UNIVERSIDAD DE BUENOS AIRES

Facultad de Ciencias Exactas y Naturales

Departamento de Computación

Factorización de enteros con hipérbolas modulares

Tesis presentada para optar al título de Doctor de la Universidad de Buenos Aires en el área de Ciencias de la Computación

Juan Pablo Di Mauro Aparicio

Director de tesis:	Hugo D. Scolnik
Consejero de Estudios:	Javier Leonardo Marenco
Lugar de trabajo:	Instituto de Ciencias de la Computación - Departamento de Computación

Buenos Aires, 2019

Índice general

A mi esposa y a mi famila.

A Hugo Scolnik.

Al pueblo, por hacer posible la educación pública, libre, gratuita y de calidad.

Tabla de notación

Cuadro 1: Símbolos y notación

$\mathbb{Z}$	El conjunto de enteros.
$\mathbb{Q}$	El conjunto de números racionales.
$\mathbb{Z}_c$	El conjunto de enteros módulo c.
$(\mathbb{Z})^2$	El subconjunto de enteros de la forma y^2 para algún y en $\mathbb{Z}$.
$(\mathbb{Z}_c)^2$	Residuos cuadráticos de $\mathbb{Z}_c$.
$\mathbb{Z}_c^*$	Grupo de las unidades de $\mathbb{Z}_c$.
$\phi(c)$	Cardinal de $\mathbb{Z}_c^*$ (Función de Euler).
$\pi(x)$	Cantidad de primos hasta x.
$mcd(a, b)$	Máximo común divisor entre a y b.
$\lfloor x \rfloor$	Máximo entero menor o igual a x.
$\lceil x \rceil$	Mínimo entero mayor o igual a x.
$a \mid b$	a divide a b.
$\left(\frac{\cdot}{\cdot}\right)$	Símbolo de Legendre.
$a \mod c$	Mínimo entero no negativo congruente a a módulo c
$Im(f)$	La imagen de f.
$\|A\|$	Cardinal del conjunto A.
$exp(x)$	e^x
$Mul(k)$	Cantidad de operaciones elementales que requiere multiplicar dos números de k bits.
$Sqrt(k)$	Cantidad de operaciones elementales que se necesitan para sacar la raíz cuadrada entera de un número de k bits.

Abstract

Motivated by the RSA cryptosystem, we treat here the problem of integer factorization. In the first part a brief survey of some actual algorithms is presented. In the second, we develop the theory needed for a new algorithm of integer factorization.

For a composite n and an odd c with c not dividing n, the number of solutions to the equation $n + a \equiv b \mod c$ with a, b quadratic residues modulus c is calculated. We establish a direct relation with those modular solutions and the distances between points of a modular hyperbola. Furthermore, for certain composite moduli c, an asymptotic formula for quotients between the number of solutions and c is provided. Finally, an algorithm for integer factorization using such solutions and of complexity approximate to $O(n^{1/3})$ is presented.

In the final part, we provide some numerical examples.

Resumen

El problema de la factorización de enteros ha cobrado una importancia capital en las últimas tres décadas, en gran parte debido al uso extendido del criptosistema RSA[20]. La cuestión es teórica y prácticamente relevante. Un ejemplo inmediato del primer caso es la pregunta: ¿quebrar RSA es equivalente a resolver el problema de factorización de enteros en tiempo polinomial?[1], [5].

En lo que respecta a cuestiones prácticas, numerosos algoritmos para factorizar (de aquí en adelante se hablará de factorización de enteros), han sido propuestos en los últimos treinta años (o más). No obstante pasaron más de quince desde el avance significativo más reciente, la invención de la criba del cuerpo de números (NFS, [17]).

Aún así, utilizando NFS en el año 2009, la factorización de un número de 768 bits representativo del problema requirió el esfuerzo conjunto de varias instituciones y una cantidad enorme de tiempo y procesamiento (sólo la elección del polinomio a utilizar en el algoritmo tomó 3 meses en 80 procesadores [16]).

Pueden encontrarse, por una parte, algoritmos determinísticos o estocásticos cuya complejidad no puede ser demostrada rigurosamente. Por otra, algoritmos cuya cota para el tiempo de ejecución es lo suficientemente ajustada y ha sido probada rigurosamente, pero con desempeño peor que los del primer grupo. El algoritmo rho de Pollard [19], la criba cuadrática ([7], sección 6.1) y la criba del cuerpo de números [17] son ejemplos del primer tipo, mientras que el algoritmo de Strassen ([7] sección 5.5) lo es del segundo y hasta el momento, el de menor complejidad en su clase. Sin embargo, ninguno de ellos está acotado polinomialmente y aún no se conoce un algoritmo de factorización en tiempo polinomial en el paradigma computacional clásico [25].

Teniendo en cuenta ese panorama y que muchos algoritmos subexponenciales fueron secuelas de métodos exponenciales, es que se justifica el estudio y desarrollo de nuevos procedimientos para la factorización de enteros, aún cuando sean exponenciales (pero de una complejidad moderada).

Adicionalmente, muchos algoritmos comparten un antecedente común debido a Fermat: la relación entre los factores de un entero n y las soluciones a la ecuación diofántica $n + x^2 = y^2$. Este trabajo retoma esa idea inicial. Está basado en la observación hecha por H. Scolnik de que existen soluciones modulares únicas y fácilmente calculables de $n + x^2 \equiv y^2 \mod c$ para algunos enteros c, lo que posteriormente llevó al mismo autor a definir *targets* para un entero [24]: un target para n es una terna de enteros (a, b, c) con $0 \leq a, b < c$ que satisface $n + a \equiv b \mod c$ y tanto a como b son residuos cuadráticos de c.

A propósito de ello, no es evidente cómo podrían ser de utilidad los targets para factorizar un entero. Más aún, es necesario saber cuántos targets hay

para un c fijo y el crecimiento del número de ellos en relación a c. Como puede verse al principio del capítulo 3, la reducción en la magnitud de las incógnitas que se consigue utilizando targets, se ve contrarrestada por el incremento en la cantidad de targets posibles y por eso es necesario profundizar sobre este tema para conseguir un procedimiento para factorizar, con complejidad aceptable.

El aporte fundamental de este trabajo es, en primer lugar la construcción de una teoría necesaria para targets y su relación con las hipérbolas modulares, estudiadas por Shparlinski [27]. (Como nota histórica, ambos conceptos se habían desarrollado de forma independiente hasta el momento.) Los resultados obtenidos al respecto son de interés en sí mismos y en relación con los desarrollos previos (por ejemplo, puede verse en [22] y en [13] otros usos de las hipérbolas modulares en la factorización de enteros).

Como consecuencia de lo anterior, y segunda contribución, presentamos un algoritmo determinista para factorizar enteros utilizando distancias entre puntos de una hipérbola modular (o targets según la sección). Proponemos también variantes para su implementación y por último, analizamos su complejidad.

Suponiendo que la parte x de la solución a la ecuación de Fermat sea menor en valor absoluto que $n^{1-\epsilon}$, el tiempo de ejecución del algoritmo de factorización con targets es de $O(2^{-m}n^{1-\epsilon}M)$ con M y m dependiendo de $\log n$. En la sección 2.4, teorema 8 puede encontrarse una referencia sintética sobre el algoritmo y su tiempo de ejecución. Por ejemplo, cuando $\epsilon = 0{,}57$ ($|x| < n^{0{,}43}$) para n de 1024 bits, la cantidad elemental de operaciones es alrededor de $O(n^{1/3}M)$.

Aunque el tiempo de ejecución del algoritmo del capítulo 3 es exponencial y similar a otros existentes, su desarrollo aporta un enfoque nuevo al problema y muestra la utilidad de las hipérbolas modulares en un problema computacional concreto.

Los algoritmos fueron programados en Python, usando una librería para enteros grandes. Los resultados de los experimentos numéricos se resumen en la sección 3.4 y como puede verse, el cálculo teórico de la complejidad es ratificado en la ejecución.

Capítulo 1

Preliminares y antecedentes

1.1. Preliminares

1.2. Factorización de enteros y RSA

La dificultad de factorizar enteros en un tiempo *razonable* es hipótesis base del criptosistema RSA y ello puede verse con relativa facilidad. En primer lugar, nótese que existe un método eficiente para un ataque de recuperación de clave privada, suponiendo que se conoce la factorización del módulo n. En efecto, sea (n, e), (n, d) las claves públicas y privadas de una instancia del criptosistema RSA. Supóngase que un atacante $\mathcal{A}$ conoce (n, e) y la factorización de $n = p \cdot q$. Entonces puede calcular fácilmente $\phi(n) = (p-1)(q-1)$. Luego puede utilizar algún algoritmo eficiente (por ejemplo, el algoritmo extendido de Euclides) para hallar el inverso de e módulo $\phi(n)$. Suponiendo que $mcd(e, \phi(n)) = 1$, ese inverso es único y es d.

Más aún, si existiese un algoritmo probabilístico eficiente $\mathcal{F}$ para factorizar enteros, entonces existiría un algoritmo probabilístico eficiente para recuperar la clave de una instancia de RSA. El atacante $\mathcal{A}$, dado (n, e) como entrada procedería de la siguiente forma

1. Utilizar $\mathcal{F}$ para obtener $n = p \cdot q$.

2. Calcular $\phi(n) = (p-1)(q-1)$.

3. Utilizar el algoritmo extendido de Euclides para calcular e^{-1}, el inverso de e módulo $\phi(n)$.

4. Hacer de (n, e^{-1}) la clave privada.

Claramente el procedimiento anterior es eficiente pues $\mathcal{F}$ lo es y las operaciones aritméticas, así como el algoritmo de Euclides también lo son.

Reciprocamente, si se conocen los pares (n, e) y (n, d) entonces la factorización de n puede obtenerse en tiempo polinomial con un algoritmo determinístico (ver [6]). [1]

1.3. Definiciones de complejidad

Antes de abordar la discusión de los primeros algoritmos para factorizar enteros, es necesario precisar la noción de complejidad que se utilizará. La

[1] No obstante, es importante notar lo siguiente: la *hipótesis* RSA consiste en que dado $a \in \mathbb{Z}_n^*$ encontrar $x \in \mathbb{Z}_n$ para el cual $x^e \equiv a \mod N$ es computacionalmente difícil. No está probado que exista una reducción polinomial entre un algoritmo que resuelve el problema anterior a un algoritmo que factorize n ([1], [5])

longitud de un entero n se definirá como

$$\lg n = \begin{cases} 1 & \text{si } n = 0 \\ 1 + \lfloor \log_2(|n|) \rfloor & \text{si } n \neq 0 \end{cases} \tag{1.1}$$

Frecuentemente cuando se asuma que n es un entero positivo, utilizaremos $\log n$ en lugar de $\lg n$, en virtud de que $\log n = C \log_2 n$ para una constante C.

Las operaciones primitivas (de costo constante) son aquellas que operan sobre bits, como la disyunción, conjunción, complemento, etc.

En algunos casos, la complejidad de algunas operaciones aritméticas se dejará indicada, pero sin especificar (en particular la multiplicación): cuando se utilizan enteros grandes, la diferencia entre algoritmos estándar y rápidos para multiplicar (por ejemplo, la transformada rápida de Fourier) es notoria.

Por último se asumirá en nuestro modelo que calcular la raíz cuadrada de un entero n tiene el mismo costo que multiplicar dos enteros de longitud n (una presentación adecuada y más extensa de todo lo anterior puede encontrarse en [4]).

La complejidad de los algoritmos de los que nos ocuparemos necesita ser medida en bits (o alguna noción equivalente). Un error frecuente, originado por el supuesto de que las operaciones aritméticas tienen el mismo costo para operandos de longitud arbitraria, consiste en pensar que factorizar un entero n debe tener complejidad a lo sumo lineal, del orden de $O(n)$ en el peor caso porque es suficiente buscar los divisores de n entre todos los naturales mayores que 2 y menores o iguales que $\sqrt{n}$. Sin embargo la experiencia ofrece una evidencia de lo contrario, por ejemplo, con el intento de obtener por el procedimiento anterior la factorización de un entero del tipo usado en RSA, pero de tamaño muy modesto (una veintena de dígitos).

Sea $\mathcal{A}$ un algoritmo que toma como entrada un entero n. Diremos que $\mathcal{A}$ tiene complejidad (del peor caso):

1. Exponencial con respecto a la longitud de n si la cantidad de operaciones de $\mathcal{A}$ es $O(2^{c \lg n})$ para alguna constante $c > 0$.

2. Subexponencial con respecto a la longitud de n si la cantidad de operaciones de $\mathcal{A}$ es $O(2^{c(\lg n)^{1/\alpha}})$ para $\alpha > 1$, $c > 0$.

3. Polinomial con respecto a la longitud de n si la cantidad de operaciones de $\mathcal{A}$ es $O(f(\lg n))$ para algún polinomio f.

Por ejemplo, buscar un divisor de n probando entre los números mayores a 2 y menores o iguales que $\lceil \sqrt{n} \rceil$ tiene complejidad exponencial en bits (en el peor caso): si n es primo, habrá que probar $O(\sqrt{n}) = O(2^{\frac{\log_2 n}{2}})$ números.

1.4. Antecedentes

Los algoritmos presentados en esta sección han sido largamente estudiados. El resumen que sigue es a favor de la completitud del texto y en algunos casos (como el algoritmo de Fermat y el de Lehman), para destacar aspectos que serán de importancia en el desarrollo del método de factorización que se propone en el capítulo 3. Además de las referencias indicadas en cada caso, [7] y [15] tienen buenas, y a veces breves, exposiciones de estos temas.

1.4.1. Algoritmo de Fermat

Uno de los métodos más antiguos y quizás el origen de los métodos modernos, es el algoritmo de Fermat. Consiste en la sencilla, pero poderosa, observación de que para n impar, las soluciones a la ecuación cuadrática diofántica $n + x^2 = y^2$ permiten deducir los factores de n. En efecto, $n = y^2 - x^2 = (y - x)(y + x)$. El procedimiento es en extremo sencillo:

Algoritmo 1 Algoritmo de Fermat

$y \leftarrow \lceil \sqrt{n} \rceil$.
while $y \leq (n+9)/6$ **do**
  $x \leftarrow \lfloor \sqrt{y^2 - n} \rfloor$.
  if $x^2 = y^2 - n$ **then**
    return $y - x, y + x$.
  end if
  $y \leftarrow y + 1$.
end while
return $1, n$.

Es fácil ver que en el caso en que n sea un número compuesto, el algoritmo anterior devuelve dos factores p, q no triviales de n, de modo que $y = \frac{p+q}{2}$ y $x = \frac{p-q}{2}$.

1.4.2. Método de los multiplicadores, de Lehman

El algoritmo de Fermat puede mejorarse sustancialmente en los casos en los que los factores de n difieran significativamente. La idea es hallar $k = u \cdot v$ con u, v enteros de forma que $|up - vq|$ sea pequeño y hacer algo similar al algoritmo de Fermat reemplazando n por $4kn$ y reduciendo el intervalo de búsqueda.

Por ejemplo, para $n = pq = 596052983 \cdot 59138501 = 35249679931198483$ tenemos $|p - q| \approx n^{0,53}$. Sin embargo, tomando $u = 2 \cdot 19$, $v = 383$ y $k = uv$ vemos que $4kn = 4uvpq = (up + vq)^2 - (up - vq)^2$ y $|up - vq| \approx n^{0,27}$.

Lo anterior equivale a decir que para una de las soluciones (x^*, y^*) no triviales de la ecuación diofántica $4kn + x^2 = y^2$, $|x^*|$ es mucho menor que el valor absoluto del x correspondiente a la solución no trivial de $n + x^2 = y^2$.

El algoritmo 2 devuelve dos factores no triviales de un entero n buscando el multiplicador adecuado, en el caso en que n sea compuesto y la cantidad de iteraciones es $O(n^{1/3})$ [7].

Algoritmo 2 Método de Lehman

1: **for** $k = 2, \ldots, \lceil n^{1/3} \rceil$ **do**
2: **if** $k \mid n$ **then**
3: **return** $k, n/k$ {Busca factores de n no mayores a $\lceil n^{1/3} \rceil$}
4: **end if**
5: **end for**
6: $k \leftarrow 1$
7: **while** $k < \lfloor n^{1/3} \rfloor$ **do**
8: $b \leftarrow 2\lceil (kn)^{1/2} \rceil$
9: $b_{max} \leftarrow \lfloor 2(kn)^{1/2} + n^{1/6}/(4k^{1/2}) \rfloor$
10: **while** $b \leq b_{max}$ **do**
11: **if** $b^2 - 4kn$ es un cuadrado perfecto **then**
12: $a \leftarrow \sqrt{b^2 - 4kn}$
13: $g \leftarrow mcd(a + b, n)$
14: **return** $g, n/g$
15: **end if**
16: $b \leftarrow b + 1$
17: **end while**
18: $k \leftarrow k + 1$
19: **end while**
20: **return** $1, n$

El método de Fermat tiene el mejor desempeño cuando los factores de n están próximos. Informalmente, el procedimiento de los multiplicadores de Lehman permite tratar todas las instancias del problema como si fueran la instancia más favorable para el algoritmo de Fermat. Estas ideas y el algoritmo de Lehman serán particularmente importantes en el capítulo 3.

1.4.3. Método $p - 1$ de Pollard

La idea central del método $p - 1$ de Pollard es la siguiente: consideremos $n = p \cdot q$ y M un entero de manera que $(p - 1) \mid M$ pero $(q - 1) \nmid M$. Sea c un elemento de $\mathbb{Z}_n$ escogido al azar. Entonces, puesto que $(p - 1) \mid M$, por el pequeño teorema de Fermat se tiene que $c^M \equiv 1 \mod p$. Además,

si c mód q tiene orden $q-1$ en $\mathbb{Z}_q$ entonces $c^M \not\equiv 1$ mód q. Es decir que $c^M - 1 \equiv 0$ mód p y $c^M - 1 \not\equiv 0$ mód q o lo que es lo mismo, $p \mid (c^M - 1)$ pero $q \nmid (c^M - 1)$, por lo tanto $1 < mcd(c^M - 1, n) < n$.

Por ejemplo si $n = 11 \cdot 19 = 209$ y tomamos $c = 9$, $M = 100$, tenemos que c^{100} mód $209 = 199 = 11 \cdot 18 + 1$, entonces c^{100} mód $209 - 1 = 199 = 11 \cdot 18$ en consecuencia $gcd(c^{100}$ mód $209 - 1, n) = 11$.

No obstante, la cuestión de cómo conseguir M y c con las propiedades referidas no es trivial.

Una manera de hallar el M apropiado es construir la sucesión $\{M_k\}$ dónde M_k es el mínimo común múltiplo del conjunto $\{1, 2, \ldots, k\}$. Los términos pueden ser construidos recursivamente si se define $M_1 = 1$, $M_2 = 2$ y para cualquier $k > 2$, suponiendo que se conoce M_{k-1} se calcula

$$M_k = \begin{cases} sM_{k-1} & \text{si } k = s^\alpha, \text{ con s primo} \\ M_{k-1} & \text{si no} \end{cases}$$

entonces, hay que establecer una cota $B \in \mathbb{N}$ apropiada y utilizar $M = M_B$. Notemos además que será necesario calcular la exponenciación modular $c^{M_B} - 1$ mód n cuya cantidad de operaciones es de orden $O(\log M_B Mul(\log n))$. Por otra parte si B es muy pequeño, es más difícil que $p - 1$ divida a M_B.

Finalmente, para hallar c es importante notar que $\mathbb{Z}_q^*$ es un grupo cíclico de orden $q - 1$ (porque q es primo) y además, si g genera a $\mathbb{Z}_q^*$, cualquier g^k con k coprimo con $q - 1$ genera al grupo (ello es fácil de ver si se considera que $\mathbb{Z}_q^* = \{g^1, g^2, \ldots, g^{q-1}\}$ y que para cualquier $1 < k < q - 1$ coprimo con $q - 1$, el orden de g^k es exactamente $q - 1$). Es decir que si se escoge c al azar, suponiendo que todos los elementos tienen la misma probabilidad de ser elegidos, la probabilidad de que c sea generador del grupo es $p_g = \phi(q-1)/(q-1)$.

La cantidad t de intentos de escoger c hasta obtener un generador tiene entonces distribución geométrica con parámetro $p_g = \phi(q-1)/(q-1)$. El valor esperado de t es por lo tanto $1/p_g = (q-1)/\phi(q-1)$. Por el Teorema B.15 en [15] se tiene que $(q-1)/\phi(q-1) < 2\lceil \log_2 (q-1) \rceil$. Se puede estimar la cantidad de operaciones elementales *para cada* M_B dado, como la cantidad de intentos hasta encontrar un generador c de $\mathbb{Z}_q^*$ por lo que requiere la exponenciación modular c^{M_B} mód n. Eso es $O(2\lceil \log_2 (q-1) \rceil \log M_B Mul(\log n))$. Si $q \approx \sqrt{n}$ la cota anterior para cada M_B dado es aproximadamente $O(\log (n) \log (M_B) Mul(\log n))$

1.4.4. Método rho, de Pollard

Sea $f : \mathbb{Z}_n \to \mathbb{Z}_n$ una función que además cumple $f(x) \equiv f(x')$ mód p siempre que $x \equiv x'$ mód p. Puesto que $\mid \mathbb{Z}_p \mid < \infty$, la sucesión $\{f^k(x)\}_k$ para

cualquier $x \in \mathbb{Z}_p$ necesariamente debe ser periódica. Más aún, pueden darse dos casos:

$$f^i(x) \equiv f^j(x) \mod p \text{ y } f^i(x) \equiv f^j(x) \mod n$$

o

$$f^i(x) \equiv f^j(x) \mod p \text{ y } f^i(x) \not\equiv f^j(x) \mod n$$

El último es el que nos interesa, pues en ese caso $mcd(f^i(x) - f^j(n), n)$ es un factor no trivial de n.

