplo de esto tenemos el valor n

3157504871812953365536297860297641189352804363345861894302085472545165186662878881271590483629198554281309654161198099964228119107019931585224700003277376186728261568101689889604950867498538273994702592365795041672751248613879788232139306094278102199390804697731852513743735888753253345042786690874619312859971905554681011138724083315838937055892389834503391778097535869543154008292685701854020583440886278654802635335505165540639915210734611460534435070870812324453566166194292057624229520651471858543138127567103429885753585188830810361225863841,

que tiene 547 dígitos decimales y presenta una factorización de la forma

119702852789038811***119702852882216971***143287729769879882229936938263431 61***205313735026061214477452724785780565621076249617659494989383263765457***421537297903516755666829713234311571351632864183381275137129370622540711214630082652867363910872676767223988276340223425042975717351727 61***177693693523798231772352187522867317609533265873772093688965030021212224182139226521557173469073380000964857343139651141555757474921409388549420074992568875383518092215384931125112954928025978312365668893567781688069718126428063740231655625168914628109014368235742827263791 3.

Donde el quinto factor tiene 137 dígitos decimales, y se encontró en un tiempo aproximado de 54 minutos trabajando con 32 procesos paralelos.

Estos factores grandes se deben a que, a diferencia del algoritmo $p - 11$ y $p + 1$, este algoritmo no busca factores primos con ciertas características, sino más bien actúa sobre valores compuestos con ciertas características, y a diferencia de los algoritmos de propósito general, este algoritmo no actúa sobre números primos de un determinado tamaño, sino que actúa sobre el tamaño en dígitos del campo de búsqueda.

La implementación serial del algoritmo, con el conjunto de valores analizados, tarda más tiempo que la versión paralela en buscar los factores debido a cada punto que debe escanear, pero este algoritmo presenta una gran cantidad de datos que son independientes entre sí, por tal motivo la implementación paralela se lleva a cabo de una forma sencilla, esperando trabajar con p procesadores para p raíces de n(Ver pag. 74).

Esta forma de búsqueda tiene la desventaja de que sólo servirá para números grandes que tengan sus factores primos cercanos a sus raíces (Para el caso de los números pequeños siempre servirá), puede encontrar factores primos más grandes que los actuales algoritmos, pero solo si n cumple con la característica mencionada.

1.2. Planteamiento

Como podemos darnos cuenta el problema de factorizar un número en sus elementos primos es todavía un campo nuevo[2] y prácticamente inexplorado, por lo que realizaremos el análisis de los distintos algoritmos de factorización, para así tener un conocimiento amplio de las características positivas y negativas de estos algoritmos y sus posibles convenientes e inconvenientes en el momento de intentar factorizar un determinado valor compuesto.

1.3. Metodología

Los algoritmos presentados a lo largo de esta tesis (Exceptuando los del capitulo 5, que ya se explicarán en su momento) fueron realizados en un lenguaje C con el editor Dev-cpp versión 4,9,9,2, compilados y ejecutados en un procesador Core(TM)2 Duo T5550 a 1,83 GHz, bajo el sistema operativo Windows XP profesional con Service Pack 2.

Las versiones paralelas, de los algoritmos presentados a lo largo de este proyecto, no se han retomado en este trabajo de tesis, dichas versiones son tomadas solo como referencias.

El primer problema al que nos enfrentamos es el hecho de que los tipos de datos de C no admiten números grandes, por lo que, para llevar a cabo los algoritmos con números grandes utilizamos la librería GMP, que nos permite trabajar con números tan grandes como la memoria nos lo permita. El manual[8] se puede descargar de la página electrónica http://gmplib.org/.

Dentro de todo el trabajo, un factor importante, es la búsqueda de la información especializada y actualizada. Existen muchas herramientas de búsqueda de información, que presentan una gran parte del presente trabajo de tesis, para la realización de esta tesis utilizamos como medio de apoyo las herramientas de búsqueda de información que a continuación se presentan

1. Las fuentes de investigación actualizada generalmente tienen un costo elevado , pero, en la máxima casa de estudios contamos con el apoyo de la Biblioteca Digital de la Universidad Nacional Autónoma de México, gracias a la cual fue posible tener acceso a todos los artículos que son citados en la tesis.

[2]Aunque es un problema que ha existido desde los antiguos griegos, decimos que es nuevo por que no se han reportados avances realmente significativos y capaces de dar soluciones generales.

2. Quiero hacer mención a Springer, un caso especifico dentro de las bases de datos proporcionadas por la UNAM, ya que a través de esta base de datos se pudieron obtener textos formales que proporcionaron una gran ayuda a la realización de este proyecto.

Los trabajos de investigación requieren estándares que cumplir en su edición y presentación, también es necesario el uso de símbolos y formulas matemáticas. Estos lineamientos pueden provocar dificultades para un procesador de textos convencional, por lo que, esta tesis fue escrita con LaTeX2$_\epsilon$, que ofrece una gran libertad de acción así como una gran calidad tipográfica para la edición de textos científicos y de investigación.

1.4. Organización de la tesis

En el capitulo 2 se da una pequeña introducción a los fundamentos matemáticos útiles para la comprensión de esta tesis, entre los que se encuentra los conceptos de los números primos, así como algunos algoritmos complementarios para la realización de esta tesis.

En el capitulo 3 exploramos los conceptos de los test de primalidad y realizamos un análisis de las características de estas pruebas. Exploramos las características de números especiales y su comportamiento.

En el capitulo 4 realizamos una revisión y un análisis de los algoritmos de descomposición de valores compuestos en sus factores primos, sus ventajas y sus desventajas. Para esto hemos retomado algunos valores que han sido tratados por otros autores.

En el capitulo 5 proponemos una forma diferente de realizar búsquedas de factores primos para números enteros con características particulares.

En el capitulo 6 presentamos los avances que se han obtenido en la factorización de enteros en la práctica y como se han llevado a cabo estas factorizaciones.

Capítulo 2

Conceptos fundamentales de la teoría de números

La teoría de números es una parte fundamental en el aprendizaje y desarrollo de sistemas criptográficos, sistemas de factorización de números enteros y pruebas de primalidad. En este capitulo damos una revisión de los principios básicos de uso general en teoría de números para la realización de los sistemas de pruebas de primalidad y factorización de enteros, basándonos principalmente en [9], [10] y [11] para la realización de este capitulo. Debido al gran uso de estos teoremas no nos detendremos en dar las demostraciones ya que no es parte esencial de este trabajo, solo se presentaran las demostraciones de los teoremas que tengan una alta repercusión en la finalidad de la tesis.

2.1. Primos y el teorema fundamental de la aritmética

Decimos que un entero a es factor de otro entero b(Se escribe $a|b$ y se pronuncia a divide a b o a es divisor de b) si $b = ac$ para cualquier entero c. Un primo (positivo) es un entero $p > 1$ que no tiene factores positivos además de 1 y p, mientras un *c*ompuesto es un entero $n > 1$ que tiene factores diferentes de 1 y p: $n = ab$ cuando $a > 1$ y $b > 1$.

2.1.1. Teoremas fundamentales de aritmética

Teorema 2.1 *Todo entero* > 1 *tiene una factorización dentro de los números primos:* $n = p_1^{n_1} p_2^{n_2} \dots p_k^{n_k}$,*donde* $p_1 < p_2 < \dots < p_k$ *son primos y* $n_1, \dots, n_k$ *son enteros* > 0, *mientras* $k \geq 1$. *Además* p_i *y el* n_i *son únicamente determinados por* n.

$$n = p_1^{n_1} p_2^{n_2} \dots p_k^{n_k} = \prod_{i=1}^{k} p_i^{n_i} \tag{2.1}$$

La factorización en el teorema 2.1 es referida como *factorización de potencia de primos* de n. Según el teorema 2.1, el número 1 no tiene factorización de potencia de primos, por

lo que podemos darle un 1 artificial tomando $k = 0$ (factorización vacía) y declaramos que el lado derecho es 1. Este artificio no nos causará ningún problema.

El producto de cada uno de los valores n es único hasta en el orden que son acomodados sus factores primos. Por ejemplo $16200 = 2^3 \cdot 3^4 \cdot 5^2$, $72000 = 2^6 \cdot 3^2 \cdot 5^3$, $765234125341898321765923562395823 = 317 \cdot 5801 \cdot 4161330420796035502232692 19$ y $101 = 101$.

Teorema 2.2 *Un primo p es un factor de $n \Leftrightarrow p$ que pertenece a los p_i definidos en el teorema 2.1. Si p y q son primos distintos y $p|n$, $q|n$, entonces $pq|n$. Lo mismo se mantiene para 3 o más primos distintos.*

Por supuesto los números negativos también pueden factorizarse. Si queremos factorizar $n < -1$, factorizamos $-n$ y ponemos el signo de menos en la parte frontal por ejemplo: $-200 = -2^3 \cdot 5^2$. Un entero $p < 0$ es declarado primo si y sólo si $-p$ es primo. Quizá la mejor manera de pensar de todo esto es decir que 1 y -1 son unidades en Z (Conjunto de enteros), y p es primo si y sólo si no es unidad y para cualquier factorización de p, $p = ab$ cuando a y $b \in \mathbb{Z}$ y a o b es unidad.

Cuando tenemos 2 números a y b a considerar es frecuentemente conveniente usar los mismos primos en la descomposición, condicionando $n_i \geq 0$; note la cualidad en el teorema 2.1 entonces solamente aplicamos primos con potencia> 0 e incluimos solamente primos que están en a o b o ambos. Por ejemplo,

$$28 = 2^2 \cdot 5^0 \cdot 7^1, y, 200 = 2^3 \cdot 5^2 \cdot 7^0 \tag{2.2}$$

Si $a = p_1^{a_1} p_2^{a_2} \ldots p_k^{a_k}$ y $b = p_1^{b_1} p_2^{b_2} \ldots p_k^{b_k}$ tenemos $ab = p_1^{a_1+b_1} p_2^{a_2+b_2} \ldots p_k^{a_k+b_k}$ y $a_i + b_i > 0$ para cada i. Tenemos un primo. Entonces $p|ab$ si y sólo si p esta en p_i.

Teorema 2.3 *Si p es primo y $p|ab$ entonces $p|a$ o $p|b$, o ambos.*

Esto es muy importante, no debe ser malinterpretado como diciendo que, para cualquier n si $n|ab$ entonces $n|a$ o $n|b$. Como prueba de esto tenemos $4|2 \cdot 6$, pero 4 no divide ni a 2 ni a 6. En el teorema 2.3, todos los valores p son primos.

Teorema 2.4 *$a|b$ si y sólo si cualquier primo dentro de la descomposición de la potencia de primos de a ocurre también en la de b, y la potencia dentro de a es $\leq$ que en b.*

Vemos que una "factorización" de un número n es justamente un expresión $n = ab$ donde $a > 1$ y $b > 1$. La completa factorización de n puede ser sumamente difícil de establecer.

Si los factores primos son lo suficientemente largos, entonces, basandonos en las estimaciones de velocidad computacional que existen en la actualidad, necesitaríamos tomar

mucho más tiempo que la edad del universo para factorizar por medio del juicio de división.

Todos los primos largos conocidos hoy en día también son de la forma $2^n - 1$ y son conocidos como *primos de Mersenne*[1].

Fermat 1601−1665 conjetura que todos los números de la forma $2^{2^n}+1$ (Llamados números de Fermat y representados como F_n) pueden ser primos, pero solo se han probando los números $n = 1, 2, 3, 4$ como primos, ya que en 1732 Euler descubrio que 641 es un divisor de ${F_5}^2$.

Teorema 2.5 *Hay una cantidad infinita de números primos.*

Prueba.(Euclides) Existe, sin duda, al menos un número primo, suponemos que $p_1 = 2 < p_2 = 3 < \cdots < p_r$ son todos los números primos. Ahora tenemos un valor $P = p_1 p_2 \cdots p_r + 1$ y tenemos que p es un número primo que divide a P, entonces p no puede ser ninguno de los valores $p_1, p_2, \ldots, p_r$, de otra manera p puede dividir la diferencia $P - p_1 p_2 \cdots p_r = 1$, lo que es imposible. Entonces este valor p es otro número primo, y p_1, p_2, p_r no serían todos los números primos existentes. Por lo tanto siempre existe un número infinito de primos.
◇

La prueba de Euclides es muy simple, sin embargo, no nos da ninguna información acerca del nuevo número primo, solamente nos dice que será igual al número P.

Como una manera de descubrir nuevos primos, el procedimiento empleado en la demostración es menos que exitoso. Por ejemplo tenemos $2 \cdot 3 \cdot 5 \cdot 7 \cdot 11 + 1 = 2311$ que es primo, y lo sumamos a la lista $2 \cdot 3 \cdot 5 \cdot 7 \cdot 11 \cdot 2311 + 1 = 5338411$, este valor obtenido es un valor compuesto que es factorizado como $13 \cdot 19 \cdot 21613$.

2.1.2. Prueba del teorema fundamental 2.1

Primero, mostramos las factorizaciones existentes. Empezamos con $n > 1$, si n es primo entonces hemos terminado; de otra manera $n = ab$ donde $a > 1$ y $b > 1$. Aplicando el mismo argumento en a y en b cada uno entre los 2 es primo o, si no, es producto de 2 números, ambos> 1. Los otros números que envuelven primos en la expresión para n son > 1 y decrecen en cada paso. Por lo tanto eventualmente todos los números deben ser primos.

Ahora, llegamos a la parte difícil: cualidad única. Supón que el teorema es falso y dejamos el menor número > 1 por lo que el teorema falla al ser n. De esa manera

[1]Datos interesantes de estos números pueden ser revisados en las direcciónes electrónicas http://www.mersenne.org/ y http://primes.utm.edu/mersenne/index.html

[2]Hasta el momento se han realizado muchos esfuerzos para encontrar las factorizaciones de los números de Fermat, en http://www.prothsearch.net/fermat.html podemos revisar algunos datos de las factorizaciones realizadas a estos números.

$$n = p_1 p_2 \dots p_r = q_1 q_2 \dots q_s \tag{2.3}$$

donde p_r y q_s son primos. Claramente r y s deben ser> 1(de otra forma n es primo, o un primo es igual al compuesto). Si por ejemplo p_1 es uno de los q_s, entonces n/p_1, puede tener 2 expresiones como un producto de primos, pero $n/p_1 < n$ así este puede contradecir la definición de n. Por lo tanto q_s no es igual a cualquiera de los p_r y similarmente ninguno de los p_r es igual a cualquiera de los q_s. Podemos suponer $p_1 < q_1$, definido:

$$N = (q_1 - p_1)\, q_2 q_3 \dots q_s = p_1 \left(p_2 p_3 \dots p_r - q_2 q_3 \dots q_s\right) \tag{2.4}$$

Naturalmente $1 < N < n$, así N es únicamente factorizable dentro de los primos. Sin embargo p_1 *no divide* $(q_1 - p_1)$, puesto que $p_1 < q_1$ y q_1 es primo, por lo tanto las expresiones de arriba para n tienen un componente p_1 y el otro no. Estas contradicciones prueban el resultado: no puede haber excepciones al teorema.

Existen muchas pruebas para el teorema 2.1, que podemos revisar en [12]. Existen también, otro tipo de pruebas como la que nuestra *Björn von Sydow* [13], en donde realiza la demostración a través del sistema **ALF**[3].

2.2. Congruencias: propiedades básicas

La idea detras de las congruencias es muy simple, la notación fue introducida por C. F. Gauss(1777-1855) en su trabajo *Disquisitiones Arithmeticae* que fue publicado en 1801. Recientemente *Maarten Bullynck* nos muestra una clase de problemas matemáticos elementales que implican división y residuo, que tienen una larga y culturalmente rica historia desde el siglo XIII hasta el siglo XVIII en [14] publicado en 2009.

Definición 2.1 *Tenemos a, b y m que son enteros, con $m \neq 0$. Definimos $a \equiv b (mod\ m)$ en el sentido que $m|(a - b)$. Este número m es llamado el modulo.*

Dados $m > 0$ y a, podemos escribir $a = qm + b$ para únicamente determinar b con $0 \leq b < m$, claramente $q = [a]$ y $a \equiv b (mod\ m)$. A este número b lo determinamos como *mínimo residuo positivo* de $a\ mod\ m$. (Pero el nombre de *mínimo residuo no negativo* podría ser mejor, por lo que podemos incluir al 0).

2.2.1. Propiedades de $\equiv$

1. $a \equiv b\ (mod\ m)$ es siempre verdadera si $m = 1$.

[3]ALF es un lenguaje de programación que combina los paradigmas lógico y funcional. Para más referencias dirigirse a http://www.informatik.uni-kiel.de/ mh/systems/ALF.html

2. Si $a \equiv b$ y $b \equiv c \ (mod\ m)$, entonces $a \equiv c \ (mod\ m)$. Para $a - b = km; b - c = jm$ implica que $a - c = (k + j)m$.

3. Si $a \equiv b$ y $c \equiv d \ (mod\ m)$ entonces $a + c \equiv b + d$ y $a - c \equiv b - d \ (mod\ m)$. Para $a - b = km,\ c - d = jm$ implica que $(a + c) - (b + d) = (k + j)m$.

4. Si $a \equiv b \ (mod\ m)$ entonces, para algun entero c, $ac \equiv bc \ (mod\ m)$. Para $a - b = km$ implica que $ac - bc = kcm$.

5. Si $a \equiv b$ y $c \equiv d \ (mod\ m)$, entonces $ac \equiv bd \ (mod\ m)$. Para $ac \equiv bc$ y $bc \equiv bd$. Usando la propiedad 4. Ahora usamos la propiedad 2. Note el corolario: Si $a \equiv b \ (mod\ m)$ entonces $a^k \equiv b^k \ (mod\ m)$ para cualquier enetero $k \geq 1$.

6. Si $ac \equiv bc \ (mod\ m)$ y $(c, m) = d$, entonces $a \equiv b \ (mod\ m/d)$. En particular si c y m son coprimos(por lo tanto $d = 1$); c puede ser cancelado.

7. Si $a \equiv b \ (mod\ m)$ y $a \equiv b \ (mod\ n)$, entonces $a \equiv b \ (mod\ [m, n])$, donde $[m, n]$ es el mcm de m y n. En particular si m y n son coprimos, entonces $a \equiv b \ (mod\ mn)$.

8. Si $a \equiv b \ (mod\ m_i)$ para $i = 1, \ldots, r$, cuando $(m_i, m_j) = 1$ para $i \neq j$, entonces $a \equiv b \ (mod\ m_1 m_2 \ldots m_r)$.

9. $a \equiv b \ (mod\ m)$ si y solo si $ac \equiv bc \ (mod\ mc)$.

Proposición 2.1 *Suponemos que los enteros x, y, n satisfacen $x^2 \equiv y^2 (mod\ n)$, pero que $x \neq y,\ x \neq -y (mod\ n)$. Entonces $(x - y, n)$ es un factor de n. Lo mismo se aplica a $(x + y, n)$.*

Prueba. Por supuesto $h = (x - y, n)$ es un factor de n. Ahora $n|(x - y)(x + y)$. Si $h = 1$ entonces se deduce que $n|(x + y)$, que es, $x \equiv -y (mod\ n)$, contrariamente a la hipotesis. Si $h = n$, entonces $n|(x - y)$ ya que $h|(x - y)$, esto es, $x \equiv y (mod\ n)$, nuevamente contrario a la hipótesis. Así h es el correcto factor de n.
$\diamond$

2.3. Máximo común divisor y mínimo común múltiplo

Suponemos que damos 2 enteros positivos a y b; ¿Cómo podemos determinar su común divisor, que es, el número d, por el que $d|a$ y $d|b$?. En principio la respuesta es dada por el teorema 2.1: si $a = p_1^{a_1} p_2^{a_2} \ldots p_k^{a_k}, b = p_1^{b_1} p_2^{b_2} \ldots p_k^{b_k}$ con a_i y $b_i \geq 0$, entonces $d|a$ y $d|b$ si y solo si $d = p_1^{d_1} p_2^{d_2} \ldots p_k^{d_k}$ donde $d_i \leq a_i$ y $d_i \leq b_i$, tal que, $d_i \leq min\,(a_i, b_i)$.

Naturalmente el máximo de este común divisor es el común divisor h dado por $d_i = (a_i, b_i)$. Si a o b es un entero negativo entonces necesitamos insertar un sigo negativo antes de uno

u otro número descompuesto en su *potencia de primos* y un signo $\pm$ anteponiéndolo a d. El máximo valor de d está entonces dado con un signo + y el mismo antes que d_i. Note que si decimos que $a = 1$ todos los a_i son 0 y $h = 1$.

Definición 2.2 *El común divisor h mencionado, donde la potencia de p_i es el $min(a_i, b_i)$ es llamado el máximo común divisor (gcd, por sus siglas en ingles) de a y b. Esto también es conocido como el factor común alto (hcf, por sus siglas en ingles) de a y b. Note que tiene la propiedad de que cualquier común divisor, es un factor de h, si $d|a$ y $d|b$ entonces $d|h$. La notación es $h = gcd(a, b)$ o solo $h = (a, b)$. Dos enteros a y b con $(a, b) = 1$ son llamados coprimos o mutuamente primos o primos relativos.*

Por supuesto$(a, b) = (b, a)$ y $(a, 1) = |a|$ para cualquier valor no cero de a o b. Si queremos un significado para $(a, 0)$ este es usualmente tomado para ser el valor absoluto $|a|$ de a, a menos que $a = 0$ cuando no hay un significado asignado. A continuación mostramos unas útiles propiedades del gcd.

- Si p es primo entonces $(p, a) = p$ cuando $p|a$ y $(p, a) = 1$ de otra manera. Además si p no divide a a entonces $(p^n, a) = 1$ para cualquier $n \geq 1$.
- Si $(a, b) = h$ entonces $(a/h, b/h) = 1$.
- Si $a|bc$ y $(a, b) = 1$ entonces $a|c$.
- Si p es primo $p^n|ab$ y $p \not| (a, b)$, entonces $p^n|a$ o $p^n|b$. Si $(a, b) = 1$ entonces la hipótesis $p \not| (a, b)$ puede caerse.

Tenemos que p_i denota el i esimo primo ($p_1 = 2, p_2 = 3$, etc.) y tenemos que $p_k < n < p_{k+1}^2($ para cualquier $k \geq 1$). Decimos que m denota el producto de los primeros k primos y tenemos que $m = qn + r$ donde $0 \leq r < n$, así r es el resto en la división de m por n. (Por ejemplo, $7 < 101 < 11^2 (k = 4)$ y $m = 210 = 2 \cdot 101 + 8$). Asumimos que $r \neq 0$.

Proposición 2.2 *Suponiendo que $(a, b) = 1$ y que $d|ab$. Entonces existen números únicos d', d'' con $d = d'd'', d'|a, d''|b$. Es automáticamente cierto que $(d', d'') = 1$. De manera opuesta, si $d'|a$ y $d''|b$, entonces $d'd''|ab$, así los divisores de ab están en uno a uno correspondiendo con los pares de divisores de a y b.*

Tenemos, como antes, $a = p_1^{a_1} p_2^{a_2} p_k^{a_k}$, $b = p_1^{b_1} p_2^{b_2} p_k^{b_k}$,con a_i y $b_i \geq 0$. Si tomamos el producto de las potencias de los primos $\ell = p_1^{\ell_1} p_2^{\ell_2} p_k^{\ell_k}$, con $l_i = max(a_i, b_i)$ obtenemos un número ℓ el que es un múltiplo común de a y b. Además cualquier múltiplo de a o b debe tener esos números o al menos, esas potencias, así es un múltiplo de ℓ si $a|m$ y $b|m$ entonces $\ell|m$.

El número ℓ es llamado el *mínimo común múltiplo* de a y b, denotado lcm(a,b) o [a,b]. Si a o b es < 0 tomamos $[a, b] > 0$, éste es el mínimo (positivo) común múltiplo. Puesto que $max(a_i, b_i) + min(a_i, b_i) = a_i + b_i$ el siguiente resultado es inmediato.

Proposición 2.3 *Para valores no cero de a, b $mcm(a, b) \times GCD(a, b) = |ab|$.*

2.4. Algoritmo de Euclides para el GCD

La implementación del GCD es ampliamente usada en cálculos racionales, criptografía de clave pública o álgebra computacional. Muchos algoritmos para calcular el gcd han sido diseñados desde el algoritmo de Euclides, en [15] podemos encontrar una comparación de la eficiencia de algunos algoritmos para el calculo del GCD, realizado por *Jebelean*. La mayoría de los algoritmos para encontrar el GCD calculan una secuencia de residuos por medio de divisiones sucesivas [16].

Recientemente se ha encontrado que el algoritmo de Euclides también sirve para la decodificación de Códigos Reed-Solomon[4] como lo muestra *Priti Shankar* en [17].

Proposición 2.4 *Para cualquier entero* $k, (a, b) = (a + kb, b)$.

*Prueba.*El camino simple para comprobar esto es mostrar que el conjunto de comunes divisores de $a + kb$ y b coinciden con el conjunto de comunes divisores de a y b, que es, $d|a$ y $d|b \Leftrightarrow d|(a + kb)$ y $d|b$. Este factor es más o menos inmediato por la definición de divisor. Por supuesto tiene la consecuencia que el *máximo* común divisor de a y b es igual que el de $a + kb$ y b.
◇

La propiedad de la división, o división algorítmica nos dice que, tenemos a y b como enteros con $B \neq 0$. Entonces existen enteros únicos q y r (el cociente y el resto) que cumplen:

$$a = bq + r, \quad 0 \leq r \leq |b| \tag{2.5}$$

Usualmente aplicamos la división cuando $b > 0$ y entonces el residuo no-negativo r es menor que el número por el que divides.

2.4.1. Algoritmo de Euclides para el GCD de a y b

Tenemos que a y b son enteros con $b > 0$ (ya que $(a, b) = (-a, -b)$). Aplicamos la propiedad de la división repetidamente, como sigue

[4]Reed-Solomon es un código cíclico no binario para la detección y corrección de errores de la información transmitida sobre un canal de comunicaciones. Este tipo de código pertenece a la categoría FEC (*Forward Error Correction*), es decir, corrige los datos alterados en el receptor y para ello utiliza unos bits adicionales que permiten esta recuperación a posteriori, utilizado en áreas como CD´s, telefonia móvil y sondas espaciales.

$$
\begin{aligned}
a &= bq_1 + r_2, && 0 \le r_2 < b; && 330 = 140 \cdot 2 + 50;\\
b &= r_2q_2 + r_3, && 0 \le r_3 < r_2; && 140 = 50 \cdot 2 + 40;\\
r_2 &= r_3q_3 + r_4, , && 0 \le r_4 < r_3; && 50 = 40 \cdot 1 + 10;\\
&\ldots && \ldots && 40 = 10 \cdot 4\\
r_{n-2} &= r_{n-1}q_{n-1} + r_n, && 0 \le r_n < r_{n-1};\\
r_{n-1} &= r_nq_n.
\end{aligned}
$$

Los residuos $r_2, r_3, \ldots$ todos son ≥ 0 y decrecen constantemente, asi pueden eventualmente convertirse en cero. Repitiendo la aplicación de la proposición 2.4 mostramos que $(a, b) = (r_2, b) = (r_2, r_3) = \ldots = (r_{n-1}, r_n) = r_n$, dado que en el factor $r_n|r_{n-1}$. Por lo tanto el GCD de a y b es el último residuo en el proceso repetido de la división.

En el ejemplo, $(98703475982345923874572345000, 876587623578976345687872438 08) = 8$, este valor fue obtenido sin que se factoricen ambos números. Lo que hicimos, fue dividir un número dentro de otro hasta encontrar el cociente y el residuo. Afortunadamente los computadores están bien equipados para realizar estas operaciones. Cuando $b > 0$ tenemos.

$$a = bq + r, \;\; 0 \le r < b \quad \textit{implica} \quad \frac{a}{b} = q + \frac{r}{b}, \; 0 \le \frac{r}{b} < 1. \tag{2.6}$$

Consecuentemente q es sólo "parte del número entero" de a/b. Con mayor precisión, q es el próximo entero menor de a/b, o a/b así mismo si éste es un entero. Esta cantidad es llamada la parte entera de a/b:

Definición 2.3 *La parte entera de un número real x denotada $[x]$, es el máximo entero $\leq x$.*

Aquí se encuentra otra notación común para la parte entera, *la función piso*, denotada $\lfloor x \rfloor$, que contrasta con la *función techo*, denotada $\lceil x \rceil$, la cual es el mayor entero $\geq x$, a menos que x sea asimismo un entero en el cual $\lceil x \rceil = x$.

En la división $a = bq + r$, con $b > 0$, tenemos $q = [a/b]$. (El residuo r es en cualquier momento llamado *el residuo positivo menor de a mod b* y denotado $\langle a \rangle_b$.)

Proposición 2.5 *Tenemos $(a, b) = h$. Entonces existen enteros s y t tal que $as + bt = h$.*

En el método mencionado arriba para calculo de s y t no es ideal desde un punto de vista computacional ya que éste inicia en el fin del algoritmo de Euclides y trabaja volviendo al principio. La ecuación dada en la proposición 2.5 también puede ser escrita como

$bt \equiv h \; mod \; a$ [17].

Corolario 2.1 *$(a, b) = 1$ si y sólo si existen enteros s y t tal que $as + bt = 1$.*

Prueba.$(a, b) = 1$ implica que s y t existan por la proposición 2.5.Sí $as + bt = 1$ y $d|a,\ d|b$, entonces $d|(as + bt)$ asi $d|1$ por lo tanto $d = \pm 1$. Por lo tanto el GCD de a y b es 1. $\diamond$

Con la llegada del paradigma de computo paralelo, muchos de los algoritmos existentes del gcd han sido analizados nuevamente, esperando poder encontrar su implementación en forma paralela [18], es así como en 1984 *Kannan, Miller y Rudolph* nos muestran una implementación paralela del algoritmo de Euclides para calcular el GCD de 2 números enteros [18], encontrando un tiempo de ejcución de $O(n \; log \; log \; n/log \; n)$ trabajando con una cantidad de $n^2(log \; n)^2$ procesadores trabajando de forma paralela. Recientemente *Sidi Mohamed Sedjelmaci* propone un nuevo algoritmo paralelo para el calculo del GCD [19], proponiendo un nuevo paso de reducción que es fácilmente obtenido de los primeros $O(log \; n)$ bits significantes de las entradas. Y basándose en esta reducción nuevos algoritmos secuenciales y paralelos han sido diseñados, su complejidad temporal es $O_\epsilon(n/log \; n)$ usando solamente $n^{1+\epsilon}$ procesadores.

2.4.2. Inversos mod m y soluciones para congruencias ciertas

b es llamado el inverso de $a \; (mod \; m)$ y que podemos escribir $b \equiv a^{-1} \; (mod \; m)$. O también podemos escribir que $b \cdot a^{-1} = 1$.

Proposición 2.6 *Dado a y m, existe b tal que $ab \equiv 1(mod \; m)$ si y sólo si $(a, m) = 1$. Cuando b existe, este es también coprimo con m, y es único (mod m). La última afirmación significa: si $ab \equiv ab' \equiv 1(mod \; m)$ entonces $b \equiv b'(mod \; m)$.*

*Prueba.*Si $ab \equiv 1(mod \; m)$, entonces $ab = 1 + km$, así que cualquier factor común de a y m debe dividir 1, lo que da $(a, m) = 1$, tanbien $(b, m) = 1$. Por el contrario, si $(a, m) = 1$, entonces, por el algoritmo de Euclides, existen enteros b y k tal que $ab + mk = 1$. Para esta b, $ab \equiv 1(mod \; m)$. $\diamond$

Si p es primo, entonces cualquier número que no sea múltiplo de p tiene inverso (mod p)(Esta expresión en términos de teoría de grupos dice que los números $1, 2, \ldots, p - 1$ forman un grupo bajo la multiplicación mod p).

Douglas R. Stinson [20] nos muestra la implementación de un sencillo algoritmo para encontrar el inverso multiplicativo de $a \; (mod \; m)$, mostrado como algoritmo 1.

El algoritmo 1 sólo nos proporciona el valor de a^{-1} cuando la multiplicación de $b \cdot a^{-1} = 1$, por ejemplo la congruencia

Algoritmo 1 Algoritmo para encontrar el inverso multiplicativo de a $(mod\ m)$

Entradas: Valor del modulo a y elemento a invertir b.
Salida: Valor invertido de b.
$a_0 \leftarrow a$
$b_0 \leftarrow b$
$t_0 \leftarrow 0$
$t \leftarrow 1$
$q \leftarrow \left\lfloor \frac{a_0}{b_0} \right\rfloor$
$r \leftarrow a_0 - qb_0$
while $r > 0$ **do**
    $temp \leftarrow (t_0 - qt)\ mod\ a$
    $t_0 \leftarrow t$
    $t \leftarrow temp$
    $a_0 \leftarrow b_0$
    $b_0 \leftarrow r$
    $q \leftarrow \left\lfloor \frac{a_0}{b_0} \right\rfloor$
    $r \leftarrow a_0 - qb_0$
end while
if $b_0 \neq 1$ **then**
    b no tiene inverso modulo a
else
    Regresa t
end if

$$98234609832749238460913287462346897612346 3 \equiv 1(mod\ 87345876198235615686871),$$

el algoritmo 1 calcula rápidamente el valor $a^{-1} = 16974968995937598221322$, como vemos el valor de $b \equiv 1\ (mod\ m)$ tiene inverso por que $(m, b) = 1$, pero si $(m, b) \neq 1$, no existiría el inverso multiplicativo como en el caso de

$$8234986981345098768134983458999234 60 \equiv 1(mod\ 8713498613529183475),$$

donde $(m, b) = 5$.

