

**ESCUELA TÉCNICA SUPERIOR
DE INGENIEROS DE TELECOMUNICACIÓN**

**Departamento de Física Aplicada
a las Tecnologías de la Información**



TESIS DOCTORAL

**Números primos especiales
y sus aplicaciones criptográficas**

AUTOR

José Raúl Durán Díaz
Licenciado en Ciencias (Físicas)

DIRECTORES

Dr. FAUSTO MONTOYA VITINI
Dr. JAIME MUÑOZ MASQUÉ

MADRID 2003

TESIS DOCTORAL

Números primos especiales
y sus aplicaciones criptográficas

TRIBUNAL CALIFICADOR

Presidente D. _____

Vocal D. _____

Vocal D. _____

Vocal D. _____

Secretario D. _____

Realizado el acto de defensa y lectura de la tesis en Madrid, el día de

CALIFICACIÓN: _____

EL PRESIDENTE

LOS VOCALES

EL SECRETARIO

A mis padres y a mis hermanos

Agradecimientos

Mentiría si dijera que estoy escribiendo con soltura estas líneas de agradecimiento: atribuir el justo reconocimiento a cada uno de los actores de un proceso tan largo como es el escribir una tesis doctoral es una tarea imposible. Vaya por delante este descargo de conciencia.

Es también un momento de emoción y de recuerdos. Muchas veces he imaginado cómo sería el momento —parecía tan lejano— en que pudiera sentarme tranquilamente y dar rienda suelta a la expresión de esos sentimientos, totalmente ausentes en la sobriedad científica de los capítulos que siguen, indicadores de que el trabajo veía por fin su remate. Por lo demás, nada extraordinario ocurre: simplemente se comprueba una vez más la eficacia del trabajo constante, los resultados que pueden llegar a obtenerse por el sencillo procedimiento de dar un paso detrás de otro, aunque cada uno sea pequeño.

Todas estas lecciones, y otras muchas, me han venido como regaladas de lo alto por tener la oportunidad de trabajar junto a Jaime del que puede decirse con toda verdad que es un buen maestro y un maestro bueno: en él, cabeza y corazón van unidos. Y para mí ha sido más que maestro: también amigo a quien se puede confiar el alma. Es fácil llegar muy alto cuando uno se apoya en los hombros de un gigante. Y, para colmo, no hay manera de conseguir invitarlo a café.

A mi querido y viejo amigo Pedro Martínez debo el haber encontrado esta joya: Pedro, muchísimas gracias. No sé cómo convenciste al pobre Jaime de que aceptara un alumno más, estando como estaba —y sigue estando— sobrecargado de trabajo. Si estas líneas se escriben hoy, te deben a ti al menos ser el punto de inicio. Y, como dicen los clásicos, comenzar es tener la mitad hecho.

Densas horas de labor han conocido los muros del entrañable edificio en que se alberga el Instituto de Física Aplicada del Consejo Superior de Investigaciones Científicas. Al Director del Departamento de Tratamiento de la Información y Codificación, mi querido Fausto Montoya, debo una cálida acogida y una solicitud sin reservas ante todo lo que me ha sido menester, incluyendo también las necesidades materiales que acompañan siempre los trabajos de los hombres. Quisiera que los años me dieran a mí también su misma disposición de ánimo, capaz de convertir en fácil lo que se antojaba imposible a miradas —como la mía— mucho menos perspicaces.

Es una suerte haber podido disfrutar también la amistad de muchos otros que me han precedido o me han acompañado en este camino: Luis Hernández con quien he compartido provechosas horas de café; Lola, Amparo, Slobodan y Gonzalo; Miguel Romera y Gerardo Pastor, fuentes inagotables de sabiduría en los más diversos campos; Alfonso Blanco, al que más de una vez hice sufrir con

mi torpeza para el mundo hardware; y, cómo no, Alberto Peinado, pues una tarde cualquiera en tu despacho, vimos por fin *la idea*. No puedo dejar pasar a mis queridos Ignacio Álvarez y Juan José Marina, eternamente bien dispuestos a echar una mano “en lo que sea”. Y, es de justicia, también tiene su aportación Jesús Negrillo, a cuyos sabios conocimientos de los “misterios informáticos” debo la gracia de poder teclear hoy —ya no se garabatea, como antes— estas líneas.

Mis amigos han sido siempre fuentes de aliento durante la noche oscura del alma que todo tesinando necesariamente sufre. Para esto viene muy bien la ayuda de los que han pasado por ella, como mi querido amigo Marco, manantial perpetuo de consejos animosos y prudentes a quien debo en lo académico y en lo personal, mucho más de lo que imagino. Espero que podamos reanudar ahora nuestras fantásticas excursiones por la sierra. A él debo la amistad con algunos de los miembros del Departamento de Física Aplicada de la Universidad Politécnica, que han tenido la amabilidad de recibirmee como candidato: José de Frutos, mi tutor, en primer lugar; pero también me han honrado con sus amistad y sus consejos Amador González, Pedro Sánchez, Ángel Sanz y Pedro Salas, con los que todavía me río recordando nuestras aventuras en el congreso de “Quantum Information” de Almagro.

No puedo pasar por alto a cuantos han soportado sin quejas mis estridencias, sin quizá entender mucho ni poco si yo estudiaba primos especiales, o por el contrario el “primo especial” era yo mismo. Luis, Ana, Elena, PacoCa y PacoRo, Puri, M. Ángeles, Jesús y Antonio; Vicente y Emilio y, a su manera, también Alfonso y Manuel. Recuerdo con mucho cariño las charlas en que yo me bebía las palabras de José María hablando de Platón (“no digas, por favor, que Hegel es idealista”, me commina): no tardará mucho en ser doctor. Quizá lo gane mi querido José Ángel, aunque su vocación es, claramente, la diplomacia; o bien José María, que comienza flamante su actividad profesional. Edu: tú también tienes tu sitio, pero que no se te suba a la cabeza.

Mención aparte hago de mis queridísimos Elena y Manuel José, y de Ángel. Siempre me habéis mirado con cariño por haber “atentado tesis civil” —que diría Ángel— y habéis estimulado en mí, con vuestro ejemplo y vuestra amistad, el deseo de ir *citius, altius, fortius* a conseguir unos objetivos en que creíais —estoy seguro— con mucha más intensidad que yo mismo.

Quisiera dedicar un recuerdo a todos mis profesores, con una entrañable mención a la persona que de alguna manera sembró en mí y en otros muchos una semilla de largo alcance: se trata de la maestra de mi pueblo, a quien respetuosamente llamábamos doña Isabel, que desveló para mí la magia escondida en el inagotable mundo de los libros por el sencillo expediente de enseñarme a leer.

A mi familia, puerto seguro y refugio final. Resplandecen deslumbradoras las palabras del libro sagrado: ¡Ay del que está solo! Gracias a Dios, no es mi caso.

A cuantos involuntariamente han quedado omitidos, a quienes consciente o inconscientemente, por su parte o por la mía, han contribuido a que este proyecto se realice.

A todos, sinceramente, gracias.

Madrid, 8 de junio de 2003

*En teoría, entre la teoría y la práctica no hay
ninguna diferencia, pero en la práctica sí la hay*

*En el principio era el Verbo
(S. Juan, 1, 1)*

Índice general

Introducción	1
Resumen del trabajo	1
Clasificación	2
Evolución histórica	4
Desarrollos recientes	7
Resumen de contenidos	8
Capítulo 1: Preliminares	8
Capítulo 2: Criptosistemas de clave pública y primos especiales	9
Capítulo 3: Tests de primalidad y otros algoritmos empleados	12
Capítulo 4: Primos seguros	14
Capítulo 5: Primos robustos	15
Capítulo 6: Aplicaciones criptográficas de los primos especiales	17
Capítulo 7 y Apéndice I	18
1 Preliminares	19
1.1 Herramientas matemáticas	19
1.1.1 Notación	19
1.1.2 Teoría de la divisibilidad	20
1.2 Nociones de complejidad computacional	34
1.2.1 Algoritmos	34
1.2.2 Tamaño de un entero	34
1.2.3 Tiempo de ejecución de la suma y el producto	35
1.2.4 Tiempo de ejecución de un algoritmo	36
1.2.5 Clasificación de los algoritmos	37
1.3 Herramientas de computación	38
1.3.1 Bibliotecas de programación con multiprecisión	39
1.3.2 Aplicaciones de computación simbólica	42
2 Criptosistemas de clave pública y primos especiales	45
2.1 Orígenes de los criptosistemas de clave pública	45
2.1.1 Introducción	45
2.1.2 El cambio de clave de Diffie-Hellman	47
2.1.3 Criptoanálisis del cambio de clave de Diffie-Hellman	48
2.1.4 Funciones unidireccionales	49
2.1.5 Función unidireccional <i>exponenciación discreta</i>	50
2.2 Criptosistemas de clave pública	54

2.2.1	Formalismo de clave pública	54
2.2.2	Firma digital	55
2.3	Criptoanálisis para los sistemas de clave pública	56
2.3.1	Ataques pasivos	56
2.3.2	Ataques activos	57
2.4	Criptosistema RSA	57
2.4.1	Elección de claves	58
2.4.2	Envío de mensajes	58
2.4.3	Descifrado del mensaje	58
2.4.4	Condiciones de p, q y criptoanálisis	59
2.4.5	Firma digital en el RSA	62
2.4.6	Firma digital en una red	62
2.5	Otros criptosistemas de tipo RSA	64
2.5.1	Criptosistema de Rabin	64
2.5.2	Criptosistemas de Williams, Kurosawa <i>et al.</i> y Loxton <i>et al.</i>	65
2.5.3	Los criptosistemas de Takagi	66
2.6	Criptosistema de ElGamal	67
2.6.1	Descripción del criptosistema	67
2.6.2	Firma digital en ElGamal	68
2.6.3	Criptoanálisis	69
2.6.4	Ataques de Pohlig-Hellman al logaritmo discreto	69
2.6.5	Firma digital estándar del NIST	70
2.7	Criptosistemas probabilísticos	73
2.7.1	Definición de los criptosistemas probabilísticos	73
2.7.2	Criptosistema de Goldwasser-Micali	75
2.7.3	Criptosistema de Blum-Goldwasser	76
2.7.4	Criptosistema de Blum-Goldwasser mejorado	83
2.8	Clave secreta versus clave pública	87
2.8.1	Ventajas de la clave secreta	87
2.8.2	Desventajas de la clave secreta	87
2.8.3	Ventajas de la clave pública	88
2.8.4	Desventajas de la clave pública	88
3	Tests de primalidad y otros algoritmos empleados	91
3.1	Introducción explicativa	91
3.2	Tests de primalidad	92
3.2.1	La noción de test de primalidad	92
3.2.2	El algoritmo de Miller-Rabin	92
3.2.3	Algoritmo de Miller-Rabin modificado	96
3.2.4	El test de Pocklington-Lehmer	97
3.3	Algoritmos de factorización	98
3.3.1	Algoritmo $p - 1$ de Pollard	98
3.3.2	Algoritmo de Pollard, fase 2	100
3.3.3	Algoritmo $p + 1$ de Williams	101
3.4	Generación de números pseudoaleatorios	103
3.4.1	Algoritmo BBS	103
3.4.2	Algoritmo de Lehmer	104

3.4.3	Algoritmo de Tausworthe	105
3.5	Generación de primos aleatorios	108
4	Primos seguros	111
4.1	Primos seguros	111
4.1.1	Definición y propiedades elementales	111
4.1.2	Primos seguros y primos de Sophie Germain	113
4.1.3	Signaturas alternadas	113
4.1.4	Cadenas de primos seguros	116
4.2	Densidad de primos 1-seguros	119
4.2.1	Función recuento de primos	119
4.2.2	Funciones recuento para primos seguros	121
4.2.3	Fórmula heurística asintótica para π_1^+	122
4.2.4	Fórmula heurística asintótica para π_1^-	126
4.2.5	El trabajo de Y. Cai	127
4.3	Densidad de primos 2-seguros	129
4.3.1	Fórmula heurística asintótica para π_2^+	129
4.3.2	Fórmula heurística asintótica para π_2^-	136
4.4	Densidad de primos k -seguros	137
4.4.1	Fórmula asintótica generalizada para π_k^\pm	137
4.5	Generación de primos seguros	140
4.5.1	Generación de primos 1-seguros	140
4.5.2	Generación de primos 2-seguros	141
5	Primos robustos	143
5.1	Diversas definiciones de primo robusto	143
5.1.1	La definición estándar	143
5.1.2	La definición de Ogiwara	144
5.1.3	El análisis de Rivest	145
5.2	Primos robustos óptimos	146
5.2.1	La noción de primo robusto óptimo	146
5.2.2	Caracterización de los primos robustos óptimos	149
5.2.3	Función recuento	154
5.3	Algoritmos para la generación de primos robustos	157
5.3.1	Algoritmo de J. Gordon	158
5.3.2	Algoritmo de M. Ogiwara	164
5.3.3	Algoritmo para primos robustos óptimos	169
6	Aplicaciones criptográficas de los primos especiales	173
6.1	Introducción	173
6.2	Aplicaciones al criptosistema RSA	173
6.2.1	Uso de primos 1-seguros	173
6.2.2	Uso de primos robustos	174
6.3	Aplicaciones al generador BBS	175
6.3.1	Uso de primos 1-seguros	175
6.3.2	Uso de primos 2-seguros	175
6.4	El problema del cambio de clave	176

6.5	Tiempo de ejecución para un primo 1-seguro	177
6.5.1	Estimación del número de tiradas	177
6.5.2	Tiempo de ejecución teórico	179
6.5.3	Datos experimentales	180
6.5.4	Algoritmo mejorado para primos 1-seguros	184
6.6	Tiempo ejecución para un primo 2-seguro	185
6.6.1	Estimación del número de tiradas	186
6.6.2	Tiempo de ejecución teórico	186
6.6.3	Datos experimentales	188
6.6.4	Algoritmo mejorado para primos 2-seguros	192
6.7	Tiempo de ejecución para un primo robusto óptimo	193
6.7.1	Estimación del número de tiradas	193
6.7.2	Tiempo de ejecución teórico	194
6.7.3	Datos experimentales	194
6.7.4	Conjetura sobre la función recuento π_σ^*	200
7	Conclusiones, aportaciones y desarrollos futuros	205
7.1	Conclusiones	205
7.2	Aportaciones	207
7.3	Desarrollos futuros	208
Apéndice I		211
I.1	Propósito	211
I.2	Algoritmo 1.9 (Euclides)	211
I.3	Algoritmo 1.14 (EuclidesExt)	212
I.4	Algoritmo 1.44 (Jacobi)	213
I.5	Algoritmo 3.5 (TestMillerRabin)	215
I.6	Algoritmo 3.8 (MillerRabin)	216
I.7	Algoritmo 3.18 (NumeroAleatorioBBS)	218
I.8	Algoritmo 3.21 (NumeroAleatorioLehmer)	219
I.9	Algoritmo 3.23 (NumeroAleatorioTausworthe)	221
I.10	Algoritmo 3.27 (GeneraPrimoAleatorio)	223
I.11	Algoritmo 4.41 (Primo1Seguro)	223
I.12	Algoritmo 4.42 (Primo2Seguro)	224
I.13	Algoritmo 5.24 (GordonStrong)	225
I.14	Algoritmo 5.26 (GordonStrong2)	226
I.15	Algoritmo 5.30 (OgiwaraStrong)	227
I.16	Algoritmo 5.33 (StrongOptimo)	231
I.17	Cálculo de la constante C_2 (véase 4.3.1)	232
I.18	Cálculo de la constante C_σ (véase 5.2.3)	233
Referencias		235

Índice de figuras

3.1	Tiempo de ejecución para n primo	95
3.2	Tiempo de ejecución para n compuesto	95
4.1	Valor de la constante B_1	125
4.2	Valor de la constante B_2	136
5.1	Gráfica de la función $n \mapsto \sigma(n)$	152
5.2	Gráfica de la función $n \mapsto \sigma(n)$	153
5.3	Gráfica de la función $p \mapsto \sigma(p)$	153
5.4	Función recuento π_σ	154
5.5	Valor de la constante B_σ	157
6.1	Tiradas T_1 frente a número de bits	181
6.2	Valores teórico y experimental de la razón α	182
6.3	Tiempo de ejecución en segundos frente a número de bits	184
6.4	Tiradas T_2 frente a número de bits	189
6.5	Valores teórico y experimental de la razón β	189
6.6	Tiempo de ejecución en segundos frente a número de bits	191
6.7	Tiradas T_s frente a tamaño en bits	196
6.8	Valores teórico y experimental de la razón γ	196
6.9	Tiempo de ejecución en segundos frente a número de bits	200

Introducción

Resumen del trabajo

El objeto de esta memoria es el estudio de ciertas clases de primos que, por estar dotados de propiedades especiales, resultan de interés para su uso en los criptosistemas de clave pública.

Las clases de primos consideradas han sido las siguientes:

1. Los primos 1-seguros, determinados por la siguiente propiedad: un primo p se denomina 1-seguro si y sólo si $p = 2q + 1$, donde q es otro primo.
2. Los primos 2-seguros, determinados por la siguiente propiedad: un primo p se dice 2-seguro si $p = 2q + 1$ y además q es 1-seguro.
3. Los primos robustos¹. Adelantándonos a las definiciones más rigurosas, podemos decir que esta clase de primos presenta varias variantes, que comparten entre sí la propiedad de que si p es un primo robusto entonces $p + 1$ y $p - 1$ contienen factores primos “grandes”; y además algunos de estos factores presentan a su vez esta misma propiedad. Dejamos para su lugar el precisar más esta noción y las diversas variantes.

En la Definición 4.1 de este trabajo hemos generalizado las definiciones de los puntos 1 y 2 introduciendo la noción de primo k -seguro de signatura arbitraria. Por ejemplo, de acuerdo con tal definición existen dos clases de primos 1-seguros: los de signatura $+1$, que coinciden con los definidos en el punto 1 anterior; y los de signatura -1 , que se escriben como $p = 2q - 1$, donde q es otro primo. Obsérvese que la condición “ $p + 1$ contiene un factor primo grande” se verifica de modo óptimo cuando p es un primo 1-seguro de signatura -1 . Análogamente, la condición “ $p - 1$ contiene un factor primo grande” se verifica de modo óptimo cuando p es un primo 1-seguro de signatura $+1$. Por consiguiente, los primos seguros así generalizados se convierten en los “ladrillos” que permiten construir los primos robustos y otras clases de primos especiales.

Por este motivo esta memoria dedica una especial atención a la noción general de primo seguro, haciendo hincapié en las clases de primos 1- y 2-seguros, que son los que más se usan en las aplicaciones.

¹Queremos proponer este término como equivalente para el mundo hispanohablante de lo que en la literatura anglosajona se conoce con el nombre de “strong primes”. Algunos autores de habla castellana utilizan también el término *primos fuertes*.

También se ha introducido una clase novedosa de primos robustos que designamos como “primos robustos óptimos”. La novedad consiste en definir una cierta función σ de variable discreta que permite caracterizar el grado de “robustez” de un primo robusto. Concretamente, los primos robustos óptimos son los mínimos de la función σ en el conjunto de los primos mayores o iguales que 23.

Para cada clase de primos propuesta se ha estudiado:

1. su distribución;
2. su función recuento;
3. la probabilidad de seleccionar uno de ellos aleatoriamente dentro del conjunto de los enteros positivos;
4. el tiempo de computación asociado a la extracción aleatoria de uno de ellos.

Con estos datos, es sencillo predecir un parámetro de importancia vital para los criptosistemas de clave pública; a saber, el tiempo necesario para el cambio de las claves, estimado con suficiente precisión: un buen sistema criptográfico para el que fuera muy costosa la modificación de claves resultaría inútil en la práctica.

Muchos de los resultados obtenidos no están demostrados rigurosamente, si bien todos ellos se apoyan en conjeturas que, establecidas por autores clásicos (por ejemplo, [7, 8, 9, 21, 29, 50, 68, 102]) están confirmadas por múltiples experimentos numéricos dentro de los rangos que se utilizan en las aplicaciones actuales. Conviene no perder de vista que las demostraciones de las conjeturas clásicas en teoría de números avanzan muy lentamente: sin ir más lejos, ni siquiera está demostrado rigurosamente que existan infinitos primos 1-seguros (véase [94]). Sin embargo, los desarrollos prácticos exigen conocer con la máxima precisión posible los tiempos de computación necesarios para obtener primos de las diversas clases utilizadas para las claves de algunos criptosistemas de clave pública. Aquí radica el interés de esta memoria, que proporciona estimaciones heurísticas fiables acerca de los tiempos de computación antes referidos.

Dedicamos la última parte de la memoria a presentar las aplicaciones prácticas de aquello que valoramos como de más interés en este trabajo. En concreto, a lo largo del último capítulo se proporcionan los datos de esfuerzo computacional necesario para obtener primos de las distintas clases estudiadas. Estos experimentos numéricos vienen a confirmar en la práctica la exactitud de las predicciones teóricas.

Queda aclarada así la motivación que ha dado origen a esta memoria: el estudio de un subconjunto de primos dotados de propiedades especiales que los hacen de interés para su uso en los sistemas criptográficos actuales de clave pública.

Los resultados de este trabajo son deudores en gran medida de las herramientas computacionales que existen actualmente. Hemos recurrido a dos grupos de herramientas: las aplicaciones informáticas de álgebra computacional, como MAPLE o MATHEMATICA; y las bibliotecas de programación que permiten desarrollar programas más eficientes para realizar computaciones concretas.

Clasificación

Presentamos a continuación la clasificación del presente trabajo de acuerdo a la *American Mathematical Society Classification* correspondiente al año 2000, actualmente en vigor. Este sistema de clasificación es comúnmente aceptado y de amplio uso.

Según ello, hemos asignado a esta memoria una clave primaria y un conjunto de claves secundarias. Para facilitar la lectura, ofrecemos también el significado de cada clave.

Primaria

94A60 Cryptography.

Secundarias

11A05	Multiplicative structure; Euclidean algorithm; greatest common divisors.
11A07	Congruences; primitive roots; residue systems.
11A15	Power residues, reciprocity.
11A25	Arithmetic functions; related numbers; inversion formulas.
11A41	Primes
11A51	Factorization; primality.
11A63	Radix representation; digital problems.
11D85	Representation problems.
11K45	Pseudo-random numbers; Monte Carlo methods.
11M06	$\zeta(s)$ and $L(s, \chi)$; results on $L(1, \chi)$.
11N05	Distribution of primes.
11N13	Primes in progressions.
11N32	Primes represented by polynomials; other multiplicative structure of polynomial values.
11N69	Distribution of integers in special residue classes.
11P55	Applications of Hardy-Littlewood method.
11T71	Algebraic coding theory; cryptography.
11Y05	Factorization.
11Y11	Primality.
11Y16	Algorithms; complexity.
11Y35	Analytic computations.
11Y60	Evaluations of constants.
62B10	Information-theoretic topics.
65C10	Random number generation.
68P25	Data encryption.
68Q25	Analysis of algorithms and problem complexity.
68W20	Randomized algorithms.
90C60	Abstract computational complexity for mathematical programming problems.
94A05	Communication theory.
94A15	Information theory, general.

Secundarias

94A62 Authentication and secret sharing.

Evolución histórica

Sin mucha discusión se suele admitir el destacado papel que desempeña la comunicación en la vida humana como causa y efecto a la vez de la misma sociedad. Como contraejemplo se puede aducir el atraso cultural que experimentan aquellos pueblos que han permanecido incomunicados por razones geográficas, históricas, lingüísticas o de cualquier otro tipo.

El siglo XX parece haber conocido la realización más acabada de la profecía ilustrada acerca del papel preponderante de la ciencia y la tecnología en el progreso de los pueblos. Y, aunque es evidente que no todas las aplicaciones tecnológicas han sido positivas para la humanidad, sí se puede afirmar que un ámbito como las telecomunicaciones ha sido beneficiario privilegiado de los avances científicos, especialmente de los de la física del estado sólido.

El desarrollo y espectacular miniaturización de los circuitos integrados, particularmente los microprocesadores, singulares dispositivos capaces de ser programados para realizar virtualmente cualquier tipo de operación con velocidades superiores actualmente a los miles de millones por segundo, ha permitido la realización del viejo sueño de grandes visionarios: conseguir una red que interconectara sistemas de computadores situados en cualquier parte del mundo y, al tiempo, disponer de esos sistemas de forma sencilla y económica. Todo ello ha devenido en lo que el mundo anglosajón, con su característica expresividad, ha denominado *worldwide web*, es decir, la telaraña de ámbito mundial, la red Internet. Y esto es sólo el principio.

La fulgurante extensión de los sistemas de comunicación ha propiciado simultáneamente el resurgir de las ciencias y las técnicas que desde tiempos inmemoriales se habían ocupado de la transmisión *segura* de los datos. Con este adjetivo quiero referirme a unas propiedades que una comunicación *segura* garantiza con respecto al mensaje transmitido; por ejemplo y sin ánimo de ser exhaustivo:

- la *integridad*, es decir, el mensaje ha llegado íntegro a su destino;
- la *autenticidad*, es decir, el mensaje no ha sido falsificado por un tercero durante la transmisión;
- la *firma*, es decir, el mensaje ha sido realmente emitido por quien dice ser su autor;
- la *confidencialidad*, es decir, el contenido del mensaje ha permanecido en secreto durante el curso de la transmisión.

De estos y parecidos temas se ha ocupado tradicionalmente la Criptología², en sus dos vertientes: la Criptografía, que estudia las técnicas de cifrado de la información; y el Criptoanálisis, cuyo objeto es encontrar puntos débiles y establecer

² Adjetivo griego *κρυπτός*: oculto, secreto.

métodos para atacar los sistemas criptográficos y lograr así acceso a la información transmitida mediante ellos aun sin estar en posesión de la debida autorización.

Históricamente, el interés de estas ciencias y técnicas estaba restringido fundamentalmente a los ámbitos militares o diplomáticos y se consideraban de “interés nacional”, por lo que en no pocos casos, sus logros no eran conocidos públicamente. Ha sido justamente la *revolución informática* a que me refería antes la que trajo consigo la demanda de servicios de seguridad y de protección de la información, almacenada ya casi siempre en formatos digitales.

Los sistemas más antiguos —y todavía vigentes— se denominan *criptosistemas de clave secreta* porque se basan en el conocimiento exclusivo y simultáneo por las partes usuarias de cierta información, la clave, cuya posesión da acceso a la información trasmitida. El algoritmo usado puede ser públicamente conocido o no, pero las claves han de estar reservadas a los usuarios. Como ejemplos muy antiguos están los algoritmos de *trasposición* y de *sustitución*. El primero consiste simplemente en barajar los símbolos del mensaje original y producir una cadena desordenada que consta de esos mismos símbolos: la clave secreta es conocer cómo se ha realizado esa “desordenación”; el segundo, un poco más elaborado, se basa en establecer una correspondencia entre los símbolos con que está escrito el mensaje (típicamente, las letras del alfabeto) y otro conjunto de símbolos (que podría ser el mismo alfabeto). La clave secreta, en este caso, es conocer cuál es esa correspondencia para poder descifrar así el mensaje original. Estos sistemas son muy primitivos y no resisten los ataques de tipo estadístico, que se basan en la frecuencia de aparición de cada símbolo según el idioma en que esté escrito el mensaje. Hoy en día los algoritmos utilizados son más complicados y difíciles de atacar, pero la idea del criptosistema de clave secreta permanece sustancialmente idéntica.

Sin embargo, estos sistemas presentan inconvenientes fácilmente comprensibles: hay que mantener una clave distinta para cada par de transmisores; cada parte ha de conocerla con antelación al establecimiento de la sesión; no es posible *firmar* el mensaje, etc. Además, cuando la red es grande, los problemas se multiplican cuadráticamente. Así las cosas, en la década de los años 70 apareció el novedoso concepto de la *criptografía de clave pública*. Como ya se explicará con más detalle, los autores W. Diffie y M. Hellman, en su archicitado artículo *New Directions in Cryptography* (la referencia exacta es [30]) desarrollaron un método por el cual dos partes podían intercambiar una información a través de un canal público sin que el conocimiento aportado por la escucha del canal pueda conducir a averiguar nada respecto a la información intercambiada. Por ello, este protocolo recibe el nombre de *intercambio de clave de Diffie-Hellman*.

Este ingenioso protocolo despertó el interés investigador en la comunidad de criptógrafos y pronto, en 1978, los investigadores R. Rivest, A. Shamir y L. Adleman inventaron el primer criptosistema de clave pública seguro y de uso práctico, que recibió, a partir de sus autores, el nombre de RSA y está hoy en día ampliamente difundido. La referencia exacta es [97].

La idea de este nuevo sistema consiste en que cada usuario maneja en realidad dos claves: una es la clave pública, que el usuario da a conocer al resto de los usuarios del sistema; y una clave privada, que el usuario se reserva exclusivamente para sí. Si el usuario *A* desea enviar a *B* un mensaje, basta que obtenga la clave

pública de B , cifre el mensaje con ella y lo envíe por un canal público. El punto básico es que sólo quien tenga el conocimiento de la clave privada es capaz de descifrar ese mensaje que circula por el canal: en el ejemplo, el usuario B .

Otro punto básico es la posibilidad de la firma digital: garantizar que, efectivamente, un determinado mensaje ha sido emitido, sin ningún género de duda, por un determinado usuario. Utilizando la criptografía de clave pública, es posible diseñar un protocolo que permita asegurar la identidad de la parte emisora del mensaje: es el proceso conocido genéricamente con el nombre de *autenticación*.

La seguridad de los algoritmos que se emplean en los criptosistemas de clave pública descansa en la dificultad de resolver ciertos problemas matemáticos, conocidos como funciones unidireccionales. Una función unidireccional $f : \mathcal{M} \mapsto \mathcal{C}$ es aquella tal que $f(m)$ es “fácil” de calcular para todo $m \in \mathcal{M}$ pero es computacionalmente inabordable calcular $m = f^{-1}(c)$ para “casi” todo $c \in \mathcal{C}$. De entre estas funciones, es particularmente interesante el caso de las *funciones unidireccionales con trampilla*. Éstas se caracterizan porque existe una información peculiar (llamada “trampilla”) cuyo conocimiento hace posible el cálculo de la función inversa que, sin él, se convierte en computacionalmente inabordable.

Existen diversos tipos de funciones unidireccionales con trampilla, que dan lugar a posibles criptosistemas de clave pública. Aunque de todo ello se dará cumplida cuenta más tarde, es interesante introducir ahora con algún detalle el RSA por su relación con el objeto de la presente memoria. El problema matemático ligado con el RSA es la factorización de números enteros. De la aritmética elemental es conocido que todo entero n puede ser descompuesto en un producto de factores primos

$$n = p_1^{e_1} p_2^{e_2} \cdots p_k^{e_k},$$

donde p_i , $1 \leq i \leq k$ son números primos. Así como resulta elemental conocer n dada su factorización (basta hacer la cuenta de multiplicar) es computacionalmente inabordable el problema general de descomponer cualquier número n en sus factores primos. Más adelante describiremos con detalle el protocolo exacto de este sistema: bástenos por ahora decir que emplea típicamente un número n , producto de dos números primos p y q , grandes y próximos; y que su seguridad descansa en la dificultad de factorizar ese número. La información trampilla para este caso es precisamente el conocimiento de los factores de n , es decir, la función inversa para este sistema se hace “fácil” de calcular con esa información adicional. Así pues, los esfuerzos por atacar RSA se centran en la factorización del número n , conocido como el “módulo” del criptosistema.

La aparición del RSA reavivó la llama del estudio de la factorización, movido por este nuevo objetivo. En efecto, con los años se habían desarrollado métodos y técnicas que permiten factorizar en ciertos supuestos, especialmente cuando los factores primos del módulo n cumplen ciertas propiedades. Recíprocamente, si los factores primos pertenecen a ciertas clases especiales, los algoritmos conocidos no alcanzan su objetivo de factorizar. Como ejemplo, Pollard presentó en 1974 un algoritmo (la referencia exacta es [89]) que permite factorizar eficientemente un número n si uno de sus factores primos p satisface la propiedad de que todos los factores primos de $p - 1$ son más pequeños que una cota B suficientemente “baja” (a efectos prácticos, se entiende por “baja” un valor del orden de 10^6 ó 10^7). Pues

bien, una forma de frustrar este ataque es elegir p de tal modo que $p - 1$ sólo contenga factores primos “grandes”. Como hemos dicho más arriba, este tipo de primos recibe el nombre de robustos.

Otro ejemplo muy actual es el desarrollo del generador pseudo-aleatorio de números BBS, debido a L. Blum, M. Blum y M. Shub, quienes lo describen por primera vez en [13], y se basa en una iteración cuadrática módulo un entero. Se trata de un generador criptográficamente seguro, es decir, dada una cantidad de bits generada, menor de cierto límite, es computacionalmente inabordable el problema de averiguar cuál será el bit siguiente con probabilidad mayor de $1/2$. Por lo tanto, este generador como veremos en su momento, puede ser utilizado para construir un criptosistema. Ahora bien, un problema fundamental de este tipo de generadores es que trabaja cíclicamente, es decir, sólo produce secuencias de un número finito de elementos. Así pues resulta de capital importancia poder predecir la longitud mínima del ciclo en que el generador se encuentre en un momento dado. Como veremos, en el caso del BBS el punto clave es elegir el módulo de tal modo que sus factores primos cumplan ciertas propiedades, entre las que se encuentra ser 1-seguro o 2-seguro.

Un ejemplo de parecida importancia es el criptosistema de ElGamal que se basa en la dificultad de invertir la función exponencial de variable discreta, llamada, por analogía, logaritmo discreto (véase [39]). El único ataque conocido al logaritmo discreto es el algoritmo desarrollado por Pohlig y Hellman para el grupo multiplicativo \mathbb{Z}_p^* (véase [88]). Si todos los factores primos de $p - 1$ son pequeños, el tiempo de ejecución es *polinómico*, lo cual se traduce en la práctica, como ya veremos, en que es eficiente. Por tanto, también en este caso, usar primos 1-seguros resulta útil, porque permite evitar este ataque contra el sistema.

Desarrollos recientes

Comparada con otras áreas de la Matemática, la teoría de números ha conocido históricamente un avance más bien lento. Ocurre así que persisten sin demostrarse añejas conjeturas como la de Goldbach, que éste planteó en 1742 en dos cartas dirigidas a Euler y afirma que todo número par mayor que 2 es la suma de dos primos; otra conjetura famosísima es la de Riemann: afirma que los ceros no triviales de la función $\zeta : \mathbb{C} \rightarrow \mathbb{C}$ (que lleva su nombre) tienen todos como parte real $1/2$; y así existen muchas otras no tan conocidas. La mayor parte de los avances —lentos y difíciles— se hacen dando por buenas todas esas conjeturas.

Un primer campo concierne a la generación de números primos, particularmente la de los que presenten propiedades especiales. En este sentido destacan los trabajos de Maurer y Mihailescu (véanse [70, 75]) que tienen por objeto generar rápidamente primos (casi)-probados usando progresiones aritméticas y que satisfacen ciertos requisitos para ser utilizados en el criptosistema RSA, en concreto resisten al ataque del cifrado reiterado.

El criptoanálisis, especialmente el dirigido al RSA, también presenta resultados como los de Durfee usando métodos de retículos (véanse [37] o Gysin *et al.* [49]) que investiga los ataques cíclicos. Boneh en [16] trabaja junto con Durfee los ataques cuando el exponente privado es pequeño.

Se han mejorado los algoritmos de factorización. A este respecto están los trabajos de Pollard y Williams, con sus algoritmos del tipo $p \pm 1$ (véanse [89, 122]) que venían a complementar el más clásico de Brillhart *et al.* (véase [18]). Otro método que ha recibido un considerable impulso es la llamada criba del cuerpo de números (véase el resumen histórico [11]). Por otro lado se ha desarrollado el método de las curvas elípticas, para el que son de interés los trabajos de Atkin y Morain (véanse [4, 78]). Con estos resultados, se ha llegado a factorizar números de 512 bits, como el presentado en el EUROCRYPT 2000 ([22]). Véase [32] para un estudio detallado sobre los módulos de RSA factorizados recientemente.

Aunque existen muchos estudios modernos referentes a las distribución de primos especiales —esto es precisamente uno de los motivos de nuestro trabajo— es interesante el debido a Y. Cai (véase [21]) que estudia la densidad de los primos de Sophie Germain (el autor los denomina “seguros”: más adelante matizaremos su distinción), usando para ello resultados del profesor C. Pan sobre la conjectura de Goldbach (véase [84]).

Existen también trabajos en la línea de la generación de primos de la clase robustos. La más tradicional fue inaugurada por Gordon en su clásico artículo [46], al que otros autores, como Shawe-Taylor, añadieron mejoras (véanse [105, 106]). Ogiwara presentó en [82] un método para generar primos robustos de “6 vías” (los anteriores eran de “3 vías”) junto con un algoritmo de comprobación de primalidad. Por su interés, en este trabajo presentamos una implementación de Gordon y Ogiwara.

Por último, destacamos también la aparición de nuevos criptosistemas de clave pública como el propuesto en [54] y basado en el generador de números aleatorios BBS; o variantes del RSA como los criptosistemas propuestos por Kurosawa *et al.* en [57], Loxton *et al.* en [64], o Takagi en [112, 113].

Resumen de contenidos

A continuación presentamos un resumen pormenorizado de los contenidos de cada uno de los capítulos.

Capítulo 1: Preliminares

Comenzamos la memoria con este capítulo dedicado a nociones preliminares, en el que introducimos notación y herramientas que serán necesarias para el desarrollo del trabajo.

Bajo el título “Herramientas matemáticas”, presentamos en la primera sección la parte más propiamente matemática. Ofrecemos en primer lugar la *notación* que se seguirá en lo sucesivo y, seguidamente, una breve pero necesaria introducción a la *teoría de la divisibilidad*. Con este nombre agrupamos los resultados clásicos que se refieren a la aritmética de números enteros, desde el teorema de la descomposición única en factores primos, al indicador de Euler y sus propiedades, pasando por el algoritmo de Euclides que permite el cálculo sencillo del máximo común divisor de dos números y su variante extendida.

Hacemos un especial énfasis en la estructura algebraica del conjunto de los enteros módulo n , representado como \mathbb{Z}_n , por ser muy utilizado en el ámbito de

la criptografía de clave pública. Repasamos que si n es primo, entonces \mathbb{Z}_n es un cuerpo finito de característica n y, en caso contrario, \mathbb{Z}_n es un anillo. Como veremos, el conjunto de los elementos invertibles para la operación de multiplicar forman el llamado conjunto de las unidades de \mathbb{Z}_n que presenta estructura de grupo multiplicativo y lo representamos como \mathbb{Z}_n^* . Explicamos el teorema chino del resto que permite resolver un sistema de congruencias simultáneas. Revisamos a continuación los teoremas de Fermat y de Euler referentes a la exponenciación modular e introducimos los símbolos de Legendre y Jacobi junto con los algoritmos que permiten su cómputo. Estos símbolos, como veremos en el siguiente capítulo, tienen aplicación en algunos criptosistemas de clave pública, como el de Goldwasser-Micali.

Muchos criptosistemas de clave pública se apoyan en la existencia de un “problema matemático” cuya solución se considera inabordable. Por eso, la siguiente sección está dedicada a la *teoría de complejidad computacional*, cuyo objetivo es proporcionar mecanismos que permitan clasificar los problemas computacionales de acuerdo con los recursos que se necesitan para resolvérlos, típicamente tiempo de proceso y espacio de almacenamiento en memoria. El elemento básico en este contexto es el *algoritmo*, que se define a continuación. Pasamos a explicar qué se entiende por *tiempo de ejecución de un algoritmo* e introducimos la *notación asintótica* que permite dar aproximaciones razonables para esos tiempos de computación. Termina la sección explicando cómo los algoritmos se clasifican de acuerdo con su tiempo de ejecución, apareciendo así, por ejemplo, *algoritmos de tiempo polinómico*, de *tiempo exponencial*, etc.