Por ejemplo, si tomamos la sucesión definida inductivamente como

$$x_0 = 101$$

$$x_i = x_{i-1}^2 + 1 \mod 113$$

tenemos los términos

$$x_0 = 101, x_1 = 32, x_2 = 8, x_3 = 65, x_4 = 45, x_5 = 105, x_6 = 65, x_7 = 45$$

En efecto, aunque los primeros cinco términos parecen escogidos al azar, el sexto término es uno que ya teníamos y lo mismo el séptimo. Concretamente, esa sucesión tiene una parte periódica y una aperiódica. Notemos que $x_3 = x_6 = \ldots = x_{3k}$ y para cualquier $0 \leq t \leq 2$

$$x_{3k+t} = x_{3(k+1)+t} = \ldots$$

para $k \geq 0$. Si ahora observamos la misma sucesión pero módulo $113 \cdot 7 = 791$, los primeros seis términos son

$$y_0 = 101, y_1 = 710, y_2 = 234, y_3 = 178, y_4 = 45, y_5 = 444, y_6 = 178$$

Pero como la expresión para la sucesión es la misma y el módulo es un múltiplo de 113, necesariamente

$$y_{3k+t} \equiv y_{3(k+1)+t} \equiv \ldots \mod 113.$$

Más aún, $mcd(y_5 - y_2, 791) = 7$ es un factor de 791.

No obstante, hay varios aspectos que merecen ser analizados. En primer lugar, ¿cuántos elementos de la sucesión necesitamos calcular? Mas importante aún ¿cuántos pares $(f^i(x), f^j(x))$ necesitamos probar para encontrar un divisor no trivial?

La (mal llamada) paradoja del cumpleaños nos dice que encontrar una colisión en un conjunto de ℓ elementos aleatorios, distribuidos uniformemente tiene probabilidad mayor a 0,5 cuando se han examinado alrededor de $\sqrt{\ell}$

Cuadro 1.1: Seudocódigo del algoritmo Rho de Pollard.

1. Inicializar $a \xleftarrow{R} \in \{1, \ldots, n-3\}$, $s \xleftarrow{R} \in \mathbb{Z}_n$, $u \leftarrow s$, $v \leftarrow s$. Sea $f(x) = x^2 + a \mod n$ una función.
2. Calcular $u \leftarrow f(u)$, $v \leftarrow f(f(v))$.
3. Sea $g = mcd(u - v, n)$. Si $g = 1$ volver a 2. Si $g = n$ volver a 1.
4. Devolver g.

elementos. Hagamos $m = \lceil \sqrt{\ell} \rceil = \lceil \sqrt{\sqrt{n}} \rceil$. Entonces la probabilidad de encontrar una colisión en el conjunto $\{f(x), f^2(x), \ldots, f^m(x)\}$ es mayor a 0,5. Lo que nos interesa no es sólo un par $1 < i, j < \ell$ de índices en la lista $\{f(x), f^2(x), \ldots, f^\ell(x)\}$ de manera que $f^i(x) \equiv f^j(x) \mod n$ sino que además, $1 < gcd(f^i(x) - f^j(x), n) < n$. Ciertamente buscar entre los pares $(i, j) \in \{1, \ldots, m\} \times \{1, \ldots, m\}$ nos llevaría $O(m^2) = O(\sqrt{n})$ operaciones, casi lo mismo que probar todos los naturales menores que $\sqrt{n}$ para encontrar un divisor. Una observación clave es que no necesitamos buscar pares sino sólo la longitud del periodo o un múltiplo de la longitud del periodo.

En efecto, si hay una colisión en $\{f(x), f^2(x), \ldots, f^\ell(x)\}$, entonces hay un $1 < c < \ell$ de modo que $f^i(x) = f^{i+c}$ para todo $c \leq i \leq 2c$, en particular para $i = c$. Es decir, sólo debemos buscar c de forma que $1 < gcd(f^c(x) - f^{2c}(x), n) < n$.

Por el análisis hecho antes, sabemos que el algoritmo ejecuta una cantidad esperada de $O(\sqrt{\sqrt{n}})$ veces el paso 2. Calcular el *mcd* del paso 2 tiene complejidad $O(\log(n)^2)$ en el peor caso, pero con alguna precaución puede reducirse a una multiplicación módulo n, de modo que la cantidad de operaciones estimadas (y esperadas) es del orden de $O(n^{1/4} \log(n)^2)$.

1.4.5. Algoritmo Hide and Seek de Rubinstein

Aunque este algoritmo no es muy popular, las ideas geométricas que desarrolla son de interés en este trabajo. La exposición completa puede encontrarse en [22], así como ideas para conseguir un tiempo de ejecución subexponencial.

Dado un entero compuesto $n = pq$ con p, q primos y cualquier $a \in \mathbb{Z}^+$

coprimo con n deben existir $x, y \in \mathbb{Z}_a$ de modo que

$$n \equiv xy \mod a \tag{1.2}$$

$$p \equiv x \mod a, \qquad q \equiv y \mod a \tag{1.3}$$

De forma que puede escribirse

$$n = (ua + x)(va + y)$$

con u, v enteros.

Si ahora el módulo es $a - \delta$ y reescribimos la ecuación anterior, tenemos

$$n = (u(a - \delta + \delta) + x)(v(a - \delta + \delta) + y) \tag{1.4}$$

$$\equiv (u\delta + x)(v\delta + y) \mod a - \delta \tag{1.5}$$

Supóngase que para $(x, y) \in \mathbb{Z}_a \times \mathbb{Z}_a$ se cumple 1.3 y para $(z, w) \in \mathbb{Z}_{a+\delta} \times \mathbb{Z}_{a+\delta}$ se tiene

$$p \equiv z \mod a - \delta, \quad q \equiv w \mod a - \delta \tag{1.6}$$

por 1.4 y 1.5 se sigue que

$$z = u\delta + x, \quad w = v\delta + y \tag{1.7}$$

y finalmente

$$p = \tfrac{(z-x)a}{\delta} + x \quad \text{y} \quad q = \tfrac{(w-y)a}{\delta} + y. \tag{1.8}$$

Nótese que la cantidad de $u_0, v_0 \in \mathbb{Z}_a$ que satisfacen 1.2 es $\phi(a)$ puesto que tomando un par $u_0, v_0 \in \mathbb{Z}_a$ que cumpla $n \equiv u_0 v_0 \mod a$, podemos definir el conjunto de pares que cumplen esa condición como

$$\{(u, v) \mid u = ku_0,\ v = k^{-1}v_0 \text{ para algún } k \in \mathbb{Z}_a^*\}.$$

(ese conjunto es la hipérbola modular para n módulo c).

Como sugiere lo anterior, para cada par (x, y) en $\mathbb{Z}_a \times \mathbb{Z}_a$ se podría probar con los pares (z, w) en $\mathbb{Z}_{a-\delta} \times \mathbb{Z}_{a-\delta}$ hasta que $\frac{(z-x)a}{\delta} + x$ sea un factor propio de n.

Como veremos, para un (x, y), los (z, w) que pueden dar los factores no triviales están a una cierta distancia de (x, y) y eso reduce la cantidad de pares a probar.

Tomaremos $\delta = 1$ y por simplicidad vamos a asumir que $q < p$ y $p \leq 2q$ (con un poco más de trabajo pueden relajarse estas hipótesis). Por lo tanto $p < \sqrt{2n}$ (o sea $p^{2/3} < (2n)^{1/3}$). Por la condición 1.2,

$$u_1 < p^{1/3} \approx a^{1/2} \quad \text{y} \quad v_1 < q^{1/3} < p^{1/3} \approx a^{1/2} \tag{1.9}$$

Lo anterior nos da una idea sobre dónde buscar los pares que necesitamos. Los puntos $(x, y) \in \mathbb{Z}_a \times \mathbb{Z}_a$ y $(z, w) \in \mathbb{Z}_{a+\delta} \times \mathbb{Z}_{a+\delta}$ pueden agruparse en cuadrados con lados de longitud $h = \lceil a^{1/2} \rceil$. Concretamente, se considera el rectángulo

$$R = \{(x', y') \mid 0 \leq x' \leq a, 0 \leq y' \leq a\}$$

y en él una grilla de $h \times h$ cuadrados B_{ij}, cada uno de $h \times h$ unidades:

$$R = \bigcup_{0 \leq i < h, 0 \leq j < h} B_{ij}$$

Para un B_{ij}, los cuadrados adyacentes a él serán los que están una posición arriba o una posición a la derecha o ambas, módulo h (para tener en cuenta la adyacencia de las puntas, de modo que la grilla forma en realidad un toro).

Por las desigualdades en 1.9 para un par (x, y), el punto (z, w) que buscamos estará en la región

$$B = \{(x', y') \mid x \leq x' \leq x + h, y \leq y' \leq y + h\}.$$

Eso significa que si (x, y) está en B_{ij} entonces (z, w) deberá estar en B_{ij} o en alguno de los cuadrados adyacentes.

El algoritmo es bastante simple y se detalla en el Cuadro 1.2.

Puede verse en [22] que la cantidad de pares (x, y) que satisfacen 1.2, en un rectángulo $S \subset R$ guardan relación con el área del rectángulo sin importar su ubicación, salvo un error pequeño. Eso implica que todos los B_{ij} contienen aproximadamente la misma cantidad de puntos.

En consecuencia, si $c_s(a)$ es la cantidad de pares (x, y) que satisfacen 1.2 y $c_s(a-1)$ es la cantidad de pares (z, w) que satisfacen $n \equiv zw \mod a-1$, el algoritmo debe examinar $c_s(a)c_s(a-1)$ pares. Puede probarse (ver [22]) que $O(c_s(a)c_s(a-1)) = O(c_s(a)^2 + c_s(a-1)^2)$ y más aún que $O(c_s(a)^2) = O(a^{1+\epsilon})$ por lo tanto, el tiempo de ejecución es del orden de $O(a^{1+\epsilon}) = O(n^{1/3+\epsilon})$ con las hipótesis que tomamos.

1.5. Algoritmos subexponenciales

1.5.1. Criba cuadrática

Este algoritmo es el primer algoritmo subexponencial que trataremos. Se basa en dos ideas simples. La primera de ella tiene que ver con los cuadrados congruentes módulo n y sus antecedentes pueden rastrearse hasta el algoritmo de Dixon [8] y las ideas de Kraitchik.

Cuadro 1.2: Seudocódigo del algoritmo Hide and Seek de Rubinstein
Entrada: $n \in \mathbb{Z}$, $n > 1$

1. Calcular $a \leftarrow \lceil (2n)^{1/3} \rceil$.
2. Si $1 < mcd(a, n) < n$ devolver $(mcd(a, n), n/mcd(a, n))$ y salir. Si $1 < mcd(a - 1, n) < n$, devolver $(mcd(a, n), n/mcd(a, n))$ y salir.
3. Calcular todos los puntos (x, y) con $n \equiv xy \ mod \ a$ y guardarlos en el conjunto H_a.
4. Inicializar $B_{ij} \leftarrow \emptyset$ para $0 \leq i, j < \lceil \sqrt{a} \rceil$.
5. Para cada elemento (x, y) de H_a
   a) Hacer $i \leftarrow \lfloor x/\lceil \sqrt{a} \rceil \rfloor$, $j \leftarrow \lfloor y/\lceil \sqrt{a} \rceil \rfloor$.
   b) (Asignar el punto a B_{ij}), $B_{ij} \leftarrow B_{ij} \cup \{(x, y)\}$
6. Para cada elemento (x_1, y_1) con $n \equiv x_1 y_1 \ mod \ (a - 1)$
   a) Hacer $s \leftarrow \lfloor x_1/\lceil \sqrt{a} \rceil \rfloor$, $t \leftarrow \lfloor y_1/\lceil \sqrt{a} \rceil \rfloor$.
   b) Para cada B adyacente a B_{st}:
      1) para cada (x, y) en B, sea $p \leftarrow ((x_1 - x)a + x)$. Si $1 < mcd(p, n) < n$ devolver $(p, n/p)$ y salir.
7. Devolver $(1, n)$ (n es primo).

Para n compuesto, la idea es conseguir cuadrados congruentes x^2, y^2 módulo n tales que $x \not\equiv \pm y \mod n$ y luego un factor de n mediante $mcd(x-y,n)$ o $mcd(x+y,n)$.

La pregunta es ¿cómo conseguimos los cuadrados congruentes? Claramente necesitamos un método mejor que la generación de enteros al azar.

Supongamos que $\mathcal{B}$ es un conjunto de primos $p_1, \ldots, p_r$. y que $a_1, a_2, \ldots, a_k$ son enteros cuyos factores primos están en $\mathcal{B}$. Entonces basta agrupar algunos a_i según sus partes "no cuadradas".

Vectores de exponentes

Concretamente, definimos al vector $v(x) \in \mathbb{Z}_0^+$ para x entero como $v(x) = (v_i(x))$ con $v_i(x) = e_i$, el máximo entero k de manera que p_i^k divida a x. Por ejemplo, si $\mathcal{B} = \{2,3,5,7\}$ se tiene

$$v(588) = v(2^2 \cdot 3 \cdot 7^2) = (2,1,0,2)$$

Notemos que si x tiene todos sus factores en $\mathcal{B}$, entonces x es un cuadrado si y sólo si $v(x)$ tiene todas sus componentes pares, o sea si y sólo si $v(x)$ mód 2 es el vector nulo. Además, $v(xy) = v(x) + v(y)$ para cualquier par de enteros x, y.

Cuando un número x tiene todos sus factores primos menores que una cota B, se dice que x es B-smooth o simplemente smooth si el límite está sobreentendido. De modo similar, para una base de factores $\mathcal{B}$, se dice que x es smooth sobre $\mathcal{B}$ (o simplemente smooth) si x se factoriza completamente con los elementos de $\mathcal{B}$.

La criba

Lo anterior nos dice que, definida una base de factores $\mathcal{B}$, podemos juntar un conjunto de elementos $a_1, a_2 \ldots, a_m$ smooth sobre esa base; el conjunto de sus vectores de exponentes módulo 2, $L = \{v(a_1) \mod 2, v(a_2) \mod 2 \ldots, v(a_m) \mod 2\}$ pertenece al espacio vectorial de dimensión $|\mathcal{B}|$ sobre el cuerpo $\mathbb{Z}_2$. Si hay más de $|\mathcal{B}|$ vectores en L entonces el conjunto no puede ser linealmente independiente, es decir que existe $\emptyset \neq L' \subset L$ de modo que

$$\sum_{v \in L'} v \equiv (0,0,\ldots,0) \text{ en } \mathbb{Z}_2$$

Hay dos cuestiones que resolver. La primera de ellas es cómo encontrar una cantidad grande de números smooth, sin tener que factorizar cada vez. La segunda de ellas es que si formar un cuadrado es costoso, sería bueno que

al menos de un lado de la congruencia siempre haya un cuadrado, es decir que los números smooth que buscamos sean siempre cuadrados módulo n.

La segunda de esas preguntas tiene una respuesta bastante simple y nos ayudará a responder la primera. El polinomio $F(x) = x^2 - n$ es siempre un cuadrado módulo n: podría utilizarse F como fuente para generar enteros probando valores $F(x_1), \ldots, F(x_k)$ hasta encontrar una cantidad lo suficientemente grande de smooths $F(x_{i_1}), \ldots, F(x_{i_k})$ de modo que puedan combinarse y formar un cuadrado en $\mathbb{Z}$. Por construcción $F(x_{i_j})$ es siempre un cuadrado módulo n (por lo tanto cualquier producto de los $F(x_{i_j})$ es un cuadrado módulo n). De modo que se habrá conseguido un par x, y con $x^2 \equiv y^2$ mód n.

El problema es que $F(x)$ no es smooth en general. Sin embargo, si tomamos x cerca de $\lceil \sqrt{n} \rfloor$, los valores de $F(x)$ serán pequeños y sería de esperar que sus factores también sean pequeños de forma que el mayor de ellos no sobrepase al mayor primo en $\mathcal{B}$.

Para responder a la primer pregunta será necesario introducir una idea ingeniosa que marca la diferencia entre la criba cuadrática y los algoritmos precedentes.

Hagamos $\ell_i \leftarrow F(x_0 + i)$ para algún entero x_0 escogido. Supóngase que se tiene una lista

$$\ell_0, \ell_1, \ldots \ell_{m-1}$$

el procedimiento ingenuo para encontrar smooths en esa lista sería el siguiente:

1. Tomar p, el primer elemento de $\mathcal{B}$.

2. Recorrer la lista dividiendo por la máxima potencia de p que se pueda en cada caso.

3. Si p no es el último elemento de $\mathcal{B}$, tomar como p al próximo elemento de $\mathcal{B}$ y volver al paso anterior.

4. Si p es el último elemento de $\mathcal{B}$, tomar como conjunto de números smooth a
$$\{F(x_0 + i)\}$$
para todos los i con $\ell_i = 1$.

La cantidad de iteraciones para el procedimiento anterior es $m|\mathcal{B}| = mr$.

No obstante, podemos hacerlo mejor. Para eso hay que notar que si $p \mid F(x)$ entonces $p \mid F(x + kp)$ para cualquier k entero. Por lo tanto para cada $p \in \mathcal{B}$ podemos restringir nuestra atención a un intervalo pequeño, digamos

$[x_0, x_0 + p - 1]$ y buscar allí un x para el cual $F(x) \equiv 0 \mod p$. Además, ese x existe si y sólo si n es un cuadrado módulo p.

En ese caso, podemos recorrer todos los elementos de la forma $F(x_0 + i + kp)$ para k entero y dividir por una potencia de p en cada paso:

1. Eliminar de $\mathcal{B}$ todos los p para los cuales n no sea un cuadrado módulo p (puede hacerse en tiempo $O(\log n \log p)$ ver [4], Teorema 5.9.3).

2. Tomar p, el primer elemento de $\mathcal{B}$.

3. Buscar i en $[0, p-1]$ de forma que $F(x_0 + i) \equiv 0 \mod p$.

4. Para $k = 1, 2, \ldots$ mientras que $i + kp$ sea menor o igual al tamaño de la lista, dividir ℓ_{i+kp} por la máxima potencia de p que se pueda, en cada caso.

5. Si p no es el último elemento de $\mathcal{B}$, tomar como p al próximo elemento de $\mathcal{B}$ y volver al paso anterior.

6. Si p es el último elemento de $\mathcal{B}$, tomar como conjunto de números smooth a
$$\{F(x_0 + i)\}$$
para todos los i de manera que $\ell_i = 1$.

Con algunas mejoras más, un algoritmo similar para encontrar smooths puede funcionar en tiempo $O(m \log \log B)$ con B la mejor cota para la base de factores.

Análisis

El análisis de complejidad es heurístico: hace uso de algunas afirmaciones que hasta ahora no han sido probadas y permanecen como conjeturas (por ejemplo que la probabilidad de ser smooth en un subconjunto especial de los enteros es la misma que en un intervalo de enteros acotados, ver [7] sección 6.1.1). No obstante, ello no lo hace menos valioso y su tiempo de ejecución esperado ha sido confirmado en todos los casos reales.

Una cuestión importante es qué tamaño debe tener la lista en la que buscamos los smooths. Y, relacionado con esto ¿cuán grande debe ser la base de factores?

Puede verse ([7]) que si B es una cota para la base de factores (todo elemento de $\mathcal{B}$ es menor o igual que B), entonces la probabilidad estimada de que un número elegido al azar sea B smooth es u^{-u} con $u = \frac{\log n}{2 \log B}$.

La cantidad esperada de elementos de la lista que debemos recorrer hasta encontrar un smooth es, por lo tanto u^u.

El teorema de la cantidad de números primos ([14] capítulo 2, sección 4) nos dice que la cantidad de primos menores que B es, aproximadamente $B/\log B$. Vamos a suponer que nuestra base de factores está compuesta por una cantidad $K = \frac{B}{2\log B}$ de primos, pues de todos los primos menores que B, aproximadamente la mitad no tienen a n como cuadrado. Entonces $|\mathcal{B}| \approx K$.

Los vectores de exponentes estarán en el espacio $\mathbb{Z}_2^K$ y necesitamos al menos $K+1$ vectores para tener seguridad de encontrar un subconjunto de ellos linealmente dependientes.

La cantidad esperada de elementos a analizar hasta conseguir un número suficiente de smooths es $(K+1)u^u$ y ese es el tamaño adecuado para la lista de enteros.

Por lo tanto, el procedimiento bosquejado antes para encontrar smooths requiere

$$T(B) = m \log\log B = (K+1)u^u \log\log B$$

(nótese que u y K son expresiones dependientes sólo de B)

Sólo falta hallar el valor de B para que la expresión anterior sea mínima. En [7], puede verse una manera de estimar ese valor y el resultado es

$$B = exp\Big(\frac{1}{2}\sqrt{\log n \log\log n}\Big)$$

.

Reemplazando B en la expresión de T se tiene

$$T(B) \approx B^2 = exp(\sqrt{\log n \log\log n})$$

1.5.2. Criba del cuerpo de números

Este método es, hasta el momento, el algoritmo de propósito general con complejidad más baja. Como veremos, la cantidad de operaciones requeridas es del orden de $exp((\log n)^{1/3})$ y aunque no es polinomial, sí es subexponencial.

El algoritmo es complejo y la descripción que daremos aquí del algoritmo será un tanto sucinta para favorecer la claridad de la exposición. Una lectura adecuada y completa para este tema es [17].

Preliminares

La idea principal consiste en buscar, como en la criba cuadrática, cuadrados congruentes módulo n. La diferencia es que para ello utilizaremos una

extensión algebraica de $\mathbb{Q}$. Sea d un entero positivo y $m \approx n^{1/d}$. Consideremos el desarrollo en base m de n, es decir

$$n = f_0 + f_1 m + \ldots + f_d m^d$$

con $0 \leq f_i \leq m-1$ (y $f_d \neq 0$). Sea f el polinomio cuyos coeficientes son los f_i anteriores

$$f(X) = f_0 + f_1 X + \ldots + f_d X^d$$

y α una raíz de f. La primer observación es que puede asumirse que f es irreducible en $\mathbb{Z}[X]$. Si no lo fuese, digamos $f(X) = g(X)h(X)$, entonces $n = f(m) = g(m)h(m)$ y habríamos factorizado n. Lo anterior implica que $\alpha \notin \mathbb{Q}$, por lo tanto el cuerpo $K = \mathbb{Q}(\alpha)$ que resulta de agregar α a $\mathbb{Q}$ es una extensión no trivial de $\mathbb{Q}$, es decir un espacio vectorial sobre $\mathbb{Q}$ de dimensión mayor a uno. Eso significa que existe una base $\{u_1, u_2, \ldots, u_n\}$ de modo que todo elemento de $K = \mathbb{Q}(\alpha)$ es de la forma $\sum_{i=0}^{n} a_i u_i$ para algunos $a_i \in \mathbb{Q}$. Un ejemplo clásico es $K = \mathbb{Q}(i)$, el cuerpo que resulta de agregar una raíz de $X^2 + 1$ a los racionales. Su base es $\{1, i\}$ y todo elemento de K tiene la forma $a + bi$ con $a, b \in \mathbb{Q}$.