En ciertas ocasiones, será necesario que podamos obtener un valor de a^{-1} cuando la multiplicación de $b \cdot a^{-1}$ sea diferente de 1, es decir $b \equiv c\ (mod\ m)$ y $c = b \cdot a^{-1}$, donde $c \neq 1$ para este caso sólo basta con sumarle al algoritmo 1 las siguientes líneas:

```
if(bo = 1)
{
     t=a-(-t)
     t=(t*c)
     t=t mod a
}
```

donde $c = b \cdot a^{-1}$ y devuelve el valor de t.

Como ejemplo, tenemos la siguiente congruencia

$$92137491040717419832847612351685 23 \equiv 8712359162598216 37(mod\ 869123656780981624901),$$

la cual nos devuelve un valor de $a^{-1} = 643479983026606380675$

Otro algoritmo sencillo en su implementación,es el algoritmo 2, presentado por *Sattar J. Aboud* [21], que consta sólo de unas cuantas líneas de código.

Algoritmo 2 Algoritmo de Sattar J.A. para encontrar el inverso multiplicativo

1: **Entradas**: Valor del modulo a y el elemento a invertir b.
2: **Salida**: Valor invertido de b.
3: $d \leftarrow 1$
4: **repeat**
5: $\quad d \leftarrow (d + a)/b$
6: **until** d sea entero
7: $b^{-1} = d$

2.5. Teorema chino del residuo

El teorema chino del residuo(TCR) es una inteligente, y muy vieja idea a partir de la cual se puede deducir un valor sobre la base de sus residuos. El TCR, fue descrito por *Sun-Tzi* en el primer siglo d.c.

Teorema 2.6 *Tenemos m y n que son enteros positivos, para cualquier entero a y b existe un entero x tal que*

$$x \equiv a \ (mod\ m) \tag{2.7}$$

y

$$x \equiv b \ (mod\ n) \tag{2.8}$$

si y sólo si

$$a \equiv b \ (mod(m, n)). \tag{2.9}$$

Si x es una solución de las congruencias 2.7 y 2.8, entonces el entero y es también una solución si y sólo si

$$x \equiv y \ (mod\ [m, n]). \tag{2.10}$$

*Prueba.*Si x es una solución de la congruencia 2.7, entonces $x = a + mu$ para algún entero u. Si x es también una solución para la congruencia 2.8, entonces

$$x = a + mu \equiv b \ (mod\ n). \tag{2.11}$$

esto es

$$a + mu = b + nv \tag{2.12}$$

para algún entero v y tenemos que

$$a - b - nv - mu \equiv 0 \ (mod\ (m, n)). \tag{2.13}$$

Y viceversa, si $a - b \equiv 0 \ (mod(m, n))$, existen enteros u y v tales que $a - b = nv - mu$. Entonces $x = a + mu = b + nv$ es una solución de las dos congruencias. $\diamond$

Un entero y es otra solución de las congruencias si y sólo si

$$y \equiv a \equiv x \ (mod\ m) \tag{2.14}$$

y

$$y \equiv b \equiv x \ (mod\ n), \tag{2.15}$$

esto es, si y sólo si $x - y$ es un múltiplo común de m y n,o, de forma equivalente, $x - y$ es divisible por el mínimo común múltiplo $[m, n]$, esto completa la prueba.

Ejemplo Tenemos el sistema de congruencias

$$x \equiv 5 \ (mod\ 21)$$
$$x \equiv 19 \ (mod\ 56),$$

tiene solución, ya que $(56, 21) = 7$ y $19 \equiv 5 \, (mod\ 7)$.

El entero x es una solución si existe un entero u tal que

$$x = 5 + 21u \equiv 19 \ (mod\ 56).$$

esto es

$$21u \equiv 14 \ (mod\ 56),$$
$$3u \equiv 2 \ (mod\ 8)$$
$$u \equiv 6 \ (mod\ 8).$$

Entonces

$$x = 5 + 21u = 5 + 21(6 + 8v) = 131 + 168v$$

es una solución del sistema de congruencia para cualquier entero v, y el conjunto de todas las soluciones es la clase de congruencias $131 + 168Z$.
Ejemplo Tenemos el siguiente sistema de congruencias

$$x \equiv \ 2 \ (mod\ 3),$$
$$x \equiv \ 3 \ (mod\ 5),$$
$$x \equiv \ 5 \ (mod\ 7),$$
$$x \equiv \ 7 \ (mod\ 11).$$

La solución para las primeras congruencias es la clase de congruencias

$$x \equiv \ 8 \ (mod\ 15).$$

La solución para las primeras tres congruencias es

$$x \equiv \ 68 \ (mod\ 105),$$

la solución de las cuatro congruencias es

$$x \equiv \ 1118 \ (mod\ 1155).$$

Ejemplo Considere

$$x \equiv 5 (mod\ 6),$$
$$x \equiv 9 (mod\ 10)\ y$$

$$x \equiv 2(mod\ 9).$$

Así $x = 5 + 6u$, sustituyendo en la segunda congruencia, da $6u \equiv 4\ (mod\ 10)$ por lo tanto $3u \equiv 2\ (mod\ 5)$, así que $u \equiv 4\ (mod\ 5)$, $u = 4 + 5v$, así que $x = 29 + 30v$. Sustituyendo en la tercera congruencia da $30v \equiv 0\ (mod\ 9)$,por lo tanto $10v \equiv 0\ (mod\ 3)$, asi que $v \equiv 0\ (mod\ 3)$ y finalmente $x \equiv 29\ (mod\ 90)$ es la solución. Note que las congruencias dadas dentro de $x \equiv 1\ (mod\ 2),\ x \equiv 4\ (mod\ 5)\ y\ x \equiv 2\ (mod\ 9)$, el cual puede ser resuelto por la formula de la prueba de el teorema chino del residuo.

En la actualidad el teorema chino del residuo ha sido de gran utilidad para la reducción de tiempo de calculo para los sistemas RSA, *Chung-Hsien Wu, Jin-Hua Hong y Cheng-Wen Wu* [22] nos dicen que la encriptación y la operación de firma pueden ser aceleradas por medio del uso del TCR, donde los factores del $mod\ n$,se asume que son conocidos, donde nos dice que los cálculos de $m = c^d(mod;)$ pueden ser particionadas en 2 partes:

$$M_p = c_p^{d_p}\ (mod\ p),$$
$$M_q = c_q^{d_q}\ (mod\ q),$$

donde

$$c_p = c\ (mod\ p),\ d_p = d\ (mod\ P - l),$$
$$c_q = c\ (mod\ q),\ d_q = d\ (mod\ q - 1).$$

y finalmente se calcula m por medio del TCR

$$m = [m_p(q^{-1}\ (mod\ p))q + m_q(p^{-1}\ (mod\ q))p]\ (mod\ n).$$

Por otro lado *Murakami, Katayanagi y Kasahara* también han propuesto una nueva clase de criptosistemas de clave pública basándose en el TRC [23] y discutido acerca de su seguridad [24]. Como vemos, el TCR lejos de ser un tema olvidado por el tiempo, ha sido renovado y empleado con una nueva visión de aplicación.

Capítulo 3

Pruebas de Primalidad

El problema de poder reconocer un número primo de un compuesto es uno de los más importantes y fundamentales dentro de la teoría de números, algoritmos y aritmética, este problema ha quedado rondando el pensamiento matemático desde tiempos antiguos hasta hoy en día, como una cuestión importante de resolver. En años recientes los algoritmos rápidos han sido un objeto popular de investigación y algunos de los métodos modernos son ahora incorporados en sistemas de algebra computacional como un estándar[25].

A lo largo de este capitulo presentamos un análisis y la revisión de los algoritmos de primalidad más conocidos, basándonos principalmente en los textos [26] y [9]. Este es un capitulo en demasía interesante y trata de explorar ese "hueco" excitante y a la vez frustrante de la teoría de números.

El análisis realizado en este capítulo nos servirá para determinar que prueba de primalidad utilizaremos en los capítulos siguientes.

3.1. Primos por divisiones sucesivas

Divisiones sucesivas, algoritmo 3, es el método secuencial de divisiones para encontrar divisores dentro de un número n. Es el algoritmo más sencillo, con mayor exactitud y fácil de entender para la factorización de enteros.

Proposición 3.1 *Tenemos un número compuesto $n > 1$: entonces n tiene un factor primo p que satisface $p \leq \sqrt{n}$. De forma equivalente: si $n > 1$ no tiene un factor primo $p \leq \sqrt{n}$, entonces n es primo.*

*Prueba.*Tenemos un $n > 1$ compuesto, $n = ab$ donde $a > 1,\ b > 1$. Entonces cualquiera a o b debe ser $\leq \sqrt{n}$, ya que si ambos son $> \sqrt{n}$ entonces ab puede ser $> n$. Suponemos $1 < a \leq \sqrt{n}$. Entonces a tiene un factor primo p, $p \leq a$, y en consecuencia $p \leq \sqrt{n}$.

$\diamond$

Todos los números pares aparte del 2 son compuestos así pueden ser descartados de una sola vez. El primer, y más sencillo, método toma cada candidato n a su vez (que es: $n = 3, 5, 7, 9, 11, \ldots$) y lo evalúa con todos los números impares $\leq \sqrt{n}$. Claramente si el factor no es encontrado entonces no hay primo menor que $\sqrt{n}$ que pueda dividir a n, asi n es primo.

Ejemplo Tenemos un número $n = 149149$. Comenzamos a dividir el número n con cada uno de los números primos pequeños, por 2, 3 y 5, y ninguno de estos es factor, ahora el próximo número por el que dividimos n es 7, este es un factor, ya que al dividir nos da un cociente de 21307 y nos arroja un residuo de 0, nuevamente comenzamos dividiendo 21307 entre 7 y luego entre 11 que nos arroja un residuo 0 y un cociente 1937, ahora al dividir 1937 entre 13 obtenemos un residuo de 0 y un cociente igual a 149, podríamos dividir nuevamente 149 entre 13, pero, esta cantidad excede el valor de $\left\lfloor \sqrt{149} \right\rfloor = 12$, por lo tanto 149 es un valor primo, por lo que tenemos una factorización de la forma $149149 = 7 \cdot 11 \cdot 13 \cdot 149$.

Algoritmo 3 Algoritmo de las divisiones sucesivas

1: **Entrada**: Entero $n \geq 2$
2: $i \leftarrow 2$
3: **while** $i * i \leq n$ **do**
4: **if** $i|n$ **then**
5: **Regresa:** El valor es compuesto.
6: **end if**
7: $i \leftarrow i + 1$;
8: **end while**
9: **Regresa:** El valor es primo.

Es razonable usar la prueba de divisiones sucesivas como un test de primalidad cuando el valor n no sea un número largo, claro que "número largo" es una cualidad subjetiva, tal criterio también depende de la velocidad de computo y de cuanto tiempo se va a permitir a una computadora ejecutar el algoritmo. Podemos usarlo con un número que tenga 20 o 25 dígitos decimales, sin embargo, cuando tengamos un número con más de 50 dígitos decimales no lo podemos usar, por que tardaría demasiado su tiempo de ejecución, por ejemplo si tuviéramos un número n con 62 dígitos decimales, se aplicaría la división por más de 10^{31} veces, y ejecutando la división en $1 * 10^{-9}$ segundos, una simple estimación nos dice que tardaríamos más de 10^{13} años en tiempo de ejecución en una computadora sencilla.

Para poder determinar si un número largo es primo por medio del método de divisiones sucesivas, podemos dar alrededor de 1000,000,000 ciclos y tal vez después de este valor de divisiones y de no haber encontrado un factor del número n, aplicar un test de primalidad diferente.

La implementación paralela del método de divisiones sucesivas es realmente sencilla. Con P procesadores podemos ejecutar hasta P divisiones en paralelo. Así, un mejoramiento lineal, a través de la implementación paralela, ha sido obtenido. La versión secuencial de las divisiones sucesivas toma un tiempo $O(p \cdot log\ n)$ para encontrar el menor factor primo p de n[27].

3.1.1. El número de factores primos de n

Para cualquier $n \geq 1$ la función $\Omega(n)$ es definida como el número total de factores primos de n, y($\Omega(1)$) se toma como 0. Así $200 = 2^3 \cdot 5^2$, así $\Omega(200) = 3 + 2 = 5$. (También existe una función ω la que cuenta los distintos factores primos de n: $\omega(200) = 2$.)

3.2. Criba de Eratóstenes

La criba de Eratóstenes, algoritmo 4, es un algoritmo que permite hallar todos los números primos, menores que un número natural dado n, este algoritmo fue encontrado en el tercer siglo A.C. por el astrónomo y geógrafo griego Eratóstenes, pero ha probado ser uno de las más eficientes algoritmos de criba hasta hoy.

Para llevar a cabo este algoritmo se forma una tabla con todos los números naturales comprendidos entre 2 y n y se van tachando los números que no son primos. Para llevarlo a cabo iniciaremos con los enteros ≥ 2 en una lista, como se muestra en el cuadro 3.1.

Cuadro 3.1: Ejemplo de la criba de Eratóstenes.

2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
2		x		x		x		x		x		x		x		x		x	
	3			x			x			x			x			x			x
			5					x					x					x	
					7							x							x

22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	...
x		x		x		x		x		x		x		x		x		x	
		x			x			x			x			x			x		
			x					x					x					x	
						x							x						

El primer paso, es cribar por 2, por lo tanto removemos todos los múltiplos de 2, excepto 2. Entonces en el cuadro 3.1 colocamos una x en todos los múltiplos de 2. El número pequeño no cribado > 2 es ahora 3, y ahora cribamos por 3, por lo tanto removemos todos los múltiplos de 3 del cuadro 3.1 excepto el mismo. La segunda fila de cruces esta hecha. El número pequeño no filtrado > 3 es ahora 5 y repetimos la criba para 5, y así de forma sucesiva. En este método todos los números compuestos son filtrados, o eliminados, y los restos son primos. Note que en

el ejemplo que contienen los números hasta el 40, hemos cribado por 2, 3, 5 y 7 hemos dejado sólo los valores primos.

La secuencia de números pequeños que sobran después de la sucesión de filtros son primos consecutivos iniciando en 2.

Podemos apresurar el filtrado por unos pocos trucos. Cuando filtremos por un primo $p > 2$, los números $2p, 3p, \ldots, (p-1)p$ se habrán ido en una fase anterior(siendo divisible por algunos primos $\leq \sqrt{(p-1)p}$ y por consiguiente $< p$). Así el primer número p que fue cribado y que ya no se criba es p^2. Sin embargo $p^2 + p,\ p^2 + 3p,\ p^2 + 5p, \ldots$, siendo pares, ya se han filtrado por 2. Así podemos iniciar el filtrado por p con la eliminación de p^2 y continuando en pasos de $2p$.

Con el fin de estar seguros de encontrar todos los números primos $\leq N$ necesitamos solamente cribar números primos $\leq \sqrt{N}$. Ejemplo, para $N = 10000$ necesitamos solamente filtrar con primos ≤ 100.

Algoritmo 4 La criba de Eratóstenes

```
1: Entrada: Número entero ≥ 2
2: m[2...n], Array de enteros
3: for j de 2 hasta n do
4:    m[j] ← 0;
5: end for
6: j ← 2;
7: while j ≤ n do
8:    if m[j] = 0 then
9:       i ← j;
10:      while i ≤ n do
11:         if m[i] = 0 then
12:            m[i] ← j;
13:         end if
14:         i = i + j;
15:      end while
16:   end if
17: end while
18: Regresa m[2...n]
```

El algoritmo 4 muestra una complejidad algorítmica de $O(N \cdot log\ log\ N)$, pero la gran limitación de la criba en un sistema computador, es el espacio enorme de memoria que consume. En ocasiones es necesario segmentar el array de 2 hasta n. Pero en épocas relativamente recientes se han buscado algoritmos que minimicen el espacio consumido de memoria tal es el caso del algoritmo propuesto por *Pritchard* que requiere solamente de $O(N/log\ log\ N)$ bits[28] basándose en el hecho de que

$$(p_1 \cdot p_2 \cdots p_i,\ p_1 \cdot p_2 \cdots p_i + k) = 1$$

si $(k, p_i) = 1$, donde p_i denota el i esimo primo y $1 \leq j \leq i$. Por su parte *Dunten, Jones Y*

Cuadro 3.2: Números primos en el intervalo $10000 - 10500$ obtenidos por medio de la criba de Eratóstenes.

10009	10037	10039	10061	10067	10069	10079	10091	10093
10099	10103	10111	10133	10139	10141	10151	10159	10163
10169	10177	10181	10193	10211	10223	10243	10247	10253
10259	10267	10271	10273	10289	10301	10303	10313	10321
10331	10333	10337	10343	10357	10369	10391	10399	10427
10429	10433	10453	10457	10459	10463	10477	10487	10499

Sorenson presentan un algoritmo basándose en el algoritmo de Pritchard[1] que usa como menos $O(N/log\ log\ N)$ operaciones aritméticas y usa un almacenamiento de $O(N/(log\ log\ log\ N))$[30]. Recientemente *S. Hwang, K. Chung y D. Kim* en [31] han desarrollado la criba de Eratóstenes de forma paralela, con la cual pudieron encontrar números primos hasta 10^{12}, en un rango de 24 horas, suponiendo así, que el conjunto completo de números primos hasta 10^{14} pueda ser obtenidos en dias de ejecución y el conjunto de 10^{16} en un año.

A modo de ejemplo en el cuadro 3.2 mostramos los números primos existentes entre 10000 y 10500, valores encontrados con el algoritmo de Eratóstenes.

3.3. Teorema de Euler y teorema pequeño de Fermat.

El teorema de Euler y el teorema de Fermat, son herramienta fundamentales en la teoría de números, con muchas aplicaciones en matemáticas y en ciencias de la computación. Pero ahora veremos como los teoremas de Euler y de Fermat pueden ser usados para determinar cuando un entero es primo o compuesto.

3.3.1. Función φ de Euler

La función φ de Euler es una función importante en teoría de números. Si n es un número entero positivo, entonces $\varphi(n)$ se define como el número de enteros positivos menores o iguales a n y coprimos con n.

La función φ de Euler presenta las siguientes propiedades:

1. $\varphi(1) = 1$
2. $\varphi(p) = p - 1$, si p es primo.
3. $\varphi(p^k) = (p-1)p^{k-1}$, si p es primo y k es un número natural.
4. φ es una función multiplicativa condicional: si m y n son primos entre sí, entonces $\varphi(mn) = \varphi(m)\varphi(n)$.

[1] Jonathan Sorenson hace una comparativa detallada del algoritmo de Pritchard y el de Eratóstenes en [29].

Por lo que el valor de $\varphi(n)$ puede calcularse empleando el teorema fundamental de la Aritmética: si

$$n = p_1^{k_1} \cdots p_r^{k_r}$$

donde los p_j son números primos distintos, por lo tanto tenemos

$$\varphi(n) = (p_1 - 1)p_1^{k_1-1} \cdots (p_r - 1)p_r^{k_r-1}. \tag{3.1}$$

Y de esta forma obtenemos una formula multiplicativa que es un producto de Euler

$$\varphi(n) = n \prod_{p|n} \left(1 - \frac{1}{p}\right) \tag{3.2}$$

donde p son los distintos primos que dividen n.

Ejemplo Tenemos el número 826875 que es igual a $3^3 5^4 7^2$ y

$$\varphi(826875) = \varphi(3^3)\varphi(5^4)\varphi(7^2)$$

utilizando las propiedades 3 y 4 de la función de Euler

$$\varphi(826875) = ((3-1)(3^{3-1}))((5-1)(5^{4-1}))((7-1)(7^{2-1})) = 378000$$

Teorema 3.1 (Teorema de Euler) *Tenemos que m es un entero positivo, y a es un entero relativamente primo con m, entonces*

$$a^{\varphi(m)} \equiv 1 \ (mod\ m) \tag{3.3}$$

Prueba. Tenemos que $\{r_1, ..., r_{\varphi(m)}\}$ es un conjunto reducido de residuos modulo m, ya que $(a, m) = 1$, tenemos que $(ar_i, m) = 1$, para $i = 1, ..., \varphi(m)$, consecuentemente para cualquier $i \in \{1, ..., \varphi(m)\}$ existe $\sigma(1) \in \{1, ..., \varphi(m)\}$ tal que

$$ar_i \equiv r_{\sigma(i)} \ (mod\ m). \tag{3.4}$$

Además, $ar_i \equiv ar_j \ (mod\ m)$, si y sólo si $i = j$, y σ es una permutación del conjunto $\{1, ..., \varphi(m)\}$ y $ar_1, ..., ar_{\varphi(m)}$ que es tambien un conjunto de residuos modulo m y seguimos con

$$\begin{aligned} a^{\varphi(m)} r_1 r_2 ... r_{\varphi(m)} &\equiv (ar_1)(ar_2) \cdots (ar_{\varphi(m)}) \ (mod\ m) \\ &\equiv r_\sigma(1) r_\sigma(2) \cdots r_\sigma\varphi(m) \ (mod\ m) \\ &\equiv r_1 r_2 \cdots r_{\varphi(m)} \ (mod\ m). \end{aligned}$$

Dividiendo por $r_1 r_2 \cdots r_{\varphi(m)}$, obtenemos

$$a^{\varphi(m)} \equiv 1 \ (mod\ m).$$

$\diamond$

3.3.2. Teorema pequeño de Fermat

Fermat introdujo el famoso resultado conocido como *Teorema pequeño de Fermat*,teorema 3.2, dejándonos una poderosa herramienta para el estudio de la primalidad, ya que de una forma eficiente nos permite conocer si un número es compuesto, sin siquiera conocer alguno de sus divisores.

Teorema 3.2 *Tenemos p, que es un número primo: Si el entero a no es divisible por p, entonces*

$$a^{p-1} \equiv 1 \ (mod\ p). \tag{3.5}$$

Además,

$$a^{p} \equiv a \ (mod\ p). \tag{3.6}$$

Para cualquier entero a.

Prueba. Si p es un primo y no divide a a, entonces $(a, p) = 1$, $\varphi(p) = p - 1$, y

$$a^{p-1} = a^{\varphi(p)} \equiv 1 \ (mod\ p) \tag{3.7}$$

por el teorema de Euler. Multiplicando esta congruencia por a, obtenemos

$$a^{p} \equiv a \ (mod\ p) \tag{3.8}$$

$\diamond$

Podemos consultar otra prueba dada por *Giedrius Alkauskas* en [32], en la cual la teoría de grupos y las propiedades de los coeficientes binomiales no se manifiestan en absoluto.

Teorema de Fermat usado como prueba de números compuestos

Tenemos que si $GCD(a, n) = 1$ y

$$a^{n-1} \not\equiv 1 \ (mod\ n). \tag{3.9}$$

Entonces n es compuesto.
Desafortunadamente el teorema de Fermat no puede ser utilizado como una prueba de primalidad en si, a pesar de la abrumadora cantidad de números compuestos n que pueden ser probados aplicando las características anteriores, ya que existen ciertas combinaciones de a y compuestos n para los que $a^{n-1} \equiv 1 \ (mod\ n)$, y esos valores de n pueden no ser mostrados como compuestos por este criterio. Un ejemplo simple es $n = 342 = 11 * 314$ dando $2^{340} \equiv 1 \ (mod\ 341)$. En este caso, sin embargo, un cambio de base de 2 a 3 nos da: $3^{340} \equiv 56 \ (mod\ 341)$, que prueba que n es compuesto.

3.4. Números de Carmichael

Existe otro hecho que nos demuestra que, no todos los números n con $GCD(a, n) = 1$ y $a^{n-1} \equiv 1 \ (mod\ n)$ para $1 \leq a \leq n - 1$ son primos. Existen números compuestos n que pasan la prueba de Fermat, siempre que a y n son relativamente primos, tales números son llamados números

de Carmichael, debido a su descubridor *Robert Daniel Carmichael (1879-1967)*. Algunos valores pequeños de estos curiosos números son

$$561 = 3 * 11 * 17,\ 1105 = 5 * 13 * 17,\ 1729 = 7 * 13 * 19.$$

Los números de Carmichael muestran otra característica, que cada uno de estos valores tiene por lo menos 3 diferentes factores primos. Fue en los inicios de 1990 cuando fue demostrado que existen infinitos números de Carmichael[33]. Un número compuesto n es un número de Carmichael si y sólo si es libre de cuadrados y $(p-1)|(n-1)$ para cualquier primo p que divida n.

Los números de Carmichael no son muy abundantes ya que, según *Pomerance, Selfridge y Wagstaffs*[34], sólo hay 2,163 números de Carmichael menores de 25,000,000,000 y sólo 16 (561, 1105, 1729, 2465, 2821, 6601, 8911, 10585, 15841, 29341, 41041, 46657, 52633, 62745, 63973 y 75361) son menores de 100,000. En este aspecto *Richard G.E. Pinch* ha dedicado varios artículos a la búsqueda de los números de Carmichael existentes ([25],[35],[36],[37],[38]), mostramos todos sus resultados obtenidos en el cuadro 3.4[39].

Además de los números de Carmichael, existen más números compuestos impares n por lo que hay números naturales con $GCD(a, n) = 1$ y $a^{n-1} \equiv 1\ (mod\ n)$, tales números son conocidos como *números de base a*, así, nuevamente tenemos un problema con un test basado en el teorema de Fermat para determinar primalidad, ya no es capaz de decir si este número n es primo o compuesto con la forma $a^{n-1}\ (mod\ n)$ con $(a, n) = 1$. Por ejemplo $561 = 3 * 11 * 17$, que es un número compuesto y la prueba de Fermat nos diría que es primo. Y tenemos que a es un número relativamente primo con 561, entonces

$$a^2 \equiv 1\ (mod\ 3),$$

y también

$$a^{560} = (a^2)^{280} \equiv 1\ (mod\ 3).$$

Similarmente

$$a^{10} \equiv 1\ (mod\ 11),$$

y también

$$a^{560} = (a^{10})^{56} \equiv 1\ (mod\ 11).$$

Finalmente

$$a^{16} \equiv 1\ (mod\ 17),$$

y

$$a^{560} = (a^{16})^{35} \equiv 1\ (mod\ 17).$$

Ya que $a^{560} - 1$ es divisible por 3, 11 y 17, también, es divisible por sus productos, por lo tanto

$$a^{560} \equiv 1\ (mod\ 561).$$

Esto prueba que 561 es un pseudo primo con base a para cualquier a tal que $(a, n) = 1$

Cuadro 3.3: Números de Carmichael($C(n)$) en 10^n.

n	12	13	14	15	16	17	18	19	20	21
$C(n)$	8241	19279	44706	105212	246683	585355	1401644	3381806	8220777	20138200

3.5. Prueba de primalidad de Lehmann

Ahora consideramos otro algoritmo para el problema de primalidad de un número, el algoritmo de Lehmann es mostrado como algoritmo 5.

El algoritmo 5 devuelve 0 si es primo y 1 si es compuesto, el alma del algoritmo se encuentra en las líneas 4-8, en la línea 4 se invoca la el porceso de aleatorización que es importante en muchos algoritmos eficientes, por lo que puede escoger un número $a \in \{1, \ldots, n-1\}$ uniformemente, en la i esima ejecución del loop, el algoritmo escoge un número a_i y calcula $c_i = a_i^{(n-1)/2};\ mod\ n$, el residuo cuando $a_i^{(n-1)/2}$ es dividido por n. Si c_i es diferente de 1 y $n-1$, entonces la salida es 1 y el algoritmo para, de otra forma, en la línea 8, c_i es almacenado en el array $b[i]$. Si todos los c_i's $\in \{1, n-1\}$. Si $n-1$ aparece al menos una vez la salida es 0, si no es así la salida es 1.

3.6. Test de primalidad de Miller-Rabin

3.6.1. Raices cuadradas de 1 no triviales

Consideramos otra propiedad aritmética del modulo p para un número primo p, que puede ser usado como un certificado de composición.

Definición 3.1 *Un número a, $1 \leq a < n$, es llamado un F-testigo para n si $a^{n-1}\ mod\ n \neq 1$.*

Si n tiene un F-testigo, es compuesto, y la función $F(n)$ denota el número de falsos *testigos* de n, que son los números $a\ mod\ n$ con $a^{n-1} \equiv 1\ mod\ n$, si n es primo entonces $F(n) = n-1$ y $F(n)$ es el grupo entero de residuos reducidos modulo n[40]. Es importante notar que un F-testigo a para n indica que n es compuesto, pero no revela ninguna información acerca de una posible factorización de n. Para el número compuesto $341 = 11 * 31$ tenemos $2^{340} mod\ 341 = 1$, uno puede darse cuenta al ver $2^{340} mod\ 341$, que ahí no hay una indicación de que 341 no sea primo. Llamamos a 2 un *mentiroso de Fermat* para 341, en el sentido de la definición 3.2.

Definición 3.2 *Para un número compuesto impar n llamamos un elemento a, $1 \leq a \leq n-1$, un F-mentiroso si $a^{n-1} mod\ n \equiv 1$*

Note que 1 y $n-1$ son F-mentirosos para todos los compuestos impares n, ya que $1^{n-1} mod\ n = 1$ en cualquier caso, y $(n-1)^{n-1} \equiv (-1)^{n-1} \equiv 1\ (mod\ n)$ ya que $n-1$ es par. Ahora siguiendo con $n = 341$ tenemos $3^{340} mod\ 341 = 56$, así 3 es un número F-testigo para 341.

Definición 3.3 *Tenemos $1 \leq a < n$. Entonces el número a es llamado raíz cuadrada de 1 modulo n si $a^2\ (mod\ n) = 1$.*

En esta definición, los números 1 y $n-1$ son siempre raíces cuadradas de 1 mod n ($(n-1)^2 \equiv (-1)^2 \equiv 1\ (mod\ n)$); son llamados las raíces cuadradas de 1 *mod n triviales*. Si n es un número primo, no hay otra raíz cuadrada de 1 *mod n*.

Lema 3.1 *Si p es un número primo y* $1 \leq a < p$ *y* $a^2 mod\ p = 1$*, entonces* $a = 1$ *o* $a = p-1$.

Así, si encontramos alguna raíz cuadrada de 1 ($mod\ n$) no trivial, entonces n es con certeza compuesto.

Por ejemplo, todas las raíces cuadradas de 1 (mod 91) son 1, 27, 64 y 90.

Revisamos el teorema 3.2. Veamos $a^{n-1}\ (mod\ n)$ un poco más de cerca. Por supuesto, estamos solamente interesados en números impares n. Entonces $n-1$ es par, y lo podemos escribir como $(n-1) = u * 2^k$, para algunos números impares u y algunos $k \geq 1$. Así, $a^{n-1} \equiv (a^u)\ (mod\ n)^{2^k}\ (mod\ n)$, lo que significa que podemos calcular $a^{n-1}\ (mod\ n)$ con $k+1$ pasos intermedios: si tenemos

$$b_0 = a^u\ (mod\ n);\ b_i = (b_{i-1}^2)\ (mod\ n),\ para\ i = 1, ..., k, \tag{3.10}$$

entonces $b_k = a^{n-1}\ (mod\ n)$. Por ejemplo, para $n = 325 = 5^2 * 13$ tenemos que $n - 1 = 324 = 81 * 2^2$. En el cuadro 3.4[26] calculamos las potencias de a^{81}, a^{162} y a^{324}, todas modulo 325, para varias a.

Algoritmo 5 Algoritmo de Lehmann

```
1: Entrada: Número entero ≥ 2
2: b[2...l], Array de enteros
3: for i de 1 hasta l do
4:    a ← a, escogida aleatoriamente {1, ..., n − 1};
5:    a^((n−1)/2) mod n;
6:    if c ∈ {1, n − 1} then
7:       Regresa 1;
8:    else
         b[i] ← c;
9:    end if
10: end for
11: if b[1] = ··· = b[l] = 1 then
12:    Regresa 1;
13: else
       Regresa 0;
14: end if
```

En el cuadro 3.4, vemos que 2 es un F-testigo para 325 de $\mathbb{Z}_{325}^*$, por otro lado vemos que 7, 32, 49 y 126 son todos F-mentirosos para 325. Calculando 201^{324} (mod 325) con 2 pasos intermedios nos lleva a detectar que 51 es una raíz cuadrada de 1 no trivial, lo que prueba que 325 no es primo. Por otro lado, el calculo correspondiente con base 7, 32 o 49 no dan ninguna información,

Cuadro 3.4: Potencias a^{n-1} $(mod\ n)$.

a	$b_0 = a^{81}$	$b_1 = a^{162}$	$b_2 = a^{324}$
2	252	129	66
7	307	324	1
32	57	324	1
49	324	1	1
65	0	0	0
126	1	1	1

ya que $7^{162} \equiv 32^{162} \equiv -1$ $(mod\ 325)$ y $49^{81} \equiv -1$ $(mod\ 325)$. De forma similar, calculando las potencias de 126 nos revela una raíz cuadrada no trivial de 1,ya que 126^{81} $(mod\ 325) = 1$.