Cerramos el capítulo con la sección dedicada a describir las herramientas de computación de que hemos hecho uso a lo largo de la memoria. Resulta imprescindible disponer de bibliotecas de programación con multiprecisión, que permiten el uso programático de la aritmética con números enteros de tamaño arbitrario. Aunque existen muchas, hemos optado por usar GMP (véase [47]), una biblioteca de uso público y desarrollo abierto (es decir, cualquiera puede participar en él) que ofrece muchas utilidades y funciona con aceptable rendimiento: describimos someramente su interfaz programática, pensada para los lenguajes C y C++. Existen también otras herramientas computacionales muy útiles: los paquetes comerciales de álgebra simbólica MAPLE (véase [118]) y MATHEMATICA (véase [124]): remitimos al lector a los respectivos fabricantes para más detalles. Nosotros hemos utilizado el primero de ellos, pues las facilidades que ofrecen ambos son muy similares y disponíamos de licencia para él.

Capítulo 2: Criptosistemas de clave pública y primos especiales

Este capítulo trata de enmarcar el objeto principal que nos hemos planteado en la memoria: el uso de primos dotados de ciertas propiedades especiales que los hacen particularmente aptos para ser usados en los principales criptosistemas de clave pública.

Comenzamos la primera sección con un brevísimo resumen histórico de los criptosistemas de clave pública y las ventajas que ofrecen frente a sus homólogos de clave secreta, parte de cuyos problemas resuelve. Describimos el primer logro en el camino de la criptografía de clave pública, conocido como el *protocolo*

de cambio de clave de Diffie-Hellman, que permite a dos usuarios compartir una información secreta usando como vehículo un canal público. Explicamos a continuación qué se entiende por funciones unidireccionales y ponemos de manifiesto cómo el protocolo de Diffie-Hellman se apoya precisamente en una de ellas: la función *exponenciación discreta* y su inversa, el *logaritmo discreto*. Analizamos los tiempos de computación de cada una de ellas, evidenciando que el tiempo de ejecución de la exponenciación discreta es de tipo polinómico, mientras que su inversa, el logaritmo discreto, es de tiempo de ejecución subexponencial, con lo que se justifica el atributo de función unidireccional para aquella.

Con estos conceptos, pasamos en la siguiente sección a explicar formalmente el protocolo general de un sistema criptográfico de clave pública. Incluimos también el protocolo de la *firma digital* que permite a un destinatario asegurarse de que el mensaje que ha recibido procede realmente de quien dice ser su autor. Ésta es precisamente una de las utilidades de que carecen los criptosistemas de clave secreta.

Dedicamos la siguiente sección a explicar en qué consiste el *criptoanálisis de los criptosistemas de clave pública* y cuáles son sus principales técnicas, introduciendo además la nomenclatura que nos será de utilidad en lo sucesivo. El criptoanálisis engloba las técnicas que permiten abordar el ataque a un determinado criptosistema, estableciendo las probabilidades de éxito de los distintos tipos de ataque.

En las secciones sucesivas, pasamos a describir los diversos criptosistemas de clave pública existentes para los que es ventajoso utilizar primos dotados de propiedades especiales. Comenzamos con el más popular actualmente, el *criptosistema RSA*, basado en el problema de la factorización de números enteros. Uno de los parámetros más importantes del sistema es el *módulo*, un entero que típicamente es producto de dos números primos. Describimos el protocolo para la elección de claves, el cifrado de mensajes y su envío y el descifrado de los criptogramas. Abordamos entonces el criptoanálisis de este sistema para poner de manifiesto las condiciones que deben cumplir los factores primos del módulo para dotar al sistema de la máxima seguridad. Así aparece el primer conjunto de requisitos para esos factores primos, entre los que podemos destacar, por ejemplo, la conveniencia de que los factores primos sean del tipo robustos. Por último explicamos el protocolo de firma digital haciendo uso de RSA y cómo realizar esa firma en una red.

En la siguiente sección introducimos otros criptosistemas de clave pública similares a RSA porque basan su seguridad en la dificultad de factorizar un número entero, que también actúa de módulo. En primer lugar describimos el *criptosistema de Rabin*, basado en el problema de la raíz cuadrada, que consiste en, dado un entero a , encontrar otro entero b tal que $b^2 \equiv a \pmod{n}$. Se puede ver que este problema se resuelve si se puede factorizar n . Otros criptosistemas similares que describimos son los de *Williams* (véase [121]) y *Kurosawa et al.* (véase [57]), que presentan ciertas variantes respecto al de Rabin. Por su parte, *Loxton et al.* (véase [64]) proponen un criptosistema análogo a RSA pero trabajando en el anillo factorial de los enteros de Eisenstein $\mathbb{Z}[\omega]$, siendo $\omega = \exp(2\pi i/3)$. Por último, describimos las *propuestas de Takagi* consistentes en trabajar usando como módulo n^k (véase [112]), o bien $p^k q$ (véase [113]); describimos las ventajas, y también algún inconveniente, de cada uno. Lo importante de todo ello es que para todos los

criptosistemas descritos es de aplicación el mismo criptoanálisis que para el RSA; por ello, los factores primos del módulo han de satisfacer los mismos requisitos que para RSA y, por iguales razones, se hace interesante también aquí el uso de primos con propiedades especiales.

La siguiente sección está dedicada al criptosistema de ElGamal (véase [39]). Este criptosistema se basa en el problema del cálculo del logaritmo discreto a que hemos hecho referencia al describir el cambio de clave de Diffie-Hellman. Explicamos el protocolo de este sistema, que exige la elección de un grupo cíclico como parámetro global del sistema; ElGamal eligió \mathbb{Z}_p^* , donde p es un primo. Explicamos a continuación la implementación de la firma digital. Realizamos después el criptoanálisis de la firma digital y del criptosistema, para llegar a la conclusión de que el único ataque eficiente conocido es el diseñado por Pohlig y Hellman (véase [88]). Sin embargo, la discusión posterior pone de manifiesto que el ataque deja de ser de utilidad cuando el mayor factor primo de $p - 1$ es comparable en tamaño al propio p . Ahora bien, justamente ese es el caso si p es un primo 1-seguro, pues en tal caso el mayor factor primo de p es $(p - 1)/2$. Aparece de nuevo el interés del estudio de esta clase de primos especiales para poder construir un sistema más resistente a los ataques. Para concluir esta sección, dedicada al criptosistema de ElGamal, describimos con detalle la *firma digital estándar* propuesta por el NIST³, que recibe el nombre de DSS. Esta firma es sustancialmente una variante de la firma digital de ElGamal; presentamos también una lista de las principales ventajas e inconvenientes que esta propuesta comporta.

La siguiente sección aborda los *criptosistemas de clave pública de tipo probabilístico*. Este tipo de sistemas se caracteriza porque un determinado mensaje no produce el mismo criptograma si se cifra dos veces seguidas. Presentamos una motivación para su existencia y las ventajas e inconvenientes que presentan.

Introducimos en primer lugar el *criptosistema de Goldwasser-Micali* (véase [44]). Este criptosistema está basado en el problema de la *residualidad cuadrática*. Dado $a \in \mathbb{Z}_n$, tal que el símbolo de Jacobi $(\frac{a}{n}) = 1$, este problema consiste en decidir si a es resto cuadrático módulo n o no. El número n es, como en casos anteriores, el producto de dos primos. Sucede que el problema de la residualidad cuadrática planteado se vuelve trivial si se conoce la factorización de n , por lo que los requisitos que han de cumplir sus factores primos son análogos al caso del RSA: ello justifica de nuevo el estudio de los primos cuyas propiedades especiales dificultan la tarea de la factorización de n y, por tanto, robustecen el sistema.

Pasamos a continuación al *criptosistema de Blum-Goldwasser* (véase [14]), también de tipo probabilístico. Este criptosistema se apoya en el generador de números aleatorios BBS (véase [13]). Por ello, comenzamos la sección con una explicación detallada acerca de qué se entiende por sucesiones pseudo-aleatorias y cómo se reconocen; introducimos la importante noción de la *complejidad lineal* y definimos también qué se entiende por un *generador pseudo-aleatorio criptográficamente seguro*. Con todo ello, pasamos a describir con más detalle el generador BBS, que consiste en iterar la función cuadrática $x^2 \pmod{n}$ en el conjunto de los restos cuadráticos de \mathbb{Z}_n . Este generador permite plantear un nuevo cripto-

³ Acrónimo de *National Institute of Standards and Technology*, organismo oficial de normalización en Estados Unidos.

sistema, que recibe el nombre de Blum-Goldwasser. Describimos con detalle su protocolo de funcionamiento para cifrar y descifrar un mensaje. Como en los sistemas precedentes, de nuevo la seguridad del sistema se apoya en la dificultad para factorizar el módulo n ; en otras palabras, si la factorización es conocida, se hace muy sencillo descifrar un criptograma cifrado con este sistema. Así pues, son nuevamente de aplicación los comentarios anteriores acerca de los factores primos de n .

El criptosistema de Blum-Goldwasser ha quedado sustancialmente mejorado merced a los resultados presentados en la referencia [53]. Después de analizar el caso de las órbitas cuadráticas en \mathbb{Z}_p , los autores pasan a su estudio en \mathbb{Z}_n , cuando n es producto de dos factores primos, que corresponde justamente al generador BBS. Pues bien, en la citada referencia se caracterizan los factores primos de n que proporcionan las órbitas de longitud máxima, así como también las semillas que las generan. Este resultado permite modificar el criptosistema original de Blum-Goldwasser introduciendo unas mejoras que permiten plantear también un protocolo de firma digital utilizando esta nueva versión del criptosistema. Con esto cerramos esta sección.

Para terminar este capítulo, dedicamos una última sección a establecer una comparación entre los clásicos criptosistemas de clave secreta y los más modernos de clave pública. Como suele ocurrir, ninguno de los dos anula al otro, sino que resultan complementarios reforzando cada uno de ellos las carencias del otro. Cabe decir entonces que la investigación en ambos tipos de sistemas sigue gozando de gran interés por sus amplias aplicaciones prácticas.

Capítulo 3: Tests de primalidad y otros algoritmos empleados

Dedicamos este capítulo a describir los algoritmos que van a servir de herramienta para el desarrollo de los resultados de la memoria y además ayudan a entender algunas definiciones.

Para comenzar, justificamos brevemente el lugar que hemos asignado a los diversos algoritmos que aparecen en la memoria. Los que hacen referencia a los resultados principales se encuentran colocados en su lugar correspondiente; hemos agrupado, en cambio, en este capítulo aquellos otros que son de dominio público y sirven de base para el desarrollo de los demás.

Sin duda el algoritmo más importante en este trabajo es el que realiza el *test de primalidad*. Damos comienzo, pues, a la siguiente sección explicando la noción de test de primalidad y la distinción que existe entre *test de primalidad* y *test de composición*, que también se llama *test probabilístico*, pues no determina con total seguridad la primalidad del candidato, sino con una probabilidad que se puede hacer tan alta como se quiera.

El test probabilístico más conocido y utilizado actualmente es el *test de Miller-Rabin* (véanse [76, 92]). Dedicamos la siguiente sección a explicar su fundamento y después presentamos el algoritmo propiamente dicho. Para este y sucesivos algoritmos, utilizamos una mezcla de lenguaje de programación y lenguaje natural que facilite la comprensión a los lectores menos familiarizados con el desarrollo de software. Este algoritmo comprueba la primalidad del candidato y devuelve una respuesta que es correcta con probabilidad $1 - 2^{-2t}$, donde t es un parámetro

entero ajustable. Seguidamente exponemos el tiempo teórico de computación de este algoritmo.

Otro de los tests más populares es el *test de Solovay-Strassen* (véase, por ejemplo, [25]). A efectos de comparación, describimos las ventajas que el test de Miller-Rabin tiene frente a él. De hecho son tales que lo han desplazado por completo en la práctica: nosotros no haremos uso de él.

En la siguiente sección presentamos una sencilla mejora del algoritmo de Miller-Rabin que permite disminuir a una décima parte el tiempo de computación del test original.

Presentamos a continuación un ejemplo de test de primalidad determinista: se trata del *test de Pocklington-Lehmer*. Apoyándose en un antiguo resultado de Pocklington (véase [87]), Brillhart *et al.* desarrollaron en [18] un test determinista para comprobar un candidato n . El inconveniente es que necesita una factorización parcial de $n - 1$. Hay algunos casos, sin embargo, en que esto no supone mucho problema, como es, por ejemplo, comprobar si n es un primo 1-seguro, pues en este caso la factorización de $n - 1$ ha de ser, por definición $n - 1 = 2q$, siendo q un primo.

Aunque a lo largo de la memoria no se hará uso de ellos, consideramos de interés introducir en este momento los *algoritmos de factorización de Pollard* (véase [89]) y de *Williams* (véase [122]), conocidos como los *algoritmos de tipo $p \pm 1$* , ahora justificaremos por qué.

Explicamos en la siguiente sección el algoritmo de Pollard, presentando primero su justificación teórica y después el algoritmo propiamente dicho. Supongamos que el entero n es el candidato a ser factorizado y que es producto de tan solo dos factores primos p y q . Pues bien, el punto de interés reside en el hecho de que este algoritmo resulta eficiente precisamente cuando uno de esos factores primos, por ejemplo p , es tal que todos los factores primos de $p - 1$ son menores que una cota más o menos “pequeña”. El algoritmo de Pollard admite una segunda fase si la cota que se ha propuesto resulta demasiado baja: describimos esta segunda fase y aportamos la descripción del algoritmo. En todo caso, es claro que forma de frustrar este algoritmo consiste en asegurar que los factores primos de $p - 1$ y de $q - 1$ son “grandes”.

En la siguiente sección nos centramos en el algoritmo de Williams. Este algoritmo se basa en las sucesiones de Lucas y, de una forma análoga al de Pollard, resulta eficiente cuando los factores primos de $p + 1$, donde p es un factor primo de n , son todos menores de cierta cota. Así pues, para frustrar este algoritmo, nos encontramos también en la precisión de asegurar que los factores primos de $p + 1$ y de $q + 1$ son suficientemente “grandes”. El algoritmo de Williams admite también una segunda fase que describimos para finalizar la sección.

Con lo dicho queda plenamente justificado el nombre de “algoritmos tipo $p \pm 1$ ” que reciben los de Pollard y Williams. Es claro también el interés en usar primos con propiedades especiales como factores de n si queremos frustrar sus ataques. Ello justifica también la definición que se dio de primo robusto en el resumen del trabajo y el interés que para los criptosistemas de clave pública presenta esta clase de primos.

La siguiente parte del capítulo está dedicada a presentar un conjunto de algoritmos que permiten generar números pseudo-aleatorios. Comenzamos con el ge-

nerador BBS, ya descrito teóricamente en el capítulo anterior. Ofrecemos después dos algoritmos de “peso ligero”, que presentan una velocidad de ejecución mucho mayor que la del generador BBS. En primer lugar describimos un generador atribuido a Lehmer y que implementamos de acuerdo a la referencia [85]; se trata en realidad de un generador lineal recursivo modular, del tipo $x_{n+1} \equiv Kx_n \pmod{m}$. El segundo lo describe Tausworthe en [114] y lo implementamos de acuerdo a las mejoras que L’Ecuyer proporciona en la referencia [59].

Por último, para cerrar el capítulo, ofrecemos un algoritmo sencillo que permite generar primos de tamaño aleatorio utilizando alguno de los generadores de números pseudo-aleatorios anteriormente descritos.

Capítulo 4: Primos seguros

En este capítulo presentamos los principales resultados de nuestro trabajo concernientes a una de las clases de primos especiales consideradas, a saber, los *primos seguros*.

En primer lugar extendemos la definición existente de primos 1- y 2-seguros introduciendo la noción de *primo k-seguro con signatura*: es la Definición 4.1. ExPLICAMOS A CONTINUACIÓN LAS PROPIEDADES QUE SE PUEDEN DEDUCIR DE FORMA ELEMENTAL Y LO COMPARAMOS CON LAS DEFINICIONES YA EXISTENTES DE PRIMO 1-SEGUR Y DE *PRIMO DE SOPHIE GERMAIN*. DEFINIMOS EL CONCEPTO DE *SIGNATURA ALTERNADA* Y DEMOSTRAMOS QUE LOS CONJUNTOS DE PRIMOS DE SIGNATURA ALTERNADA ESTÁN VACÍOS CUANDO $k \geq 5$. ESTAS DEFINICIONES FORMALIZAN LA IDEA DE *CADENA DE PRIMOS SEGUREOS* Y, COMO EJEMPLO, APORTAMOS LOS CARDINALES DE LOS CONJUNTOS DE TODAS LAS SIGNATURAS HALLADAS AL RECORRER LOS NÚMEROS PRIMOS HASTA $n = 7500000$, LISTANDO TAMBIÉN HASTA LOS CINCUENTA PRIMEROS ELEMENTOS DE CADA SIGNATURA. COMO DATO INTERESANTE, DE ENTRE LOS PRIMOS HASTA n EXISTEN UN TOTAL DE 460416 QUE NO ESTÁN EN NINGUNA SIGNATURA, ES DECIR, NO SON SEGUREOS. NO ES CONOCIDO SI EXISTEN CADENAS DE PRIMOS SEGUREOS DE LONGITUD ARBITRARIA, AUNQUE APORTAMOS ALGUNOS RESULTADOS AL RESPECTO, COMO, POR EJEMPLO, [125]. PARA CERRAR ESTA SECCIÓN, SE INCLUYEN LOS ALGORITMOS QUE PERMITEN GENERAR PRIMOS 1-SEGUREOS Y 2-SEGUREOS.

En las siguientes secciones presentamos los resultados principales de esta memoria concernientes a los primos 1-seguros, 2-seguros y k -seguros. Aportamos en cada una de ellas los resultados que hemos obtenido relativos a la densidad de este clase de primos en el conjunto de los números primos.

Primero se examina π_1^+ , que es la función recuento para los primos 1-seguros de signatura positiva. Se desarrolla una justificación de tipo heurístico para llegar a que

$$\pi_1^+(x) \simeq \frac{C}{2} \int_5^x \frac{dt}{\ln t \ln \frac{t-1}{2}},$$

donde C es la constante de los primos gemelos, cuyo valor es

$$C = 2 \prod_{p \geq 3} \left(1 - \frac{1}{(p-1)^2} \right),$$

cuando p recorre los números primos. Para evaluar la exactitud del resultado, hemos realizado el experimento de calcular el número exacto de primos 1-seguros

hasta 2^{32} y contrastarlo con lo obtenido en forma teórica. Ofrecemos el resultado en la figura 4.1, que registra una magnífica coincidencia.

Después acometemos igual tarea para π_1^- , la función recuento de los primos 1-seguros con signatura negativa. El resultado es completamente análogo al de sus homónimos de signatura positiva, pues se obtiene

$$\pi_1^- \simeq \frac{C}{2} \int_3^x \frac{dt}{\ln t \ln \frac{t+1}{2}},$$

resultado que concuerda también muy bien con los valores de recuento obtenidos experimentalmente.

Demostrar estos resultados de forma rigurosa excedería los propósitos de esta memoria, si bien encuentran confirmación exacta con los datos experimentales. Sin embargo, con el fin de reforzar nuestros argumentos, presentamos también el trabajo del profesor Cai (véase [21]) en el que ataca el problema de la función recuento de los primos de Sophie Germain. Los resultados a que llega son totalmente congruentes con los nuestros.

Siguiendo un camino análogo al caso de los 1-seguros, pasamos ahora al problema de encontrar las funciones recuento para los primos 2-seguros de signatura positiva y negativa. Un razonamiento heurístico conduce al siguiente resultado

$$\pi_2^+(x) \simeq \frac{9}{8} C_2 \int_{11}^x \frac{dt}{\ln t \ln \frac{t-1}{2} \ln \frac{t-3}{4}},$$

donde

$$C_2 = \prod_{p \geq 5} \frac{p^2(p-3)}{(p-1)^3},$$

extendiendo el producto a todos los primos mayores o iguales a 5. Nuevamente contrastamos este resultado teórico con el experimento de calcular exactamente el valor de la función recuento de primos 2-seguros para $p < 2^{32}$. El resultado, que proporciona una coincidencia muy aproximada con lo predicho por la teoría, es presentado en la figura 4.2. Por muy similares caminos se llega a un resultado similar para el conjunto de primos 2-seguros de signatura negativa, para los cuales la función recuento es

$$\pi_2^-(x) \simeq \frac{9}{8} C_2 \int_5^x \frac{dt}{\ln t \ln \frac{t+1}{2} \ln \frac{t+3}{4}},$$

nuevamente en franco acuerdo con los valores experimentales.

Finalmente, para cerrar este capítulo, los resultados obtenidos por Bateman, Horn y Schinzel, entre otros (véase al respecto [7, 8, 102]) nos han permitido extender los resultados anteriores al caso de primos k -seguros, con lo que obtenemos una fórmula general. De hecho, finalizamos nuestro capítulo aplicando esa fórmula general al caso de los primos 2-seguros y comprobamos que se obtiene un resultado asintóticamente idéntico.

Como resumen, este capítulo ofrece las fórmulas para las funciones recuento generalizadas de los primos k -seguros, las cuales permiten conocer la densidad de tales primos en un entorno cualquiera y, por lo tanto, determinar la probabilidad de obtener uno de ellos mediante selección aleatoria en ese entorno.

Capítulo 5: Primos robustos

Llegamos, con este capítulo, a los resultados centrales de esta memoria. Comenzamos a tratar aquí con los primos que la literatura denomina “strong”, término anglosajón que proponemos hispanizar con el vocablo *robusto*. Esta clase de primos resulta de aplicación en la mayoría de los sistemas criptográficos de clave pública que necesitan protegerse frente a algoritmos de factorización como los que presentamos en el capítulo 3, pues esos algoritmos dejan de ser eficientes si los factores primos elegidos para el módulo n del criptosistema son de tipo robusto. Por esta razón, dedicamos este capítulo a su estudio.

En primer lugar revisamos las definiciones que existen en la literatura. Ciertamente esto resulta un primer obstáculo, por dos razones principales. De una parte, no existe un acuerdo completo en cuanto a qué deba entenderse por primo robusto: se pueden encontrar fácilmente tres definiciones distintas; de otra parte, las definiciones son siempre más bien cualitativas, lo que impide caracterizar un primo como robusto de forma unívoca.

Repasamos en principio la noción estándar más frecuente en la literatura, también llamada primo robusto de 3 vías (es la que aparece, por ejemplo en [73, Nota 8.8]), justificando cada uno de los requisitos. A continuación presentamos la noción más estricta de M. Ogiwara (véase [83]), denominada también primo robusto de 6 vías. Existe, sin embargo, una amplia discusión en la comunidad criptográfica respecto a la conveniencia o necesidad del uso de esta clase de primos especiales y no parece que haya acuerdo en este punto. Por ello presentamos también una selección de ventajas e inconvenientes que puede representar el uso de esta clase de primos. Se puede afirmar, desde luego, que la discusión está lejos de estar cerrada.

Pasamos con esto a presentar una de las partes más novedosas de esta memoria. Introducimos el concepto de primo robusto óptimo. Como ya se apuntaba en párrafos anteriores, las definiciones que existen para los primos robustos son todas ellas de tipo cualitativo. Nuestra novedad consiste en introducir una función, que hemos denominado σ , que puede proporcionar una cierta “medida” cuantitativa acerca de la *robustez* del candidato a primo robusto. La función

$$\sigma: \mathbb{N} \setminus \{1, 2\} \rightarrow \mathbb{N}$$

aparece en la Definición 5.9 como

$$\sigma(n) = \frac{n-1}{S(n-1)} + \frac{n+1}{S(n+1)} + \frac{S(n-1)-1}{S(S(n-1)-1)},$$

donde $S(n)$ representa el factor primo mayor del entero n si $n \geq 2$ y $S(1) = 1$. Con esto, el resultado principal es el Teorema 5.12, que nos dice que para todo primo $p \geq 23$ se verifica que $\sigma(p) \geq 12$.

Este resultado permite disponer de un criterio cuantitativo para determinar el grado de robustez de un candidato a primo robusto y, de rechazo, permite caracterizar un tipo de primos robustos que hemos denominado óptimos. En efecto, es inmediato asignar tal carácter a aquellos primos que hagan mínimo el valor de la función σ : esto es precisamente lo que nos presenta el Corolario 5.13, que nos dice que un primo p es robusto óptimo cuando $\sigma(p) = 12$. Los siguientes resultados son

el Teorema 5.14 y el Corolario 5.15 que proporcionan unas condiciones operativas para decidir si un primo es robusto óptimo.

Tras presentar unas gráficas en que vemos el aspecto de la función σ para argumentos enteros cualesquiera y para argumentos primos, pasamos al punto clave de la distribución y densidad de los recién introducidos primos robustos óptimos. Denotamos por π_σ la función recuento de este tipo de primos y desarrollamos el argumento heurístico que nos permite llegar al resultado siguiente:

$$\pi_\sigma(x) \simeq \frac{1}{12} C_\sigma \int_{31}^x \frac{dt}{(\ln \frac{t-7}{12})^4},$$

donde

$$C_\sigma = \frac{42875}{6144} \prod_{p>7} \frac{p^3(p-4)}{(p-1)^4},$$

extendido el productorio a todos los primos mayores que 7. Para evaluar la exactitud del resultado, hemos realizado el experimento de calcular el número exacto de primos robustos óptimos hasta $5 \cdot 10^7$ y contrastarlo con lo obtenido en forma teórica. Ofrecemos el resultado en la figura 5.5.

En la siguiente sección del capítulo nos ocupamos de presentar los algoritmos ya existentes en la literatura que permiten calcular primos robustos. El más consagrado se debe a J. Gordon quien lo presentó en la referencia [47]. Nosotros damos una implementación práctica así como una tabla de primos robustos obtenida con este algoritmo. La observación interesante es que, aplicada la función σ a cada uno de los primos así generados, se obtiene un valor que está muy lejos del óptimo. Los primos generados son robustos, ciertamente, pero su grado de robustez es discutible.

Seguidamente aplicamos este mismo ejercicio a otro algoritmo muy interesante debido a M. Ogiwara (véase [83]): proporcionamos la descripción y los resultados experimentales obtenidos al generar primos robustos con este algoritmo. Presentamos también en la misma tabla el resultado de aplicar la función σ a cada uno de los primos obtenidos y, al igual que en el caso del algoritmo de Gordon, se comprueba que el grado de robustez es más bien bajo y los valores de σ pueden llegar a ser del orden de 10^{24} , cuando los primos generados son del orden de 10^{44} .

Cerramos entonces el capítulo explicando el algoritmo que nosotros propone mos para generar primos robustos óptimos. Por medio de él, hemos realizado un experimento numérico consistente en generar una tabla que contiene primos de esta clase para un rango comprendido entre 2^{24} y 2^{123} . Ofrecemos también unas gráficas en que se puede ver el número de ensayos necesarios para obtener uno de ellos para diversos valores comprendidos en el citado rango.

Capítulo 6: Aplicaciones criptográficas de los primos especiales

En este capítulo trataremos de mostrar las aplicaciones criptográficas de los primos especiales que se han ido considerando en los capítulos precedentes.

Consideramos en primer lugar el criptosistema RSA y, tras recordar los requisitos criptoanalíticos para los factores primos del módulo, se propone el uso de los primos 1-seguros o bien de los primos robustos óptimos.

Pasamos revista a continuación al criptosistema BBS y mostramos cómo el uso de primos 2-seguros permite garantizar las órbitas de periodo máximo. Éstas también se pueden garantizar si se utilizan primos 1-seguros, siempre que se satisfagan algunas condiciones adicionales. En todo caso, ambas clases de primos resultan del máximo interés.

Antes de pasar a exponer resultados prácticos, exponemos en unos breves párrafos el problema del cambio de clave en los criptosistemas de clave pública, haciendo ver que, entre otras cosas, es necesario conocer de antemano el tiempo de computación necesario para la obtención de primos de las diversas clases consideradas, que pasamos a presentar en las secciones subsiguientes.

Iniciamos esta parte tratando de los primos 1-seguros. En primer lugar, se da una estimación teórica acerca del número de ensayos necesario para conseguir un primo de esta clase con un tamaño en bits determinado; a estos ensayos los denominamos también ‘tiradas’. Es claro que esta información se puede obtener a partir de la función recuento, pues, conocida ésta, su derivada en un punto será precisamente la densidad de primos 1-seguros en ese punto. Ahora bien, la inversa de la densidad nos indica justamente el número promedio de ensayos a realizar para obtener uno de ellos en ese punto. Puesto que cada ensayo comporta un cierto tiempo de computación, estamos en condiciones de saber cuánto será ese tiempo en cada punto. Así derivamos en esta sección el tiempo de computación teórico para los primos 1-seguros. Hecha la estimación teórica, se acompañan unos datos de computación reales hechos sobre una plataforma de computación concreta, cuyas características se aportan. Por último, damos las gráficas de los datos experimentales y de su ajuste a los valores teóricos.

El resto del capítulo se completa repitiendo el mismo esquema para los primos 2-seguros y, por fin, para los primos robustos óptimos.

Capítulo 7 y Apéndice I

El capítulo 7 está dedicado a presentar las conclusiones principales del trabajo, incluyendo también un elenco de posibles líneas de futuros desarrollos sobre las materias estudiadas en esta memoria.

De igual manera, ofrecemos en el Apéndice I el código fuente de la implementación de los algoritmos más importantes aparecidos a lo largo del trabajo.

Capítulo 1

Preliminares

Resumen del capítulo

Se introducen las principales herramientas matemáticas necesarias para el desarrollo del resto del trabajo. Se presenta una breve introducción a la teoría de la complejidad computacional, que permite definir el tiempo de ejecución de un algoritmo y clasificarlo. Finalmente se introducen también las herramientas computacionales, básicamente bibliotecas de programación y aplicaciones de computación simbólica, utilizadas en la memoria.

1.1 Herramientas matemáticas

Dedicamos esta sección a proporcionar una colección de herramientas matemáticas de las que se hará amplio uso a lo largo de la memoria. Estas herramientas cubren una serie de ramas de las matemáticas, como son la teoría de números, el álgebra abstracta, los cuerpos finitos, la teoría de la complejidad computacional o la estadística.

Naturalmente, no tenemos ninguna pretensión de ser exhaustivos ni completamente formales. Tan sólo pretendemos facilitar la lectura secuencial de la memoria, procurando mantener al mínimo la necesidad de acudir a fuentes externas.

El lector interesado en profundizar en alguna de las herramientas presentadas puede consultar las referencias que proporcionamos en la parte final.

1.1.1 Notación

Utilizaremos la siguiente notación típica:

1. \mathbb{N} representa el conjunto de los números naturales, es decir,

$$\mathbb{N} = \{0, 1, 2, \dots\}.$$

2. \mathbb{Z} representa el conjunto de los números enteros, es decir,

$$\mathbb{Z} = \{\dots, -2, -1, 0, 1, 2, \dots\}.$$

3. \mathbb{Q} denota el conjunto de los números racionales, es decir,

$$\mathbb{Q} = \left\{ \frac{a}{b} : a, b \in \mathbb{Z}, b \neq 0 \right\}.$$

4. \mathbb{R} denota el conjunto de los números reales.
5. Las constantes π y e habitualmente tienen su significado típico. No obstante, la letra π también representa las funciones recuento de diversos tipos de números primos y la letra e se utiliza como exponente entero.
6. La expresión $[a, b]$ denota el intervalo cerrado de extremos a y b ; esto es, $[a, b] = \{x \in \mathbb{R} : a \leq x \leq b\}$.
7. La expresión $\lceil x \rceil$ significa el menor entero $a \in \mathbb{Z}$ que verifica $x \leq a$.
8. La expresión $\lfloor x \rfloor$ significa el mayor entero $a \in \mathbb{Z}$ que verifica $x \geq a$.
9. Representamos el número de elementos de un conjunto finito A por $\#A$, al que llamamos cardinal de A .
10. Cuando hablamos de la función $\ln x$ nos referimos siempre al logaritmo en base natural e . En cualquier otro caso diremos explícitamente a qué base nos referimos, usando la notación $\log_b x$, es decir, logaritmo en base b de x .

A la hora de decidir un formato idóneo para presentar los algoritmos, hemos tratado de huir de los dos extremos: ni una presentación demasiado verbal, que no deje ver fácilmente cómo se puede implementar aquello en la práctica; ni una presentación en puros términos de un lenguaje de programación concreto, que lo haga ininteligible para quien no domina ese lenguaje.

Dicho esto, hemos tratado de mezclar en cantidades razonables la explicación verbal, cuando la programación es farragosa, y la presentación programática cuando lo que se hace farragoso es la explicación verbal. Nos basamos en un subconjunto particular de instrucciones de C —por ser un lenguaje extendido— sacrificando libremente su sintaxis en favor de la claridad y la sencillez. Subdividimos también cada algoritmo en apartados para realzar su estructura lógica. Sólo queda que el lector juzgue si el esfuerzo ha merecido la pena.

1.1.2 Teoría de la divisibilidad

Tratamos en este apartado siempre con números enteros. Comenzamos con la siguiente

Definición 1.1 Sean a, b dos números enteros. Decimos que a divide a b si existe otro entero c tal que $b = ac$. Denotamos este hecho con la expresión $a|b$. Decimos también que a es un divisor o un factor de b .

Escribimos $a \nmid b$ para expresar que a no divide a b .

Sean $a, b, c \in \mathbb{Z}$. Algunas propiedades elementales de la divisibilidad son las siguientes:

1. $a|a$.
2. Si $a|b$ y $b|c$ entonces $a|c$.
3. Si $a|b$ y $a|c$ entonces $a|(bx + cy)$ para todo $x, y \in \mathbb{Z}$.
4. Si $a|b$ y $b|a$ entonces $a = \pm b$.

La reglas elementales de la división permiten escribir que si a y b son enteros, con $b > 0$, entonces la división de a entre b da como resultado un cociente c y un resto r , tales que

$$a = cb + r,$$

donde $0 \leq r < b$; además c y r son únicos.

Teorema 1.2 (de Euclides) Si un número divide el producto de dos enteros, entonces divide al menos a uno de ellos.

Introducimos ahora el llamado teorema fundamental de la aritmética:

Teorema 1.3 Todo número entero $n \geq 2$ admite una factorización única como producto de potencias de primos:

$$n = p_1^{e_1} p_2^{e_2} \cdots p_k^{e_k},$$

donde los p_i son primos tales que $p_1 < \dots < p_k$ y los e_i son números positivos.

Definición 1.4 Dados a, b, c enteros, llamamos a c común divisor de los enteros a y b si $c|a$ y $c|b$.

Definición 1.5 Decimos que un entero positivo d es el máximo común divisor de dos enteros positivos a y b si

1. d es un común divisor de a y de b , y
2. si otro entero c verifica que $c|a$ y $c|b$, entonces $c|d$.

Denotamos este hecho como $d = \text{mcd}(a, b)$.

Definición 1.6 Decimos que un entero positivo l es el mínimo común múltiplo de dos enteros positivo a y b si

1. $a|l$ y $b|l$, y
2. si otro entero c verifica que $a|c$ y $b|c$, entonces $l|c$.

Denotamos este hecho como $l = \text{mcm}(a, b)$.

Es fácil ver que $a \cdot b = \text{mcd}(a, b) \cdot \text{mcm}(a, b)$.

Definición 1.7 Decimos que a y b son primos entre sí (o coprimos) si verifican que $\text{mcd}(a, b) = 1$.

Para calcular el máximo común divisor de dos números enteros basta con multiplicar sus factores comunes, como es fácil ver a partir de la definición. Sin embargo, en la práctica este procedimiento es poco útil si es difícil determinar los factores de los enteros en cuestión, lo que suele ocurrir si esos números son grandes. Para resolver este problema, se acude al algoritmo de Euclides, que está basado en el siguiente

Teorema 1.8 (de la división de Euclides) Dados dos números enteros $a > b > 0$ se verifica

$$\text{mcd}(a, b) = \text{mcd}(b, r),$$

donde r es el resto de la división de a por b , es decir, $a = bc + r$, con $b > r$.

Demostración Puesto que $a = bc + r$, si $d|a$ y $d|b$, es obvio que también $d|r$, y recíprocamente. Por tanto, el máximo común divisor de a y b es también el máximo común divisor de b y de r . ■

Con ayuda del teorema de la división de Euclides, se tiene el

Algoritmo 1.9 (de Euclides) Este algoritmo tiene como entrada dos enteros $a > b > 0$, y como salida el $\text{mcd}(a, b)$. Consiste en realizar divisiones sucesivas en las que se toma como dividendo y como divisor el divisor y el resto de la división anterior respectivamente, hasta que el resto se haga cero. El último divisor es el máximo común divisor.

LLAMADA: `Euclides(a, b);`

ENTRADA: Dos enteros, $a > b > 0$.

SALIDA: $\text{mcd}(a, b)$.

1. [Lazo]

`while (b > 0)`

{

`r ≡ a(mod b);`

`a = b;`

`b = r;`

}

`return a;`

Proposición 1.10 Sean $a > b > 0$ y sea $R = \frac{1}{2}(1+\sqrt{5})$. El número de iteraciones del lazo es, como máximo,

$$\frac{\ln b}{\ln R} + 1.$$

Demostración Véase, por ejemplo, [19, sección 1.8]. ■

Ejemplo 1.11 Para este ejemplo, hagamos $r_0 = a$ y $r_1 = b$, con $a > b$, y además, para $k \geq 1$,

$$r_{k+1} = r_{k-1} - c_k r_k, \quad (1.1)$$

donde

$$c_k = \left\lfloor \frac{r_{k-1}}{r_k} \right\rfloor.$$

Si queremos calcular $\text{mcd}(210, 91)$, utilizamos la siguiente tabla

k	0	1	2	3	4
r_k	210	91	28	7	0
c_k		2	3	4	

que se puede leer como

$$\begin{aligned} 210 &= 91 \cdot 2 + 28 \\ 91 &= 28 \cdot 3 + 7 \\ 28 &= 7 \cdot 4 + 0 \end{aligned}$$

o, de otra forma,

$$\text{mcd}(210, 91) = \text{mcd}(91, 28) = \text{mcd}(28, 7) = 7.$$

Del cómputo del máximo común divisor por el algoritmo de Euclides se deduce la siguiente propiedad

Proposición 1.12 Si $d = \text{mcd}(a, b)$, entonces existen números enteros, u, v , tales que:

$$d = u \cdot a + v \cdot b.$$

Demostración En efecto, es fácil ver que si vamos despejando los restos r_k en las divisiones sucesivas, podemos finalmente expresar el máximo común divisor en función de a y b . Este método se conoce como algoritmo de Euclides extendido. De la expresión (1.1), podemos escribir lo siguiente:

$$\begin{aligned} r_2 &= r_0 - c_1 r_1, \\ r_3 &= r_1 - c_2 r_2 = -c_2 r_0 + (1 + c_1 c_2) r_1, \\ r_4 &= r_2 - c_3 r_3 = (1 + c_2 c_3) r_0 - (c_1 + c_3 (1 + c_1 c_2)) r_1, \\ &\dots \end{aligned}$$

El último resto no nulo es, como sabemos, el máximo común divisor, que quedará expresado como combinación lineal de $r_0 = a$ y de $r_1 = b$. ■

Observación 1.13 Es de notar que la Proposición 1.12 establece la existencia de los valores u y v , pero no su unicidad. Sin embargo, todos los posibles valores de u son congruentes módulo b/d y todos los posibles valores de v son congruentes módulo a/d .

Basándose en la prueba de la Proposición 1.12, se puede desarrollar el siguiente algoritmo, que requiere muy poca memoria auxiliar:

Algoritmo 1.14 (de Euclides extendido) Este algoritmo recibe como entradas dos enteros $a > b > 0$ y proporciona a su salida números u, v, d tales que $d = u \cdot a + v \cdot b$ y $d = \text{mcd}(a, b)$. Usaremos como variables auxiliares v_1, v_3, t_1, t_3 .

LLAMADA: `EuclidesExt(a, b);`

ENTRADA: Dos enteros, $a > b > 0$.

SALIDA: $d = \text{mcd}(a, b)$ y u, v , tales que $d = ua + vb$.