Más aún, para cualquier elemento x de K podemos definir la matriz de multiplicación por x. Sólo basta ver cuál es el efecto de x por cada elemento de la base. Por ejemplo, si $x = 1 + i$ entonces $x \cdot 1 = 1 + i$ y $x \cdot i = -1 + i$ es decir que la matriz de multiplicación por x es

$$m(x) = \begin{pmatrix} 1 & -1 \\ 1 & 1 \end{pmatrix}$$

Podemos definir una aplicación desde K hacia $\mathbb{Q}$ observando que, puesto que todas las entradas de la matriz de multiplicación son racionales, el determinante también será un racional. La *norma* de un elemento x de K es el determinante de la matriz de multiplicación por ese elemento. Formalmente, la norma de x es $N(x) = det(m(x))$ (pueden consultarse [3] y algunas partes de [2], para un tratamiento detallado de los cuerpos de números, normas, etc.).

Para el ejemplo anterior, tenemos que $N(1+i) = 1^2 + 1^2 = 2$. En general, la norma de cualquier elemento $a + bi$ en $K = \mathbb{Q}(i)$ será $N(a + bi) = a^2 + b^2$ (que es exactamente el módulo al cuadrado del número complejo). Por último, es claro que la norma es multiplicativa. Esto ultimo implica una propiedad fundamental para el algoritmo: si $x = a + b\alpha \in \mathbb{Q}(\alpha)$ con $a, b \in \mathbb{Z}$ y x es un cuadrado en $\mathbb{Q}(\alpha)$ entonces $N(x)$ es un cuadrado en $\mathbb{Z}$ (lo recíproco no es cierto en general).

Un ejemplo menos simple es $K = \mathbb{Q}(\sqrt{-5})$ cuya base es $1, \sqrt{-5}$. La matriz de multiplicación de un elemento $x = a + b\sqrt{-5} \in K$ es

$$m(x) = \begin{pmatrix} a & -5b \\ b & a \end{pmatrix}$$

y su norma es $N(a + b\sqrt{-5}) = a^2 + 5b^2$.

Una definición equivalente de la norma de un elemento $x = a_0 + a_1\alpha + a_2\alpha^2 + \ldots \in \mathbb{Q}(\alpha)$ es la siguiente: Si $\alpha = \alpha_1, \alpha_2, \ldots, \alpha_m$ son los conjugados de α (es decir, las otras raíces del polinomio minimal de α con coeficientes en $\mathbb{Q}$, entonces la norma de x es el producto de todos los elementos x_i que resultan de reemplazar α por α_i en la expresión de x, eso es

$$N(x) = (a_0 + a_1\alpha_1 + \ldots)(a_0 + a_1\alpha_2 + \ldots)\ldots(a_0 + a_1\alpha_m + \ldots)$$

Notemos que existe un homomorfismo de anillos $\phi : \mathbb{Z}[\alpha] \to \mathbb{Z}_n$ definido por llevar α a m y reducir módulo n. La idea central del algoritmo es conseguir un producto $\theta_1 \ldots \theta_r = \gamma^2$ en $\mathbb{Z}[\alpha]$ y por otra parte, un entero v (que podría obtenerse de alguna forma de ese producto) tales que $\phi(\gamma^2) = u^2 \equiv v^2 \mod n$ y entonces tratar de factorizar n mediante $mcd(u - v, n)$.

Criba y vectores de exponentes

Ahora es necesario hablar sobre cómo conseguir los cuadrados congruentes. En lo que sigue el desarrollo estará centrado en el anillo $\mathbb{Z}[\alpha]$ y nos interesaran los elementos de la forma $a - b\alpha$ de ese anillo. El objetivo es conseguir un conjunto $\mathcal{C}$ de pares (a, b) de modo que

$$\prod_{(a,b)\in\mathcal{C}} (a - b\alpha) = \gamma^2$$

para algún $\gamma \in \mathbb{Z}[\alpha]$ y simultáneamente

$$\prod_{(a,b)\in\mathcal{C}} (a - bm) = v^2$$

para algún entero v.

La forma de hacerlo es similar a la usada en la criba cuadrática, juntar una cantidad grande de elementos que sean *smooth* para alguna base de factores $\mathcal{B}$ determinada, construir vectores de exponentes y combinar los elementos en productos para que sus vectores de exponentes den un vector con elementos pares. Diremos que un elemento $a - b\alpha$ es smooth si su norma es smooth.

Pero, de lo anterior surgen muchas preguntas. En primer lugar, para juntar elementos smooth necesitamos cribar pero, ¿cribar qué?. En segundo lugar ¿cómo tienen que ser los vectores de exponentes?

La primer pregunta tiene una respuesta simple. La norma de un elemento $x - \alpha y$ en $\mathbb{Z}[\alpha]$ puede ser considerada como un polinomio de coeficientes enteros, sobre las variables x e y. En efecto, la norma es el determinante de la matriz de multiplicación. En la matriz de multiplicación sólo aparecen enteros y las variables x e y; finalmente el determinante hace productos y sumas de esos elementos. Llamemos $F(a, b) = N(a - b\alpha)$ a ese polinomio. Entonces podemos utilizar el procedimiento usual de criba sobre $F(a, b)$: se deja fija a y se criba sobre $F(a, y)$, un polinomio sobre las variables y y luego lo mismo dejando fija b. Para buscar elementos de la forma $a - bm$ smooths se criba de la misma forma sobre el polinomio de grado 1, $G(a, b) = a - bm$. Finalmente para buscar pares a, b tal que $a - b\alpha$ y $a - bm$ sean smooths, se criba sobre el polinomio $H(a, b) = F(a, b)G(a, b)$.

La cuestión de los vectores de exponentes es algo más compleja. En los enteros cada elemento se factoriza en primos de un único modo y ello permite la correcta definición del vector de exponentes. A diferencia de los enteros, el anillo $\mathbb{Z}[\alpha]$ puede no ser un dominio de factorización única. Sin embargo, sabemos que $\mathbb{Z}[\alpha]$ está contenido en un dominio de Dedekind y allí, los ideales no nulos se factorizan en forma única como producto de ideales primos.

Ahora observemos lo siguiente:

Lema 1. *Para un primo p sea $R(p)$ el conjunto de todos los elementos r de $\mathbb{Z}_p$ que son raíces de $f(x)$ módulo p. Si a, b son enteros coprimos $F(a, b) \equiv 0$ mód p sí y sólo si $a - br \equiv 0 \mod p$ para algún $r \in R(p)$.*

Demostración. Como $F(x, y) = N(x - y\alpha) = (x - \alpha_1 y)(x - \alpha_2 y) \dots (x - \alpha_d y)$, tenemos que

$$F(x, y) = y^d(x/y - \alpha_1)(x/y - \alpha_2) \dots (x/y - \alpha_d) = y^d f(x/y).$$

Si $a \equiv br \mod p$ entonces $F(a, b) \equiv F(br, b) = b^d f(r) \mod p$. Pero $f(r) \equiv 0 \mod p$ por que $r \in R(p)$.

Si $p \mid F(a, b)$ entonces $p \mid b$ o $p \mid f(a/b)$. Pero $p \mid b$ implica que $p \mid a$ por que $F(a, b)$ puede escribirse como

$$F(a, b) = f_0 a^m + f_1 a^{m-1} b + \ldots + f_{m-1} a b^{m-1} + f_m b^m$$

para algunos enteros $f_1, f_2, \ldots, f_m$. Como a y b son coprimos, lo anterior es imposible por lo tanto, debe ser que $p \mid f(a/b)$ y entonces $a/b \equiv r \mod p$ con $r \in R(p)$. □

Queremos construir vectores de exponentes de manera que cuando las sumas, módulo 2 den como resultado el vector nulo, podamos asegurar que el producto de los elementos correspondientes a esos vectores forman un cuadrado.

Para una base de primos $\mathcal{B} = \{p_1, \ldots, p_m\}$ vamos a construir un vector de exponentes con tantas componentes como pares (p_i, r) con $r \in R(p_i)$ y los definimos como:

$$v(a - b\alpha) = (v_{p_i,r}) \text{ donde } v_{p_i,r} = \begin{cases} 0 & \text{si } a - br \not\equiv 0 \mod p_i \\ v_{p_i}(F(a,b)) & \text{si } a - br \equiv 0 \mod p_i \end{cases}$$

y $v_{p_i}(x)$ es el máximo exponente de p_i que divide a x.

Un resultado importante en este sentido es el siguiente lema, cuya demostración omitiremos (puede consultarse [7], Lema 6.2.1).

Lema 2. *Si $\mathcal{C}$ es un conjunto de pares a, b coprimos, cada $a - b\alpha$ es smooth sobre $\mathcal{B}$ y $\prod_{a,b\in\mathcal{C}}(a + b\alpha)$ es un cuadrado en el anillo de enteros algebraicos de $\mathbb{Q}[\alpha]$, entonces*

$$\sum_{a,b\in\mathcal{C}} v(a + b\alpha) \equiv 0 \mod 2$$

Complejidad

Como en casos anteriores, el análisis de complejidad es heurístico.

El funcionamiento básico de la criba del cuerpo de números puede resumirse del modo siguiente: A es una fuente aleatoria de números produciendo la sucesión $\{a_i\}$. Se examina esa sucesión hasta que se encuentra una subsucesión $\{a_i\}_{i\in I}$ cuyo producto es un cuadrado. La cota del tiempo de ejecución del algoritmo está dominada por el tamaño de la subsucesión más chica. Cribar los números para quedarnos con los smooths facilita la tarea de encontrar los cuadrados utilizando álgebra lineal en un conjunto de vectores de exponentes. Entonces, para calcular la complejidad del algoritmo, lo que debemos hacer es estimar la longitud de esa subsucesión.

Llamemos L a la función definida como

$$L(x) = e^{\sqrt{\log x \log\log x}}.$$

Puede verse ([7], Sección 6.2.3) que:

1. Los elementos a_i de la sucesión están acotados en valor absoluto por un entero X.

2. X es tal que $\log X \sim \frac{4}{3^{1/3}}(\log n)^{2/3}(\log\log n)^{1/3}$ y $\log\log X \sim \frac{2}{3}\log\log n$

3. El valor esperado para la longitud de la subsucesión cuyo producto es un cuadrado es $L(X)^{\sqrt{2}+o(1)}$ (Teorema 6.2.2. en [7]).

evaluando $L(X)$ con las aproximaciones de $\log X$ y $\log\log X$ resulta

$$L(X)^{\sqrt{2}+o(1)} \approx \exp\left((64/3)^{1/3} + o(1))(\log n)^{1/3}(\log(\log n)\right)^{2/3}$$

qué, multiplicado por la cantidad de operaciones elementales que tome la aritmética de enteros, da la complejidad (heurística) del algoritmo.

Capítulo 2

Targets

2.1. Introducción y definiciones

Las soluciones a la congruencia

$$n + x^2 \equiv y^2 \mod c \,, \tag{2.1}$$

módulo un entero c de tamaño modesto pueden obtenerse en un tiempo razonable (al menos tan razonable como c): es suficiente con probar la mitad de los elementos $\bar{x} \in \mathbb{Z}_c$ y guardar los que satisfacen $n + \bar{x}^2 \in (\mathbb{Z}_c)^2$. Además, 2.1 es la reducción módulo c de la ecuación de Fermat

$$n + x^2 = y^2 \tag{2.2}$$

y cualesquiera que sean las soluciones x^*, y^* de 2.2, deben ser equivalentes módulo c a un par $\bar{x}, \bar{y} \in \mathbb{Z}_c$ solución de 2.1. El par solución de 2.1, desafortunadamente, es único sólo para una cantidad finita de enteros c pero como se verá más adelante, bajo ciertas condiciones el cociente entre la cantidad de soluciones y c se aproxima de modo asintótico a cero.

Lo anterior es la motivación para las definiciones que siguen

Definición 1. *Un target de n es una terna de naturales (a, b, c) con $a, b \in (\mathbb{Z}_c)^2$ de modo que $n + a \equiv b \mod c$.*

Definición 2. *Un target (a, b, c) para n es un target* correcto *si para alguna solución x^*, y^* de $n + x^2 = y^2$ se tiene $a \equiv x^{*2} \mod c$ y $b \equiv y^{*2} \mod c$.*

Por ejemplo, si $n \equiv 1 \mod 3$ entonces $(0, 1, 3)$ es un target para n y es el único de la forma $(a, b, 3)$, por lo tanto es correcto. Lo mismo puede decirse para cualquier $n \equiv 1 \mod 4$ y el target $(0, 1, 4)$.

Por otra parte, puede haber más de un target correcto, por ejemplo si $n = 13{\cdot}17$ entonces el target $(16, 9, 19)$ corresponde a las soluciones $x = \pm 110$, $y = \pm 111$ (que da la factorización trivial de n) y el target $(4, 16, 19)$ a las soluciones $x = \pm 2$, $y = \pm 15$ que dan la factorización de n en primos.

Aún más importante es el hecho de que, tal como se dijo antes, si $n \equiv 1$ mód 3 el par solución x^*, y^* de 2.2 debe cumplir

$$x^{*2} = 0 + 3t \quad y^{*2} = 1 + 3u$$

para algún par de enteros t, u.

Si se observara todo en base 3, los primeros dígitos de x^* y y^* serían conocidos y aunque parezca poco, esa información parcial sobre la solución resulta de utilidad para reducir el tiempo de búsqueda, tal como se verá en el capítulo 3.

Por otra parte, un target *induce* expresiones cuadráticas para las posibles soluciones a 2.2. En efecto, supongamos que (a, b, c) es un target para n y

por simplicidad, que es único. Entonces x^* debe ser equivalente módulo c a alguna raíz cuadrada de a (módulo c), por eso para algún entero z y alguna $0 \leq \alpha < c$ con $\alpha^2 \equiv a \mod c$,

$$x^* = \alpha + cz \tag{2.3}$$

De igual modo:

$$y^* = \beta + cw \tag{2.4}$$

para algún entero w y alguna raíz cuadrada β de b módulo c.

En lo que sigue *parábola para x* se referirá a la expresión cuadrática inducida por elevar al cuadrado ambos miembros de la ecuación 2.3, y *parábola para y* a la correspondiente a 2.4.

Por ejemplo, $(0, 1, 3)$ es un target único para $n = 91$ y las raíces cuadradas de 1 modulo 3 son 1 y 2. La parábola inducida para x es $(3z)^2$ y las parábolas inducidas para y

$$(1 + 3w)^2, \quad (2 + 3w)^2$$

Lo anterior puede resumirse en la siguiente definición

Definición 3. *Si (a, b, c) es un target para n, las parábolas para x inducidas por el target son las expresiones dependientes de z de la forma*

$$P(z) = (\alpha + cz)^2$$

con $\alpha^2 \equiv a \mod c$, $0 \leq \alpha < c$.

Las parábolas para y inducidas por el target son las expresiones dependientes de w de la forma

$$Q(w) = (\beta + cw)^2$$

con $\beta^2 \equiv b \mod c$, $0 \leq \beta < c$.

Se restringirá el dominio de las parábolas sólo a los enteros. Tomando definiciones prestadas de la teoría de formas cuadráticas, diremos que la parábola P representa a $m \in \mathbb{Z}$ si existe $z \in \mathbb{Z}$ de modo que $P(z) = m$ (o también se dirá: $z \in \mathbb{Z}$ es una representación de $m \in \mathbb{Z}$ por P). Dos parábolas P_1 y P_2 son equivalentes si representan los mismos números, eso es si $Im(P_1) = Im(P_2)$.

Por último, es importante notar que si $\alpha^2 \equiv \alpha'^2 \mod c$ y $\alpha = c - \alpha'$ entonces las parábolas inducidas por α y α' son equivalentes porque

$$(\alpha + cz)^2 = (c - \alpha' + cz)^2 = (-\alpha' + c(z + 1))^2$$

2.2. Hipérbolas modulares y targets

El conjunto de enteros

$$\mathcal{H}_{n,c} = \{(x,y) \mid xy \equiv n \mod c,\ 0 \leq x,y < c\}$$

es llamado hipérbola modular de n módulo c. Notemos que si $mcd(n,c) = 1$ entonces $\mid \mathcal{H}_{n,c} \mid = \phi(c)$ (porque $(1, n \mod c) \in \mathcal{H}_{n,c}$ y para cada u en $\mathbb{Z}_c^*$, $(u, u^{-1}n \mod c) \in \mathcal{H}_{n,c}$). Si se considera a $\mathcal{H}_{n,c}$ en el espacio euclidiano, puede definirse el conjunto de distancias de la hipérbola como

$$\mathcal{D}_{n,c} = \{|x-y| \mid x,y \in \mathcal{H}_{n,c}\}$$

En lo que sigue, p denotará a un primo. Como es usual, $(\frac{n}{p})$ es el símbolo de Legendre (ver [14] capítulo 5).

Teorema 1. *Si $p > 2$ y $mcd(n,p) = 1$*

$$\mid \mathcal{D}_{n,p} \mid = \begin{cases} \frac{p-1}{4} + (1 + (\frac{n}{p}))/2 & \text{si } p \equiv 1 \mod 4 \\ \frac{p-3}{4} + 1 & \text{si } p \equiv 3 \mod 4 \end{cases}$$

La demostración del teorema se pospondrá hasta la sección siguiente. Una prueba distinta puede encontrarse en [27], Teorema 1. Sin embargo, puede tenerse una intuición geométrica al ver que los puntos fijos de $f_1(x,y) = (y,x)$ son de la forma (x,x) y en caso de existir, son las raíces cuadradas modulares de n. De modo similar los puntos fijos de $f_2(x,y) = (p-y, p-x)$ son de la forma $(x, p-x)$ y en caso de existir, son exactamente las raíces cuadradas modulares de $-n$.

Sean

$$R_1 = \{(x,y) \in \mathcal{H}_{n,p} \mid\ 0 \leq x < p, 0 \leq y < \text{mín}(x, p-x)\}$$

$$\bar{R}_1 = \{(x,y) \in \mathcal{H}_{n,p} \mid\ 0 \leq x < p, 0 \leq y \leq \text{mín}(x, p-x)\}$$

(R_1 es el triángulo con vértices $(0,0)$, $(0,p)$, $(\frac{p-1}{2}, \frac{p-1}{2})$ y $\bar{R}_1$ es R_1 más sus bordes).

Hagamos

$$R_2 = f_2(R_1), R_3 = f_1(f_2(R_1)), R_4 = f_1(R_1)$$

y en general $\bar{R}_j = f(\bar{R}_i)$, remplazando f por f_1, f_2 o una composición de ambas. Nótese además que $\mathcal{H}_{n,p}$ es la unión disjunta de $\bar{R}_1, R_2, \bar{R}_3, R_4$.

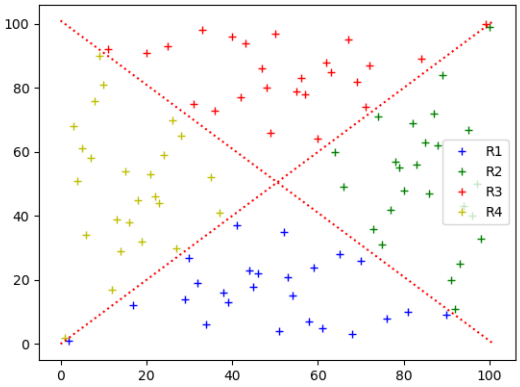


Figura 2.1: Hipérbola modular para $n = 507527$, $p = 101$

Lema 3. *Para* $0 \leq u \leq p-1$, *sea*

$$A_u = \{(x, y) \in \mathcal{H}_{n,p} \mid u = |x - y|\}.$$

Entonces

1. *Si* y *es raíz de* $Y(Y-u) \equiv n \mod p$, *entonces* $(y-u, y) \in A_u$
2. *Si* y *es raíz de* $Y(Y+u) \equiv n \mod p$, *entonces* $(y+u, y) \in A_u$
3. *Si* $(x, y) \in A_u$ *entonces* y *es raíz de la ecuación en 1 o en 2*

Demostración. Los cálculos son fácilmente verificables; se omite la demostración. □

Como corolario, se ve que si $(\frac{u^2+4n}{p}) = 1$ entonces $|A_u| = 4$ y si $u^2+4n \equiv 0$ mód p, se tiene $|A_u| = 2$. Por otra parte si $(\frac{u^2+4n}{p}) = -1$ entonces $|A_u| = 0$.