Definición 3.4 *Tenemos un número $n \geq 3$ que es impar, y escribimos $n = u * 2^k$, donde u es impar, y $k \geq 1$. Un número a, $1 \leq a < n$, es llamado un A-testigo para n si a^u mod $n \neq 1$ y a^{u*2^i} mod $n \neq n-1$ para todo i, $0 \leq i < k$. Si n es compuesto y a no es un A-testigo para n, entonces a es llamado un A-mentiroso para n.*

Lema 3.2 *Si a es un A-testigo para n, entonces n es compuesto.*

Combinamos esta observación con la idea de escoger algún número a de $\{2, ..., n-2\}$ de forma aleatoria dentro de un fortalecimiento de la prueba de Fermat, llamándolo *la prueba de Miller-Rabin*, mostrado como el algoritmo 6.

Artjuhov propuso estudiar la secuencia $b_i = a^{u*2^i} mod\ n$, para $0 \leq i \leq k$, para probar n para saber si es compuesto. Después, Miller uso el criterio en su algoritmo determinístico que tendrá tiempo polinomial de ejecución si la hipótesis extendida de Riemann(HER) era cierta[41]. El demostró que, asumiendo la HER, el menor A-testigo para un número compuesto n será de forma $O((ln\ n)^2)$. Después *Bach* da un resultado de $2(ln\ n)^2$ para los A-testigos menores. El test de primalidad resultante es obvio: para $a = 2, 3, ..., \lfloor 2(ln\ n)^2 \rfloor$ comprueba si a es un A-testigo para n. Si todos estos a's fallan para ser A-testigos, entonces n es un número primo. El algoritmo usa $O((log\ n)^3)$ operaciones aritméticas.

Después, alrededor de 1980, *Rabin*(e independientemente *Monier*) reconoció la posibilidad de cambiar la búsqueda determinista de Miller para un A-testigo dentro de un eficiente algoritmo *aleatorizado*. Independientemente, un alternativo algoritmo aleatorizado con muchas propiedades similares, pero basados en diferentes principios de la teoría de números, fue descubierto por *Solobay y Strassen*.

A continuación describimos la versión aleatorizada de la prueba para saber si un número es compuesto basado en el concepto del A-testigo, llamado comúnmente *test de Miller-Rabin*.

Ahora analizaremos el tiempo de ejecución del algoritmo 6, le toma al menos $log\ n$ divisiones para encontrar u y k. El cálculo de a^u $mod\ n$ por exponenciación rápida en la línea 3 toma

$O(log\ n)$ operaciones aritméticas, y finalmente el loop en las líneas 5 a 8 es ejecutado $k \leq log\ n$ veces; en cada iteración la multiplicación modulo n es la operación más costosa. En total, el algoritmo usa $O(log\ n)$operaciones aritméticas.

En el algoritmo 6 existen 2 posibilidades para que a la salida se obtenga una respuesta "es compuesto".

Caso 1: Hay algun $i,\ 1 \leq i \leq k-1$, tal que en el curso de la i-teava ejecución de la línea 7 el algoritmo encuentra que $b_i = 1$. Por el test en la línea 4, y el test en las líneas 7 y 8 se llevan a cabo durante la ejecución previa del loop, conocemos que $b_0, ..., b_{i-1} \notin \{1, n-1\}$, esto implica que $b_0, .., b_{k-1}$ no contiene $n-1$, por lo tanto a es un A-testigo.

Caso 2: Línea 9 es ejecutada, por el test realizado por el algoritmo en la línea 4 y en la $k-1$ ejecuciones de líneas 8 y 9, esto indica que $b_0, ..., b_{k-1}$ son todos diferentes de 1 y $n-1$. Nuevamente a es un A-testigo.

3.7. Criterio de Solovay-Strassen

Este test de primalidad, que vemos como algoritmo 8 es similar al test de Miller-Rabin. Así como el test de Miller-Rabin, que también es un algoritmo de procedimiento aleatorio(*Gregory J. Chaitin y Jacob T. Schwartz1* en [42] nos muestran un análisis detallado de estos 2 métodos de pruebas de primalidad), es capaz de reconocer números compuestos con una probabilidad de al menos 1/2, para poder dar una explicación de este test de primalidad primero tenemos que dar algunos conceptos matemáticos como los residuos cuadráticos, el símbolo de Legendre y el símbolo de Jacobi.

3.7.1. Residuos cuadráticos

Los residuos cuadráticos es una de las gemas de la teoría de números, para explicarlo un poco tenemos

$$\left(\frac{p}{q}\right)\left(\frac{q}{p}\right) = (-1)^{(p-1)(q-1)/4}$$

si p o q son congruentes con 1 (mod 4), entonces p es un residuo cuadrático $mod\ q$ si y sólo si q es un residuo cuadrático $mod\ p$. Si p y q son congruentes con 3 mod 4 entonces p es un residuo cuadrático $mod\ q$ si y sólo si q es un residuo cuadrático $mod\ p$[43].

Definición 3.5 *Para $m \geq 2$ y $a \in \mathbb{Z}$ con $(a, m) = 1$ decimos que es un* **residuo cuadrático** *modulo m si $a \equiv x^2\ (mod\ m)$ para algun $x \in \mathbb{Z}$. Si a satisface $(a, m) = 1$ y no es un residuo cuadrático modulo m, este es llamado un* **residuo no cuadrático**.

Ejemplo Para $m = 11$, los cuadrados modulo 11 de $1, 2, ..., 11$ son 1, 4, 9, 5, 3, 3, 5, 9, 4, 1, por lo tanto los residuos cuadrados son 1, 3, 4, 5, 9. Para $m = 20$, los residuos cuadrados son 0,1,4,5,9,16, y para $m = 7$ los residuos cuadrados son 1,2,4.

Observamos que en el caso de $m = 11$ hay 5 residuos y 5 no residuos. Este comportamiento es típico para $m = p$, donde p es un número primo[26].

Lema 3.3 Criterio de Euler *Si p es un número impar entero, entonces, el conjunto de residuos cuadrados es un subgrupo de $\mathbb{Z}_p^*$ de tamaño $(p-1)/2$. Además, para $a \in \mathbb{Z}_p^*$, tenemos*

$$a^{(p-1)/2} \begin{cases} 1, \text{ si } a \text{ es un residuo cuadrado modulo } p \\ -1, \text{ si } a \text{ es un no residuo modulo } p \end{cases} \quad (3.11)$$

Lema 3.4 *Si p es un número primo con $p \equiv 3 \ (mod\ 4)$, entonces para cada residuo cuadrado $a \in \mathbb{Z}_p^*$ el elemento $x = a^{(p+1)/4}$ satisface $x^2 = a$.*

Por ejemplo, consideramos $p = 11$. El número $a = 3$ satisface $a^5 \ (mod\ 11) = 1$, por lo tanto es un residuo cuadrado. Una raíz cuadrada de 3 es $3^{(11+1)/4} \ (mod\, 11) = 3^3 \ mod\ 11 = 5$ el otro es $11 - 5 = 6$.

Buscar raíces cuadradas modulo p para números primos $p \equiv 1 \ (mod\ 4)$ no es tan fácil, pero hay algoritmos eficientes que realizan esta tarea, como el descrito en [44].

Definición 3.6 *Para un número primo $p \geq 3$ y un entero a tenemos*

$$\left(\frac{a}{p}\right) = \begin{cases} 1, \text{ si } a \text{ es un residuo cuadrado modulo } p \\ -1, \text{ si } a \text{ es un no residuo modulo } p \\ 0, \text{ si } a \text{ es un multíplo de } p, \end{cases} \quad (3.12)$$

donde $\left(\frac{a}{p}\right)$ es llamado el *Símbolo de Legendre* de a y p.

El símbolo de Legendre satisface algunas simples reglas lo que es obvio por la definición y el lema 3.3.

Lema 3.5 *Para un número primo $p \geq 3$ tenemos:*

1. $\left(\frac{a*b}{p}\right) = \left(\frac{a}{p}\right) * \left(\frac{b}{p}\right)$, *para todo* $a,\ b \in \mathbb{Z}$.
2. $\left(\frac{a*b^2}{p}\right) = \left(\frac{a}{p}\right)$, *para* $a, \in \mathbb{Z}$, *donde* $p \not| b$.
3. $(a + cp) = \left(\frac{a}{p}\right)$, *para todos lo senteros a y c. En particuloar,* $\left(\frac{a \bmod p}{p}\right)$, *para todo a.*
4. $\left(\frac{-1}{p}\right) = (-1)^{(p-1)/2}$.

La característica 4 dice que -1 es un residuo cuadrado modulo p si y sólo si $p \equiv 1 \ (mod\ 4)$. Por ejemplo 12 es un residuo cuadrado modulo 13, mientras 10 no es un residuo cuadrado modulo 11.

3.7.2. El símbolo de Jacobi

Asumiendo que $(a, n) = 0$, podemos distinguir los casos donde a es un residuo cuadrado modulo n y donde a no es residuo.

Definición 3.7 *Tenemos un número $n \geq 3$ y un entero impar con descomposición de primos $n = p_1 \cdots p_r$. Para enteros a, tenemos*

$$\left(\frac{a}{n}\right) = \left(\frac{a}{p_1}\right) \cdots \left(\frac{a}{p_r}\right), \tag{3.13}$$

donde $\left(\frac{a}{n}\right)$ es llamado el **símbolo de Jacobi**.

Para familiarizarse con esta definición, vamos a considerar algunas de estas características. Si n es un número primo, entonces el símbolo de Jacobi $\left(\frac{a}{n}\right)$ es el mismo que el símbolo de Legendre, así, es indistinto que usemos la misma notación para ambos. Si a y n tienen un factor común, entonces uno de los p_i's divide a a, por lo tanto $\left(\frac{a}{n}\right) = 0$. Esto es, por ejemplo, que $\left(\frac{9}{21}\right) = 0$, también es un residuo cuadrado modulo 21. Obtenemos interesantes valores si a y n son relativamente primos. Es importante notar que también en este caso $\left(\frac{a}{n}\right)$ no indica si a es un residuo cuadrado modulo n o no. Por ejemplo $\left(\frac{2}{15}\right) = \left(\frac{2}{3}\right) * \left(\frac{2}{5}\right) = (-1)(-1) = 1$ mientras 2 no es un residuo cuadrado modulo 15.

Si n y m es rango sobre enteros impares ≥ 3, y a,b rango sobre todos los enteros, entonces

1. $\left(\frac{a*b}{n}\right) = \left(\frac{a}{n}\right) * \left(\frac{b}{n}\right)$;
2. $\left(\frac{a*b^2}{n}\right) = \left(\frac{a}{n}\right)$, si $(b, n) = 1$;
3. $\left(\frac{a}{n*m}\right) = \left(\frac{a}{n}\right) * \left(\frac{a}{m}\right)$;
4. $\left(\frac{a}{n*m^2}\right) = \left(\frac{a}{n}\right)$, si $(a, m) = 1$;
5. $\left(\frac{a+cn}{n}\right) = \left(\frac{a}{n}\right)$, para todos los enteros c (en particular $\left(\frac{a}{n}\right) = \left(\frac{a \bmod n}{n}\right)$);
6. $\left(\frac{a^{2k}*a}{n}\right) = \left(\frac{a}{n}\right)$, y $\left(\frac{a^{2k+1}*a}{n}\right) = \left(\frac{2}{n}\right) * \left(\frac{a}{n}\right)$, para $k \geq 1$;
7. $\left(\frac{-1}{n}\right) = (-1)^{(n-1)/2}$;
8. $\left(\frac{0}{n}\right) = 0$ y $\left(\frac{1}{n}\right) = 1$.

$$n \equiv (p_1 \bmod 4) \cdots (p_r \bmod 4) \equiv (-1)^s \pmod 4. \tag{3.14}$$

Esto dice que $\frac{1}{2}(n-1)$ es impar si y sólo si s es impar, por lo tanto $(-1)^s = (-1)^{(n-1)/2}$.

En 1992 *Jeffrey S.* propuso un nuevo algoritmo para calcular el símbolo de Jacobi basado en el algoritmo binario para calcular el (a, b) propuesto por *Stein*[45], por otro lado *Shawna M. y Jonathan P.* en [46] propusieron 2 nuevos algoritmos para la búsqueda del símbolo de Jacobi que toman $O(n^2/log\ n)$ en tiempo de ejecución y $O(n)$ en espacio. En este mismo trabajo presentan las versiones paralelas de cada uno de estos algoritmos con $O_\epsilon(n/log\ log\ n)$ en tiempo de ejecución trabajando con $O(n^{1+\epsilon})$ procesadores y recientemente *George P., Carla P. y Kiran V.* en [47] presentan 2 algoritmos binarios que son fácilmente implementados a nivel hardware.

3.7.3. La ley de la reciprocidad cuadrada

La ley de la reciprocidad cuadrada ha fascinado a matemáticos por cerca de 300 años y sus generalizaciones y analogías ocupan un lugar central en la teoría de números hoy en día. La brillantez de Fermat(1640) y las pruebas de Gauss(1796) han dado lugar a una impresionante construcción abstracta llamada *clase de teoría de campos*[48].

Teorema 3.3 *Si $m, n \geq 3$ son enteros primos, entonces*

$$\left(\frac{m}{n}\right) = \begin{cases} \left(\frac{n}{m}\right) \ si\ n \equiv 1\ o\ m \equiv 1\ (mod\ 4), \\ -\left(\frac{n}{m}\right) \ si\ n \equiv 3\ y\ m \equiv 3\ (mod\ 4), \end{cases} \quad (3.15)$$

Proposición 3.2 *Sí $n \geq 3$ es un entero impar, entonces*

$$\left(\frac{2}{n}\right) = \begin{cases} 1, \ si\ n \equiv 1\ o\ n \equiv 7\ (mod\ 8) \\ -1, \ si\ n \equiv 3\ o\ n \equiv 5\ (mod\ 8) \end{cases}$$

notamos como estas leyes dadas nos muestran una forma eficiente para la evaluación del símbolo de Jacobi $\left(\frac{a}{n}\right)$ para cualquier entero impar $n \geq 3$ y cualquier entero a, la idea es más fácil formulada relativamente. Tenemos un entero arbitrario a y un número impar $n \geq 3$ que son dados.

1. Si a no está en el intervalo $\{1, ..., n-1\}$, el resultado es $\left(\frac{a\ mod\ n}{n}\right)$.
2. Si $a = 0$, el resultado es 0.
3. si $a = 1$, el resultado es 1.
4. Si $4|a$, el resultado es $\left(\frac{a/4}{n}\right)$.
5. Si $2|a$, el resultado es $\left(\frac{a/2}{n}\right)$ si $n\ mod\ 8 \in \{1, 7\}$ y $-\left(\frac{a/2}{n}\right)$ si $n\ mod\ 8 \in \{3, 5\}$.
6. Si $a > 1$ y $a \equiv 1$ o $n \equiv 1\ (mod\ 4)$, el resultado es $\left(\frac{n\ mod\ a}{a}\right)$.
7. Si $a \equiv 3$ y $n \equiv 3\ (mod\ 4)$, el resultado es $-\left(\frac{n\ mod\ a}{a}\right)$.

Estas reglas hacen fácil el calcular $\left(\frac{a}{n}\right)$ por partes para a y n que no sean muy grandes, estas reglas están representadas dentro del algoritmo 7.

Para su implementación es preferible un procedimiento iterativo. Para el algoritmo 7 usamos una variable s(signo), para acumular los factores (-1).

Para entender lo que el algoritmo 7 hace, imaginemos que en b y en c dos coeficientes son almacenados, lo que indica que el símbolo de Jacobi $\left(\frac{b}{c}\right)$ esta listo para ser evaluado. El loop *while* grande (líneas de la 6 a la 23) es iterado hasta que el componente b ha alcanzado un valor en $\{0, 1\}$, tenemos $b = \left(\frac{b}{c}\right)$. La variable s contiene un número $s \in \{-1, 1\}$, lo que acumula los cambios de signo causada por las iteraciones de el ciclo while. Entonces, en las líneas 19-22, la ley de la reciprocidad cuadrada es aplicada, nuevamente cambiamos s si es necesario. Notamos

Algoritmo 6 Test de primalidad de Miller-Rabin

Entrada: Número entero impar ≥ 3
Encontrar un valor impar u y k que cumplan $n = u * 2^k$;
Escogemos un valor a de forma aleatoria en $\{2, ..., n-2\}$;
$b \leftarrow a^u \ mod \ n$;
if $b \in \{1, n-1\}$; **then**
  Regresa: n es primo;
end if
for $i = 1$ hasta $k-1$ **do**
  $b \leftarrow b^2 \ mod \ n$;
  if $b = n-1$ **then**
    Regresa: n es primo;
  end if
  if $b = 1$ **then**
    Regresa: n es compuesto;
  end if
end for
Regresa: n es compuesto.

que no es necesaria un evaluación extra del máximo común divisor de a y n.

Ejemplo Queremos encontrar el valor $\left(\frac{773}{1373}\right)$, Ambos valores son números primos, aplicando estas reglas obtenemos:

$$\left(\frac{773}{1373}\right) \overset{(6)}{=} \left(\frac{600}{773}\right) \overset{(4)}{=} \left(\frac{150}{773}\right) \overset{(5)}{=} -\left(\frac{75}{173}\right) \overset{(6)}{=} -\left(\frac{23}{75}\right) \overset{(7)}{=} \left(\frac{6}{23}\right) \overset{(5)}{=} \left(\frac{3}{23}\right) \overset{(7)}{=} -\left(\frac{2}{3}\right) \overset{(5)}{=} \left(\frac{1}{3}\right) \overset{(3)}{=} 1$$

así, 773 es un residuo cuadrado modulo 1373.

3.7.4. Test de primalidad por residuos cuadrados

Lema 3.6 *Si p es un número entero impar, entonces*

$$a^{(p-1)/2} * \left(\frac{a}{p}\right) \ mod \ p = 1, \ para \ todo \ a \in \{1, ..., p-1\}. \tag{3.16}$$

Notamos que si $n \geq 3$ es impar y $a \in \{2, ..., n-1\}$ satisface $a^{(n-1)/2} \cdot \left(\frac{a}{n}\right) \ mod \ n \neq 1$, entonces n no puede ser un número primo. Recordemos que llamamos a un elemento a, $1 \leq a < n$, un F-testigo para un número compuesto impar n si $a^{n-1} \ mod \ n \neq 1$ y un F-mentiroso para otro n. En referencia al criterio de Euler del lema 3.3 ahora definimos:

Definición 3.8 *Tenemos que n es un númeo compuesto impar. Un número a, $1 \leq a < n$ es llamado un* **E-testigo** *para n si $a^{(n-1)/2} \cdot \left(\frac{a}{n}\right) \ mod \ n \neq 1$. De lo contrario es llamado un* **E-mentiroso**.

Ejemplo: Consideremos el número compuesto $n = 325$. Para $a = 15$, tenemos $(15, 325) = 5$, por lo tanto $\left(\frac{15}{325}\right) = 0$, y 15 es un E-testigo. Para $a = 2$, tenemos $2^{162} mod\ 325 = 129$ así 2 es un E-testigo. Para $a = 7$, tenemos $7^{162} mod\ 325 = 324$ y $\left(\frac{7}{325}\right) = -1$, esto nos dice que 7 es un E-mentiroso para 325.

Lema 3.7 *Tenemos $n \geq 3$ que es un número compuesto impar. Entonces todos los E-mentirosos para n también son F-mentirosos para n*

Lema 3.8 *Tenemos $n \geq 3$ que es un número compuesto impar. Entonces, el conjunto $\{a|a$ es un E-mentiroso para $n\}$es un subgrupo característico de $\mathbb{Z}_n^*$.*

Algoritmo 7 Símbolo de Jacobi

```
1: Entrada: Número entero impar n ≥ 3 y un entero a
2: b, c, s: son enteros.
3: b ← a mod n;
4: c ← n;
5: s ← 1;
6: while b ≥ 2 do
7:    while 4|b do
8:       b ← b/4
9:    end while
10:   if 2|b then
11:      if c mod 8 ∈ {3, 5} then
12:         s ← (−s);
13:      end if
14:      b ← b/2;
15:   end if
16:   if b = 1 then
17:      break;
18:   end if
19:   if b mod 4 = c mod 4 = 3 then
20:      s ← (−s);
21:   end if
22:   (b, c) ← (c mod b, b);
23: end while
24: Regresa: s ∗ b.
```

El lema 3.7 nos dice que si n es un número primo el algoritmo regresara un "es primo". Pero si n es compuesto, el algoritmo regresara un "es primo" si el valor a escogido de forma aleatoria es un E-mentiroso. Por el lema 3.8 sabemos que el conjunto de E-mentirosos es un subgrupo de $\mathbb{Z}_n^*$ y por lo tanto comprenden no más que $\frac{1}{2}\varphi(n)$ elementos. Exactamente, como en el caso del test de Fermat, podemos ver que la probabilidad de que un elemento a escogido de forma aleatoria sea un E-mentiroso es menor que $\frac{1}{2}$.

3.8. Resultados

En el cuadro 3.5 podemos observar el tiempo que le lleva a cada una de las pruebas de primalidad explicadas en este capitulo determinar si un número dado es primo o no.

Algoritmo 8 Test de Solovay-Strassen

1: **Entrada:** Número entero impar ≥ 3.
2: Escogemos un número aleatorio a del conjunto $\{2, ..., n-2\}$
3: **if** $a^{(n-1)/2} \cdot \left(\frac{a}{n}\right) \ mod\ n \neq 1$ **then**
4: **Regresa:** Es compuesto.
5: **else**
6: **Regresa:** Es primo.
7: **end if**

Para poder realizar las pruebas de primalidad, lo primero que necesitamos es tener números primos para aplicarles dichas pruebas, existen una cantidad enorme de algoritmos generadores de números primos[2], la mayoría de los trabajos son para la generación de algoritmos que generan números para un propósito en particular como [49] y [50].Pero, para nuestros propósitos más generales utilizamos el algoritmo 9 para generar los valores primos con los que realizamos las pruebas, donde S es un valor primo que nosotros ingresamos. En el apéndice B colocamos algunos valores primos grandes generados a través del algoritmo 9.

Para la obtención de resultados se hicieron pruebas en varios valores, con aproximadamente la misma cantidad de dígitos decimales, obteniendo valores similares o iguales de los tiempos de ejecución de cada número. Colocar todos los valores analizados en un cuadro nos ocuparía un espacio considerable, por lo que, en el cuadro 3.5, sólo colocamos una muestra de cada conjunto de números, con un tamaño determinado en dígitos decimales, y su tiempo de ejecución.

Como podemos observar en el cuadro 3.5, para valores pequeños no existe una diferencia considerable entre sus tiempos de ejecución, pero, al trabajar con números grandes los tiempos comienzan a cambiar, lo que nos muestra que los algoritmos de Miller-Rabin, Solovay-Strassen y el de Fermat son los más rápidos en su ejecución, dejando muy por detrás al método de divisiones sucesivas, el cual después de un valor de 18 cifras comienza a ser ineficiente. Como ya mencionamos anteriormente, la prueba de Fermat puede encontrarse con valores que son F-mentirosos, pero la prueba de Fermat iterada es eficiente para, aunque existan F-mentirosos en un valor, determinar si es primo o no, exceptuando el caso de los números de Carmichael, para los cuales ni el algoritmo de Fermat simple ni el iterado, son capaces de reconocer si son o no primos.

Tomando el ejemplo de 1105, que es un número de Carmichael, vemos que las pruebas que menos tienen dificultades para determinar si es primo o no, son la de Solovay-Strassen iterado, seguido del de Solovay-Strassen sin iteraciones y el más eficiente para reconocerlo es el método de las divisiones sucesivas, ya que, debido a su forma de trabajo (ver pag. 21), es el único que siempre

[2]En el presente trabajo no nos adentramos en la teoría de generadores de números primos, ya que sólo utilizamos un algoritmo generador como herramienta de pruebas y generación de primos de forma general.

Cuadro 3.5: Tiempos de ejecución de los algoritmos de primalidad, dados en milisegundos. Donde *** significa que sobrepasaron un tiempo de ejecución de 10 horas, este tiempo fue tomado como referencia de forma arbitraria.

Valor evaluado	No. de dígitos	Fermat	Fermat iterado	Lehmann	Miller-Rabin	Por divisiones	Solovay-Strassen	S-S iterado
113	3	0	0	0	0	0	0	0
623578687	9	0	0	0	0	46	0	0
140133369504679123	18	0	0	0	0	***	0	0
1757642099179598955758791698423320-58652600111	45	0	0	0	0	***	0	0
1328743013705315636426328115203523-5256248716958642622992193049813280-26829577802771331727	88	7	9	2	0	***	2	15
1763146774021970974653474894884101-2263169838469558810938797470023160-659278827654217444950160514924657 7	102	7	9	2	1	***	3	20
1400218991438570086354007815530250-4646069454316745038807686994739737-4981306727284398195975298975768237-4300085209865028314783887905778279-9213158720422302333916754902366989-17263561100677	184	16	45	33	16	***	16	93
3035154389296044588383812306260421-8292306481491688449446406916762664-1774160971849146857907020289954774-7825897210945867080007356471801351-0682141397254076243379756039541143-5343383791935933874145483019576281-9350176494865106208164901534852919-7513386237284820243117331655911271-2350943785371937405701164711386105-0031970441237423690875142735973707-7666975479698089289530662572806951-2962052147704601982495559422354478-5244711201533308967645906126785969-8597141124196963493	461	187	313	297	185	***	187	650

acertara en la respuesta.

Este último método en particular, es un método con una eficiencia total en cuanto a determinar si un número es primo o no, ya que bajo cualquier circunstancia es el único método que siempre nos arrojara una respuesta correcta, lamentablemente, como ya lo mencionamos, su eficiencia en cuestión de tiempo es muy baja, ya que tardaría años para calcular factores primos grandes de un número compuesto n.

Algoritmo 9 Generador de números primos

Entrada: S
Escogemos un número aleatorio R en el intervalo $S \leq R \leq 4S + 2$.
$N = SR + 1$.
Verificamos si los números primos pequeños son divisores de N.
Si un valor es divisor de N escogemos otro valor R aleatorio.
Usamos el algoritmo de Miller-Rabin
if N=compuesto **then**
    Escogemos otro valor R aleatorio.
end if
Escogemos un número aleatorio a, $1 < a < N$
Verificamos $a^{N-1} \equiv 1 \; mod \; N$ y $(a^R - 1, N) = 1$
if las relaciones anteriores son ciertas **then**
    N es primo.
else
    Escogemos otro valor de a
end if

Capítulo 4

Algoritmos de factorización de números enteros en sus factores primos

Este capitulo está centrado principalmente en la verificación de la confirmación de éxito y análisis de algunos algoritmos de factorización de números enteros. Tomando como base algúnos valores expuestos con anterioridad por distintos autores. Para la realización de este capitulo nos basamos principalmente en [20], [51], [52] y [53].

Por medio del *teorema fundamental de la aritmética*,teorema 2.1, sabemos que todo número compuesto es resultado de la multiplicación de sus factores primos, el problema de encontrar estos factores primos, aparentemente sencillo, es uno de los problemas abiertos en teoría de números y en computación, más complejos, importantes y útiles, debido en gran parte a las aplicaciones que puede tener en el campo de la seguridad informática.

Cuando tenemos un valor n "pequeño" , por ejemplo $n = 231$, podemos encontrar rápidamente sus factores primos $3 \cdot 7 \cdot 11$, gracias a que estos factores son pequeños y podemos realizar los cálculos casi de inmediato, ahora bien, esto cambia cuando empezamos a trabajar con números grandes, digamos un número n con 50 dígitos decimales, es difícil para nosotros como personas el simple hecho de tratar de imaginar un posible factor, peor aún, si este valor tiene un factor primo p de 13 dígitos decimales, no podríamos encontrar ese valor, más que por casualidad. Las computadoras son de gran ayuda en el momento de realizar estos cálculos, pero, aún así utilizan una gran cantidad de recursos en espacio de memoria y tiempo de ejecución para poder encontrar una solución, por ejemplo, si suponemos que una computadora puede realizar $1 \cdot 10^8$ cálculos por segundo, tardaríamos aproximadamente 27 horas para encontrar ese valor, probando con cada uno de los valores menores que p.

Hasta este momento no existe un algoritmo que pueda factorizar números enteros en su forma general, y es gracias a la dificultad que tiene una computadora en poder factorizar enteros con factores primos grandes, que hasta nuestros días sigue siendo útil el algoritmo de encriptación RSA[1].

[1]Los laboratorios RSA animan a los investigadores a encontrar algoritmos eficientes para la factorización de números grandes a través de retribuciones económicas, en la dirección electrónica http://mathworld.wolfram.com/RSANumber.html podemos ver los valores a factorizar y los que ya han

4.1. Criterios de divisibilidad

Existen criterios de divisibilidad que pueden ser usados para determinar si un valor m es divisible por un valor en especifico, tenemos que m es un entero con la forma

$$m = \sum_{k=0}^{n} a_k 10^k, \tag{4.1}$$

con un vector asociado v_m definido como

$$v_m = (a_0, a_1, a_2, \ldots a_{n-1}, a_n),$$

donde v_m no es más que la sucesión en orden inverso de las cifras del número en su notación decimal. Por ejemplo $v_{123} = (3, 2, 1, 0)$.

Los criterios de divisibilidad más comunes son los que se enumeran a continuación

1. Un número m es divisible por 2 si, el número termina con un digito 0 o par, $2|m$ si $2|m_0$.
2. m es divisible por 3 si, la suma de sus dígitos decimales es divisible entre 3, $3|m$ si $3|a_0 + a_1 + \cdots a_n$.
3. m es divisible por 4 si, los dos últimos dígitos son 0 o forman un valor múltiplo de 4, $4|m$ si $4|(a_0 + 10a_1)$.
4. m es divisible por 5 si, el ultimo digito es 0 ó 5, $5|m$ si $5|a_0$.
5. Para saber si m es divisible por 7, primero, dividimos m en grupos de 3 dígitos $\{a_0, a_1, a_3\}\cdots$, y a cada grupo se le resta el último digito multiplicado por 2 y si la suma y resta, alternativamente de cada uno de los valores de estos grupos es múltiplo de 7, entonces 7 divide a m, $7|m$ si $7|(a_0+10a_1+100a_2+a_6+10a_7+100a_8+\ldots-a_3-10a_4-100a_5-a_9-10a_{10}-100a_{11}-\ldots$
6. m es divisible por 8 si, el número formado por los tres últimos dígitos es divisible por 8, $8|m$ si $8|a_0 + 10a_1 + 100a_2$.
7. m es divisible por 9 si, la suma de sus dígitos es múltiplo de 9, $9|m$ si $9|(a_0 + a_1, \ldots, a_n$.
8. m es divisible por 11 si, la diferencia de la suma de dígitos pares por un lado y dígitos impares por otro lado, es 0 o múltiplo de 11, $11|m$ si $11|((a_0 + a_2 + a_4 + \cdots) - (a_1 + a_3 + a_5 + \cdots))$.
9. Para saber si m es divisible por 13, dividimos el valor en grupos de 3 dígitos $\{a_0, a_1, a_3\}\cdots$, a cada grupo le separamos los dos primeros dígitos y se le suma el último digito multiplicado por 4, luego realizamos sumas y restas alternativas de derecha a izquierda con cada resultado de cada grupo, si el resultado es múltiplo de 13, m también lo es, $13|m$ si $13|(a_0+10a_1+100a_2+a_6+10a_7+100a_8+\ldots-a_3-10a_4-100a_5-a_9-10a_{10}-100a_{11}-\ldots$

sido factorizados.

Estos criterios son criterios específicos, sin embargo, *Octavio Montoya* de la Universidad de Tolima nos describe un criterio general de divisibilidad en los enteros positivos y nos da el siguiente teorema

Teorema 4.1 *Sean p, m enteros positivos, p es divisible por m si y sólo si $v_p \cdot v_m^*$ es divisible por m.*

Donde v_p es el vector asociado a p y v_m^* es el vector adjunto de m.

Sea m un entero positivo. El vector adjunto de m lo denotaremos v_m^* y está definido como

$$v_m^* = (x_0, x_1, \cdots, x_n, \cdots), \tag{4.2}$$

donde para cada $n \geq 0$, x_n es el entero más próximo a cero que es solución de la congruencia lineal

$$x \equiv 10^n \ (mod\ m). \tag{4.3}$$

Por ejemplo, si tenemos $m = 8$, realizamos los cálculos con la congruencia 4.3.

$n = 0$, $x_0 \equiv 10^0 \ (mod\ 8)$, tenemos que $x_0 = 1$.
$n = 1$, $x_1 \equiv 10^1 \ (mod\ 8)$, tenemos que $x_1 = 2$.
$n = 2$, $x_2 \equiv 10^2 \ (mod\ 8)$, tenemos que $x_2 = 4$.
$n = 3$, $x_3 \equiv 10^3 \ (mod\ 8)$, tenemos que $x_3 = 0$.
$n = 4$, $x_0 \equiv 10^4 \ (mod\ 8)$, tenemos que $x_4 = 0$.

De x_3 en adelante todos los valores de x son igual a 0, por lo tanto tenemos el vector adjunto de 8 como $v_8^* = (1, 2, 4, \overline{0})$, donde $\overline{0}$ significa que 0 se repite de forma infinita.