1. [Inicialización]

 $u = 1;$ $d = a;$ $v_1 = 0;$ $v_3 = b;$

2. [Lazo]

while ($v_3 > 0$)

{

 $q = \frac{d}{v_3};$ $t_3 \equiv d(\text{mod } v_3);$ $t_1 = u - q \cdot v_1;$ $u = v_1;$ $d = v_3;$ $v_1 = t_1;$ $v_3 = t_3;$

}

 $v = \frac{d - u \cdot a}{b};$ **return** $u, v, d;$

Ejemplo 1.15 Si hacemos, como antes, $a = 210$ y $b = 91$, tenemos

$$28 = 210 - 91 \cdot 2,$$

de aquí

$$91 = 3(210 - 91 \cdot 2) + 7,$$

de donde

$$7 = 91 - 3(210 - 91 \cdot 2) = -3 \cdot 210 + 7 \cdot 91,$$

es decir, $u = -3$ y $v = 7$.

Definición 1.16 Sea $n \geq 1$ un entero. Denotamos por $\varphi(n)$ el número de enteros en el intervalo $[1, n]$ que son primos con n . Esta función se denomina indicador o función φ de Euler.

Veamos algunas propiedades del indicador de Euler que serán de interés en adelante:

1. Si p es un primo, entonces $\varphi(p) = p - 1$.
2. La función φ de Euler es multiplicativa. Esto significa que dados dos enteros, m y n , tales que $\text{mcd}(m, n) = 1$, entonces $\varphi(m \cdot n) = \varphi(m) \cdot \varphi(n)$.
3. Si $n = p^e$, entonces $\varphi(n) = \varphi(p^e) = (p - 1)p^{e-1}$.

4. Si $n = p_1^{e_1} p_2^{e_2} \cdots p_k^{e_k}$, entonces de las anteriores propiedades se deduce que

$$\varphi(n) = n \prod_{i=1}^k \left(1 - \frac{1}{p_i}\right).$$

5. Si n es un entero, entonces

$$\sum_{d|n, d>0} \varphi(d) = n.$$

Los enteros módulo n

En lo que sigue, consideramos un entero $n \geq 2$.

Definición 1.17 Dados dos enteros a, b , decimos que a es congruente con b módulo n si $n|(a - b)$. Denotamos esta propiedad por $a \equiv b \pmod{n}$. El entero n es el módulo de la congruencia.

Enunciemos ahora las propiedades básicas de las congruencias que se usarán frecuentemente en esta memoria. Sean a, a_1, b, b_1, c números enteros.

1. $a \equiv b \pmod{n}$ si y sólo si a y b dejan el mismo resto cuando se dividen por n .
2. $a \equiv a \pmod{n}$ (propiedad reflexiva).
3. Si $a \equiv b \pmod{n}$, entonces $b \equiv a \pmod{n}$ (propiedad simétrica).
4. Si $a \equiv b \pmod{n}$ y $b \equiv c \pmod{n}$, entonces $a \equiv c \pmod{n}$ (propiedad transitiva).
5. Si $a \equiv a_1 \pmod{n}$ y $b \equiv b_1 \pmod{n}$, entonces $a+b \equiv a_1+b_1 \pmod{n}$ y además $ab \equiv a_1b_1 \pmod{n}$.
6. Si d es un divisor común de a y b , o sea $a = da'$, $b = db'$ y $\text{mcd}(d, n) = 1$, entonces $a \equiv b \pmod{n}$ implica $a' \equiv b' \pmod{n}$, como se sigue directamente del teorema de Euclides.

Las propiedades anteriores permiten establecer una relación de equivalencia en que las clases de equivalencia de un entero a son el conjunto de todos los enteros congruentes con a módulo n . Para n fijado, la relación de congruencia módulo n partitiona el conjunto \mathbb{Z} en clases de equivalencia.

También es claro que si escribimos $a = cn + r$ con $0 \leq r < n$, entonces $a \equiv r \pmod{n}$. Por lo tanto, cada entero a es congruente a un único entero r tal que $r \in [0, n-1]$ al que llamamos el resto mínimo de a módulo n . Puesto que a y r están en la misma clase de equivalencia, r puede ser considerado el representante canónico de esa clase.

Definición 1.18 Denotamos por \mathbb{Z}_n el conjunto de las clases de equivalencia de los enteros módulo n .

Puesto que las clases admiten como representante canónico un entero $r \in [0, n - 1]$, es obvio que se puede efectuar la siguiente identificación:

$$\mathbb{Z}_n = \{0, 1, 2, \dots, n - 1\}.$$

En este conjunto se pueden realizar las operaciones de suma y producto ordinarias, pero tomándolas siempre módulo n .

Definición 1.19 Sea $a \in \mathbb{Z}_n$. El inverso multiplicativo de a módulo n es un entero (si existe) $x \in \mathbb{Z}_n$ tal que $ax \equiv 1 \pmod{n}$. Si tal entero x existe, entonces es único módulo n , y se dice que a es invertible o también que es una unidad; denotamos el inverso de a por a^{-1} . En este caso, se puede efectuar la división de cualquier elemento $b \in \mathbb{Z}_n$ por a , y su valor es $ba^{-1} \pmod{n}$.

Es fácil ver que si $a \in \mathbb{Z}_n$, entonces a es invertible si y sólo si $\text{mcd}(a, n) = 1$; es decir, si a y n son primos entre sí. Podemos generalizar esto diciendo que la ecuación en congruencias

$$ax \equiv b \pmod{n}$$

tiene solución para x si y sólo si $d = \text{mcd}(a, n)$ divide a b , en el cual caso hay exactamente d soluciones entre 0 y $n - 1$; estas soluciones son todas congruentes módulo n/d .

Cálculo del inverso módulo n Surge ahora de modo natural la cuestión de cómo calcular el inverso de un entero módulo n . Para ello, el resultado de la Proposición 1.12 resulta de aplicación. En efecto, puesto que $a \in \mathbb{Z}_n$ es invertible si sólo si $\text{mcd}(a, n) = 1$, existen enteros u, v tales que:

$$u \cdot a + v \cdot n = 1.$$

Pero si tomamos módulos a cada lado de esta expresión, tendremos

$$u \cdot a + v \cdot n \equiv u \cdot a \equiv 1 \pmod{n}.$$

Por consiguiente, $u = a^{-1} \pmod{n}$. Así pues, utilizando el algoritmo de Euclides extendido, el cálculo del inverso es inmediato. En la sección 1.2, dedicada al estudio del tiempo de ejecución de los algoritmos, explicaremos que el algoritmo de Euclides es eficiente en términos del tiempo de ejecución, con lo que resulta de gran ayuda para el cómputo del inverso.

Ejemplo 1.20 Supongamos que $a = 11$ y $n = 35$. Queremos calcular $\text{mcd}(35, 11)$ utilizando el algoritmo de Euclides. Escribimos la siguiente tabla

k	0	1	2	3
r_k	35	11	2	1
c_k		3	5	

de donde vemos que $\text{mcd}(35, 11) = 1$ y, por tanto, 11 es invertible en \mathbb{Z}_{35} . Siguiendo la tabla, tendremos la siguiente secuencia

$$\begin{aligned} 2 &= 35 - 11 \cdot 3 \\ 1 &= 11 - 5 \cdot 2. \end{aligned}$$

De aquí

$$1 = 11 - 5(35 - 11 \cdot 3) = 16 \cdot 11 - 5 \cdot 35,$$

es decir, $11^{-1} = 16$ en \mathbb{Z}_{35} .

Con todo este material, podémos introducir ahora un teorema conocido de muy antiguo en el ámbito de la teoría de números que recibe el curioso nombre de *teorema chino del resto*. Enunciémoslo:

Teorema 1.21 Sean n_1, n_2, \dots, n_k un conjunto de enteros primos entre sí dos a dos. El sistema de congruencias simultáneas

$$\begin{aligned} x &\equiv a_1 \pmod{n_1} \\ x &\equiv a_2 \pmod{n_2} \\ &\vdots \\ x &\equiv a_k \pmod{n_k} \end{aligned}$$

tiene una solución única módulo $n = n_1 n_2 \cdots n_k$.

Aunque resulta interesante saber que el problema tiene solución, lo que inmediatamente deseamos es un algoritmo que nos permita llegar a ella. Gauss nos ofrece el siguiente:

Algoritmo 1.22 La solución x del sistema de congruencias simultáneas en el teorema chino del resto se puede calcular como

$$x = \sum_{i=1}^k a_i N_i M_i \pmod{n}$$

donde $N_i = n/n_i$ en \mathbb{Z} y $M_i = N_i^{-1} \pmod{n_i}$. Observemos que N_i siempre es invertible módulo n_i pues $\text{mcd}(N_i, n_i) = 1$, por construcción.

Se dice que los generales chinos contaban la fuerza presente ordenando a la tropa disponerse sucesivamente en grupos de tamaño un número primo. Los soldados que quedaban sobrantes representan los distintos valores a_i que se obtienen al hacer la congruencia tomando como módulo p_i ; además, por ser primos, cumplen de inmediato la condición de ser primos entre sí, por lo que se puede aplicar el teorema del resto. Como la solución se obtiene módulo n , y $n = \prod_{i=1}^k p_i$, k ha de ser tal que se estime $x < n$, siendo x el número total de soldados.

El grupo multiplicativo \mathbb{Z}_n^*

Introducimos ahora una estructura algebraica que será de interés en lo que sigue, por lo que trataremos de ella con cierto detalle: se trata del grupo multiplicativo \mathbb{Z}_n^* .

Definición 1.23 Definimos \mathbb{Z}_n^* como el conjunto de las unidades de \mathbb{Z}_n , es decir, el conjunto de elementos invertibles de \mathbb{Z}_n . Por definición, sus elementos son

$$\mathbb{Z}_n^* = \{a \in \mathbb{Z}_n : \text{mcd}(a, n) = 1\}.$$

En particular, si n es un primo, $\mathbb{Z}_n^* = \mathbb{Z}_n \setminus \{0\}$.

Proposición 1.24 El conjunto \mathbb{Z}_n^* dotado de la operación multiplicación modular presenta estructura algebraica de grupo.

Demostración Véase, por ejemplo, [19, sección 2.8]. ■

Se sigue de la Definición 1.16 que el orden de \mathbb{Z}_n^* es $\#\mathbb{Z}_n^* = \varphi(n)$.

Teoremas de Euler y Fermat

Pasamos ahora a estos dos teoremas omnipresentes en la teoría de números. Presentamos en primer lugar el teorema de Euler:

Teorema 1.25 Sea $n \geq 2$ un entero. Si $a \in \mathbb{Z}_n^*$, es decir $\text{mcd}(a, n) = 1$, entonces se verifica

$$a^{\varphi(n)} \equiv 1 \pmod{n}.$$

Un caso particular del teorema de Euler ocurre si consideramos que n es un número primo. Se conoce con el nombre de teorema (pequeño) de Fermat, y afirma lo siguiente:

Teorema 1.26 Sea p un primo y sea a un entero tal que $\text{mcd}(a, p) = 1$. Entonces se verifica

$$a^{p-1} \equiv 1 \pmod{p}.$$

De estos dos teoremas se deducen algunas proposiciones interesantes en las aplicaciones:

Proposición 1.27 Sea n un entero. Si se verifica

$$r \equiv s \pmod{\varphi(n)},$$

entonces también se verifica

$$a^r \equiv a^s \pmod{n},$$

para cualquier entero a .

Demostración Observemos que la congruencia

$$r \equiv s \pmod{\varphi(n)}$$

significa

$$r = s + k\varphi(n)$$

para algún entero k . Pero entonces

$$a^r = a^{s+k\varphi(n)} = a^s a^{k\varphi(n)} = a^s \left(a^{\varphi(n)} \right)^k,$$

luego

$$a^r = a^s \left(a^{\varphi(n)} \right)^k \equiv a^s \pmod{n},$$

en virtud del teorema de Euler. ■

Observación 1.28 Observemos que, al hacer exponenciaciones módulo n , la Proposición 1.27 nos facilita reducir los exponentes módulo $\varphi(n)$.

Análogamente obtenemos la siguiente

Proposición 1.29 Sea p un primo. Si se verifica que

$$r \equiv s \pmod{p-1},$$

entonces se verifica también que

$$a^r \equiv a^s \pmod{p},$$

para cualquier entero a .

Demostración Observemos que

$$r \equiv s \pmod{p-1}$$

significa que

$$r = s + k(p-1)$$

para algún entero k . Pero entonces

$$a^r = a^{s+k(p-1)} = a^s a^{k(p-1)} = a^s \left(a^{(p-1)} \right)^k,$$

luego

$$a^r = a^s \left(a^{(p-1)} \right)^k \equiv a^s \pmod{p},$$

en virtud del teorema pequeño de Fermat. ■

Observación 1.30 Observemos que, al trabajar módulo p , la Proposición 1.27 nos permite reducir los exponentes módulo $p-1$. En particular,

$$a^p \equiv a \pmod{p}$$

para todo entero a .

Introducimos ahora el concepto de orden de un elemento de un grupo, aplicándolo en particular al grupo multiplicativo \mathbb{Z}_n^* .

Definición 1.31 Sea $a \in \mathbb{Z}_n^*$. Denominamos orden de a al más pequeño de los enteros positivos r tales que $a^r = 1$. Lo denotamos como $\text{ord}(a)$.

Proposición 1.32 Si el orden de $a \in \mathbb{Z}_n^*$ es r y además $a^s \equiv 1 \pmod{n}$, entonces r divide a s . En particular, r divide a $\varphi(n)$.

Demostración Esto se deduce inmediatamente de la definición de orden de un elemento y del teorema de Euler. ■

Definición 1.33 Sea $g \in \mathbb{Z}_n^*$. Si el orden de g es $\varphi(n)$, entonces se dice que g es un generador o elemento primitivo de \mathbb{Z}_n^* .

Veamos ahora algunas propiedades de los generadores del grupo multiplicativo \mathbb{Z}_n^* :

1. \mathbb{Z}_n^* tiene un generador si y sólo si $n = 2, 4, p^k$, o $2p^k$, donde p es un primo impar (es decir, diferente de 2) y k un entero positivo. Es obvio que \mathbb{Z}_p^* siempre tiene un generador.
2. Si g es un generador de \mathbb{Z}_n^* , entonces se puede describir \mathbb{Z}_n^* así:

$$\mathbb{Z}_n^* = \{g^i \pmod{n} : 0 \leq i < \varphi(n)\}.$$

3. Supongamos que g es un generador de \mathbb{Z}_n^* . Entonces $b = g^i \pmod{n}$ es también un generador de \mathbb{Z}_n^* si y sólo si

$$\text{mcd}(i, \varphi(n)) = 1, \quad 0 \leq i < \varphi(n).$$

Por lo tanto el número de generadores totales será $\varphi(\varphi(n))$.

4. Un elemento $g \in \mathbb{Z}_n^*$ es generador de \mathbb{Z}_n^* si y sólo si

$$g^{\varphi(n)/d} \not\equiv 1 \pmod{n}$$

para cada divisor d de $\varphi(n)$.

Definición 1.34 Se dice que un elemento $q \in \mathbb{Z}_n^*$ es resto cuadrático módulo n si existe otro elemento $x \in \mathbb{Z}_n^*$ tal que $x^2 \equiv q \pmod{n}$. Si no existe tal elemento x , se dice que q es un no-resto cuadrático.

Notación 1.35 El conjunto de todos los restos cuadráticos módulo n se denota por Q_n y su complementario por \bar{Q}_n .

Proposición 1.36 Sea p un primo impar y sea g un generador de \mathbb{Z}_p^* . Entonces, $q \in \mathbb{Z}_p^*$ es un resto cuadrático si y sólo si $q \equiv g^e \pmod{p}$ donde e es un número par. Por lo tanto, $\#Q_p = \frac{1}{2}(p-1)$ y $\#\bar{Q}_p = \frac{1}{2}(p-1)$. La mitad de los elementos de \mathbb{Z}_p^* son restos cuadráticos y la otra mitad no.

Por su interés para los criptosistemas que veremos más adelante, son de notar los siguientes apartados.

Proposición 1.37 Sea n un producto de dos primos distintos, p y q . Entonces un elemento $a \in \mathbb{Z}_n^*$ es resto cuadrático módulo n si y sólo si $a \in Q_p$ y $a \in Q_q$. Se sigue que

$$\#Q_n = \#Q_p \#Q_q = \frac{(p-1)(q-1)}{4}.$$

Definición 1.38 Sea $a \in Q_n$. Si x satisface $x^2 \equiv a \pmod{n}$ entonces x se llama raíz cuadrada de a módulo n .

Es interesante la cuestión de cuántas raíces cuadradas puede presentar un resto cuadrático. Se tiene:

1. Si p es un primo impar y $a \in Q_p$, entonces a tiene exactamente dos raíces cuadradas módulo p .
2. Con toda generalidad, sea $n = p_1^{e_1} p_2^{e_2} \cdots p_k^{e_k}$ donde todos los p_i son diferentes y los $e_i \geq 1$. Si $a \in Q_n$, entonces a tiene exactamente 2^k raíces cuadradas módulo n .

Símbolo de Legendre

El símbolo de Legendre es útil para señalar cuándo un número entero a es un resto cuadrático módulo un primo p . Se tiene la siguiente

Definición 1.39 Sea p un primo impar y a un entero. El símbolo de Legendre $\left(\frac{a}{p}\right)$ se define como

$$\left(\frac{a}{p}\right) = \begin{cases} 0, & \text{si } p|a \\ 1, & \text{si } a \in Q_p \\ -1, & \text{si } a \in \bar{Q}_p \end{cases}$$

Dicho con palabras: para cualquier a que sea primo con respecto a p , el símbolo vale 1 si a es resto cuadrático módulo p , y -1 si no lo es.

Propiedades del símbolo de Legendre Sea p un primo impar y $a, b \in \mathbb{Z}$. El símbolo de Legendre tiene las siguientes propiedades:

1. $\left(\frac{a}{p}\right) \equiv a^{(p-1)/2} (\bmod p)$. En particular, $\left(\frac{1}{p}\right) = 1$ y $\left(\frac{-1}{p}\right) = (-1)^{(p-1)/2}$. Por tanto, $-1 \in Q_p$ si $p \equiv 1 (\bmod 4)$, y $-1 \in \bar{Q}_p$ si $p \equiv 3 (\bmod 4)$.
2. $\left(\frac{ab}{p}\right) = \left(\frac{a}{p}\right) \left(\frac{b}{p}\right)$. De aquí que, si $a \in \mathbb{Z}_p^*$, entonces $\left(\frac{a^2}{p}\right) = 1$.
3. Si $a \equiv b (\bmod p)$, entonces $\left(\frac{a}{p}\right) = \left(\frac{b}{p}\right)$.
4. $\left(\frac{2}{p}\right) = (-1)^{(p^2-1)/8}$. De aquí que $\left(\frac{2}{p}\right) = 1$ si $p \equiv 1$ ó $7 (\bmod 8)$, y $\left(\frac{2}{p}\right) = -1$ si $p \equiv 3$ ó $5 (\bmod 8)$.
5. Si q es un primo impar distinto de p , entonces

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

Es decir, $\left(\frac{p}{q}\right) = \left(\frac{q}{p}\right)$ a no ser que tanto p como q sean congruentes con 3 módulo 4, en el cual caso $\left(\frac{p}{q}\right) = -\left(\frac{q}{p}\right)$. Esta propiedad se denomina ley de la reciprocidad cuadrática.

Símbolo de Jacobi

El símbolo de Jacobi es una generalización del símbolo de Legendre para enteros impares n no necesariamente primos.

Definición 1.40 Sea $n \geq 3$ un número impar con factorización $n = p_1^{e_1} p_2^{e_2} \cdots p_k^{e_k}$. El símbolo de Jacobi $(\frac{a}{n})$ se define como

$$\left(\frac{a}{n}\right) = \left(\frac{a}{p_1}\right)^{e_1} \left(\frac{a}{p_2}\right)^{e_2} \cdots \left(\frac{a}{p_k}\right)^{e_k}.$$

Observemos que si n es primo, el símbolo de Jacobi es simplemente el de Legendre.

Propiedades del símbolo de Jacobi Sean ahora dos enteros impares, $m \geq 3$, $n \geq 3$, y $a, b \in \mathbb{Z}$. El símbolo de Jacobi tiene las siguientes propiedades:

1. De la definición se ve que $(\frac{a}{n}) = 0, 1, \text{ o } -1$. Además, $(\frac{a}{n}) = 0$ si y sólo si $\text{mcd}(a, n) \neq 1$.
2. $(\frac{ab}{n}) = (\frac{a}{n})(\frac{b}{n})$. De aquí, si $a \in \mathbb{Z}_n^*$, entonces $(\frac{a^2}{n}) = 1$.
3. $(\frac{a}{mn}) = (\frac{a}{m})(\frac{a}{n})$.
4. Si $a \equiv b \pmod{n}$, entonces $(\frac{a}{n}) = (\frac{b}{n})$.
5. $(\frac{1}{n}) = 1$.
6. $(\frac{-1}{n}) = (-1)^{(n-1)/2}$. De aquí $(\frac{-1}{n}) = 1$ si $n \equiv 1 \pmod{4}$ y $(\frac{-1}{n}) = -1$ si $n \equiv 3 \pmod{4}$.
7. $(\frac{2}{n}) = (-1)^{(n^2-1)/8}$. De aquí $(\frac{2}{n}) = 1$ si $n \equiv 1 \text{ ó } 7 \pmod{8}$ y $(\frac{2}{n}) = -1$ si $n \equiv 3 \text{ ó } 5 \pmod{8}$.
8. $(\frac{m}{n}) = (\frac{n}{m})(-1)^{(m-1)(n-1)/4}$. Es decir, $(\frac{m}{n}) = (\frac{n}{m})$ a no ser que tanto m como n sean congruentes con 3 módulo 4, en el cual caso $(\frac{m}{n}) = -(\frac{n}{m})$.

Observación 1.41 De las propiedades del símbolo de Jacobi, se sigue que si n es impar y $a = 2^e a_1$, con a_1 impar, entonces

$$\begin{aligned} \left(\frac{a}{n}\right) &= \left(\frac{2^e}{n}\right) \left(\frac{a_1}{n}\right) \\ &= \left(\frac{2}{n}\right)^e \left(\frac{a_1}{n}\right) \\ &= \left(\frac{2}{n}\right)^e \left(\frac{n \pmod{a_1}}{a_1}\right) (-1)^{(a_1-1)(n-1)/4}. \end{aligned}$$

Esta observación nos lleva a un algoritmo recursivo que permite calcular el símbolo de Jacobi $(\frac{a}{n})$ sin que sea necesario conocer de antemano la factorización de n .

Observación 1.42 El símbolo de Jacobi no dice nada respecto a la pertenencia o no de a al conjunto Q_n de restos cuadráticos módulo n . En otras palabras, si $a \in Q_n$, entonces $(\frac{a}{n}) = 1$; en cambio la recíproca no es cierta. Por ejemplo, si consideramos $5 \in \mathbb{Z}_{21}^*$, es fácil ver que $(\frac{5}{21}) = 1$ y sin embargo $5 \notin Q_{21}$.

Este hecho da lugar a la siguiente

Definición 1.43 Sea $n \geq 3$ un entero impar. Sea $J_n = \{a \in \mathbb{Z}_n^* : (\frac{a}{n}) = 1\}$. Definimos entonces el conjunto de los pseudorrestos cuadráticos módulo n , y lo denotamos \tilde{Q}_n , al conjunto de todos los elementos de J_n que no están en Q_n , es decir, $\tilde{Q}_n = J_n - Q_n$.

Cálculo de los símbolos de Jacobi y de Legendre

Pasamos así a describir un algoritmo que permite computar el símbolo de Jacobi y por tanto también el de Legendre.

Algoritmo 1.44 Este algoritmo recibe como entrada un entero impar $n \geq 3$ y otro entero a , tal que $0 \leq a < n$ y calcula el símbolo de Jacobi $(\frac{a}{n})$.

LLAMADA: $\text{Jacobi}(a, n)$;

ENTRADA: Un entero impar n y otro entero a , tal que $0 \leq a < n$.

SALIDA: Símbolo de Jacobi $(\frac{a}{n})$.

1. [Inicialización]

```
if (a == 0) return 0;
if (a == 1) return 1;
Escribimos  $a = 2^e a_1$ , donde  $a_1$  es impar
```

2. [Cuerpo]

```
if (e es par)
    s = 1;
else
{
    if ( $n \equiv 1 \pmod{8}$ ) o bien  $n \equiv 7 \pmod{8}$ )
        s = 1;
    else
        s = -1;
}
if ( $n \equiv 3 \pmod{4}$ ) y  $a_1 \equiv 3 \pmod{4}$ )
    s = -s;
n_1 = n(mod a_1);
if (a_1 == 1)
    return s;
else /* El algoritmo se invoca a sí mismo */
    return s.Jacobi(n_1, a_1);
```

Observemos que si n es un número primo, habremos calculado el símbolo de Legendre.

1.2 Nociones de complejidad computacional

Como iremos viendo a lo largo de la memoria, muchos criptosistemas están basados en la existencia de un “problema” matemático cuya solución se considera en la actualidad inabordable, teórica o prácticamente. Precisamente quien fuera capaz de resolver ese “problema”, habría descifrado definitivamente el sistema: sería como hallarse en posesión de una llave maestra que permite abrir cualquier cerradura, es decir, descifrar cualquier mensaje cifrado con cualquier clave.

Es claro que resultaría interesante poder cuantificar de alguna manera el grado de dificultad de cada “problema”, introduciendo alguna métrica que permita compararlos. En este contexto se inserta la *teoría de complejidad computacional*, de la que vamos a dar ahora una breve introducción que será de interés para lo sucesivo.

1.2.1 Algoritmos

El principal objetivo de la teoría de complejidad computacional es proporcionar mecanismos que permitan clasificar los problemas computacionales de acuerdo con los recursos que se necesitan para resolverlos. Idealmente, esta clasificación no debe depender de un modelo computacional particular sino que debe reflejar la dificultad intrínseca del problema. Los recursos computacionales suelen ser tiempo de proceso y necesidades de almacenamiento en memoria. El elemento básico en este contexto es el *algoritmo*.

Definición 1.45 Se denomina *algoritmo* un procedimiento computacional que toma una entrada y produce una salida.

Aunque esta definición no es muy precisa, sí da la idea intuitiva de que un algoritmo no es más que un programa escrito para una máquina determinada en un cierto lenguaje de programación y que, alimentado con una entrada, produce una salida, utilizando recursos computacionales de tiempo de ejecución y memoria.

Una distinción interesante es la siguiente:

Definición 1.46 Se denomina *algoritmo determinista* al que sigue la misma secuencia de operaciones cada vez que se ejecuta con la misma entrada.

Definición 1.47 Se denomina *algoritmo probabilístico* al que realiza ciertas decisiones aleatorias en el curso de su ejecución, por lo que no sigue necesariamente la misma secuencia de operaciones aunque se le proporcione sucesivamente la misma entrada.

1.2.2 Tamaño de un entero

Generalmente interesa encontrar el algoritmo más eficiente para resolver un problema computacional determinado. Típicamente, el tiempo depende del “tamaño” de las entradas y del número de operaciones que hayan de realizarse sobre ellas. Para dar una noción más precisa del tamaño de un entero, recordemos que

cualquier entero n puede ser representado mediante la expansión b -ádica de la siguiente manera

$$n = \sum_{i=1}^k a_i b^{k-i},$$

con $a_1 \neq 0$, $0 \leq a_i < b$ y $b > 1$. Además

$$k = \lfloor \log_b n \rfloor + 1.$$

Decimos que n es un k -dígito en la base b . Cuando la base $b = 2$, hablamos de expresión binaria, cada a_i es un bit y n es un k -bit. Con ello, tenemos la siguiente

Definición 1.48 El *tamaño* de un entero es el número total de bits necesario para representarlo en expresión binaria.

A continuación nos planteamos el problema del tiempo de ejecución. Para ello comenzamos evaluando el coste computacional de las operaciones elementales, que son básicamente la suma y el producto.

1.2.3 Tiempo de ejecución de la suma y el producto

Sean a y b dos números enteros, que suponemos representados en forma binaria. Para sumarlos, bastará colocarlos uno encima de otro y sumar bit a bit, junto con el acarreo, tal como se aprende en la escuela. Veámoslo con un

Ejemplo 1.49 Sea $a = 110101$ y $b = 1101$. Colocamos un número encima de otro, rellenando con ceros por la izquierda el entero con tamaño más pequeño. Realizamos la suma de acuerdo a la siguiente tabla

acarreo	1	1	1	1	0	1	
	0	1	1	0	1	0	1
+	0	0	0	1	1	0	1
	1	0	0	0	0	1	0

Cada columna representa una *operación bit*. Ésta consiste en sumar un bit de a , un bit de b y el acarreo de la operación anterior y colocar el resultado y el acarreo en la filas correspondientes de la siguiente columna. Por lo tanto, si m es el tamaño de a , y n es el tamaño de b , para realizar la suma serán necesarias $\max(m, n)$ operaciones bit. Análogamente se procede para la resta, que necesita el mismo número de operaciones bit.

La operación de multiplicar se reduce como es natural a realizar una suma repetida. Pero veamos en el siguiente ejemplo el método de la escuela.

Ejemplo 1.50 Sean ahora $a = 110101$ y $b = 1001$, como antes. Calculamos el producto de la siguiente forma

	1	1	0	1	0	1	.	1	0	0	1	
acarreo		0	0	1	0	0	0	0	0	0	0	1
								1	1	0	1	0
+		1	1	0	1	0	1					
	1	1	1	0	1	1	1	0	1			

Lo que hacemos es repetir el valor de a tantas veces como *unos* contenga la representación binaria de b , colocado de tal forma que esté desplazado a la izquierda tantas columnas como sean necesarias para que el bit menos significativo coincida con la columna del *uno* de b .

Si m es el tamaño de a y n es el tamaño de b , la multiplicación consistirá en sumar como máximo n veces un número que tiene un tamaño de, como máximo, $m + n$ bits y, por lo tanto, la multiplicación llevará $(m + n)n$ operaciones bit.

1.2.4 Tiempo de ejecución de un algoritmo

Definición 1.51 El *tiempo de ejecución* de un algoritmo es el número de operaciones bit o “pasos” que ejecuta hasta conseguir producir la salida.

Corrientemente, el tiempo de ejecución de un algoritmo depende del tamaño de la entrada.

Definición 1.52 El *tiempo de ejecución promedio* de un algoritmo es el tiempo de ejecución promediado sobre todas las posibles entradas de un tamaño fijo, expresado como función del tamaño de la entrada.

En el caso de los algoritmo probabilísticos, hablaremos del *tiempo de ejecución esperado*, que definimos así:

Definición 1.53 El tiempo de ejecución esperado para un algoritmo probabilístico es una cota superior del tiempo de ejecución esperado para cada entrada (medida sobre las diversas salidas que el proceso aleatorio intrínseco al algoritmo genera), expresado como función del tamaño de la entrada.

Puesto que es difícil normalmente saber con exactitud el tiempo de ejecución de un algoritmo, se hace necesario contentarse con aproximaciones. La más usada es la asintótica, es decir, se estudia cómo aumenta el tiempo de ejecución al aumentar indefinidamente el tamaño de la entrada. Introducimos para ello la *notación asintótica*, que usaremos con frecuencia en lo sucesivo.

Definición 1.54 Si f y g son dos funciones de variable discreta con términos no negativos, entonces

$$f(n) = O(g(n))$$

indica que existe una constante $c > 0$ y un entero n_0 tales que

$$f(n) \leq cg(n), \quad \forall n \geq n_0.$$

Definición 1.55 Sean f y g igual que en la Definición anterior. Entonces

$$f(n) = \Omega(g(n))$$

indica que existe una constante $c > 0$ y un entero n_0 tales que

$$0 \leq cg(n) \leq f(n), \quad \forall n \geq n_0.$$

Definición 1.56 Sean f y g igual que en la Definición anterior. Entonces

$$f(n) = \Theta(g(n))$$

indica que existen constantes $c_1, c_2 > 0$ y un entero n_0 tales que

$$c_1 g(n) \leq f(n) \leq c_2 g(n), \quad \forall n \geq n_0.$$

Definición 1.57 Sean f y g igual que en la Definición anterior. Entonces

$$f(n) = o(g(n))$$

indica que para cualquier $c > 0$, existe un entero n_0 tal que

$$0 \leq f(n) < cg(n), \quad \forall n \geq n_0.$$

Intuitivamente $f(x) = O(g(x))$ significa que f se mantiene asintóticamente igual a g , salvo una constante, al aumentar x . En cambio, $f(x) = o(g(x))$ significa que f se hace infinitamente pequeña frente a g conforme x aumenta. Por ejemplo, $f(x) = O(1)$ indica que f se hace constante cuando x tiene a infinito; en cambio $f(x) = o(1)$ significa que el límite de f es 0 cuando x tiende a infinito.

Ejemplo 1.58 Si $f(x)$ es un polinomio de grado n , con el coeficiente de mayor grado positivo, entonces $f(x) = \Theta(x^n)$.

Ejemplo 1.59 Para toda constante $c > 0$, $\log_c x = \Theta(\log_2 x)$.

Con estas definiciones, podemos precisar lo siguiente

1. La suma de dos enteros a y b requiere $O(\log_2(\max(a, b)))$ operaciones bit.
2. El producto de dos enteros a y b por el método “escolar” requiere $O(\log_2 a \cdot \log_2 b)$ operaciones bit. Existen otros algoritmos que reducen este tiempo hasta $O(k \cdot \ln k \cdot \ln \ln k)$, donde $k = \max(\log_2 a, \log_2 b)$.

1.2.5 Clasificación de los algoritmos

Se tienen las siguientes definiciones:

Definición 1.60 Un *algoritmo de tiempo polinómico* es aquel cuyo tiempo de ejecución para el caso peor es de la forma $O((\log_2 n)^k)$, donde n es la entrada y k una constante.

Definición 1.61 Un *algoritmo de tiempo exponencial* es aquel cuyo tiempo de ejecución para el caso peor no se puede acotar polinómicamente.

En la práctica, los únicos algoritmos realmente útiles son los que exhiben tiempo de ejecución polinómico, pues tardan un tiempo “razonable” en producir sus resultados, aunque las entradas sean de tamaños “grandes”. Por ello, se dice genéricamente que los algoritmos de este tipo son eficientes. Los de tiempo exponencial

se consideran, sin embargo, ineficientes. Existe toda una clasificación de los problemas matemáticos en función de la clase de algoritmos que se conozcan para resolverlos. Una explicación detallada puede encontrarse en [72, sección 2.3.3].

A lo largo de este trabajo, entenderemos que un problema es “computacionalmente inabordable” cuando el algoritmo que lo resuelve no es de tiempo polinómico.

Existe un “caso intermedio” que viene explicado mediante la siguiente

Definición 1.62 Un *algoritmo de tiempo subexponencial* es aquel cuyo tiempo de ejecución para el caso peor es de la forma $\exp(o(\log_2 n))$, donde n es la entrada.

Dicho con palabras, un algoritmo de tiempo subexponencial es asintóticamente más rápido que uno exponencial, pero más lento que uno de tiempo polinómico.

Ejemplo 1.63 Sea A un algoritmo cuya entrada es un entero q , o elementos de un cuerpo finito \mathbb{F}_q . Si el tiempo de ejecución esperado es de la forma

$$L_q[\alpha, c] = O\left(\exp((c + o(1))(\ln q)^\alpha (\ln \ln q)^{1-\alpha})\right),$$

donde c es una constante positiva y $0 < \alpha < 1$ es otra constante, entonces es claro que A es un algoritmo de tiempo subexponencial. Este ejemplo es relevante porque aparece de hecho en varios de los algoritmos que se usan en la práctica.

Ejemplo 1.64 El algoritmo de Gauss (ver Algoritmo 1.22) para resolver el sistema de congruencias del teorema chino del resto se puede ejecutar en $O((\log_2 n)^2)$ operaciones bit, donde n es el módulo de la congruencia. De acuerdo con la Definición 1.60 se trata de un algoritmo de tiempo polinómico.

Ejemplo 1.65 El algoritmo de Euclides (ver Algoritmo 1.9) necesita ejecutar $O(\log_2 a \cdot \log_2 b)$ operaciones, es decir, es un algoritmo de tiempo polinómico y, por tanto, eficiente.

1.3 Herramientas de computación

Cuando se plantea el objetivo de implementar en un computador los diferentes algoritmos que aparecerán a lo largo de la presente memoria surge inmediatamente la necesidad de disponer de herramientas computacionales que manejen la *aritmética de multiprecisión*.

En efecto, en los momentos actuales, los computadores más asequibles son capaces de manejar números enteros en el rango

$$[-2^{31}, 2^{31} - 1] = [-2147483648, 2147483647],$$

puesto que sus procesadores disponen de 32 bits: es el caso de los procesadores de la familia Pentium, fabricados por Intel, y sus clónicos; en este tipo de procesadores se basan las plataformas PC actuales. Están bastante avanzados los procesadores que permitirán el manejo de 64 bits, lo cual posibilitará trabajar con enteros en el rango

$$[-2^{63}, 2^{63} - 1] = [-9223372036854775808, 9223372036854775807];$$

es el caso de la arquitectura IA64 que desarrollan conjuntamente HP e Intel, cuyo primer retoño es el procesador Itanium. Sin embargo, los sistemas basados en este procesador no son todavía asequibles popularmente.

Con ser estos números muy grandes, son sin embargo insuficientes cuando se trata de trabajar, por ejemplo, en el mundo de la Criptografía, que maneja enteros de cientos o miles de cifras decimales: se necesita poder usar números de precisión arbitraria y este es, justamente, el problema que la *aritmética de multiprecisión* trata de resolver.

A lo largo del presente trabajo, hemos recurrido básicamente a dos tipos de herramientas, cada una de ellas muy valiosa en su ámbito: las *bibliotecas de programación* y las *aplicaciones de computación simbólica*. Esta distinción es un tanto artificial, pues, con frecuencia, éstas últimas ofrecen también aquéllas como parte del paquete.

1.3.1 Bibliotecas de programación con multiprecisión

Una biblioteca de programación consiste en un conjunto de subrutinas o funciones agrupadas, que pueden ser explotadas mediante programas escritos en un lenguaje compilable, como C, o incluso directamente en el ensamblador del procesador correspondiente.

Generalmente estas bibliotecas ofrecen unos tipos de datos de tamaño arbitrario y unas funciones para manejarlos: las operaciones elementales de suma, resta, multiplicación y división, comparaciones, entrada/salida y otras de más alto nivel, como cálculo de máximo común divisor, o exponentiaciones modulares.

Existe un número de bibliotecas de programación de este tipo, muchas de ellas amparadas por una licencia de uso de tipo GNU, lo que se traduce en que cualquiera puede usarla gratuitamente. Algunos ejemplos son los siguientes:

1. Biblioteca GMP.
2. Biblioteca LIP de Arjen Lenstra.
3. Biblioteca Pari.

En esta memoria hemos elegido utilizar la biblioteca GMP, de la que ofrecemos una descripción más detallada.

Biblioteca GMP

GMP es una biblioteca para trabajar en aritmética de multiprecisión que admite como tipos de datos los enteros con o sin signo, los racionales y los reales de coma flotante y está cubierta por una licencia de tipo GNU. Para ver la documentación completa y detalles, consúltese la referencia [48].

La biblioteca es en sí misma un conjunto de programas escritos la mayoría de ellos en lenguaje C y un pequeño subconjunto escrito en ensamblador de diversos procesadores. La idea es tratar de optimizar la velocidad de ejecución, manteniendo al mismo tiempo la sencillez y la transportabilidad del desarrollo, es decir, que la biblioteca sea utilizable sin modificaciones en el mayor número de plataformas

possible. No tiene más límite a la precisión que la disponibilidad de memoria de la máquina en que se trabaje. En la explicación que sigue, supondremos utilizado el lenguaje de programación C.

Para hacer uso de la biblioteca, se ha de comenzar incluyendo la línea siguiente:

```
# include "gmp.h"
```

Con ella, se tiene acceso a todos los tipos de datos que la biblioteca soporta. Existen varias clases de funciones, para cada uno de los tipos de datos. Se sigue el convenio de que cada función lleva como prefijo el tipo de dato a que se refiere.