Lema 4. *Si* $p > 2$, $gcd(n, p) = 1$, $p \equiv 1 \mod 4$ *y* $(\frac{n}{p}) = -1$ *hay una relación biyectiva entre* R_1 *y* $\mathcal{D}_{n,p}$

Demostración. Para un par $(x, y) \in \mathcal{H}_{n,p}$, sea $u = |x - y| \in \mathcal{D}_{n,p}$,

$$A = \{(x, y), (y, x), (p-x, p-y), (p-y, p-x)\}$$

y $\psi : R_1 \to \mathcal{D}_{n,p}$ la aplicación que lleva puntos en R_1 a distancias en $\mathcal{D}_{n,p}$, de modo que

$$\psi(x, y) = |x - y|$$

Claramente $A \subset A_u$. Los elementos de A son todos distintos porque $(\frac{n}{p}) = -1$ y $(\frac{-n}{p}) = -1$ y por ello $A = A_u$ (por hipótesis, $|A_u| = 4$). Si $(x, y) \in R_1$ entonces $\psi(x, y) = |x - y| = u$. Por el contrario, si $(x, y) \notin R_1$, las transformaciones f_1, f_2 o una composición de ellas lleva (x, y) a un punto $(x', y') \in R_1$ con $|x - y| = |x' - y'| = u$, eso prueba la sobreyectividad de ψ. Por otra parte, ψ es inyectiva, en efecto si

$$\psi(x_1, y_1) = \psi(x_2, y_2)$$

entonces

$$u = |x_1 - y_1| = |x_2 - y_2|$$

y $(x_1, y_1), (x_2, y_2) \in R_1 \cap A_u$, pero A_u puede tener sólo un elemento de R_1, con lo cual necesariamente $(x_1, y_1) = (x_2, y_2)$ □

La condición $(\frac{n}{p}) = -1$ y $p \equiv 1 \mod 4$ impide que el conjunto A en la demostración anterior, tenga elementos en el borde de R_1. Consideremos ahora la extensión de ψ para incluir los bordes de R_1.

Lema 5. *Si $p > 2$ y $mcd(n, p) = 1$, hay una relación biyectiva entre $\bar{R}_1$ y $\mathcal{D}_{n,p}$*

Demostración. Sea $\bar{\psi} : \bar{R}_1 \to \mathcal{D}_{n,p}$ la aplicación definida por $\bar{\psi}(x, y) = |x-y|$.

Si $u \in \mathcal{D}_{n,p}$ entonces existe un par $(x, y) \in \mathcal{H}_{n,p}$ con $|x - y| = u$. Si $(x, y) \in \bar{R}_1$ entonces $\bar{\psi}(x, y) = u$. Por el contrario, si $(x, y) \notin \bar{R}_1$ entonces las transformaciones f_1, f_2 o una composición de ellas lleva (x, y) a un punto $(x', y') \in \bar{R}_1$ con $|x - y| = |x' - y'| = u$. Eso prueba que $\bar{\psi}$ es sobreyectiva.

Por otra parte, $\bar{\psi}$ restringida a R_1 es inyectiva, porque si $(x_1, y_1), (x_2, y_2) \in R_1$ con $\bar{\psi}(x_1, y_1) = |x_1 - y_1| = \bar{\psi}(x_2, y_2) = |x_2 - y_2| = u$ entonces

$$A_u = \{(x_1, y_1), (y_1, x_1), (p - x_1, p - y_1), (p - y_1, p - x_1)\}$$

y $(x_1, y_1), (x_2, y_2) \in A_u$. Pero A_u tiene exactamente un elemento de R_1, por lo tanto $(x_1, y_1) = (x_2, y_2)$.

Sólo falta considerar la inyectividad en los puntos de $\bar{R}_1$ que están en los bordes.

Supongase que $\bar{\psi}(x_1, y_1) = \bar{\psi}(x_2, y_2)$.

Si para $(x_1, y_1) \in \bar{R}_1$ se tiene $x_1 = y_1$, en ese caso $u = |x_1-y_1| = |x_2-y_2| = 0$. Exactamente un punto del conjunto $A_0 = \{(x_1, x_1), (p - x_1, p - x_1)\}$ está en $\bar{R}_1$, por lo tanto $(x_1, y_1) = (x_2, y_2) = (x_1, x_1)$.

Por el contrario si $(x_1, y_1) \in \bar{R}_1$ cumple $y_1 = p - x_1$, en ese caso para $u = |x_1 - (p - x_1)| = |2x_1 - p|$ se tiene $A_u = \{(x_1, p - x_1), (p - x_1, x_1)\}$ y exactamente un punto de A_u está en $\bar{R}_1$, por lo tanto $(x_1, y_1) = (x_1, p-x_1) = (x_2, y_2)$. □

Por lo anterior, la relación biyectiva entre R_1 y $\mathcal{D}_{n,p}$ puede extenderse a una entre $\bar{R}_1$ y $\mathcal{D}_{n,p}$.

En el lema anterior, R_1 puede cambiarse por cualquier R_i, en virtud de las transformaciones f_1 y f_2 (y lo mismo con $\bar{R}_1$ y $\bar{R}_i$).

El conjunto de distancias tiene una relación importante con las soluciones de la ecuación de Fermat $n + x^2 = y^2$ módulo p y la factorización de n.

Llamamos $T_{n,c}$ al conjunto de targets de n módulo c.

Teorema 2. *Sea $p > 2$ con $mcd(n, p) = 1$. Hay una relación biyectiva entre $T_{n,p}$ y $\mathcal{D}_{n,p}$*

Demostración. Hay una relación biyectiva entre $T_{n,p}$ y $\bar{R}_4$. En efecto, la aplicación $\gamma_1 : \bar{R}_4 \to T_{n,p}$ definida por

$$(x, y) \mapsto (2^{-2}(x - y)^2, 2^{-2}(x + y)^2) = (a, b)$$

lleva puntos de $\bar{R}_4$ a targets.

La aplicación $\gamma_2 : T_{n,p} \to \bar{R}_4$ definida por

$$(a, b) \mapsto A_u \cap \bar{R}_4 = \{(x, y)\}$$

con $0 \leq u = 2\alpha < p$, $\alpha^2 \equiv a \mod p$ y $\alpha < p/2$ lleva targets en puntos de $\bar{R}_4$ y claramente son inversas la una de la otra (obsérvese que $2^{-1}u = \alpha$ es una raíz cuadrada de a, módulo p por lo tanto, $u = x - y$ o $u = y - x$).

Del Lema 5 se sigue el teorema.

□

2.3. Cantidad de targets

Naturalmente, para encontrar la solución a la ecuación de Fermat usando targets debemos usar el target correcto. Por inspección puede corroborarse que no hay targets únicos de la forma $(a, b, 13)$ para $n \equiv 1 \mod 13$ o $(a, b, 17)$ para $n \equiv 1 \mod 17$ y eso vale para todos los c primos impares excepto 3 y 5, de modo que la estrategia de buscar un c grande tiene el efecto no deseado de aumentar la cantidad de targets y por lo tanto, la cantidad de parábolas para x.

Pero no alcanza con saber sólo que la cantidad de targets aumenta en proporción a c, sino que es importante determinar esa proporción. Esta sección presenta esos resultados así como un análisis asintótico de la cantidad de targets en relación a la magnitud de c, para el caso en que c sea un producto de primos impares consecutivos, libre de cuadrados.

Se define $\tau : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$ como

$$\tau(n, c) = \mid \{a, b \in (\mathbb{Z}_c)^2 \mid n + a \equiv b \mod c)\} \mid \tag{2.5}$$

Como en la sección precedente, p denota a un primo impar. Nótese que $\left(\frac{n}{p}\right) = 1$ si y sólo si existe un target de la forma $(0, n, p)$ para n. Del mismo modo, $\left(\frac{-n}{p}\right) = 1$ sí y sólo si existe un target de la forma $(-n, 0, p)$ para n.

Para una ecuación modular $f(x) \equiv a \mod p$, sus soluciones son los elementos en $\mathbb{Z}_p$ que satisfacen la congruencia. Se llamará $N(x^2 = a)$ a la cantidad de soluciones a $x^2 \equiv a \mod p$ y de modo similar $N(n + x^2 = y^2)$ a la cantidad de soluciones a $n + x^2 \equiv y^2 \mod p$. Como puede comprobarse fácilmente, $N(x^2 = a) = \left(1 + \left(\frac{a}{p}\right)\right)$. El lema a continuación es intuitivo si se recuerda la ecuación canónica de una hipérbola y la observación sobre la cantidad de puntos en $\mathcal{H}_{n,c}$ para c primo.

Lema 6. *Para un entero n y un primo impar p con $p \nmid n$, la cantidad de soluciones a $n + x^2 \equiv y^2 \mod p$ es $p - 1$.*

Demostración. El número de soluciones a $n+x^2 \equiv y^2 \mod p$ puede escribirse como

$$N(n + x^2 = y^2) = \sum_{n+a \equiv b \mod p} N(x^2 = a) N(y^2 = b) \tag{2.6}$$

como se dijo antes $N(x^2 = a) = 1 + \left(\frac{a}{p}\right)$, y por eso, 2.6 es igual a

$$\sum_{n+a \equiv b \mod p} \left(1 + \left(\frac{a}{p}\right)\right)\left(1 + \left(\frac{b}{p}\right)\right) = \tag{2.7}$$

$$\sum_{a \in \mathbb{Z}_p} 1 + \sum_{a \in \mathbb{Z}_p} \left(\frac{a}{p}\right) + \sum_{b \in \mathbb{Z}_p} \left(\frac{b}{p}\right) + \sum_{n+a \equiv b \mod p} \left(\frac{a}{p}\right)\left(\frac{b}{p}\right). \tag{2.8}$$

Para un caracter no trivial χ sobre un cuerpo $\mathbb{F}_p$, la suma $\sum_{a\in\mathbb{F}_p^*}\chi(a)$ es 0 ([14] Proposición 8.1.2). El símbolo de Legendre es un caracter no trivial sobre $\mathbb{Z}_p^*$, así que la segunda y tercer suma son iguales a 0.

Por otra parte, renombrando $a=-na'$ y $b=nb'$ se obtiene

$$\begin{aligned}\sum_{n+a\equiv b}\left(\frac{a}{p}\right)\left(\frac{b}{p}\right) &= \sum_{a'+b'\equiv 1}\left(\frac{-na'}{p}\right)\left(\frac{nb'}{p}\right)\\ &= \left(\frac{-n}{p}\right)\left(\frac{n}{p}\right)\sum_{a'+b'\equiv 1}\left(\frac{a'}{p}\right)\left(\frac{b'}{p}\right)\\ &= (-1)^{\frac{p-1}{2}}\sum_{a'+b'\equiv 1}\left(\frac{a'}{p}\right)\left(\frac{b'}{p}\right)\end{aligned}$$

(las equivalencias son módulo p). La suma $\sum_{a'+b'\equiv 1}\left(\frac{a'}{p}\right)\left(\frac{b'}{p}\right)$ es una suma de Jacobi de la forma $J(\chi,\chi)$, con el símbolo de Legendre como el caracter χ. Más aún, como es un caracter de orden 2, $J(\chi,\chi)=J(\chi,\chi^{-1})$ y el caso especial

$$J(\chi,\chi^{-1})=-(-1)^{\frac{p-1}{2}}$$

se cumple (ver Teorema 1 en el capítulo 8, sección 3 de [14]). Finalmente,

$$N(n+x^2=y^2)=p+(-1)^{\frac{p-1}{2}}(-(-1)^{\frac{p-1}{2}})=p-1$$

□

Teorema 3. *Si p es un primo impar coprimo con n, entonces*

$$\tau(n,p)=\begin{cases}\frac{p-1}{4}+(1+(\frac{n}{p}))/2 & \text{si } p\equiv 1 \mod 4\\ \frac{p-3}{4}+1 & \text{si } p\equiv 3 \mod 4\end{cases}$$

Demostración. Es útil notar primero que $N(x^2=a)=2$ si a es un residuo cuadrático de p distinto de cero y $N(x^2=a)=1$ si $a\equiv 0 \mod p$.

La prueba es por casos. Si $p\equiv 1 \mod 4$ y $\left(\frac{n}{p}\right)=-1$ no hay ningún target (a,b,p) con $a=0$ o $b=0$, y cada target (a,b,p) da cuatro soluciones a la ecuación $n+x^2\equiv y^2 \mod p$. Eso es,

$$4\tau(n,p)=p-1.$$

por el Lema 6. O sea

$$\tau(n,p)=\frac{p-1}{4}.$$

Si $p \equiv 1 \mod 4$ y $\left(\frac{n}{p}\right) = 1$ entonces también $\left(\frac{-n}{p}\right) = 1$ en consecuencia existe un target $(0, b, p)$ y otro $(a, 0, p)$ con $a \neq 0$, $b \neq 0$, cada uno da dos soluciones a $n + x^2 \equiv y^2 \mod p$ y el resto de los targets dan cuatro. Eso significa que

$$4(\tau(n,p) - 2) + 4 = p - 1$$

luego

$$\tau(n,p) = \frac{p-1}{4} + 1$$

Finalmente, si $p \equiv 3 \mod 4$, exactamente uno del par $\{n, -n\}$ es un residuo cuadrático modulo p. Entonces existe exactamente un target (a, b, p) de n con $a = 0$ o $b = 0$. Ese target da dos soluciones a $n + x^2 \equiv y^2 \mod p$ y cada uno del resto da cuatro. En este caso,

$$4(\tau(n,p) - 1) + 2 = p - 1$$

es decir

$$\tau(n,p) = \frac{p-3}{4} + 1$$

□

Si s y t son enteros positivos coprimos y $mcd(n, st) = 1$ entonces $\tau(n, st) = \tau(n, t)\tau(n, s)$ por aplicación del teorema chino del resto. En efecto, si ψ es el isomorfismo $\mathbb{Z}_s \times \mathbb{Z}_t \to \mathbb{Z}_{st}$, y (a_s, b_s, s), (a_t, b_t, t) son targets para n entonces $(\psi(a_s, a_t), \psi(b_s, b_t), st)$ es un target para n porque $\psi(a_s, a_t)$ es un residuo cuadrático módulo st dado que a_s y a_t son residuos cuadráticos módulo s y t respectivamente. Lo mismo ocurre con b_s y b_t. Por último, $(n \mod s, n \mod t) + (a_s, a_t) \equiv (b_s, b_t) \mod st$ porque $n + a_s \equiv b_s \mod s$ y $n + a_t \equiv b_t \mod t$.

Como ejemplo de lo anterior, si $\left(\frac{n}{5}\right) = -1$, n tiene target único de la forma $(a, b, 5)$. Además cualquier n tiene target único de la forma $(a, b, 3)$ entonces, si $\left(\frac{n}{5}\right) = -1$, n tiene target único de la forma $(a, b, 15)$.

Por inspección es fácil comprobar que si n no es divisible por 3, tiene target único módulo 9, pero no hay targets únicos módulo 81. Tampoco hay targets únicos módulo 25 y por último, existen targets únicos módulo 32 pero no módulo 64. Como ningún otro primo impar, excepto 3 y 5 tienen targets únicos, el mayor c para el cuál existen targets únicos es $c = 2^5 3^2 5 = 1440$.

El teorema que sigue es importante porque da una relación asintótica entre la cantidad de targets para un módulo determinado y la magnitud de ese módulo. Como es usual, π es la función de contar primos (eso es, $\pi(B)$ es la cantidad de primos menores o iguales que B).

Teorema 4. *Si*

$$c = \prod_{2<p\leq B} p$$

y $\left(\frac{n}{p}\right) = -1$ *para todos los primos* $p \leq B$ *con* $p \equiv 1 \mod 4$*, entonces*

$$\frac{\tau(n,c)}{c} = O(4^{-\pi(B)} \log B)$$

.

Demostración. Observemos que

$$\Big(\prod_{2<p\leq B} \frac{p-1}{p}\Big)\Big(\prod_{2<p\leq B} \frac{p+1}{p}\Big) = \prod_{2<p\leq B} 1 - \frac{1}{p^2} = O(1).$$

Además, por el teorema de Mertens (Corolario 2.10 en [13] y Teorema 5.13 en [26])) $\prod_{2<p\leq B} \frac{p-1}{p} = O(\frac{1}{\log B})$. De allí

$$\prod_{2<p\leq B} \frac{p+1}{p} = O(\log B)$$

y por lo tanto, puesto que τ es multiplicativa, $\frac{\tau(n,c)}{c} = \prod_{2<p\leq B} \frac{\tau(n,p)}{p}$ y

$$\prod_{2<p\leq B} \frac{p-1}{4p} \leq \prod_{2<p\leq B} \frac{\tau(n,p)}{p} \leq \prod_{2<p\leq B} \frac{p+1}{4p}.$$

De lo anterior se sigue el resultado. □

Teorema 5. *Para c como en el enunciado del teorema anterior y coprimo con n, se tiene que*

$$\frac{1}{c} \prod_{2<p\leq B} \Big(\tau(n,p) - \big(1 + \Big(\frac{n}{p}\Big)\big)/2\Big) = O(4^{-\pi(B)} \log B)$$

.

Demostración. Se sigue de la prueba anterior y la desigualdad $\tau(n,p) - \big(1 + \Big(\frac{n}{p}\Big)\big)/2 \leq \frac{p+1}{4}$. □

Teorema 6. *Sea p un primo impar, n un entero con* $p \nmid n$ *y* $k > 0$ *un entero.*

1. *Si* (a, b, p^k) *es un target para n con* $a \neq 0$ *y* $b \neq 0$*, entonces hay p targets* (a', b', p^{k+1}) *de n con* $a \equiv a' \mod p^k$ *y* $b \equiv b' \mod p^k$*.*

2. *Si* $(0, b, p^k)$ *es un target para* n*, la cantidad de targets* (a', b', p^{k+1}) *de* n *con* $0 \equiv a' \mod p^k$ *y* $b \equiv b' \mod p^k$ *es*

   a) $|(\mathbb{Z}_p)^2|$ *si* k *es par.*

   b) 1 *si* k *es impar.*

3. *Si* $(a, 0, p^k)$ *es un target para* n*, la cantidad de targets* (a', b', p^{k+1}) *de* n *con* $a \equiv a' \mod p^k$ *y* $0 \equiv b' \mod p^k$ *es*

   a) $|(\mathbb{Z}_p)^2|$ *si* k *es par.*

   b) 1 *si* k *es impar.*

Demostración. 1. Si g es un residuo cuadrático módulo p^k, $g + tp^k$ es un residuo cuadrático de p^{k+1} para $0 \leq t < p$. (Eso puede verse de modo elemental pues si $g = \alpha^2 + rp^k$ entonces $(\alpha + sp^k)^2 \equiv g - rp^k + 2\alpha sp^k$ mód p^{k+1} y basta escoger $0 \leq s < p$ de modo que $2\alpha s - r \equiv t \mod p$).

   El conjunto

$$\{(a + tp^k, b + t'p^k, p^{k+1}) \mid t' = (n + a - b)/p^k + t,\ 0 \leq t < 0\}$$

   es un conjunto de targets para n.

2. Si k es par, tp^k es residuo cuadrático módulo p^{k+1} siempre que $t \in (\mathbb{Z}_p)^2$. Además $b + t'p^k$ es residuo cuadrático para $0 \leq t' < p$. El conjunto

$$\{(tp^k, b + t'p^k, p^{k+1}) \mid t' = (n - b)/p^k + t,\ 0 \leq t < 0\}$$

   es un conjunto de targets para n.

   Por el contrario, si k es impar, tp^k es residuo cuadrático módulo p^{k+1} sólo para $t = 0$.

3. Análogo al caso anterior.

□

Para abreviar la notación se escribirá $s_p(n) = (1 + (\frac{n}{p}))/2 + (1 + (\frac{-n}{p}))/2$.

Teorema 7. *Si* p *es un primo impar con* $p \nmid n$*, entonces para* $k > 1$

$$\tau(n, p^{k+1}) = \begin{cases} [\tau(n, p^k) - s_p(n)]p + s_p(n)|(\mathbb{Z}_p)^2| & \text{si } k \text{ es par} \\ [\tau(n, p^k) - s_p(n)]p + s_p(n) & \text{si } k \text{ es impar} \end{cases}$$

Demostración. Consecuencia inmediata del Teorema 6. □

2.4. Factorización con targets

Habiendo definido targets y establecido su relación con las hipérbolas modulares, podemos adelantar el objetivo principal del resto del trabajo. Para un entero n producto de dos primos p, q con $\frac{1}{2}|p-q| < n^{1-\epsilon}$ y $0 < \epsilon < 1$ daremos un algoritmo determinístico para factorizar n utilizando targets con tiempo de ejecución del orden de $O(2^{-m}n^{1-\epsilon}M)$.

Las cantidades m y M dependen de $\log n$. m es la cantidad máxima de primos consecutivos impares $p_1, \ldots, p_m$ de modo que su producto es menor o igual a $n^{1-\epsilon}$ (como se verá m puede estimarse en función de $\log n$). M es el máximo en operaciones elementales que se requiere para multiplicar números de tamaño $\log n$ o calcular la raíz cuadrada entera de números de tamaño algo mayor a $\log n$.

En detalle, el resultado principal es el siguiente:

Teorema 8. *Para $n = pq$ con p, q primos impares y $0 < \epsilon < 1$ de modo que $\frac{|p-q|}{2} \leq n^{1-\epsilon}$ el Algoritmo 8 de la sección 3.1 con entradas n, ϵ, $m = \lfloor \frac{0{,}79 \log n}{\log((1-\epsilon)\log n)} \rfloor$ y $r = \lfloor m/2 \rfloor$, devuelve el par p, q o q, p. La cantidad estimada de operaciones elementales en bits que requiere es de orden*

$$O(2^{-m} \log(p_1 \ldots p_m) n^{1-\epsilon} M)$$

con p_i el i-ésimo primo impar y $M = \text{máx}(Mul(\log n, Sqrt(\text{máx}(1, 2(1-\epsilon))\log n)))$.

El argumento para la correctitud del algoritmo puede encontrarse en la sección 3.1, así como su presentación detallada.

El cálculo de complejidad y la estimación de m se desarrollan principalmente en la sección 3.2.

Por otra parte, es útil notar que el producto $2^{-m}n^{1-\epsilon}$ puede aproximarse como $2^{-\frac{0{,}79 \log n}{\log((1-\epsilon)\log n)}} n^{1-\epsilon}$ o de forma más compacta, $n^{1-\epsilon-a\log 2}$ para $a = 0{,}79/\log\Big((1-\epsilon)\log n\Big)$.

De interés en sí mismo y como requisito para el objetivo principal, es necesario un algoritmo para calcular todos los targets de n módulo $c = p_{i_1} \ldots p_{i_k}$ con eficiencia, así como las raíces cuadradas modulares de las primeras componentes de los targets.