Ejemplo Vamos a verificar si 12296 es divisible por 8 a través del método criterio de divisibilidad general.
Obtenemos el vector asociado de 12296, $v_{12296} = (6, 9, 2, 2, 1, \overline{0})$, ya tenemos el vector adjunto de 8, $v_8^* = 1, 2, 4, \overline{0}$, ahora realizamos la multiplicación de los vectores

$$v_{12296} \cdot v_8^* = 6 + 18 + 8 = 32$$

realizamos nuevamente la operación con el valor resultante 32,

$$v_{23} \cdot v_8^* = 2 + 6 = 8$$

y verificamos que $v_{23} \cdot v_8^*$ es divisible por 8.
Todo el análisis de esta teoría y demostración se encuentran en [54].

4.2. Algoritmo de Fermat

El primer algoritmo moderno de factorización de enteros que presentamos es el algoritmo de Fermat, algoritmo 10. En estos días este algoritmo ya no es usualmente implementado a menos que sepamos que el número de sus factores sea 2 y que están relativamente cerca de la raíz cuadrada del número dado n, pero contiene la clave principal de 2 de los más potentes algoritmos de

factorización hoy en día con factores primos largos, los algoritmos de fracciones continuas y la criba cuadrática.

Este algoritmo también fue modificado por *R. Sherman Lehman* en [55], para factorizar números un poco grandes, ya que permite factorizar un número n en $O(n^{1/3})$ operaciones elementales, donde la suma, resta, multiplicación, división y la raíz cuadrada son consideradas como operaciones elementales.

La idea de Fermat es la siguiente. Si n es un número a ser factorizado y si n puede escribirse como una diferencia de 2 cuadrados perfectos

$$n = x^2 - y^2 \tag{4.4}$$

entonces

$$n = (x - y) \cdot (x + y) \tag{4.5}$$

y hemos logrado romper n dentro de dos factores pequeños. Además, si asumimos que el valor n que queremos factorizar es impar, entonces, toda representación de n tiene un producto de dos enteros planteados por esta vía.

Para ver esto, tenemos que $n = a \cdot b$, donde a y b son impares porque n es impar, tenemos

$$x = \frac{(a+b)}{2} \; ; \; y = \frac{(a-b)}{2} \tag{4.6}$$

Entonces

$$x^2 - y^2 = (a^2 + 2ab + b^2 - a^2 + 2ab - b^2)/4 = a \cdot b = n.$$

El algoritmo de Fermat trabaja en dirección opuesta a la división sucesiva. En el algoritmo de divisiones sucesivas, iniciamos buscando pequeños factores y trabajando hasta la raíz cuadrada de n. Y en el algoritmo de Fermat, nosotros iniciamos buscando valores cercanos a la raíz cuadrada de n y buscamos hacia abajo.

Dado un entero impar n para ser factorizado, buscamos enteros x e y tales que $x^2 - y^2 = n$. Iniciamos con x igual al menor entero mayor que o igual al la raíz cuadrada de n y probamos incrementando y hasta $x^2 - y^2$, bien es igual o menor que n. En el primer caso hemos terminado, en el segundo caso incrementamos x en 1 e iteramos. Continuamos hasta que tengamos éxito. Si nos fijamos r es igual a $x^2 - y^2 - n$. Entonces tenemos éxito cuando $r = 0$.

Este algoritmo mejora por medio del seguimiento de $u = 2x + 1$ y $v = 2y + 1$ en vez de x y y. La variable u muestra los incrementos de r cuando x^2 es reemplazada por $(x+1)^2$, la variable v muestra los decrementos de r cuando y^2 es reemplazada por $(y+1)^2$. Ya que x e y incrementan por 1, u y v incrementan por 2.

En el algoritmo 10 tenemos que s es el menor entero mayor o igual con $\sqrt{n}$. El valor inicial de x es s y el valor inicial de $y = 0$.

Algoritmo 10 Algoritmo de Fermat para factorización de enteros

1: **Entradas**: Entero n para factorizar.
2: $s \leftarrow \lceil \sqrt{n} \rceil$
3: $u \leftarrow 2 \cdot s + 1$
4: $v \leftarrow 1$ $r \leftarrow s \cdot s - n$
5: **while** $\neq 0$ **do**
6: **if** $r > 0$ **then**
7: **while** $r > 0$ **do**
8: $r \leftarrow r - v$
9: $v \leftarrow v + 2$
10: **end while**
11: **end if**
12: **if** $r < 0$ **then**
13: $x \leftarrow x + u$
14: $u \leftarrow u + 2$
15: **end if**
16: **end while**
17: $a \leftarrow (u + v - 2)/2$
18: $b \leftarrow (u - v)/2$
19: **Salida:** a, b

Este algoritmo tiene algunas características interesantes, una de las principales, es que los *loops* no incluyen multiplicación ni división, así que los ciclos son bastante rápidos. El problema, por supuesto, es la gran cantidad de ciclos requeridos, para factorizar por ejemplo,

$$141063954395943949 = 377505979 \cdot 373673431,$$

requiere de 4889 ciclos del ciclo $while(r \neq 0)$ y 1916274 ciclos $while(r > 0)$.

4.3. Algoritmo Pollard Rho

En 1975 *Pollard* desarrollo un algoritmo de factorización[56] basado en la combinación de 2 ideas. La primera idea es la *paradoja del cumpleaños*, que dice que: un grupo de al menos 32 personas (elegidas aleatoriamente) contiene dos personas con la misma fecha de cumpleaños en más del 50 % de los casos, más generalmente si los números son elegidos al azar de un grupo que contiene p números, la probabilidad de tomar el mismo número dos veces es superior al 50 % después de que $1{,}177\sqrt{p}$ números han sido escogidos. El primer duplicado puede ser esperado después de que $c \cdot \sqrt{p}$ números han sido seleccionados para algunas constantes pequeñas c. La segunda idea es: si p es un divisor desconocido de n y x e y son dos enteros que son sospechosos de ser idénticos modulo p, por lo tanto $x \equiv y \ (mod\ p)$, entonces puede ser verificado por medio del calculo del $GCD(|x - y|, n)$, este cálculo puede revelar un factor primo de n, a menos que x e y sean también idénticos modulo n[57].

Tenemos que p es un divisor pequeño de n. Ahora suponemos que existen dos enteros $x, x' \in \mathbb{Z}_n$, tal que $x \neq x'$ y $x \equiv x' \ (mod\ p)$. Entonces $p \leq (x - x', n) < n$, así obtenemos un factor no trivial

de n por medio del computo del gcd.(Notamos que el valor de p no necesita ser conocido antes de tiempo para este método de trabajo).

Suponemos que intentamos factorizar n en primer lugar escogiendo un subconjunto de $X \subseteq \mathbb{Z}_n$, y entonces calculamos $(x - x', n)$ para todos los distintos valores $x,\ x' \in X$. Este método puede ser exitoso si y sólo si el mapeo $x \mapsto x\ mod\ p$ contiene al menos una colisión para $x \in X$. Esta situación se analiza usando la paradoja del cumpleaños, que como ya vimos, existe un 50 % de probabilidades de que haya al menos una colisión, y por lo tanto un factor no trivial de n será encontrado. Sin embargo, con el fin de encontrar una colisión de la forma $x\, mod\ p$, necesitamos calcular $(x - x', n)$. En 2006 *Stephen D. Miller y Ramarathnam Venkatesan*[58] mostraron que el algoritmo de Pollard Rho produce una colisión en un tiempo esperado de $O(\sqrt{n}(log\ n)^3)$.

El algoritmo *Pollard Rho* incorpora una variante de esta técnica que requiere pocas computaciones del GCD y menor memoria. Suponemos que la función f es un polinomio con coeficientes enteros, por ejemplo $f(x) = x^2 + a$, donde a es una constante pequeña, comúnmente se usa el valor de $a = 1$. Asumimos que el mapeo $x \mapsto f(x)\ mod\ p$ se comporta como un mapeo aleatorio (esto por supuesto no es aleatorio, lo que nos dice que estamos presentando un análisis heurístico en lugar de una prueba rigurosa),ver figura 4.1. Tenemos $x_1 \in \mathbb{Z}_n$, y consideramos la secuencia $x_1, x_2, ...,$ donde

$$x_j = f(x_{j-1})\ mod\ n, \tag{4.7}$$

para todo $j \geq 2$. Tenemos que m es un entero, y definimos $X = \{x_1, ..., x_m\}$. Para simplificar, suponemos que X consiste de m distintos residuos modulo n. Ojala sea el caso que X es un subconjunto de m elementos de $\mathbb{Z}_n$.

Buscamos dos distintos valores $x_i,\ x_j \in X$ tal que $(x_j - x_i, n) > 1$. Cada vez que calculamos un nuevo termino x_j en la secuencia, podríamos computar $(x_j - x_i, n)$ para todo $i < j$. Sin embargo, resulta que podemos reducir mucho el número de computaciones de GCD. Ahora describiremos como puede llevarse a cabo.

Suponemos que $x_i \equiv x_j\ (mod\ p)$. Usando el hecho de que f es un polinomio con coeficientes enteros, tenemos que $f(x_i) \equiv f(x_j)\ (mod\ p)$. Recordamos que $x_{i+1} = f(x_i)\ mod\ n$ y $x_{j+1} = f(x_{x_j})\ mod\ n$. Entonces

$$x_{i+1}\ mod\ p = (f(x_i)\ mod\, n)\ mod\ p = f(x_i)\ mod\ p \tag{4.8}$$

porque $p|n$. Similarmente,

$$x_{j+1}\ mod\ p = f(x_j)\ mod\ p. \tag{4.9}$$

Por consiguiente, $x_{i+1} \equiv x_{j+1}\ mod\ p$. Repitiendo este argumento, obtenemos el siguiente importante resultado:

Si $x_i \equiv x_j\ (mod\ p)$, entonces $x_{i+\delta} \equiv x_{j+\delta}\ (mod\ p)$ para todos los enteros $\delta \geq 0$.

Denotamos $l = j - i$, y se deduce que $x_{i'} \equiv x_{j'}\ (mod\, p)$ si $j' > i' \geq i$ y $j' - i' \equiv 0\ (mod\ l)$.

Suponemos que construimos un gráfico G en el conjunto de vértices $\mathbb{Z}_p$, donde, para todo $i \geq 1$, tenemos un extremo dirigido desde $x_i\ mod\ p$ hasta $x_{i+1}\ mod\ p$. Debe existir un primer par $x_i,\ x_j$

con $i < j$ tal que $x_i \equiv x_j \ (mod\ p)$. Por la observación hecha arriba, es fácil ver que el grafico G consiste de una 'cola'

$$x_1\ mod\ p \to x_2\ mod\ p \to \cdots \to x_i\ mod\ p, \tag{4.10}$$

y un ciclo infinitamente repetido de longitud l, teniendo vértices

$$x_i\ mod\ p \to x_{i+1}\ mod\ p \to \cdots \to x_j\ mod\ p = x_i\ mod\ p. \tag{4.11}$$

así G Parece como la letra griega ρ, por lo que esa es la razón para el nombre de *algoritmo rho.*

Ejemplo Suponemos que $n = 10873 = 83 \cdot 131$, $f(x) = x^2 + 1$ y $x_1 = 1$, la secuencia de x_i's inicia de la siguiente forma

1	2	5	26	677	1664	7155
3942	1848	983	9466	764	7428	5583
7872	3158	2424	4357	10065	485	6893

Los valores obtenidos, cuando los reducimos modulo 83, obtenemos lo siguiente:

1	2	5	26	13	4	17
41	22	70	4	17	41	22
70	4	17	41	22	70	4

La primera colisión que se da en la lista es

$$x_6\ mod\ 83 = x_{11}\ mod\ 83 = 4.$$

Por consiguiente el gráfico G consiste de una cola de longitud 6 y un ciclo de longitud 5.

Ya hemos mencionado que nuestra meta es descubrir dos términos $x_i \equiv x_j \ (mod\ p)$ con $i < j$, por la computación del GCD. No es necesario que se descubra la primera aparición de una colisión de este tipo. Para simplificar y mejorar el algoritmo, restringimos nuestra búsqueda de colisiones por medio de $j = 2i$[20]. El algoritmo resultante es presentado como algoritmo 11.

En el ejemplo anterior, la primera colisión modulo 83 ocurre para $i = 6$, $j = 11$, el entero más pequeño $i' \geq 6$ que es divisible por 5 es $i' = 10$. Por consiguiente el algoritmo 11 descubrirá el factor 83 de n cuando se compute $(x_{10} - x_{20}, n) = (983 - 485, 10873) = 83$.

En general, ya que $p < \sqrt{n}$, la complejidad esperada del algoritmo es $O(n^{1/4})$ (Ignorando factores logarítmicos). Sin embargo, nuevamente enfatizamos que es un análisis heurístico, y no una prueba matemática. Por otro lado, la actual ejecución del algoritmo en la práctica es similar a esta estimación.

Es posible que el algoritmo 11 pueda fallar para encontrar un factor no trivial de n. Esto sucede si y sólo si los primeros valores x y x' que satisfacen $x \equiv x' \ (mod\ p)$, satisfacen $x \equiv x' \ (mod\ n)$ (esto es equivalente con $x = x'$, por que x y x' son reducidas modulo n). Podemos estimar heurísticamente que la probabilidad de que esto ocurra es más pequeña que p/n, lo que es bastante pequeño cuando n es largo, por que $p < \sqrt{n}$. Si el algoritmo falla por este camino, es mejor

Algoritmo 11 Algoritmo Pollard rho

1: **Entradas**: Entero n y x_1
2: $x \leftarrow x_1$
3: $x' \leftarrow f(x)\ mod\ n$
4: $p \leftarrow gcd(x - x', n)$
5: **while** $p = 1$ **do**
6: $\quad x \leftarrow f(x)\ mod\ n$
7: $\quad x' \leftarrow f(x')\ mod\ n$
8: $\quad x' \leftarrow f(x')\ mod\ n$
9: $\quad p \leftarrow gcd(x - x', n)$
10: **end while**
11: **if** $p = n$ **then**
12: $\quad$ **Regresa:** Fallo.
13: **else**
14: $\quad$ **Regresa:** (p).
15: **end if**

ejecutarlo nuevamente con valores iniciales diferentes.

En 1980, *Richard Brent* publicó una variante más rápida del algoritmo Pollard Rho[59], y lo denotamos como algoritmo 12.

En el algoritmo 12 tenemos que n es el entero a ser factorizado, *máximo* es el máximo número de ciclos antes de ser abortado, *rango* es el número de veces que usaremos el valor actual de x_1 antes de asignar nuevos valores, *producto* es el producto de las ultimas diez diferencias. En el línea 9 vemos que x_1 no es modificado en el ciclo, de la línea 8 a la línea 18, por lo que permanece fijo, mientras x_2 se modifica.

En el algoritmo 12, si $GCD > 1$ entonces g es un divisor del número n, o el algoritmo no trabaja con el valor de c. Por último en la parte donde inicia la nueva asignación de valores, línea 20, tenemos que, colocamos en x_1 el último valor de x_2, duplicamos el valor del *rango* y encontramos el próximo valor de x_2, e iniciamos nuevamente el ciclo.

Dos situaciones pueden salir mal en el algoritmo 12. Una es que el primer GCD mayor que uno sea n, esto significa que hemos tenido mala suerte al momento de escoger el polinomio $f(x)$ (o que realmente n es primo). Entonces escogemos un nuevo polinomio y lo ejecutamos nuevamente. La otra situación que puede salir mal, es que este algoritmo necesitara un tiempo muy largo para encontrar un divisor. Mientras el periodo del ciclo se espera que sea alrededor de la raíz cuadrada del menor divisor primo, este puede ser tan grande como el divisor primo más pequeño. Este también es el problema, que nosotros no conoceremos el tamaño del más pequeño divisor primo de n. Si es un hecho que es muy extenso, las y_i's tendrán una extensión de ciclo muy prolongado. Podremos estar preparados para abortar y tratar con un diferente polinomio $f(x)$ o un algoritmo completamente diferente.

Como la gran mayoría de algoritmos, este algoritmo no ha escapado a su implementación de una forma paralela como es el trabajo que presenta *R.E. Crandall* en [60], donde con el uso de

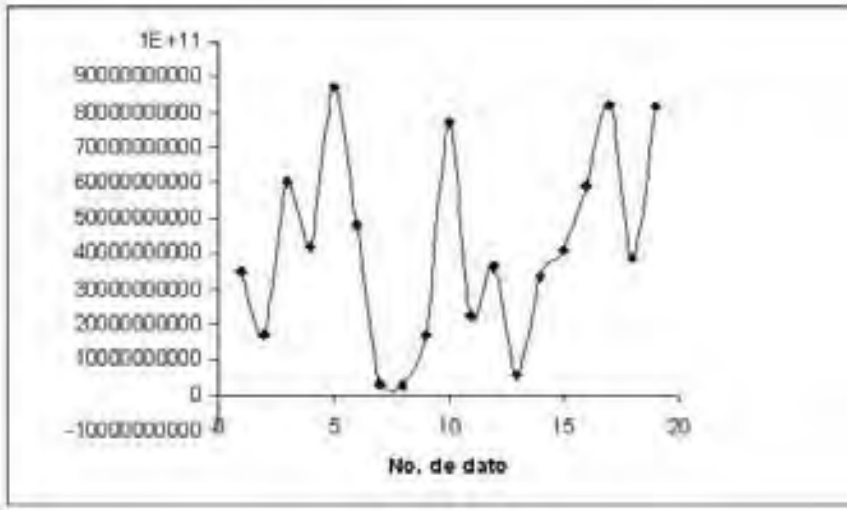

Figura 4.1: Figura que muestra el comportamiento del mapeo $x \mapsto f(x) \ mod \ p$, probado con el valor n=87523897453, el eje de las y muestra el valor del mapeo x.

Algoritmo 12 Versión de Brent del algoritmo Pollard Rho

Entradas: Entero n y c
$x_1 \leftarrow 2$
$x_2 \leftarrow 4 + c$
$rango \leftarrow 1$
$producto \leftarrow 1$
$termino \leftarrow 0$
while $termino \leq maximo$ **do**
  for $j = 1$ hasta $j = rango$ **do**
    $x_2 \leftarrow (x_2 \cdot x_2 + c) \ mod \ n$
    $termino \leftarrow termino + 1$
    if $termino \% 10 = 0$ **then**
      $g \leftarrow GCD(n, producto)$
      if $g > 1$ **then**
        return g
        break
      end if
      $producto \leftarrow 1$
    end if
  end for
  $x_1 \leftarrow x_2$ /*Inicio de reseteo de variables*/
  $rango \leftarrow 2 \cdot rango$
  for $j = 1$ hasta $rango$ **do**
    $x_2 \leftarrow (x_2 \cdot x_2 + c) \ mod \ n$
  end for
end while

m procesos en paralelo, reduce el tiempo esperado de factorización sólo por $\sqrt{m}$.

4.4. Algoritmo Pollard $p-1$

El algoritmo Pollard $p-1$(algoritmo13),fue diseñado por Pollard en 1974[61], es aproximado al método de Pollard Rho, pero reemplaza la paradoja del cumpleaños por el teorema pequeño de Fermat.

Algoritmo 13 Algoritmo Pollard $p-1$

1: **Entradas**: Entero n y B
2: $a \leftarrow 2$
3: **for** $j \leftarrow$ hasta B **do**
4: $\quad a \leftarrow a^j \ mod \ n$
5: **end for**
6: $d \leftarrow (a-1, n)$
7: **if** $1 < d < n$ **then**
8: $\quad$ **Regresa:** (d)
9: **else**
10: $\quad$ **Regresa:** fallo
11: **end if**

Definición 4.1 *Un positivo entero es llamado $B-$suave si ninguno de sus factores primos son mayores que B.*

Por ejemplo el número 3150, tiene una factorización de $2 \cdot 3^2 \cdot 5^2 \cdot 7$, por lo que 3150 es un 7-suave, ya que ninguno de sus factores primos es mas grande que 7.

Suponemos que p es un divisor primo de n, y suponemos que $q \leq B$, donde B es un limite que nosotros especificamos, para cualquier primo $q|(p-1)$.

Entonces puede ser el caso que

$$(p-1)|B! \tag{4.12}$$

Al final del ciclo *for*, tenemos que

$$a \equiv 2^{B!} \ (mod \ n). \tag{4.13}$$

Ya que $p|n$, puede ser el caso que

$$a \equiv 2^{B!} \ (mod \ p). \tag{4.14}$$

Ahora,

$$2^{p-1} \equiv 1 \ (mod \ p) \tag{4.15}$$

por el teorema de Fermat. Ya que $(p-1)|B!$, se deduce que

$$a \equiv 1 \ (mod \ p), \tag{4.16}$$

y por lo tanto $p|(a-1)$. Ya que también tenemos que $p|n$, vemos que $p|d$, donde $d = (a-1, n)$. El entero d será un divisor no trivial de n (a menos que $a = 1$). Habiendo encontrado un factor no trivial d, podemos entonces proceder a intentar factorizar d y n/d si se espera que sean compuestos.

Ejemplo Suponemos que $n = 101412777941$. Si aplicamos el algoritmo 13 con $B = 318453$, entonces encontramos que $a = 52249469107$ y el calculo para d es $d = 249427$. La factorización completa en números primos es

$$101412777941 = 249427 * 406583$$

En este ejemplo la factorización se lleva a cabo por que $p-1 = 249426$ tiene solamente pequeños factores primos:

$$249426 = 3 * 3 * 3 * 2 * 31 * 149.$$

En el algoritmo Pollard $p-1$, existen $B-1$ exponenciaciones modulares, cada uno requiriendo al menos 2 $log_2 \ B$ multiplicaciones modulares usando el algoritmo del cuadrado y multiplicación. El GCD puede ser computado en tiempo $O((log \ n)^3)$usando el algoritmo extendido de Euclides. Por lo tanto, la complejidad del algoritmo es $O(B \ log \ B(log \ n)^2 + (log \ n)^3)$. Si el entero B es $O((log \ n)^i)$ para algún entero fijo i, entonces el algoritmo es en verdad un algoritmo de tiempo polinomial(como una función de $log \ n$); sin embargo, para tal elección de B la probabilidad de éxito será demasiado pequeña. Por otro lado, si incrementamos el tamaño de B drásticamente, decimos que $\sqrt{n}$, entonces el algoritmo es garantizado a ser exitoso.

Así, el inconveniente de este método es que requiere que n tenga un factor primo p tal que $p-1$ tiene solamente factores primos pequeños. Podrá ser muy fácil construir un modulo RSA $n = pq$ que podrá resistir factorizaciones por este método. Podemos empezar por encontrar un primo largo p_1 tal que $p = 2p_1 + 1$ también es primo, y un primo largo q_1 tal que $q = 2q_1 + 1$ también es primo. Entonces el modulo RSA $n = pq$ será resistente a factorizaciones por medio del algoritmo $p-1$.

Una muestra de lo que Pollard p-1 puede hacer es el factor que encontró Baillie del número de Mersenne $M_{257} = 2^{257} - 1$

$$p_{25} = 1155685395246619182673033,$$

donde $p_{25} - 1 = 2^3 \cdot 3^2 \cdot 19^2 \cdot 47 \cdot 67 \cdot 257 \cdot 439 \cdot 119173 \cdot 1050151$, que como vemos esta compuesto de factores pequeños.

4.5. Algoritmo de Williams $p+1$

En 1982, *H.C. Williams* describe un método de factorización análogo al método $p-1$ de Pollard en [7], este algoritmo es conocido como el algoritmo $p+1$(algoritmo 14). En esta sección retomamos el trabajo de Williams.

Tenemos un valor n que tiene un divisor primo p tal que $p+1$ tiene solamente divisores primos pequeños. Este método permite determinar p dado n.

4.5.1. Funciones de Lucas

Antes de iniciar con el algoritmo vamos a presentar unas propiedades básicas de las funciones de Lucas, no vamos a ahondar en el tema solo presentamos algunas propiedades de dichas funciones.

Tenemos que P y Q son enteros, y tenemos que α y β son los ceros en $x^2 - Px + Q$, definimos la función de Lucas como

$$U_n(P,Q) = (\alpha^n - \beta^n)/(\alpha - \beta),\ V_n(P,Q) = \alpha^n + \beta^n. \tag{4.17}$$

También ponemos $\Delta(\alpha-\beta)^2 = P^2-4Q$, esas funciones satisfacen un largo número de identidades que requeriremos . Se indican a continuación

$$\begin{cases} U_{n+1} = PU_n - QU_{n-1}, \\ V_{n+1} = PV_n - QV_{n-1} \end{cases} \tag{4.18}$$

$$\begin{cases} U_{2n} = V_n U_n, \\ V_{2n} = V_n^2 - 2Q^n, \end{cases} \tag{4.19}$$

$$\begin{cases} U_{2n-1} = U_n^2 - QU_{n-1^2}, \\ V_{2n-1} = V_n V_{n-1} - PQ^{n-1}, \end{cases} \tag{4.20}$$

$$\begin{cases} \Delta U_n = PV_n - 2QV_{n-1}, \\ V_n = PU_n - 2QU_{n-1}, \end{cases} \tag{4.21}$$

$$\begin{cases} U_{m+n} = U_m U_{n+1} - QU_{m-1}U_n, \\ \Delta U_{m+n} = V_m V_{n+1} - QV_{m-1}V_n, \end{cases} \tag{4.22}$$

$$\begin{cases} U_n(V_k(P,Q), Q^k) = U_{nk}(P,Q)/U_k(P,Q), \\ V_n(V_k(P,Q), Q^k) = v_{nk}(P,Q). \end{cases} \tag{4.23}$$

Estas identidades pueden ser verificadas por directa sustitución de 4.17, sabiendo que $P = \alpha + \beta$, y $Q = \alpha\beta$. También notamos que si $(N,Q) = 1$ y $P'Q \equiv P^2 - 2Q(mod\ n)$, entonces $P' \equiv \alpha/\beta + \beta/\alpha$ y $Q' \equiv \alpha/\beta \cdot \beta/\alpha = 1$, por lo tanto

$$U_{2m}(P,Q) \equiv PQ^{m-1}U_m(P',1)\ (mod\ n). \tag{4.24}$$

Teorema 4.2 *Si p es primo, $p \not| Q$ y el símbolo de Legendre $(\Lambda/p) = \epsilon$, entonces*

$$U_{(p-\epsilon)m}(P,Q) \equiv 0\ (mod\ p)$$

$$U_{(p-\epsilon)m}(P,Q) \equiv 2Q^{m(1-\epsilon)/2}\ (mod\ p).$$

4.5.2. Cuestiones matemáticas del algoritmo $p+1$

Suponemos que p es un divisor primo de n y

$$p = \left(\prod_{i=1}^{k} q_i^{\alpha_i}\right) - 1, \tag{4.25}$$

donde q_i es el i-esimo primo y $q_i^{\alpha_i} \leq B_1$. Si R es definido como

$$R = \prod_{i=1}^{k} q_i^{\beta_i}, \tag{4.26}$$

tenemos $p+1|R$, por el teorema 4.2 vemos que si $(Q,n)=1$ y $(\Delta/p)=-1$, entonces $p|U_R(P,Q)$, y por consiguiente $p|(U_R(P,Q),n)$.

Para encontrar $U_R(P,Q)$ pueden ser usadas las formulas 4.18, 4.19 y 4.20 junto con la formula 4.21 para obtener

$$\begin{gathered} U_{2n-1} = U_n^2 - QU_{n-1}^2, \\ U_{2n} = U_n(PU_n - 2QU_{n-1}), \\ U_{2n+1} = PU_{2n} - QU_{2n-1}. \end{gathered}$$

Si $p|(U_R(P,Q))$, entonces por 4.19 $p|U_{2R}(P,Q)$, así de la ecuación 4.24 tenemos $p|U_R(P',1)$. Por el teorema 4.2, también tenemos

$$V_{(p-\epsilon)m}(P,1) \equiv 2 \ (mod\ p);$$

por lo tanto, si $p|U_R(P,1)$, entonces $p|(V_R(P,1)-2)$. Asumiremos que $Q=1$.

El primer paso de $p+1$ es el siguiente:
Tenemos $R = r_1 r_2 \dots r_m$ y buscamos p_0 tal que $(P_0^2-4,n)=1$. Definimos $V_n(P) = V_n(P,1)$, $U_n(P) = U_n(P,1)$ y $P_j \equiv V_{r_j}(P_{j-1})\ (mod\ n)(j=1,2,\dots,m)$.

Por la segunda formula de 4.23 vemos que

$$P_m \equiv V_R(P_0)\ (mod\ n). \tag{4.27}$$

Ahora calculamos $(p_m - 2, n)$. Para encontrar $V_r = V_r(P)$ desde P necesitamos solamente usar las formulas

$$\begin{cases} V_{2f-1} \equiv V_f V_{f-1} - P, \\ V_{2f} \equiv V_f^2 - 2, V_{2f+1} {}_f^2 - V_f V_{f-1} - P\ (mod\ n). \end{cases} \tag{4.28}$$

Tenemos

$$r = \sum_{l=0}^{t} b_l 2^{t-l}\ (b_l = 0,1),$$

$f_0 = 1$, y $f_{k+1} = 2f_k + b_{k+1}$, entonces $f_t = r$. También, si $V_0(P) = 2$, $v_i(P) = P$, entonces para encontrar el par $(V_{f_{k+1}}), V_{f_{k+1}-1}$ desde (V_{f_k}, V_{f_k-1}) necesitamos usar sólo la formula

$$(V_{f_{k+1}}, V_{f_{k+1}-1}) = \begin{cases} (V_{2f_k}, V_{2f_k-1}) \textit{ donde } b_{k+1} = 0, \\ (V_{2f_k+1}, V_{2f_k}) \textit{ cuando } b_{k+1} = 1, \end{cases} \quad (4.29)$$

junto con 4.28.

Algoritmo 14 Algoritmo de Williams $p + 1$

1: **Entradas**: Entero n y max.
2: Escoge p aleatoriamente.
3: $cont \leftarrow 1$;
4: $v \leftarrow p$;
5: **while** $GCD(v - 2, n) = 1$ y $cont \leq max$ **do**
6: **for** $i = 1$ to $i = 10$ **do**
7: $v \leftarrow siguiente_v(1, p, cont, n)$;
8: $p \leftarrow v$;
9: $cont \leftarrow cont + 1$;
10: **end for**
11: **end while**
12: **Regresa:** $GCD(v - 2, n)$;

En al algoritmo 14 tenemos que max es el número máximo de ciclos que se realizan antes de abortar, y $siguiente_v(1, p, cont, n)$ es una función que calcula el próximo valor de v dada por el algoritmo 15.

Donde n es un residuo cuadrático modulo p, h es escogido de tal forma que $h^2 - 4n$ no sea un residuo cuadrático modulo p y j es un valor aleatorio.

4.6. Método de Curvas elípticas

4.6.1. Curvas Elípticas

Una curva elíptica[2]se define como una ecuación $y^2 = x^3 + Ax + B$, donde A y B son constantes y cumplen con $4A^3 + 27B^2 \neq 0$ Esta ecuación es conocida como ecuación de *Weierstrass*. La figura 4.1 muestra la forma que puede tomar la gráfica de una curva elíptica dependiendo de los valores de A y B.

Suma de puntos

Supongamos que tenemos dos puntos P y Q en una curva. Observe la Figura 4.3. Si trazamos una línea entre estos dos puntos veremos que corta a la curva en un tercer punto. Si reflejamos este punto a través del eje (cambiamos el signo de su coordenada y), obtendremos un punto R. Dicho punto R es la suma de P y Q. Esto se conoce como ley del grupo. Pero no es necesario

[2]Esta pequeña introducción de Curvas elípticas, el apartado de *Suma de puntos* y *Multiplicación de puntos*, así como las tres figuras que se presentan fueron obtenidas de [62], que se puede descargar de http://www.opendomo.com/dlerch/

Algoritmo 15 Algoritmo que calcula el siguiente valor de v

Entradas: Entero n, h, j y p;
$m \leftarrow n$;
$v \leftarrow h$;
$w \leftarrow (h^2 - 2 \cdot m) \ mod \ p$;
$i \leftarrow 0$
while $j > 0$ **do**
    $i \leftarrow i + 1$;
    $b_i \leftarrow j \ mod \ 2$;
    $j \leftarrow \lfloor j/2 \rfloor$;
end while
$t \leftarrow i$
for $k \leftarrow t - 1$ to 1 **do**
    $x \leftarrow (v * w - h * m) \ mod \ p$;
    $v \leftarrow (v^2 - 2m) \ mod \ p$;
    $w \leftarrow (w^2 - 2 * n * m) \ mod \ p$;
    $m \leftarrow m^2 \ mod \ p$;
    if $b_k = 0$ **then**
        $w \leftarrow x$;
    else
        $v \leftarrow x$; $m \leftarrow n * m \ mod \ p$
    end if
end for
Salida: v.