No damos detalles acerca de la compilación, que depende de la plataforma y compilador utilizados; como ejemplo, si nuestro programa está almacenado en el fichero `uno.c`, una forma típica en entornos unix, linux, (o similares como Cygwin) podría ser:

```
gcc uno.c -I/usr/local/include -L/usr/local/lib -lgmp -o uno
```

Las funciones para la aritmética de enteros con signo comienzan con el prefijo `mpz_` y el tipo de dato asociado es `mpz_t` de las que existen unas 150 en la biblioteca. Casi todas siguen el convenio de que el primer argumento es de salida y los siguientes de entrada, en analogía con el operador de asignación. Puede usarse la misma variable para entrada y para salida. Antes de usar una variable ha de ser “inicializada” utilizando la función `mpz_init`: esta función reserva espacio en la memoria. Cuando ya no sea necesaria, ese espacio puede —a veces debe— liberarse con la función `mpz_clear`.

La siguiente función utiliza internamente una variable `n` que inicializa al principio, utiliza y, finalmente, libera:

```
void no_hace_nada(void)
{
    mpz_t n;
    int i;

    mpz_init(n);

    for (i = 1; i < 100; i++)
    {
        mpz_mul(n, ...);
        mpz_fdiv_q(n, ...);
        ...
    }

    mpz_clear(n);

}
```

Los parámetros de las funciones se pasan siempre por referencia. Esto significa que si la función almacena en ellos algún valor, esto modifica el valor original en la función llamadora. Veamos otro ejemplo, más completo, de una función que

acepta un parámetro de tipo `mpz_t`, hace un cálculo y devuelve el resultado en otra variable distinta también de tipo `mpz_t`.

```
void calc(mpz_t result, const mpz_t param, unsigned long n)
{
    unsigned long i;

    mpz_mul_ui(result, param, n);

    for (i = 1; i < n; i++)
        mpz_add_ui(result, result, i*7);
}

int main(void)
{
    mpz_t r, n;

    mpz_init(r);
    mpz_init_set_str(n, "123456", 0);
    calc(r, n, 20L);
    mpz_out_str(stdout, 10, r);
    return 0;
}
```

En este ejemplo hemos hecho uso de varias funciones que pasamos a describir:

```
void mpz_init(mpz_t n)
Inicializa la variable n, y coloca en ella el valor 0.

void mpz_init_set_str(mpz_t n, char *str, int base)
Inicializa la variable n, y coloca en ella el valor representado por la cadena str, expresada en la base base. Si ésta es 0, se sobrentiende base 10.

void mpz_add(mpz_t r, mpz_t s1, mpz_t s2)
void mpz_add_ui(mpz_t r, mpz_t s1, unsigned long int s2)
Calcula  $r = s1 + s2$ .

void mpz_mul(mpz_t r, mpz_t s1, mpz_t s2)
void mpz_mul_ui(mpz_t r, mpz_t s1, unsigned long int s2)
Calcula  $r = s1 \cdot s2$ .

void mpz_fdiv_qr(mpz_t q, mpz_t r, mpz_t n, mpz_t d)
Realiza la división entera  $n/d$  y coloca el cociente en q y el resto en r.

size_t mpz_out_str(FILE *stream, int base, mpz_t r)
Escribe en el canal de salida stream el valor de la variable r expresado en la base base. Devuelve el número de bytes escritos o un 0 si hubo algún error.
```

Existen otras muchas que pueden ser consultadas en la documentación de la biblioteca (véase [47]). Resultan de interés para los cálculos típicos en Criptografía la exponentiación modular, el cómputo del máximo común divisor, del mínimo

común múltiplo o de los símbolos de Legendre y Jacobi. En el Apéndice I se pueden ver ejemplos reales de uso de estas funciones.

Biblioteca LIP

Esta biblioteca fue escrita inicialmente por Arjen Lenstra y está a la libre disposición de las instituciones con fines educativos o de investigación. Es análoga a la biblioteca GMP aunque sólo proporciona la aritmética de números enteros con signo de precisión arbitraria. El tipo de dato básico se denomina *verylong* y puede albergar números arbitrariamente grandes, sin más límite que la memoria de la máquina.

Biblioteca Pari

Está integrada junto al paquete de computación simbólica Pari/GP que describimos en la sección siguiente.

1.3.2 Aplicaciones de computación simbólica

Las aplicaciones de computación simbólica están destinadas al usuario final. Ofrecen una interfaz interactiva que permite realizar operaciones simbólicas, como puede ser cálculo, álgebra, matemática discreta, gráficos, cálculo numérico, etc. Además suelen ofrecer un lenguaje tipo “script” interno que permite una cierta programación de tareas a realizar y un enlace con programas externos.

También existen varios paquetes de este estilo en el mercado, aunque los más conocidos son MAPLE¹ y MATHEMATICA². Como ejemplo de aplicación libre, no comercial, tenemos el paquete Pari/GP.

Este tipo de aplicaciones tienen una gran utilidad práctica, pues eliminan la necesidad de realizar manualmente cálculos laboriosos en los que la probabilidad de equivocarse es, francamente, muy alta.

Aplicación MAPLE

Maple es un entorno completo para resolver problemas matemáticos que soporta una variedad de operaciones: análisis numérico, álgebra simbólica, gráficos, etc. Este es el paquete que hemos utilizado en esta memoria; algunos de los programas que se incluyen en el apéndice están escritos en el lenguaje propio de MAPLE.

La aplicación abre una “ventana de comandos”, en donde se teclean las órdenes a las que el sistema responde. Un ejemplo de comando interactivo y su respuesta podría ser el siguiente:

```
> int( sin(x), x );
                                         - cos(x)
```

¹Véase <http://www.maplesoft.com>

²Véase <http://www.wolfram.com>

La orden solicita la integral indefinida de la función $\sin(x)$ y el sistema responde con el conocido $-\cos(x)$.

```
> diff( tan(x), x );
```

$$1 + (\tan(x))^2$$

En este ejemplo, ha calculado la derivada $\frac{d}{dx}(\tan x) = 1 + (\tan x)^2$. El paquete admite también programación con un lenguaje propio: en el Apéndice I pueden verse algunos ejemplos reales.

Aplicación MATHEMATICA

Se trata de un paquete del todo análogo a Maple, con muy similares capacidades. También presenta un entorno interactivo para trabajar y posibilidad de programación con su lenguaje propio. No se ha utilizado en esta memoria.

Aplicación Pari/GP

Pari/GP es un sistema de computación algebraica diseñado para realizar cálculos rápidos especialmente en el ámbito de la teoría de números: factorizaciones, curvas elípticas, etc. También se puede trabajar con otras entidades, como matrices, polinomios, series de potencias, números algebraicos o funciones trascendentes. Esta aplicación también dispone de una biblioteca de funciones que pueden llamarse desde un lenguaje de programación, típicamente C. Pueden consultarse su documentación y descargarse la aplicación desde la dirección <http://www.parigp-home.de/>.

Originalmente fue desarrollado en la Universidad de Burdeos (1985-1996) por Henri Cohen y su equipo. Actualmente lo mantiene Karim Belabas (Universidad de Paris XI/Orsay) con la ayuda de un conjunto de voluntarios.

Capítulo 2

Criptosistemas de clave pública y primos especiales

Resumen del capítulo

Se describe el formalismo de los criptosistemas de clave pública y su desarrollo histórico. Se explica el protocolo de la firma digital y el criptoanálisis de este tipo de sistemas. Se detallan en particular el sistema RSA y sus variantes, el sistema de ElGamal y los criptosistemas probabilísticos de Blum-Goldwasser y Goldwasser-Micali. Para cada uno se destaca la importancia de usar primos especiales. Se ofrece finalmente una comparación entre los sistemas de clave pública y los sistemas de clave secreta.

2.1 Orígenes de los criptosistemas de clave pública

2.1.1 Introducción

A mediados de los años 70 la Criptografía tradicional experimentó una revolución profunda con la aparición de los esquemas de clave pública. El desarrollo y proliferación de los equipos de electrónica digital muy económicos liberó a los procesos criptográficos de las antiguas limitaciones de la computación mecánica y permitió un uso generalizado de las técnicas de protección de datos. A su vez, esta generalización creó la necesidad de nuevos tipos de sistemas criptográficos que resolvieran los problemas de la distribución de las claves de una forma segura y proporcionaran un equivalente digital a la firma de un mensaje.

El desarrollo de las comunicaciones está propiciando un contacto fácil y económico entre personas y entidades que tiende a sustituir los mecanismos tradicionales de correo escrito, a la vez que induce su utilización para la transmisión de todo género de datos, también los que tienen un alto valor por ser confidenciales u otras causas. Resulta, por lo tanto, natural pensar que ese desarrollo debe ir acompañado de las medidas de seguridad necesarias para evitar las interferencias ilegítimas de terceros no autorizados: escuchas ilegales, envío de mensajes falsos, suplantación de la personalidad son ejemplos de los peligros que potencialmente acechan cualquier transmisión que pretenda ser segura.

Hemos asistido además en estos últimos años al imparable desarrollo de Internet, que ha colocado literalmente en la punta de los dedos todo un universo de información, entretenimiento y servicios. Es fácil poner ejemplos de algunos servicios en los que la seguridad de la transmisión es una pieza fundamental para que el servicio pueda ser prestado y utilizado con confianza tanto por quien lo explota como por los clientes.

Todo ello explica la expansión que se ha producido en el ámbito de las técnicas de protección de datos y su transmisión a través de un canal inseguro.

Tradicionalmente la protección y transmisión segura de datos se ha encomendado a los *criptosistemas de clave secreta*. En un criptosistema de clave secreta el remitente y el destinatario comparten en secreto una clave para cifrar y una clave para descifrar los mensajes. Tales sistemas se han usado desde la más remota antigüedad hasta nuestros días. Un estudio histórico puede encontrarse en [55].

Entre los inconvenientes más comúnmente citados de los criptosistemas de clave secreta se suelen señalar los siguientes:

1. *Distribución de claves.* Dos usuarios tienen que seleccionar una clave en secreto y comunicársela mutuamente, o bien desplazándose a un lugar común para tal efecto, o bien enviándola por medio de un canal inseguro.
2. *Almacenamiento de claves.* En una red de n usuarios el número de claves necesarias es $\frac{1}{2}n(n - 1)$, cantidad que crece cuadráticamente con el número de usuarios.
3. *Carencia de firma digital.* El destinatario no puede estar seguro de que quien le envía el mensaje sea realmente el remitente; esto es, en los criptosistemas de clave secreta no hay posibilidad de firmar digitalmente los mensajes. Una firma digital es lo análogo a una firma manual o rúbrica en una red de comunicaciones. Realmente, en los criptosistemas de clave secreta no es necesaria la firma digital si se restringe el conocimiento de la clave a los dos usuarios. El problema surge cuando uno de ellos tiene que convencer a un tercero (por ejemplo, al juez) —fehacientemente y sin que sea necesario revelarle la clave secreta— de que un mensaje es auténtico, es decir, ha sido verdaderamente escrito y firmado por quien dice ser su autor.

La criptografía de clave pública nació básicamente para resolver estos inconvenientes. En un criptosistema de clave pública cada usuario elige y maneja, en realidad, dos claves, íntimamente relacionadas: una es la *clave pública* que el usuario pone a disposición del resto de usuarios del sistema; otra es la *clave privada*, sólo conocida por él. Supongamos que un usuario *A* quiere enviarle al usuario *B* un mensaje cifrado. Los pasos serían los siguientes:

1. *A* selecciona en el directorio de claves públicas la clave correspondiente a *B*.
2. *A* cifra su mensaje aplicando la clave pública de *B* y se lo envía.

Los pasos del usuario *B* serían:

1. *B* recibe el mensaje cifrado.

2. B descifra el mensaje aplicando su clave privada, sólo conocida por él.

El sistema basa su seguridad en la dificultad que representa para cualquier usuario distinto de B —desprovisto, por lo tanto, de la clave privada— descifrar el mensaje. Cuanto mayor sea esa dificultad, más seguro podemos considerar el sistema, más difícil será el trabajo del *criptoanalista*. Esta figura, siempre presente en el mundo de la Criptografía, representa el adversario que trata de encontrar los puntos débiles del sistema para adquirir la información a la que no tenía acceso. El conjunto de sus técnicas se conocen con el nombre de Criptoanálisis.

Aunque los criptosistemas de clave pública resuelven —como veremos— los inconvenientes que presentan los sistemas de clave secreta, plantean, sin embargo, otros problemas. Uno muy inmediato está relacionado con la *autenticación* de la clave pública. En efecto, el remitente debe estar seguro de que está cifrando con la clave pública “auténtica” del destinatario, pues esa clave ha sido quizás publicada u obtenida a través de un canal inseguro, lo que permite el llamado “ataque del criptoanalista activo” (véase la sección 2.3.1 para una exposición más detallada). Un criptoanalista activo es aquel que no sólo escucha en la red sino que influye en ella enviando y falsificando la información que puede interferir. Un tal criptoanalista C puede proceder como sigue:

1. A quiere enviar a B un mensaje, para lo cual solicita al directorio la clave pública de B .
2. El criptoanalista C interfiere la comunicación y envía maliciosamente a A una clave pública errónea e' de B .
3. A envía un mensaje a B usando e' .
4. C lo intercepta y lo descifra usando su propia clave privada d' .
5. C lo cifra de nuevo con la clave pública e de B y lo envía a B .

Este problema se resuelve en la práctica con un protocolo de certificado de clave pública que se basa en la capacidad de firma disponible en los criptosistemas de clave pública y en la autoridad de un tercero de confianza¹. Pueden verse más detalles, por ejemplo, en [72, sección 1.52].

2.1.2 El cambio de clave de Diffie-Hellman

La criptografía de clave pública nació con el trabajo de Diffie-Hellman ([30]) en el que los autores describen un método para compartir información sin necesidad de una clave secreta común y por medio de un canal inseguro. Este protocolo no puede considerarse un criptosistema de clave pública propiamente dicho, pues no permite cifrar mensajes arbitrarios. Sin embargo representa el primer logro sustantivo en el camino de la criptografía de clave pública. Pasemos a describir el protocolo.

Para que los usuarios A y B puedan comunicarse y compartir una información, deben dar los siguientes pasos:

¹Se suele denominar TTP, del inglés “Trusted Third Party”.

1. A y B seleccionan públicamente un grupo cíclico G de orden n , y un generador $\alpha \in G$.
2. A elige en secreto un entero $a \in \mathbb{Z}$, y transmite a B el elemento α^a .
3. B elige en secreto un entero $b \in \mathbb{Z}$, y transmite a A el elemento α^b .
4. A recibe α^b y calcula $(\alpha^b)^a$.
5. B recibe α^a y calcula $(\alpha^a)^b$.
6. A y B comparten un secreto en común: el elemento α^{ab} , sin necesidad de desplazarse ni de enviarlo por la red.

Obsérvese que los usuarios intercambian una información que puede usarse como clave, pero no intercambian propiamente un mensaje, por lo que el protocolo no llega a constituir un sistema criptográfico.

Para un grupo cíclico arbitrario, este protocolo recibe el nombre de *cambio de clave de Diffie-Hellman generalizado*. En realidad, en su artículo original, Diffie y Hellman usaron el grupo multiplicativo de los enteros módulo un número primo p ; esto es $G = \mathbb{Z}_p^*$, y, en este caso, el protocolo se llama simplemente *cambio de clave de Diffie-Hellman*.

2.1.3 Criptoanálisis del cambio de clave de Diffie-Hellman

El criptoanalista conoce $G, \alpha, \alpha^a, \alpha^b$, pero para atacar el cambio de clave de Diffie-Hellman debe determinar α^{ab} . Una forma de hacerlo es resolver alguna de las ecuaciones

$$\log_\alpha(\alpha^a) = a, \quad \log_\alpha(\alpha^b) = b. \quad (2.1)$$

Por este motivo, se dice que la seguridad del cambio de clave de Diffie-Hellman está basada en la dificultad de computación del logaritmo discreto. El *logaritmo discreto* es la noción análoga a la de los números reales pero para un grupo finito. Por definición, si α es un generador de un grupo cíclico finito G de orden n , la aplicación

$$\begin{aligned} \mathbb{Z}_n &\rightarrow G \\ k &\mapsto \alpha^k \end{aligned}$$

es un isomorfismo, donde $\mathbb{Z}_n = \{0, 1, \dots, n-1\}$ es el grupo aditivo de los números enteros módulo n . Pues bien, se llama logaritmo discreto en base α a la aplicación inversa $\log_\alpha: G \rightarrow \mathbb{Z}_n$.

El *Problema del Logaritmo Discreto Generalizado* (GDLP) consiste en calcular a conociendo G, α y α^a . El adjetivo “generalizado” se aplica porque el problema original fue formulado en el grupo multiplicativo de los enteros módulo un número primo p , esto es, cuando $G = \mathbb{Z}_p^* = \{1, 2, \dots, p-1\}$. El *Problema de Diffie-Hellman Generalizado* GDHP consiste en calcular α^{ab} conociendo G, α, α^a y α^b .

Está claro que la solución del GDLP implica la solución del GDHP, pero no se sabe si, en general, el recíproco es cierto. En principio, podría caber la posibilidad de que un criptoanalista calculara α^{ab} por otro método que no fuera el de resolver las ecuaciones logarítmicas (2.1).

Antes de explicar los primeros resultados acerca de estos problemas recordemos la siguiente

Definición 2.1 Si $B > 0$ es una constante, se dice que un entero es B -uniforme² o, también, uniforme con cota B , si ninguno de los factores primos de su descomposición excede el valor B .

El primer resultado publicado acerca de la equivalencia de los problemas GDLP y GDHP se debe a B. den Boer. Éste probó en [15] que la equivalencia se daba para el grupo multiplicativo \mathbb{Z}_p^* , siendo p un primo tal que el indicador de Euler $\varphi(\varphi(p)) = \varphi(p - 1)$ es B -uniforme, donde $B = Q(\log_2 p)$, siendo Q un polinomio fijo. Es de notar que este resultado no dice nada acerca de la existencia real de algún algoritmo que compute eficazmente el logaritmo discreto en \mathbb{Z}_p^* cuando p es de la forma antes descrita; más bien que, justamente mientras ese algoritmo no se conozca, el cambio de clave de Diffie-Hellman será seguro.

En un trabajo posterior, U. Maurer (véase [68]) extendió el resultado de [15] a grupos más generales, la factorización de cuyo orden contiene tan solo “factores primos pequeños” en un sentido que el autor define con precisión, pero que no es necesario detallar aquí. Los últimos resultados sobre el problema de equivalencia se encuentran en [70, 71].

Desde un punto de vista más práctico, el protocolo de cambio de clave de Diffie-Hellman es vulnerable al llamado *ataque del intermediario*. Este ataque forma parte del conjunto de ataques denominados genéricamente *ataques activos*, cuya definición presentamos en la sección 2.3.2. En efecto, el protocolo no garantiza la autenticidad de la clave pública del receptor del mensaje; por lo tanto, el emisor no tiene la seguridad de estar intercambiando la clave con quien cree estar haciéndolo. Si un adversario tiene la capacidad de intervenir la comunicación de las partes, puede recibir un mensaje de A y, una vez alterado, reenviarlo a B y viceversa. Para frustrar este ataque, los usuarios pueden intercambiar exponentiaciones autenticadas, es decir, firmadas digitalmente, asegurándose así que proceden del legítimo emisor.

2.1.4 Funciones unidireccionales

El cambio de clave de Diffie-Hellman se apoya, como tratamos en las secciones 2.1.2 y 2.1.3, en el problema del cómputo del logaritmo discreto. Iremos viendo cómo otros criptosistemas de clave pública también se apoyan en algún “problema”, para el que no se conoce ninguna solución que no sea muy difícil y hasta inabordable.

Para poder formular este hecho desde un punto de vista matemático, es conveniente introducir un tipo de funciones que reciben el nombre de *funciones unidireccionales*³, cuya definición es la siguiente:

Definición 2.2 Una función invertible $f(x)$ recibe el nombre de función unidireccional si es “fácil” calcular $y = f(x)$ para todo x de su dominio, pero es

²En inglés, “ B -smooth”.

³En inglés, “one-way functions”.

computacionalmente inabordable calcular $f^{-1}(y)$ para “casi” todos los elementos de la imagen de f .

Tales funciones, pues, presentan la propiedad de que ellas y sus inversas son completamente asimétricas en cuanto a tiempos de computación se refiere. Un ejemplo sencillo de función con esta propiedad es $f(x) = x^3$, pues, obviamente, es mucho más fácil elevar al cubo que extraer la raíz cúbica. Obsérvese, no obstante, que para esta función se tiene trivialmente que $f^{-1}(0) = 0$, $f^{-1}(\pm 1) = \pm 1$. Con este sencillo ejemplo se ve la necesidad de precisar que la función inversa $f^{-1}(y)$ es difícil de computar para “la mayoría” de los elementos de la imagen de f .

No es fácil aquilatar la noción de “facilidad” o “dificultad” de cálculo, pues ello dependerá del estado de la tecnología y la ciencia de la computación en cada instante de tiempo. Sin embargo, podríamos decir que una función se puede considerar unidireccional cuando el esfuerzo necesario para calcular la función inversa se mantenga muchos órdenes de magnitud por encima del necesario para calcular la directa. Por ejemplo, en [88], los autores afirman que el esfuerzo criptoanalítico debe ser por lo menos 10^9 veces mayor que el esfuerzo para cifrar o descifrar.

2.1.5 Función unidireccional *exponenciación discreta*

Una vez introducido el concepto de función unidireccional (véase sección 2.1.4), es fácil aplicarlo inmediatamente al ejemplo que venimos considerando hasta ahora: el cambio de clave de Diffie-Hellman, que está asociado, como hemos visto, al problema del logaritmo discreto.

La función unidireccional que aparece en este caso es justamente la *exponenciación discreta* y su inversa, el *logaritmo discreto*.

Tiempo de ejecución de la *exponenciación discreta*

Sea G un grupo cíclico de orden n y α un generador de G . La función *exponenciación discreta* consiste en realizar el cómputo

$$\alpha^k, \quad 1 \leq k \leq n.$$

Por ejemplo, si elegimos como grupo \mathbb{Z}_m^* , es decir, el grupo multiplicativo de las unidades del grupo de los enteros módulo m , se demuestra (véase por ejemplo [26, sección 1.2], [57, sección 4.6.3]) que el tiempo de computación de la exponenciación discreta, o sea, el cálculo de a^k , $1 \leq k \leq \varphi(m)$, donde a es un elemento cualquiera de \mathbb{Z}_m^* y $\varphi(m)$ el indicador de Euler, es

$$O\left((\log_2 m)^2 \cdot \log_2 |k|\right) \leq O\left((\log_2 m)^3\right) \quad (2.2)$$

operaciones bit, en donde ya hemos utilizado la notación asintótica definida en la sección 1.2. Nótese que el tiempo de ejecución del producto de dos números menores o iguales que m es $(\log_2 m)^2$, de modo que el tiempo de cálculo de la exponenciación discreta es $\log_2 k$ veces el de un producto. En efecto, para calcular a^k no es necesario efectuar $k - 1$ productos porque se puede usar la expresión de

k en base 2. Por ejemplo, para a^{25} basta efectuar 6 productos, como se indica a continuación:

$$a^{25} = a^{2^4+2^3+1} = \left(\left((a^2)^2 \right)^2 \right)^2 \cdot \left((a^2)^2 \right)^2 \cdot a.$$

Ésta es precisamente la idea que da lugar al tiempo de computación indicado en la fórmula (2.2).

El tiempo de computación expresado en (2.2) es polinómico, lo cual hace ver que la exponenciación discreta es una función “fácil” desde el punto de vista computacional. Recordemos (véase la sección 1.2) que de un algoritmo se dice que se ejecuta en *tiempo polinómico* si su tiempo de ejecución es polinómico respecto del número de bits del parámetro n que rige su entrada. Pasemos ahora a su función inversa.

Tiempo de ejecución del *logaritmo discreto*

Sea nuevamente G un grupo cíclico de orden n y α un generador de G . La función inversa de la exponencial discreta, esto es, el *logaritmo discreto* $\log_\alpha: G \rightarrow \mathbb{Z}_n$, es una función que requiere, como veremos, mucho tiempo de computación. El problema consiste, dado un valor β , calcular $e \in \mathbb{Z}_n$ tal que $\beta = \alpha^e$. En este caso, se escribe $e = \log_\alpha \beta$.

Indiquemos brevemente los algoritmos actuales para el cálculo del logaritmo discreto junto con sus tiempos de ejecución.

1. La búsqueda exhaustiva: consiste simplemente en calcular los valores sucesivos $\alpha^0, \alpha^1, \alpha^2, \dots$ hasta obtener β . Este método requiere $O(\#G)$ operaciones en el grupo, lo que es muy ineficiente a poco grande que sea $\#G$.
2. Método del “paso gigante-paso enano”⁴ (cf. [73, sección 3.6.2]). Pongamos $m = \lceil \sqrt{n} \rceil$. Observemos que, si $\beta = \alpha^e$, entonces podemos escribir $e = im + j$, con $0 \leq i < m$, $0 \leq j < m$. De aquí, $\alpha^e = \alpha^{im}\alpha^j$, es decir, $\beta(\alpha^{-m})^i = \alpha^j$. Ello induce a seguir la siguiente estrategia: se construye una tabla con entradas (j, α^j) con $0 \leq j < m$ y se ordena por la segunda componente. Se hace $\gamma \leftarrow \beta$ y se busca γ en la tabla. Si se encuentra, entonces $e = j$. En caso contrario, se repite i veces la sustitución $\gamma \leftarrow \gamma \cdot \alpha^{-m}$ hasta conseguir que γ sea uno de los que aparece en la tabla, con lo que $e = im + j$.

Construir la tabla requiere $O(\sqrt{n})$ multiplicaciones y $O(\sqrt{n} \log_2 n)$ comparaciones para ordenarla. Una vez construida la tabla, se requieren $O(\sqrt{n})$ multiplicaciones y $O(\sqrt{n})$ búsquedas en ella. Suponiendo que, en promedio, una multiplicación en el grupo tarda más tiempo que $\log_2 n$ comparaciones, el tiempo de ejecución resulta ser $O(\sqrt{n})$.

3. Algoritmo ρ de Pollard para el cálculo del logaritmo discreto. No lo detallaremos, pues su tiempo de ejecución es el mismo que el del apartado anterior; véase, por ejemplo, [73, sección 3.6.3].

⁴En inglés: “baby-step giant-step”.

4. Método del cálculo del índice (*cf.* [73, sección 3.6.5]). Este método es el más potente que se conoce en la actualidad a la hora de calcular logaritmos discretos, aunque no es aplicable a todo tipo de grupos. Requiere la selección de un subconjunto relativamente pequeño S de G , llamado la *base de factores*, de tal manera que una fracción significativa de los elementos de G se pueda expresar como productos de elementos de S . Hecho esto, el algoritmo precalcula una tabla que contenga los logaritmos de todos los elementos de S , de la que luego se vale para calcular el logaritmo de cualquier elemento de G que se le pida.

Los pasos son los siguientes:

- 4.1 Elijamos un subconjunto $S = \{p_1, p_2, \dots, p_s\}$ de G como base de factores.
- 4.2 Generemos un conjunto de relaciones lineales entre los logaritmos de los elementos de S de la siguiente manera:
 - 4.2.1 Escojamos un entero k aleatoriamente, con $0 \leq k \leq n - 1$ y calculemos α^k .
 - 4.2.2 Tratemos de escribir α^k como producto de elementos de S :

$$\alpha^k = \prod_{i=1}^s p_i^{c_i}, \quad c_i \geq 0. \quad (2.3)$$

Si lo conseguimos, tomemos logaritmos en base α en ambos miembros de (2.3) para obtener la relación lineal

$$k \equiv \sum_{i=1}^s c_i \log_\alpha p_i \pmod{n}. \quad (2.4)$$

- 4.2.3 Repitamos los dos pasos anteriores hasta conseguir $s + c$ relaciones de la forma (2.4), donde c es un valor pequeño (por ejemplo, 10) de forma que el sistema de ecuaciones dado por las $s + c$ relaciones tenga solución única con toda probabilidad.
- 4.3 Resolvamos módulo n (usando el teorema chino del resto) las $s + c$ congruencias lineales de la forma (2.4) para obtener así los valores de $\log_\alpha p_i$, $1 \leq i \leq s$.
- 4.4 Para calcular ahora el logaritmo de β , que era nuestro problema, demos los pasos siguientes:
 - 4.4.1 Escojamos un entero k aleatoriamente, con $0 \leq k \leq n - 1$ y calculemos $\beta \cdot \alpha^k$.
 - 4.4.2 Tratemos de escribir $\beta \cdot \alpha^k$ como producto de elementos de S :

$$\beta \cdot \alpha^k = \prod_{i=1}^s p_i^{d_i}, \quad d_i \geq 0.$$

Si lo conseguimos, análogamente a como se hizo en uno de los pasos previos, tomando logaritmos en base α en ambos miembros, tendremos $\log_\alpha \beta = (\sum_{i=1}^s d_i \log_\alpha p_i - k) \pmod{n}$ y hemos terminado. Si no lo conseguimos, repetiremos los dos pasos anteriores.

Este algoritmo presenta el problema de la selección del subconjunto S y de la generación de las relaciones (2.3) y (2.4). En el caso en que $G = \mathbb{Z}_p^*$, se puede seleccionar como base de factores los s primeros números primos; para generar una relación de tipo (2.3), se computa $\alpha^k \pmod{p}$ y se usa el algoritmo de la división entera para ver si ese entero es un producto de los s primeros primos. El tiempo de ejecución del algoritmo es

$$O\left(\exp\left((c + \varepsilon_n)\sqrt{\ln p \cdot \ln(\ln p)}\right)\right)$$

operaciones bit, donde c es una constante y $\varepsilon_n \rightarrow 0$. Véase, por ejemplo, [72].

Otro grupo interesante es $G = \mathbb{F}_{2^n}^*$, representado por los polinomios en $\mathbb{Z}_2[x]$ de grado menor o igual que n , en donde las multiplicaciones se realizan módulo un polinomio prefijado $f(x)$ de grado n irreducible en $\mathbb{Z}_2[x]$. La base de factores S se puede elegir como el conjunto de polinomios irreducibles en $\mathbb{Z}_2[x]$ de grado menor o igual que una cota b prefijada. Nuevamente, para generar una relación de tipo (2.3), se calcula $\alpha^k \pmod{f(x)}$ y se usa el método de las divisiones para comprobar si ese polinomio es producto de los polinomios de S . En este grupo, el tiempo de ejecución del algoritmo se puede acotar por

$$O\left(\exp\left((c + \varepsilon_n)\sqrt{n \cdot \ln n}\right)\right)$$

operaciones bit, donde c y ε_n son, como antes, una constante y una sucesión que tiende a 0, respectivamente.

5. El método de la criba de los cuerpos de números de Lenstra ([11, 20, 61]). Actualmente se considera este método, que es una variante del método del cálculo del índice, como el mejor algoritmo para calcular logaritmos en \mathbb{Z}_p^* . Su tiempo de ejecución es

$$O\left(\exp\left((c + \varepsilon_n)(\ln p)^{\frac{1}{3}}(\ln(\ln p))^{\frac{2}{3}}\right)\right),$$

donde $c = \sqrt[3]{64/9} < 1,923$ y ε_n es, como antes, una sucesión que tiende a 0.

Para el caso $\mathbb{F}_{2^n}^*$, el mejor algoritmo, también una variante del cálculo del índice, se debe a Coppersmith ([26]) para el que el tiempo de ejecución es

$$O\left(\exp\left((c + \varepsilon_n)n^{\frac{1}{3}}(\ln n)^{\frac{2}{3}}\right)\right),$$

donde $c < 1,405$ y ε_n son, como antes, una constante y una sucesión que tiende a 0, respectivamente.

Observación 2.3 En resumen, se ve que todos los métodos de cómputo del logaritmo discreto mostrados presentan tiempos de ejecución de tipo (sub)exponencial, con lo que queda justificada la caracterización de función unidireccional que tiene la exponenciación discreta, tal como anunciamos al principio de la sección.

2.2 Criptosistemas de clave pública

Equipados con los conceptos y definiciones que se han presentado hasta aquí, pasamos a formalizar el tratamiento de los criptosistemas de clave pública.

2.2.1 Formalismo de clave pública

Definición 2.4 Una función unidireccional con trampilla⁵ es una función unidireccional que se puede invertir fácilmente cuando se conoce una información adicional, llamada justamente “trampilla”.

Definición 2.5 Sea \mathcal{M} el conjunto de los mensajes en claro y \mathcal{C} el conjunto de los mensajes cifrados. Un criptosistema de clave pública es una familia de funciones unidireccionales con trampilla $f_k: \mathcal{M} \rightarrow \mathcal{C}$ tales que la trampilla t_k es fácil de obtener para cada $k \in \mathcal{K}$. Además, para cada $k \in \mathcal{K}$, se debe poder escribir un algoritmo eficiente que permita calcular f_k pero haga inabordable el cálculo de k y de t_k .

Resumiendo:

1. dada una clave $k \in \mathcal{K}$, la trampilla t_k es “fácil” de obtener;
2. para cada $k \in \mathcal{K}$, existe un algoritmo eficiente que calcula f_k ;
3. sin el conocimiento de la clave k , no existe ningún algoritmo eficiente que permita obtener t_k .

Cada usuario A elige una clave $k_A \in \mathcal{K}$, y publica un algoritmo E_A que permite calcular eficientemente la función $f_A = f_{k_A}$.

El algoritmo E_A recibe el nombre de clave pública de A y la trampilla $t_{k_A} = t_A$ es la clave privada de A .

Si el usuario A quiere enviar un mensaje m al usuario B , ha de dar los siguientes pasos:

1. Obtiene la clave pública de B , E_B , previamente publicada por B de acuerdo con la Definición 2.5.
2. Calcula f_B .
3. Transmite $f_B(m) = c$ a B .

Para recuperar el mensaje, B , que es el único capaz de invertir f_B rápidamente, calcula

$$(f_B)^{-1}(c) = (f_B)^{-1}(f_B(m)) = m, \quad (2.5)$$

recobrando de este modo el texto en claro.

⁵En inglés, “TOF, trapdoor one-way function”.

2.2.2 Firma digital

El protocolo de clave pública permite que el destinatario se asegure de que el mensaje que recibe ha sido enviado de verdad por quien dice ser el remitente. Para ello se usa el siguiente *protocolo de firma digital*.

Definición 2.6 Se llama rúbrica r de un usuario A para un mensaje m al resultado de descifrar m como si fuera un mensaje cifrado que A recibe; esto es,

$$r = f_A^{-1}(m).$$

Definición 2.7 Se llama firma digital s de un usuario A para un mensaje m con destinatario B , al resultado de cifrar la rúbrica r de m ; esto es,

$$s = f_B(r) = f_B(f_A^{-1}(m)).$$

El proceso que debe seguir A para enviar un mensaje firmado m a B es el siguiente:

1. A cifra el mensaje

$$c = f_B(m).$$

2. A calcula la rúbrica del mensaje

$$r = f_A^{-1}(m).$$

3. A calcula la firma digital de m

$$s = f_B(r).$$

4. A envía a B el par (s, c) .

Por su parte, B , tras recuperar el mensaje, debe verificar que la firma es correcta, es decir, que el mensaje procede realmente de A . Para ello, da los pasos siguientes:

1. Recupera el mensaje, aplicando la función inversa:

$$f_B^{-1}(c) = f_B^{-1}(f_B(m)) = m.$$

2. Calcula la rúbrica de A :

$$f_B^{-1}(s) = f_B^{-1}(f_B(r)) = r.$$

3. Comprueba que se verifica:

$$f_A(r) = m,$$

esto es, la rúbrica, cifrada como si fuera un mensaje para ser enviado a A , debe coincidir con el mensaje en claro. En tal caso, la firma es correcta.

Observación 2.8 La firma digital permite asegurar la autenticidad del mensaje si efectivamente $f_A(r)$ coincide con m , porque sólo A con su clave privada puede calcular la rúbrica r a partir del mensaje m .

Observación 2.9 En la práctica, enviar la rúbrica $r = f_A^{-1}(m)$ es peligroso porque el criptoanalista puede obtener m usando la clave pública de A ; pero si sólo interesa la autenticidad de m puede enviarse (r, c) .

2.3 Criptoanálisis para los sistemas de clave pública

Como dijimos en la sección 2.1.1, el Criptoanálisis engloba el conjunto de técnicas matemáticas que permiten abordar el problema de romper un criptosistema dado.

En Criptografía de Clave Pública existen diversos ataques tipificados que pasamos a describir. El objetivo general de estos ataques es tratar sistemáticamente de recuperar el texto en claro a partir del texto cifrado sin el conocimiento de la clave e incluso, si fuera posible, la misma clave.

Para iniciar un ataque, es preciso disponer de cierta información parcial acerca del texto en claro o del criptograma. Suponemos que el criptoanalista conoce perfectamente el mecanismo de cifrado y descifrado, pero desconoce la clave. El tipo de observaciones y manipulaciones que el criptoanalista pone en juego determinan la clase de ataques, que pueden clasificarse en dos grandes grupos:

1. *Ataque pasivo.* Es aquel en que el adversario realiza una escucha del canal de comunicación (diríamos que “pincha el canal”) poniendo en peligro la confidencialidad del mensaje que por él viaja.
2. *Ataque activo.* Es aquel en que el criptoanalista trata de añadir, borrar o modificar de alguna manera el mensaje, poniendo así en peligro no sólo la confidencialidad sino también la integridad del mismo mensaje.

En las secciones que siguen, expondremos algunos de estos ataques sin pretensión de exhaustividad.

2.3.1 Ataques pasivos

El objetivo de estos ataques es lograr un procedimiento sistemático que permita recuperar el texto en claro a partir del criptograma e, incluso, obtener la clave privada. Los principales son los siguientes:

1. *Ataque del criptograma conocido.* El adversario trata de deducir el texto en claro (o incluso la clave) a partir de la observación de un criptograma, sin tener conocimiento previo en absoluto del texto en claro correspondiente. Un criptosistema vulnerable a este ataque es considerado totalmente inseguro.
2. *Ataque del texto claro conocido.* Aquí el criptoanalista conoce todo o parte de un texto en claro y su criptograma correspondiente, de donde trata de deducir el texto en claro en general o, mejor aún, la clave.
3. *Ataque del texto claro elegido.* En este caso, el adversario tiene la posibilidad de elegir un determinado texto en claro y obtener el criptograma que le corresponde. De ello trata de deducir información que le permita después conocer el texto en claro a partir de un criptograma cualquiera, como ocurriría si, por ejemplo, consigue averiguar la clave.
4. *Ataque adaptativo del texto claro elegido.* Una variante del anterior en que el criptoanalista no sólo puede elegir el texto en claro a cifrar, sino que puede modificar esa elección en función de los cífrados anteriores.

5. *Ataque del criptograma elegido.* Ahora el adversario recibe la posibilidad de seleccionar cualquier criptograma y se le entrega el texto en claro correspondiente. Supongamos que el adversario dispone de la “máquina” de descifrar, pero no de la clave. En tal caso, puede alimentarla con cualquier criptograma para ver qué texto en claro resulta y de ahí, deducir información suficiente como para poder descifrar cualquier criptograma sin tener ya acceso a tal “máquina”.
6. *Ataque adaptativo del criptograma elegido.* Una variante del anterior en que la elección del criptograma se puede hacer depender de los textos en claro obtenidos en los descifrados anteriores.

Observación 2.10 El ataque más propio de los sistemas de clave pública es el del texto claro elegido, porque por definición, la clave de cualquier usuario del sistema es pública. Por tanto, es posible cifrar cuantos textos en claro se desee.

Observación 2.11 Hay algunos otros ataques pasivos que no implican ningún criptoanálisis: por ejemplo, si a un usuario se le instala un virus en su teclado que registra todo lo que teclea, el adversario puede llegar a conocer una clave sin necesidad de criptoanalizar ningún dato. El sistema queda totalmente comprometido, especialmente si el usuario no advierte la maniobra.