En la sección 2.5 daremos un algoritmo simple para calcular targets y raíces cuadradas modulares en el caso de módulos primos (que se espera sean pequeños). El algoritmo toma un entero n y un primo impar p. Devuelve $T_{n,p}$ el conjunto de targets de n módulo p y $R_{n,p}$ las raíces cuadradas módulo p de las primeras componentes de cada target. El tiempo de ejecución es del orden $O(p \log p Mul(\log p))$.

Por otra parte, en la sección siguiente mostraremos un procedimiento que puede usarse para construir el conjunto de targets de n módulo $c = p_{i_1} \ldots p_{i_k}$

a partir de los conjuntos $T_{n,p_{i_j}}$ para $j = 1, \ldots, k$. La cantidad de operaciones para hacerlo es $O(\tau(n,c)\log c)$.

Finalmente, una vez que se cuenta con el conjunto de targets $T_{n,c} = \{(a_i, b_i, c)\}_{i=1}^{\tau(n,c)}$, se puede construir el conjunto $R_{n,c}$ de raíces cuadradas módulo c para todos los a_i a partir de los conjuntos $R_{n,p_{i_1}}, \ldots, R_{n,p_{i_k}}$ utilizando el Algoritmo 5. El número de operaciones elementales que requiere es $O(2^{-2k}ck \log p_{i_k} Mul(\log p_{i_k}) + 2^{-k}k \log c)$.

Antes de terminar esta sección es necesario decir algo sobre el tiempo de ejecución del algoritmo principal, el parámetro ϵ y para qué casos funciona mejor. La cantidad $n^{1-\epsilon}$ es una cota para el valor absoluto de la parte x de la solución a la ecuación diofántica $n + x^2 = y^2$. El algoritmo busca esa solución y para hacerlo considera al intervalo de enteros $[-n^{1-\epsilon}, n^{1-\epsilon}]$ como el conjunto de posibles valores que puede tomar x. Una forma especial de expresar x utilizando targets módulo c y c' reduce notablemente el espacio de búsqueda. La magnitud de esa reducción se manifiesta en el orden de complejidad del algoritmo con el exponente m. Informalmente, cada primo en el producto $c \cdot c' = p_1 \ldots p_m$ divide por la mitad el espacio de búsqueda, mientras que el mayor primo lo incrementa logarítmicamente. La idea detrás del algoritmo es utilizar la mayor cantidad de primos posibles en $c \cdot c'$.

El algoritmo es especialmente indicado para valores de ϵ mayores o iguales a $1/2$, es decir cuando $|x| \leq n^{1/2}$. Incluso para números grandes, un ϵ cercano a 0,55 asegura un tiempo de ejecución del orden $O(n^{1/3}M)$ (por ejemplo, $\epsilon = 0{,}57$ y n de 1024 bits ó $\epsilon = 0{,}56$ y n de 512 bits).

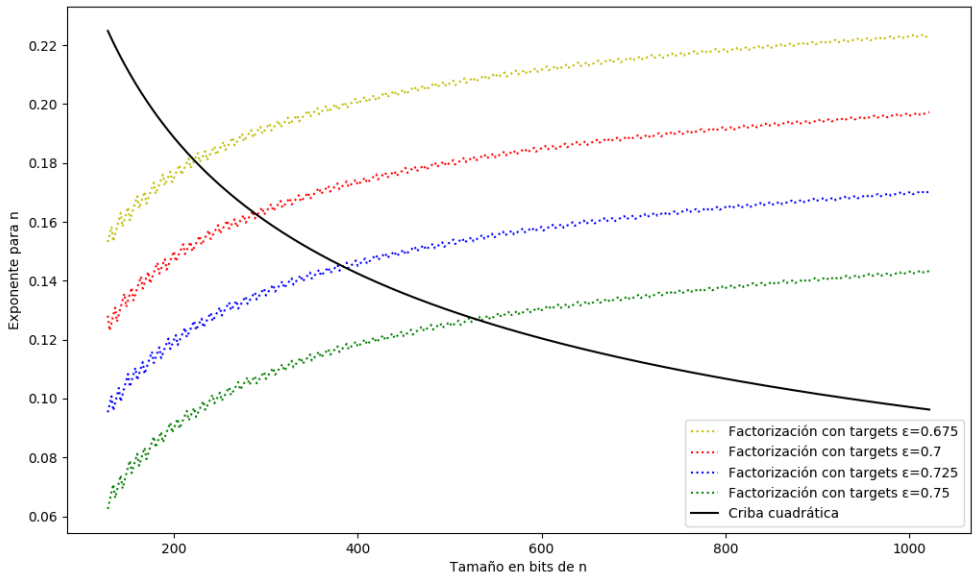
Para los casos en los que $\epsilon < 1/2$ (eso implica $|x| > n^{1/2}$) la modificación al método de Lehman para utilizar targets propuesta en la sección 3.3 es más indicada. El algoritmo de Lehman y el algoritmo de factorización con targets son de complejidad exponencial y determinísticos. Ninguno de los dos métodos es considerado con preferencia para atacar de forma práctica el problema para números grandes.

Los métodos subexponenciales tratados en los antecedentes tienen mejor desempeño aunque ha pasado mucho tiempo desde el último récord (ver [16]). Incluso el método rho de Pollard es bastante útil para números pequeños (ocupa muy poca memoria, entre otras cosas) y es implementado en el programa *factor*, disponible en la mayoría de los sistemas operativos Linux.

Sin embargo, por ejemplo para la criba cuadrática y algunos casos particulares de n ($\epsilon = 0{,}75$ y n de 530 bits o menos) el tiempo de ejecución estimado para el algoritmo que presentamos es menor que el correspondiente para la criba. Ello puede verse en la Figura 2.2

Finalmente, más allá de las ventajas en casos muy particulares, las herramientas que se proponen colaboran en mostrar una nueva vía de ataque

Figura 2.2: Comparación de los exponentes de n en la cota para el tiempo de ejecución del algoritmo de factorización con targets y la criba cuadrática



a un problema difícil como la factorización de enteros.

2.5. Algoritmos para targets y raices

Una pregunta no menor relativa a la eficiencia es ¿cómo obtener todos los targets módulo c para un n dado? La solución ingenua es construir una lista L de residuos cuadráticos módulo c, y buscar exhaustivamente en $L \times L$ pares de la forma $(a, n + a \mod c)$.

Aún en el caso en que el cardinal de $(\mathbb{Z}_c)^2$ sea menor en magnitud que c, resta decir cómo construir la lista L sin necesidad de recorrer $\mathbb{Z}_c$.

Afortunadamente existe un algoritmo eficiente para hacer ambas cosas. El procedimiento es esencialmente una aplicación del teorema chino del resto, con una manipulación inteligente de la representación para calcular los elementos del conjunto.

Concretamente, si $m_1, \ldots, m_k$ es una lista de enteros coprimos, M el producto de ellos y $A_1, A_2, \ldots, A_k$ son subconjuntos de enteros de la forma

$$A_i = \{a_{i,1}, \ldots, a_{i,\kappa_i}\} \subset \mathbb{Z}_{m_i},$$

los x del conjunto

$$A = \{x \in \mathbb{Z}_M \mid \forall i,\ \exists j,\ x \equiv a_{i,j} \mod m_i\}$$

(con $1 \leq i \leq k$, $1 \leq j \leq \kappa_i$) son el resultado de aplicar la biyección natural (definida por el teorema chino del resto) $\mathbb{Z}_{m_1} \oplus \mathbb{Z}_{m_2} \oplus \ldots \oplus \mathbb{Z}_{m_k} \to \mathbb{Z}_M$ a las k-uplas del producto cartesiano $\prod_{i=1}^{k} A_i$.

Como cada x_j de A puede escribirse como $a_{1,j_1}M_1 + a_{2,j_2}M_2 + \ldots + a_{k,j_k}M_k$ mód M para un conjunto de enteros $M_1, M_2, \ldots, M_k$, es posible construir A de forma inductiva:

$$\begin{array}{ll}
x_1 & = a_{1,1}M_1 + a_{2,1}M_2 + \ldots + a_{k,1}M_k \mod M \\
x_2 & = x_1 + (a_{1,2} - a_{1,1})M_1 \mod M \\
\ldots & \\
x_{\kappa_1} & = x_{\kappa_1 - 1} + (a_{1,\kappa_i} - a_{1,\kappa_i - 1})M_1 \mod M \\
x_{\kappa_1 + 1} & = x_{\kappa_1} + (a_{1,1} - a_{1,\kappa_i})M_1 + (a_{2,2} - a_{2,1})M_2 \mod M \\
x_{\kappa_1 + 2} & = x_{\kappa_1 + 1} + (a_{1,2} - a_{1,1})M_1 \mod M \\
\ldots &
\end{array} .$$

El procedimiento completo se detalla en el Algoritmo 3. La complejidad del tiempo de ejecución es del orden $O(|A| \log(M))$ siempre que $|A| > \sum_{i=1}^{k} \kappa_i \cdot \log M$. Un análisis completo del algoritmo puede consultarse en [13], Teorema 4.2.

Por las observaciones de la sección anterior, si c es el producto de $p_1, \ldots, p_k$ factores coprimos, es suficiente calcular T_{n,p_i}, los targets de n módulo p_i para $i = 1, \ldots, k$ y luego utilizar el algoritmo anterior para conseguir el conjunto $T_{n,c}$ de todos los targets de n módulo c.

Algoritmo 3 Algoritmo para construir eficientemente el conjunto resultante de aplicar el teorema chino del resto

Require: $m_i \in \mathbb{N}$ coprimos y conjuntos $A_i \subset \mathbb{Z}_{m_i}$ con $\kappa_i = |A_i|$ para $i = 1, \ldots, k$ y $\kappa_1 \geq \kappa_2 \geq ... \geq \kappa_k$.

1: Calcular $M \leftarrow m_1 m_2 \ldots m_k$, $card_A \leftarrow \kappa_1 \kappa_2 \ldots \kappa_k$.
2: **for** $i = 1, \ldots, k$ **do**
3: Calcular $c_i \leftarrow (M/m_i)^{-1} \mod m_i$, $M_i \leftarrow M c_i / m_i$.
4: Inicializar $r_i \leftarrow 0$.
5: Hacer $\Delta_{i,0} \leftarrow (a_{i,1} - a_{i,\kappa_i}) M_i \mod M$
6: **for** $l = 1, \ldots, \kappa_i - 1$ **do**
7: $\Delta_{i,l} \leftarrow (a_{i,l+1} - a_{i,l}) M_i \mod M$
8: **end for**
9: **end for**
10: $x_1 \leftarrow a_{1,1} M_1 + a_{2,1} M_2 + \ldots + a_{k,1} M_k \mod M$
11: $A \leftarrow \{x_1\}$
12: **for** $i = 1, \ldots, card_A - 1$ **do**
13: Calcular $r_1 \leftarrow r_1 + 1 \mod \kappa_i$ y hacer $\mu \leftarrow 1$.
14: **while** $r_\mu = 0$ **do**
15: $\mu \leftarrow \mu + 1$
16: $r_\mu \leftarrow r_\mu + 1 \mod \kappa_\mu$
17: **end while**
18: $x_{i+1} \leftarrow x_i + \sum_{j=1}^{\mu} \Delta_{i,r_i} \mod M$
19: $A \leftarrow A \cup \{x_{i+1}\}$
20: **end for**
21: **return** A.

Algoritmo 4 Cálculo de targets, residuos cuadráticos y raíces para n módulo un primo impar p

1: {Cálculo de residuos cuadráticos y raíces}
2: $RC \leftarrow \{(0,(0,0))\}$ {RC es un conjunto de elementos de la forma $(a,(\alpha_1,\alpha_2))$ con α_1,α_2 las raíces cuadradas de a módulo p, ordenado de menor a mayor por el primer elemento de los pares $(a,(\alpha_1,\alpha_2))$}
3: $T \leftarrow \emptyset$ { Conjunto de targets de n módulo p}
4: $R \leftarrow \emptyset$ { Conjunto de raíces para los targets de n módulo p}
5: **for** $i = 1,\ldots,(p-1)/2$ **do**
6: $a \leftarrow i^2 \mod p$
7: Agregar $(a,(i,p-i))$ a RC de forma que el conjunto resultante permanezca ordenado.
8: **end for**
9: **for** $i = 1,\ldots,(p-1)/2+1$ **do**
10: Sea $(a,(\alpha_1,\alpha_2))$ el i-ésimo elemento de RC.
11: $b \leftarrow a+n \mod p$.
12: **if** $b \in RC$ **then**
13: $T \leftarrow T \cup \{(a,b)\}$
14: $R \leftarrow R \cup \{(a,(\alpha_1,\alpha_2))\}$
15: **end if**
16: **end for**
17: **return** R,T

Concretamente, para un primo p, los targets de n módulo p pueden conseguirse con el Algoritmo 4.

Tanto la inserción en el paso 7 como la búsqueda en 12 pueden realizarse en una cantidad de operaciones del orden de $O(\log p)$ (puede hacerse una búsqueda binaria en el conjunto ordenado RC). De modo que la cantidad de operaciones del algoritmo es del orden $O(p\log(p)Mul(\log(p))$. Notar que el conjunto R que devuelve está ordenado según la primer componente de cada target.

Por otra parte, una vez que el conjunto de targets $T_{n,c}$ ha sido construido, (utilizando los Algoritmos 3 y 4), los conjuntos R que resultan de la ejecución de 4 para cada p_i, pueden guardarse llamándolos R_{p_i}. Utilizaremos el Algoritmo 5 para calcular las raíces cuadradas modulares de cada target de n módulo c.

El costo en operaciones del Algoritmo 5 es:

1. El bucle en la línea 3 tiene un costo total acotado por $O(\sum_{i=1}^{k}(\log\tau(n,p_i)Mul(\log p_i))$ que puede mayorizarse por $O(k\log\tau(n,p_k)Mul(\log p_k))$

Algoritmo 5 Algoritmo para obtener las raíces cuadradas módulo c de la primer componente de los targets

Require: $T_{n,c}$ el conjunto de targets de n módulo c, $R_{p_1}, R_{p_2}, \ldots, R_{p_k}$ el conjunto de las primeras componentes de los targets de n y sus raíces cuadradas módulo p_i

1: **for** cada target (a, b) en $T_{n,c}$ **do**
2: $S \leftarrow \emptyset$
3: **for** $i = 1, \ldots, k$ **do**
4: $a_i \leftarrow a \mod p_i$.
5: Buscar el elemento $(a_i, (\alpha'_1, \alpha'_2))$ en R_{p_i}.
6: $A_i \leftarrow \{\alpha'_1, \alpha'_2\}$.
7: **end for**
8: Sea $\{\alpha_1, \alpha_2, \ldots, \alpha_{2^k}\}$ el multiset resultado de aplicar el Algoritmo 3 con $A_1, \ldots, A_k$ y $p_1, \ldots, p_k$
9: Sea A el conjunto resultante de remover los elementos repetidos del multiset $\{\alpha_1, \alpha_2, \ldots, \alpha_{2^k}\}$.
10: $S \leftarrow S \cup \{(a, A\}$
11: **end for**
12: **return** S

2. La cantidad de operaciones para la línea 8 es de orden $O(2^k \log c)$ siempre que $2^k \geq 2k \log(c)$.

3. La cantidad de operaciones en el paso 9 es del orden de $O(2^k \log 2^k \log c) = O(2^k k \log c)$ (por ejemplo, utilizando heap-sort para ordernar en tiempo $O(2^k k)$ y luego remover los elementos repetidos en tiempo $O(2^k)$).

Considerando el bucle principal, el orden del tiempo de ejecución del Algoritmo 5 es $O(\tau(n, c)(k \log \tau(n, p_k) Mul(\log p_k) + 2^k \log c + 2^k k \log c))$.

Notando que para $c = p_{i_1} \ldots p_{i_k}$ se tiene

$$\begin{aligned} \tau(n, c) &= \tau(n, p_{i_1}) \ldots \tau(n, p_{i_k}) \\ &\leq \frac{p_{i_1} + 3}{4} \cdots \frac{p_{i_k} + 3}{4} = O(2^{-2k} c) \end{aligned}$$

la cota para el tiempo de ejecución del Algoritmo 5 es

$$\begin{aligned} &O(2^{-2k} c(k \log p_k Mul(\log p_k) + 2^k \log c + 2^k k \log c)) \\ &\quad = O(2^{-2k} ck \log p_k Mul(\log p_k) + 2^{-k} k \log c) \end{aligned} \tag{2.9}$$

Capítulo 3

Algoritmo de factorización de enteros con targets

3.1. El algoritmo

El procedimiento principal de esta sección es el Algoritmo 8. En lo que sigue haremos una exposición gradual del algoritmo, comenzando con versiones elementales que serán mejoradas a lo largo del texto.

Vamos a suponer que n es compuesto, producto de dos primos impares p, q. Si x^*, y^* es una solución no trivial a la ecuación diofántica

$$n + x^2 = y^2,$$

entonces supongamos que $|x^*| = \frac{1}{2}|p - q| < n^{1-\epsilon}$ con $0 < \epsilon < 1$.

La magnitud de un target está en relación inversa con el tamaño del espacio de búsqueda de la solución, eso puede verse fácilmente cuando se observa que para (a, b, c), un target de n, la solución x^* a la ecuación diofántica $n + x^2 = y^2$ es

$$x^{*2} = (\alpha + cz)^2$$

para α una raíz cuadrada de a módulo c. Aunque queda decir cómo crecen la cantidad de raíces cuadradas (se hará en la sección 3.2.1), para cada raíz la búsqueda es sobre $|z| \leq n^{1-\epsilon}/c$.

Asumiremos siempre que c es coprimo con n porque de lo contrario es sencillo obtener la factorización de n calculando $mcd(c, n)$. Podríamos elegir c grande, tomar un target (a, b, c) para n y buscar z entero, acotado por $n^{1-\epsilon}/c$ de forma que se satisfaga

$$n + (\alpha_{i,j} + cz)^2 \in (\mathbb{Z})^2. \tag{3.1}$$

para alguna $\alpha_{i,j}^2 \equiv a \mod c$ $(0 \leq \alpha_{i,j} < c)$. Si c es lo suficientemente grande, la cota es pequeña y aseguramos un tiempo de ejecución razonable.

El problema es que no sabemos cuál es el target correcto, (ni la raíz cuadrada correcta) y deberíamos en principio, probar cada target (a, b, c) hasta encontrar la solución. Si $w_C(n, a, c)$ es la cantidad de pasos de un algoritmo que resuelve (3.1) para un (a, b, c) dado, la complejidad del algoritmo está dominada por $\sum_{a \in T_{n,c}} w_C(n, a, c)$. Podemos suponer que la dificultad del problema 3.1 es similar para todos los targets (a, b, c) de n y acotada por una función de n y c, $m_C(n, c)$. Entonces, la cantidad de operaciones está dominada por $\tau(n, c)m_C(n, c)$.

Por el Teorema 3 en 2, sabemos que, si c es primo, $\frac{c-1}{4} \leq \tau(n, c) \leq \frac{c-1}{4} + 1$. Sea $c = n^\beta$, entonces

$$\frac{1}{4}n^\beta - 1 \leq \tau(n, c) \leq \frac{1}{4}n^\beta + 1$$

y en ese caso

$$\big(\frac{1}{4}n^\beta - 1\big)m_C(n, c) \leq \tau(n, c)m_C(n, c) \leq \big(\frac{1}{4}n^\beta + 1\big)m_C(n, c) \tag{3.2}$$

Es decir que para buscar la solución a $n + x^2 = y^2$ utilizando targets la cantidad de operaciones es

$$\tau(n,c)m_C(n,c) = O(n^\beta \cdot m_C(n,c))$$

Si el único procedimiento disponible para resolver (3.1) es probar con todos los z, entonces

$$\tau(n,c)m_C(n,c) = O(c \cdot n^{1-\epsilon-\beta}) = O(n^\beta \cdot n^{1-\epsilon-\beta}) = O(n^{1-\epsilon}).$$

Sin embargo, tomar c primo es una mala elección. Invertir un poco de trabajo en determinar un c compuesto, razonablemente grande reducirá el tiempo de ejecución.

Un resultado importante es que si $c = p_1 \dots p_m$ es un producto de primos consecutivos, entonces la cantidad de targets (a, b, c) posibles para un n dado es menor en orden de magnitud que c. Concretamente, cada primo que se agrega a c no incrementa $\tau(n, c)$ y c en la misma proporción (Teorema 4): el efecto de agregar un primo a c es aumentar la cantidad de targets en, aproximadamente, una cuarta parte de lo que se incrementa c.

Hagamos $c' \cdot c = p_1 \dots p_r p_{r+1} \dots p_m$ con $c = p_{r+1} \dots p_m$. Si queremos buscar x^* solución de $n + x^2 = y^2$ utilizando targets, podemos probar para cada target (a_i, b_i, c) de n y cada $\alpha_{i,j}$ raíz cuadrada de $a_i \mod c$, si

$$x_{i,j}(z) = \alpha_{i,j} + cz$$

es solución, o equivalentemente si $1 < gcd(\lfloor (x_{i,j}(z)^2 + n)^{1/2} \rfloor - x_{i,j}(z), n) < n$, para $|z| < n^{1-\epsilon}/c$.

Si además $\{(a'_\ell, b'_\ell, c')\}_{\ell=1}^{\tau(n,c')}$ son los targets de n módulo c' entonces $x_{i,j}(z) \equiv a'_\ell \mod c'$ para algún a'_ℓ. Más aún, si (a'_ℓ, b'_ℓ, c') es un target correcto para n, la solución x^* a la ecuación de Fermat es $x_{i,j}(z_1 + k \cdot c')$ para algún k entero, acotado y $0 \le z_1 \le c' - 1$ alguna raíz de $x_{i,j}(z)^2 \equiv a'_\ell \mod c'$. La cota para k puede determinarse fácilmente a partir de la correspondiente para z, en efecto

$$|k| \le |z|/c' \le n^{1-\epsilon}/(c \cdot c').$$

Entonces la idea es buscar x^* de la forma

$$x_{i,j}(z) = \alpha_{i,j} + c(z_1 + k \cdot c')$$

para todos los z_1 que sean raíces de $x_{i,j}(z)^2 \equiv a'_\ell \mod c'$ para algún target (a'_ℓ, b'_ℓ, c') de n. Esto se resume en el Algoritmo 6.