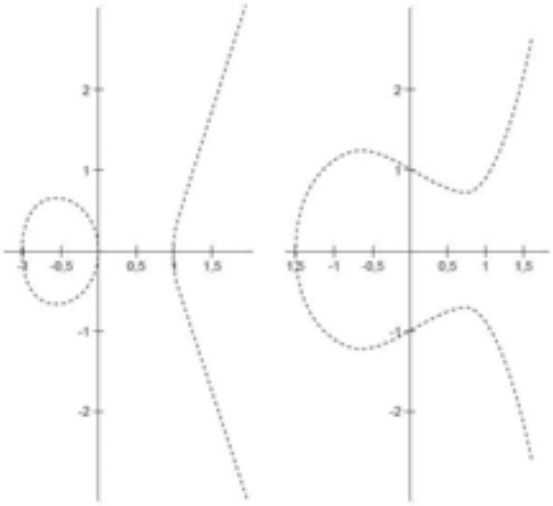

Figura 4.2: Grafica de la forma de una curva elíptica.

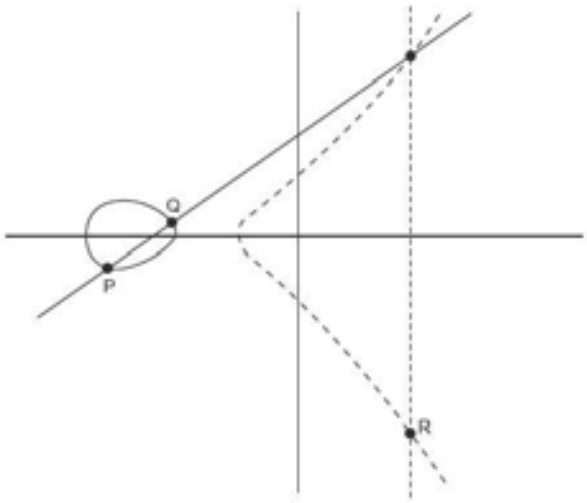

Figura 4.3: Suma de puntos en una curva elíptica

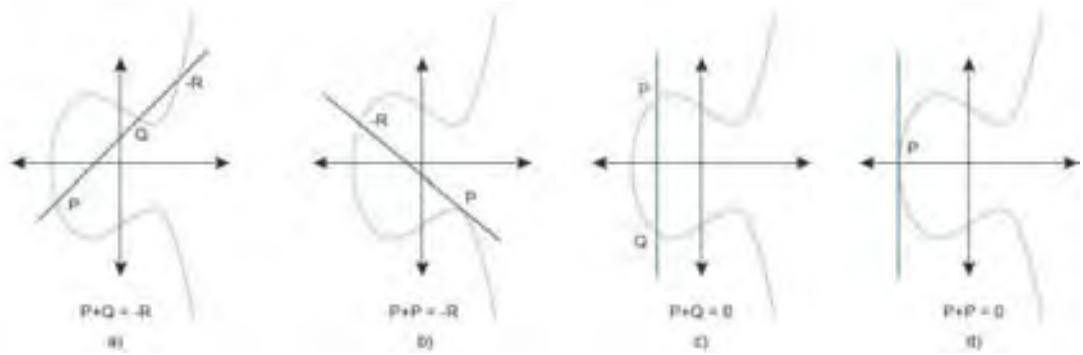

Figura 4.4: Distintos casos de suma de puntos en una curva elíptica.

disponer de dos puntos para encontrar un tercero, pues a partir de un solo punto podemos obtener otro. Imaginemos que en el caso anterior $P = Q$. Entonces la línea en lugar de cortar la curva por los puntos P y Q solo lo haría por un punto, es decir, sería una tangente, como vemos en la figura 4.4.b, en este caso la línea que dibujamos es una tangente a la curva que corta en el punto $-R$. De esta manera $P + P = 2P = -R$. En la figura 4.4.c se ilustra el caso en el que no existe una línea que pase por P y por Q que corte en otro punto a la curva elíptica. En este caso se dice que dicha línea corta la curva en un punto 0 situado en el infinito, es decir $P + Q = 0$ o $P + Q = \infty$. En 4.4.d se presenta un caso similar, en el que un único punto al ser sumado por el mismo corta la curva en el infinito.

Multiplicación de puntos

En el apartado anterior hemos aprendido a sumar puntos en una curva elíptica. En las curvas elípticas podemos realizar también la multiplicación de un punto de la curva por un escalar, es decir un número k por un punto P. Esto es más sencillo de lo que puede parecer. Supongamos que queremos calcular kP donde $k = 27$, podemos realizar un doblado de puntos de la forma siguiente: $P,\ 2P = P + P,\ 4P = 2P + 2P,\ 8P = 4P + 4P,\ 16P = 8P + 8P,\ 27P = 16P + 8P + 2P + P,$

obteniendo el punto resultante de multiplicar P por un escalar k. Este procedimiento permite calcular kP para valores de k de varios cientos de dígitos muy rápidamente. El único problema consiste en el gran tamaño que adquieren las coordenadas de los puntos que se van calculando. Pero esto no ocurre al trabajar en campos finitos como suele hacerse en criptografía de curva elíptica.
Una curva definida sobre un campo finito tiene un número finito de puntos en ella. A este número de puntos lo llamamos *orden de la curva*. Por otra parte, el número k más pequeño (pero mayor que 0) que multiplicado por un punto P da 0(infinito), se conoce como orden de ese punto. No hay confundir el orden de la curva con el orden del punto.

4.6.2. ECM para factorización

En 1987, *Lenstra* publicó el *Método de las curvas elípticas*[63](ECM por sus siglas en ingles) para factorización de números enteros, basándose en el algoritmo de Pollard $p-1$ (ver pag. 50). La mayor desventaja de Pollard $p-1$, es que sólo trabaja eficientemente cuando el número a ser factorizado tiene factores $p-1$ con factores pequeños, por el contrario ECM no tiene esta desventaja[57].

Hoy en día este método de factorización es uno de los mas populares, y rápidos métodos de búsqueda de factores primos, también es altamente adecuado para computarlo paralelamente, ya que el proceso de factorización envuelve un gran número de pruebas independientes que pueden ser implementadas en paralelo[64].

Podemos generalizar este algoritmo mediante el uso de otros grupos. Por ejemplo, escogiendo $a,\ n \in \mathbb{Z}_n$, podemos considerar la curva elíptica $E_{a,b}$ en $\mathbb{Z}_p$ haciendo todos los cálculos modulo n ya que las computaciones en el modulo p están "ocultas" en el modulo n. El orden de este grupo es un número aleatorio dentro del rango $p+1\pm 2\sqrt{p}$. Si este orden pasa a ser *B-suave*, podemos aplicar exactamente el mismo algoritmo con una complejidad de $O(B)$ operaciones aritméticas. El éxito de esta aproximación corresponde a la probabilidad de que el orden de el grupo $e_{a,b}\ mod\ p$ es *B-suave* para valores aleatorios a y b, lo que es estimado a ser u^{-u} donde $u=\frac{log\ p}{log\ B}$. Un análisis heurístico[3], muestra que este algoritmo tiene una complejidad de $O(e^{\sqrt{(1+O(1))log\ n\ log\ log\ n}})$, que es asintóticamente más pequeño que todas las complejidades de factorizaciones que hemos visto hasta ahora.

Tenemos $N>1$ es un número compuesto, con $(N,6)=1$. Este algoritmo intenta encontrar un factor no trivial de N. El método depende del uso de las curvas elípticas y es presentado como aloritmo 16.

El calculo de $kP\ mod\ N$ puede ser realizada en $O(log\ k)$ duplicados y sumas.

Como para el método $p-1$, uno puede mostrar que dado un par (E,P) es probable que el algoritmo dado sea exitoso si n tiene un factor primo p el cual es compuesto de pequeños primos

[3]La heurística es el uso de técnicas basadas en la experiencia, en juicios intuitivos o, simplemente, en el sentido común, para la resolución de un problema. Debido al rigor de las matemáticas, muchos matemáticos no aceptan a la heurística, por lo que le han restado importancia al "estudio del descubrimiento", como también se le ha denominado.

Algoritmo 16 Algoritmo para factorización cun curvas elípticas.

1: Escogemos un par aleatorio (E, P), donde E es una curva elíptica $y^2 = x^3+az+b$ sobre Z/NZ, y $P(x, y) \in E(Z/NZ)$ es un punto en E: Esto es, escogemos $a, x, y \in Z/NZ$ de forma aleatoria, y establecemos $b^2 - x^3 - ax$.
2: **if** $gcd(4a^3 + 27b^2, N) \neq 1$ **then**
3: E no es una curva eliptica, y empezamos nuevamente y escogemos un nuevo par (E, P).
4: **end if**
5: Elegimos un entero positivo k que es divisible por muchos factores primos, por ejemplo $k = lmc(1, 2, \cdots, B)$ o $k =!B$ para un adecuado limite B el valor mas grande B es el que con más probabilidad tendrá éxito en la producción de un factor.
6: Calcula el punto $kP \in E(Z/ZN)$. Usamos la siguiente formula para calcular $P_3(x_3, y_3) = P_1(x_1, y_1) + P_2(x_2, y_2)\ mod\ N$:
7: $(x_3, y_3) = (\lambda^2 - x_1 - x_2\ mod\ N,\ \lambda(x_1 - x_3) - y_1\ mod\ N)$
8: donde $\lambda = \begin{cases} \frac{m_1}{m_2} = \frac{3x_1^2+a}{2y_1} mod\ N & si\ P_1 = P_2 \\ \frac{m_1}{m_2} = \frac{y_1-y_2}{x_1-x_2} mod\ N & de\ otra\ forma \end{cases}$
9: **if** $kP \equiv O_E(mod\ N)$ **then**
10: Establecemos $m_2 = z$ y calculamos $d = (z, N)$
11: Regresamos al paso 1 para escoger un nuevo valor de a o para escoger un nuevo par (E, P).
12: **end if**
13: **if** $f < d < N$ **then**
14: d es un factor no trivial de N.
15: **end if**
16: **Return** d.

solamente. La probabilidad para que esto suceda incrementa con el número de pares (E, P) que se realicen.

Ejemplo Vamos a factorizar $N = 187$ con el método de las curvas elípticas.

1. Escogemos $B = 3$, y por lo tanto $k = lcm(1, 2, 3)$=6. Tenemos que $P = (0, 5)$ es un punto en la curva elíptica E: $y^2 = x^3 + x + 25$ lo que satisface $GCD(N, 4a^3 + 27b^2) = GCD(187, 16879) = 1$ (Notamos que $a = 1$ y $b = 25$).

2. Ya que $k = 6 = 110_2$ calculamos $6P = 2(P + 2P)$ de la siguiente manera:

   Calculamos $2P = P + P = (0, 5) + (0, 5)$:

   $$\begin{cases} \lambda = \frac{m_1}{m_2} = \frac{1}{10} \equiv 131 \ (mod\ 187) \\ x_3 = 144 \ (mod\ 187) \\ y_3 = 18 \ (mod\ 187) \end{cases}$$

   Así $2P = (144, 18)$ con $m_2 = 10$ y $\lambda = 131$.

   Ahora calculamos $3P = P + 2P = (0, 5) + (144, 18)$:

   $$\begin{cases} \lambda = \frac{m_1}{m_2} = \frac{13}{144} \equiv 178 \ (mod\ 187) \\ x_3 = 124 \ (mod\ 187) \\ y_3 = 176 \ (mod\ 187) \end{cases}$$

   Así $3P = (124, 176)$ con $m_2 = 144$ y $\lambda = 178$.

   Calculamos $6P = 2(3P) = 3P + 3P = (124, 176) + (124, 176)$;

   $$\lambda = \frac{m_1}{m_2} = \frac{46129}{352} \equiv \frac{127}{165} \equiv O_E \ (mod\ 187).$$

   Tenemos $m_1 = 127$ y $m_2 = 165$, así el inverso modular para $127/165$ *modulo* 187 no existe; pero es lo que exactamente nosotros queremos, este tipo de falla es llamada *falla pretendida*. Ahora $z = m_2 = 165$.

3. Calculamos $d = GCD(N, z) = GCD(187, 165) = 11$, ya que $1 < 11 < 187$, 11 es un factor primo de 187.

Dos memorables factorizaciones que se han obtenido utilizando ECM han sido encontrar los factores del décimo y onceavo número de Fermat[65]. En 1988 Brent encontró un factor de 21 dígitos y uno de 22 dígitos, para completar la factorización de F_{11}

$$2^{2^{11}} + 1 = 319489 \cdot 974849 \cdot 167988556341760475137 \cdot 3560841906445833920513 \cdot p_5 64,$$

donde $p_5 64$ denota un factor primo de 64 dígitos decimales, y en 1995 encontró un factor de 40 dígitos decimales de$F^{2^{10}} + 1$ que completa la factorización de F_{10}

$$2^{2^{10}} + 1 = 45592577 \cdot 6487031809 \cdot 4659775785220018543264560743076778192897 \cdot p252;$$

4.7. Resultados

Cuadro 4.1: Tiempos de ejecución de los algoritmos de factorización, dados en milisegundos. Donde *** significa que el algoritmo no obtuvo resultados en un tiempo menor a 12.5 horas(tiempo tomado arbitrariamente como referencia).

No. de prueba	Valor a factorizar	Factores	Pollard Rho	Pollard Rho modif.	Pollard $p-1$	p+1
1	65421331	491*133241	0	0	0	0
2	18446744073709551617	274177*67280421310721	31	0	***	0
3	87456732187654678934621654 1	3^{11}*43*103*3373*11015803181-9653	16	0	31	15
4	97523618768761361349877625 3-4123	111*296347*20142771413*1471-8619219363	3515	421	2718	1438
5	88965619529649331698279681 5-042678003	197002597249*47635010587*94-803416684681	17468	5984	5984	2890
6	20603186336308294025118560 7-107809124597	187333846633*4866979762781*-225974065503889	142438	19813	2144840	1594
7	17493809672325171628455215 9-44748648783155973674354767 8-63721269	8857714771093*7195712273391-89*7901346123803597*3473664-17511089201	10143800	1100500	***	13984
8	15230139750641300099827407 2-02049476338575050603049585 26-64642830161524497717837104 0-150931099	44185520789894155033573*389-132418765025689600 1*5319902-5841281128499153*1665032891-0366149531471	***	***	10953	***

Cuadro 4.2: Tiempos en los que se encontraron los factores de la prueba 7 de la tabla 4.1 en mili segundos. Donde *** significa que el algoritmo no obtuvo resultados en un tiempo mínimo de 12.5 horas(tiempo tomado arbitrariamente como referencia).

Algoritmos	Tiempos en los que se encontraron los factores:			
	8857714771093	719571227339189	7901346123803597	347366417511089201
$p-1$	0	***	***	$126 \cdot 10^5$

Las pruebas 7 y 8 no de la tabla 4.1 no obtuvieron una factorización en un tiempo de referencia minimo de 12.5 horas, cada uno de los algoritmos encontro algun factor especifico los cuales se detallan en los cuadros 4.2 y 4.3, para las pruebas 7 y 8 respectivamente.

En el cuadro 4.2 observamos que, el algoritmo $p-1$ encuentra el primer factor 8857714771093 de forma prácticamente instantánea (0 ms), el segundo factor 347366417511089201 lo encuentra en un tiempo de aproximadamente 3.5 horas, después de estos dos factores el algoritmo tarda demasiado tiempo en encontrar el siguiente factor. Notamos que si al primer factor le restamos 1, obtenemos una factorización de la forma $2^2 \cdot 37 \cdot 627 \cdot 67 \cdot 1424681$ el segundo factor encontrado menos 1 tiene una factorización en la forma $2^4 \cdot 5^2 \cdot 7^2 \cdot 127 \cdot 1979 \cdot 70515119$, por el contrario, los factores que no encontró el algoritmo $p-1$ restandoles 1 nos dan una factorización de la forma $2^2 \cdot 7 \cdot 67 \cdot 383566752313$ y $2^2 \cdot 22483 \cdot 87859117153$, y vemos que no encontró la factorización

Cuadro 4.3: Tiempos en los que se encontraron los factores de la prueba 8 de la tabla 4.1 en mili segundos. Donde *** significa que el algoritmo no obtuvo resultados en un tiempo mínimo de 12.5 horas(tiempo tomado arbitrariamente como referencia).

Algoritmos	Tiempos en los que se encontraron los factores:			
	44185520789894155033-573	38913241876502568960-01	531990258412811284 99-153	16650328910366149531-471
Pollard Rho	***	***	***	***
Pollard Rho modif.	***	***	***	***
$p+1$	***	***	10406	24156

completa debido a que dos de sus factores primos, al restarle 1, tienen un factor primo grande, y no importó mucho el tamaño en dígitos de los factores primos de n.

En el cuadro 4.3 observamos que los algoritmos de Pollard Rho y Pollard Rho modificado no pudieron encontrar ni un sólo factor primo de n en el tiempo de referencia, por otro lado, el algoritmo de Williams encontró el factor 53199025841281128499153 en 10406 ms, el factor $1665032891036614953 1471$ en 24156 ms, pero después de 12.5 horas de ejecución el algoritmo no pudo encontrar ninguno de los factores primos restantes.

Como vemos en el cuadro 4.1 existen diferencias en las formas de trabajo de los algoritmos presentados, que dependiendo del número que se quiera factorizar podríamos utilizar uno u otro algoritmo. Tenemos que, el algoritmo Pollard Rho trabaja bien con números que contengan factores primos pequeños, alrededor de 13 o 14 dígitos decimales, pero con el algoritmo modificado de Pollard Rho encontramos esos mismos valores pero en un tiempo de ejecución mucho menor, esos dos algoritmos no presentan características especiales para la búsqueda de los factores primos, salvo que estos sean pequeños. Por el lado contrario tenemos los algoritmos de Pollard $p-1$ y el algoritmo de Williams $p+1$, estos dos algoritmos como podemos ver, encuentran factores primos más grandes,22 ó 23 dígitos decimales, pero, no son capaces de encontrar todos los factores primos con esas longitudes, sólo encontrarán los factores primos que cumplan ciertas características. Por ejemplo, el algoritmo Pollard $p-1$, es capaz de encontrar el factor primo $p = 3891324187650256896001$ debido a que, al restarle 1 da como resultado un valor compuesto cuya factorización esta dada con factores primos pequeños $2^{15}*3*5^3*17*679*11*19*4243*30937$, por el contrario, si a p le sumamos 1 obtenemos un compuesto con una factorización igual a $2*31*566557*110780192194603$, por lo que el algoritmo de Williams no es capaz de encontrar p ya que $p+1$ tiene factores primos grandes. Del mismo modo el algoritmo de Williams puede encontrar $p = 719571227339189$ ya que $p+1 = 2*3*5*41039*10133*57679$ y el algoritmo de Pollard no puede encontrarlo tan fácilmente pues $p-1 = 2^2*7*67*383566752313$.

El algoritmo de Pollard $p-1$ no es capaz de encontrar la factorización del sexto número de Fermat $F_6 = 2^{64}+1$, aunque $F_6 - 1$ tiene solamente factores pequeños 2^{64}, este valor es el segundo valor de el cuadro 4.1.

Por otro lado tenemos los algoritmos de Fermat y el ECM, los cuales son muy diferentes entre sí, ya que, mientras ECM es capaz de encontrar factores primos p con más de 30 dígitos

decimales sin importar las características de éstos, el algoritmo de Fermat, también puede encontrar factores primos de varios dígitos siempre y cuando sean cercanos a la raíz cuadrada de n. Además el algoritmo de Fermat presenta otro inconveniente, como este algoritmo sólo encuentra dos factores primos a la vez, en ciertos casos es capaz de encontrar dos factores cercanos a la raíz cuadrada de n pero no necesariamente serán primos dichos factores, tal es el caso de 1524157173786973067287101 que, utilizando el algoritmo de Fermat, nos muestra una factorización igual a $1234567346571 * 1234567865431$, pero, ninguno de los dos factores es primo, la factorización real es $1989 * 30869 * 341827 * 72621639143$.

Como resultado indirecto, tenemos que, en el trabajo de Williams [7], se encuentra el número 328006342461 que es mencionado como primo, pero este número en realidad es compuesto con una descomposición igual a $3 * 7^2 * 17 * 131255039$, por lo que no es primo.

Este dato nos muestra pequeños errores que en ocasiones tienen los algoritmos de factorización en el momento de buscar factores primos, ya que existe la posibilidad de que no encuentre factores primos como tal, sino que, encuentre "factores" compuestos, por lo que, al retomar los resultados obtenidos previamente verificamos la realidad de tales datos o, al menos, conocemos que ha cambiado con el uso de equipos de computo con mayor potencia de calculo y almacenamiento(tal es el caso de los factores primos encontrados con los algoritmos $p-1$ y $p+1$ expuestos en la página 3.

Capítulo 5

Algoritmo de factorización que busca factores dentro de los puntos cercanos a la $\sqrt[x]{n}$

Sabemos que existen infinitud de números primos, lo que no sabemos es cual es la distribución de estos, no existe hasta el momento una formula o un método para determinar dónde hay o no hay un número primo. Sin embargo podemos conocer el valor aproximado de cuantos números primos existen menores que un valor dado x, denotado por $\pi(x)$, donde:

$$\pi(x) = \frac{x}{ln\ x}, \tag{5.1}$$

como mencionamos, el resultado es aproximado y no podemos conocer el valor real de $\pi(x)$[1].

Bertrand postulo en 1845, que para cualquier número $n > 3$, existe un número primo p que satisface $n < p < 2n - 2$ o también expresado como $\pi(2n) - \pi(n) > 0$ para todo $n \geq 1$. El verificó esto para $n < 3,000,000$. Y *Tchevichef* probó el postulado de Bertrand en 1850.

Por otro lado, *Leo Moser* presentó una extensión al postulado de Bertrand en [66] que nos dice que para todo entero positivo m existe un número primo p que cumple $3 \cdot 2^{2r-1} < p < 3 \cdot 2^{2r}$.

Como hemos visto el metodo de *Trial division* o divisiones sucesivas, es el método más eficaz en cuanto a determinar si un valor es o no divisor de n, también sabemos, que este método es útil sólo para valores pequeños, ya que, cuando lo usamos en valores grandes es ineficiente. La implementación paralela del método de Trial división es extremadamente sencilla. Ya que con P procesadores podemos ejecutar P sucesiones en paralelo[27]. Por lo que el campo de escaneo aumenta considerablemente conforme aumenta el número de procesadores.

Basándonos en estas características proponemos una nueva forma de buscar factores primos de números grandes, realizando una *doble división sucesiva*, esto es, si en el método de Trial division verificamos cada uno de los pósibles factores $d = 1, 2, 3 \ldots$, hasta que ocurra:

1. $d > n^{1/2}$, en tal caso n es primo,

[1]Existen varias formulas para concer el valor aproximado de $\pi(x)$ que podemos consultar en [52] y [10]

2. $d < n$ y $d|n$, en tal caso d es un factor primo de n,

3. d excede un valor preasignado $B < n^{1/2}$, en donde podemos decir que cualquier factor primo p de n satisface $p > B$.

Analogamente para este método vamos a verificar en cada uno de los posibles puntos de n, puntos determinados por $\sqrt[x]{n}$, explorando en $\sqrt[max]{n} \ldots \sqrt[2]{n}$, donde max es el valor máximo de x y está determinado por el campo de búsqueda. Después de localizar esos puntos P, en cada uno de ellos, realizaremos otra prueba por Trial division, donde buscamos los posibles valores de d, que son los divisores de n.

El *Campo de búsqueda* es un valor que asignamos para que se realice una exploración limitada, el campo de búsqueda está definido por un *límite inferior* y un *límite superior*, donde $limite\ inferior_x$ es igual a los valores en los que se lleva a cabo la exploración menores que $\sqrt[x]{n}$ y el $limite\ superior_x$ son los valores mayores de $\sqrt[x]{n}$.

Para reducir desde un principio la búsqueda de los factores primos, sólo trabajamos con números impares, dejando de lado los números pares.

Despúes de haber determinado el campo de búsqueda, calculamo el máximo valor de x, este valor lo encontramos a través de la búsqueda de una raíz x de n que se acerca al valor asignado al campo de búsqueda y, es decir, si tenemos un valor $y = 1,000,000,000$, entonces buscaremos un punto $P = \sqrt[x]{n}$ que sea cercano a este valor, con esto aseguramos que si existen factores pequeños, no importando el número de dígitos de n, siempre los encontrará, ya que si colocaramos un valor de x constante, cada que el número n aumente de tamaño, la búsqueda se alejaría cada vez más de esos posibles factores primos pequeños de n.

El algoritmo mencionado lo representamos como el algoritmo 17.

El comportamiento del algoritmo 17 lo podemos observar en la figura 5. Como vemos, en la figura 5, en el punto que le corresponde a $P = \sqrt[3]{n}$ hay una búsqueda de algún posible factor primo hacia la derecha(P_{pos}) y hacia la izquierda(P_{neg}), en la misma figura, cuando tenemos un valor n relativamente pequeño, el algoritmo 17 siempre encontrara los factores, ya que su campo de búsqueda estara actuando sobre practicamente todos los posibles fatores de n, pero, conforme n aumenta de tamaño, los limites de búsqueda se van reduciendo y por lo consiguiente su rango de éxito comienza a ser limitado.

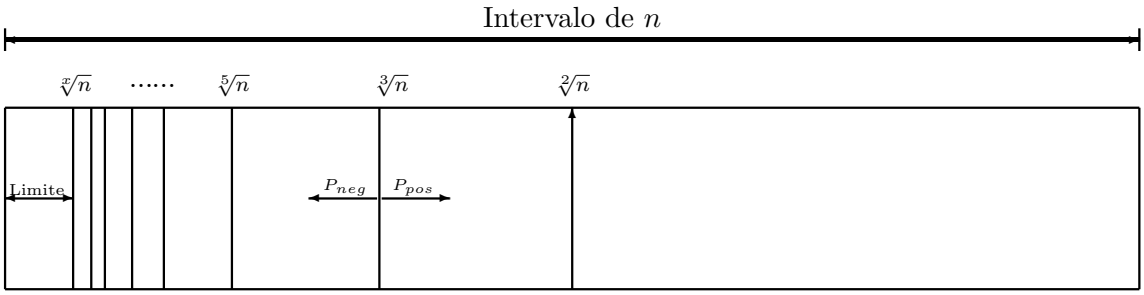

Figura 5. Puntos en los que actua el algoritmo 17.

Sabemos que el máximo posible factor primo de un número se encuentra en su raíz cuadrada, por lo tanto nuestro máximo punto de búsqueda, es precisamente $\sqrt{n}$, por lo tanto al llegar a

Algoritmo 17 Algoritmo para encontrar factores primos cercanos al punto $P = \sqrt[x]{n}$

1: **Entrada:** Valor de n.
2: **Entrada:** Valor de limite de búsqueda y.
3: Calcuar el valor máximo de x.
4: **while** $x \geq 2$ **do**
5: Calcular $\sqrt[x]{n}$
6: **for** $i = 1$ hasta $i \leq x$ **do**
7: $P_{neg} = P_{neg} - 2$
8: $factor = GCD(P_{neg}, n)$
9: **if** $factor > 1$ **then**
10: **Imprime** $factor$
11: **end if**
12: $P_{pos} = P_{pos} + 2$
13: $factor = GCD(P_{pos}, n)$
14: **if** $factor > 1$ **then**
15: **Imprime** $factor$
16: **end if**
17: **end for**
18: $x - -$
19: **end while**

la raíz cuadrada evitamos perdida de tiempo y recursos, creando solamente la búsqueda en la parte que le corresponde a P_{neg}, ya que en P_{pos} ya no hay posibilidad de encontrar un factor primo.

Este algoritmo, mantiene una buena respuesta en números que tienen sus factores primos cercanos a los valores de las raíces que usamos, este algoritmo es capaz de factorizar números de más de 1,000 digitos, pero sólo si cumplen con la condición de cercanía a sus raíces. En la tabla 5.1 tenemos dos ejmplos, en el primero de ellos tenemos un número con 567 digitos, al aplicarle el algortimo 17 encontramos sus factores primos (separados por una sucesión de 3 *'s) donde vemos que su factor primo más grande encontrado tiene 173 digitos, valor que es muy díficil encontrar para otros algoritmos de factorización hasta el momento por mi conocidos. Pero el segundo ejemplo mostrado en la tabla 5.1 observamos que es un número de 1416 dígitos decimales, y de igual forma al aplicarle el algoritmo 17 obtenemos sus factores primos, donde podemos ovservar que encontramos un factor primo de 354 digitos decimales, un factor que de otra forma podría no ser encontrado.

Como podemos observa en el cuadro 5.1, los tiempos de factorización son largos y estos tiempos dependen principalmente del campo de búsqueda que utilicemos, en este caso el campo de búsqueda fue de $1,000,000$.

El algoritmo 17, trabaja de una forma diferente al algoritmo de Trial division, ya que el algoritmo sencillo de Trial division busca factores pequeños de un número n, y sólo está enfocado en el tamaño del factor en sí, por el contrario el algoritmo 17, no realiza la búsqueda basándose en el tamaño de los factores primos de n, sino más bien, enfoca su trabajo en encontrar, como ya mencionamos, un factor primo cercano a puntos específicos de n, lo que nos permite encontrar

Cuadro 5.1: Ejemplos de números factorizados con el algoritmo 17.

Valor de n	Factores primos	Tiempo
3294213805443470598703844985451630375838- 5090965429020226857728191742869966919108- 5847660080568829553144518359249395001168- 3229405002943832026265523865404291204242- 3395691471314284821129537607815066603147- 5400295107133804966777939138105802613494- 8797927814805047927021106597561900645296- 0956170183228747579035882358578702790382- 8777076649277783871957380177609201578559- 1800227783077197951591865111230718869214- 1054915513223144291077518499011732400073- 5532445995614311024516253732686091424896- 8870162636875871191848747359272378316908- 0890660757449025603679815085553656079315- 0768867	710947691***710954971***5054517950374146- 71***2554815171065483349775545703589825 5- 1***6527080558306354951913215285768323 20- 0474795318985367128675682531543183 9***42- 6027806146207982650533981465754213582407- 1925218575454408597173635657837039886468- 8407816289318364628807475507479089834002- 2975216346632592500 9***18149969160975096- 8157962887895338709223797785250860893574- 7443340034442832151706751302237412290542- 6546311547588884407932843549867139808832- 1902307657942765506397022987713562778271- 7284095415016736144765861694796790595770- 5828587442889290633578438196152172583403- 619277749260537298554573 57	2:40 h
6882944716695768325170551686867003360503- 8605143955634231061309746315280234185966- 0553785281328713069885605176266893675085- 8683485266856863366647434229852649382434- 8577451806085987489833829351589076343246- 1911302477359359908669752840570086114945- 3013329207166223911859321551549189383780- 6866422067081162009584874900123543388568- 3204329439911841186498925316944004464124- 0699726781171504113013006106393498318227- 5047029590816070118373175514221225991415- 8951974308391229077363873109888883482822- 9037840164028451041413806070695718052964- 2038574033520906622664118755972812686764- 8912144504583828583112348508075184461088- 3958219438829306269577390282074591557966- 7274186997327392240357466042058308396840- 8139397553932442881504262796139499880678- 1014062273549040660629320721463319412498- 5423494641500642527086753392112572761878- 9244706053419880147499165606764403018557- 4391283184178108177007099369618352163980- 0341932860955498668579558944362924186400- 7675984322710276813833516687841846169287- 7072100419359095661545285746993350112792- 8467451438146015159100615363288746319348- 9009865339360226136900873821926304681606- 9660155629122989278837936862646468289847- 7698837547045525521486390182671626660629- 1970261549319264295299299321423035256062- 5520576069617008407712491519829964629924- 5666338082492933401737988143625630969179- 3802785839811511147568167933317525922352- 8720605734228039558337321757673539153505- 1302282303224086645638198791410516730978- 0732004297134823	1757642099179598955758791698423320586525- 99907***1757642099179598955758791698423 3- 2058652600111***3089305748808467172188 43- 7341950728790277711724059838710463851277- 8552223689463815746789773***954381000962- 1044068875709607758824363115756659521160- 5251496138501958850799037888271267093600- 9483098428352114914941486267812816988119- 0533285435325898442487405611602966433655- 73229***9108430949974283328251786119849 7- 9611686997342733478652950794502349007445- 5989574783164831822331828564586918517170- 8438652856742186755835349095593298568551- 4276698827062610248082764172279747634757- 8082600369650447656220759200110157144929- 5426973846676754957228699326673941278466- 3136701017203373709030615426866334291564- 9045360256724628944843366396754693942237- 99***8296351437044942544223581752084638 9- 9861233026405676696216056158810229884621- 7150091201409819059511461642965690511486- 7265898995015920541181949219638329556852- 2419351982538452710102806529960892327211- 5342632610644738867289778571261394591043- 9791689445823852258596496371979859248 03- 8794921796180282375593658429657763803491- 1726280161779117075787875430524671996783- 5105007498521956779640521397644014811789- 8940796185599604054736082150691211300184- 4395492541992968533770480599520872856538- 9693058438694717258660205611218162088323- 5433898018286349100824875907087460797260- 2166477563720946929040307496734410464310- 6016993021216241225341262400811112386694- 6039641608356923121108547637540620098955- 9779369120878506820997413730902 53	11:00 h

factores primos grandes y dejar de lado la imposibilidad de encontrar dichos factores a través del algoritmo de Trial division simple.

Estas características nos abren una amplia gama de posibilidades para la exploración para encontrar primos grandes, pero limitada también por el campo de búsqueda que usaremos para

encontrar los factores primos. La opción de la paralelización del algoritmo 17, solucionará, o ayudará a retrasar, algunos problemas de tiempo y búsqueda.