2.3.2 Ataques activos

Recordemos que, en este tipo de ataques, el adversario interviene activamente en el canal de comunicación, modificando la información transmitida entre los usuarios del sistema. En este contexto, podemos destacar:

1. *Ataque del intermediario.* El adversario se sitúa en medio del canal y tiene capacidad de sustituir toda o parte de la información transmitida entre las partes.
2. *Ataque de la duración.* Se trata de un moderno ataque que se basa en la investigación precisa del tiempo de ejecución de las operaciones que implican una exponentiación modular.
3. *Ataque del retraso forzado.* Este tipo de ataque es una variante del *ataque del intermediario*, en que éste intercepta el mensaje y lo retransmite en un momento posterior. Para frustrar estos ataques, algunos protocolos imponen que el mensaje incorpore una información horaria.

2.4 Criptosistema RSA

El criptosistema RSA fue introducido por Rivest, Shamir y Adleman en [98]. Se trata del criptosistema de clave pública más ampliamente usado y se basa en la dificultad computacional de factorizar números enteros. Si bien no ha sido demostrada rigurosamente la equivalencia entre factorizar el módulo y romper el criptosistema RSA, existen variantes del mismo para las cuales sí ha sido probada tal equivalencia; véanse sección 2.5.1 y sección 2.5.2. No obstante, existen

algunos trabajos tendentes a demostrar que, bajo ciertas condiciones, romper el criptosistema RSA puede no ser equivalente a factorizar; véase [17].

A continuación, describiremos brevemente el protocolo RSA.

2.4.1 Elección de claves

Cada usuario A que quiera utilizar el criptosistema procede como sigue:

1. Elige dos números primos distintos p_A, q_A y calcula $n_A = p_A \cdot q_A$ y $\varphi(n_A) = (p_A - 1)(q_A - 1)$. Hay especificaciones sobre p_A, q_A que afectan a la seguridad del sistema, como luego veremos.
2. Elige un entero aleatorio e_A , tal que

$$\begin{aligned} 1 < e_A < \varphi(n_A), \\ \text{mcd}(e_A, \varphi(n_A)) = 1. \end{aligned}$$

3. Utiliza el algoritmo de Euclides extendido para calcular el inverso de e_A módulo $\varphi(n_A)$; esto es, el único entero d_A tal que:

$$\begin{aligned} 1 < d_A < \varphi(n_A), \\ d_A \cdot e_A \equiv 1 \pmod{\varphi(n_A)}. \end{aligned}$$

La clave pública de A es el par (n_A, e_A) , y su clave privada el entero d_A . Los valores de p_A, q_A y $\varphi(n_A)$ también deben permanecer secretos. El usuario A deposita su clave pública en un directorio de claves con acceso universal.

Definición 2.12 Los valores e_A y d_A se denominan *exponente de cifrado* y *exponente de descifrado*, respectivamente.

Definición 2.13 El valor n_A se denomina *módulo del criptosistema*.

2.4.2 Envío de mensajes

En este sistema, los mensajes que se transmiten son elementos de \mathbb{Z}_{n_A} ; si se quisiera transmitir un mensaje más largo, debe ser subdividido en trozos, de tal manera que cada uno sea un elemento de \mathbb{Z}_{n_A} .

Supongamos ahora que B quiere enviar un mensaje m a A . Consultando el directorio público de claves, B accede al par (n_A, e_A) , correspondiente a A . Con ellos, calcula $c = m^{e_A} \pmod{n_A}$, criptomensaje que envía a A .

2.4.3 Descifrado del mensaje

Para recuperar el mensaje, A calcula

$$c^{d_A} = (m^{e_A})^{d_A} = m^{e_A d_A} \equiv m \pmod{n_A}. \quad (2.6)$$

Esto es sencillo, pues A está en posesión de su clave privada d_A . Vamos ahora a demostrar que la expresión (2.6) se cumple. Puesto que, por construcción, $d_A \cdot e_A \equiv 1 \pmod{\varphi(n_A)}$, esto significa que se verifica:

$$d_A e_A = k\varphi(n_A) + 1 = k(p_A - 1)(q_A - 1) + 1.$$

Por tanto

$$m^{e_A d_A} = m^{k(p_A - 1)(q_A - 1) + 1} = m(m^{(p_A - 1)(q_A - 1)})^k.$$

Tomando módulo p , tenemos

$$m^{e_A d_A} = m(m^{(p_A - 1)})^{k(q_A - 1)} \equiv m \pmod{p}.$$

Si p no es un divisor de m , esta congruencia es cierta por el teorema de Fermat. De lo contrario, es trivial, pues ambos lados de la congruencia serían idénticamente nulos. Por un razonamiento análogo, se sigue que

$$m^{e_A d_A} = m(m^{(q_A - 1)})^{k(p_A - 1)} \equiv m \pmod{q}.$$

Por último, recordando que p y q son primos distintos y que $m \in \mathbb{Z}_{n_A}$, queda finalmente

$$m^{e_A d_A} \equiv m \pmod{n_A}.$$

El sistema RSA es verdaderamente un criptosistema, pues se verifica la condición fundamental $f_A^{-1}(f_A(m)) = m$; véase la ecuación (2.5).

2.4.4 Condiciones de p, q y criptoanálisis

De modo general, los primos p y q deben elegirse de manera que factorizar $n = p \cdot q$ sea computacionalmente muy difícil, porque si se conocen p y q también se conoce $\varphi(n)$ y se puede calcular la clave privada d . Veamos a continuación las condiciones específicas que deben satisfacer p, q para que su factorización sea muy difícil.

(1) p y q deben tener aproximadamente la misma longitud porque a medida que un factor de n sea más pequeño es tanto más fácil de obtener. En la actualidad se recomienda que p y q tengan cada uno una longitud mínima de 512 bits (acerca del tamaño recomendado para los tamaños de las claves en este y otros criptosistemas, véase el interesante estudio [63]).

(2) p y q no deben estar demasiado cerca, porque si lo están, entonces $\frac{p+q}{2} \approx \sqrt{n}$, de modo que la diferencia

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

es pequeña. En tal caso, con pocos tanteos se puede encontrar un entero $x > \sqrt{n}$, tal que $x^2 - n = y^2$. Entonces (suponiendo $p > q$), se obtienen $p = x + y$, $q = x - y$.

- (3) $\text{mcd}(p-1, q-1)$ debe ser “pequeño”. En efecto, si $\text{mcd}(p-1, q-1)$ es grande, entonces

$$M = \text{mcm}(p-1, q-1) = \frac{\varphi(n)}{\text{mcd}(p-1, q-1)}$$

es pequeño y cada d' tal que $e \cdot d' \equiv 1 \pmod{M}$ sirve para descifrar; esto es, se verifica

$$m^{ed'} \equiv m \pmod{n},$$

para todo mensaje m . Esta propiedad se comprueba mediante el mismo razonamiento de sección 2.4.3, sin más que sustituir $\varphi(n)$ por M . Así pues, si M es pequeño, se podría romper el sistema de la siguiente manera:

- (a) Elegir un valor de M y calcular el inverso d' de e módulo M .
- (b) Cifrar varios mensajes sucesivos con el criptosistema y tratar de descifrarlos usando el exponente d' .
- (c) Si en alguno de los ensayos del punto (b) se tiene éxito, el sistema queda roto. En caso contrario, se repite el paso (a).

Si M es realmente pequeño en comparación con $\varphi(n)$ el éxito de los pasos anteriores puede alcanzarse con relativa eficacia computacional (véase [101, sección 4.2]).

- (4) $p-1$ y $q-1$ deben contener un factor primo “grande”.

- (5) d debe ser de longitud aproximadamente igual a la de n . Si

$$\text{nº de bits de } d \leq \frac{1}{4} (\text{nº de bits de } n),$$

entonces existe un algoritmo eficiente para calcular d (véase [120]). Otros trabajos más recientes, refinan los resultados de Wiener en cuanto al tamaño del exponente de descifrado; véase en concreto [16].

Se puede elegir primero d aleatoriamente y luego calcular e .

- (6) Elegir e pequeño facilita el cifrado. El menor valor es $e = 3$. Recuérdese que $\text{mcd}(e, \varphi(n)) = 1$ y $\varphi(n)$ siempre es múltiplo de 4.

Las condiciones (1)–(4) fueron originariamente introducidas de un modo informal en [98] (para una formulación más precisa, véase [73, Note 8.8]). Las tres primeras son propiedades relativas a la pareja p, q , mientras que la cuarta es una propiedad individual de cada uno.

Algunos autores recomiendan que los primos p y q sean de los llamados robustos para garantizar que las propiedades (3) y (4) se verifiquen y se soslayen los ataques mediante los algoritmos $p-1$ de Pollard y $p+1$ de Williams, que se explicarán en las secciones 3.3.1 y 3.3.3, respectivamente. De modo preciso:

Definición 2.14 Un primo impar p se dice que es robusto si verifica las tres siguientes condiciones:

- (a) $p - 1$ tiene un factor primo grande r .
- (b) $p + 1$ tiene también un factor primo grande s .
- (c) $r - 1$ tiene también un factor primo grande t .

Observación 2.15 En la sección 5.1 se explicará el sentido de las condiciones (a)-(c) anteriores.

Observación 2.16 La seguridad del RSA se basa en la dificultad de factorizar números enteros. Si un criptoanalista es capaz de factorizar el módulo n , entonces podría calcular $\varphi(n)$ y d , y descifrar los mensajes como si fueran dirigidos a él. Por otro lado, si un criptoanalista llega por algún procedimiento a conocer d , entonces no es difícil demostrar que puede factorizar n fácilmente. Otro posible caso es que el criptoanalista conozca $\varphi(n)$: también en este caso sería fácil factorizar n , dadas las identidades

$$\begin{aligned} p + q &= n - \varphi(n) + 1, \\ (p - q)^2 &= (p + q)^2 - 4n, \\ q &= \frac{1}{2}[(p + q) - (p - q)]. \end{aligned}$$

Por tanto, el problema de averiguar d , o alternativamente, $\varphi(n)$, a partir de la clave pública (n, e) es computacionalmente equivalente a factorizar n .

Observación 2.17 Varios usuarios pueden tener el mismo e pero deben tener todos distinto n . En caso contrario, el conocimiento que un usuario tiene de su par (e_i, d_i) le permitiría conocer el exponente de descifrado de todos los demás.

Observación 2.18 No se deben usar valores pequeños de e al enviar un mismo mensaje a varios destinatarios. Supongamos que $e = 3$ y que un usuario desea enviar el mismo mensaje m a tres destinatarios diferentes, cuyos módulos son n_1 , n_2 y n_3 . En tal caso, los criptogramas serían $c_i = m^3 \pmod{n_i}$ para $i = 1, 2, 3$. Puesto que los módulos n_i van a ser con toda probabilidad primos entre sí, un espía que conozca c_1 , c_2 y c_3 , podría plantear y encontrar una solución x , con $0 \leq x < n_1 n_2 n_3$ al siguiente sistema de congruencias simultáneas:

$$\begin{aligned} x &\equiv c_1 \pmod{n_1} \\ x &\equiv c_2 \pmod{n_2} \\ x &\equiv c_3 \pmod{n_3} \end{aligned}$$

Dado que $m^3 < n_1 n_2 n_3$, se ha de cumplir, por el teorema chino del resto, que $x = m^3$. El adversario puede, entonces, recobrar el mensaje m sin más que extraer la raíz cúbica de x . Para un estudio detallado de esta cuestión, planteada con toda generalidad, resultan muy interesantes las referencias [52, 53].

Observación 2.19 Un mensaje es “inocultable” si

$$m^e \equiv m \pmod{n}.$$

El número de mensajes inocultables es (véase [12]):

$$[1 + \text{mcd}(e - 1, p - 1)] \cdot [1 + \text{mcd}(e - 1, q - 1)].$$

Si se eligen p, q y e aleatoriamente el número de mensajes inocultables es muy pequeño y no afecta a la seguridad.

2.4.5 Firma digital en el RSA

Para que el usuario A envíe la firma digital de un mensaje m al usuario B , debe hacer lo siguiente:

- 1) A calcula la rúbrica de m ; esto es,

$$r = m^{d_A} \pmod{n_A}.$$

- 2) A calcula la firma digital de m ; esto es,

$$s = r^{e_B} \pmod{n_B}.$$

Para que B verifique la firma de A , debe hacer lo siguiente:

- 3) B recupera la rúbrica de A :

$$s^{d_B} \pmod{n_B} = r.$$

- 4) B comprueba que la rúbrica encriptada coincide con el mensaje:

$$r^{e_A} \pmod{n_A} = m.$$

El ataque contra el protocolo de firma digital en el RSA es el mismo que el que hay que llevar a cabo para romper el propio criptosistema.

2.4.6 Firma digital en una red

Para que A pueda calcular $r = m^{d_A} \pmod{n_A}$, el mensaje debe estar en el rango $2 \leq m \leq n_A - 1$, y para que A pueda enviar m a B el mensaje debe de estar en el rango $2 \leq m \leq n_B - 1$. Además, para que A pueda calcular $s = r^{e_B} \pmod{n_B}$ la rúbrica debe estar en el rango $2 \leq r \leq n_B - 1$. Si $r > n_B$, entonces hay que "trocear" r . Este problema se conoce como "reblocking". Se han propuesto dos modos de resolver este problema.

- (i) El primero es el método del "umbral"⁶, que ya introdujeron Rivest, Shamir y Adleman en el artículo fundacional [98]; véase también [119]. Consiste en lo siguiente:

- a) Se elige un *umbral* $h \approx 10^{199}$.

⁶En inglés, "threshold".

- b) Cada usuario A publica dos claves públicas (n_A, e_A) y (n'_A, e'_A) , tales que: $n'_A < h < n_A$. Sea d'_A la clave privada de (n'_A, e'_A) .
- c) La clave (n_A, e_A) se usa para cifrar y la clave (n'_A, e'_A) para verificar la firma digital.
- d) Los mensajes deben estar en el rango

$$1 < m < \min_A \{n'_A\}.$$

(Si un mensaje no cumple esta condición se divide en bloques que sí la cumplen.)

- e) A cifra m como siempre $c = m^{e_B} \pmod{n_B}$.
- f) A calcula la rúbrica y la firma digital como sigue:

$$r = m^{d'_A} \pmod{n'_A},$$

$$s = r^{e_B} \pmod{n_B}.$$

- g) A envía a B el par (s, c) .
- h) B calcula

$$m = c^{d_B} \pmod{n_B},$$

$$r = s^{d_B} \pmod{n_B},$$

y comprueba que se verifica

$$m = r^{e'_A} \pmod{n'_A}.$$

- (ii) El segundo método consiste en especificar una forma especial para el módulo. En este método, p y q se seleccionan de forma que n tenga la siguiente expresión: el bit más significativo es 1 y los siguientes k bits son todos cero; esto es,

$$n = 1\underbrace{0\dots 0}_{(k)}abc\dots_{(2)}$$

Para conseguir un módulo n con un determinado número de bits t , se procede como sigue. Por hipótesis,

$$2^{t-1} \leq n < 2^{t-1} + 2^{t-k-1}.$$

Se elige un primo p aleatorio con $\lceil \frac{t}{2} \rceil$ bits y se busca un primo q en el intervalo

$$\left(\left\lceil \frac{2^{t-1}}{p} \right\rceil, \left\lfloor \frac{2^{t-1} + 2^{t-k-1}}{p} \right\rfloor \right).$$

Entonces $n = pq$ es un módulo con las propiedades requeridas. Esta elección de n reduce la probabilidad de aparición de “reblocking” a un valor menor que 2^{-k} . Para más detalles véase [72, Example 11.22].

2.5 Otros criptosistemas de tipo RSA

2.5.1 Criptosistema de Rabin

Aunque no ha sido probado, se admite comúnmente que la seguridad del criptosistema RSA es equivalente a la factorización de su módulo. El criptosistema de Rabin es, en cambio, el primer ejemplo de un esquema de cifrado para el que sí se ha probado que su seguridad es equivalente computacionalmente a factorizar un número, como el propio Rabin demuestra en [91, Teorema 2].

La seguridad de este criptosistema se apoya en la dificultad para resolver el *problema de la raíz cuadrada*, cuya definición es la que sigue:

Definición 2.20 Sea n un entero compuesto y sea a un resto cuadrático módulo n . El *problema de la raíz cuadrada* consiste en encontrar un entero b tal que $b^2 \equiv a \pmod{n}$.

Observación 2.21 Si los factores p y q de n son conocidos, entonces el problema de la raíz cuadrada se puede resolver eficientemente buscando las raíces cuadradas de a módulo p y módulo q , aplicando a continuación el teorema chino del resto. Recíprocamente, si existe un algoritmo para resolver el problema de la raíz cuadrada, entonces se puede factorizar n eficientemente; véase [91, Teorema 1].

Descripción del criptosistema

Para la elección de claves, un usuario A del sistema ha de efectuar los pasos siguientes:

1. A genera dos primos distintos, p y q , que cumplan las mismas condiciones que las exigidas en el criptosistema RSA (véase sección 2.4.4).
2. A calcula $n = p \cdot q$.
3. La clave pública de A es n ; su clave privada es el par (p, q) .

Los mensajes son elementos $m \in \mathbb{Z}_n$. Para que A envíe m a B ha de hacer lo siguiente:

1. A obtiene la clave pública de B , sea ésta n_B .
2. A calcula $c = m^2 \pmod{n_B}$.
3. A envía c a B .

Para recuperar el mensaje, B ha de hacer lo siguiente:

1. B halla las cuatro raíces cuadradas de c , dos de ellas módulo p y las otras dos módulo q , usando, por ejemplo, el algoritmo de A. Tonelli descrito en [5, sección 7.1] o también en [72, sección 3.34].
2. Aplicando el teorema chino del resto, obtiene cuatro posibles mensajes, m_1, m_2, m_3, m_4 , uno de los cuales es el realmente enviado.

Un inconveniente de este sistema es que el receptor se ve abocado a la tarea de averiguar cuál de los cuatro mensajes que obtiene es el auténtico. En la práctica, esa ambigüedad es fácil de resolver (por ejemplo, uno de los mensajes tiene contenido semántico y los otros no).

Una estrategia para superar esta dificultad es añadir al principio del texto en claro una cadena preestablecida de caracteres, que llamamos *redundancia*. Entonces, con altísima probabilidad, sólo una de las cuatro raíces contendrá la redundancia, con lo que el receptor puede fácilmente resolver la ambigüedad. Además, si ninguna de las raíces la contiene, B puede rechazar el criptomensaje como fraudulento.

Este criptosistema es vulnerable básicamente a los mismos ataques que el criptosistema RSA, razón por la que se impone que p y q verifiquen las mismas condiciones que los del RSA. También sucumbe frente al ataque del criptograma elegido que se puede armar de la siguiente manera: el adversario selecciona al azar un entero m de \mathbb{Z}_n^* y calcula $c = m^2 \pmod{n}$. El adversario presenta el resultado a la máquina de descifrar de A , que descifra c y devuelve un cierto mensaje y . Luego hay un 50 % de posibilidades de que $y \not\equiv \pm m \pmod{n}$, en cuyo caso $\text{mcd}(m-y, n)$ es uno de los factores de n . Ahora bien, este ataque se puede frustrar fácilmente usando el método de la redundancia, pues la máquina de descifrar de A presentará prácticamente siempre el mensaje correcto (pues las otras tres raíces no van a contener la redundancia preestablecida) o bien fallará al no encontrar la redundancia en ninguna de ellas y no devolverá nada.

2.5.2 Criptosistemas de Williams, Kurosawa *et al.* y Loxton *et al.*

H.C. Williams presentó en [121] un nuevo criptosistema que suponía una variante del RSA con exponente de cifrado par, que presenta algunas ventajas.

Inicialmente, Williams propone utilizar un módulo $n = pq$ donde $p \equiv 3 \pmod{8}$ y $q \equiv 7 \pmod{8}$. A continuación, el autor detalla una función E_1 que, aplicada al mensaje antes de ser cifrado, permite después muy fácilmente distinguir cuál de las cuatro raíces cuadradas que aparecen en el descifrado es la correcta. La función E_1 es fácil de invertir, con lo que, aplicada a la raíz cuadrada, correctamente seleccionada, permite recuperar el mensaje en claro. El propio autor mejoró más tarde su sistema, presentando en [123] una modificación que permitía eliminar la restricción impuesta a los primos p y q .

Igual que en el caso del de Rabin, la seguridad del sistema de Williams es probabilmente equivalente a factorizar un número, aunque es vulnerable al ataque del criptograma elegido. Naturalmente son de aplicación aquí también los comentarios hechos en el sistema de Rabin respecto a la redundancia, como método de frustrar ese ataque.

Un esquema más sencillo y eficiente que también goza de las propiedades de ser probabilmente equivalente a factorizar un número y tener descifrado único fue presentado por Kurosawa *et al.* en [57].

Loxton *et al.* proponen en [64] un criptosistema de tipo RSA en el anillo factorial de los enteros de Eisenstein $\mathbb{Z}[\omega]$, siendo $\omega = \exp(2\pi i/3)$ una raíz cúbica primitiva de la unidad. La clave de cifrado consiste en un módulo n , producto de

dos primos, p y q , en $\mathbb{Z}[\omega]$ y un entero positivo e , como exponente de cifrado. La clave para descifrar es otro entero que actúa de exponente de descifrado. Como siempre, e ha de ser primo con respecto a $\varphi(n) = (p\bar{p}-1)(q\bar{q}-1)$ y d es la solución de la congruencia $de \equiv 1(\text{mod } \varphi(n))$. Los autores demuestran que cuando n es el producto de dos primos primarios en $\mathbb{Z}[\omega]$, p y q , tales que $p \equiv 8 + 6\omega(\text{mod } 9)$ y $q \equiv 5 + 6\omega(\text{mod } 9)$ y el exponente de cifrado $e \equiv 3(\text{mod } 6)$, el problema de la factorización de n y el problema del descifrado son equivalentes.

2.5.3 Los criptosistemas de Takagi

Según se ha visto en sección 2.4.2, el criptosistema RSA necesita que los mensajes que se pretende cifrar puedan representarse mediante enteros de \mathbb{Z}_n . Por tanto, los mensajes que no admitan tal representación han de “trocearse” en bloques cada uno de los cuales sea menor que n . Ello exige un par de operaciones cifrado/descifrado para cada uno de los bloques, lo cual resulta computacionalmente costoso.

Tsuyoshi Takagi ha propuesto recientemente (véase [112, 113]) dos criptosistemas; el primero de ellos trata de paliar esta dificultad, mientras que el segundo consigue mejorar la velocidad del descifrado con respecto a RSA. Pasemos a describirlos brevemente.

El criptosistema n^k

Para la generación de las claves se procede del siguiente modo.

Se eligen dos primos p, q que satisfagan las condiciones exigidas para el RSA y se calcula

$$\begin{aligned} n &= pq, \\ L &= \text{mcm}(p-1, q-1), \end{aligned}$$

junto con dos enteros e, d tales que:

$$\begin{aligned} \text{mcd}(e, L) &= \text{mcd}(e, n) = 1, \\ ed &\equiv 1(\text{mod } L). \end{aligned}$$

La clave pública está formada por e y n , y la clave privada es d , como en RSA.

Un mensaje en claro se representa como un sistema de k bloques,

$$M = (M_0, M_1, \dots, M_{k-1}),$$

tales que $M_0 \in \mathbb{Z}_n^*$, y $M_1, \dots, M_{k-1} \in \mathbb{Z}_n$. Para obtener el mensaje cifrado C se aplica la siguiente fórmula:

$$C \equiv (M_0 + nM_1 + \dots + n^{k-1}M_{k-1})^e (\text{mod } n^k).$$

Para descifrar el procedimiento es más elaborado. En primer lugar se obtiene el bloque M_0 utilizando la clave privada d y calculando

$$M_0 \equiv C^d (\text{mod } n).$$

Este paso es idéntico al correspondiente en RSA. Pero para obtener los demás bloques, M_1, \dots, M_{k-1} , es necesario resolver unas ecuaciones lineales módulo n . El detalle técnico es muy prolífico y se puede consultar en [112], donde también puede verse un programa que permite efectuar eficientemente el proceso de descifrado.

Las operaciones necesarias para cifrar son las mismas que en RSA con la salvedad de éstas han de efectuarse módulo n^k , lo que lo hace comparable a un sistema RSA que usara como módulo precisamente n^k . En cuanto al descifrado, éste se compone de una operación idéntica al de RSA más la resolución de un cierto número de congruencias lineales, lo que, según el autor, es computacionalmente eficiente.

El criptosistema p^kq

En [113], el autor refinó el *criptosistema* n^k sustituyendo el módulo n^k por p^kq ; es decir, en su nueva propuesta sólo se eleva a k uno de los factores.

El proceso de cifrado y descifrado es completamente análogo al de RSA, por lo que no lo detallaremos. En cuanto a los tiempos de ejecución, el tiempo de cifrado es idéntico al de un RSA con módulo p^kq . Sin embargo, el descifrado es más rápido que el RSA equivalente si se usa el método de Quisquater-Couvreur, introducido en [90]. Por ejemplo, si se elige un módulo p^2q de 768 bits, para dos primos p, q de 256 bits cada uno, entonces el descifrado del criptosistema considerado es tres veces más rápido que el RSA con un módulo de 768 bits.

Por contra, en [37] se nos advierte que un número del tipo $n = p^kq$ admite una factorización en tiempo polinómico cuando $k = \epsilon \cdot \ln p$, donde $\epsilon > 0$ es una constante fija.

2.6 Criptosistema de ElGamal

2.6.1 Descripción del criptosistema

Este criptosistema fue introducido por ElGamal en [39]. A continuación se describe su protocolo.

Para la elección de claves se efectúan los dos pasos siguientes:

1. Se elige públicamente un grupo cíclico G de orden n y un generador $\alpha \in G$. (En [39], ElGamal eligió $G = \mathbb{Z}_p^*$, con p primo.)
2. Cada usuario elige un $a \in \mathbb{N}$, que es su *clave privada*, y calcula α^a , que es su *clave pública*.

Los mensajes son elementos $m \in G$. Para que A envíe m a B tiene que hacer lo siguiente:

1. A genera un número aleatorio ν y calcula α^ν .
2. A obtiene la clave pública de B , que será α^b , y calcula $(\alpha^b)^\nu$ y $m \cdot (\alpha^b)^\nu = m \cdot \alpha^{b\nu}$.
3. A envía a B el par $(\alpha^\nu, m \cdot \alpha^{b\nu})$.

Para recuperar el mensaje, B calcula $(\alpha^\nu)^b$, y obtiene el mensaje en claro computando

$$m = \frac{m \cdot \alpha^{b\nu}}{(\alpha^\nu)^b}.$$

2.6.2 Firma digital en ElGamal

Vamos a elegir como grupo de trabajo $G = \mathbb{Z}_p^*$, con p primo, de modo que un mensaje es un entero m , tal que $1 \leq m \leq p - 1$. Para que el usuario A envíe la firma digital de un mensaje m al usuario B , debe hacer lo siguiente:

1. A genera un número aleatorio h tal que

$$\text{mcd}(h, p - 1) = 1.$$

2. A calcula

$$r = \alpha^h \pmod{p}.$$

3. A resuelve la congruencia

$$m \equiv ar + hs \pmod{p - 1},$$

siendo a la clave privada de A . La incógnita es s . La congruencia tiene solución porque $\text{mcd}(h, p - 1) = 1$. La firma digital para el mensaje m es el par (r, s) .

Para que B verifique la firma de A , debe hacer lo siguiente:

1. B calcula

$$r^s \cdot (\alpha^a)^r = \alpha^{hs} \cdot \alpha^{ar} \pmod{p}.$$

2. B calcula α^m y comprueba que se verifica

$$\alpha^m = r^s \cdot (\alpha^a)^r \pmod{p}.$$

Ataque a la firma digital

Para falsificar la firma de A en m el criptoanalista debe resolver la ecuación anterior, con incógnitas r y s .

Si fija r , se obtiene una ecuación de logaritmo discreto en s ,

$$s = \log_r \left(\frac{\alpha^m}{(\alpha^a)^r} \right) \pmod{p}.$$

Si fija s , entonces se obtiene una ecuación potencial-logarítmica en r , para la que no hay algoritmo conocido.

2.6.3 Criptoanálisis

El criptoanalista puede conocer $G, \alpha, \alpha^a, \alpha^b, \alpha^\nu$ y $m \cdot \alpha^{b\nu}$, porque G y α son públicos y $\alpha^a, \alpha^b, \alpha^\nu$ y $m \cdot \alpha^{b\nu}$ pueden ser obtenidos en escucha pasiva a través de un canal inseguro. Sin embargo, el método más obvio para conocer m sería calcular $\log_\alpha(\alpha^a)$ ó $\log_\alpha(\alpha^b)$.

Se ha demostrado que la seguridad del criptosistema de ElGamal es equivalente a la del cambio de clave de Diffie-Hellman (véase [72, Note 8.23]); luego la seguridad de aquél está basada también en la dificultad de computación del logaritmo discreto. Lo mismo sucede con el protocolo de firma digital de este criptosistema; para una discusión detallada, véase [72, Note 11.66].

Excepto para los primos del teorema de Pohlig-Hellman, que veremos a continuación, el criptosistema de ElGamal se considera en la actualidad seguro, pues los tiempos de computación del logaritmo discreto son de tipo subexponencial; véase sección 2.1.5 y también [1, 26, 40, 71, 81].

Es de vital importancia utilizar números aleatorios ν diferentes para enviar mensajes diferentes. En efecto, supongamos que A envía dos mensajes m_1 y m_2 a B , cuya clave privada es b , y utiliza el mismo valor ν para ambos envíos. Se tendría entonces $c_1 = m_1 \cdot \alpha^{b\nu} \pmod{p}$ y $c_2 = m_2 \cdot \alpha^{b\nu} \pmod{p}$, luego $c_1/c_2 = m_1/m_2$. Por lo tanto, m_2 puede calcularse inmediatamente si m_1 resulta conocido.

2.6.4 Ataques de Pohlig-Hellman al logaritmo discreto

En el apartado 5 de la sección 2.1.5, hemos visto que el algoritmo de Coppersmith, aplicable a \mathbb{Z}_p^* , corre en tiempo subexponencial. Además, en el año 1992 se obtuvo ya un algoritmo subexponencial para el cálculo de logaritmos en cualquier cuerpo finito; véase [1]. Estos esperanzadores resultados no deben ocultar que, sin embargo, el tiempo de ejecución de estos algoritmos sigue siendo muy considerable en comparación con uno polinómico. En este contexto es especialmente valioso el resultado del trabajo de Pohlig y Hellman, quienes, en 1978, presentaron en [88] un algoritmo para el grupo multiplicativo \mathbb{Z}_p^* de los enteros módulo un número primo p , que permite calcular logaritmos discretos eficientemente si el primo p cumple ciertas condiciones. Posteriormente el algoritmo ha sido extendido a grupos cíclicos cualesquiera (véase [72, sección 3.6.4]). Expliquemos brevemente el resultado.

Teorema 2.22 (Algoritmo de Pohlig-Hellman generalizado) Sea G un grupo cíclico de orden n y sea α un generador. Supóngase conocida la descomposición factorial de $n = p_1^{e_1} \cdots p_r^{e_r}$. Dado un elemento $\beta \in G$, existe un algoritmo que permite calcular $\log_\alpha \beta$ en

$$O\left(\sum_{i=1}^r e_i [\log_2 n + \sqrt{p_i}]\right)$$

operaciones en G .

Corolario 2.23 Si todos los factores primos de n son pequeños (de modo más preciso, n es B -uniforme), entonces el tiempo de ejecución es aproximadamente $\sum_{i=1}^r e_i \cdot \log_2 n$ veces el tiempo de ejecución de una exponencial.

En el caso particular (pero muy importante) en que $G = \mathbb{Z}_p^*$ y todos los factores primos de $p - 1$ son pequeños, se tiene un algoritmo eficiente para calcular logaritmos discretos. Por el contrario, cuando el mayor factor primo de $p - 1$, es comparable en tamaño al propio p , entonces este algoritmo deja de ser de utilidad. El caso extremo de esta situación es aquel en que el mayor factor primo de $p - 1$ es $q = \frac{1}{2}(p - 1)$. Aunque se tratarán con detalle en el capítulo 4, vamos a introducir aquí la definición formal de esta clase de números primos.

Definición 2.24 Un primo p se dice 1-seguro cuando se verifica que $p = 2q + 1$, donde q es también primo.

Queda claro, pues, que, de acuerdo con el Corolario 2.23, si p es 1-seguro el algoritmo de Pohlig-Hellman no es eficiente.

Ejemplo 2.25 Los primos de Fermat $2^{2^k} + 1$ son el paradigma de los primos para los que es aplicable el resultado de Pohlig y Hellman. Otro ejemplo es el primo

$$\begin{aligned} p &= 327992650448307347793135780859993687347740820926286423654401 \\ &= 2^{121} \cdot 5^2 \cdot 7^2 \cdot 11^2 \cdot 13 \cdot 17 \cdot 19 \cdot 23 \cdot 29 \cdot 31 \cdot 37 \cdot 41 \cdot 43 \cdot 47 \cdot 53 \cdot 59 + 1, \end{aligned}$$

que tiene 60 dígitos y, sin embargo, todos los factores primos de $p - 1$ son ≤ 59 .

Ejemplo 2.26 Un número todavía más grande, aunque con cota de uniformidad más alta, es el siguiente:

$$\begin{aligned} p &= 227088231986781039743145181950291021585250524967592855964 \\ &\quad 53269189798311427475159776411276642277139650833937 \\ &= 2^4 \cdot 104729^8 \cdot 224737^8 \cdot 350377^4 + 1, \end{aligned}$$

que tiene 107 dígitos decimales.

2.6.5 Firma digital estándar del NIST

En agosto de 1991, el NIST propuso un nuevo estándar para la firma digital, basado en un sistema que había inventado y patentado por entonces Schnorr (véase [103]) y que básicamente es una variante de la firma digital de ElGamal.

La propuesta levantó muchas críticas, hasta el punto de que la propia agencia norteamericana presentó un documento justificativo en 1992 (véase [108]). Finalmente quedó aprobada como estándar para la transmisión oficial de información no clasificada en 1994 al tiempo que se animaba a empresas y organismos civiles y particulares a utilizarlo. Este estándar se conoce con el nombre de DSS⁷.

El mecanismo de la firma requiere de una función resumen⁸, que pasamos a describir intuitivamente:

Definición 2.27 Se denomina *función resumen* o *hash* $H: \mathcal{M} \rightarrow \mathbb{Z}_q$ a una función que presenta al menos estas propiedades

⁷En inglés, “Digital Signature Standard”.

⁸En inglés, “hash”.

1. H aplica entradas x de longitud arbitraria a salidas $H(x)$ de longitud fija $n = \log_2 q$.
2. Para todo $x \in \mathcal{M}$, $H(x)$ es fácil de calcular.
3. *Resistencia a la preimagen*: para (casi) todo valor $y \in \mathbb{Z}_q$, es computacionalmente imposible averiguar algún x' tal que $y = H(x')$.
4. *Resistencia a la segunda preimagen*: fijado un $x \in \mathcal{M}$, es computacionalmente imposible averiguar $x' \in \mathcal{M}$, tal que $H(x) = H(x')$.
5. *Resistencia a la colisión*: es computacionalmente imposible encontrar dos valores x', x'' , tales que $H(x') = H(x'')$. Nótese que en este caso la elección de x', x'' es libre.

El propio estándar DSS exige que se utilice el resumen SHA-1, basado en el algoritmo SHA, el cual, a su vez, es similar al MD4 (descrito en [99]). Se trata de un algoritmo iterativo basado en una versión débil del método de Merkle-Damgård para funciones resumen; los detalles puedes consultarse en [28, 73].

El estándar DSS incluye solamente una firma digital cuyo protocolo examinaremos a continuación. En primer lugar, cada usuario A elige una colección de parámetros:

1. Un número primo q en el rango $2^{159} < q < 2^{160}$.
2. Un entero t tal que $0 \leq t \leq 8$, y un número primo p en el rango

$$2^{511+64t} < p < 2^{512+64t}$$

con la propiedad de que q divida a $p - 1$.

3. Un generador α del único subgrupo cíclico de \mathbb{Z}_p^* de orden q .
4. Un entero x en el rango $0 < x < q$. Elegido éste, calcula

$$y = \alpha^x \pmod{p}.$$

Con esto, la clave pública de A es el conjunto (p, q, α, y) . Su clave privada es x . En la propuesta inicial, el NIST sugería que los valores p , q y α podrían ser compartidos entre varios o todos los usuarios, reduciendo así la clave pública de A al valor y . El valor t debe ser único para cada firma.

El protocolo de la firma de un mensaje m sigue los siguientes pasos:

1. Se efectúa el resumen del mensaje $H(m)$.
2. Se selecciona un entero aleatorio k con $0 < k < q$.
3. Se calcula

$$r = (\alpha^k \pmod{p}) \pmod{q}.$$

4. Se resuelve la congruencia

$$H(m) \equiv -xr + ks \pmod{q},$$

siendo s la incógnita. Si ocurriera que $s = 0$, se selecciona otro valor de k .

5. La firma digital para el mensaje m es el par (r, s) .

La verificación se efectúa como sigue:

1. Se calcula

$$w = s^{-1} \pmod{q}.$$

2. Se calculan

$$\begin{aligned} u_1 &= H(m)w \pmod{q}, \\ u_2 &= rw \pmod{q}. \end{aligned}$$

3. Se calcula

$$v = (\alpha^{u_1} y^{u_2} \pmod{p}) \pmod{q}.$$

4. Se comprueba que se verifica $v = r$.

La propuesta del NIST levantó una fuerte polémica, especialmente inflamada por el ocultismo con que se desarrollaron las discusiones preliminares, que luego dieron lugar a la propuesta DSS. Vamos a resumir algunas de las ventajas y objeciones más frecuentes de esta propuesta.

Como ventajas, podemos destacar:

1. Al trabajar en un subgrupo de \mathbb{Z}_p^* las firmas tienen longitud menor. Por ejemplo, si $p \approx 2^{512}$, la firma de ElGamal tiene 1024 bits, mientras que la de la DSS sólo tiene 320 bits.
2. La exponenciación

$$r = (\alpha^k \pmod{p}) \pmod{q}$$

puede ser precomputada y no necesita calcularse al generar la firma, lo cual no sucede con el RSA.

3. La generación de claves es un proceso relativamente eficiente.
4. La firma de un mensaje es rápida.

Como principales problemas que se aducen, están:

1. El estándar propuesto sólo cubre la firma digital, pero no permite el intercambio de claves, aspecto básico a la hora de una comunicación segura.
2. La verificación de una firma es un proceso lento comparado con su análogo en RSA, que puede ser hasta 100 veces más rápido.
3. El límite de la subclave q a 160 bits es considerado escaso por algunos oponentes (véase [80]).

2.7 Criptosistemas probabilísticos

2.7.1 Definición de los criptosistemas probabilísticos

Un requisito necesario que cualquier criptosistema ha de cumplir es, obviamente, que sea difícil recuperar el texto en claro (o sea, el mensaje) a partir del texto cifrado. Sin embargo, en ciertas situaciones, puede resultar deseable un mayor nivel de seguridad en forma de requisitos más estrictos.

Todos los criptosistemas que hemos descrito hasta aquí son de tipo *determinista*: fijada una clave pública, cada mensaje m se cifrará siempre en el mismo criptomensaje c . Esto puede presentar ciertos inconvenientes:

1. El sistema no es seguro para la totalidad de mensajes presentes en el espacio de mensajes: en RSA, por ejemplo, el mensaje 0 y el mensaje 1 se cifran en sí mismos, es decir $f_A(m) = m$; véase la Observación 2.19 en sección 2.4.4.
2. Es fácil detectar cuándo el mismo mensaje ha sido enviado por segunda vez.
3. En ciertas circunstancias, es sencillo extraer alguna información parcial del texto claro a partir del criptomensaje. Por ejemplo, en RSA, si suponemos que $c = m^e \pmod{n}$ es el cifrado de m , entonces

$$\left(\frac{c}{n}\right) = \left(\frac{m^e}{n}\right) = \left(\frac{m}{n}\right)^e = \left(\frac{m}{n}\right),$$

puesto que e es impar, lo que permite a un adversario saber un bit de información acerca de m ; a saber, el símbolo de Jacobi $\left(\frac{m}{n}\right)$.