No vale la pena hacer un análisis completo del algoritmo anterior, porque pronto se verá que puede mejorarse. Sin embargo, es importante notar que

Algoritmo 6 Factorización de enteros usando targets, versión 1

Require: $\{(a_i, b_i, c), \ldots\}$ el conjunto (ordenado) de targets para n módulo c y $\{(a'_\ell, b'_\ell, c'), \ldots\}$ lo mismo módulo c'. Sea $\{\alpha_{i,j}\}_{j=0}^{\kappa_i}$ el conjunto ordenado de raíces cuadradas módulo c de a_i. Sea $0 < \epsilon < 1$ de modo que $|x^*| \leq n^{1-\epsilon}$.

$z_{max} \leftarrow \lceil \frac{n^{1-\epsilon}}{c} \rceil$
$k_{max} \leftarrow \lceil \frac{n^{1-\epsilon}}{c \cdot c'} \rceil$
for $i = 1, \ldots, \tau(n, c)$ **do**
  for $j = 1, \ldots, \kappa_i$ **do**
    Definir $x(z) = \alpha_{i,j} + cz$.
    for $\ell = 1, \ldots, \tau(n, c')$ **do**
      for $z_1 = 0, \ldots, c' - 1$ **do**
        if $x(z_1)^2 - a'_\ell \equiv 0 \mod c'$ **then**
          for $k = -k_{max}, \ldots, k_{max}$ **do**
            $x^2 \leftarrow x(z_1 + kc')^2$.{Búsqueda de la solución a partir de los puntos iniciales}
            $y \leftarrow \lfloor \sqrt{n + x^2} \rfloor$.
            $g \leftarrow |mcd(y - x, n)|$.
            if $1 < g < n$ **then**
              return $g, n/g$
            end if
          end for
        end if
      end for
    end for
  end for
end for
return $1, n$

hay como máximo 2^r enteros en $[0, c-1]$ que verifican la condición en la línea 8 y que para cada i, $\kappa_i \leq 2^m$.

En el paso 8 del algoritmo anterior, se busca z_1 para que el valor de x^2 correspondiente, o sea $(x_{i,j}(z_1))^2 = (\alpha_{i,j} + cz_1)^2$ sea congruente a la primer componente (denotada como a'_ℓ en el algoritmo) de un target de n, módulo c'.

Si llamamos $Z_{i,j}$ al conjunto de enteros $0 \leq z \leq c' - 1$ para los cuales $(\alpha_{i,j} + cz)^2 \equiv a'_\ell$ para algún target (a'_ℓ, b'_ℓ, c') entonces veremos que

$$Z_{i,j} = t_{i,j} + Z_{i,1} \mod c' = \{t_{i,j} + z \mod c' \mid z \in Z_{i,1}\} \tag{3.3}$$

para un $t_{i,j}$ que sólo depende de los targets.

En efecto, si $\alpha_{i,1}$ y $\alpha_{i,j}$ son raíces cuadradas de a_i módulo c y

$$(\alpha_{i,1} + cz_1)^2 \equiv a'_\ell \mod c'$$

entonces haciendo $t_{i,j} = d(\alpha_{i,1} - \alpha_{i,j})$, con d el menor entero positivo tal que $d \equiv c^{-1} \mod c'$ se tiene

$$(\alpha_{i,j} + c(z_1 + t_{i,j}))^2 \equiv a'_\ell \mod c'.$$

La aplicación $z \mapsto z + t_{i,j} \mod c'$ lleva puntos de $Z_{i,1}$ en puntos de $Z_{i,j}$. Eso significa que una vez hallados los puntos iniciales para una raíz $\alpha_{i,1}$, solo es necesario sumar $t_{i,j}$ a ese conjunto y tomar módulo para tener los puntos iniciales para $\alpha_{i,j}$. Las modificaciones al algoritmo original pueden verse en el Algoritmo 7, donde además se agrega el precálculo de $\alpha_{i,j} + ct_{i,j}$ para evitar calcular ese producto y esa suma en cada iteración.

De modo similar, para un i fijo, podemos trasladar los puntos en $Z_{i,1}$ a puntos en $Z_{j,1}$ para cualquier $1 \leq j \leq \tau(n, c)$, eso es

$$Z_{j,1} = \theta_j + Z_{i,1} \mod c' \tag{3.4}$$

para un θ_j que sólo depende de los targets.

Eso porque si $\alpha_{i,1}$ y $\alpha_{j,1}$ son dos raíces cuadradas modulares de a_i y a_j para dos targets distintos (a_i, b_i, c) y (a_j, b_j, c) entonces las parábolas inducidas por esas raíces son $(x_{i,1}(z))^2 = (\alpha_{i,1} + cz)^2$ y $(x_{j,1})^2 = (\alpha_{j,1} + cz)^2$. Para $\theta_j = d(\alpha_{i,1} - \alpha_{j,1})$ (con $d \equiv c^{-1} \mod c'$) se tiene

$$(x_{j,1}(z + \theta_j))^2 = (\alpha_{j,1} + cd(\alpha_{i,1} - \alpha_{j,1}) + cz)^2 \tag{3.5}$$

$$\equiv (\alpha_{i,1} + cz)^2 \mod c' \tag{3.6}$$

En particular, fijando $i = 1$ se tiene $\theta_j = d(\alpha_{1,1} - \alpha_{j,1})$ y

$$Z_{j,1} = \theta_j + Z_{1,1} \mod c' \tag{3.7}$$

Algoritmo 7 Factorización usando targets, versión 2

Require: $\{(a_i, b_i, c), \ldots\}$ el conjunto (ordenado) de targets para n módulo c y $\{(a'_\ell, b'_\ell, c'), \ldots\}$ lo mismo módulo c'. Sea $\{\alpha_{i,j}\}_{j=0}^{\kappa_i}$ el conjunto ordenado de raíces cuadradas módulo c de a_i. Sea $0 < \epsilon < 1$ (estimado) de modo que $|x^*| \leq n^{1-\epsilon}$.

1: **for** $i = 1, \ldots, \tau(n, c)$ **do**
2: **for** $j = 1, \ldots, \kappa_i$ **do**
3: $t_{i,j} \leftarrow (\alpha_{i,1} - \alpha_{i,j})(c^{-1} \mod c')$ {Precálculos}
4: $\delta_{i,j} \leftarrow \alpha_{i,j} + ct_{i,j}$
5: **end for**
6: **end for**
7: $z_{max} \leftarrow \lceil \frac{n^{1-\epsilon}}{c} \rceil$
8: $k_{max} \leftarrow \lceil \frac{n^{1-\epsilon}}{c \cdot c'} \rceil$
9: **for** $i = 1, \ldots, \tau(n, c)$ **do**
10: Definir $x(z) = \alpha_{i,1} + cz$.
11: **for** $\ell = 1, \ldots, \tau(n, c')$ **do**
12: **for** $z_1 = 0, \ldots, c' - 1$ **do**
13: **if** $x(z_1)^2 - a'_\ell \equiv 0 \mod c'$ **then**
14: $f \leftarrow 1$.
15: $\Delta \leftarrow c(z_1 + c'k)$.
16: **for** $k = -k_{max}, \ldots, k_{max}$ **do**
17: **for** $j = 1, \ldots, \kappa_i$ **do**
18: $x \leftarrow \delta_{i,j} + \Delta$
19: $y \leftarrow \lfloor \sqrt{n + x^2} \rfloor$.
20: $f \leftarrow f(y - x) \mod n$.
21: **end for**
22: $g \leftarrow |mcd(f, n)|$.
23: **if** $1 < g < n$ **then**
24: **return** $g, n/g$
25: **end if**
26: **end for**
27: **end if**
28: **end for**
29: **end for**
30: **end for**
31: **return** $1, n$

para $1 \leq j \leq \tau(n, c)$.

Por 3.3 y 3.7

$$Z_{i,j} = \theta_i + t_{i,j} + Z_{1,1} \quad \text{mód } c' \tag{3.8}$$

y sólo es necesario hallar, en el algoritmo, los θ_i, $t_{i,j}$ y el conjunto $Z_{1,1}$.

Sea $Z_{1,1} = \{z_1, z_2, \ldots, z_m\}$. Si $\alpha_{1,1}, \ldots, \alpha_{1,k_1}$ son las raíces cuadradas módulo c de $\alpha_{1,1}^2$ mód c, entonces para buscar un factor de n empezando con z_1 y utilizando los x de la forma $\alpha_{1,j} + cz_1$ puede calcularse

$$gcd(\prod_{j=1}^{k_0}[\lfloor (x_{1,j}^2(z_1) + n)^{1/2} \rfloor - x_{1,j}^2(z_1)], n) \tag{3.9}$$

con $x_{i,j}(z) = \delta_{i,j} + cz$, como en el paso 18 del Algoritmo 7.

En general, puede utilizarse cualquier z_l, $l = 1, \ldots, m$ y reemplazar z_1 por z_l para buscar un factor de n utilizando los x de la forma $\alpha_{1,j} + cz_l$. Además, por las consideraciones anteriores, reemplazando z_1 por $z_1 + \theta_i$ en 3.9 y utilizando el $\delta_{i,\cdot}$ correspondiente se tiene

$$gcd(\prod_{j=1}^{k_i}[((x_{i,j}^2(\theta_i + z_1) + n)^{1/2} - (x_{i,j}(\theta_i + z_1))], n) \tag{3.10}$$

que es la expresión para buscar los factores de n utilizando las parábolas inducidas por las raíces $\alpha_{i,1}, \ldots, \alpha_{i,k_i}$ del target (a_i, b_i) módulo c.

El Algoritmo 8 lleva a cabo estas ideas y es el que mejora a los Algoritmos 6 y 7.

Los puntos x, candidatos a la solución de la ecuación de Fermat $x^2+n = y^2$ generados por el algoritmo pueden ser ordenados en una matriz X de filas $R_1, \ldots, R_l$ donde cada fila representa a un z determinado y cada columna a una parábola inducida por una raíz $\alpha_{i,j}$ de a_i para un target (a_i, b_i, c) de n.

La primer fila es $R_1 = [r_{1,1}|r_{1,2}|\ldots|r_{1,l}]$ con

$$r_{1,1} = (\ \delta_{1,1} + c(\theta_1 + z_1) \quad \delta_{1,2} + c(\theta_1 + z_1) \quad \ldots \quad \delta_{1,k_1} + c(\theta_1 + z_1))$$

y en general,

$$r_{1,t} = (\ \delta_{t,1} + c(\theta_t + z_1) \quad \delta_{t,2} + c(\theta_1 + z_1) \quad \ldots \quad \delta_{t,k_t} + c(\theta_1 + z_1))$$

La fila siguiente es $R_2 = [r_{2,1}|r_{2,2}|\ldots|r_{2,l}]$ con

$$r_{2,t} = (\ \delta_{t,1} + c(\theta_t + z_1) \quad \delta_{t,2} + c(\theta_1 + z_1) \quad \ldots \quad \delta_{t,k_t} + c(\theta_1 + z_1))$$

Es decir, cada fila $R_i = [r_{i,1}|r_{i,1}|\ldots|r_{i,l}]$ es el resultado de reemplazar z_1 por z_i en la fila R_1.

El resto de las filas se forman reemplazando z_i por $z_i + k \cdot c'$ en cada una de las filas R_i, para $k_{min} \leq k \leq k_{max}$.

La dimensión de la matriz está directamente relacionada con la complejidad del algoritmo (ver la sección 3.2).

Algoritmo 8 Factorización usando targets

Require: n, $0 < \epsilon < 1$ y m, r enteros con $0 < r \leq m - 4$.

$c' \leftarrow p_1 \dots p_r$, $c \leftarrow p_{r+1} \dots p_m$

$z_{max} \leftarrow \lceil n^{1-\epsilon}/c \rceil$

$k_{max} \leftarrow \lceil n^{1-\epsilon}/(c \cdot c') \rceil$, $k_{min} \leftarrow -k_{max}$.

$\{(a_i, b_i, c)\}$ es el conjunto de targets para n módulo c (obtenido, por ejemplo con los targets de n módulo $p_{r+1}, \dots, p_m$ y el Algoritmo 3)

$\{(a'_j, b'_j, c')\}$ el conjunto de targets para n módulo c' (construido de forma similar).

Sean $\{\alpha_{i,1}, \dots, \alpha_{i,k_i}\}$ las raíces cuadradas módulo c de a_i para $i = 1, \dots, \tau(n, c)$ (obtenidas, por ejemplo con las raíces de a_i módulo $p_{r+1}, \dots, p_m$ y el Algoritmo 5).

$d \leftarrow c^{-1} \mod c'$

for cada target (a_i, b_i, c) **do**

  for $j = 1, \dots, k_i$ **do**

    $t_{i,j} \leftarrow (\alpha_{i,1} - \alpha_{i,j})d \mod c'$ {Precálculos}

    $\delta_{i,j} \leftarrow \alpha_{i,j} + ct_{i,j}$

  end for

  $\theta_i \leftarrow d(\alpha_{i,1} - \alpha_{1,1}) \mod c'$

end for

Sea $P(z) = (\alpha_{1,1} - cz)^2$.

$i \leftarrow 1$

for $\ell = 1, \dots, \tau_{n,c'}$ **do**

  Tomar (a'_ℓ, b'_ℓ, c'), el target ℓ de n módulo c'

  for $z = 0, \dots, c' - 1$ **do**

    if $(P(z) - a'_\ell) \mod c' = 0$ **then**

      $z_i \leftarrow z$, $i \leftarrow i + 1$ {Puntos iniciales}

    end if

  end for

  $u \leftarrow i - 1$

end for

$f \leftarrow 1$

for $s = 1, \dots, \tau_{n,c}$ **do**

  for $t = 1, \dots, k_s$ **do**

    Definir $x_{i,j} = \delta_{s,t} + c(\theta_s + c' \cdot j + z_i)$ {Búsqueda de la solución}

    $f \leftarrow \left(\prod_{i=1}^{u} \prod_{j=k_{min}}^{k_{max}} \lfloor (x_{i,j}^2 + n)^{1/2} \rfloor - x_{i,j} \right) \cdot f \mod n$

  end for

end for

$g \leftarrow |mcd(f, n)|$

return $(g, n/g)$

3.2. Análisis y complejidad

3.2.1. Estimación de la cantidad de raíces

En la etapa de búsqueda de la solución del Algoritmo 8 es necesario calcular el producto de tantos factores como raíces tiene un target. Por eso es importante saber cuántas raíces cuadradas modulares hay, en total para todos los targets. Para $c = p_1 \dots p_m$ y $a \in \mathbb{Z}_c$ hay un máximo de 2^m soluciones a $\alpha^2 \equiv a \mod c$ y podríamos tomar eso como una aproximación. Sin embargo, la estimación de 2^m raíces distintas para cada target $(a, b, p_1 \dots p_m)$ es bastante gruesa. Es importante ver que si c_k es la cantidad de targets (a, b, c) de n, con a divisible por exactamente k factores primos de c, entonces la cantidad de raíces cuadradas modulares $\alpha_{i,j}$ de a por cada uno de esos targets es 2^{m-k}. La cantidad de raíces distintas para buscar la solución x^* es

$$\mathcal{P}_c = 2^m c_0 + 2^{m-1} c_1 + \ldots + 2^{m-k} c_k + \ldots + c_m$$

Por otra parte, la cantidad de targets para n módulo c es

$$\tau(n, c) = c_0 + c_1 + \ldots + c_m$$

Lo anterior sugiere una generalización interesante: se puede definir el polinomio G, como

$$G_{n,c}(x) = \sum_{i=0}^{m} c_i x^{m-i} \tag{3.11}$$

y entonces

$$G_{n,c}(1) = \tau(n, c) \tag{3.12}$$

$$G_{n,c}(2) = \mathcal{P}_c \tag{3.13}$$

La construcción de $G_{n,c}$ es simple y tiene propiedades multiplicativas, como τ. El caso más simple es cuando c es primo y eso puede verse en el lema que sigue.

Lema 7. *Para $p > 2$ primo, que no divide a n el polinomio correspondiente $G_{n,p}$ es*

$$G_{n,p}(x) = c_0 x + c_1 = \begin{cases} \frac{p-1}{4} x + v(n) & \text{si } p \equiv 1 \mod 4 \\ \frac{p+1}{4} x & \text{si } p \equiv 3 \mod 4 \text{ y } (\frac{n}{p}) = -1 \\ \frac{p-3}{4} x + 1 & \text{si } p \equiv 3 \mod 4 \text{ y } (\frac{n}{p}) = 1 \end{cases}$$

con $v(n) = (1 + (\frac{n}{p}))/2$, c_0 la cantidad de targets para n de la forma (a, b, p) con $a \neq 0$ y c_1 la cantidad de targets para n de la forma $(0, b, p)$

Demostración. Si $p \equiv 1 \mod 4$ y $v(n) = 0$ entonces no hay ningún target de la forma $(0, b, p)$, y la cantidad de targets (a, b, p) con $a \neq 0$ es exactamente $\tau(n, p) = \frac{p-1}{4}$. En este caso, además, por la definición de $G_{n,p}$ tenemos

$$G_{n,p}(x) = c_0 x + c_1 = \frac{p-1}{4} x.$$

Si $p \equiv 1 \mod 4$ y $v(n) = 1$, hay exactamente un target de la forma $(0, b, p)$ y la cantidad de targets (a, b, p) con $a \neq 0$ es exactamente $\tau(n, p) - 1 = \tau(n, p) - v(n) = \frac{p-1}{4}$. Además, en este caso por definición resulta

$$G_{n,p}(x) = c_0 x + c_1 = \frac{p-1}{4} x + 1.$$

Si $p \equiv 3 \mod 4$ y $(\frac{n}{p}) = -1$ no hay ningún target de la forma $(0, b, p)$. La cantidad de targets (a, b, p) con $a \neq 0$ es exactamente $\tau(n, p) = \frac{p-3}{4} + 1 = \frac{p+1}{4}$. Por otra parte, por la expresión de $G_{n,p}$ en este caso

$$G_{n,p}(x) = c_0 x + c_1 = \frac{p+1}{4} x.$$

Por el contrario, si $p \equiv 3 \mod 4$ y $(\frac{n}{p}) = 1$ entonces hay exactamente un target de la forma $(0, b, p)$. El número de targets de la forma (a, b, p) con $a \neq 0$ es, entonces $\tau(n, p) - 1 = \frac{p-3}{4}$. En este caso además, por definición

$$G_{n,p}(x) = c_0 x + c_1 = \frac{p-3}{4} x + 1.$$

□

Proposición 1. *Para*

$$G_{n,p_{i_1}}(x) G_{n,p_{i_2}}(x) \dots G_{n,p_{i_k}}(x) = \sum_{j=0}^{k} c_j x^{k-j},$$

c_j es la cantidad de targets $(a, b, p_{i_1} p_{i_2} \dots p_{i_k})$ de n con a divisible por exactamente j factores primos de $p_{i_1} p_{i_2} \dots p_{i_k}$.

Demostración. Por inducción en k, la cantidad de factores primos.

El caso base, $k = 1$ se sigue del lema previo.

Para el paso inductivo, consideremos $G_{n,p_{i_1}}(x) G_{n,p_{i_2}}(x) \dots G_{n,p_{i_k}}(x) G_{n,p_{i_{k+1}}}(x)$. Por hipótesis inductiva,

$$G_{n,p_{i_1}}(x) G_{n,p_{i_2}}(x) \dots G_{n,p_{i_k}}(x) = \sum_{j=0}^{k} d_j x^{k-j}$$

con d_j la cantidad de targets $(a, b, p_{i_1} \dots p_{i_k})$ con a divisible por exactamente j factores primos de $p_{i_1} \dots p_{i_k}$ y

$$G_{n,p_{i_{k+1}}}(x) = e_0 x + e_1$$

con e_0 la cantidad de targets $(a, b, p_{i_{k+1}})$ con a coprimo con $p_{i_{k+1}}$ y e_1 aquellos para los cuales $a = 0$.

Finalmente,

$$\begin{aligned}
(G_{n,p_{i_1}}(x) \dots G_{n,p_{i_k}}(x))G_{n,p_{i_{k+1}}}(x) &= \Big(\sum_{j=0}^{k} d_j x^{k-j}\Big)(e_0 x + e_1) \\
&= \sum_{j=0}^{k} e_0 d_j x^{k+1-j} + \sum_{j=0}^{k} e_1 d_j x^{k-j} \\
&= e_0 d_0 x^{k+1} + \sum_{j=1}^{k} e_0 d_j x^{k+1-j} + \sum_{j=0}^{k-1} e_1 d_j x^{k-j} + e_1 d_k \\
&= e_0 d_0 x^{k+1} + \sum_{j=1}^{k} e_0 d_j x^{k+1-j} + \sum_{i=1}^{k} e_1 d_{i-1} x^{k+1-i} + e_1 d_k \\
&= e_0 d_0 x^{k+1} + \sum_{j=1}^{k} (e_0 d_j + e_1 d_{j-1}) x^{k+1-j} + e_1 d_k
\end{aligned}$$

Por el Teorema Chino del Resto, para $c = p_{i_1} \dots p_{i_k} p_{i_{k+1}}$:

- La cantidad de targets (a, b, c) de n con a coprimo con c es $c_0 = e_0 d_0$, (la cantidad de targets $(a, b, p_{i_1} \dots p_{i_k})$ de n con a coprimo con $p_{i_1} \dots p_{i_k}$ por la cantidad de targets $(a, b, p_{i_{k+1}})$ de n con a coprimos con $p_{i_{k+1}}$).
- La cantidad de targets (a, b, c) de n con a divisible por exactamente j factores primos de c es $c_j = e_0 d_j + e_1 d_{j-1}$.
- La cantidad de targets (a, b, c) de n con a divisible por c es $c_{k+1} = e_1 d_k$.

□

Ejemplo 1. *Para* $n = 10403$ *los targets de* n *módulo* 11 *son*

$$(1, 9), (3, 0), (4, 1)$$

como $\left(\frac{n}{11}\right) = -1$*, la definición de* G *en este caso da* $G_{n,11}(x) = c_0 x + c_1 = \frac{11+1}{4} x = 3x$.