5.1. Algoritmo paralelo

Uno de los inconvenientes que presenta esta forma de búsqueda, es que, si por ejemplo, tenemos un valor de $x = 10$ y el número n sólo tiene un factor primo que se encuentra cercano a $\sqrt[2]{n}$, primero tendrá que realizar la búsqueda por todos los puntos P desde 10 hasta 2, es decir todo ese tiempo consumido en la búsqueda de un factor primo en cada uno de los puntos P de n hasta llegar a $\sqrt[2]{n}$, ha sido prácticamente desperdiciado, ya que no encontraremos ni un solo factor primo y tendra que pasar un largo tiempo de ejecución hasta llegar a nuestro punto P deseado, y por consiguiente al encuentro de un factor p.

Como todo el tiempo que consume, se usa en la búsqueda de cada punto P, basándonos en la facilidad de paralelizar el algoritmo de Trial division, podemos realizar cada una de estas búsquedas por separado, modificando el algoritmo 17 para una ejecución de forma paralela, el algoritmo paralelo lo mostramos como el algoritmo 18.

5.2. Metodología

En al algoritmo 18, cada proceso se encarga de escanear uno o varios puntos P del número n, sí alguno de los procesos encuentra un $factor$, este factor es guardado, ya sea en un buffer en memoria RAM o en un archivo en memoria física, cuando el proceso 0 encuentra un factor almacenado, el proceso 0 es el encargado de decirnos que se ha encontrado un factor, calcular el nuevo valor de n y enviar este valor de n al resto de los procesos, cuando los procesos restantes reciben el nuevo valor de n comienza la búsqueda en este nuevo número n.

Cuando cada proceso recibe un valor de n se le realiza una prueba de primalidad, si n es primo, entonces se rompen los procesos y el algoritmo termina su búsqueda.

Dos cosas que se tienen que tener presentes es que, hay que tener cuidado de que $limite\_inferior$ de un proceso no sea 1, ya que si esta situación se da, puede devolver el valor de n como si se tratara de un factor. Otra situación a considerar, es que el valor de $raiz$ nunca puede ser un número par, púes esto provocaría que realizaramos cálculos sobre solamente valores pares, y nosotros buscamos factores impares.

Como ya mencionamos antes, para llevar a cabo los calculos con números grandes nos auxiliamos de la herramienta de GNU llamada GMP, una biblioteca que nos permite implementar todas las operaciones básicas con números enormes, capaz de calcular $!1,000,000$ sin mayor complicación con una memoria RAM de 2Gb. Esta biblioteca utiliza memoria dinámica para no depender de un espacio fijo de memoria, y así poder trabajar con números tan grandes como nuestra memoria nos lo permita.

De acuerdo con *Buchberger* y *Jebelean*[67], en computaciones algebraicas típicas, más de la mitad del tiempo empleado es usado para el calculo del GCD de enteros largos[68], por lo

Algoritmo 18 Algoritmo para encontrar factores primos cercanos al punto $P = \sqrt[x]{n}$ de forma paralela

Entradas: Entero n para factorizar y $referencia$.
Inicia MPI.
Calcula valor máximo de x.
Busca factores pequeños de n, $p \in \{2, 3, \ldots, 101\}$.
while $(primo \,!= 0)$ **do**
  if ip=0 **then**
    Envia x y n a todos los procesos.
    while $((factor < 1)$ && $(factor != n))$ **do**
      Revisar en buffer si se ha encontrado un factor.
    end while
    if $((factor > 1)$ && $(factor != 0))$ **then**
      Imprime $factor$
      $n = n/factor$
      Verifica si n es primo.
      if (n es primo) **then**
        Rompe procesos.
      else
        Calcula x nuevamente.
        Envia valores de x y n.
      end if
    end if
  end if
  if $(ip != 0)$ **then**
    Recibe x y n.
    Verifica si n es primo.
    if (n es número primo) **then**
      Rompe procesos.
    end if
    $raices = x/(total\_procesadores - 1)$
    $limite\_inferior = (ip \cdot raices) - raices + ip$
    $limite\_superior = limite\_inferior + raices$
    for $(i = limite\_superior$ **hasta** $i = limite\_inferior)$ **do**
      $raiz = \sqrt[i]{n}$
      Busca los factores.
      Revisamos si el valor de n no se ha modificado.
      if (n se ha modificado) **then**
        Rompemos el calculo actual y usamos el nuevo valor de n.
      end if
      if $((factor != 1)$&& $(factor != n))$ **then**
        Guarda el valor de factor.
      end if
      if $(i = limite\_inferior)$ **then**
        Guarda el valor de $factor = 1$.
      end if
    end for
  end if
end while
Finaliza MPI

que tratamos de reducir el uso del GCD, con este fin filtramos todos los valores $mod\ 3 = 0$, $mod\ 5 = 0$, $mod\ 7 = 0$, $mod\ 11 = 0$, lo que nos reduce los tiempo en forma considerable, por ejemplo tenemos el valor de

$$n = 426029877121680646529883567909734170164614671781222585542028051125501150 4-$$

$$4717576044760680272070914068725184640911010269087852440378782092151 91$$

y calculamos $\sqrt{n}$. Del valor de la raíz de n realizamos una cuenta regresiva de $1,000,000$ valores d y calculamos (d, n) para cada d, es decir, $1,000,000$ calculos del GCD, lo que nos regresa un último valor de

$$gcd = 652709642277238498686849782093149898507694866194089066539916381858894 13$$

Estos cálculos se realizan en un tiempo de ejecución de $49953ms$ sin los fltros y $24235ms$ con filtro. Otro ejemplo, es un valor n de 178 digitos decimales, bajo las mismas circunstancias que el anterior, que nos devuelve un $gcd = 271211$ y tarda un tiempo de ejecución de $541437ms$ sin filtros y $199421ms$ con filtros.

Por el teorema de Mertens, sabemos que con estos filtros eliminamos un alto porcentaje de valores que no serán analizados con el GCD, tenemos que $\frac{1}{p}$ es la proporción de números que son divisibles por p, ahora tenemos que, $1-\frac{1}{p}$ es la proporción de números que no son divisibles por p.

Si ser dividido por p es un evento independiente de ser dividido por q, entonces

$$\left(1-\frac{1}{p}\right)\left(1-\frac{1}{q}\right),$$

sería la proporción de números en un conjunto que no son divisibles por p ni por q, lo que queda determinado como la ecuación 5.2

$$\left(1-\frac{1}{2}\right)\left(1-\frac{1}{3}\right)\left(1-\frac{1}{5}\right)\ldots = \prod\left(1-\frac{1}{p}\right) \tag{5.2}$$

En el cuadro 5.2 podemos observar las proporción de números naturales $\leq n$ no divisibles por $2, 3, 5 \ldots p$.

Para realizar las pruebas de primalidad utilizamos el algoritmo de Miller-Rabin(Algoritmo 6), ya que es el algoritmo más rápido de los presentados, además de que tiene un alto índice de éxito al determinar si un número es primo o compuesto. El algoritmo de Fermat(Ver página 27) también presenta una buena respuesta para determinar si un número es o no primo, pero existe la situación de poder toparse con un número de Carmichael, lo que nos devuelve un resultado incorrecto.

Como vemos en la prueba de Miller-Rabin utilizamos la exponenciación modular $a^n\ (mod\ m)$, con un valor de n grande, expresión que seria bastante complicado calcular para una computadora, en especial si hablamos de valores n con más de 100 digitos decimales. Para llevar a cabo la exponenciación modular $a^n\ mod\ m$, utilizamos el algoritmo 19, que esta basado en la descomposición binaria del número n,

Cuadro 5.2: Proporción de números no divisibles por $\left(1-\frac{1}{2}\right)\left(1-\frac{1}{3}\right)\cdots\left(1-\frac{1}{p}\right)$.

p	$\prod\left(1-\frac{1}{p}\right)$	p	$\prod\left(1-\frac{1}{p}\right)$	p	$\prod\left(1-\frac{1}{p}\right)$
2	0.5	73	0.126047	179	0.106643
3	0.333333	79	0.124451	181	0.106054
5	0.266667	83	0.122952	191	0.105499
7	0.228571	89	0.121571	193	0.104952
11	0.207792	97	0.120317	197	0.104419
13	0.191808	101	0.119126	199	0.103895
17	0.180525	103	0.117969	211	0.103402
19	0.171024	107	0.116867	223	0.102939
23	0.163588	109	0.115795	227	0.102485
29	0.157947	113	0.11477	229	0.102038
31	0.152852	127	0.113866	233	0.1016
37	0.148721	131	0.112997	239	0.101175
41	0.145094	137	0.112172	241	0.100755
43	0.141719	139	0.111365	251	0.100353
47	0.138704	149	0.110618	257	0.0999628
53	0.136087	151	0.109885	263	0.0995827
59	0.13378	157	0.109185	269	0.0992125
61	0.131587	163	0.108516	271	0.0988464
67	0.129623	167	0.107866	277	0.0984896
71	0.127798	173	0.107242	281	0.0981391

$$n = n_0 \cdot 2^0 + n_1 \cdot 2^1 + n_2 \cdot 2^2 + \ldots + n_{r-1} \cdot 2^{r-1}.$$

donde n_i son los valores binarios 0 y 1, y r es la posición binaria que ocupa n_i.

Este algoritmo no es el único existente para este fin, en [69] y en [70] nos muestran algunos de los algoritmos basicos en este ramo. El mejor método para exponenciación modular depende en gran medida del grupo que lo requiere, el hardware y el sistema en donde es implementado[69].

Como ejemplo tenemos 133^{137} *mod* 101, convertimos el valor de 137 a su representación binaria 10001001, lo que en forma de potencia tendriamos $2^7 + 2^3 + 2^0 = 128 + 8 + 1$, por lo tanto $133^{128+8+1}$, por propiedades de los exponentes tenemos $133^{128} \cdot 133^8 \cdot 133^1$ *mod* 101 (Como vemos 133^{128} sigue siendo un valor grande por lo que realizamos el paso de la conversión a sistema binario nuevamente, y así sucesivamente hasta que queden sólo valores pequeños, el resultado de 133^{128} (*mod* 101) = 36), ahora realizamos el calculo de 133^1 *mod* 101 = 32, este primer resultado lo multiplicamos por 133^8 *mod* 101 = 36, el resultado de $36 \cdot 32$ *mod* 101 = 41 y así sucesivamente hasta llegar a un resultado final, que en este caso es 133^{137} *mod* 101 = 62.

Algoritmo 19 Exponenciación modular rápida

1: **Entrada:** a, n y m todos ≥ 1
2: $u \leftarrow n$;
3: $s \leftarrow a \; mod \; m$;
4: $c \leftarrow 1$;
5: **while** $(u \geq 1)$ **do**
6: **if** (u es impar) **then**
7: $c \leftarrow (c \cdot s) \; mod \; m$;
8: **end if**
9: $s \leftarrow s \cdot s \; mod \; m$;
10: $u \leftarrow u/2$;
11: **end while**
12: **Regresa** c;

5.2.1. Desarrollo del algoritmo 18 con MPI

Para la implementación del algoritmo en su forma paralela, utilizamos la biblioteca MPI(*Message Passing Interface*)[2]. MPI es una biblioteca estandar de paso de mensajes basado en los acuerdos del *MPI forum*, con participación de 40 organizaciones,reunidos desde Noviembre de 1992 hasta Abril de 1994 para discutir y definir un conjunto de estándares para la biblioteca de interface para paso de mensajes, la meta de MPI es el establecer un eficiente, portable y flexible estandar para paso de mensajes que será ampliamente utilizado para la escritura de progamas paralelos.

MPI puede ser usado bajo Fortran, C estandar y C++. En C, lo primero que necesitamos es declarar la libreria

```
#include "mpi.h",
```

y con esto podemos comenzar a utilizar las instrucciones de MPI. Existen alrededor de 129 funciones en MPI, pero nosotros sólo usamos 6 instrucciones básicas para nuestro programa en MPI. La primera instrucción es

```
int MPI_Init(int *argc, char ***argv),
```

que nos inicia un programa que se ejecutara de forma paralela, contraria a ésta tenemos

```
MPI_Finalize(),
```

la cual no indica que la ejecución paralela ha concluido, es muy importante el saber que todas las acciones pendientes han concluido antes de llamar a MPI_Finalize, despues de la escritura de este comando no se puede usar ninguna otra instrucción perteneciente a MPI, ni siquiera MPI_Init.

En la programación paralela es importante saber cuántos proceadores hay, quien soy yo y dónde estoy corriendo. El número de procesos con los que contamos nos los entrega

[2]Se pueden consultar las páginas electronicas, http://www.mcs.anl.gov/research/projects/mpi/ y http://www.mpi-forum.org/

`MPI_Comm_size(comm, n_proc),`

donde *comm*, es el comunicador con el que estamos trabajando, MPI_COMM_WORLD utilizado generalmente, que identifica todos los procesos involucrados en un cómputo, y nos devuelve el valor de *n_proc* que es el número de procesadores existentes(los procesos se enumeran desde 0). La función que nos dice que proceso somos esta dada

`MPI_Comm_rank(comm, id),`

que nos devuelva el proceso que somos a través de la variable *id*, y si queremos saber el nombre del proceso en el que estamos trabajando usamnos

`MPI_Get_processor_name(name, long_name),`

que nos regresa el nombre del proceso en la variable *name* y la cantidad de caracteres de name.

Para la comunicación entre los procesos MPI tiene dos comandos básicos, el primero de ellos

`MPI_Send(buf, count, datatype, dest, tag, comm),`

que es el encargado del envío de mensajes, donde *buf* es la dirección del *buffer* de envio, *count* es el número de elementos que se envian, *datatype* corresponde altipo de datos que se envía, *dest* nos indica a quien va dirigido el mensaje y *tag* es la etiqueta identificadora del mensaje. En sentido inverso tenemos

`MPI_Recv(buf, count, datatype, source, tag, comm, status),`

este comando recibe los mensajes que le han sido enviados, sus parámetros son los mismos que MPI_Recv salvo *source* que es el que indica quién envio el mensaje y *status* que una estructura con campos MPI_SOURCE y MPI_TAG que nos permite obtener información sobre el mensaje recibido.

Al ejecutar un programa en MPI, el programa se copia en cada uno de los procesos disponibles, la tarea que realiza cada proceso está determinada por el programador a través de la dirección de cada proceso (id). El siguiente bucle nos da una idea de cómo se lleva a cabo el control de los procesos, y sus tareas a realizar,

```
if(id==0)
{
     Tarea 1 a realizar.
}

if(id!=0)
{
     Tarea  n a realizar.
}
```

lo que nos indica que si es el proceso 0 realice una tarea, si es otro proceso realice otra tarea y así con cada tarea asignada a un proceso distinto.

Existen muchas versiones de MPI, entre las de dominio público tenemos LAM, CHIMP, UNIFY, W32MPI, MPICH (MPI/Chameleon), MPICH/NT, WinMPI,MPI-FM[71].

En nuestro programa en forma paralela utilizamos el proceso 0 como *maestro* y el resto de los procesos como *esclavos*, cada uno de los esclavos se encarga de buscar posibles factores primos en los puntos P de n, y el proceso maestro se encarga de la monitorización de cada uno de los demás procesos para verificar si se ha encontrado un factor p de n.

Ahora necesitamos saber cuantos y cuales puntos de búsqueda ejecutará cada proceso. Para determinar los puntos de búsqueda de cada proceso primero calculamos cuantos puntos le corresponden a cada proceso por medio de $raices = x/(total\_procesadores - 1)$, donde x es el número máximo de puntos P y *total_proocesadores* es el número de procesos con el que contamos. Después calculamos los límites de cada proceso a través de las ecuaciones 5.3 y 5.4,

$$limite\_inferior = (ip)(raices) - raices + ip \tag{5.3}$$

y

$$limite\_superior = limite\_inferior + raices \tag{5.4}$$

donde *limite_inferior y limite_superior* son los valores mínimo y máximo de x para cada proceso. Estos valores limite no son estáticos, ya que cada vez que se encuentra un factor estos son modificados, e *ip* es el valor identificador son el que cada proceso cuenta.

Antes de ejecutar el programa en forma paralela, se tuvieron que realizar pruebas en computadoras no paralelas, ¿Como poder ejecutar aplicaciones paralelas en sistemas no paralelos?, para esto utilizamos la versión MPICH2, en [72] se describe el comportamiento de MPICH, la cual nos permitia realizar pruebas de nuestro programa antes de su ejecución en KanBalam. Para llevar a cabo esto, primero se abre ms-dos(en Windows) y se teclea

```
mpirun -n número_de_procesos nombre_de_ejecutable.exe
```

y la aplicación se comienza a ejecutar como si estuviera en un sistema paralelo, obviamente esto sólo sirve para monitorizar fallas e incoherencias del programa ya que no puede ser útil para la medición de tiempos y eficiencia del programa en su forma paralela.

El algoritmo 18, presenta unas grandes mejorías, con respecto al algoritmo 17, en cuestiones de tiempo y rango de búsqueda en las pruebas realizadas, ya que ahora cada proceso se puede encargar de la búsqueda de un factor en cada punto P (o por lo menos se reduce el números de P's a escanear por cada proceso existente), sin necesidad de esperar a que se realicen todas las búsquedas de todos los puntos P antes de llegar a donde se encuentra un factor primo.

5.3. Resultados

A través de la versión paralela del algoritmo 17, encontramos los factores primos del segundo valor de n mostrado en el cuadro 5.1 en un tiempo de ejecución de 561660 milisegundos trabajando con 4 procesos y 547490 milisegundos trabajando con 16 procesos.

Como podemos ver el algoritmo puede encontrar factores primos grandes que serían practicamente imposibles de encontrar de otra forma, pero tienen que encontrarse cerca de sus raíces, como vemos en las pruebas realizadas, lo que es una limitante muy marcada para este algoritmo , y más tomando en cuenta que su tiempo de ejecución es de forma exponencial con respecto al rango de búsqueda.

Lo ideal para el algoritmo 18 sería tener un proceso para cada valor de x, para así trabajar sobre cada campo de búsqueda de forma simultanea y no perder tiempo en espera de que termine una búsqueda para iniciar otra.

El algoritmo 18 se enfrenta a una perdida de recursos computacionales si cuenta con mas procesos que raíces a evaluar, ya que esos procesos de más se desperdician al no tener datos a evaluar.

Cuadro 5.3: Ejemplos de números factorizados con el algoritmo 18, donde *** es la separación entre cada uno de los factores primos de n.

Valor de n	No. de procesadores	Factores primos	Referencia	Tiempo (ms)
389967515776915757051504055881242861781235309391435994931284492631462108794113914500723391889353902477212875567850166433521964501453391887790639934666672903527427568058176822358500135150576753024328051166121027457956441763966080029216053871566861670353840106411791459756961313157505109464520442816331566631345901123366583318584450851079019873352058216058338463330838558740682469517133314124686035483940682336511246720664198735304114592680517888507881283602679214403000379706191628591353732580033678156021664775984819324135982270532508761016262390256221294288309065582920749663505053235569313268133508395972949998375583500783889840205586719858855419631289461569588891908702529027723030958807102348793467962616995983140813017771719937853237233093160368544303458765754647394600195588570551517323509328043185652427054551835710	4	311042909***404221093***1257301046303611633***158080592123126923471701760313052833***2498947360599841792225085842958931408058657924036213442680326310372873***624473791104891572655690376177099551601782752195757262206228597521824157208983210285164741979174283793128385518431672731454511509850215756316	$1 * 10^8$	8156300
3157505109464520442816331566631345901123366583318584450851079019873352058216058338463330838558740682469517133314124686035483940682336511246720664198735304114592680517888507881283602679214403000379706191628591353732580033678156021664775984819324135982270532508761016262390256221294288309065582920749663505053235569313268133508395972949998375583500783889840205586719858855419631289461569588891908702529027723030958807102348793467962616995983140813017771719937853237233093160368544303458765754647394600195588570551517323509328043185652427054551835710571	16	315518417***379384679***1197028528822169713***1432877304439198620519993366355886977530979446488442343171655614616817516781494578357***421537305835333226884185113962252862397332165946367835364360360808958180807498995562943626906198565869637710930541712528710233483116596***17769370021091126017320748005827737072333044814632656933609364816707481878042407286373744520336162552279519645953893643945973444228502907588660681781288544155605633475856545118811969209304465919538346214751771113765099076647883018049141585365976178315429754375171560069922391	$1 * 10^9$	100038000
7187998502245805668108849633544566368253046242674623539879225300550173736092916142690239141952149850411580511495121603100945109720835824058577618164658337794071363773666184041144492180122513058969742857312025393787889692301577441404727143292631015699649534919578276332604855427090374259957182226359502449557501800395246246512091398730885389283865922680565091163240119639019679044675441069214192310002588482071144842918305376485257680202559552878217629813585604795104508907197809336476424493588574072486221078033323183082579200128282074050641381393094685095122179857718842725154438380585571	32	101***15518411***1497443261***2347031937268594391***5508558914658780527562587011015676033***3034422131626672209138301452882038536110680205072562812360359152781437556***9207717672905757201976562794455701967728301663036101307944655195782425331265791121047628488323956611854253919675983341354985165360733878270783008935***847820647439410127751821372535873532274016585916436856697588403468279044081841933939237411260944602555010537819820527104802428824086157605613114689353132424454082709606425563360863919124718034016598909433047087114575189154292559946894183112509651347802046715453482924897403204131193194489301071	$1 * 10^8$	6709780
7798607859550486432387167727966294763227862744864603643263979769369674768269189726355380423210428019387235132110386562389001919115762569782391925048400373091843207314774909581918289668819465767492055244339930565246487339679065536260917404086442086396900031732526009286385999600375995078543938520455575775835557285885082188995380277608407504735010388661751985496827390615930864178477608516121559904562153309424605960154904931531410310844712811770574460684491738335213992087445402264282047138907932244844261764862718559632368779479632082221706001	32	123133304214673007***1231333043078-51221***151618106182965506797766593051732213***229880501225090032414913684384449607629313367600942074716584275546773***528450448434986198939587179686531979849806489170213066937706305612237126907354875957872294841855937346288741399781284965213926236615026777***27925987645113800759609693334871876348151324750246369242918328932789218149281848887562859607815656382939811791491459274685424237018170181947670654864919403107814244525542718211115207417471869965181919952403247549155509511713590678310427204039101210505250967333214089647456631	$1 * 10^8$	4120040

Capítulo 6

Ataques a RSA

6.1. Criptografía de clave pública

La fecha oficial del nacimiento de la criptografía de clave pública es en 1976, cuando *Diffie* y *Hellman*, de la *Universidad de Stanford*, publicaron el articulo *New Directions In Cryptography*[4]. Es en este trabajo que ellos proponen una nueva idea completa de la criptografía de clave pública, pero en este momento ellos no tenian una implementación práctica de esta idea.

Despúes de la Publicación de Deffie y Hellman, *Rivest, Shamir y Adleman* del *Instituto Tecnológico de Massachusetts*, propusieron un criptosistema de clave pública práctico y factible en 1977[5], ahora conocido como *Criptosistema RSA*.

El criptosistema RSA, se basa en el problema matemático de la factorización de números grandes, esto es, sí $f : pq = n$, donde p y q son dos números primos grandes, la función f es fácil de calcular en tiempo polinomial, sin embargo, el calculo de f^{-1} es difícil de calcular.

Ahora la descripción del sistema RSA es la siguiente: Ya que hemos calculado el valor de $n = pq$, calculamos el valor

$$\phi(n) = (p-1)(q-1),$$

despues de este calculo escogemos un número aleatorio b que cumpla con la condición $1 < b < \phi(n)$, tal que $gcd(b, \phi(n)) = 1$: Con el valor elegido b, calculamos el valor de

$$a = b^{-1} \; mod \; \phi(n),$$

ya que se han calculado estos valores, publicamos n y b y los valores de p, q y a quedan como privados. La función de encriptación está dada por la ecuación 6.1

$$e_K(x) = x^b \; mod \; n \tag{6.1}$$

y la función de desencriptación queda definida como la ecuación 6.2

$$d_k(y) = y^a \; mod \; n. \tag{6.2}$$

Despues de la publicación de RSA los criptógrafos ingleses *Ellis, Cocks y Williamson* del *Grupo de Seguridad de Comunicaciones Electrónicas del Gobierno del Reino Unido / Dirección General de Comunicaciones Gubernamentales*(CESG/GCHQ, por sus siglas en ingles) afirmaron que ellos habían descubierto secretamente la encriptación de clave pública años antes que los científicos estadounidenses. Lo que nos muestra las dos tendencias de las investigaciones en cuestiones de criptografía: La criptografía pública, particularmente realizada por personas trabajando en instituciones académicas, y la criptografía secreta, particularmente realizada por personas trabajando para gobiernos y fuerzas militares.

El criptosistema RSA es más comúnmente utilizado para proveer privacidad y asegurar la autentificación de datos digitales. En estos tiempos RSA se ha destacado en muchos sistemas comerciales. Es usado por servidores web y buscadores para el tráfico seguro de la web, se utiliza para asegurar la privacidad y autenticación del correo electrónico, también utilizado para la seguridad de sesiones remotas y es el corazón de los sistemas de pago de las tarjetas de crédito electrónicas[73].

Desde su publicación, el sistema RSA ha sido analizado para encontrar vulnerabilidades por muchos investigadores, esto a provocado un gran números de ataques fascinantes, pero ninguno de ellos ha sido devastador[73], estos ataques se han presentado en las más diversas formas y tratando de entrar en vulnerabilidades de RSA[1], pero en este trabajo solo nos concentramos en los ataques referentes a la factorización de números enteros.

Algunos ataques de los algoritmos de factorización vistos hasta ahora sobre números similares a los RSA, son presentados en el cuadro 6.1.

Como vemos en el cuadro 6.1, las claves RSA atacadas con los algoritmos que hemos presentado hasta el momento son, de alguna forma, seguras, ya que, los algoritmos de propósito general, tal como el de Pollard-Rho o el algoritmo ECM, presentados en dicho cuadro después de alcanzar tamaños relativamente grandes comienzan a menguar en su capacidad de localizar los factores primos del valor atacado, ni mencionar el algoritmo de Fermat que muy pronto es rebasado por el tamaño de las claves, lo que nos permite simplemente incrementar los tamaños de los números primos p y q que conformaran nuestra clave, claro, tomando en cuenta la cercanía de sus valores y no colocar en la clave un valor primo de más tamaño que el otro. Por el contrario vemos que los algoritmos de propósito especifico, pueden atacar estas claves de RSA dependiendo de las características que presenten los números primos p y q, ya que como vemos si los factores de $p \pm 1$ son pequeños, el algoritmo $p-1$ y el algoritmo $p+1$ serán capaces de factorizarlo, por lo que debemos de cuidar al momento de generar claves RSA que nuestros valores p y q no presente estas características, aunado a esto también podemos evitar la factorización de $n = pq$ escogiendo dos valores p y q similares en tamaño que presenten una descomposición de $p-1$ con un mismo valor de suavidad, ya que esto, no permite al algoritmo $p-1$ factorizar un entero, tal es el caso de $n = 204947383$, que aparentemente es un valor "pequeño"[2] y aparentemente fácil

[1]Para conocer un poco más de esas vulnerabilidades y ataques hacia RSA, podemos revisar [74],[75], [76], [76].

[2]La interpretación de "pequeño" o "grande" ha cambiado considerablemente conforme la tecnología ha ido evolucionando. En 1960 John Brillhart y John Selfridge predijeron que la factorización de números de alrededor de 25 dígitos decimales podría ser difícil, en los 70's, Richard Guy dijo que pocos números con aproximadamente 80 dígitos podrían ser factorizados, pero en 1994 se llegaba a números de alrededor de 100 dígitos decimales[77].

Cuadro 6.1: Ataques hacia números similares a RSA por medio de los algoritmos de factorización de enteros, donde *** significa que el algoritmo no encontró los factores primos en un tiempo de 12 horas, tiempo escogido arbitrariamente.

Número de prueba	Valor n	Factores	Fermat	Pollard-Rho	Pollard-Rho modif.	Pollard $p-1$	$p+1$	ECM
1	914652763	32119*28477	0	0	15	0	0	0
2	1050562649016259087	1015348861*1034681-467	1500	1344	93	1765	938	0
3	1786851843977229075-27918979	13472948060087*1326-2515642517	***	252797	32016	***	20718	100
4	4193266407167671831-4522657556389	5307027867738937*79-01346123803597	***	4871590	1035830	2859	1906	2000
5	1887823778472212021-2996649587379527	137560275190136053*137236115285670859	***	8492390	748125	***	4354080	2700
6	7880425365677006858-4837043646984271491-64281	2610133684290404197-819*301916542172030-3175899	***	***	***	***	***	5100
7	7357034542275591931-3054441719265419022-5075883	1665032891036614953-1471*44185520789894-155033573	***	***	***	***	***	28000
8	1590936805608527518-4256756032504200725-9033254483420323904-7801112939379	2448337954336184362-0661024989756057*64-9802778571015031466-11681195207147	***	***	***	***	***	12002400
9	4044942202244375806-3407778362374746187-3555837911187631463-328525558776381333	1670318476856356725-3629254703928955751*2421659257375401033-8504843828437554083	***	***	***	***	***	***

de factorizar, este valor presenta una factorización de la forma $18787 * 10909$, valores que a su vez presentan una factorización de la forma $p-1 = 2*3*31*101$ y $p-1 = 2^2*3^3*101$, como vemos el máximo factor primo en ambos casos es 101, por lo que ambos valores son *101-smooth*, por lo que el algoritmo $p-1$ no puede factorizar a n. La prueba 9 del cuadro 6.1 se factorizó con el algoritmo ECM, pero el tiempo de ejecución rebaso las 12 horas de referencia del cuadro ya que se realizo en 29 horas de ejecución.

Cuando estos algoritmos comienzan a fallar, es tiempo de recurrir a los algoritmos más potentes de factorización existentes en este momento: la criba cuadrática(Algoritmo 20) y el algoritmo de la criba de campo de número(Algoritmo 22).

6.2. Algoritmos Fuertes contra RSA

La criba cuadrática(QS, por sus siglas en ingles), es un algoritmo (Algoritmo 20) inventado por *Pomerance* en 1981, y fue publicado en 1982[78], el cual se basa en una simple observación, sí tenemos dos enteros x y y tal que

$$x^2 \equiv y^2 \ (mod\ n),\ 0 < x < y < n,\ x \neq y,\ x + y \neq n, \tag{6.3}$$

entonces el $(x \pm y, n)$ es un posible factor de n, por que $n|(x+y)(x-y)$, pero $n \not| (x+y)$ y $n \not| (x-y)$.

Algoritmo 20 Algoritmo de la Criba Cuadrática, QS

1: **Entradas**: Entero n para factorizar.
2: Definimos un factor base como sigue $FB = \{-1, p_1, p_2 2, \ldots, p_k \leq B\}$
Donde los p_i son los primos para los cuales n es un residuo cuadrático modulo p_i Y B es el valor de suavidad del factor base.
3: Encuentra $a_1, a_2, \ldots, a_k$ cercano a $\sqrt{n}$(Esto puede ser realizado por medio de $a_i = \sqrt{n}+1, \sqrt{n}+2, \ldots, (n-1)/2)$ tal que, cada $Q(a_i) = a_i^2 - n$ sea suave.
4: Con el uso de algebra lineal, encuentra un subconjunto U del número $Q(a_i) = a_i^2 - n$ cuyo producto $\prod {p_i}^{\alpha_i}$ sea un cuadrado, dicho $y^2\ mod\ n$. Esto es, $y^2 = \prod a_i^2 - n$. Tenemos que x es el producto a_i utilizado para formar el cuadrado modulo nn, entonces
$x^2 \equiv \left(\prod_{i \in U} a_i\right)^2$
$x^2 \equiv \prod_{i \in U} (a_i^2 - n)$
$x^2 \equiv \prod_{i \in U} Q(a_i)$
$x^2 \equiv \left(\prod_{i \in U} p_j^{\alpha_{j,i}}\right)^2$
$x^2 \equiv y^2\ (mod\ n)$
5: Calcular $(f, g) = gcd(x \pm y, n)$.
6: **if** $f > 1$, $g < n$ **then**
7: Devuelve (f, g), que son los factores de n.
8: **else**
9: Busca nuevos valores de x y y, y si es necesario busca más a_i's.
10: **end if**

Existen muchos trucos para mejorar el QS básico, pero la versión más importante es la Criba cuadrática de Múltiples Polinomios (MPQS)[3]. En el algoritmo 21, presentamos la versión realizada por *Riele et al*[79]. Donde la idea es que para encontrar x y y que cumplan con la ecuación 6.3, tratamos de encontrar (U_i, V_i, W_i), para $i = 1, 2, \ldots$ tal que

$$U_i^2 \equiv V_i^2 W_i \ (mod\ n), \tag{6.4}$$

donde W es fácil de factorizar .

En el algoritmo 21 M es un entero fijo que puede ser definido como: $U(x) = a^2 x + b$, $V = a$ y $W(x) = a^2 x^2 + 2bx + c$, $x \in [-M, M)$, tal que a, b, c satisfacen las relaciones de la ecuación 6.5.

$$a^2 \approx \sqrt{2n/M},\ b^2 - n = a^2 c,\ |b| < (a^2)/2. \tag{6.5}$$

[3]Este algoritmo es capaz de factorizar el valor número 9 de la tabla 6.1, en tan solo 10 minutos.