4. Si en un sistema de clave pública determinista, con función de cifrado f , un criptoanalista quiere analizar un criptograma $c = f(m)$, tiene la posibilidad de cifrar cuantos mensajes m' quiera. Podría ocurrir que, seleccionado un mensaje m' , calcule $c' = f(m')$ y se verifique que $c' = c$. En tal caso, habría adivinado el mensaje en claro m ; pero, si no, le basta probar de nuevo. Por lo tanto, en cada cifrado que el criptoanalista realiza, siempre se “filtrá” alguna información acerca del criptosistema y, quizás, acerca de la clave privada que corresponde a la clave pública con la que hace sus ensayos.

Naturalmente esto se salva si especificamos que cierta porción de cada mensaje sea una cadena aleatoria de bits de una longitud prefijada: si ésta es larga, los ataques anteriores quedan, en la práctica, neutralizados. De hecho, para el criptosistema RSA ya existen métodos diseñados para hacerlo, como el presentado en la referencia [10]. Este método ha pasado a considerarse estándar para RSA y recibe el nombre de OAEP⁹. Véase también [78, p. 114] para una explicación más detallada. A pesar de ello, los criptosistemas deterministas a los que se agrega algún tipo de cadena aleatoria no son por lo general probadamente seguros contra cualquier tipo de ataque imaginable.

En este contexto aparecen los sistemas de tipo probabilístico, que sí son capaces de proporcionar un alto nivel de seguridad demostrable.

⁹Acrónimo de *Optimal Asymmetric Encryption Padding*.

Definición 2.28 Un *algoritmo de cifrado probabilístico* es aquel en que a un mensaje en claro le corresponde un número muy grande de criptogramas de los que el algoritmo elige uno aleatoriamente en cada ejecución.

Definición 2.29 Un *criptosistema probabilístico* es el que utiliza un algoritmo de cifrado de tipo probabilístico.

En palabras sencillas, el criptoanalista no está en condiciones de averiguar si a un criptomensaje c le corresponde algún mensaje m aleatoriamente elegido simplemente calculando $c' = f(m)$, y comprobando si $c = c'$: en efecto, por definición, cada cómputo de la función f produce una salida distinta. Intuitivamente, la “seguridad” de este sistema es más alta que en el caso determinista, pero formalicemos un poco este concepto de “seguridad”.

Definición 2.30 Un sistema de clave pública se denomina *polinómicamente seguro* si no es posible seleccionar dos mensajes, cifrarlos y después decidir correc-tamente, usando cualquier algoritmo de tiempo polinómico, qué mensaje cifrado corresponde a qué mensaje claro con probabilidad de acertar significativamente mayor de $\frac{1}{2}$.

Además, también intuitivamente, se ve que con este tipo de sistemas, el criptomensaje no “filtrá” ninguna información computable en tiempo polinómico esperable acerca del mensaje del que procede; es lo que se llama la seguridad semántica, que podemos definir así:

Definición 2.31 Un sistema de clave pública se denomina *semánticamente seguro* si cualquier información que un adversario pueda averiguar en tiempo polinómico acerca de un mensaje conociendo su criptomensaje, la podría también averiguar sin conocerlo, cualquiera que sea la elección aleatoria de mensajes que se utilice.

Con un ejemplo sencillo, un sistema es semánticamente seguro si el atacante sabe que el texto claro toma sólo dos valores, por ejemplo “sí” y “no”, pero no es capaz de averiguar nada a la vista de los criptogramas correspondientes. Es obvio que la seguridad semántica no se obtiene en aquellos sistemas que asignan siempre el mismo criptograma al mismo mensaje en claro.

Observación 2.32 En un sistema probabilístico un criptomensaje siempre es más grande que el mensaje. Esto es una consecuencia directa de que, por definición, a un mensaje le corresponde un número muy grande de criptomensajes.

Observación 2.33 En la teoría de Shannon, un criptosistema posee la propiedad del “secreto perfecto” si un adversario, aun disponiendo de infinitos recursos computacionales, no puede averiguar nada del mensaje a partir del criptomensaje, excepto, como mucho, su longitud. Según Shannon, esto sólo es posible si el número de claves es tan grande como el número de posibles mensajes. Por tanto, la clave ha de ser tan larga como el mensaje mismo y no se debe reutilizar nunca.

Si suponemos que el adversario sólo dispone de recursos acotados polinómicamente, podríamos considerar que un criptosistema semánticamente seguro es una versión “light” de un sistema de secreto perfecto, en el que es razonable utilizar claves más cortas que el mensaje mismo, sin perder sustancialmente la seguridad.

A pesar de las apariencias, las definiciones 2.30 y 2.31 son equivalentes (ver [73, sección 8.49]).

2.7.2 Criptosistema de Goldwasser-Micali

El criptosistema de Goldwasser-Micali fue publicado en [45]. Su seguridad se basa en la hipótesis de inabordabilidad del problema de la residualidad cuadrática.

Definición 2.34 Sea $J_n = \{a \in \mathbb{Z}_n : (\frac{a}{n}) = 1\}$, donde n es impar y $n \geq 3$. El *problema de la residualidad cuadrática* consiste en decidir si, dado $a \in J_n$, a es resto cuadrático módulo n o no lo es.

Observación 2.35 Recientemente se ha introducido (véase [84]) el *problema de la clase de la residualidad compuesta*, que también da origen a aplicaciones en criptografía de clave pública.

Para generar una clave en este criptosistema, un usuario U ha de dar los siguientes pasos:

1. Seleccionar dos primos distintos, grandes y aproximadamente del mismo tamaño, p y q .
2. Calcular $n = p \cdot q$.
3. Seleccionar $y \in \mathbb{Z}_n$ tal que y es un pseudorresto cuadrático módulo n ; es decir, $(\frac{y}{n}) = 1$, pero y no es resto cuadrático módulo n .
4. La clave pública de U es el par (n, y) ; su clave privada es el par (p, q) .

El algoritmo de cifrado sigue el siguiente esquema. Cuando un usuario A quiere mandar un mensaje a B ,

1. A obtiene la clave pública de B , que será el par (n, y) .
2. A reduce su mensaje m a una sucesión de bits, es decir,

$$m = (m_1, m_2, \dots, m_l),$$

con $m_i \in \{0, 1\}$, $1 \leq i \leq l$.

3. Para cada bit m_i del mensaje, A toma aleatoriamente $x \in \mathbb{Z}_n$ y calcula, o bien $c_i \leftarrow yx^2 \pmod{n}$ en caso de que $m_i = 1$, o bien $c_i \leftarrow x^2 \pmod{n}$ en caso contrario.
4. El mensaje cifrado es la l -tupla $c = (c_1, c_2, \dots, c_l)$, que A envía a B .

Una vez en posesión de B , el criptomensaje ha de ser descifrado. Para ello, B dispone de una información privada, a saber, la factorización de $n = p \cdot q$. Con ello, B da los siguientes pasos:

1. Calcula $e_i = (\frac{c_i}{p})$ para cada $i \in [1, l]$.
2. Si $e_i = 1$, entonces $m_i \leftarrow 0$; en caso contrario $m_i \leftarrow 1$.
3. El mensaje descifrado es $m = (m_1, m_2, \dots, m_l)$.

Observación 2.36 El descifrado funciona puesto que $(\frac{c_i}{p}) = 1$ si y sólo si c_i es un resto cuadrático módulo n . Ahora bien, por construcción c_i es resto cuadrático sólo cuando el bit m_i es 0.

Observación 2.37 Un inconveniente de este sistema es que, por construcción, el criptomensaje tiene un tamaño en bits del orden de $\log_2 n$ mayor que el mensaje. Esta expansión era esperable, teniendo en cuenta lo dicho en la Observación 2.32.

2.7.3 Criptosistema de Blum-Goldwasser

Antes de tratar este sistema, es conveniente introducir una serie de nociones previas.

Generadores de sucesiones pseudo-aleatorias

En Criptografía, igual que en otros campos, aparece con frecuencia la necesidad de disponer de secuencias de sucesos aleatorios, como los que se producen con experimentos físicos: lanzar una moneda al aire, lanzar un dado o extraer una carta de la baraja. Es evidente que estos procedimientos no son utilizables en situaciones prácticas y hay que sustituirlos por generadores pseudo-aleatorios, cuya definición más o menos formal presentamos ahora (véase [111]):

Definición 2.38 Sean k, l , dos enteros positivos tales que $l \geq k + 1$ y f es función polinómica de k . Un *generador pseudo-aleatorio* (k, l) es una función $f: (\mathbb{Z}_2)^k \rightarrow (\mathbb{Z}_2)^l$ que puede ser computada en tiempo polinómico en función de k . La entrada $s_0 \in (\mathbb{Z}_2)^k$ es la *semilla* y la salida, $f(s_0) \in (\mathbb{Z}_2)^l$, es una cadena pseudo-aleatoria de bits.

La función f es determinista, es decir, la cadena de bits $f(s_0)$ depende exclusivamente de la semilla s_0 . Intuitivamente, el objetivo consiste precisamente en conseguir que esa cadena pseudo-aleatoria tenga el aspecto de ser aleatoria, si se elige la semilla aleatoriamente. Para objetivar esta descripción, lo que se suele requerir de un candidato a generador pseudo-aleatorio es que las secuencias que genera sean capaces de superar una serie de tests estadísticos pre-establecidos. En concreto se tiene la siguiente

Definición 2.39 Se dice que un generador pseudo-aleatorio pasa todos los *tests estadísticos de tiempo polinómico* si no existe ningún algoritmo de tiempo de ejecución polinómico que pueda distinguir correctamente la salida del generador de otra de igual longitud producida por un generador realmente aleatorio, con probabilidad de acertar significativamente mayor de $\frac{1}{2}$.

Definición 2.40 Sea $s = s_0, s_1, \dots, s_{n-1}$ una secuencia binaria, $s_i \in \{0, 1\}$. Llamamos función de autocorrelación de la secuencia s al valor

$$C(\tau) = \frac{1}{n} \sum_{i=0}^{n-1} (2s_i - 1)(2s_{i+\tau} - 1), \quad 0 \leq \tau \leq n - 1. \quad (2.7)$$

De acuerdo con los postulados de Golomb en [45, Cap. III, sección 1.4], la función de autocorrelación debe ser bivaluada, es decir, debe existir cierto entero k , tal que

$$nC(\tau) = \begin{cases} n, & \text{si } \tau = 0 \\ k, & \text{si } 1 \leq \tau \leq n - 1 \end{cases}$$

Existen una serie de tests estadísticos normalizados, para comprobar la aleatoriedad de la secuencia s , aunque, evidentemente, ninguno puede garantizarla. Entre ellos están los siguientes:

1. *Test de frecuencia.* El propósito de este test es determinar si el número de ceros y de unos en la secuencia es aproximadamente el mismo.
2. *Test de la serie.* Este test trata de determinar si la frecuencia de aparición de las combinaciones de bits 00, 01, 10, 11 son aproximadamente las mismas, como debe esperarse de una secuencia realmente aleatoria.
3. *Test del póker.* El clásico test del póker considera n grupos de cinco enteros sucesivos y los agrupa en cinco categorías:
 - a) Todos diferentes.
 - b) Cuatro diferentes (tenemos una pareja).
 - c) Tres diferentes (tenemos doble pareja o trío).
 - d) Dos diferentes (tenemos “full” o póker).
 - e) Sólo un tipo (tenemos repóker).

Generalizando este test, teniendo en cuenta que nosotros tratamos con bits, supongamos que m es un entero positivo tal que:

$$\left\lfloor \frac{n}{m} \right\rfloor \geq 5 \cdot (2^m)$$

y sea $k = \left\lfloor \frac{n}{m} \right\rfloor$. Divídase la secuencia a comprobar en k subsecuencias que no se solapen, cada una de longitud m . Sea n_i la frecuencia de aparición del i -ésimo tipo de secuencia de longitud m , $1 \leq i \leq 2^m$. Este test determina si cada secuencia de longitud m aparece aproximadamente el mismo número de veces en la secuencia s . Nótese que si $m = 1$, nos encontramos de nuevo con el test de frecuencia.

4. *Test de rachas.* El propósito de este test es determinar si el número de rachas de ceros o de unos de variadas longitudes es el esperable en una secuencia

verdaderamente aleatoria. Para una secuencia de n bits, el número esperable de rachas de ceros o de unos con longitud i es

$$e_i = \frac{n - i + 3}{2^{i+2}}.$$

Se considera que los valores de i que tienen sentido son aquellos que hacen $e_i \geq 5$.

5. *Test de autocorrelación.* Este test trata de comprobar las correlaciones existentes entre la secuencia s y cualquier versión de ella misma desplazada en d bits, con $1 \leq d \leq \lfloor \frac{n}{2} \rfloor$. La función de autocorrelación $C(d)$, definida en la ecuación (2.7) deberá tomar un valor próximo a 0 para todos los valores de d .

Las ideas iniciales sobre estos tests fueron aportadas por Golomb en [45, Cap. III].

Existe otro importante concepto relacionado con las sucesiones aleatorias que es la *complejidad lineal*. Para poder definirlo necesitamos introducir primero un dispositivo denominado *registro de desplazamiento lineal realimentado*, abreviadamente LFSR¹⁰. Podríamos definirlo así:

Definición 2.41 Un *registro de desplazamiento lineal realimentado* (abreviadamente LFSR) de longitud L consiste en L elementos de retardo o etapas, numeradas de 0 a $L - 1$, capaz cada una de albergar un bit y con una entrada y una salida; y un reloj que controla el desplazamiento de los datos. Durante cada avance del reloj, se realizan las siguientes operaciones:

1. Se saca el contenido de la etapa 0 que pasa a formar parte de la sucesión de salida;
2. el contenido de la i -ésima etapa se mueve a la etapa $i - 1$, para cada i , $1 \leq i \leq L - 1$;
3. el contenido de la etapa $L - 1$ es el bit de realimentación s_j que se calcula sumando módulo 2 los contenidos previos de un subconjunto fijado de las etapas $0, 1, \dots, L - 1$.

Si el contenido inicial de la etapa i es $s_i \in \{0, 1\}$ para cada i , $1 \leq i \leq L - 1$, entonces $[s_{L-1}, \dots, s_1, s_0]$ se denomina *estado inicial* del LFSR.

Es interesante notar que, por construcción, si el estado inicial de un LFSR es $[s_{L-1}, \dots, s_1, s_0]$, entonces la sucesión producida en la salida queda únicamente determinada por la relación recursiva

$$s_j = (c_1 s_{j-1} + c_2 s_{j-2} + \dots + c_L s_{j-L}) \pmod{2},$$

para $j \geq L$, donde cada $c_i \in \{0, 1\}$, $1 \leq i \leq L$, representa si la etapa i -ésima contribuye o no al cómputo del bit de realimentación. Ordinariamente a cada

¹⁰Del inglés, "Linear Feedback Shift Register".

LFSR se le asocia el llamado *polinomio de conexión*, $C(D) \in \mathbb{Z}_2[D]$, definido como

$$C(D) = 1 + c_1 D + c_2 D^2 + \dots + c_L D^L.$$

El LFSR se denota entonces como $\langle C(D), L \rangle$. Si el grado de $C(D)$ es L el polinomio se denomina *no singular*. Todo un conjunto de propiedades del LFSR se pueden deducir a partir de las propiedades del polinomio de conexión. Por ejemplo, la sucesión producida por un LFSR es periódica si y sólo si su polinomio de conexión es no singular.

Visto esto, pasamos a dar la siguiente

Definición 2.42 La *complejidad lineal* de una sucesión infinita $s^\infty = s_0, s_1, s_2, \dots$, que se denota por $\mathcal{L}(s^\infty)$, es la longitud del mínimo LFSR que genera s^∞ . Por convenio, se considera que su valor es infinito si no existe ningún LFSR que genere la sucesión y cero para la sucesión $0^\infty = 0, 0, 0, \dots$

Definición 2.43 La complejidad lineal $\mathcal{L}(s^n)$ de la sucesión finita s^n se define como la mínima complejidad lineal de todas las secuencias infinitas cuyos n primeros dígitos coincidan con los de s^n .

Existe un algoritmo debido a Berlekamp y Massey (véase [66]) que permite reconstruir el LFSR que produce una determinada sucesión de bits si se conoce al menos un segmento de la sucesión de tamaño doble de su complejidad lineal. Una vez construido el LFSR, es trivial predecir los sucesivos bits de esa sucesión. Por lo tanto, desde el punto de vista criptográfico, es fundamental que la complejidad lineal sea muy alta. No olvidemos que un período muy grande no implica necesariamente una complejidad lineal grande: es necesario que se cumplan simultáneamente ambas condiciones.

Además de los tests estadísticos, es importante precisar la noción de seguridad criptográfica para las secuencias generadas pseudo-aleatoriamente.

Definición 2.44 Se dice que un generador pseudo-aleatorio de bits pasa el *test del siguiente bit* si no existe ningún algoritmo de tiempo de ejecución polinómico que tras recibir como entrada una secuencia de l bits pueda predecir el $(l+1)$ -ésimo con probabilidad significativamente mayor de $\frac{1}{2}$.

De acuerdo con [73, sección 5.7], las definiciones 2.39 y 2.44 son equivalentes.

Definición 2.45 Todo generador pseudo-aleatorio de bits que pase el test del siguiente bit recibe el nombre de *generador pseudo-aleatorio de bits criptográficamente seguro*.

Otros autores, han introducido también la siguiente noción:

Definición 2.46 (véase [13]) Se dice que un generador pseudo-aleatorio es *im-predecible a la derecha* (resp. *a la izquierda*) en tiempo polinómico si y solo si para todo segmento inicial de una serie producida por él, al que se le ha borrado el elemento de más a la derecha (resp. de más a la izquierda), no existe ningún algoritmo que se ejecute en tiempo polinómico y pueda computar el elemento borrado con probabilidad significativamente mayor de $\frac{1}{2}$.

Es claro que el hecho de ser impredecible a la derecha (o a la izquierda) equivale a pasar el test del siguiente bit.

El generador de números pseudo-aleatorios BBS

El generador de números pseudo-aleatorios BBS recibe este nombre de las iniciales de sus autores, Blum, Blum y Shub, quienes lo publicaron en [13].

Antes de describir este generador, necesitamos primero la siguiente

Definición 2.47 Un entero de Blum es un entero n que se escribe como producto de dos primos distintos, cada uno congruente con 3 módulo 4.

Con esto, el generador BBS se describe como sigue:

Definición 2.48 El generador BBS de números aleatorios consiste en iterar la función cuadrática $x^2 \pmod{n}$ en el conjunto de los restos cuadráticos de los enteros módulo un entero de Blum n . Partiendo de una semilla x_0 , e iterando $x_{i+1} \equiv x_i^2 \pmod{n}$, se obtiene la secuencia binaria $b_i = \text{paridad}(x_i)$.

El generador está basado en el problema de la residualidad cuadrática a que hace referencia la Definición 2.34. La solución de este problema no es más difícil que la de factorizar el módulo n del generador: en efecto, si ésta se puede calcular y conocemos $n = pq$, entonces basta calcular $(\frac{x}{p})$ (o alternativamente $(\frac{x}{q})$). Puesto que, por hipótesis, $(\frac{x}{n}) = 1$, se sigue que x es resto cuadrático si y solo si $(\frac{x}{p}) = 1$ (alternativamente $(\frac{x}{q}) = 1$). Por el contrario, no parece que exista ninguna forma de resolver eficientemente el problema de la residualidad cuadrática si no es conocida la factorización de n , por lo que se puede calificar de inabordable mientras no sea factible el cálculo eficiente de la factorización de n . Por su parte, Rabin en [92] demostró que extraer raíces cuadradas módulo n es polinómicamente equivalente a factorizar.

La sucesión b_i pasa los test estadísticos necesarios para ser considerada una sucesión pseudo-aleatoria. Blum, Blum and Shub probaron en su artículo inicial [13, Teorema 4] que el BBS puede ser considerado un generador pseudo-aleatorio de bits criptográficamente seguro (*cf.* Definición 2.45), suponiendo cierta la inabordabilidad del problema de la residualidad cuadrática.

En [117] apareció un resultado interesante (paradójicamente se publicó antes que [13]) acerca de la seguridad del generador BBS. En concreto, los autores establecieron que, bajo cierta condición, que ellos denominan “condición O-exclusivo”, cualquier generador puede producir $\log_2 \log_2 n$ bits con seguridad criptográfica y el BBS cumple esa condición. Dieron un paso más, estableciendo también (véase [117, Teorema 3]) que adivinar el resultado de la función paridad(x) usada para obtener la secuencia b_i es un problema equivalente a la factorización del módulo n , con lo que parece confirmarse la equivalencia en tiempo polinómico de los problemas de *residualidad cuadrática* y de *factorización* de un entero de Blum. Éste es el resultado demostrado por Alexi *et al.* [2], en su Teorema 3 y Corolario.

Por su parte, en [111, sección 12.3] Stinson expone una secuencia lógica de proposiciones, basadas en la inexistencia de algoritmos de tipo Monte Carlo de

tiempo polinómico, que le permiten inferir también la seguridad criptográfica del generador BBS.

Descripción del criptosistema de Blum-Goldwasser

El generador BBS permitió proponer un nuevo criptosistema de clave pública basado en la dificultad de extraer raíces cuadradas en \mathbb{Z}_n^* que recibe el nombre de criptosistema probabilístico de Blum-Goldwasser ([13, 14]). Es muy eficiente, comparable a RSA, tanto en términos de velocidad como de expansión del mensaje y semánticamente seguro (*cf.* Definición 2.31) suponiendo que el problema de la factorización de enteros sea inabordable. Es, sin embargo, vulnerable al ataque del criptomensaje elegido (véase sección 2.3.1).

El sistema usa el generador de números aleatorios de Blum-Blum-Shub de la siguiente manera: la secuencia de bits que éste genera, elegida alguna semilla aleatoriamente, se suma módulo 2 con el mensaje en claro, produciendo así el criptomensaje que, junto a la semilla cifrada, se transmiten al destinatario. El modo de generar una clave es el siguiente:

1. Cada usuario A elige aleatoriamente dos primos distintos “grandes” p, q tales que

$$p \equiv 3 \pmod{4}, \quad q \equiv 3 \pmod{4}$$

y calcula $n = p \cdot q$. Obsérvese que n es un entero de Blum (ver Definición 2.47).

2. A usa el algoritmo de Euclides extendido para calcular $a, b \in \mathbb{Z}$ tales que:

$$ap + bq = 1.$$

3. La clave pública de A es, entonces, el número n ; su clave privada será el conjunto (p, q, a, b) .

Cuando A quiere enviar un mensaje a B los pasos son los siguientes:

1. A obtiene la clave pública n de B .
2. Sea

$$k = \lfloor \log_2 n \rfloor, \quad h = \lfloor \log_2 k \rfloor$$

y representemos un mensaje m como una sucesión de l cadenas de bits, cada una de longitud h , es decir, $m = (m_1, \dots, m_l)$ donde cada m_i es una sucesión de h bits.

3. A selecciona aleatoriamente un resto cuadrático módulo n . Para ello le basta elegir un número aleatoriamente $r \in \mathbb{Z}_n^*$ y elevarlo al cuadrado, es decir, $x_0 \leftarrow r^2 \pmod{n}$.
4. Para cada una de las l cadenas en que hemos descompuesto el mensaje, A calcula

$$x_i \equiv x_{i-1}^2 \pmod{n}, \quad i = 1, \dots, l.$$

5. Sean p_i los h bits menos significativos de x_i . A asigna $c_i \leftarrow p_i \oplus m_i$, $i = 1, \dots, l$.
6. Ahora, A hace $x_{l+1} \leftarrow x_l^2 \pmod{n}$.
7. A envía a B el texto cifrado junto con la semilla, también cifrada

$$c = (c_1, \dots, c_l, x_{l+1}).$$

Para recobrar el mensaje, B ha de dar los siguientes pasos:

1. B calcula sucesivamente

$$d_1 \equiv \left(\frac{p+1}{4} \right)^{l+1} \pmod{(p-1)}$$

$$d_2 \equiv \left(\frac{q+1}{4} \right)^{l+1} \pmod{(q-1)}$$

$$u \equiv x_{l+1}^{d_1} \pmod{p}$$

$$v \equiv x_{l+1}^{d_2} \pmod{q}$$

$$x_0 \equiv vap + ubq \pmod{n}$$

$$x_i \equiv x_{i-1}^2 \pmod{n}, i = 1, \dots, l.$$

2. Sean p_i los h bits menos significativos de x_i . B calcula

$$m_i = p_i \oplus c_i, i = 1, \dots, l.$$

con lo que se recuperan las l cadenas de h bits cada una en que A había descompuesto su mensaje.

Debemos ahora comprobar que, efectivamente, el mensaje que se recupera es el que se cifró. Para ello basta comprobar que se recupera correctamente la semilla x_0 . Puesto que cada x_i es un resto cuadrático módulo n también lo es módulo p ; en particular, $x_l^{(p-1)/2} \equiv 1 \pmod{p}$. Se verifica que

$$x_{l+1}^{(p+1)/4} \equiv (x_l^2)^{(p+1)/4} \equiv x_l^{(p+1)/2} \equiv x_l^{(p-1)/2} x_l \equiv x_l \pmod{p}$$

Análogamente se tiene que $x_l^{(p+1)/4} \equiv x_{l-1} \pmod{p}$, de donde

$$x_{l+1}^{((p+1)/4)^2} \equiv x_{l-1} \pmod{p}.$$

Repitiendo el argumento, tendremos

$$u \equiv x_{l+1}^{d_1} \equiv x_{l+1}^{((p+1)/4)^{l+1}} \equiv x_0 \pmod{p},$$

y

$$v \equiv x_{l+1}^{d_2} \equiv x_{l+1}^{((q+1)/4)^{l+1}} \equiv x_0 \pmod{q}.$$

Finalmente, puesto que $ap+bq = 1$, $vap+ubq \equiv x_0 \pmod{p}$, $vap+ubq \equiv x_0 \pmod{q}$; por lo tanto, $x_0 \equiv vap + ubq \pmod{n}$. Así B recupera la semilla usada por A y está en condiciones de regenerar el mensaje.

2.7.4 Criptosistema de Blum-Goldwasser mejorado

Aunque, como ya hemos visto, el generador pseudo-aleatorio BBS es criptográficamente seguro e impredecible en tiempo polinómico, presenta un problema práctico que los autores no aclaran en su propuesta [13]. En efecto, puesto que la función $x^2 \pmod{n}$ presenta diferentes órbitas, en función de la semilla inicial que se elija, aparece inmediatamente la necesidad de saber determinar los módulos n que producen precisamente órbitas con períodos predecibles y suficientemente largos, además de las semillas x_0 que permiten obtenerlos. Es imprescindible poder garantizar que no se decae justamente en un ciclo de los “cortos”. Como agudamente apunta Ritter (véase [96]), es difícil sostener la seguridad criptográfica del generador si al mismo tiempo no podemos asegurar al potencial usuario que el módulo o la semilla que ha elegido no le llevarán precisamente a uno de esos ciclos más o menos “degenerados”. Por otro lado, como también señala [96], el criterio aportado en [13, sección 9] para seleccionar la semilla x_0 —elegirla de modo que el orden de x_0 en \mathbb{Z}_n sea $\lambda(n)/2$, donde λ es la función de Carmichael— requiere una cantidad considerable de esfuerzo computacional.

En este contexto, resulta interesantes los resultados obtenidos en [54], trabajo en que se determinan explícitamente tanto las clases de módulos $n = p \cdot q$ como las semillas en \mathbb{Z}_n que producen las órbitas de períodos máximos para la función $x^2 \pmod{n}$; se aporta también una cota superior muy afinada para los períodos de las órbitas de la función cuadrática tanto en \mathbb{Z}_p como en \mathbb{Z}_n . Estos resultados permiten mejorar tanto el criptosistema original propuesto en [13], como el protocolo de firma digital. Veámoslo en las siguientes secciones.

Órbitas cuadráticas en \mathbb{Z}_p^*

Resumimos en esta sección los resultados obtenidos en [54] relativos a las órbitas cuadráticas en \mathbb{Z}_p^* .

Se ha cuestionado en ocasiones (véase, por ejemplo, [75]) si la sucesión b_i también es pseudo-aleatoria cuando se utiliza para generarla un primo arbitrario (secreto), separando así el problema de la seguridad criptográfica del de la dificultad para factorizar el módulo. La pregunta es lógica pues se obtienen algunas ventajas prácticas al usar un primo como módulo: se pueden usar enteros mucho más pequeños (*cf.* [75, p. 196]). No es difícil comprobar que la sucesión b_i , obtenida usando la función $x^2 \pmod{p}$ pasa los tests de la distribución, serie y correlación; también presenta un buen perfil (casi ideal) de complejidad lineal. Aunque la seguridad criptográfica no queda garantizada en este caso, hay algunas cuestiones relativas al generador BBS (en particular, las concernientes al período máximo de las órbitas) que se reducen al caso $x^2 \pmod{p}$. Por esta razón, consideramos en primer lugar el caso de un módulo primo.

Sea $f: X \rightarrow X$ una aplicación definida en un conjunto finito y sea $O(x) = \{f^n(x); n \in \mathbb{N}\}$ la f -órbita de un elemento $x \in X$. Sea $h = h(x)$ el mínimo entero positivo para el cual existe otro entero $k = k(x)$ tal que: 1) $0 \leq k < h$, y 2) $f^k(x) = f^h(x)$. Nótese que el par (h, k) que satisface ambas condiciones es único. Los elementos $x = f^0(x), f^1(x), \dots, f^{k-1}(x)$ se llaman *cola* de la órbita, mientras que los elementos $f^k(x), \dots, f^{h-1}(x)$ constituyen el *ciclo* de la órbita. El entero

$l(x) = h - k$ es la *longitud* o *periodo* del ciclo (cf. [80, XII]). Para una función $f: \mathbb{Z}_n \rightarrow \mathbb{Z}_n$, se denotará por $\pi_n(x)$ la longitud del ciclo de la f -órbita de un entero $x \pmod n$.

En la práctica, es importante que la semilla $x = f^0(x)$ sea elegida aleatoriamente, por lo que es necesario asegurarse de que la mayoría de las semillas producen órbitas largas, y por ende, colas cortas, tan cortas como sea posible.

En esta sección, salvo indicación en contra, se hablará siempre de las órbitas de la función cuadrática $f(x) = x^2$ módulo un primo p . Se denotará $v_p(n)$ el exponente de la potencia más alta del primo p que divide a n .

Los principales resultados en torno a las f -órbitas son los siguientes.

Proposición 2.49 (véase [54]) Con las hipótesis y notaciones de más arriba se tiene

1. Si p es impar, la longitud de la cola de $O(x)$ es, como mínimo, $v_2(r)$, donde r es el orden de x en \mathbb{Z}_p^* . La cola de $O(x)$ es maximal si y solo si x no es resto cuadrático y, entonces, $k(x) = v_2(p - 1)$.
2. Si $p \equiv 3 \pmod 4$, entonces
 - (a) Para todo $x \in \mathbb{Z}_p^*, x \neq 1$, $\pi_p(x)$ coincide con el orden de 2 en \mathbb{Z}_r^* , donde r es el orden de x o de x^2 en \mathbb{Z}_p^* , según que x sea un resto cuadrático o no lo sea.
 - (b) Para todo $x \in \mathbb{Z}_p^*$, $\pi_p(x) \leq \frac{1}{2}(p - 3)$.
 - (c) Si x es un resto cuadrático, entonces la cola de $O(x)$ está vacía; si no lo es, entonces la cola de $O(x)$ es $\{x\}$.
3. Sea $p = 2p' + 1$ un primo 1-seguro.
 - (a) Si 2 es un generador de $\mathbb{Z}_{p'}^*$, entonces todo resto cuadrático $x \in \mathbb{Z}_p^*, x \neq 1$, produce una órbita de longitud máxima, $l = \frac{1}{2}(p - 3)$.
 - (b) Si -2 es un generador de $\mathbb{Z}_{p'}^*$, pero 2 no es generador de $\mathbb{Z}_{p'}^*$, entonces $p' \equiv 3 \pmod 4$ y todo resto cuadrático $x \in \mathbb{Z}_p^*, x \neq 1$, produce una órbita de longitud $l = \frac{1}{4}(p - 3)$.
 - (c) Además, si $p' > 3$ es también seguro (esto es, p es 2-seguro), entonces 2 es un generador de $\mathbb{Z}_{p'}^*$ si y solo si $\frac{1}{4}(p' + 1) = \frac{1}{8}(p + 1)$ es impar.

Órbitas cuadráticas en \mathbb{Z}_n

Consideramos ahora el caso de un módulo compuesto, $n = p \cdot q$. Primero se analiza el problema de caracterizar los módulos n que producen las órbitas de máxima longitud. Después se pasa al de la elección acertada de las semillas que dan lugar precisamente a esas órbitas de periodo máximo.

Proposición 2.50 (véase [54, 86]) Sean $p = 2p' + 1$, $q = 2q' + 1$ dos primos 1-seguros. Hagamos $n = p \cdot q$. Existe una órbita de la función $x^2 \pmod n$ cuyo ciclo tiene un periodo máximo igual a $\frac{1}{8}(p - 3)(q - 3)$, si y solo si las dos condiciones siguientes se satisfacen:

1. O bien 2 es un generador de $\mathbb{Z}_{p'}^*$, y o bien 2 o -2 es generador de $\mathbb{Z}_{q'}^*$; o 2 es generador de $\mathbb{Z}_{q'}^*$, y o bien 2 o -2 es generador de $\mathbb{Z}_{p'}^*$.
2. $\text{mcd}(p' - 1, q' - 1) = 2$.

Estas condiciones que determinan las máximas longitudes de los ciclos son más sencillas de comprobar que las impuestas en [13, Theorem 9]. En particular, se satisfacen si p y q son ambos primos 2-seguros, con $p, q > 11$ y o bien $p' \equiv 3 \pmod{8}$ o $q' \equiv 3 \pmod{8}$.

Proposición 2.51 Con idénticas hipótesis y condiciones que en la Proposición 2.50, toda semilla $x \not\equiv 1, -1, 0 \pmod{p}$, $x \not\equiv 1, -1, 0 \pmod{q}$, da una órbita de periodo máximo. Por tanto, la probabilidad de encontrar una semilla que proporcione un periodo máximo es mayor o igual que $1 - 3(p^{-1} + q^{-1})$. Además,

1. Si $x \equiv 1, -1, 0 \pmod{p}$ y $x \not\equiv 1, -1, 0 \pmod{q}$ (resp. $x \equiv 1, -1, 0 \pmod{q}$ y $x \not\equiv 1, -1, 0 \pmod{p}$), entonces
 - (a) Si 2 es un generador de $\mathbb{Z}_{q'}^*$ (resp. 2 es un generador de $\mathbb{Z}_{p'}^*$), entonces $\pi_n(x) = \frac{1}{2}(q - 3)$ (resp. $\pi_n(x) = \frac{1}{2}(p - 3)$).
 - (b) Si -2 es un generador de $\mathbb{Z}_{q'}^*$ pero 2 no es un generador de $\mathbb{Z}_{q'}^*$ (resp. -2 es un generador de $\mathbb{Z}_{p'}^*$ pero 2 no es un generador de $\mathbb{Z}_{p'}^*$), entonces $\pi_n(x) = \frac{1}{4}(q - 3)$ (resp. $\pi_n(x) = \frac{1}{4}(p - 3)$).
2. Finalmente, si $x \equiv 1, -1, 0 \pmod{p}$ y $x \equiv 1, -1, 0 \pmod{q}$ entonces $\pi_n(x) = 1$.

Descripción del criptosistema

Con los resultados anteriores a la vista, estamos en condiciones de proponer un criptosistema de clave pública tipo BBS y un protocolo de firma digital. Este criptosistema mejora el esquema de Blum-Goldwasser (propuesto en [14]), pues, a partir de la proposición 2.51, se puede dar un rango explícito en el que el remitente puede elegir aleatoriamente su semilla con la seguridad de que va a generar un ciclo de longitud máxima. Recordemos que el único criterio de elección de semilla que se proporcionaba en [13] implica calcular el orden de la semilla módulo n , cálculo que requiere un tiempo de computación muy considerable. Por otro lado, la caracterización de los módulos n garantiza longitudes de ciclo maximales para la órbita de la función $x^2 \pmod{n}$.

Supongamos que el usuario B tiene como clave pública $n = p \cdot q$, donde n satisface las condiciones de la Proposición 2.50, y guarda en secreto su clave privada (p, q) . Un mensaje es un sucesión de k bits, con $k < (p - 3)(q - 3)/8$. En una implementación real, en donde p y q tengan del orden de 500 bits, la longitud del mensaje es mucho más pequeña que la longitud de las órbitas. El protocolo por el que un usuario A puede enviar un mensaje de k bits $m = (m_1, \dots, m_k)$ al usuario B es como sigue:

1. A selecciona aleatoriamente un entero $x_0 \in \mathbb{Z}_n$ que proporcione una órbita de longitud máxima (cf. Proposición 2.51) y lo eleva al cuadrado, para obtener un resto cuadrático x_1 ;

2. A computa la secuencia x_1, \dots, x_k , iterando la función $x^2 \pmod{n}$, y de ella la secuencia $b = (b_1, \dots, b_k)$, donde $b_i = \text{paridad}(x_i)$;
3. A cifra el mensaje m , calculando

$$m \oplus b = (m_1 \oplus b_1, \dots, m_k \oplus b_k) = c;$$

4. A envía a B el mensaje cifrado c junto con x_{k+1} .

Por sencillez, se ha considerado que solamente se extrae un bit, el menos significativo, de la secuencia producida por el generador $x^2 \pmod{n}$; sin embargo, tal como dijimos antes, a partir de los resultados reportados en [117], se podría extraer hasta $\log_2 \log_2 n$ bits menos significativos.

Para asegurar que la semilla x_0 está realmente en una órbita máxima es suficiente forzar que $x_0 \geq 2$ y que la longitud en bits de x_0 es menor que ν , donde ν es un cierto número menor que la longitud en bits de p y q , como se deduce de la Proposición 2.51. Esta limitación debe estar incluida en las especificaciones de la clave pública del criptosistema.

Para recuperar el mensaje original, B ha de calcular la semilla x_1 y así generar la secuencia b . Para ello, B da los siguientes pasos:

1. B recobra x_1 calculando

$$x_{k+1}^{2(\pi_n(x_0)-k)} \equiv (x_1^{2^k})^{2(\pi_n(x_0)-k)} \equiv x_1^{2\pi_n(x_0)} \equiv x_1 \pmod{n},$$

- donde $\pi_n(x_0) = (p-3)(q-3)/8$ es la longitud del ciclo de $x_0^2 \pmod{n}$;
2. B reconstruye la secuencia $b = (b_1, \dots, b_k)$ previamente usada por A;
 3. B recupera el mensaje original calculando simplemente

$$c \oplus b = (c_1 \oplus b_1, \dots, c_k \oplus b_k) = (m_1, \dots, m_k) = m.$$

El criptosistema presentado admite una firma digital que podría considerarse generalización del esquema de firma digital de Rabin (véase [72, sección 11.3.4]). Sea $H(m)$ un resumen del mensaje m que suponemos, módulo n contenido en la órbita de longitud máxima. Sean n y (p, q) las claves pública y privada de A, respectivamente. A firma el mensaje m mediante el siguiente protocolo:

1. A calcula el cuadrado del resumen: $M \equiv H(m)^2 \pmod{n}$,
2. A determina su firma digital, s , para el mensaje M calculando

$$s \equiv \sqrt[2^h]{M} \pmod{n} \equiv M^{2(\pi_n(x)-h)(\text{mod } \frac{(p-1)(q-1)}{2})} \pmod{n},$$

- donde $2 \leq h \leq \pi_n(x)$ es un número arbitrario y prefijado (es decir, un parámetro del sistema) y $\pi_n(x)$ es la longitud de del ciclo de $x^2 \pmod{n}$.
3. A envía a B el mensaje m y su firma s .

Para verificar la firma digital, B da los siguientes pasos:

1. B calcula $s^{2^h} \pmod{n}$ y obtiene

$$s^{2^h} \equiv \left(\sqrt[2^h]{M}\right)^{2^h} \equiv M \pmod{n},$$

2. B verifica que $M \equiv H(m)^2 \pmod{n}$.

2.8 Clave secreta versus clave pública

En esta sección queremos enumerar algunas de las ventajas e inconvenientes de los criptosistemas de clave secreta y de clave pública comúnmente aceptados por los especialistas (*cf.* [72]), sin pretender agotar el elenco.