Los targets de n módulo 13 *son*

$$(0,3),(1,4),(9,12),(10,0).$$

Como $\left(\frac{n}{13}\right) = 1$, *por la definición de* G *se tiene* $G_{n,13}(x) = d_0x + d_1 = \frac{13-1}{4}x + 1 = 3x + 1$.

Tenemos 12 *targets módulo* $11 \cdot 13 = 143$ *para* n*:*

$$\begin{aligned}&(1,108),(14,121),(23,130),(26,133),\\&(36,0),(48,12),(78,42),(91,55),\\&(92,56),(100,64),(113,77),(114,78).\end{aligned}$$

Como puede comprobarse, hay $c_0d_0 = 3 \cdot 3 = 9$ *targets* (a,b) *módulo* 143 *con* a *coprimo con* 143 *y* $c_0d_1 + c_1d_0 = 3 + 0$ *targets* (a,b) *módulo* $11 \cdot 13$ *con* a *divisible por exactamente un factor primo de* 143.

Podemos además, estimar el cociente $\mathcal{P}_c/c$. En efecto, por la definición de G,

$$\begin{aligned}\mathcal{P}_c/c &= \frac{G_{n,p_1}(2)\dots G_{n,p_m}(2)}{c}\\&\le \frac{1}{c}\Big(\frac{p_1+1}{2}\cdots\frac{p_m+1}{2}\Big)\\&= 2^{-m}\Big(\frac{p_1+1}{p_1}\cdots\frac{p_m+1}{p_m}\Big)\\&= O\Big(2^{-m}\log p_m\Big).\end{aligned} \tag{3.14}$$

(la última igualdad se sigue del Corolario 2.10 en [13]).

3.2.2. Tiempo de ejecución

Recordemos que $c' = p_1 \dots p_r$ es el producto de los primeros r primos consecutivos impares, y c el producto $p_{r+1} \cdots p_m$ de los primos consecutivos siguientes.

Primero se analizará la parte principal del algoritmo sin tener en cuenta el cálculo de los targets ni los precálculos. Es decir, analizaremos primero el bucle que comienza en 27.

La cantidad de operaciones que insumen la multiplicación y la raiz cuadrada entera de números de k bits se dejarán indicadas como $Mul(k)$ y $Sqrt(k)$.

Notemos que en el peor caso, para calcular la raiz cuadrada entera en un intervalo de longitud R se puede utilizar bisección para acotar la cantidad

de operaciones elementales por $\log(R)Mul(\log R)$. Aunque en la práctica se suelen emplear métodos como el de Newton o el de Goldschmidt (para aproximar una raíz cuadrada real y luego tomar parte entera, ver por ejemplo [9] y [10]) que son mucho más veloces dado que se estiman buenos valores iniciales para iterar mediante, por ejemplo, aproximaciones lineales (el método de Newton es usado en Python en la función *math.isqrt*)

La línea 30 tiene $u(k_{max} - k_{min}) + 1$ miembros en el producto. Los puntos iniciales $z_1, z_2, \ldots, z_u$ son raíces de a' para algún target (a', b', c') de n, hay $u = \mathcal{P}_{c'}$ de ellos. Además,

$$-k_{max} = k_{min} = \left\lceil \frac{n^{1-\epsilon}}{c \cdot c'} \right\rceil$$

por lo que el producto en el paso 30 tiene

$$2\mathcal{P}_{c'} \left\lceil \frac{n^{1-\epsilon}}{c \cdot c'} \right\rceil + 1$$

miembros y todos las multiplicaciones son módulo n.

La línea 30 se ejecuta para cada raíz cuadrada modular $\alpha_{i,j}$ de a para algún target (a, b, c) de n, es decir $\mathcal{P}_c$ veces. Entonces, la etapa de búsqueda se realiza en tiempo

$$O\left(\mathcal{P}_c \mathcal{P}_{c'} \left\lceil \frac{n^{1-\epsilon}}{c \cdot c'} \right\rceil \text{máx}\left(Mul(\log n), Sqrt(S \log n)\right) \right) \tag{3.15}$$

con $S = \text{máx}\,(1, 2(1-\epsilon))$ de modo que $S \log n$ es el tamaño máximo de los enteros en el cálculo de la raíz cuadrada entera en la línea 30.

Vamos a denotar como $T(n)$ a la cantidad de sumas, productos y raíces cuadradas enteras del bucle que comienza en la línea 27. Por el análisis previo

$$T(n) = O\Big(\mathcal{P}_c \mathcal{P}_{c'} \frac{n^{1-\epsilon}}{c \cdot c'}\Big) \tag{3.16}$$

siempre que $n^{1-\epsilon} \geq c \cdot c'$.

Como $\mathcal{P}_c \mathcal{P}_{c'} = \mathcal{P}_{c \cdot c'}$, por la ecuación 3.14 se tiene

$$T(n) = O(2^{-m} \log p_m n^{1-\epsilon}). \tag{3.17}$$

La expresión anterior depende de m, la cantidad de primos en el producto $c \cdot c'$, algo sobre lo cuál nada dijimos aún y trataremos en lo que sigue.

Si $k_{max} = \lceil n^{1-\epsilon}/(c \cdot c') \rceil = 1$, entonces $n^{1-\epsilon} \leq (c \cdot c')$ y agregar un factor más a $c \cdot c'$ deja constante k_{max} (y la diferencia $k_{max} - k_{min}$) mientras que incrementa $\mathcal{P}_c$.

Por lo tanto, el producto $c \cdot c'$ óptimo es aquel que minimiza $n^{1-\epsilon}/(c \cdot c')$ sujeto a

$$n^{1-\epsilon}/(c \cdot c') \geq 1. \tag{3.18}$$

La pregunta es cómo determinar m (la cantidad de primos) para satisfacer la desigualdad y que el lado izquierdo de ella sea mínimo.

Lo razonable es que m sea proporcional a la magnitud de n, es decir a $\log n$: podemos escribir

$$m = a \log n \tag{3.19}$$

para algún $0 < a < 1$. La magnitud del primo k-ésimo, p_k crece también con k, en consecuencia no puede asumirse a a constante.

El producto $c \cdot c' = p_1 \dots p_{a \log n}$ puede aproximarse como $e^{p_{a \log n}}$ (ver [23]) y tomando logaritmos en 3.18 la restricción es que

$$(1-\epsilon) \log n - p_{a \log n} \tag{3.20}$$

sea mayor o igual a 0.

Si se incorpora la constante $(1-\epsilon)$ haciendo $a \leftarrow a(1-\epsilon)$ la expresión anterior tiene la forma

$$(1-\epsilon) \log n - p_{a(1-\epsilon) \log n} = t - p_{at} \tag{3.21}$$

con $t = (1-\epsilon) \log n$. Al dividir por at y notar que $K \log p_{at} < p_{at}/(at)$ para una constante $K > 0$ (Corolario 1 en [21]), se tiene

$$\frac{1}{a} - \frac{p_{at}}{at} < \frac{1}{a} - K \log p_{at}. \tag{3.22}$$

Pero, por 3.20 y 3.21, $\log p_{at} \approx \log \Big((1-\epsilon) \log n \Big)$, es decir que

$$\frac{1}{a} - K \log p_{at} \approx \frac{1}{a} - K \log \Big((1-\epsilon) \log n \Big). \tag{3.23}$$

por lo tanto, una elección razonable para a es

$$a = \frac{K}{\log \Big((1-\epsilon) \log n \Big)}. \tag{3.24}$$

Puede verse en [21] que $K = 1/1{,}25506 \approx 0{,}79$ satisface 3.22.

Con las ecuaciones 3.15 y 3.14, tomando $m = \lfloor a \log n \rfloor$ y $r = \lfloor m/2 \rfloor$, es posible estimar

$$T(n) = O\Big(2^{-\lfloor a \log n \rfloor} \log p_{\lfloor a \log n \rfloor} n^{1-\epsilon} \Big) \tag{3.25}$$

o lo que es lo mismo

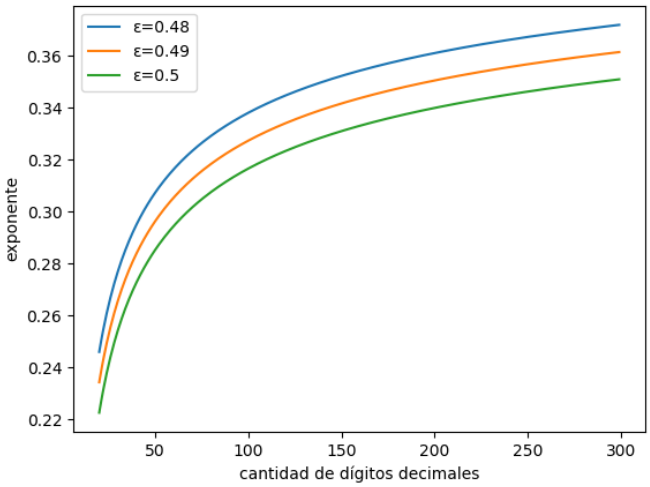
$$T(n) = O\Big(\log p_{\lfloor a\log n\rfloor} n^{1-\epsilon-a\log 2+o(1)}\Big) \tag{3.26}$$

y la complejidad estimada de esta sección del algoritmo es la cantidad de iteraciones por el máximo de lo que requiere multiplicar enteros de tamaño a lo sumo $\log(n)$ y tomar raíz cuadrada.

$$O\Big(\log p_{\lfloor a\log n\rfloor} n^{1-\epsilon-a\log 2+o(1)}\,\text{máx}\,(Mul(\log n), Sqrt(S\log n))\Big) \tag{3.27}$$

El comportamiento del exponente de n en la ecuación 3.27 con respecto a $\log n$ puede verse en la Figura 3.1. Por ejemplo, para n de 100 dígitos decimales y $\epsilon = 0{,}55$ el valor del exponente es aproximadamente 0,3309 (eso significa que el bucle del algoritmo que analizamos se realiza en $O(\log p_m n^{1/3})$ pasos).

Figura 3.1: Logaritmo de n vs logaritmo de la cantidad estimada de iteraciones, $\epsilon \approx 0{,}5$



Ahora nos ocuparemos del análisis del resto del algoritmo. El paso 7 puede ser realizado con el algoritmo extendido de Euclides y requiere $O(\log c \log c')$ operaciones. De modo similar, el paso 34 puede realizarse con el algoritmo de Euclides y necesita del orden de $O((\log n)^2)$ operaciones. Como el bucle en la línea 27 consume una cantidad de operaciones exponencial en $\log(n)$, podemos excluir los pasos 7 y 34 del análisis siguiente.

Vamos a asumir que m es lo suficientemente grande ($m \geq 34$, $r \geq 17$) de modo que se cumplan las Proposiciones 4 y 5 del Apéndice.

El Algoritmo 4 de la sección 2.5 es usado para construir el conjunto de targets y raíces cuadradas para cada primo p_i. La complejidad para cada primo es $O(p_i \log(p_i) Mul(\log p_i))$ por lo tanto, la complejidad total para construir T_{n,p_i} el conjunto de los targets de n módulo p_i y R_{p_i} las raíces cuadradas módulo p_i para $i = 1, \ldots, r$ es

$$O(\sum_{i=1}^{r} p_i \log p_i Mul(\log p_i)) = O(rp_r \log p_r Mul(\log p_r)) \tag{3.28}$$

y del mismo modo para $i = r+1, \ldots, m$ el costo total de construir los T_{n,p_i} es

$$O(\sum_{i=r+1}^{m} p_i \log p_i) = O((m-r)p_m \log p_m Mul(\log p_m)) \tag{3.29}$$

El Algoritmo 3 dado en la sección 2.5 puede utilizarse para construir el conjunto de targets de n módulo c y c', a partir de los conjuntos T_{n,p_i}, $i = r+1, \ldots, m$ y T_{n,p_j}, $j = 1, \ldots, r$.

Claramente $|T_{n,p_i}| = \tau(n, p_i)$ y el conjunto $T_{n,c}$ de todos los targets de n módulo c se obtiene con el Algoritmo 3 en una cantidad de operaciones acotada por $O(|T_{n,c}| \log c)$ o lo que es lo mismo $O(\tau(n,c) \log c)$, (ver Proposición 4 en el Apéndice).

Con el mismo razonamiento puede verse que el Algoritmo 3 construye los targets para n módulo c' (a partir de los targets para n y cada primo en c') en $O(\tau(n,c') \log c')$ operaciones.

El costo total de construir los targets de n módulo c es entonces

$$O((m-r)p_m \log p_m Mul(\log p_m) + \tau(n,c) \log c) \tag{3.30}$$

De modo similar, el costo total de construir los targets de n módulo c' es

$$O(rp_r \log p_r Mul(\log p_r) + \tau(n,c') \log c') \tag{3.31}$$

Para construir las raíces cuadradas modulares de los targets se utiliza el Algoritmo 5 cuyo costo de ejecución es del orden

$$O(2^{-2(m-r)} c(m-r) \log p_m Mul(\log p_m) + 2^{-(m-r)}(m-r) \log c)$$

(ver sección 2.5).

Como sumario, se tiene el siguiente cálculo de complejidad para el Algoritmo 8

1. La cantidad de operaciones en la línea 4 está acotada por

$$O((m-r)p_m \log p_m Mul(\log p_m) + \tau(n,c)\log c).$$

2. El costo de la línea 5 está acotado por

$$O(rp_r \log p_r Mul(\log p_r) + \tau(n,c')\log c').$$

3. Para la línea 6 la cantidad de operaciones está dominada por

$$O(2^{-2(m-r)}c(m-r)\log p_m Mul(\log p_m) + 2^{-(m-r)}(m-r)\log c).$$

4. La cantidad de operaciones en el bucle del paso 8 es

$$O(\mathcal{P}_c Mul(\log c))$$

(hay tantas iteraciones como raíces cuadradas de los targets de n módulo c y las operaciones son sobre enteros de $O(\log c)$ cantidad de dígitos)

5. El costo para el bucle de la linea 17 en cantidad de operaciones está dominado por

$$O(\tau(n,c')c' Mul(\log c'))$$

(todas las operaciones son módulo c')

6. La cantidad de operaciones en la etapa de búsqueda de la solución es

$$O\Big(2^{-m}\log(p_m)n^{1-\epsilon}\,\text{máx}\,(Mul(\log n), Sqrt(S\log n))\Big)$$

sujeto al parámetro ϵ. Como $p_m < c$ podemos escribir,

$$O\Big(2^{-m}\log(c)n^{1-\epsilon}\,\text{máx}\,(Mul(\log n), Sqrt(S\log n))\Big) \tag{3.32}$$

en lugar de la expresión anterior como cota para la cantidad de operaciones del bucle que comienza en la línea 27.

Es importante notar que para $c = p_{r+1} \dots p_m$ se tiene

$$\begin{aligned} \tau(n, c) &= \tau(n, p_{r+1}) \dots \tau(n, p_m) \\ &\leq \frac{p_{r+1}+3}{4} \dots \frac{p_m+3}{4} = O(2^{-2(m-r)}c) \end{aligned} \tag{3.33}$$

y para $c' = p_1 \dots p_r$

$$\begin{aligned} \tau(n, c') &= \tau(n, p_1) \dots \tau(n, p_r) \\ &\leq \frac{p_1+3}{4} \dots \frac{p_r+3}{4} = O(2^{-2r}c') \end{aligned} \tag{3.34}$$

Vamos a ver que todas las cotas pueden ser mayorizadas por 3.32. Recordemos que $m = \lfloor a \log n \rfloor$, $r = \lfloor m/2 \rfloor$

a. Como $2^m(m-r)p_m < 2^m m p_m \leq c \cdot c' \leq n^{1-\epsilon}$ se tiene que $(m-r)p_m \log p_m < (m-r)p_m \log(c) = O(2^{-m} \log p_m n^{1-\epsilon})$. Por otra parte, por 3.33

$$\tau(n, c) \log c = O(2^{-2(m-r)} c \log c) = O(2^{-m} c \log c)$$

y como $c < n^{1-\epsilon}$ se sigue que 3.32 domina a 1.

b. La cota 1 domina a 2 porque $p_r < p_m$ y $c' < c$. Por el punto anterior se sigue que 3.32 domina a 2.

c. Se tiene $c(m-r) \leq c(r+1) \leq c \cdot c' \leq n^{1-\epsilon}$ por lo tanto

$$2^{-m} c(m-r) \log p_m \leq 2^{-m} \log(c) n^{1-\epsilon}.$$

Por otra parte,

$$2^{-(m-r)}(m-r) \log c = 2^{-m} 2^r (m-r) \log c = O(2^{-m} \log(c) n^{1-\epsilon})$$

porque $2^r < c'$, $m - r < c$ y $c \cdot c' \leq n^{1-\epsilon}$. Por lo anterior, 3.32 domina a 3.

d. Notemos que $\mathcal{P}_c \leq \mathcal{P}_c \mathcal{P}_{c'}$. En virtud de la ecuación 3.16, 3.32 domina a 4.

e. Finalmente $\tau(n, c') c' Mul(\log c') = O(2^{-2r} c'^2 Mul(\log c'))$ por 3.34 y

$$2^{-2r} c'^2 \leq 2^{-m+1} c \cdot c' \leq 2^{-m+1} n^{1-\epsilon} \leq 2 \cdot 2^{-m} \log(c) n^{1-\epsilon}$$

con lo cual, 3.32 domina a 5.

Por lo tanto, para $m = \lfloor a \log n \rfloor$, $r = \lfloor m/2 \rfloor$, con $m \geq 34$ la cantidad de operaciones elementales del Algoritmo 8 está acotada por

$$O\Big(2^{-m} \log(c) n^{1-\epsilon} \operatorname{máx}(Mul(\log n), Sqrt(S \log n))\Big)$$

sujeto al parámetro $0 < \epsilon < 1$.

3.3. Alternativas algorítmicas

El Algoritmo 8 es trivialmente paralelizable: la búsqueda de la solución, el bucle en 27, puede dividirse para que cada proceso tome un par de valores s, t.

En la práctica, a cambio de utilizar la aproximación $m = a \log n$ para la base de factores, se construye la máxima lista de primos $p_1, \ldots, p_m$ cuyo producto no supere a $n^{1-\epsilon}$ y se toma $r = \lfloor m/2 \rfloor$.

En lugar de calcular $mcd(n, f)$ en la penúltima línea, podría calcularse después de cada producto en el paso 30 y devolver un factor de n tan pronto aparece, para evitar ir hasta el final del bucle.

Por otra parte, el producto $c \cdot c'$ puede completarse multiplicando por los primos pequeños que aparecen allí, de modo que $c \cdot c'$ pueda contener algunas potencias de primos y el cociente $n^{1-\epsilon}/(c \cdot c')$ disminuya.

Además puede conseguirse una mejora en el desempeño si se realiza una elección al azar de las raíces y los targets en los pasos 27 y 28 del Algoritmo 8.

Los resultados numéricos de estas alternativas pueden verse en los cuadros de la sección 3.4.

Si la diferencia entre los factores de n es muy grande (ϵ muy pequeño), el tiempo de ejecución de los algoritmos anteriores se ve afectado; después de todo la cantidad de iteraciones está en función de ϵ. Este caso es el equivalente del método de Fermat versus el método de Lehman, presentados en la sección de antecedentes.

El conjunto de valores para x en la línea 29 del Algoritmo 8 es un conjunto de posibles soluciones para x, como los calculados en el paso 3 del algoritmo de Fermat, que contiene a una solución de $n + x^2 = y^2$ relacionada con los factores no triviales de n.

Las líneas 10 y 16 del algoritmo de Lehman pueden reemplazarse con llamadas al Algoritmo 8 con los parámetros adecuados, como se ve en el Algoritmo 10. En ese caso, el procedimiento de factorización con targets es reemplazado por una versión modificada consignada en el Algoritmo 9.

Es importante notar que cuando k es coprimo con $c \cdot c'$ y $b_{max} - b_{min}$ es lo suficientemente grande, el mismo análisis de complejidad de la sección 3.2 aplica para el Algoritmo 9. Sin embargo, la desventaja es tener que calcular los targets y las raíces cuadradas modulares en cada iteración. Los resultados pueden verse en los cuadros 3.3 y 3.4 de la sección 3.4.

.

Algoritmo 9 Búsqueda de y^*

Require: $b_{min} < b_{max}$, k un multiplicador y n, el número a factorizar.

Sea $m = \lfloor a \log(b_{max} - b_{min}) \rfloor$, $r = \lfloor m/2 \rfloor$.
Sea $c' = p_1 \dots p_r$ y $c = p_{r+1} \dots p_m$.
$c' \leftarrow c'/mcd(4k, c')$ {c' debe ser coprimo con $4kn$}
$c \leftarrow c/mcd(4k, c)$ {c debe ser coprimo con $4kn$}
if $c = 1$ o $c' = 1$ **then**
  for $b = b_{min}, \dots, b_{max}$ **do**
    if $b^2 - 4kn$ es un cuadrado perfecto **then**
      $a \leftarrow \sqrt{b^2 - 4kn}$
      $g \leftarrow mcd(a + b, n)$
      return $g, n/g$
    end if
  end for
end if
{(a_i, b_i, c)} es el conjunto de targets para $4kn$ módulo c.
{(a'_j, b'_j, c')} el conjunto de targets para $4kn$ módulo c'.
$c_{inv} \leftarrow c^{-1}$ mód c'
for cada target (a_i, b_i, c) **do**
  Sean $\{\beta_{i,1}, \dots, \beta_{i,k_i}\}$ las raíces cuadradas módulo c de b_i.
  for $j = 1, \dots, k_i$ **do**
    $t_{i,j} \leftarrow (\beta_{i,1} - \beta_{i,j}) c_{inv}$ mód c' {Precálculos}
    $\delta_{i,j} \leftarrow \beta_{i,j} + c t_{i,j}$
  end for
  $\theta_i \leftarrow c_{inv}(\beta_{i,1} - \beta_{1,1})$ mód c'
end for
Sea $P(z) = (\beta_{1,1} - cz)^2$.
$i \leftarrow 1$
for $\ell = 1, \dots, \tau_{n,c'}$ **do**
  Tomar (a'_ℓ, b'_ℓ, c'), el target ℓ de n módulo c'
  for $z = 1, \dots, c' - 1$ **do**
    if $(P(z) - b'_\ell)$ mód $c' = 0$ **then**
      $z_i \leftarrow z$, $i \leftarrow i + 1$ {Puntos iniciales}
    end if
  end for
  $u \leftarrow i - 1$
end for
$k_{max} \leftarrow \lceil b_{max}/(c \cdot c') \rceil$, $k_{min} \leftarrow \lfloor b_{min}/(c \cdot c') \rfloor$
$f \leftarrow 1$
for $s = 1, \dots, \tau_{n,c}$ **do**
  for $t = 1, \dots, k_s$ **do**
    Definir $y_{i,j} = \delta_{s,t} + c(\theta_s + c' \cdot j + z_i)$ {Búsqueda de la solución}
    $f \leftarrow (\prod_{i=1}^{u} \prod_{j=k_{min}}^{k_{max}} \lfloor (y_{i,j}^2 - 4kn)^{1/2} \rfloor + y_{i,j}) \cdot f$ mód n
  end for
end for
$g \leftarrow |mcd(f, n)|$
return $(g, n/g)$

Algoritmo 10 Algoritmo de Lehman modificado

Require: $0 < \epsilon \leq 1$.