Algoritmo 21 Algoritmo de la Criba Cuadrática con Múltiples Polinomios, MPQS

1: **Entradas**: Entero n para factorizar.
2: Escogemos B y M, y calculamos el factor base FB.
3: Generamos un polinomio cuadrático $W(x)$. El polinomio $W(x)$ en $(U(x))^2 \equiv V(x)W(x)\ (mod\ n)$ asume valores extremos en $x = 0, \pm M$ tal que $|W(0)| \approx |W(\pm M)| \approx M\sqrt{n/2}$. Sí $M << n$, $W(x) << n$, así $W(c)$ es más fácil de factorizar que n.
4: Resuelve $W(x) \equiv 0\ (mod\ q)$ para todo $q = p^e < B$, para todos los primos $p \in FB$, y almacena las soluciones para cada q.
5: Inicializa la matriz de criba $SI[-M, M)$ a zero.
6: Suma $log\ p$ a todos los elementos $SI(j)$, $j \in [-M, M]$, para el cual $W(j) \equiv 0\ (mod\ q)$ para todo $q = p^e < B$, y para todos los primos $p < PFB$.
7: Seleccionar los $j \in [-M, M)$ para los que $SI(j)$ es cercano a $log(M/2\sqrt{N/2})$.
8: **if** El número de valores de $W(x)$ es $< L + 2$ **then**
9: Construye un nuevo polinomio $W(x)$
10: **end if**
11: Realizar Eliminación Gaussiana a la matriz de exponentes $mod\ 2$ de $W(x)$. Asociado con cada $W(x)$, definimos el vector α_i de la siguiente forma
$\alpha_i^T = (\alpha_{i0}, \alpha_{i1}, \cdots, \alpha_{iL})\ (mod\ 2)$.
Ya que tenemos más vectores α_i(al menos $L + 2$) que componentes $(L + 1)$, existe al menos un subconjunto S del conjunto $\{1, 2, \cdots, L + 2\}$ tal que
$\sum_{i \in S} \alpha_i \equiv 0\ (mod\ 2)$,
así $\prod_{i \in S} W(x) = Z^2$
de lo que tenemos
$[\prod_{i \in S}(a^2 x_i + b)] \equiv Z^2 \prod_{i \in S} a^2\ (mod\ n)$, que es la forma requerida.
12: Calculamos $gcd(x \mp y, n)$ para encontrar los factores primos.

En el algoritmo 21 vemos que el potencial factor primo p de un polinomio cuadrático dado $W(x)$, puede ser definido como: sí $p|W(x)$, entonces

$$a^2W(x) = (a^2x + b)^2 - n \equiv 0 \ (mod\ p).$$

Antes de iniciar con el siguiente algoritmo, hasta la fecha el más rápido conocido, queremos presentar el lema 6.1

Lema 6.1 *Tenemos que $f(x)$ es un polinomio irreducible de grado d sobre los enteros, y m es un entero tal que $f(m) \equiv 0 \ (mod\ n)$. Tenemos que α es una raíz compleja de $f(x)$ y $\mathbb{Z}[\alpha]$ es el conjunto de todos los polinomios en α con coeficientes enteros, entonces, existe un mapeo único $\Phi : \mathbb{Z}[\alpha] \mapsto \mathbb{Z}_n$ que satisface*

1. $\Phi(ab) = \Phi(a)\Phi(b),\ \forall a, b \in \mathbb{Z}[a]$;
2. $\Phi(a + b) = \Phi(a) + \Phi(b),\ \forall a, b \in \mathbb{Z}[a]$;
3. $\Phi(za) = z\Phi(a),\ \forall a \in Z[a],\ z \in \mathbb{Z}$;
4. $\Phi(1) = 1$
5. $\Phi(\alpha) = m \ (mod\ n)$

La idea básica del algoritmo de *Criba de Campo Numérico*[4] (NFS por sus siglas en ingles), visto como algoritmo 22 es, en primer lugar, encontrar un polinomio irreducible $f(x)$ de grado d en $\mathbb{Z}[x]$, y un entero m tal que $f(m) \equiv 0 \ (mod\ n)$. Ahora, tenemos a $\alpha \in C$ que es un número algebraico que es la raíz de $f(x)$, y denota el conjunto de polinomios en α con coeficientes enteros como $\mathbb{Z}[\alpha]$. Después de esto, se define el mapeo $\Phi : \mathbb{Z}[\alpha] \mapsto \mathbb{Z}_n$ por medio de $\Phi(\alpha) = m$ lo que asegura que para cualquier $f(x) \in \mathbb{Z}[x]$, tenemos $\Phi(f(\alpha)) \equiv f(m) \ (mod\ n)$.

Se busca un conjunto finito U de enteros coprimos (a, b) tal que

$$\prod_{(a,b)\in U} (a - b\alpha) = \beta^2, \quad \prod_{(a,b)\in U} (a - bm) = y^2. \tag{6.6}$$

Para $\beta \in \mathbb{Z}[\alpha]$ y $y \in \mathbb{Z}$. Tenemos $x = \Phi(\beta)$. Entonces

$$\begin{array}{rcl} x^2 & \equiv & \Phi(\beta)\Phi(\beta) \\ & \equiv & \Phi(\beta^2) \\ & \equiv & \Phi\left(\prod_{(a,b)\in U}(a - b\alpha)\right) \\ & \equiv & \prod_{(a,b)\in U} \Phi(a - b\alpha) \\ & \equiv & \prod_{(a,b)\in U} \Phi(a - bm) \\ & \equiv & y^2 \ (mod\ n) \end{array} \tag{6.7}$$

Algoritmo 22 Algoritmo la Criba de campo numérico(NFS)

1: **Selección de polinomios**. Seleccionamos 2 polinomios irreducibles $f(x)$ y $g(x)$ con coeficientes enteros pequeños, para los que exista un entero tal que $f(m) \equiv g(m) \equiv 0 \ (mod\ n)$. Estos polinomios no deben de tener un factor comun sobre $\mathbb{Q}$.

2: **Criba**. Encontrar pares (a, b) con $gcd(a, b) = 1$ tal que
$n(a - b\alpha) = b^{grado(f)} f(a/b),\ n(a - b\beta) = b^{grado(g)} g(a/b)$.
sean suaves con respecto a un factor base escogido.

3: **Algebra Lineal**. Usando técnicas de algebra lineal buscamos un conjunto S de índices, tal que los dos productos
$\prod_{i \in S}(a_i - b_i a),\ \prod_{i \in S}(a_i - b_i \beta)$ sean cuadrados de productos de primos.

4: **Raíz cuadrada**. Use el conjunto S en el paso anterior para encontrar un número algebraico $\alpha' \in \mathbb{Q}(\alpha)$ y $\beta' \in \mathbb{Q}(\beta)$ tal que
$(\alpha')^2 = \prod_{i \in S}(a_i - b_i a),\ (\beta')^2 = \prod_{i \in S}(a_i - b_i \beta)$
Definiendo $\Phi_\alpha \to \mathbb{Z}_n$ y $\Phi_\beta :\to \mathbb{Z}_n$ por medio de $\Phi_\alpha(\alpha) = \Phi_\beta(\beta) = m$, donde m es la raíz común de f y g. Entonces

$$\begin{aligned} x^2 &\equiv \Phi_\alpha(\alpha')\Phi(\beta) \\ &\equiv \Phi_\alpha((\alpha')^2) \\ &\equiv \Phi_\alpha\left(\textstyle\prod_{i \in U}(a_i - b_i\alpha)\right) \\ &\equiv \textstyle\prod_{i \in U} \Phi_\alpha(a_i - b_i\alpha) \\ &\equiv \textstyle\prod_{i \in U}(a - b_i m) \\ &\equiv \Phi_\beta(\beta')^2 \\ \equiv\ y^2 \ (mod\ n) & \end{aligned} \tag{6.8}$$

que es la forma requerida de la congruencia de factorización y esperamos que un factor primo pueda ser encontrado por medio del calculo de $(x \pm y, n)$.

En 2003 *Adi Shamir* y *Eran Tromer* proponen un sistema hipotético para realizar la criba del algoritmo NFS, el cual en la practica es la parte mas costosa del algoritmo NFS, denominado TWIRL[80], ellos mismos estiman que si TWIRL fuera realizado, podría realizar la factorización de números de 1024 bits en menos de un año, al costo de pocas docenas de millones de dólares. TWIRL es más eficiente que diseños previos, debido a la paralelización del algoritmo. También mencionan que en factorizaciones de enteros de 512 bits, este dispositivo es 1600 veces más rápido que los diseños previamente publicados [5].

El algoritmo NFS es más rápido que QS, pero para números en el rango de 100 a 150 dígitos, no hay una gran diferencia, por lo que ambos son rápidos[6][6].

6.3. Retos de factorización exitosos de los laboratorios RSA

El *desafío de factorización RSA* fue un desafío realizado por *RSA Laboratories* el 18 de Marzo de 1991, con la finalidad de motivar a investigadores a incursionar en el problema de factorizar números primos grandes y romper las claves de RSA, esto con la finalidad de conocer los avances en cuestión de factorización de números enteros y que longitud deberían de tener las claves de RSA para ser seguras. Estos laboratorios publicaron una lista de valores semiprimos[7] conocidos como los números RSA y ofrecían una remuneración económica a quien lograra factorizar uno de estos números.

Los números RSA fueron generados en una máquina Compaq que no tenía ningún tipo de conexión hacia ninguna otra maquina. Después de la generación de los números el disco duro de la computadora fue destruido, así que ningún dato de los números debería de existir. Los primeros números RSA generados, de RSA-100 a RSA-500, fueron nombrados de acuerdo a la cantidad de dígitos decimales que tenía cada número, mas tarde se les nombro de acuerdo al número de dígitos binarios que tenía cada valor, iniciando con RSA-576. Una excepción es el valor RSA-617 que fue creado antes del cambio de nombramiento de los números.

Los desafíos de los laboratorios RSA termino en el año 2007, pero en el tiempo de existencia de estos desafíos se lograron importantes avances y con ello la factorización de varios números RSA, a continuación damos la descripción de como se llevaron a cabo dichas factorizaciones, y en el cuadro 6.6 presentamos los valores factorizados y sus factores primos.

En Julio de 1993 *T. Denny, B. Dodson, A. K. Lenstra y M.S. Manasse*, factorizaron el número RSA-120[6] utilizando el algoritmo QS, esto lo llevaron a cabo utilizando 3 programas independi-

[4]Se puede obtener una implementación de este algoritmo en http://sourceforge.net/projects/factor-by-gnfs/

[5]En 1999 Adi Shamir describe un aparato hipotético que puede acelerar el paso de criba en el algoritmo de NFS. Basado en un simple dispositivo opto electrónico portable. Este aparato puede incrementar el tamaño de números factorizables de 100 a 200 bits, y en particular puede factorizar números RSA de 512 bits[81].

[6]El algoritmo NFS presenta una gran complejidad por lo que para ahondar más en detalles de este algoritmo recomendamos leer el libro *The development of the number field sieve* de Lenstra[82].

[7]Números semiprimos, son números que se descomponen en exactamente dos factores primos.

entes corriendo en 4 diferentes lugares, Denny utilizo su implementación de QS en la *Universidad de Saarbrucken*, Lenstra utilizo su implementación SIMD(*Single Instruction Multiple Data*) de QS[83] en una maquina masivamente paralela de *Bellcore*, el resto del equipo utilizo el programa descrito en [84] en estaciones de trabajo de la *Universidad Lehigh, Bellcore y DEC SRC*.

El número total de polinomios generados durante la etapa de la criba fue de aproximadamente 13 millones, y el número total de cribas reportadas fue de aproximadamente 100 millones. La factorización tomo aproximadamente 825 MIPS años[8] y fue completada en tres meses de tiempo real.

En Abril de 1994, un grupo internacional de matemáticos y científicos de la computación resolvieron la factorización del número RSA-129, número de 129 dígitos decimales.

El grupo fue dirigido por *Arjen Lenstra* e incluyo a *Paul Leyland* de Oxford y estudiantes del MIT y del estado de Iowa. Ellos usaron el método de factorización MPQS, y usaron el Internet para solicitar la ayuda de alrededor de 600 voluntarios y sus computadoras alrededor del mundo, cada voluntario recibió un programa que aprovechaba los momentos ociosos de su CPU. Después de determinados cálculos, el ordenador o el propio usuario devolvía sus resultados parciales a la sede central, que los almacenaba para su posterior uso. El proyecto tomo ocho meses y la equivalencia aproximada de 5000 MIPS años.

En Abril de 1996 *Arjen Lenstra* presento la factorización del número RSA-130. La factorización fue realizada usando el algoritmo de NFS, apaleando al record de RSA-129. La cantidad de tiempo usado en resolver este problema fue solo una fracción del anterior record registrado. Para este reto usaron el polinomio,

$$\begin{aligned} &5748,30224,87384,05200X^5 + 9882,26191,74822,86102X^4 \\ &-13392,49938,91281,76685X^3 + 16875,25245,88776,84989X^2 \\ &+3759,90017,48552,08738X - 46769,93055,39319,05995 \end{aligned}$$

y su raíz $125,74411,16841,80059,80468$ modulo RSA-130. Este polinomio fue seleccionado de una lista de 14 candidatos provistos por *Scott Huddleston*. La criba fue realizada en una gran variedad de estaciones de trabajo, en muchos lugares diferentes, como lo podemos ver en el cuadro 6.2.

Excepto por una relativa parte pequeña de la contribución del CWI(Centrum Wiskunde & Informatica) y la entera contribución de la *Universidad de Saarland*, todos los participantes usaron el programa de criba NFS. Se puede decir que se realizo en aproximadamente 500 MIPS años (lo que significa, un 10 % del tiempo de computo en el record RSA-129).

En 1999 un gran equipo conformado por *Stefania Cavallar, Bruce Dodson, Arjen Lenstra, Paul Leyland, Walter Lioen, Peter L. Montgomery, Brian Murphy, Herman Te Riele y Paul Zimmermann*, mostraron al mundo la factorización del número RSA-140[86], usando el algoritmo

[8]MIPS(Million Instructions Per Second) es una medida de pasos y MIPS años es igual al número de pasos procesados en un año a un millón de pasos por segundo, que podría ser igual a $1,000,000 \cdot (365 días/año) \cdot (86400 segundos/día)$, aproximadamente 31.5 trillones de instrucciones. Para consultar un poco más acerca del MIPS, se puede revisar [85].

Cuadro 6.2: Porcentaje de trabajo realizado en el cálculo de la criba para la factorización del número RSA-130.

Porcentaje de trabajo de criba	Investigador	Lugar donde se realizo
28.37 %	Bruce Dodson	Universidad de Lehigh
27.77 %	Marije Elkenbracht-Huizing	CWI, Amsterdam
19.11 %	Arjen K. Lenstra	Bellcore
17.17 %	Participantes del *Factoring project*	
4.36 %	Matt Fante	IDA
1.66 %	Paul Leyland	Universidad de Oxford
1.56 %	Damian Weber	Universidad de Saarland

NFS. El coste computacional de tiempo en este Nuevo reto fue de 2000 MIPS años.

En dicho trabajo se usaron dos polinomios

$$\begin{aligned} F_1(x,y) = 439682082840x^5 + 390315678538960y * x^4 \\ -7387325293892994572y^2 * x^3 \\ -1902715324374298871482 4y^3 * x^2 \\ -6344102569446461791393061 3y^4 * x \\ +3185539170714743503922235074 94y^5 \end{aligned}$$

$$F_2(x,y) = x - 3443565780924253695177900 7y.$$

Estos polinomios fueron seleccionados con la ayuda de un método de búsqueda polinomial desarrollado por *Peter Montgomery y Brian Murphy*[87] y la selección tomo 2000 horas CPU en cuatro procesadores SGI a 250 Mhz en el CWI, llevándose a cabo en 4 semanas.

El polinomio $f_1(x,y)$, fue escogido por tener una combinación de 2 propiedades; Siendo inusualmente pequeño en su región de criba y teniendo inusualmente muchas raíces modulo *pequeños primos*. El efecto de la segunda propiedad daba a $f_1(x,y)$ una suavidad de rendimiento comparable a la de un polinomio elegido al azar para un entero de 121 dígitos decimales.

La criba fue realizada en cerca de 125 SGI y estaciones de trabajo *Sun* corriendo a 175 MHz en promedio. El tiempo total de uso de CPU en este paso fue de 8.9 años CPU, ejecutándose en 1 mes de tiempo real, y fue realizada en 5 diferentes lugares teniendo las contribuciones que se muestran en al cuadro 6.3.

Los datos fueron tomados en el CWI y requirieron 3.7 GBytes de memoria en disco. El filtrado de los datos y la construcción de la matriz fueron realizados en el CWI y se llevo a cabo en una

Cuadro 6.3: Porcentaje de trabajo realizado en el cálculo de la criba para la factorización del número RSA-140.

Porcentaje de trabajo de criba	Investigador	Lugar donde se realizó
36.8 %	Peter L. Montgomery, Stefania Cavallar, Herman J.J. te Riele, Walter M. Lioen	CWI, Amsterdam, Paices bajos
28.8 %	Paul C. Leyland	Microsoft Research Ltd, Cambridge, Reino Unido
26.6 %	Bruce Dodson	Universidad de Lehigh, Belen, USA
5.4 %	Paul Zimmermann	Inria Lorraine y Loria, Nancy, Francia
2.5 %	Arjen K. Lenstra	Citibank, Parsippany, USA, Universidad de Sidney, Australia

semana. La matriz resultante tuvo 4 671 181 renglones y 4 704 451 columnas.

Durante Febrero de 1999 cuatro diferentes procesos de raíces cuadradas (El procedimiento para realizar el cálculo de estas raíces cuadradas se encuentra descrito en [88]) fueron iniciados en paralelo en cuatro diferentes procesadores de un SGI Origin 2000 a 250 MHz del CWI. Después de 14.2 horas CPU de trabajo, uno de los cuatro procesos se detuvo, dando como resultado los factores primos del número RSA-140.

En Agosto de 1999 un gran equipo rompía el número RSA-155[89], este reto representaba una paso adelante en la investigación de sistemas de RSA, ya que, según palabras de los mismos autores, las claves RSA de 512-bits protegían al 95 % del comercio electrónico en Internet, al menos fuera de Estados Unidos y eran usados en la *Seguridad de Capa de Transporte*(SSL por sus siglas en ingles). Sin embargo el 12 de Enero del 2000, el *Departamento de Comercio de la Oficina de exportaciones* de estados Unidos decreto una nueva encriptación RSA con claves mayores a 512-bits. Como resultado, se pudieron reemplazar claves de 512-bits por claves de 768-bits o 1024-bits, creando unas condiciones más favorables para la seguridad de las comunicaciones en Internet.

La factorización del valor RSA-155, fue realizada nuevamente por medio del algoritmo NFS. Consumiendo un tiempo total estimado de 8000 MIPS años, la ganancia se debió al nuevo método de selección de polinomios para NFS de *Murphy y Montgomery*[87], que también fue aplicado para romper el RSA-140. Cuatro de los investigadores pasaron aproximadamente 100 MIPS años buscando polinomios para su uso en el reto RSA-155, y fue utilizado el par encontrado

por Dodson:

$$F_1(x,y) = 119377138320x^5 - 80168937284997582y * x^4 \\ -662698522341185744 45y^2 * x^3 \\ +1181684843007952188035685 2y^3 * x^2 \\ +7459661580071786443919743056y^4 * x \\ -4067984354236215936191370840 5064y^5$$

$$F_2(x,y) = x - 391230797211680007713134490 81y$$

Los polinomios escogidos presentan las mismas características que los polinomios usados en la descomposición de RSA-140. La selección de estos se realizo en aproximadamente 100 MIPS años realizados en procesadores de un SGI Origin 2000 a 250 MHz. El código para la selección de polinomios fue realizado por Arjen Lenstra, para usar su librería de aritmética de precisión múltiple LIP[9]. Brian Murphy, Peter Montgomery, Arjen Lenstra y Bruce Dodson realizaron la búsqueda de polinomios con este código, tomando aproximadamente 9 semanas de búsqueda.

El paso de criba fue realizado en cerca de 160 SGI y estaciones de trabajo Sun, corriendo entre 175 a 400 MHz, 8 procesadores de un SGI Origin 2000 a 300 MHz, 120 PC's Pentium II a 300-450 MHz y en 4 *Digital/Compaq boxes* a 500 MHZ. El tiempo total estimado en la criba fue de 35.7 años CPU, con una equivalencia aproximada a 8000 MIPS años, y un tiempo de calendario de 3 meses y medio, estos datos se presentan con más detalle en el cuadro 6.4.

Los resultados fueron recogidos en el CWI y requirieron 3.7 GBytes de espacio en disco, el filtrado de los datos y la construcción de la matriz fueron realizados a lo largo de un mes en el CWI. La matriz resultante tuvo 6 699 191 renglones y 6 711 336 columnas.

En Agosto de 1999, 4 diferentes raíces cuadradas (Estas raíces cuadradas se encuentran descritas en [88]) fueron iniciadas en paralelo en cuatro diferentes procesadores de un SGI Origin 2000 de CWI. Un proceso encontró la factorización después de 39.4 horas CPU.

De acuerdo a las factorizaciones realizadas hasta ese momento los autores de [89] determinaron la formula 6.9, donde y es el año y D es el número de dígitos de un número cualquiera.

$$y = 13{,}24D^{1/3} + 1928{,}6 \tag{6.9}$$

De acuerdo con la formula 6.9, un número de 768 bits(231 dígitos), podrá ser factorizado en el año 2010 y un número de 1024 bits(309 dígitos) será factorizado alrededor del año 2018.

Este mismo valor ,RSA-155, fue factorizado en 2008 por *Jiun-Ming Chen, Shoou-I Yu, Yi Ou-Yang, Po-Han Wang, Chi-Hung Lin, Po-Yi Huang, Bo-Yin Yang y Chi-Sung Laih* en [90], en un tiempo de 3 días, una reducción considerable en comparación con los 6 meses de ejecución

[9]Esta libreria se encuentra disponible en la página principal de Lenstra, http://www.win.tue.nl/~klenstra/

Cuadro 6.4: Porcentaje de trabajo realizado en el cálculo de la criba para la factorización del número RSA-155.

Porcentaje de trabajo de criba	Tiempo(días CPU)	Investigador	Lugar donde se realizo
20.1 %	3057	Alec Muffett	Sun Microsystems Professional Services, Camberley, Reino Unido
17.5 %	2092	Paul Leyland	Microsoft, Cambridge, Reino Unido
14.6 %	1819	Peter L. Montgomery, Stefania Cavallar	CWI, Amsterdam
13.6 %	2222	Bruce Dodson	Universidad de Lehigh, Bethlehem, PA, USA
13.0 %	1801	Francois Morain y Gerard Guillerm	Escuela Politécnica, Palaiseau, Francia
6.4 %	576	Joel Marchand	Escuela Politécnica /CNRS, Palaiseau, Francia
5.0 %	737	Arjen K. Lenstra	Citibank, Parsippany, NJ, USA y Universidad de Sydney, Australia
4.5 %	252	Paul Zimmermann	Inria Lorraine y Loria, Nancy, Francia
4.0 %	366	Jeff Gilchrist	Tecnologías confiables Ltd., Ottawa, Canadá
0.65 %	62	Karen Aardal	Universidad de Utrecht, Países Bajos
0.56 %	47	Chris Y Craig Putnam	

para romper el record descrito en [89].

Los pasos de la selección del polinomio, la criba y la raíz cuadrada se realizaron en un cluster HP(Que cuenta con 424 núcleos, 106 nodos), y el paso de la reducción de la matriz, se realizo en una supercomputadora IBM P595 SMP (Symetric Multi-Processing) de la *Universidad Nacional de Taiwán*(NTU por sus siglas en ingles), esto debido a que este proceso solo puede ser parcialmente paralelizado. El paso de la criba se realizo en 76.2 horas. Por otro lado la selección polinomial se trabajo en 50 núcleos del cluster y tomo 24 horas de ejecución, obteniendo los

polinomios

$$F_1 = 83772000x^5 - 55340006499600x^4 - 57899874664053626478x^3$$
$$+427645602820216392547945 7x^2 + 2350079225298842053344018218 00x$$
$$-14068501632188545244303052842000 79$$
$$F_2 = 2054098293316505557x - 167185341081359137443707501330$$

El procesamiento de la reducción matricial tomo 2.2 horas, teniendo esto, la supercomputadora SMP trabajando con OpenMP, necesito 37.5 horas para resolver el problema y por último el paso de la raíz cuadrada se realizo en 2.98 horas. Con lo que llegan a la conclusión de que con los 424 núcleos del cluster, el tiempo de calendario de factorización es menor que 3 días.

En el mes de Abril del 2003 *Jens Franke* presentaba la factorización del número RSA-160 de 530 bits, factorizado por medio del algoritmo NFS. Los cálculos para la factorización del RSA-160 se llevaron a cabo en la Oficina Federal Para la seguridad de la información(BSI por sus siglas en Alemán) en Bonn, Alemania, la criba se realizo entre el 20 de Diciembre del 2002 y el 6 de Enero del 2003, utilizando 32 R12000 y 72 Alpha EV67. Después del paso del filtrado de datos se obtuvo una matriz de tamaño casi cuadrado con 5037191 columnas. Con el algoritmo de *Block Lanczos*, descrito en [91], se trabajo esta matriz en un periodo de 148 horas en 25 CPU's R12000. El paso de la raíz cuadrada se realizo en un promedio de 1.5 horas CPU a 1.8 GHz, devolviendo el resultado después de la sexta solución de Lanczos.

La factorización de RSA-160 tomo menos tiempo y requirió menos maquinas que la factorización del valor RSA-155, presumiblemente debido a los avances en el hardware, sin embargo el calculo en MIPS años no han sido dados.

El 3 de Diciembre del 2003, un equipo de investigadores alemanes obtuvo la factorización del número RSA-576 de 176 dígitos decimales, este valor fue el primer desafío de RSA que uso la "nueva" forma de nombrar a los números RSA, es decir nombrados en base al número de bits, el resultado fue dado a conocer por J. Franke.

La criba "lattice" fue realizada por *J. Franke* y *T. Kleinjung* usando hardware del *Instituto de Computación Científica* y del *Instituto de Matemáticas Puras* en la *Universidad de Bonn*, del *Instituto Max Planck para Matemáticas en Bonn* y del *Instituto de Matemáticas experimentales en Essen*. La criba lineal fue realizada por *P. Montgomery y H. te Riele* en el CWI, por *F. Bahr* y su familia, y por NFSNET . No se han dado más detalles del proceso.

En el mes de Noviembre del 2005 *F. Bahr, M. Boehm, J. Franke* y *T. Kleinjung* factorizaron el número RSA-640, por medio del algoritmo NFS. La factorización se llevo acabo con una matriz de $36 \cdot 10^6$ columnas y renglones, teniendo $74 \cdot 10^8$, entradas no cero, la matriz nuevamente fue resuelta con el algoritmo descrito en [91].

La criba fue realizada en 80 CPU's Opteron a 2.2 GHz y se realizo en 3 meses. El paso de la matriz fue llevado a cabo en un cluster de 80 CPU's Opterons a 2.2 GHZ conectados por medio de una red Gigabit y tomo aproximadamente 1.5 meses de ejecución.

El tiempo total de factorización se realizo en 5 meses.

El equipo formado por *F. Bahr, M. Boehm, J. Franke* y *T. Kleinjung* lograron la factorización del número RSA-200 de 663 bits, el 9 de Mayo del 2005, este es el número más largo factorizado hasta la fecha.

La criba tomo un tiempo estimado de 55 años en un simple CPU Opteron a 2.2 GHz, el paso de la matriz fue reportado con un tiempo de ejecución de 3 meses en un cluster de 80 Opterons a 2.2 GHZ. La criba comenzó a finales del 2003 y el paso de la matriz fue completado en Mayo del 2005. P. Montgomery, H. te Riele del CWI, y F. Bahr y su familia también contribuyeron en el proyecto.

La Universidad de Bonn, el CWI y el BSI proveyeron de recursos computacionales. El valor de MIPS años no ha sido inmediatamente disponible.

En abril del 2006 *Kazumaro Aoki, Yuji Kida, Takeshi Shimoyama* y *Hiroki Ueda*[1],llevaron a cabo la implementación del Algoritmo FSN y factorizaron números RSA de 100 hasta 150 dígitos, todos estos números factorizados bajo las mismas condiciones. Esto debido a que hasta 2006 muchos tiempos de ejecución para el algoritmo NFS habían sido anunciados, estas estimaciones son usualmente basados en resultados previos de factorización. Sin embargo, ya que los resultados previos fueron realizados por varios programas y/o computadoras, es difícil comparar los tiempos de ejecución, por lo que se quiso "estandarizar" el tiempo de ejecución del FSN.

Estos resultados se obtuvieron con maquinas Pentium 4 (*Northwood*), 2.53GHz, FSB 533MHz, Intel Desktop Boards D850EMV2, i850e chip set, 1024MB RDRAM, PC800, FreeBSD 4.7-RELEASE-p13, y podemos observar los resultados en el cuadro 6.5.

Cuadro 6.5: Números RSA factorizados en el trabajo [1] por Kazumaro Aoki, Yuji Kida, Takeshi Shimoyama y Hiroki Ueda.

Número factorizado	Tiempo en segundos
RSA-100	54 271
RSA-110	193 676
RSA-120	790 138
RSA-130	2 315 347
RSA-140	6 726 258
RSA-150	20 597 260

Cuadro 6.6: Números RSA factorizados y sus factores primos (separados por un *).

Nombre	Número	Factores	Año de factorización
RSA-120	227010481295437363334259960947493668895875336466084780038173258247009162675779735389791151574049166747880487470296548479	3274145556934980157511463037491414880636424032401714634068836939326711083018119732540189970064136196586312733668067301 3	1993
RSA-129	114381625757888867669235779976146612010218296721242362562561842935706935245733897830597123563958705058989075147599290026879543541	3490529510847650949147849619903898133417764638493387843990820577*327691329932667095499619881908344614131776429679929425397982885 33	1994
RSA-130	180708208868740480595165616440590556627810251676940134917012702145005666254024404838734112759081230337178188796656318201321488055 7	3968599945959745429016112616288378606757644911281006483255515724 3*45534498646735972188403686897274408864356301263205069600999044599	1996
RSA-140	212902463182587575474978820162715174978067039632772162782333832153819499840564959113665738530219183167831073879953172308895692308734419364 71	3398717423028438554530123627613875835633986495969597423490929302771479*626420018740128509615165494826444221930203717862350901911660653946049	1999
RSA-155	109417386415705274218097073220403576120037329454492059909138421314763499842889347847179972578912673324976257528997818337970765372440271467435315933543338 97	1026395928297411057720541965739916759007165678080380668033419335217907113077791*1066034883801684548209272203600128786792079585759892915222706082371930628086 43	1999
RSA-160	215274110271888970189601520131282542925777358884567598017049767677813314521885913567301105977349105960249790711158521430207931466520284014061994699492757040775 3	4542789285848139407168619064973883165613714577846979325095998470925000415733535 9*4738809060383201619663383230378895197326892292104095794474135464881202849390 9367	2003
RSA-576	188198812920607963838697239461650439807163563379417382700763356422988859715234665485319060606504743045317388011303396716199692321205734031879550656996221305168759307650257059	3980750864240649373971255005503864911990643623425267084063851895759463889572617685833 17*472772146107435302536223072973048224632914695302097116459852171130520711256363590397527	2003
RSA-640	3107418240490043721350750035888567930037346022842727545720161948823206440518081504556346829671723286782437916272838033415471073108501919548529007337724822783525742386454014691736602477652346609	1634733645809253848443133883865090859841783670033092312181110852389331001045081512121181675115 79*190087128166482211312685157393541397547189678996851549366663853908802710380210449895719126146557 1	2005
RSA-200	279978339112213278708294676387226016210704467869554285375600099293261284001076093456710529553608560618223519109513657886371059544820065767750985805576135790987349501441788631789462951872378692218239833532461934402770121272604978198464368671197400197625036493034687761212536794232000585479565280883497925869954478333033347085841480059687737975857364219960734330341455767872818152135381409304740185467	2005

Capítulo 7

Conclusiones

Logramos conocer algunas de las características de los algoritmos de factorización de números enteros y sabemos que hay algoritmos de propósito especifico que trabajan de forma rápida y eficiente, pero, siempre y cuando se cumplan ciertas características de los números a factorizar o de los números primos encontrados, así como también, es necesario que los factores primos a encontrar no sean lo suficientemente grandes, para que los algoritmos puedan encontrarlos. Por otro lado tenemos los algoritmos de propósito general, los cuales presentan deficiencias al momento de intentar factorizar números n grandes, son útiles y rápidos cuando trabajan con números relativamente pequeños, pero, conforme el valor n va creciendo, la eficiencia del algoritmo va disminuyendo.