Si bien la criptografía de clave pública ha resuelto algunos de los inconvenientes tradicionales de la clave secreta, no es cierto en modo alguno que la haya desterrado, especialmente debido a su lentitud de operación para transmitir datos masivos. En la práctica coexisten ambos sistemas, utilizándose uno u otro según las necesidades o prioridades del usuario por lo que, más que antagónicos, pueden considerarse complementarios.

2.8.1 Ventajas de la clave secreta

1. Los sistemas de clave secreta gozan de una alta velocidad de operación, especialmente cuando se implementa en hardware. Por ejemplo, ya en 1993 se había desarrollado un circuito integrado que permitía cifrar usando DES a una velocidad de 1 Gbit por segundo (véase [38]). En software, se puede llegar a unos pocos megabytes por segundo.
2. Las claves secretas son relativamente cortas: están entre 112 y 256 bits. El actual estándar de clave secreta, el AES¹¹, cifra bloques de 128 bits usando claves secretas de 128, 192 ó 256 bits.
3. Los cifradores de clave secreta se pueden usar también como elementos básicos para construir otros dispositivos criptográficos, tales como generadores de números pseudoaleatorios, funciones resumen, etc.
4. Con cifradores de clave secreta se pueden construir sistemas más robustos: con transformaciones simples, fáciles de analizar y débiles en sí mismas, se pueden sin embargo fabricar otros más seguros.

2.8.2 Desventajas de la clave secreta

1. En las comunicaciones bilaterales entre dos usuarios la clave secreta debe ser compartida por ambos y permanecer secreta. Ello tiene en particular los siguientes inconvenientes:

¹¹Acrónimo de *Advanced Encryption Standard*.

- (i) Cuando se cambian las claves hay que volverlas a distribuir, desplazándose a un lugar común para tal efecto, o bien enviándolas por medio de un canal inseguro. Además, la práctica criptográfica dicta que las claves secretas se cambien frecuentemente, incluso en cada sesión.
 - (ii) Si un usuario quiere convencer a un tercero (por ejemplo, a un juez) de la autenticidad de un mensaje recibido no tiene más remedio que compartir con él la clave del remitente.
2. Si la red de usuarios es grande también lo es el número de pares de claves a gestionar. De hecho, el número de claves aumenta proporcionalmente al cuadrado del número de usuarios, lo cual da idea de su complejidad de manejo y almacenamiento.
 3. Los mecanismos de firma digital necesarios para el cifrado secreto requieren el uso o bien de largas claves para la función de verificación o bien un TTP¹².

2.8.3 Ventajas de la clave pública

1. La clave privada es puramente tal y, por lo tanto, no necesita ser distribuida a nadie. Basta que el usuario la mantenga en secreto para sí.
2. La gestión de claves en una red requiere únicamente la presencia de un TTP con *confianza limitada* (es decir, de una entidad que se supone honesta, pero sin acceso a la clave privada de los usuarios de la red).
3. Dependiendo del modo de uso, un par (clave privada, clave pública) se puede mantener durante un largo periodo de tiempo.
4. Bastantes esquemas de clave pública proporcionan mecanismos de firma digital relativamente eficientes. La clave que se necesita para la función de verificación pública es típicamente mucho más pequeña que su equivalente en un esquema de clave secreta. Por ejemplo, en el esquema uniuso de Rabin pueden necesitarse claves del orden de miles de bytes; para el esquema de Merkle el tamaño típico puede ser muchos cientos de bytes. Sin embargo, en los esquemas de clave pública, las claves son del orden de, como mucho, decenas o cientos de bytes.
5. En una red grande se puede reducir sustancialmente —en relación con los sistemas de clave secreta— el número de claves necesarias.

2.8.4 Desventajas de la clave pública

1. La velocidad de operación de los sistemas de clave pública es muy inferior a los de clave secreta. Como se ha dicho antes, existen implementaciones del sistema DES en hardware que permiten velocidades nominales de operación del orden de gigabits por segundo.

¹²En inglés “Trusted Third Party”, un tercero de confianza.

2. Los tamaños de las claves son usualmente mucho más grandes que los requeridos en clave secreta. Por ejemplo, el tamaño mínimo recomendado para el módulo de RSA está en estos momentos alrededor de 1024 bits, para un uso de baja seguridad. Para las autoridades de certificación, que exigen un nivel de seguridad más alto, lo recomendado es utilizar un módulo de al menos $1024 + 1024 = 2048$ bits (véase [62]). Por contraste, en el AES el tamaño de la clave es un orden de magnitud menor.
3. Los tamaños de las firmas digitales en clave pública son más grandes que las etiquetas¹³ que suministran la autenticación de origen con las técnicas de clave secreta.
4. No se ha demostrado “matemáticamente” que haya un sistema de clave pública criptográficamente seguro. Nótese que algo parecido puede afirmarse del cifrado en bloque. De hecho, los sistemas de clave pública más efectivos que se han encontrado hasta la fecha basan su seguridad en la “supuesta dificultad” de un pequeño conjunto de problemas de teoría de números algunos de los cuales (básicamente el logaritmo discreto y la factorización de números enteros) han sido comentados anteriormente en este capítulo.

La clave pública y la privada tienen un número de ventajas complementarias que los criptógrafos actuales tratan de explotar. Un ejemplo interesante de uso complementario es el llamado protocolo de la *envoltura o sobre digital*, que funciona como sigue. Supongamos que el usuario *A* desea establecer una sesión de comunicación con el usuario *B* utilizando criptografía de clave secreta. El protocolo consiste, entonces, en que *A* genera aleatoriamente una clave secreta para el sistema que *A* y *B* hayan acordado utilizar, por ejemplo, el criptosistema AES. A continuación, *A* obtiene la clave pública de *B* y cifra con ella esa clave secreta, para transmitírsela después a *B*. Recibido el criptograma, *B* utiliza su propia clave privada para descifrarla. A partir de ese momento, *A* y *B* están compartiendo una clave secreta que puede ser usada para cifrar el resto de la sesión mediante, en este ejemplo, el criptosistema AES. Esta clave secreta, así compartida, se denomina “clave de sesión”.

La ventaja de este protocolo, comúnmente usado en la actualidad, es que permite la transmisión segura —e, incluso, firmada— de la clave secreta entre las partes y disfrutar al mismo tiempo de la mayor velocidad de operación de los sistemas de clave secreta: puesto que el cifrado de datos es la parte que más tiempo importa habitualmente en el proceso de comunicación, usar el esquema de clave pública para compartir la clave secreta supone tan sólo una pequeña fracción del tiempo total del proceso.

¹³En inglés, “tags”.

Capítulo 3

Tests de primalidad y otros algoritmos empleados

Resumen del capítulo

Se describe aquí una colección de algoritmos que son de interés, alguno de los cuales será utilizado después en la memoria. En primer lugar se tratan los algoritmos de comprobación de primalidad. Después se explican los principales algoritmos de factorización. Se introducen también varios algoritmos para generar números aleatorios. Finalmente se presenta también un algoritmo para generar primos aleatorios.

3.1 Introducción explicativa

Dedicamos este breve capítulo a la exposición de un conjunto de algoritmos que, si bien son de dominio público, son necesarios para entender ciertas definiciones y sirven también como herramienta para otros algoritmos que se usan en diversas partes de la memoria.

Sin embargo, los algoritmos que hacen referencia directa a los resultados principales de la memoria se han colocado en las secciones correspondientes. De entre éstos, algunos son originales y otros se han implementado a partir de las referencias.

Concretamente, los algoritmos originales son:

1. Algoritmo 4.41: generación de primos 1-seguros.
2. Algoritmo 4.42: generación de primos 2-seguros.
3. Algoritmo 5.33: generación de primos robustos óptimos.

En cuanto a los algoritmos implementados a partir de las referencias, tenemos:

1. Algoritmo 5.24 de Gordon: generación de primos robustos de “3 vías”.
2. Algoritmo 5.30 de Ogiwara: generación de primos robustos de “6 vías” (véase también la sección 5.1.2 para una explicación detallada de este tipo de primos).

3.2 Tests de primalidad

3.2.1 La noción de test de primalidad

Uno de los intereses principales en teoría de números es poder determinar si un entero n es primo o bien es compuesto. Parece que lo más obvio sería intentar factorizarlo: si se consigue, evidentemente el número es compuesto; si, por el contrario, se llega a demostrar que la factorización es imposible, es primo. Como esta tarea en la práctica suele ser inabordable, se han desarrollado otros métodos que permiten establecer la *primalidad* de un número sin necesidad de factorizarlo.

Aun así, los métodos de comprobación de primalidad están lejos de ser sencillos. Ordinariamente se basan en comprobar el cumplimiento, por parte del candidato a primo, de ciertas condiciones: si las cumple, entonces es primo; de lo contrario, es compuesto. Sin embargo tal comprobación exige con frecuencia mucho esfuerzo si se quiere establecer con todo rigor la primalidad del candidato. Por ello, en la práctica suele recurrir a métodos que exigen comprobaciones menos costosas, pero que, en contrapartida, no dan con toda seguridad una respuesta correcta: hay circunstancias en que fallan, es decir, declaran que un número es primo cuando en realidad es compuesto. Esto motiva las siguientes definiciones:

Definición 3.1 Se llama *test de primalidad* a un algoritmo que determina que un número candidato es primo basándose en el cumplimiento de ciertas propiedades por parte del candidato.

Definición 3.2 Se llama *test de composición* a un algoritmo que determina que un número candidato es compuesto basándose en el cumplimiento o incumplimiento de ciertas propiedades por parte del candidato.

En otras palabras, un test de primalidad decide con total seguridad acerca de la primalidad del candidato. Sin embargo, el test de composición sólo es capaz de determinar con toda seguridad que el candidato es compuesto. Por esta razón, los tests de primalidad se conocen también con el nombre de *tests deterministas*, mientras que los tests de composición se conocen con el nombre de *tests de primalidad probabilísticos*.

Como se dijo antes, los tests deterministas exigen de ordinario una mayor cantidad de recursos computacionales que los probabilísticos. Cuál de ellos deba ser usado en cada caso concreto dependerá de las necesidades y recursos disponibles.

Para comenzar, presentamos el algoritmo probabilístico más utilizado.

3.2.2 El algoritmo de Miller-Rabin

El teorema de Fermat es la base de los algoritmos probabilísticos de comprobación de primalidad. Recordemos que este teorema afirma que si p es primo y $\text{mcd}(p, b) = 1$, entonces

$$b^{p-1} \equiv 1 \pmod{p}.$$

Si queremos comprobar si un número p es primo y encontramos una base prima con p en que el teorema de Fermat no se verifica, podemos estar seguros de que p es

compuesto. Ahora bien, la recíproca no es cierta: el hecho de que no encontremos ninguna base en que no se verifique el teorema de Fermat no garantiza que el número p que probamos sea primo.

En todo caso, desde un punto de vista computacional, el cálculo de la exponencial modular es rápido, usando los algoritmos específicos para ello (véase, por ejemplo, [25, §1.2]). Además es interesante destacar que, aunque la condición que nos proporciona sea sólo necesaria, el número de excepciones, conocidas en general como *pseudoprimos*, es muy pequeño. De hecho, los números n que satisfacen la congruencia $b^{n-1} \equiv 1 \pmod{n}$ para toda base $b \in [2, n-1]$ tal que $\text{mcd}(b, p) = 1$ reciben el nombre de números de Carmichael. El más pequeño de ellos es $n = 3 \cdot 11 \cdot 17 = 561$.

Estas consideraciones abren el camino a uno de los tests probabilísticos de primalidad más usados: el test de Miller-Rabin. Antes de explicarlo, vamos a dar la siguiente

Definición 3.3 Sea n un número entero impar positivo y sea a otro entero. Escribamos $n-1 = 2^s q$, con q otro entero impar. En estas condiciones, decimos que n es un pseudoprimo robusto en la base a si o bien $a^q \equiv 1 \pmod{n}$ o bien existe un e tal que $0 \leq e < s$ y $a^{2^e q} \equiv -1 \pmod{n}$.

Observación 3.4 Si p es un primo impar, es fácil ver que también p es un pseudoprimo robusto en cualquier base b tal que $\text{mcd}(b, p) = 1$. Recíprocamente, se puede probar (véase, por ejemplo, [56]) que si p no es primo, existen menos de $p/4$ bases b tales que $1 < b < p$ para las cuales p es un pseudoprimo robusto en la base b .

Con esto, Miller (véase [76]) y Rabin (véase [92]) desarrollaron el test que describimos a continuación.

Descripción del test

Algoritmo 3.5 Dados $n, t \in \mathbb{N}$, este algoritmo determina si n es primo con una probabilidad de acierto de $1 - 2^{-2t}$.

LLAMADA: `TestMillerRabin(n, t);`

ENTRADA: Un entero n y un parámetro de seguridad $t \in \mathbb{N}$.

SALIDA: n es primo con probabilidad de acierto $1 - 2^{-2t}$.

1. [Inicialización]

Se eligen $s, q \in \mathbb{Z}$ de tal modo que $n-1 = 2^s q$ con q impar.

2. [Lazo]

`while (t > 0)`

`{`

Elegimos un entero $a \in [2, n-1]$ aleatoriamente

`b = aq % n;`

`if (b == 1 o bien b == n-1) goto seguir;`

```

for ( $j \in [1, s - 1]$ )
{
     $b = b^2 \pmod n$ ;
    if ( $b == n - 1$ ) goto seguir;
    if ( $b == 1$ ) return " $n$  es compuesto con toda seguridad";
}
return " $n$  es compuesto con toda seguridad".

seguir:
 $t = t - 1$ ;
}
return " $n$  es primo con probabilidad  $1 - 2^{-2t}$ ".
```

Tiempo de ejecución del test de Miller-Rabin

Proposición 3.6 El tiempo de ejecución esperado para el algoritmo de Miller-Rabin es $O((\log_2 n)^3)$.

Demostración El tiempo es esencialmente el mismo que el del algoritmo de exponentiación empleado, es decir, $O((\log_2 n)^3)$. ■

Observación 3.7 Experimentalmente se comprueba, sin embargo, que el tiempo de ejecución de este algoritmo es muy distinto dependiendo de si la entrada resulta ser de hecho un número primo o no. Cuando la entrada n es un primo el tiempo de ejecución es el predicho teóricamente; en caso contrario, el tiempo de ejecución es $O((\log_2 n)^2)$.

Ello se pone de manifiesto en las gráficas de las figuras 3.1 y 3.2, en donde hemos representado el tiempo de ejecución frente al número de bits de n para los casos en que n es primo y n es compuesto, junto con las funciones $\alpha \cdot (\log_2 n)^3$, y $\beta \cdot (\log_2 n)^2$ respectivamente, y α y β son ciertas constantes apropiadas.

Comparación con el test de Solovay-Strassen

El test de Solovay-Strassen fue publicado en las referencias [109, 110], aunque también aparece descrito en otros lugares como, por ejemplo, en [25, sección 8.2]. Este test ha quedado superado por el de Miller-Rabin, por las siguientes razones:

1. El test de Solovay-Strassen es más costoso en tiempo de computación porque requiere el cálculo del símbolo de Jacobi, mientras que el de Miller-Rabin no.
2. El test de Solovay-Strassen es más difícil de implementar que el de Miller-Rabin.
3. El número de bases para las que potencialmente puede fallar el test son, como mucho, $n/2$. En cambio, para el test de Miller-Rabin tales bases son, como mucho, $n/4$.

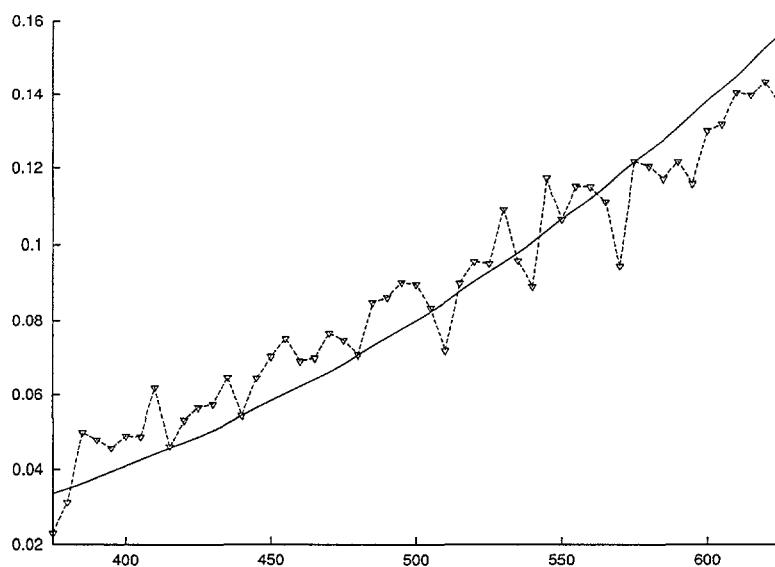


Figura 3.1: Tiempo de ejecución para n primo

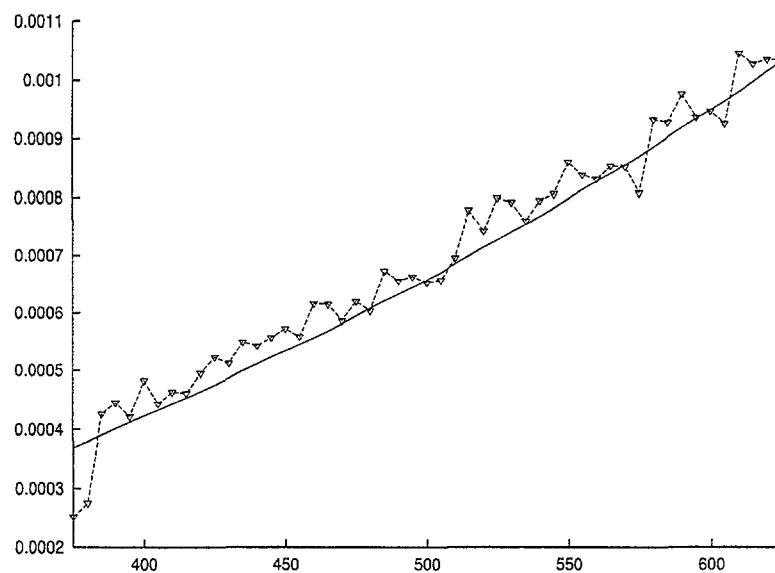


Figura 3.2: Tiempo de ejecución para n compuesto

3.2.3 Algoritmo de Miller-Rabin modificado

Algoritmo 3.8 Dados $n, t \in \mathbb{N}$, este algoritmo determina si n es primo con probabilidad $1 - 2^{-2t}$.

LLAMADA: `MillerRabin(n, t);`

ENTRADA: Un entero n y un parámetro de seguridad $t \in \mathbb{N}$.

SALIDA: n es primo con probabilidad de acierto $1 - 2^{-2t}$.

1. [Inicialización]

Se eligen $s, q \in \mathbb{Z}$ de tal modo que $n - 1 = 2^s q$ con q impar.

2. [Ensayo de divisiones]

Dividimos n por todos los primos menores que 256. Si obtenemos división exacta en algún caso, devolvemos “ n es compuesto con toda seguridad”.

3. [Ensayo de Fermat]

Usando una base $b = 2 \cdot 3 \cdot 5 \cdot 7$ fija, ensayamos por el teorema de Fermat, pues es barato computacionalmente; esto es,

```
if ( $b^{n-1} \pmod n \neq 1$ )
    return "n es compuesto con toda seguridad"
```

Observemos que, por construcción, se cumple que $\text{mcd}(b, n) = 1$; de lo contrario, el algoritmo se habría detenido en el paso 1.

4. [Algoritmo de Miller-Rabin]

```
resultado = TestMillerRabin(n, t);
if (resultado == NO)
    return "n es compuesto con toda seguridad"
else
    return "n es primo con probabilidad de acierto de  $1 - 2^{-2t}$ .
```

Tiempo de ejecución del algoritmo de Miller-Rabin modificado

Proposición 3.9 El tiempo de ejecución esperado para el algoritmo de Miller-Rabin modificado es $O((\log_2 n)^3)$.

Demostración El tiempo es esencialmente el mismo que el del algoritmo de Miller-Rabin en que se apoya. ■

Observación 3.10 No obstante es interesante observar que, a efectos prácticos,

la división por los primos menores de 256,

$$\begin{aligned} B &= \{p \leq 256\} \\ &= \{2, 3, 5, 7, 11, 13, 17, 19, \\ &\quad 23, 29, 31, 37, 41, 43, 47, 53, \\ &\quad 59, 61, 67, 71, 73, 79, 83, 89, 97, \\ &\quad 101, 103, 107, 109, 113, 127, 131, \\ &\quad 137, 139, 149, 151, 157, 163, 167, \\ &\quad 173, 179, 181, 191, 193, 197, 199, \\ &\quad 211, 223, 227, 229, 233, 239, 241, 251\} \end{aligned}$$

elimina, en promedio, aproximadamente el 90 %, lo que supone un ahorro práctico muy importante. En efecto, la probabilidad de que un entero no sea divisible por alguno de los primos de B es

$$P = \prod_{p < 256} \left(1 - \frac{1}{p}\right),$$

que vale

$$P = 0,1003532963,$$

como demuestra un cálculo inmediato. Este resultado no modifica el tiempo de computación de la Proposición 3.9, pero reduce en un orden de magnitud el número de veces que el algoritmo básico de Miller-Rabin 3.5 ha de ser ejecutado.

3.2.4 El test de Pocklington-Lehmer

Pasamos ahora a considerar el problema práctico de probar rigurosamente que un número n es primo. Naturalmente, sólo tiene sentido acometer esta prueba cuando haya una seguridad moral de que el candidato es primo; esto puede conseguirse usando el test de Miller-Rabin.

La siguiente Proposición nos servirá para desarrollar uno de los métodos más conocidos: el test Pocklington-Lehmer.

Proposición 3.11 Sea $n \geq 3$ un entero. Entonces n es primo si y sólo si existe un entero b que satisface

$$\begin{aligned} b^{n-1} &\equiv 1 \pmod{n}, \\ b^{\frac{(n-1)}{q}} &\not\equiv 1 \pmod{n} \end{aligned}$$

para cada divisor primo q de $n - 1$.

Demostración Se deduce inmediatamente del hecho de que \mathbb{Z}_n^* contiene un elemento de orden $n - 1$ si y sólo si n es primo. ■

Naturalmente, el inconveniente es que necesitamos conocer la factorización completa de $n - 1$ para poder aplicar este método. Pocklington (véase [87]) demostró un resultado que permite aprovechar una factorización parcial de $n - 1$, descrita por Brillhart *et al.* (véase [18]). El resultado de Pocklington es el siguiente:

Proposición 3.12 Sea n un entero positivo y sea p un divisor primo de $n - 1$. Supóngase que puede encontrarse un entero a_p tal que

$$a_p^{n-1} \equiv 1 \pmod{n}$$

y, además,

$$\text{mcd}(a_p^{\frac{n-1}{p}} - 1, n) = 1.$$

Entonces, si d es cualquier divisor de n , se tiene

$$d \equiv 1 \pmod{p^{\alpha_p}},$$

donde p^{α_p} es la mayor potencia de p que divide a $n - 1$.

Una demostración de esta Proposición puede verse en [25, Proposition 8.3.1] o en [94, p. 33].

La idea de Lehmer es entonces la siguiente: supongamos ahora que se puede escribir $n - 1 = F \cdot U$, donde F y U son primos entre sí, F está factorizado completamente y $F > \sqrt{n}$. Si para cada primo p que divide a F se puede encontrar un a_p que satisface las condiciones de la Proposición 3.12, entonces n es primo. Recíprocamente, si n es primo, entonces para cualquier factor primo p de $n - 1$ se puede encontrar un a_p que satisface las condiciones de la Proposición 3.12.

Observación 3.13 Este algoritmo se aplica con especial sencillez al caso de los primos 1-seguros de los que se hablará en el Capítulo 4. En efecto, por definición, si p es un primo 1-seguro, entonces $p - 1 = 2q$ donde q es un primo impar. Si hacemos $F = q$ y $U = 2$ está claro que se verifican las condiciones necesarias para poder aplicar el algoritmo de modo óptimo.

3.3 Algoritmos de factorización

En la sección 2.4.4, se explicaron las condiciones que deben cumplir los factores primos del módulo de RSA. Allí se introducía la noción de primo robusto y se justificaba la necesidad de utilizar este tipo de primos en los factores del módulo precisamente para evitar el criptoanálisis basado en los algoritmos $p - 1$ de Pollard y $p + 1$ de Williams. Por este motivo, presentamos seguidamente una breve introducción a cada uno de ellos.

3.3.1 Algoritmo $p - 1$ de Pollard

J.M. Pollard publicó en [89] un método de factorización formalizando una serie de reglas que eran conocidas desde antiguo. La idea es utilizar la información relativa al orden de algún elemento a del grupo \mathbb{Z}_n^* para deducir propiedades sobre los factores de n . En efecto, por el teorema de Fermat, sabemos que si p es un primo y $\text{mcd}(a, p) = 1$, entonces $a^{p-1} \equiv 1 \pmod{p}$. Con esto, si Q es un entero tal que $p - 1|Q$, entonces $p|a^Q - 1$, puesto que $a^Q \equiv 1 \pmod{p}$. Por lo tanto, si coincide que alguno de los p es un factor de n , resulta que p divide a $\text{mcd}(a^Q - 1, n)$, luego basta comprobar si $\text{mcd}(a^Q - 1, n) \neq 1$ y $\text{mcd}(a^Q - 1, n) \neq n$.

La idea consiste, pues, en elegir números Q con muchos divisores de la forma $p - 1$, con p cualquier primo, y así buscar factores de n en un solo golpe de entre los diversos primos con los que se ha construido el número Q .

El método de Pollard resulta especialmente interesante si da la casualidad de que todos los factores de $p - 1$, donde p es uno de los factores de n , son menores que una cierta cota M . Recordemos que este tipo de números se denominan M -uniformes según la Definición 2.1. Entonces es sencillo construir números Q que sean productos de esos primos “pequeños” y calcular $\text{mcd}(a^Q - 1, n)$. El problema original queda así convertido en el problema de generar múltiplos de todos los enteros que contengan divisores pequeños.

Una forma de hacerlo puede ser la siguiente: sea $L(k)$ la sucesión formada tomando el mínimo común múltiplo de todos los números menores o iguales a k . Existe una forma recursiva de generar esta sucesión. Obviamente, $L(1) = 1$, $L(2) = 2$, $L(3) = 6$. Supongamos calculado $L(k)$. Si $k + 1$ no es primo, ni potencia de primo, $L(k + 1) = L(k)$; si, por el contrario, $k + 1 = p^\alpha$, entonces $L(k + 1) = pL(k)$. Supongamos que se dispone de la lista de primos

$$p_1 < p_2 < \dots < p_m \leq M.$$

Sea b_i el entero más grande tal que $p_i^{b_i} \leq M$. Con esto, el algoritmo procede de la siguiente forma:

Algoritmo 3.14 Dados un entero n a factorizar y una cota M , encontrar un factor no trivial de n .

LLAMADA: $\text{Pollard}(n, M)$;
 ENTRADA: Un entero n y una cota $M \in \mathbb{N}$.
 SALIDA: Un factor no trivial de n o 0 si falla.

1. [Inicialización]

Encontrar la sucesión de primos $p_1 < p_2 < \dots < p_m \leq M$.

2. [Lazo de potencias]

for ($a \in [2, n - 1]$ aleatoriamente elegido)

{

for ($i \in [1, m]$)

for ($j \in [1, b_i]$)

$a = a^{p_i} (\text{mod } n);$

3. [Cómputo del máximo común divisor]

$g = \text{mcd}(a - 1, n);$

if ($1 < g < n$)

return g ; /* El algoritmo tiene éxito */

}

return 0; /* El algoritmo ha fallado */

El algoritmo falla una vez se han ensayado todos los valores posibles para la base a . En ese caso, se ha de modificar la cota M y tomando un valor más alto, volver a ejecutar el algoritmo.

¿Cuánto tiempo tardará el algoritmo en detectar un factor de n ? Supongamos que

$$n = \prod_i p_i^{\alpha_i} \quad \text{y que} \quad p_i - 1 = \prod_j q_{ij}^{\beta_{ij}}. \quad (3.1)$$

Sea ahora q^β la potencia prima más alta en la factorización de $p_i - 1$. Entonces, el factor p_i aparecerá tan pronto como el lazo del algoritmo haya pasado por el valor q^β en la lista de las potencias primas utilizadas. Esto significa que el factor p_i de n para el que el valor q^β es el más pequeño de entre todos los factores p de n es el primero que aparece.

Observación 3.15 Podría ocurrir que, aunque $p_i - 1$ contenga sólo factores pequeños, éstos vengan afectados de exponentes muy grandes, tales que excede la cota de búsqueda. En ese caso, el algoritmo no los detectaría.

3.3.2 Algoritmo de Pollard, fase 2

Cuando el algoritmo falla, existe una variante, llamada *fase 2*, que consiste en lo siguiente. Supongamos que no se ha hallado factor alguno usando como cota M . Hagamos, como en la ecuación (3.1),

$$p_i - 1 = \prod_{j=1}^s q_{ij}^{\beta_{ij}} = F \cdot \prod_{j=1}^{s-1} q_{ij}^{\beta_{ij}},$$

donde sólo el factor F de $p_i - 1$ excede M . Elijamos una nueva cota M' y obtenemos la lista de primos $M < q_1 < \dots < q_{m'} \leq M'$. Denotemos por b el último valor obtenido al final de la *fase 1* del algoritmo, es decir, b es igual a a elevado al producto de todas las potencias primas por debajo de M , reducido módulo n . Calculemos ahora recursivamente $b^{q_{i+1}}$ como

$$b^{q_{i+1}} = b^{q_i} \cdot b^{q_{i+1}-q_i} \pmod{n},$$

y comprobemos si $\text{mcd}(b^{q_{i+1}}, n) > 1$. Puesto que las diferencias entre dos primos consecutivos son pequeñas y pares, merece la pena tener precomputados los valores

$$b^2, b^4, b^6, \dots, b^{\max(q_{i+1}-q_i)} \pmod{n}$$

y usar estos valores en la recursión. De este modo, el algoritmo ahora rueda mucho más deprisa que en la *fase 1*, con lo que lo que interesa es dividir el tiempo de computación entre ambas fases. Con ello, el valor de M' puede llegar a ser del orden de $M \ln M$ (véase [27, sección 5.4]). Con esto, el algoritmo queda como sigue:

Algoritmo 3.16 Dados n , número entero a factorizar, y dos cotas, $M_1 < M_2$, encontrar un factor no trivial de n .

LLAMADA: *Pollard2(n, M)*;

ENTRADA: Un entero n y una cota $M \in \mathbb{N}$.

SALIDA: Un factor no trivial de n o 0 si falla.

1. [Fase 1]

Haciendo $M = M_1$, utilizar la fase 1 para factorizar. Si se produce éxito, se finaliza el algoritmo. En caso contrario, sea b el valor de a elevado al producto de todas las potencias primas por debajo de M_1 , reducido módulo n .

2. [Inicialización]

Encontrar la sucesión de primos $p_1 < p_2 < \dots < p_m \leq M_1$ y la sucesión de primos $M_1 < q_1 < \dots < q_l \leq M_2$. Precomputar las diferencias $d_i = q_{i+1} - q_i$, y almacenar los valores b^{d_1}, \dots, b^{d_l} . Hacer $x = b$.

3. [Lazo]

```
for (i ∈ [1, l])
{
    x = x · bdi (mod n);
    g = mcd(x - 1, n);
    if (1 < g < n)
        return g; /* El algoritmo ha tenido éxito */
    }
return 0;           /* El algoritmo ha fallado */
```

Esta *fase 2* del algoritmo rueda mucho más deprisa que la *fase 1* pues el lazo sólo ha de efectuar una multiplicación modular en cada vuelta.

3.3.3 Algoritmo $p + 1$ de Williams

El método de Williams está descrito en [122]. Utiliza sucesiones de Lucas en vez de potencias y alcanza una rápida factorización de n si algún factor p de n es tal que $p + 1$ se descompone en factores primos “pequeños”, es decir, si es M -uniforme, con M un valor razonable, por ejemplo 10^7 .

Para este método se necesita utilizar las sucesiones o funciones de Lucas, cuya definición es como sigue: dados dos enteros P y Q , sean α, β las raíces del polinomio $x^2 - Px + Q$. Con esto, las funciones de Lucas, $U_n(P, Q)$ y $V_n(P, Q)$, se definen como:

$$U_n(P, Q) = \frac{\alpha^n - \beta^n}{\alpha - \beta},$$

$$V_n(P, Q) = \alpha^n + \beta^n.$$

También se utiliza $\Delta = (\alpha - \beta)^2 = P^2 - 4Q$. Estas funciones presentan un gran número de propiedades que se pueden ver, por ejemplo, en [94].

Necesitamos también el siguiente

Teorema 3.17 Si p es un primo impar que no divide a Q y el símbolo de Legendre $\left(\frac{\Delta}{p}\right) = \varepsilon$, entonces

$$\begin{aligned} U_{(p-\varepsilon)m}(P, Q) &\equiv 0 \pmod{p}, \\ V_{(p-\varepsilon)m}(P, Q) &\equiv 2Q^{m(1-\varepsilon)/2} \pmod{p}. \end{aligned}$$

La demostración puede consultarse en [60].

Fase 1 del algoritmo

Supongamos que p es un divisor de n y

$$p = \prod_{i=1}^k q_i^{\alpha_i} - 1,$$

donde q_i es el i -ésimo primo y $q_i^{\alpha_i} \leq M$, es decir, $p+1$ es un número M -uniforme. Si hacemos

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

donde β_i es tal que $q_i^{\beta_i} \leq M < q_i^{\beta_i+1}$, $1 \leq i \leq k$. Es claro que $p+1|R$. Por el Teorema 3.17, vemos que si $\text{mcd}(Q, n) = 1$ y $\left(\frac{\Delta}{p}\right) = -1$, entonces $p|U_R(P, Q)$ y, por ser p un divisor de n , también $p|\text{mcd}(U_R(P, Q), n)$.

El problema asociado a este método reside en la dificultad de encontrar de manera eficiente un no-residuo cuadrático junto con la sucesión de Lucas adecuada. Williams desarrolla en su artículo un procedimiento muy complejo basado en las propiedades de las funciones de Lucas para resolverlo. Remitimos al artículo original para ver el desarrollo concreto.

Fase 2 del algoritmo

Análogamente a la fase 2 del algoritmo $p-1$ de Pollard, ahora suponemos que

$$p = s \prod_{i=1}^k q_i^{\alpha_i} - 1,$$

donde s es un primo, los q_i igual que en la fase 1 y $M_1 < s \leq M_2$. Denominemos s_j la lista ordenada de primos tales que $M_1 < s_j \leq M_2$. También en este caso la idea es aprovechar que las diferencias $d_j = s_{j+1} - s_j$ crecen muy lentamente, por lo que no habrá muchas d_j diferentes: de hecho, si llamamos $d(x)$ al mayor valor de d_j para todos los primos comprendidos entre 1 y x , esto es, $d(x) = \max(d_j, p_j \leq x)$, se tiene que $d(200000) = 43$. Williams aprovecha este hecho para tener precalculados los valores necesarios de las funciones de Lucas y acelerar así la fase 2 del algoritmo.

3.4 Generación de números pseudoaleatorios

Ya hemos hablado del interés que tiene poder generar sucesiones de números que tengan la *apariencia* de ser aleatorios, con utilidad en diversas situaciones. En particular, en la sección 2.7.3 se introdujo el criptosistema de Blum-Goldwasser, junto con una explicación del generador de Blum, Blum y Shub. En los siguientes apartados vamos a presentar una descripción algorítmica de este y otros generadores, que nos serán de utilidad para las secciones sucesivas.

3.4.1 Algoritmo BBS

Este algoritmo se basa en iterar la congruencia $x_{i+1} \equiv x_i^2 \pmod{N}$, donde N es un entero de Blum (véase [13] y la sección 2.7.3). En cada paso de la iteración se obtiene un bit, el menos significativo, como salida del generador. Sin embargo, Vazirani y Vazirani, en la referencia [117], mejoran la eficiencia de este generador. En efecto, suponiendo cierta la inabordabilidad del problema de la factorización de enteros, aseguran que se pueden considerar aleatorios $c \log_2 \log_2 N$ bits de los producidos en cada iteración, donde c es una constante.

Algoritmo 3.18 Dado un entero n , este algoritmo consigue un número aleatorio de n bits.

LLAMADA: `NumeroAleatorioBBS(n);`
 ENTRADA: Un entero n .
 SALIDA: Un número aleatorio de n bits.

1. [Inicialización]

Se fija un N entero de Blum y se genera una semilla $s \in \mathbb{Z}_N^*$ aleatoria.

```
s = GeneraSemillaAleatoria();
x0 = s;
resultado = 0;
```

2. [Lazo]

```
while (n > 0)
```

```
{
```

```
    xi+1 = xi^2(mod N);
    bits = ⌊log2 log2 N⌋ bits menos significativos de xi+1;
    resultado = resultado concatenado con bits;
    i = i + 1;
    n = n - ⌊log2 log2 N⌋;
```

```
}
```

3. [Final]

Se devuelve `resultado`.

Observación 3.19 Obsérvese que el paso [Inicialización] exige la generación de una semilla por algún método aleatorio, que no detallamos. Véase a este respecto la sección 2.7.3.

Proposición 3.20 El tiempo de ejecución de este algoritmo es $O(n)$.

Demostración Observemos que el lazo del algoritmo se reduce a elevar al cuadrado elementos en \mathbb{Z}_N^* , por lo que el coste de cada operación es $O((\log_2 N)^2)$. Puesto que en cada iteración se pueden obtener $\log_2 \log_2 N$ bits, es claro que el coste total para obtener n bits será

$$O((\log_2 N)^2) \frac{n}{\log_2 \log_2 N} = O(n).$$

3.4.2 Algoritmo de Lehmer

Se atribuye a D.H. Lehmer la sugerencia de construir un generador de números pseudoaleatorios utilizando la siguiente fórmula recursiva:

$$x_{n+1} \equiv Kx_n \pmod{m}.$$

La elección de adecuados valores para el módulo m y para la constante $K \in \mathbb{Z}_m$ son decisivas a la hora de que el generador produzca secuencias estadísticamente válidas. En la literatura (véase, por ejemplo, [85]) se recomienda utilizar un valor de m de la forma q^n o $2q^n$, donde q es un primo impar, y un valor de K que sea una raíz primitiva para ese módulo (es decir, K genera el grupo \mathbb{Z}_m^*) Esto garantiza que el ciclo tiene de longitud el orden de \mathbb{Z}_m^* , y, si m es primo, la longitud del ciclo será justamente $m - 1$.

El propio Lehmer sugiere que se utilice

$$K = 14^{29}, \quad m = 2^{31} - 1.$$

Este valor de m es, en particular, un primo de Mersenne. El número 14 es raíz primitiva en el grupo $\mathbb{Z}_{2^{31}-1}^*$, pero se toma $K = 14^{29}$ porque 29 no divide la longitud del ciclo, es decir $29 \nmid (2^{31} - 2)$ y así se introduce cierta “aleatoriedad” en la secuencia.

Otro posible candidato es el primo de Mersenne $m = 2^{61} - 1$ al que correspondería como longitud de ciclo $2^{61} - 2$. Una raíz primitiva en $\mathbb{Z}_{2^{61}-1}^*$ es 37, con lo que se podría tomar como candidato para K el número 37^{29} : también en este caso se verifica que $29 \nmid (2^{61} - 2)$.

Hemos implementado este algoritmo —y el código se ofrece en el Apéndice I— eligiendo $m = 2^{31} - 1$, con lo que el algoritmo genera números de como máximo 31 bits; si se tomara, por ejemplo, $m = 2^{61} - 1$, se podría llegar a 61 bits. En cualquier caso, cuando se requieren números de un tamaño mayor, el algoritmo concatena el resultado de varias iteraciones hasta llegar a la cantidad necesaria.