1: **for** $k = 1, \ldots, \lceil n^{1/3} \rceil$ **do**
2: **if** $k \mid n$ **then**
3: **return** $k, n/k$
4: **end if**
5: **end for**
6: $k \leftarrow 1$
7: **while** $k < \lfloor n^{1/3} \rfloor$ **do**
8: $b_{min} \leftarrow 2\lceil (kn)^{1/2} \rceil$
9: $b_{max} \leftarrow \lfloor 2(kn)^{1/2} + n^{1/6}/(4k^{1/2}) \rfloor$
10: Ejecutar el Algoritmo 9 con b_{min} y b_{max}. Sea p, q el resultado de la ejecución.
11: **if** $1 < p < n$ **then**
12: **return** p, q.
13: **else**
14: $k \leftarrow k + 1$
15: **end if**
16: **end while**

3.4. Ejemplos numéricos

Los experimentos numéricos tienen el objetivo de mostrar cómo funcionan algunas alternativas algorítmicas, aún cuando en la práctica su ejecución pueda ser más lenta comparada con las implementaciones de otros algoritmos.

Las pruebas se corrieron en una PC de escritorio, con procesador Intel i7 con 8 Gb de memoria RAM y sistema operativo Ubuntu.

Cuadro 3.1: Resultados del Algoritmo 8 para números de 24 dígitos, $\epsilon = 0{,}45$. Se permiten potencias de primos pequeños en el producto $c \cdot c'$ y las raíces se eligen al azar. Se calcula $mcd(n, f)$ para etapas intermedias y se sale cuando se encuentra un factor no trivial. $\log_n it$ es el logaritmo en base n de la cantidad de veces que se ejecuta la línea 30. t_1 es el tiempo que toma calcular los targets, las raíces cuadradas modulares y los precálculos. t_{total} es el tiempo total de ejecución

n	$\log_n(it)$	factor	t_1(seg.)	t_{total}(seg.)
589624680386966668089551	0,3435	965584647161	1,01	145,15
708425399280292823538587	0,3374	967017426253	1,95	116,80
681419919378281072833579	0,3444	694149461443	0,90	163,82
635471613344782021729459	0,3349	829693018471	0,92	97,93
464220680256868165917991	0,3468	595288518953	1,07	164,02
442146710282730469791227	0,3451	538590603443	1,83	147,88
503895810239521161796259	0,3236	803879951009	0,98	46,81
563083876736998960324733	0,3199	894410459221	1,90	41,73
707313111580418105889887	0,3380	964016348039	1,82	118,73
654099769535329328117717	0,2968	830391385217	1,86	13,87
605340337418462978810281	0,3319	716966128643	0,92	79,36
545481278034448888743691	0,3364	657334455499	0,92	100,04
261814780318262875512763	0,3095	513255708803	0,69	19,22
401031652227581800294757	0,3452	593933542049	2,01	142,38
555738499644344367122351	0,3443	998979862471	1,05	152,21
531080606243537586467831	0,3440	688695309187	0,90	149,69
539785657478159743892971	0,3215	890889696293	0,90	43,84

Cuadro 3.2: Resultados del Algoritmo 8 para números de 25 dígitos, $\epsilon = 0{,}5$. Se permiten potencias de primos pequeños en el producto $c \cdot c'$ y las raíces se eligen al azar. Se calcula $mcd(n, f)$ para etapas intermedias y se sale cuando se encuentra un factor no trivial. it es la cantidad de veces que se ejecuta la línea 30. $\log_n it$ es el logaritmo en base n de la cantidad de veces que se ejecuta la línea 30.

n	$\log_n(it)$	factor	it.	tiempo(seg.)
682477184971355479212149	0,3517	880630149337	241441560	185,529181957
811186075190619841909829	0,3303	817407119611	79337271	64,476211071
370540697598358482257837	0,3361	623898314033	83578179	65,8998150826
549633956189670612148117	0,3385	649658824697	109144659	85,5851099491
681363604359254974897211	0,3348	753312539251	95562141	76,9818730354
338052836707814238778067	0,3528	506453050637	200203164	156,666330099
514063929456372672901933	0,3408	834180850267	120731286	94,4620101452
346664546699507037228947	0,3445	604859459929	129185397	101,933067083

Cuadro 3.3: Resultados para números de 20 dígitos, del Algoritmo 10. it es la cantidad de veces que se ejecuta la línea 41 $\log_n(x^*)$ es el logaritmo en base n de x^* la solución a la ecuación de Fermat

n	$\log_n(it)$	factor	it.	tiempo(seg.)	$\log_n x^*$
55992839498077043761	0,3356	1592192093	4257318	21,8491721153	0,5330
75905773789412497759	0,2913	5638419299	620958	2,81703400612	0,4976
83107941138079923953	0,3385	3585558829	5540984	28,598952055	0,5166
47810309339526944893	0,2913	1962398279	542470	2,46533298492	0,5259
94832308940796146119	0,3236	5303437613	2915748	14,5576331615	0,5055
78345511916547043499	0,3178	9382738031	2101153	10,3880369663	0,4531
15144938807001066829	0,3100	2226745783	883999	4,29500198364	0,5036
34273425094887270233	0,3188	9814024637	1690673	8,36893987656	0,5017
53005605203100366811	0,3353	35585742139	4111041	21,0445420742	0,5339
66109570157129872783	0,2734	7677747881	263196	1,09171104431	0,4525

Cuadro 3.4: Resultados para números de 22 dígitos, del Algoritmo 10. $\log_n x^*$ es el logaritmo en base n de la solución no trivial x^* de la ecuación de Fermat

n	factor	tiempo(seg.)	$\log_n x^*$
2621780906943286872857	773048985493	59,6722872257	0,5549
3847769114533845365339	40522611061	42,6600549221	0,4973
654097612050973924159	676411733707	58,6144869328	0,5683
3627210961442967215491	87721591337	102,500969172	0,4947
3713623044286754450503	49684198159	30,9010460377	0,4821
9662423389533821358769	15030583193	154,032242775	0,5366
2033634884361379268789	109936927253	41,3747410774	0,5144
5611533409206893502629	54196217927	19,2871549129	0,4916
3492255446035043878219	10237631587	105,882713795	0,5347
5621360962748906064257	347474951231	27,1270010471	0,5296

Cuadro 3.5: Resultados para números de 22 dígitos, del Algoritmo 10. $\log_n x^*$ es el logaritmo en base n de la solución no trivial x^* de la ecuación de Fermat

n	factor	tiempo(seg.)	$\log_n x^*$
2621780906943286872857	773048985493	59,6722872257	0,5549
3847769114533845365339	40522611061	42,6600549221	0,4973
654097612050973924159	676411733707	58,6144869328	0,5683
3627210961442967215491	87721591337	102,500969172	0,4947
3713623044286754450503	49684198159	30,9010460377	0,4821
9662423389533821358769	15030583193	154,032242775	0,5366
2033634884361379268789	109936927253	41,3747410774	0,5144
5611533409206893502629	54196217927	19,2871549129	0,4916
3492255446035043878219	10237631587	105,882713795	0,5347
5621360962748906064257	347474951231	27,1270010471	0,5296

Capítulo 4

Discusión final

Como el lector habrá podido ver, el método propuesto para atacar el problema de la factorización de enteros hace uso de una idea simple, pero poco explorada. Sólo en el método de Rubinstein se ve una cierta conexión con las ideas propuestas y la pregunta que persiste es si esa poca atención se debe a la falta de atractivo sobre las posibilidades que puede ofrecer este abordaje o sólo a la poca promoción que ha recibido hasta el momento.

Si bien es cierto que comparativa y prácticamente, las ventajas que pueda reportar usar algunos de los algoritmos del capítulo 3 no son significativas, en muchos casos tampoco lo fueron las que ofrecían en su momento otros algoritmos hoy muy populares. Por eso merece la pena indagar sobre las posibilidades aún no desarrolladas del método.

Por una parte, el algoritmo es esencialmente determinista y exceptuando alguna elección aleatoria que se realice sobre la forma de escoger las raíces y los targets, el método de búsqueda de la solución termina siendo una búsqueda exhaustiva sobre un espacio, que a pesar de ser reducido, aún es demasiado grande (la matriz al final de la sección 3.1). Alguna forma de recorrer aleatoriamente ese espacio podría ser de ayuda si la longitud de ese recorrido es mucho menor que el espacio total y se garantiza que la solución será visitada.

Por otra parte, el espacio de las posibles soluciones merecería ser considerado en el marco del álgebra lineal o la teoría de retículos, lo que permitiría utilizar la gran variedad de algoritmos disponibles en esas áreas para resolver el problema de la búsqueda de la solución.

En la sección 3.3, el análisis del Algoritmo 10 involucra estimar la cantidad de números no-smooths en intervalos variables. Además de ser un asunto no muy simple, suele requerir (hasta el momento) asumir como ciertas algunas conjeturas y vuelve el análisis heurístico. El Algoritmo 10 podría representar una mejora significativa al método de Lehman y quizás por ese mismo hecho, valga la pena un análisis más profundo. En este punto no se pueden invocar resultados numéricos pues, de ser una mejora, sólo sería notable para números relativamente grandes y en esos casos el procedimiento original de Lehman y el propuesto aquí requieren mucho tiempo de cómputo.

Finalmente y en el aspecto teórico, es necesario mencionar que el problema de conocer para un n determinado si un target (a, b, c) es correcto, sin necesidad de factorizar n, podría estar conectado con problemas largamente estudiados y cuya complejidad es conocida, lo que daría teóricamente, un límite para las posibilidades del algoritmo.

Capítulo 5

Apéndice

Proposición 2. *Para $k \geq 17$,*

$$2\log p_k \leq \frac{1}{4}\Big(\frac{p_{k-2}-1}{4}\Big)\Big(\frac{p_{k-3}-1}{4}\Big)\Big(\frac{p_{k-4}-1}{4}\Big)$$

Demostración. Primero notemos algunos hechos sobre primos e índices: para $i \geq 1$ se tiene $p_{i+1} < 2p_i$.

Además para $i \geq 4$ se cumple

$$\frac{p_i^2-1}{p_i} \geq 2^3 \tag{5.1}$$

(notar que el lado izquierdo de la desigualdad crece con i) de allí se sigue que

$$p_i < (p_{i-2}-1)(p_{i-3}-1) \tag{5.2}$$

para $i \geq 7$. En efecto,

$$\begin{aligned} p_i &< 2p_{i-1} < 2^2 p_{i-2} < 2^3 p_{i-3} \\ &\leq p_{i-3}^2 - 1 && \text{por 5.1} \\ &= (p_{i-3}+1)(p_{i-3}-1) \\ &= (p_{i-3}+2-1)(p_{i-3}-1) \\ &\leq (p_{i-2}-1)(p_{i-3}-1). \end{aligned}$$

Por otra parte para $i \geq 13$

$$(p_i - 1)(i+4) \geq 1{,}25506 \cdot 2^9 \tag{5.3}$$

(el lado izquierdo de la desigualdad crece con i).

Sea $C = 1{,}25506$, el Corolario 1 en [21] afirma

$$k < C \cdot p_k / \log p_k \tag{5.4}$$

para $k \geq 1$ o lo que es lo mismo

$$\log p_k < C \cdot p_k / k \tag{5.5}$$

Por otra parte, si $k \geq 17$ entonces

$$\begin{aligned} 2^9 \cdot C \cdot p_k &\leq (p_{k-2}-1)(p_{k-3}-1)(2^9 \cdot C) && \text{por (5.2)} \\ &\leq (p_{k-2}-1)(p_{k-3}-1)(p_{k-4}-1)k && \text{por (5.3)} \end{aligned}$$

de allí se sigue

$$2\log p_k \leq 2 \cdot C \cdot p_k/k \leq \frac{1}{4}\frac{(p_{k-2}-1)}{4}\frac{(p_{k-3}-1)}{4}\frac{(p_{k-4}-1)}{4} \tag{5.6}$$

□

Proposición 3. *Para $k \geq 17$, $1 < r \leq k-4$ y $mcd(p_r \dots p_k, n) = 1$ se tiene*

$$\tau(n, p_r \dots p_k) \geq \sum_{i=r}^{k} (\tau(n, p_i) \log p_i)$$

Demostración. Notemos que

$$\prod_{i=r}^{k} \frac{p_i - 1}{4} \leq \prod_{i=r}^{k} \tau(n, p_i) = \tau(n, p_r \dots, p_k) \tag{5.7}$$

porque τ es multiplicativa y $\tau(n, p_i) \geq \frac{p_i - 1}{4}$ para $i > 1$ y cualquier $n \in \mathbb{Z}$ con $mcd(p_i, n) = 1$.

Por otra parte,

$$\sum_{i=r}^{k} (\tau(n, p_i) \log p_i) \leq \sum_{i=r}^{k} \left(\left(\frac{p_i + 3}{4} \right) \log p_i \right) \tag{5.8}$$

porque $\tau(n, p_i) \leq \frac{p_i + 3}{4}$ para $i > 1$ y cualquier $n \in \mathbb{Z}$ con $mcd(p_i, n) = 1$.

Además,

$$\begin{aligned}
\sum_{i=r}^{k} \left(\left(\frac{p_i + 3}{4} \right) \log p_i \right) &\leq (k - r) \frac{p_k + 3}{4} \log p_k \\
&\leq k \frac{p_k + 3}{4} \log p_k \\
&\leq (p_{k-1} - 1) \frac{p_k + 3}{4} \log p_k \\
&= (p_{k-1} - 1)(\frac{p_k - 1}{4} + 1) \log p_k \\
&\leq 2(p_{k-1} - 1) \frac{p_k - 1}{4} \log p_k \\
&= \frac{p_k - 1}{4} \frac{p_{k-1} - 1}{4} \frac{p_{k-2} - 1}{4} \frac{p_{k-3} - 1}{4} \frac{p_{k-4} - 1}{4} \quad \text{Prop. (2)} \\
&\leq \prod_{i=r}^{k} \frac{p_i - 1}{4}.
\end{aligned} \tag{5.9}$$

De 5.7, 5.8 y 5.9 se sigue

$$\sum_{i=r}^{k} (\tau(n, p_i) \log p_i) \leq \sum_{i=r}^{k} \left(\left(\frac{p_i + 3}{4} \right) \log p_i \right) \leq \prod_{i=r}^{k} \frac{p_i - 1}{4} \leq \tau(n, p_r \dots, p_k) \tag{5.10}$$

□

Proposición 4. *Para c como en la línea 1 del Algoritmo 8, siempre que $m \geq 17$ el Algoritmo óptimo para el teorema chino del resto de la sección 2.5 (Algoritmo 3) con los conjuntos $A_i = T_{n,p_i}$ y $m_i = p_i$ para $i = r+1, \ldots, m$, consume una cantidad de operaciones elementales acotada por $O(\tau(n,c)\log c)$ para cualquier $n \in \mathbb{Z}$ coprimo con c.*

Demostración. Consecuencia inmediata del Teorema 4.2 en [13] y la Proposición 3 □

Proposición 5. *Para c como en la línea 1 del Algoritmo 8, para $m > 10$ la cantidad de operaciones de la línea 8 del Algoritmo 5 es de orden $O(2^(m - r)\log c)$*

Por el Teorema 4.2 en [13], basta verificar que $2^{m-r} \geq 2(m-r)\log c$ o lo que es lo mismo (puesto que $(m-r) \geq r+1$), que $2^k \geq 2k\log c$, para que la proposición sea cierta. No hemos sido capaces aún de probar la desigualdad anterior, pero ofrecemos la Tabla 5.1 de los primeros 20 valores de 2^k y $2k\log c$ como evidencia:

Cuadro 5.1: Crecimiento de 2^k y $2k\log c$

k	2^k	$2k\log c$
2	4	10.832
3	8	27.923
4	16	68.145
5	32	113.51
6	64	189.86
7	128	268.64
8	256	388.61
9	512	504.03
10	1024	664.30
11	2048	818.08
12	4096	1027.4
13	8192	1222.3
14	16384	1476.5
15	32768	1713.0
16	65536	2018.1
17	131072	2299.7
18	262144	2655.1
19	524288	2980.2
20	1048576	3379.1

Bibliografía

[1] D. Aggarwal and U. Maurer, Breaking RSA Generically Is Equivalent to Factoring, Advances in Cryptology - EUROCRYPT 2009, Lecture Notes in Computer Science, vol. 5479, pp. 36-53.

[2] R. B. Ash, *Basic Abstract Algebra: For Graduate Students and Advanced Undergraduates*, Dover (2006).

[3] R. B. Ash, *A Course in Algebraic Number Theory*, Dover (2010).

[4] E. Bach, J. Shallit, *Algorithm Number Theory: Efficient Algorithms*, MIT Press (1996).

[5] D. Boneh and R. Venkatesan, Breaking RSA may not be equivalent to factoring, Advances in Cryptology-EUROCRYPT 1998, Lecture Notes in Computer Science, vol. 1403 (1998), pp. 59–71.

[6] J-S. Coron, A. May, *Deterministic polynomial time equivalence of computing the RSA secret key and factoring*, IACR Cryptology ePrint Archive, 2004:208, 2004.

[7] R. Crandall, C. Pomerance, *Prime numbers. A computational perspective*, Springer (2000).

[8] J. D. Dixon, *Asymptotically fast factorization of integers*. Mathematics of Computation, 36 (1981), no. 153, pp. 255-60. doi:10.1090/s0025-5718-1981-0595059-1

[9] M. D. Ercegovac, L. Imbert, D. W. Matula, J.-M. Muller & G. Wei, *Improving Goldschmidt division, square root, and square root reciprocal*. IEEE Transactions on Computers, 49(7), (2000), pp. 759–763

[10] R. E. Goldschmidt, *Applications of Division by Convergence*, MSc dissertation, M.I.T., 1964

[11] G. Hardy, E. M. Wright, *An introduction to the Theory of Numbers*, 5ta. ed, Oxford University Press, New York, (1979).

[12] G. A. Hiary, *A Deterministic Algorithm for Integer Factorization*, Mathematics of Computation, 85 (2015), no. 300, pp 2065-2069.

[13] M. Hittmeir, *A Babystep-Giantstep Method for Faster Deterministic Integer Factorization*, Mathematics of Computation, 87 (2018), no. 314, pp 2915-2935.

[14] K. Ireland, M. Rosen, *A Classical Introduction to Modern Number Theory*, Springer, 1982.

[15] J. Katz, Y. Lindell, *Introduction to Moderm Cryptography and Network Security*, 2da edición, CRC, (2015).

[16] T. Kleinjung et al. , *A heterogeneous computing environment to solve the 768-bit RSA challenge*, Cluster Computing, 15 (2010), no. 1, pp. 53–68. doi:10.1007/s10586-010-0149-0

[17] A. K. Lenstra, H. W. Lenstra, Jr. (eds.)n *The development of the number field sieve*. Lecture Notes in Math. (1993) 1554. Springer-Verlag.

[18] A. K. Lenstra, *Integer Factoring*, Designs, Codes and Cryptography, 19, 101–128 (2000).

[19] J. M. Pollard, *A Monte Carlo Method for Factorization*, BIT 15 (1975), pp. 831-884.

[20] R. L. Rivest, A. Shamir, and L. Adleman, A method for obtaining digital signatures and public-key cryptosystems, Commun. ACM 21, 2 (February 1978), pp. 120-126.

[21] Rosser, J. B. and Schoenfeld, L. *Approximate Formulas for Some Functions of Prime Numbers.* Illinois J. Math. 6, 64-97, 1962.

[22] M. Rubinstein, *Distribution of Solutions to $xy = N$ mod a with an application to factoring integers*, https://arxiv.org/abs/math/0610612v5

[23] S. M. Ruiz, *A Result on Prime Numbers*, Math. Gaz. 81, 269, 1997.

[24] H. D. Scolnik, Nuevos Algoritmos de Factorización de Enteros para atacar RSA, X Reunión Española sobre Criptología y Seguridad de la Información, Universida de Salamanca (Julio, 2008), available at `http://campus.usal.es/~xrecsi/Imagenes/conferenciantes/Scolnik.pdf` (accedido el 30 de Julio de 2019).

[25] P. E. Shor, *Polynomial-time Algorithms for Prime Factorization and Discrete Logarithms on Quantum Computer*, SIAM J. Comput., no. 5, pp. 1484-1509,(1997).

[26] V. Shoup, *A Computational Introduction to Number Theory and Algebra*, Cambridge University Press, Cambridge, 2005.

[27] I. E. Shparlinski, A. Winterhof, *On the number of distances between the coordinates of points on modular hyperbolas*, *Journal of Number Theory*, 128 (2008), pp 1224-1230

[28] S. Y. Yan, *Computational Number Theory and Modern Cryptography*, Wiley (2013).