Con los resultados obtenidos mediante el análisis y verificación de resultados comprobamos que, por ejemplo, en el caso de las claves para RSA, tienen que ser números primos grandes p que presenten a su vez factores primos grandes cuando se nos de el caso de realizar las operaciones $p+1$ y $p-1$, para poder evitar un ataque por medio del algoritmo de Williams $p+1$ y el de Pollard $p-1$, también tenemos que verificar que los números primos p y q, deben ser lo suficientemente alejados entre sí para evitar factorizaciones por medio del algoritmo de Fermat.

Hasta ahora las claves para RSA, con números grandes(De aproximadamente 1024 bits) siguen siendo eficientes, ya que, no existe un algoritmo de factorización capaz de factorizar un número grande en un tiempo humanamente razonable, claro que, los algoritmos QS y NFS han presentado adelantos significativos en este campo, así como el adelanto presentado por las supercomputadoras en cuestión de hardware.

También presentamos la factorización de números n grandes, pero, con la desventaja que sólo es en situaciones especificas donde n tiene una forma determinada, aunado a eso tenemos el factor tiempo, que debe ser mejorado, esperamos realizar el mejoramiento de estos tiempos de ejecución y el aumento de la eficiencia del algoritmo 18, para así poder ampliar su campo de búsqueda y tener la posibilidad de realizar mejores exploraciones para encontrar un factor primo.

El trabajo en esta área del conocimiento es relevante y apasionante, pero, existen grandes huecos que todavía es necesario llenar para poder tener algoritmos de factorización eficientes para cualquier tipo de número n, sin tener limitantes dadas por su tamaño o su forma o de sus factores primos p.

Apéndice A

KANBALAM

Kanbalam[1] es una supercomputadora paralela de memoria distribuida, de las más poderosas existentes en América latina con una capacidad de procesamiento de 7.113 Teraflops (7.113 billones de operaciones aritméticas por segundo). Contiene 1,368 procesadores AMD Opteron a 2,6GHz, alrededor de 3,000 Gbytes de memoria RAM y 160 Terabytes de almacenamiento distribuidos en 337 nodos de cálculo, cada uno con 8 GB RAM y dos procesadores duales y en 5 nodos especializados, con 64GB RAM.

KanBalam es siete mil veces más potente que la primera supercomputadora de la Universidad, CRAY-YMP (1991), y 79 veces más poderosa en cálculo que el equipo AlphaServer SC45, adquirido en 2003. Esta supercomputadora ofrece sus recursos a investigadores y estudiantes de posgrado para la realización de proyectos de investigación relacionados con diversas áreas del conocimiento. La supercomputadora ha sido definida, también, como una máquina repatriadora de cerebros, porque los investigadores que permanecen en el extranjero y los que trabajan en México podrán realizar los más complejos procesos de cómputo en este avanzado equipo que ya se encuentra en Ciudad Universitaria[2]. Kanbalam tiene como ubicación el campus principal de la Universidad Nacional Autónoma de México, en el D.F.

Basándonos en el manual [92], damos una pequeña lista de comandos que son útiles para iniciar el computo en Kanbalam. Cabe mencionar que para usuarios de Windows es necesaria la aplicación *SSH Secure Shell Client* para la conexión remota.

Kanbalam utiliza un sistema de colas de tipo **lfs**.

Usamos el comando **bqueues** para poder visualizar las colas que existen y poder ver cual es la que nos conviene utilizar, en el cuadro A.1 podemos ovservar las colas existentes, .

El comando busca procesos que tengamos pendientes en nuestra cuenta, es decir, aquellos que todavía no han entrado en ejecución, pero ya han sido lanzados por nosotros.
Un comando útil es **man bjobs**, que nos muestra las opciones que podemos utilizar.
bnist nos muestra los históricos de los *jobs*.

[1]Si se quiere ahondar un poco más en la información referente a KanBalam, en la página http://www.super.unam.mx/ se encuentran todas las especificaciones y manuales de esta y otra supercomputadoras

[2]Información tomada de la Gaceta UNAM N°3953, con fecha de publicación 18 de enero de 2007

Cuadro A.1: Nombre de las colas existentes en KanBalam.

especial2	pruebas_s	q_12h_128p
q_12h_64p	q_12h_32p	q_12h_16p
q_12h_4p	q_2d_128p	q_2d_64p
q_2d_32p	q_2d_16p	q_2d_4p
q_5d_128p	q_5d_64p	q_5d_32p
q_5d_16p	q_5d_4p	q_10d_128p
q_10d_64p	q_10d_32p	q_10d_16p
q_10d_4p	q_dw	especial
pruebas	q_ch	

Para subir archivos a la computadora Kanbalam utilizamos la siguiente instrucción:

```
scp nombre_del_archivo usuario@nombre_maquina:./
```

después de teclear *enter* tecleamos nuestro pasword personal, después de esto nuestro archivo será copiado en la memoria de Kanbalam. Ahora si lo que queremos es copiar un archivo de la memoria de Kanbalam a la memoria de nuestra computadora, lo que hacemos es teclear el siguiente comando

```
scp Lusuario@maquina:ruta_de_archivo/archivo* ./
```

tecleamos enter e ingresamos nuestro pasword, después de esto la información ha sido copiada en nuestro disco duro.

Para la compilación de estos programas es necesario agregar las cabeceras de las librerías de **GMP**, ya que de otro modo no lo compilara, la forma de compilarlos es de la siguiente manera

```
mpiCC nombre_del_archivo.cpp -lgmp -lgmpxx
```

Ya que tenemos compilado nuestro programa, lo mandamos ejecutar de la siguiente forma

```
bsub -n numero_de_procesos -o archivo_de_salida.txt -q nombre_de_cola mpirun -srun ./nombre_de_archivo
```

este comando nos colocara en fila para ejecutar el programa, lo que significa que no se ejecutara de forma instantánea, sino que tendrá que esperar a que la cola se desocupe. Kanbalam nos dará un número identificador del programa que mandemos a ejecutar, esto para poder saber cual es su estado, este dato nos lo muestra através de **job<número_identificador>**. Si lo que nosotros queremos es ver nuestro programa corriendo, tenemos que teclear el comando **bpeek**, que nos muestra su ejecución siempre y cuando el programa se encuentre en *status RUN*.

Ahora bien, si hemos mandado un proceso a una cola, y después de ese envío nos damos cuenta que tiene un error o queremos agregarle algo antes de probarlo, podemos retirar ese proceso de esa

cola através de **bkill<número_identificador>**, que elimina el job pendiente. Pero si tenemos en ejecución el programa y lo queremos detener utilizamos **bstop<número_identificador>**.

Y por último tenemos el comando **sinfo** que nos muestra todos los procesos que están activos dentro de Kanbalam.

Apéndice B

Tablas de números primos grandes

No. de digitos	Valor primo
613	1813524129856179104350319779308454504251294814373172645208870949013599441238965525073711462725936394706341995620879593314161108700867824856541202745623060122991445220881104498381292082102954918609830057035254302285176855620220967679509441202690337509729754682671588544615517055157094732648170252917168840389121635672475250616282625371646861993233807451581396042682207933052618343538285556797723885864146921850973031588709217332165159334500387551136500922291089744574104660499031223314081516293523873181860020874500791558395211124076952149066670218248699302363554901076058179789108357583673860334990147394823961137
609	3692027951661602411136644501849459495625600192127794473145095580239412543238936329547458189588632725379360740270520344694953397192320490343121341094509487221073789130458274630255073457049989655150305490706950941134317702809896106839392184858972669171202269305113168861187941887534801980146926410661988681573944698030283490668327820382017227184922246440515871422398631785530574803620288185663118660146878912563055846068219090659945356951344437196939130542123554040256727729028972360167104064115480197845806231422029298775235450334032883039630843278193606071587041736718359486541344376188261116317162352188159533
605	2178958894984420686459303884472060608844192747950775774991203718271607969333649863991653794610855007896223288639353366793527736775448825745468213582689735139916070072272352827109934759826481146807309661654243945428657756615849921411350439600387905404392274141355741773600060131925638562409659118662646766745718070131187140384990451122531413588835131279813427421151222725171491267481284339980594111009612778125770529158889228674453810018374907766580449158529482651101081664838820370642534470261772910637736439531517309889945381865795141822452245235965156165698007656794048600480485385752376128205687107659833
601	1288257594291368503286806127747463999553146948061236712185883716608494719956042251384447082068614761674484621402006247365216824391302368301683938502240590717698989045921930251336132647408348792010943396981343233669538699666459691031896913562958439993137208313441966284498084505099703536957348420635359327625468883842489736540729839850142730039514680903283331808650362259176712349226253009329900739089587194784028196517022754371844684985070993062436142595066114763515913827827955366220494680511841732515882934441984804835907425007657217822231555137549699454286435241776330853024035625826984321897168663

No. de digitos	Valor primo
489	29887411513113581813299982201993055082186063791890901129716222512483266024931292768298925425219795359778419429494926064975264764940563220363626197087709604816805014640228147096506221341378858719450176587504998756156909944505962469588959010271708893277864177451914149879704989734624693374062058295660269939658791378209447361217502867645118727545798156963791061000696040873018828823816082125347106703485468581585415014306262014245154736718384485779859107624873497932363615550781925672646012 7
481	26545227244261241774530076101955090824501741204989198673725220040253533605359140662613222157746516083287591856301558568919105490774999492287331841172029679811791107898728521014224412680060314416013908903158353388445704726161300593233703469506519139228003422678605921343316721753560560062443315210015064446326063395145857528279379654072781972872352243488716085424784614983792267875159162700921969833453013544430145148894107694605672989399591842912273319852335622702083443067947887837
477	31149058019550858688723393689222120188338114532960805765929617507924822348461793783869070825799713779966664933468151336445793816915042821271217837563986951199003881599071252070199967941868475024658423965217499869098456613660291707619928971493216544505985269638651975396816730269426895827778868927658583035978213977304124950473828374475263704205025152413941391982922025155975919680346898626167212314632874259917216016124239551814984620976200925983024170996663021615623598544331
470	30212002548506384662045393756471403148290729273804485833323263177220161809769968442651783120533254362654570766572149546123235543303928184694726209451635038834512637630328968774684357897325041536366481184790051627810292579029124385673868273133701654005358332910318751153892541945947191491478199070398382084853990042570059528705066495037444977013453249262275405229688193505821046393106110732148406262815792796612003695008054147735339182027844468561401973489903456553111067
457	11802591341172984089219988747318485881282657291837163418263694494736419900488872653089823729207106656353343821787781408772777090028781113191064472411757755241777896850654664153981643176772209732579700430935410547887949549728437165713380462557701912149953430983563406405067364712388605223884184988517319416122382837304547455634138834717583246876254269597950415900864276250953218309696208765048316050685268100735902594499662890681466426064652239305156760070374
444	18038306724583377013064114292668278127177449565969699535473015774815071419391698583135608929320478837849630234388589029357996133844712998535823002066183255612169251091553771336209724643226059537213743869310670197888878089810513969168637824967386441911015195271713167771492221945755101421909866104674786543801858092739580872441233690876741549816425049530611465815994512914111995836203138438027384157500325986104589389129094092852328554928955990 3
417	46417666446909406245250003551891332610739449736744210406414772508291780642687399930452039673723332282477187344443987412342134893075546352285613176817791198345339136458492842484638008149127819160356282146687533766111167868918280837719382155181122315086516069196228807754139760502383759301984595691697889655294951321706649621144287897515937545620725282247416162096799277008590627921586562127950758500684529130919460739

No. de digitos	Valor primo
325	382356339485542826054730279379350257044095523533512283921935322183-502292161070822487878418313950319816616554215131868248072248995015-885151598138079889584312445351470883160167277293168493762970116163-982446919944385041500344149748833593331302906473991419105298219817-214846860342110271190558535541162802324345104895330408475557 9
316	713625374502078089024513373383468435287694010839863963742943591423-214223797521375367245328234178359675978417667115839588425725103618-114449775082075896409496145560154068333608321924229874205526337056-050809612610861765687081134884455736537424183248520954708383183288-811475945607859302618657172971514268796280418321453 9
299	608896537368378600240338162752389491590026686032996830554355673899-687835544029997501760503305242491803659686593013154267410692107333-675166889523704847033264172488910989359114908612146796364050297702-034485180772851996133949498802879711768773697448361183152294998166-559658841022260570487508316713858069
295	426040118505722502267239128710040226413396785637417317768230950111-732322658851103765575499094068354186719623980557762571655955854557-567287216291425165850310783997278889839850901631784772155086970124-569329121727436325310627972853960055813583611424825904808490762780-968135209223524048759801509332 9
280	285227828123368996844096257968664577528396568266380257523640907594-273642906654865802574401106472445454738155451024341502384304967069-054722284541811275406810820881517012423381554294275836878746411688-135808327839715046203042792305621833579515053095128333252679940082-1362950032917493
266	946711117444503371967647652191463255247750968962584421834009743155-116698879645133783794057073813173023917136022927516394788777609469-710352667114935512704294971888774949601097080740861415638438815169-162835147971397323453083984171363301726467823510445946169531649315-69
258	144824654151176771317297415556831062555890151439892985801385965478-674311287024795577941039063894102284307441862606373261633316788832-210600181551861932234100800430019528014712598679221612136998569480-439253838239874007771255056820921438944231922740701645283219
236	779816305914359385764524078285635625058411815582882382126919863509-982158737752977143097153924213839208385124710068747916084558231225-625686433989647791521077446842849332331569110174387040754922723726-140330525054605980857514615188963964 07
220	266782493291027868072560719782628719694805717034421300290922724082-696749887746660917278544872431560194997135293017295430393079605967-924827518506288779448921188359396281319271046246295384122714198132-7556789477751771771199
203	263440677352441909034476160694285417115413068251519151911165030242-578961654487501145951777571314017846402381648785714864125196123144-749085555058506086079589786397823071073039815901077101117935394214-24557
178	48369519010088349200112477286542441076877343632239603905239654114 5-866639847916880331624175034746837536693360794847866953799871689971-171326793926396715508948013505701642616895811 7
149	909805519169168723889534265586227648117422442030717254770696770690-670172431148026024810814376883311089628038791978211119782633790623-31679117881963377
145	463429869177449431484074096162503895740333354742622888534380995665-581791173160159955588230632071776227398145268937556601356272305737-2233043901893

Números primos de 107 dígitos decimales
10037056897598308769206656434231211229065141960521875100413067761339202477579237479511-55581343823503674042 3
10559405886582401350337176896602023876174856396516637513014264808370813566365887439672-246810449840582336771
12728364933556083789865894313201358510199962034720054948119897579819953650268069724577-897506650348377627567
12759498317005466791390229969898956614994580775938285820394141112424486761520254159289-461870710642747894899
13679662761176119947553928267850189490035534683054887156499561076069576494084816340764-586408492676358018267
13880300121183254846266313611012488387600855459794597222266908286187678766598893808905-778976881240077518851
13973700271531403850839320581105282701984711683449289839089638884001278100355447113040-472069062123188320847
14290223003266797699670066424197530100730002219167970373877781465480698064752655532608-043103675115952705389
15286491273647053748448807438520669454157801938151358286653574508825757624822557443378-102753604535801260013
17066283027503448668923329146400027778250173311126889818331163122718233818072434294389-198899051363968209159
17343024213720186460250757205934233154202339899733386460768883412536305918091851491825-326579587313926140999
17633602459247761141145566684451487989521148177702079353284897168452816231122395491332606441500613819694 31
18187084831681236723799412787513559550856447994983201220203930296481425823151073944005-516005221961297833111
19164327145509092049424393122743722284687536260999892488812130069901492926344640922451-841877114534586779921
20279940052445316270713087487740987706494707821319832078639189988230596079547916499616-231589275082854692651
21779531355257264177468588285341963087435510523331285759848586808682274271528133438223-248458179261689235809
22070109600784838858362387747852878732185285441368107234408193112991249976548521495531-182522742009145064241
23796282749811740831767220269197484764686924537801130041613473420731474700419636264539-028930084997007664093
23868927311193634501990670134825213675874368267310335410253374996808718626674733278866-012446225683871621201
23886223635332180613948634388546101511871378679098241450405732514922348132925946853705-770426259180743991941
25207662799517103567537103372821932182042974139694262918045846898803642410518663971463-280100818341793116477
25685041345740976257576916775518436455560461505040469626250914398739816783052158637040-600349742855470548901
25908163927128221101834655648517889539921895817104457544216326382405637413692813752473-478292174965124131447

Apéndice C

Comandos para usar el archivo .h

Como trabajo extra para esta tesis, realizamos un archivo .h para c, que contiene los algoritmos básicos para trabajos relacionados con la criptografía. Este archivo lo denominamos *herramientas_cripto.h*. Esta cabecera debe de ser instalada y utilizada, después de haber instalado los archivos del proyecto GMP, ya que trabaja con el tipo de datos de dichas bibliotecas.

El archivo fue realizado y compilado con Dev-c++ 4.9.9.2.

A continuación damos una breve explicación del uso de las funciones que contiene el archivo.

```
mpz_class ex_mod_rap(mpz_class a, mpz_class n, mpz_class m);
```

La función *ex_mod_rap()*regresa el resultado de una operación de la forma $a^n \ mod \ m$.

```
mpz_class inverso_multi(mpz_class b, mpz_class a);
```

Esta otra función regresa el inverso multiplicativo de b $(mod\ a)$, si la función egresa 0 es que no hay inverso multiplicativo de b $(mod\ a)$.

```
mpz_class gcd (mpz_class a, mpz_class b);
```

La función *gcd()*, regresa el máximo común divisor de los números (a, b).

```
mpz_class gcd_bin(mpz_class a,mpz_class b);
```

gcd_bin(), como la función anterior, también devuelve el máximo común divisor de (a, b), pero este es calculado de forma binaria, en comparación con gcd(), este algoritmo es más rápido si se ejecuta en bajo nivel, y más lento si se ejecuta en alto nivel de programación.

```
mpz_class mcm (mpz_class a, mpz_class b);
```

mcm() regresa el mínimo común múltiplo de dos números $[a, b]$.

```
mpz_class potencia_perfecta(mpz_class n);
```

potencia_perfecta() determina si un valor es una potencia perfecta, regresando 1 si lo es y 0 si no lo es.

```
int simbolo_jacobi(mpz_class a, mpz_class b);
```

simbolo_jacobi(), calcula el valor del símbolo de Jacobi, devolviendo el valor de 1 si a es un residuo cuadrático modulo n, devuelve el valor de -1 si a no es un residuo modulo n y devuelve el valor de 0 si a es un múltiplo de n.

```
mpz_class genera_primo(mpz_class s);
```

genera_primo(), genera un número primo aproximadamente del doble de tamaño del número primo s que ingresamos. La máxima longitud que puede duplicar es de 2^{40000}, es decir, despues de rebasar este valor cada número generado no será duplicado en tamaño al número s, sino que solamente se agrandara 2^{40000} digitos aproximadamente, con respecto a s.

Las siguientes 4 funciones realizan un test de primalidad, pero basándose en diferentes algoritmos (Divisiones sucesivas, Fermat, Lehman, Solovay-Strassen y Miller-Rabin respectivamente), todas regresan un valor 0 si el número ingresado n es primo y 1 si n no es primo.

```
int primality_divisiones(mpz_class n);

int primality_fermat(mpz_class n);

int primality_lehman(mpz_class n);

int primality_st(mpz_class n);

int primality_mr(mpz_class n);
```

Las siguientes cuatro funciones, son las que utilizamos para factorizar un valor n que se ingrese en ellas, estas funciones devolverán un factor primo de n, cada una de estas funciones está basada en un algoritmo diferente de factorización de enteros. Tenemos los algoritmos de Fermat, Pollard Rho, Pollard $P-1$, algoritmo $P+1$(Para esta función, también es necesario ingresar el valor máximo de búsqueda *max*) y el algoritmo para buscar factores primos cercanos a las raíces de n, respectivamente.

```
mpz_class factor_fermat(mpz_class n);

mpz_class factor_prm(mpz_class n);
```

```
mpz_class factor_p1(mpz_class n);

mpz_class factor_williams(mpz_class n,mpz_class max);

mpz_class factor_raices(mpz_class n_ra,mpz_class referencia);
```

Bibliografía

[1] K. Aoki, Y. Kida, T. Shimoyama, and H. Ueda. Gnfs factoring statistics of rsa-100, 110,..., 150. Technical report, Citeseer.

[2] R.A. Mollin. *Codes: the guide to secrecy from ancient to modern times*. CRC Press, 2005.

[3] C. Shannon. Communication theory of secrecy systems. bell systems techn. *Journal*, 28:656–715, 1949.

[4] W. Diffie and M. Hellman. New directions in cryptography. *IEEE Transactions on information Theory*, 22(6):644–654, 1976.

[5] RL Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.

[6] T. Denny, B. Dodson, A.K. Lenstra, and M.S. Manasse. On the factorization of rsa-120. *Lecture Notes in Computer Science*, 773:166–174, 1994.

[7] HC Williams. A $p+1$ method of factoring. *Mathematics of Computation*, pages 225–234, 1982.

[8] GMP team. *GNU MP, The GNU Multiple Precision Arithmetic Library*, 4.2.4 edition, September 2008.

[9] Peter Giblin. *Primes and programing, An introduction to number theory with computing*. Tercera edition, 1997.

[10] Gerhard Rosenberger Benjamine Fine. *Number theory, An introduction via the distribution of primes*. 2007.

[11] R.E. Crandall and C. Pomerance. *Prime numbers: a computational perspective*. Springer Verlag, 2005.

[12] Paulo Ribenboim. *The little book of the big primes*. 1991.

[13] B. von Sydow, Chalmers tekniska högskola, and Göteborgs universitet. *A machine-assisted proof of the fundamental theorem of arithmetic*. Citeseer, 1992.

[14] M. Bullynck. Modular arithmetic before cf gauss: Systematizations and discussions on remainder problems in 18th-century germany. *Historia Mathematica*, 36(1):48–72, 2009.

[15] T. Jebelean. *Comparing several gcd algorithms*. Citeseer.

[16] E. Cesaratto, J. Clément, B. Daireaux, L. Lhote, V. Maume-Deschamps, and B. Vallée. Regularity of the euclid algorithm; application to the analysis of fast gcd algorithms. *Journal of Symbolic Computation*, 2008.

[17] P. Shankar. Decoding reed-solomon codes using euclids algorithm. *Resonance*, 12(4):37–51, 2007.

[18] R. Kannan, G. Miller, and L. Rudolph. Sublinear parallel algorithm for computing the greatest common divisor of two integers. pages 7–11, 1984.

[19] S.M. Sedjelmaci. A parallel extended gcd algorithm. *Journal of Discrete Algorithms*, 6(3):526–538, 2008.

[20] Douglas R. Stinson. *Cryptography Theory and practice*. 2006.

[21] SJ Aboud. Baghdad method for calculating multiplicative inverse. In *Information Technology: Coding and Computing, 2004. Proceedings. ITCC 2004. International Conference on*, volume 2, 2004.

[22] C.H. Wu, J.H. Hong, and C.W. Wu. Rsa cryptosystem design based on the chinese remainder theorem. pages 391–395, 2001.

[23] K.; Kasahara M. Murakami, Y.; Katayanagi. A new class of cryptosystems based on chinese remainder theorem. pages 1–6, 2008.

[24] K. Katayanagi, Y. Murakami, and M. Kasahara. New product-sum type public-key cryptosystems with selectable encryption key based on chinese remainder theorem. *IEICE Transactions on fundamentals of electronics communications and computer sciences e series A*, 85(2):472–480, 2002.

[25] RGE Pinch. Some primality testing algorithms. *Notices Amer. Math. Soc*, 40:1203–1210, 1993.

[26] M. Dietzfelbinger. *Primality Testing in Polynomial Time: From Randomized Algorithms to'primes is in P'*. Springer Verlag, 2003.

[27] R.P. Brent, Computer Sciences Laboratory, Australian National University, and Centre for Mathematical Analysis. 2parallel algorithms for integer factorisation. 1989.

[28] P. Pritchard. A sublinear additive sieve for finding prime number. *Communications of the ACM*, 24(1):18–23, 1981.

[29] J. Sorenson. *An Analysis of Two Prime Number Sieves*. University of Wisconsin-Madison, Computer Sciences Dept., 1991.

[30] B. Dunten, J. Jones, and J. Sorenson. A space-efficient fast prime number sieve. *Information Processing Letters*, 59(2):79–84, 1996.

[31] S. Hwang, K. Chung, and D. Kim. Load balanced parallel prime number generator with sieve of eratosthenes on cluster computers. In *7th IEEE International Conference on Computer and Information Technology, 2007. CIT 2007*, pages 295–299, 2007.

[32] G. Alkauskas. A curious proof of fermat's little theorem. *American Mathematical Monthly*, 116(4):362–364, 2009.

[33] WR Alford, A. Granville, and C. Pomerance. There are infinitely many carmichael numbers. *Annals of Mathematics*, pages 703–722, 1994.

[34] C. Pomerance, JL Selfridge, and S.S. Wagstaff Jr. The pseudoprimes to 25 109. *MATHEMATICS OF COMPUTATION*, 35(151):1003–1026, 1980.

[35] RGE Pinch. The carmichael numbers up to \$10ˆ{15}\$. *Mathematics of Computation*, 61(203):381–391, 1993.

[36] R.G.E. Pinch. The carmichael numbers up to \$10ˆ{16}\$. *Arxiv preprint math/9803082*, 1998.

[37] R.G.E. Pinch. The carmichael numbers up to \$10ˆ{17}\$. *Arxiv preprint math/0504119*, 2005.

[38] R.G.E. Pinch. The carmichael numbers up to \$10ˆ{18}\$. *Arxiv preprint math.NT/0604376*, 2006.

[39] R.G.E Pinch. The carmichael numbers up to \$10ˆ{21}\$. 2007.

[40] P. Erdös and C. Pomerance. On the number of false witnesses for a composite number. *Mathematics of Computation*, 46(173):259–279, 1986.

[41] G.L. Miller. Riemann's hypothesis and tests for primality. In *Proceedings of seventh annual ACM symposium on Theory of computing*, pages 234–239. ACM New York, NY, USA, 1975.

[42] G.J. Chaitin and J.T. Schwartz. A note on monte carlo primality tests and algorithmic information theory. *Information, randomness & incompleteness: papers on algorithmic information theory*, 31:197, 1990.

[43] D.A. Cox. Quadratic reciprocity: its conjecture and application. *American Mathematical Monthly*, pages 442–448, 1988.

[44] R. Peralta. A simple and fast probabilistic algorithm for computing square roots modulo a prime number (Corresp.). *IEEE Transactions on Information Theory*, 32(6):846–847, 1986.

[45] J. Stein. Computational problems associated with racah algebra. *Journal of Computational Physics*, 1(3):397–405, 1967.

[46] SM Eikenberry and JP Sorenson. Efficient algorithms for computing the jacobi symbol. *Journal of Symbolic Computation*, 26(4):509–523, 1998.

[47] G. Purdy, C. Purdy, and K. Vedantam. Two binary algorithms for calculating the jacobi symbol and a fast systolic implementation in hardware. 1, 2006.

[48] BF Wyman. What is a reciprocity law? *American Mathematical Monthly*, pages 571–586, 1972.

[49] U.M. Maurer. Fast generation of prime numbers and secure public-key cryptographic parameters. *Journal of Cryptology*, 8(3):123–155, 1995.

[50] R.D. Silverman. Fast generation of random, strong rsa primes. *CryptoBytes*, 3(1):9–13, 1997.

[51] D.M. Bressoud. *Factorization and primality testing*. Springer, 1989.

[52] H. Riesel. *Prime numbers and computer methods for factorization*. Springer, 1994.

[53] S. Vaudenay. *A classical introduction to cryptography: applications for communications security*. Springer, 2005.

[54] O. Montoya. Un criterio general de divisibilidad. *http://www.usergioarboleda.edu.co/matematicas/memorias/memorias13/Un%20criterio%20general%20de%20divisibilidad.pdf*.

[55] R.S. Lehman. Factoring large integers. *Mathematics of Computation*, pages 637–646, 1974.

[56] JM Pollard. A monte carlo method for factorization. *BIT Numerical Mathematics*, 15(3):331–334, 1975.

[57] A.K. Lenstra. Integer factoring. *Designs, codes and cryptography*, 19(2):101–128, 2000.

[58] S.D. Miller and R. Venkatesan. Spectral analysis of pollard rho collisions. *Lecture Notes in Computer Science*, 4076:573, 2006.

[59] R.P. Brent. An improved monte carlo factorization algorithm. *BIT Numerical Mathematics*, 20(2):176–184, 1980.

[60] R.E. Crandall. Parallelization of pollard-rho factorization. *preprint*, 23, 1999.

[61] J.M. Pollard. Theorems on factorization and primality testing. 76(03), 1974.

[62] D. Lerch and D. Electrónico. Criptografía de curva elíptica: Ataque de rho de pollard. *hakin9*, 2007.

[63] H.W. Lenstra Jr. Factoring integers with elliptic curves. *Annals of mathematics*, 126(2):649–673, 1987.

[64] R.P. Brent, Australian National University, and Centre for Mathematical Analysis. Some integer factorization algorithms using elliptic curves. 1985.

[65] R.P. Brent. Factorization of the tenth fermat number. *Mathematics of Computation*, 68(225):429–452, 1999.

[66] L. Moser. A theorem on the distribution of primes. *American Mathematical Monthly*, pages 624–625, 1949.

[67] B. Buchberger and T. Jebelean. Parallel rational arithmetic for computer algebra systems: Motivating experiments. pages 93–3, 1993.

[68] T. Jebelean. A generalization of the binary gcd algorithm. pages 111–116, 1993.

[69] D.M. Gordon. A survey of fast exponentiation methods. *Journal of algorithms*, 27(1):129–146, 1998.

[70] B. Moller. Improved techniques for fast exponentiation. *Lecture notes in computer science*, pages 298–312, 2002.

[71] F. Hidrobo and H. Hoeger. Introducción a mpi (message passing interface).

[72] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the mpi message passing interface standard. *Parallel Computing*, 22(6):789–828, 1996.

[73] D. Boneh et al. Twenty years of attacks on the rsa cryptosystem. *NOTICES-AMERICAN MATHEMATICAL SOCIETY*, 46:203–213, 1999.

[74] D. Coppersmith, M. Franklin, J. Patarin, and M. Reiter. Low-exponent rsa with related messages. *Lecture notes in computer science*, 1070:1–9, 1996.

[75] F. Bao, R.H. Deng, YF Han, ABR Jeng, A.D. Narasimhalu, and T.H. Ngair. Breaking public key cryptosystems on tamper resistant devices in the presence of transient faults. *Lecture notes in computer science*, pages 115–124, 1998.

[76] D. Boneh, R.A. DeMillo, and R.J. Lipton. On the importance of checking cryptographic protocols for faults. *Lecture Notes in Computer Science*, 1233:37–51, 1997.

[77] P.L. Montgomery. A survey of modern integer factorization algorithms. *CWI Quarterly*, 7(4):337–365, 1994.

[78] C. Pomerance. Analysis and comparison of some integer factoring algorithms. *Mathematisch Centrum Computational Methods in Number Theory, Pt. 1 p 89-139(SEE N 84-17990 08-67)*, 1982.

[79] HJJ te Riele, W. Lioen, and D. Winter. Factoring with the quadratic sieve on large vector computers. *Journal of Computational and Applied Mathematics*, 27(1):267–278, 1989.

[80] A. Shamir and E. Tromer. Factoring large numbers with the twirl device. 2003.

[81] A. Shamir. Factoring large numbers with the twinkle device. *Lecture notes in computer science*, pages 2–12, 1999.

[82] HW Lenstra. *The development of the number field sieve*. Springer, 1993.

[83] B. Dixon and A.K. Lenstra. Factoring integers using simd sieves. *Lecture Notes in Computer Science*, 765:28–39, 1994.

[84] A.K. Lenstra and M.S. Manasse. Factoring by electronic mail. 89:355–371, 1990.

[85] R.D. Silverman. Exposing the mythical mips year. *IEEE Computer*, 32(8):22–26, 1999.

[86] S. Cavallar, B. Dodson, A. Lenstra, P. Leyland, W. Lioen, P.L. Montgomery, B. Murphy, H. Te Riele, and P. Zimmermann. Factorization of rsa-140 using the number field sieve. *Lecture notes in computer science*, pages 195–207, 1999.

[87] P.L. Montgomery and B. Murphy. Improved polynomial selection for the number field sieve. 1317, 1999.

[88] P.L. Montgomery. Square roots of products of algebraic numbers. *Mathematics of Computation*, 1993:567–571, 1943.

[89] S. Cavallar, B. Dodson, A.K. Lenstra, W. Lioen, P.L. Montgomery, B. Murphy, H. Te Riele, K. Aardal, J. Gilchrist, G. Guillerm, et al. Factorization of a 512-bit rsa modulus. 1807(2000):1–18, 2000.

[90] J.M. Chen, S.I. Yu, Y. Ou-Yang, P.H. Wang, C.H. Lin, P.Y. Huang, B.Y. Yang, and C.S. Laih. Improved factoring of rsa modulus. *http://algo2008.csie.chu.edu.tw/paper/25th*$_p$*aper*/052$_2$5*th*$_a$*lgo*2008.*pdf*.

[91] P.L. Montgomery. A block lanczos algorithm for finding dependencies over gf (2). *Lecture Notes in Computer Science*, 921:106–120, 1995.

[92] Hewlett-Packard. *HP XC System Softwar,e User's Guide, Version 3.0*, 2006.