Algoritmo 3.21 Dado un entero positivo n este algoritmo genera números pseudoaleatorios de n bits.

LLAMADA: `NumeroAleatorioLehmer(n);`
 ENTRADA: Un entero n .
 SALIDA: Un número aleatorio de n bits.

1. [Inicialización]

Se inicializan las constantes del sistema. Se eligen semillas aleatorias en $\mathbb{Z}_{2^{32}}^*$.

$$m = 2^{31} - 1;$$

$$K = 14^{29}(\text{mod } m);$$

$$x_0 = \text{GeneraSemillaAleatoria}();$$

$$\text{resultado} = 0;$$

2. [Lazo]

`while (n > 0)`

{

$$x_{i+1} = Kx_i(\text{mod } m);$$

$$\text{resultado} = x_{i+1};$$

$$n = n - \lfloor \log_2 \text{resultado} \rfloor;$$

}

3. [Final]

Se devuelve `resultado`.

Proposición 3.22 El tiempo de ejecución de este algoritmo es $O(n)$.

Demostración Observemos que el lazo del algoritmo se consiste en realizar multiplicaciones de elementos en \mathbb{Z}_m^* , por lo que el coste de cada operación es $O((\log_2 m)^2)$. Puesto que en cada iteración se pueden obtener $\log_2 m$ bits, es claro que el coste total para obtener n bits será

$$O((\log_2 m)^2) \frac{n}{\log_2 m} = O(n).$$

■

3.4.3 Algoritmo de Tausworthe

El generador de Tausworthe produce números pseudoaleatorios generando una sucesión de bits a partir de una fórmula de recurrencia lineal módulo 2 y tomando bloques de bits sucesivos. De modo más preciso, sea \mathbb{F}_2 un cuerpo finito de característica 2 y sea $P(z) = z^k - a_1z^{k-1} - \dots - a_k$ un polinomio con coeficientes en \mathbb{F}_2 . Consideraremos la ley de recurrencia

$$x_n = a_1x_{n-1} + \dots + a_kx_{n-k}, \quad (3.2)$$

cuyo polinomio característico es precisamente $P(z)$. Fijando un estado inicial $s_0 = (x_0, \dots, x_{k-1}) \in \mathbb{F}_2^k$, definimos

$$u_n = \sum_{i=1}^L x_{ns+i-1} 2^{i-1}, \quad (3.3)$$

donde s y L son enteros positivos. Si P es un polinomio primitivo, $s_0 \neq \mathbf{0}$ y $\rho = 2^k - 1$ es primo con respecto a s , entonces las sucesiones de bits (3.2) y (3.3) son periódicas con período ρ .

En principio, calcular u_{n+1} a partir de u_n implica realizar s pasos de la recurrencia y esto puede llegar a ser lento en general. Sin embargo, en ciertas condiciones, este proceso se puede acelerar. Según se describe en [60], supongamos que se cumple lo siguiente:

1. $P(z)$ es un trinomio primitivo, de la forma $P(z) = z^k - z^q - 1$, tal que $0 < 2q < k$;
2. Se toma s tal que $0 < s \leq k - q \leq L$;
3. $\text{mcd}(s, 2^k - 1) = 1$;
4. L es el tamaño de palabra del procesador (en la actualidad, suelen ser de 32 bits).

Se describe ahora el algoritmo que, con las anteriores condiciones, permite calcular $s_n = (x_{ns}, \dots, x_{ns+L-1})$ a partir de $s_{n-1} = (x_{(n-1)s}, \dots, x_{(n-1)s+L-1})$ de forma rápida. Sean A , B y C vectores de \mathbb{F}_2^L . Supongamos que, inicialmente, $A = s_{n-1}$, mientras que C es una “máscara” de bits compuesta de k unos y seguida de $L - k$ ceros. El algoritmo da los siguientes pasos:

1. $B \leftarrow A$ desplazado a la izquierda en q bits;
2. $B \leftarrow A \oplus B$;
3. $B \leftarrow B$ desplazado a la derecha en $(k - s)$ bits;
4. $A \leftarrow A \& C$;
5. $A \leftarrow A$ desplazado a la izquierda en s bits;
6. $A \leftarrow A \oplus B$.

En este algoritmo, los símbolos $\&$ y \oplus representan las operaciones “Y” y “O-exclusivo”, es decir, la suma y el producto en \mathbb{F}_2 . Estas operaciones se implementan en hardware de forma inmediata por lo que este generador tiene un gran interés práctico.

Por desgracia, un generador de esta forma no es tan deseable pues presenta ciertos defectos en la distribución estadística de los números generados y, además, su período máximo nunca puede exceder 2^b donde b es el tamaño de la palabra del procesador, típicamente, 32 bits. Por ello, en [60] se propone combinar de

cierta manera tres trinomios como el descrito de forma que se puedan conservar las características de cada uno de ellos por separado. En particular, para cada $j = 1, \dots, J$, consideremos un generador de Tausworthe con polinomio característico $P_j(z)$, con grado k_j y con $s = s_j$ tal que $\text{mcd}(s_j, 2^{k_j} - 1) = 1$. Definamos la salida como la combinación de las salidas de los J generadores mediante la operación “O-exclusivo”. Puesto que los polinomios característicos son primos entre sí, se tiene que el período del generador combinado será

$$\rho = \text{mcm}(2^{k_1} - 1, \dots, 2^{k_J} - 1) \quad (3.4)$$

$$= (2^{k_1} - 1) \times \dots \times (2^{k_J} - 1).$$

El autor demuestra que el siguiente trío presenta una distribución máximamente equidistribuida:

$$(k_1, q_1, s_1) = (31, 13, 12)$$

$$(k_2, q_2, s_2) = (29, 2, 4)$$

$$(k_3, q_3, s_3) = (28, 3, 17)$$

cuando se combinan las salidas de los tres generadores mediante la operación “O-exclusivo” descrita anteriormente. El período del generador será, de acuerdo a la fórmula (3.4), $(2^{31} - 1)(2^{29} - 1)(2^{28} - 1) \simeq 2^{88}$.

Presentamos una implementación de este algoritmo con estos parámetros en el Anexo I. Al igual que en el caso del generador de Lehmer, este generador produce números de 32 bits. Así pues, cuando son necesarios números de un tamaño mayor, el algoritmo concatena el resultado de varias iteraciones hasta llegar a la cantidad requerida.

Algoritmo 3.23 Dado un entero positivo n este algoritmo genera números pseudoaleatorios de n bits.

LLAMADA: `NumeroAleatorioTausworthe(n);`

ENTRADA: Un entero n .

SALIDA: Un número aleatorio de n bits.

1. [Inicialización]

Se eligen semillas aleatorias en $\mathbb{Z}_{2^{32}}^*$.

`s1 = GeneraSemillaAleatoria();`

`s2 = GeneraSemillaAleatoria();`

`s3 = GeneraSemillaAleatoria();`

`resultado = 0;`

2. [Lazo]

`while (n > 0)`

```

{
    b = (((s1 <<13)^s1) >>19);
    s1 = (((s1 & 4294967294U) <<12)^b);
    b = (((s2 <<2)^s2) >>25);
    s2 = (((s2 & 4294967288U) <<4)^b);
    b = (((s3 <<3)^s3) >>11);
    s3 = (((s3 & 4294967280U) <<17)^b);
    resultado = s1^s2^s3;
    n = n - [log2(resultado)];
}

```

3. [Final]

Se devuelve `resultado`.

Observación 3.24 Hemos utilizado la notación estándar de C, en la que el símbolo `<<n` representa desplazamiento a la izquierda en n bits; el símbolo `>>n` representa desplazamiento a la derecha en n bits; y el símbolo `^` representa la operación “O-exclusivo”.

Proposición 3.25 El tiempo de ejecución de este algoritmo es $O(n)$.

Demostración Observemos que el lazo del algoritmo consiste en realizar operaciones básicas de desplazamiento y “O-exclusivo”. Estas operaciones tienen un coste fijo, con independencia del número de bits, por lo que el coste total de la generación depende directamente del número total de bits a generar, es decir, es $O(n)$. ■

Observación 3.26 Si bien los algoritmos 3.18, 3.21 y 3.23 presentados comparten el tiempo asintótico de ejecución $O(n)$, en la práctica resultan más rápidos el de Lehmer y Tausworthe frente a BBS, pues éste último proporciona menos bits en cada vuelta del lazo. Sin embargo, como contrapartida, la seguridad criptográfica de los generadores de Lehmer y Tausworthe resulta deficiente y no sirven como base para ningún sistema criptográfico.

3.5 Generación de primos aleatorios

Equipados con los algoritmos descritos hasta ahora, es trivial generar números primos pseudoaleatorios de tamaño elegido.

Algoritmo 3.27 Dados enteros positivos n y t , este algoritmo genera un número primo pseudoaleatorio de n bits con probabilidad mayor de $1 - 2^{-2t}$.

LLAMADA: `GeneraPrimoAleatorio(n, t);`

ENTRADA: Enteros n y t .

SALIDA: Un número primo aleatorio de n bits
con parámetro de seguridad t .

1. [Inicialización]

Se elige la semilla para el generador aleatorio BBS.

2. [Lazo]

```
while (encontrado == NO)
{
    p = NumeroAleatorioBBS(n);
    if (MillerRabin(p, t))
    {
        encontrado = SI;
        return p;
    }
}
```

Hemos hecho uso del generador de Blum, Blum y Shub para obtener números aleatorios del tamaño requerido, pero el algoritmo, obviamente, admite el uso de cualquier otro generador aleatorio que funcione correctamente.

Capítulo 4

Primos seguros

Resumen del capítulo

Se discute la noción de primo seguro y se generaliza mediante la introducción de la signatura; se pone en relación con las cadenas de primos de Cunningham y los primos de Sophie Germain. Se aportan las funciones recuento para los primos 1- y 2-seguros junto con datos y gráficas experimentales. Se aportan también las funciones generalizadas de recuento para los primos k -seguros.

4.1 Primos seguros

4.1.1 Definición y propiedades elementales

Definición 4.1 Un número primo impar p se dice que es k veces seguro (o que es un primo k -seguro) de signatura $(\varepsilon_1, \dots, \varepsilon_k)$, donde $\varepsilon_1, \dots, \varepsilon_k \in \{+1, -1\}$, si existen k números primos impares p_1, \dots, p_k tales que

$$p = 2p_1 + \varepsilon_1, \quad p_1 = 2p_2 + \varepsilon_2, \quad \dots, \quad p_{k-1} = 2p_k + \varepsilon_k.$$

El entero k se denomina orden de la signatura.

Notación 4.2 El conjunto de los números primos se denota por \mathbb{P} . Para cada $k > 0$, el conjunto de los números primos k veces seguros de signatura $(\varepsilon_1, \dots, \varepsilon_k)$ se denota por $\mathbb{P}(\varepsilon_1, \dots, \varepsilon_k)$. Cuando se hable de primos k -seguros sin especificar su signatura se entenderá que ésta es $\varepsilon_1 = \dots = \varepsilon_k = +1$ y se escribirá \mathbb{P}_k^+ en vez de $\mathbb{P}(\varepsilon_1, \dots, \varepsilon_k)$. También escribiremos $\mathbb{P}_k^- = \mathbb{P}(\varepsilon_1, \dots, \varepsilon_k)$, cuando $\varepsilon_1 = \dots = \varepsilon_k = -1$.

Proposición 4.3 Si $p > 5$ es un primo k -seguro de signatura $(\varepsilon_1, \dots, \varepsilon_k)$, entonces

$$p \equiv 2^k + \sum_{h=1}^k \varepsilon_h 2^{h-1} \pmod{2^{k+1}}.$$

Demostración Se deduce por recurrencia sobre k , teniendo en cuenta que si p es un primo k veces seguro con signatura $(\varepsilon_1, \dots, \varepsilon_k)$, entonces p_1 es un primo $k-1$ veces seguro con signatura $(\varepsilon'_1 = \varepsilon_2, \dots, \varepsilon'_{k-1} = \varepsilon_k)$. En efecto, si $k=1$ entonces p_1 es impar ya que $p > 5$; por tanto, $p_1 = 2q + 1$, de donde $p = 2 + \varepsilon_1 + 2^2 q$, que es equivalente a la fórmula del enunciado para este caso. Suponiendo $k > 1$ y aplicando la hipótesis de inducción a p_1 se tiene:

$$p_1 \equiv 2^{k-1} + \sum_{i=1}^{k-1} \varepsilon'_i 2^{i-1} \pmod{2^k},$$

o equivalentemente,

$$p_1 = 2^{k-1} + \sum_{i=1}^{k-1} \varepsilon_{i+1} 2^{i-1} + 2^k m.$$

Por tanto,

$$\begin{aligned} p &= 2p_1 + \varepsilon_1 = 2 \left(2^{k-1} + \sum_{i=1}^{k-1} \varepsilon_{i+1} 2^{i-1} + 2^k m \right) + \varepsilon_1 \\ &= 2^k + \sum_{i=1}^{k-1} \varepsilon_{i+1} 2^i + \varepsilon_1 + 2^{k+1} m \stackrel{(h=i+1)}{=} 2^k + \sum_{h=1}^k \varepsilon_h 2^{h-1} + 2^{k+1} m \\ &\equiv 2^k + \sum_{h=1}^k \varepsilon_h 2^{h-1} \pmod{2^{k+1}}, \end{aligned}$$

con lo que se concluye. ■

Corolario 4.4 Los conjuntos de primos k veces seguros de signaturas distintas son dos a dos disjuntos; esto es,

$$\mathbb{P}(\varepsilon_1, \dots, \varepsilon_k) \cap \mathbb{P}(\varepsilon'_1, \dots, \varepsilon'_k) = \emptyset, \text{ si } (\varepsilon_1, \dots, \varepsilon_k) \neq (\varepsilon'_1, \dots, \varepsilon'_k).$$

Demostración Sean $1 \leq i_1 < \dots < i_l \leq k$ los valores del índice h para los cuales $\varepsilon_h = +1$. Basta tener en cuenta que al recorrer $(\varepsilon_1, \dots, \varepsilon_k)$ todos los valores posibles, los números

$$\begin{aligned} 2^k + \sum_{h=1}^k \varepsilon_h 2^{h-1} &= 1 + \sum_{h=1}^k (1 + \varepsilon_h) 2^{h-1} \\ &= 1 + \sum_{j=1}^l 2 \cdot 2^{i_j-1} = 1 + \sum_{j=1}^l 2^{i_j} \end{aligned}$$

recorren una sola vez todos los números impares menores que 2^{k+1} . Se concluye en virtud de la Proposición 4.3. ■

Corolario 4.5 Si una signatura $\mathbb{P}(\varepsilon_1, \dots, \varepsilon_k)$ está vacía, también lo están todas las de orden superior.

Demostración En efecto, por definición $p \in \mathbb{P}(\varepsilon_1, \dots, \varepsilon_k)$ si y sólo si $p = 2p_1 + \varepsilon_1$, con $p_1 \in \mathbb{P}(\varepsilon'_1, \dots, \varepsilon'_{k-1})$. Por tanto, si esta signatura es vacía necesariamente lo será también aquélla. ■

Ejemplo 4.6 Existen dos clases de primos 1-seguros: los de signatura $+1$ (habitualmente conocidos como 1-seguros) y los de signatura -1 . Los primeros son congruentes con 3 módulo 4 , mientras que los segundos son congruentes con 1 módulo 4 . Por ejemplo, $7, 11, 23$ son 1-seguros de signatura $+1$, mientras que $13, 37, 61$ son de signatura -1 . Los primos $17, 19, 29$ no son 1-seguros ni de signatura positiva ni de signatura negativa.

Observación 4.7 Obsérvese que el orden de los signos en la signatura es importante, porque si π es una permutación de $\{1, \dots, k\}$, en general se tiene $\mathbb{P}(\varepsilon_1, \dots, \varepsilon_k) \neq \mathbb{P}(\varepsilon_{\pi(1)}, \dots, \varepsilon_{\pi(k)})$. Por ejemplo, $7 \in \mathbb{P}(+1, -1)$ y $7 \notin \mathbb{P}(-1, +1)$, lo que se sigue del Corolario 4.4, pues la signatura debe considerarse siempre como un sistema ordenado; es decir, un elemento de $\{+1, -1\}^k$. Por tanto, dos signaturas se considerarán iguales si y sólo si $\varepsilon_1 = \varepsilon'_1, \dots, \varepsilon_k = \varepsilon'_k$.

4.1.2 Primos seguros y primos de Sophie Germain

No hay que confundir los primos 1-seguros con los primos llamados de Sophie Germain, que esta autora introdujo alrededor de 1825 en sus trabajos sobre “el primer caso del último teorema de Fermat” (véase, por ejemplo, [94, 2.VII p. 66], [125]). Un primo q se dice que es de Sophie Germain si $2q+1$ es primo. Por ejemplo, el 7 es un primo 1-seguro que no es de Sophie Germain, mientras que, como hemos visto, 29 no es 1-seguro pero sí es un primo de Sophie Germain. El teorema de Sophie Germain establece que si p es un primo de Sophie Germain entonces no existen enteros x, y, z no nulos y no múltiplos de p tales que $x^p + y^p = z^p$.

Durante mucho tiempo el mayor número de Sophie Germain conocido fue $p = 39051 \cdot 2^{6001} - 1$, descubierto por Wilfrid Keller, de Hamburgo, en 1986; véase [94, 2.VII p. 66], [125]. Esto puede reformularse diciendo que $p' = 2p+1$ es un primo 1-seguro de más de 1800 cifras. Otros primos de Sophie Germain son $296385 \cdot 2^{4251} - 1$ y $53375 \cdot 2^{4203} - 1$: para más detalles, véase [94, p. 166].

Sin embargo, en enero de 1994 (véase [32]), Harvey Dubner obtuvo un primo de Sophie Germain mayor: $q = c \cdot 3003 \cdot 10^b - 1$, donde $c = 1803301$, $b = 4526$. Nótese que p tiene alrededor de 1812 dígitos, mientras que q tiene 4536.

Los primos de Sophie Germain son también importantes para determinar números de Mersenne que no son primos. En efecto, si p es un primo de Sophie Germain, entonces $2^p - 1$ es divisible por $2p+1$; por ejemplo, $2^{3539} - 1$ es divisible por 7079. Tienen también otras aplicaciones al estudio de los *repunits*, que son los números cuya expresión en base 10 está formada por n unos, de modo que su valor es $\frac{1}{9}(10^n - 1)$.

4.1.3 Signaturas alternadas

Definición 4.8 Se dice que una signatura $(\varepsilon_1, \dots, \varepsilon_k)$ es alternada cuando existan índices $1 \leq i < j \leq k$ tales que $\varepsilon_i \varepsilon_j = -1$.

Proposición 4.9 Si n es un entero impar no divisible por 3, entonces $n^2 - 1$ es divisible por 24.

Demostración Como n es impar, se tiene $n = 2m + 1$. Por tanto $n^2 - 1 = 4m(m + 1)$ y puesto que, o bien m o bien $m + 1$ es par, se sigue que $n^2 - 1$ es divisible por 8.

Supongamos $m \equiv 1 \pmod{3}$; es decir $m = 1 + 3l$. Entonces, sustituyendo queda

$$n = 2m + 1 = 2(1 + 3l) + 1 = 3(1 + 2l),$$

lo cual es imposible en virtud de la hipótesis. Luego, o bien $m \equiv 0 \pmod{3}$, o bien $m \equiv 2 \pmod{3}$. En ambos casos el producto $m(m + 1)$ es divisible por 3, con lo que se concluye. ■

Lema 4.10 Sean p, p_1, p_2 tres primos impares tales que, o bien $p = 2p_1 + 1$, $p_1 = 2p_2 - 1$, o bien $p = 2p_1 - 1$, $p_1 = 2p_2 + 1$. Entonces $p^2 - 1 = 8p_1p_2$.

Demostración En el primer caso, se tiene

$$\frac{p-1}{2} = p_1, \quad \frac{p+1}{4} = p_2,$$

de donde se sigue lo pedido. Análogamente, en el segundo caso, se tiene

$$\frac{p+1}{2} = p_1, \quad \frac{p-1}{4} = p_2,$$

con lo que se concluye. ■

Proposición 4.11 Los conjuntos $\mathbb{P}(\varepsilon_1, \dots, \varepsilon_k)$ que presenten signatura alternada se clasifican como sigue:

1. Si $k = 2$, entonces

$$\mathbb{P}(+1, -1) = \{11\}, \quad \mathbb{P}(-1, +1) = \{13\}.$$

2. Si $k = 3$, entonces $\mathbb{P}(+1, +1, -1) = \{23\}$ y todas las demás signaturas están vacías.
3. Si $k = 4$, entonces $\mathbb{P}(+1, +1, +1, -1) = \{47\}$ y todas las demás signaturas están vacías.
4. Si $k \geq 5$, entonces todas las signaturas alternadas están vacías.

Demostración Dividimos la demostración según los apartados del enunciado.

1. Los elementos de $\mathbb{P}(+1, -1)$ y $\mathbb{P}(-1, +1)$ son primos impares. Aplicando el Lema 4.10 y la Proposición 4.9, se sigue que si $p \in \mathbb{P}(+1, -1) \cup \mathbb{P}(-1, +1)$, entonces $8p_1p_2$ es divisible por 3. Luego, o bien $p_1 = 3$, o bien $p_2 = 3$. En el primer caso, se tiene:
 - a) Si $p \in \mathbb{P}(+1, -1)$, entonces $p_2 = 2$, que está excluido por la definición.

- b) Si $p \in \mathbb{P}(-1, +1)$, entonces $p_2 = 1$, que está excluido por la definición.

En el segundo caso, se tiene:

- a) Si $p \in \mathbb{P}(+1, -1)$, entonces $p = 11$.
- b) Si $p \in \mathbb{P}(-1, +1)$, entonces $p = 13$.

2. En este caso existen 6 signaturas, que analizamos una por una:

- a) Si $p \in \mathbb{P}(+1, +1, -1)$, entonces $p_1 \in \mathbb{P}(+1, -1) = \{11\}$; luego $p = 23$.
- b) Si $p \in \mathbb{P}(+1, -1, +1)$, entonces $p_1 \in \mathbb{P}(-1, +1) = \{13\}$; de donde $p = 2p_1 + 1$ no es primo, luego $\mathbb{P}(+1, -1, +1) = \emptyset$.
- c) Si $p \in \mathbb{P}(-1, +1, +1)$, entonces $p \in \mathbb{P}(-1, +1) = \{13\}$, de donde $p_1 = 7$, $p_2 = 3$ y como $p_2 = 2p_3 + 1$, p_3 ya no puede ser primo; luego $\mathbb{P}(-1, +1, +1) = \emptyset$.
- d) Si $p \in \mathbb{P}(-1, -1, +1)$, entonces $p_1 \in \mathbb{P}(-1, +1) = \{13\}$; de donde $p = 2p_1 - 1$ no es primo; luego $\mathbb{P}(-1, -1, +1) = \emptyset$.
- e) Si $p \in \mathbb{P}(-1, +1, -1)$, entonces $p_1 \in \mathbb{P}(+1, -1) = \{11\}$; de donde $p = 2p_1 - 1$ no es primo; luego $\mathbb{P}(-1, +1, -1) = \emptyset$.
- f) Si $p \in \mathbb{P}(+1, -1, -1)$, entonces $p \in \mathbb{P}(+1, -1) = \{11\}$, de donde $p_1 = 5$, $p_2 = 3$ y como $p_2 = 2p_3 + 1$, p_3 ya no puede ser primo; luego $\mathbb{P}(+1, -1, -1) = \emptyset$.

3. En principio, en este caso existen 14 signaturas, pero todas las signaturas que no contengan como subsignatura la $(+1, +1, -1)$ están vacías, en virtud de lo demostrado en el apartado anterior. Por tanto, de acuerdo con el Corolario 4.5 sólo necesitamos considerar las 4 que aparecen, cuando se le añada $+1$ ó -1 , delante o detrás, a la signatura $(+1, +1, -1)$. Es decir:

- a) Si $p \in \mathbb{P}(+1, +1, +1, -1)$, entonces $p_2 \in \mathbb{P}(+1, -1) = \{11\}$, de donde $p_1 = 2p_2 + 1 = 23$, $p = 2p_1 + 1 = 47$, luego $\mathbb{P}(+1, +1, +1, -1) = \{47\}$.
- b) Si $p \in \mathbb{P}(+1, +1, -1, +1)$, entonces $p_2 \in \mathbb{P}(-1, +1) = \{13\}$, de donde $p_1 = 2p_2 + 1$ no es primo; luego $\mathbb{P}(+1, +1, -1, +1) = \emptyset$.
- c) Si $p \in \mathbb{P}(-1, +1, +1, -1)$, entonces $p_2 \in \mathbb{P}(+1, -1) = \{11\}$, de donde $p_1 = 2p_2 + 1 = 23$ y, por consiguiente, $p = 2p_1 - 1$ ya no es primo; luego $\mathbb{P}(-1, +1, +1, -1) = \emptyset$.
- d) Si $p \in \mathbb{P}(+1, +1, -1, -1)$, entonces $p_1 \in \mathbb{P}(+1, -1) = \{11\}$, de donde $p_2 = 5$, $p_3 = 3$ y como $p_3 = 2p_4 - 1$, p_4 ya no puede ser primo impar; luego $\mathbb{P}(+1, +1, -1, -1) = \emptyset$.

4. Para este último caso, existen en principio 30 signaturas, pero todas las que no contengan la subsignatura $(+1, +1, +1, -1)$ estarán vacías, en virtud de lo demostrado en el apartado anterior. Así pues, nuevamente de acuerdo con el Corolario 4.5 nos queda sólo considerar los 4 casos en que esta subsignatura va precedida o seguida de $+1$ ó -1 . Esto es:

- a) Si $p \in \mathbb{P}(+1, +1, +1, +1, -1)$, entonces $p_3 \in \mathbb{P}(+1, -1) = \{11\}$, de donde $p_2 = 2p_3 + 1 = 23$, $p_1 = 2p_2 + 1 = 47$ y $p_1 = 2p_2 + 1$ ya no es primo; luego $\mathbb{P}(+1, +1, +1, +1, -1) = \emptyset$.
 - b) Si $p \in \mathbb{P}(+1, +1, +1, -1, +1)$, entonces $p_3 \in \mathbb{P}(-1, +1) = \{13\}$, de donde $p_2 = 2p_3 + 1$ no es primo; luego $\mathbb{P}(+1, +1, +1, -1, +1) = \emptyset$.
 - c) Si $p \in \mathbb{P}(-1, +1, +1, +1, -1)$, entonces $p_3 \in \mathbb{P}(+1, -1) = \{11\}$, de donde $p_2 = 2p_3 + 1 = 23$, $p_1 = 2p_2 + 1 = 47$ y $p_1 = 2p_2 + 1$ ya no es primo; luego $\mathbb{P}(-1, +1, +1, +1, -1) = \emptyset$.
 - d) Si $p \in \mathbb{P}(+1, +1, +1, -1, -1)$, entonces $p_2 \in \mathbb{P}(+1, -1) = \{11\}$, de donde $p_3 = 5$, $p_4 = 3$ y como $p_4 = 2p_5 - 1$, p_4 ya no puede ser primo impar; luego $\mathbb{P}(+1, +1, +1, -1, -1) = \emptyset$.
-

4.1.4 Cadenas de primos seguros

Definición 4.12 Una cadena de primos seguros de longitud k es una sucesión de números primos p, p_1, \dots, p_{k-1} tales que

$$p = 2p_1 + \varepsilon, p_1 = 2p_2 + \varepsilon, \dots, p_{k-2} = 2p_{k-1} + \varepsilon, \quad \varepsilon \in \{-1, +1\}.$$

Observación 4.13 Nótese que un primo p ocupa el primer lugar en una cadena de longitud k si y sólo si $p \in \mathbb{P}_{k-1}^+ \cup \mathbb{P}_{k-1}^-$.

Ejemplo 4.14 A continuación se adjuntan los cardinales de los conjuntos de todas las signaturas halladas al recorrer los números primos hasta $n = 7500000$. Se listan también como máximo los cincuenta primeros elementos de cada signatura. Entre los primos hasta 7500000 existen 460416 que no están en ninguna signatura, es decir, no son seguros. El número total de primos se obtiene sumando estas cantidades:

$$\begin{aligned} 460416 + 21970 + 21966 + 1800 + 1807 + 131 + 123 + 18 + 24 \\ + 3 + 2 + 1 + 1 = 508262. \end{aligned}$$

$$\mathbb{P}_1^+(n) = \{p \in \mathbb{P}(+1) : p \leq n\} \quad (\#\mathbb{P}_1^+(n) = 21970)$$

7,	59,	83,	107,	179,
227,	263,	347,	383,	467,
479,	503,	563,	587,	839,
863,	887,	983,	1019,	1187,
1283,	1307,	1319,	1367,	1487,
1523,	1619,	1823,	1907,	2027,
2063,	2099,	2207,	2447,	2459,
2579,	2819,	2903,	2963,	2999,
3023,	3119,	3167,	3203,	3467,
3623,	3779,	3803,	3863,	3947, ...

$$\mathbb{P}_1^-(n) = \{p \in \mathbb{P}(-1) : p \leq n\} (\#\mathbb{P}_1^-(n) = 21966)$$

5,	37,	61,	157,	193,
277,	397,	421,	457,	541,
613,	661,	673,	733,	757,
877,	997,	1093,	1153,	1201,
1213,	1237,	1381,	1453,	1621,
1657,	1873,	1933,	2017,	2137,
2341,	2557,	2593,	2797,	2857,
2917,	3061,	3217,	3253,	3517,
3733,	4021,	4057,	4177,	4261,
4357,	4441,	4561,	4621,	4933, ...

$$\mathbb{P}_2^+(n) = \{p \in \mathbb{P}(+1,+1) : p \leq n\} (\#\mathbb{P}_2^+(n) = 1800)$$

11,	167,	359,	2039,	4127,
4919,	5639,	5807,	5927,	6047,
7247,	7559,	7607,	7727,	10799,
13799,	13967,	14159,	15287,	15647,
20327,	21767,	23399,	24407,	24527,
25799,	28607,	29399,	31607,	35879,
36887,	42359,	42767,	42839,	44687,
45887,	46199,	47207,	47639,	48407,
48479,	51287,	51839,	52919,	56039,
56999,	57287,	58679,	58967,	60647, ...

$$\mathbb{P}_2^-(n) = \{p \in \mathbb{P}(-1,-1) : p \leq n\} (\#\mathbb{P}_2^-(n) = 1807)$$

5,	73,	313,	1321,	1753,
1993,	2473,	3313,	4273,	5113,
6121,	8353,	8521,	8713,	9241,
11161,	12073,	12433,	12721,	15073,
16633,	16921,	18553,	19441,	19801,
21673,	22993,	24841,	26833,	27361,
29473,	29641,	30241,	30553,	32833,
34513,	36433,	36793,	37273,	38953,
41281,	43321,	43633,	47353,	48193,
54673,	56113,	61153,	61561,	62473, ...

$$\mathbb{P}_3^+(n) = \{p \in \mathbb{P}(+1,+1,+1) : p \leq n\} (\#\mathbb{P}_3^+(n) = 131)$$

23,	719,	4079,	9839,	11279,
21599,	28319,	51599,	84719,	92399,
95279,	96959,	137279,	157679,	159119,
178799,	209519,	219839,	243119,	349199,
429119,	430799,	441839,	462719,	481199,
491279,	507359,	533999,	571199,	597599,

602639, 659999, 700319, 786959, 811199,
 834959, 903359, 976799, 1017119, 1115519,
 1148879, 1190639, 1333919, 1341839, 1342079,
 1402799, 1489199, 1519439, 1574159, 1650959, ...

$$\mathbb{P}_3^-(n) = \{p \in \mathbb{P}(-1, -1, -1) : p \leq n\} \ (\#\mathbb{P}_3^-(n) = 123)$$

12241, 17041, 18481, 49681, 54721,
 59281, 82561, 123121, 133201, 144961,
 180241, 184081, 195121, 237361, 254161,
 282481, 331921, 354961, 371281, 436801,
 450481, 463921, 482641, 491041, 541201,
 624241, 667441, 683041, 775441, 798961,
 846961, 875761, 1008001, 1077841, 1160161,
 1201201, 1253521, 1299601, 1322641, 1327201,
 1366801, 1481041, 1596961, 1641361, 1675441,
 1702801, 1795201, 1804801, 1809601, 1849921, ...

$$\mathbb{P}_4^+(n) = \{p \in \mathbb{P}(+1, +1, +1, +1) : p \leq n\} \ (\#\mathbb{P}_4^+(n) = 18)$$

47, 1439, 858239,
 861599, 982559, 1014719,
 1067999, 2034239, 2297759,
 2683679, 2978399, 3301919,
 4288799, 4737599, 5454719,
 6484319, 6753119, 7145759.

$$\mathbb{P}_4^-(n) = \{p \in \mathbb{P}(-1, -1, -1, -1) : p \leq n\} \ (\#\mathbb{P}_4^-(n) = 24)$$

24481, 109441, 246241,
 266401, 709921, 927841,
 1334881, 1693921, 2320321,
 2402401, 2654401, 3193921,
 3350881, 3405601, 3699841,
 3880801, 4660321, 5114881,
 5538241, 6039841, 6106081,
 6169441, 7131841, 7222561.

$$\mathbb{P}_5^+(n) = \{p \in \mathbb{P}(+1, +1, +1, +1, +1) : p \leq n\} \ (\#\mathbb{P}_5^+(n) = 3)$$

2879, 2029439, 4068479.

$$\mathbb{P}_5^-(n) = \{p \in \mathbb{P}(-1, -1, -1, -1, -1) : p \leq n\} \ (\#\mathbb{P}_5^-(n) = 2)$$

532801, 5308801.

$$\mathbb{P}_6^-(n) = \{p \in \mathbb{P}(-1, -1, -1, -1, -1, -1) : p \leq n\} \quad (\#\mathbb{P}_6^-(n) = 1)$$

$$1065601.$$

$$\mathbb{P}(-1, +1)(n) = \{p \in \mathbb{P}(-1, +1) : p \leq n\} \quad (\#\mathbb{P}(-1, +1)(n) = 0)$$

$$\mathbb{P}(-1, +1, -1)(n) = \{p \in \mathbb{P}(-1, +1, -1) : p \leq n\} \quad (\#\mathbb{P}(-1, +1, -1)(n) = 1)$$

$$13.$$

Observación 4.15 En la literatura existen cadenas de primos de diversas clases: las cadenas de Cunningham, las cadenas de Shanks, etc. La que está relacionada con nuestro concepto de cadena de primos seguros es la de Cunningham.

Una cadena de Cunningham (véase por ejemplo [49, A7], [116]) es una sucesión de $k \geq 2$ primos p_1, \dots, p_k tales que $p_{i+1} = 2p_i + \varepsilon$, $i = 1, \dots, k-1$, $\varepsilon \in \{-1, +1\}$. Nótese que si $k = 2$, entonces una cadena de Cunningham de longitud 2 no es más que un par $(q, 2q+1)$, donde q es un primo de Sophie Germain (ver la sección 4.1.2).

Ejemplo 4.16 Para $k = 5$ y $\varepsilon = 1$ una cadena de Cunningham es 2, 5, 11, 23, 47, que da lugar a la cadena de primos seguros de longitud $k = 5$ y signatura $\varepsilon = 1$ siguiente: 47, 23, 11, 5, 2; esto es, $47 \in \mathbb{P}_4^+$.

Observación 4.17 No es conocido si existen cadenas de primos seguros de longitud k arbitraria, o equivalentemente si para todo $k \in \mathbb{N}$ los conjuntos $\mathbb{P}_k^+, \mathbb{P}_k^-$ son no vacíos. Samuel Yates [125] ha demostrado que la máxima longitud posible de una cadena de primos de Cunningham cuyo primer término es p y $\varepsilon = 1$, es $p - 1$. Esta es una cuestión importante en las aplicaciones a la Criptografía de clave pública donde se requiere generar primos seguros de longitud muy grande.

Los récords más recientes de cadenas de Cunningham han sido obtenidos por Forbes en [42]. Para $\varepsilon = 1$ la cadena más larga obtenida es de longitud $k = 14$, cuyo primer término es $p_1 = 23305436881717757909$, y para $\varepsilon = -1$ se tiene $k = 16$, con primer primo $p_1 = 3203000719597029781$.

4.2 Densidad de primos 1-seguros

4.2.1 Función recuento de primos

Definición 4.18 Sirviéndonos de la Notación 4.2, sea $\pi: [1, \infty) \rightarrow \mathbb{R}$ la función recuento definida por:

$$\pi(x) = \#\{p \in \mathbb{P} : p \leq x\}.$$

Esto es, $\pi(x)$ es el número de primos que no exceden el valor x .

Como es bien sabido (véanse, por ejemplo, [28, §1.1.5], [94]), los analistas Hadamard y La Vallée-Poussin demostraron en 1896, cada uno independientemente del otro, el llamado *teorema de los números primos*, que afirma:

$$\lim_{x \rightarrow \infty} \frac{\ln x}{x} \pi(x) = 1. \tag{4.1}$$

Con ello, se puede establecer que, aproximadamente, la función recuento de números primos es:

$$\pi(x) = \frac{x}{\ln x}. \quad (4.2)$$

Una aproximación aún mejor había sido ya conjeturada por Gauss, con la siguiente fórmula:

$$\pi(x) \sim \text{Li}(x) = \int_2^x \frac{1}{\ln t} dt. \quad (4.3)$$

De hecho, es fácil ver que la integral de Gauss se puede aproximar por el resultado de la fórmula (4.2). Vamos a presentar un argumento estadístico aproximado que permite justificar la fórmula (4.3) (véase [104]) a partir del teorema de los números primos (4.1) con ayuda del llamado *segundo teorema de Mertens* (véase, por ejemplo, [115, p. 21]), que éste demostró en 1874.

En primer lugar, observemos que la probabilidad de que un número entero cualquiera x sea divisible por p es obviamente $1/p$, pues cada p -ésimo número entero es divisible por p . Por lo tanto la probabilidad de que x no sea divisible por p será:

$$1 - \frac{1}{p}.$$

Por definición, un número cualquiera x es primo si y sólo si no es divisible por ninguno de sus posibles factores primos menores que él. En otras palabras, x es primo si para todo $p_i \leq \sqrt{x}$, se cumple que p_i no divide a x . Así pues, si suponemos que los sucesos “ser divisible por p_i ” para todos los $p_i \leq \sqrt{x}$ son estadísticamente independientes, podremos decir que, en primera aproximación, la probabilidad de que un entero cualquiera x sea primo viene dada por:

$$\prod_{p \leq \sqrt{x}} \left(1 - \frac{1}{p}\right), \quad p \in \mathbb{P}. \quad (4.4)$$

Sin embargo, para valores grandes de p , esta independencia no es del todo cierta. En efecto, el segundo teorema de Mertens demuestra que:

$$\prod_{p \leq x} \left(1 - \frac{1}{p}\right) = \frac{e^{-\gamma}}{\ln x} + O(1) \quad (x \geq 1), \quad (4.5)$$

donde γ representa la constante de Euler o de Mascheroni, que se define como:

$$\gamma = \lim_{n \rightarrow \infty} \left(1 + \frac{1}{2} + \cdots + \frac{1}{n} - \ln n\right) = 0,577215665\dots$$

Si en la ecuación (4.5) hacemos el cambio de variable $x \rightarrow \sqrt{x}$, podremos reescribirla como:

$$\prod_{p \leq \sqrt{x}} \left(1 - \frac{1}{p}\right) = \frac{2 \cdot e^{-\gamma}}{\ln x} + O(1) \quad (x \geq 1). \quad (4.6)$$

Reordenando términos y combinando con el teorema de los números primos, se puede afirmar que:

$$\frac{\pi(x)}{x} \sim \frac{1}{2} e^{-\gamma} \prod_{p \leq \sqrt{x}} \left(1 - \frac{1}{p}\right) \sim \frac{1}{\ln x}. \quad (4.7)$$

Este resultado nos impulsa a “corregir” el valor de la probabilidad obtenido en la ecuación (4.4), y denominarla $W(x)$, asignándole el valor:

$$W(x) = \frac{1}{2} e^\gamma \prod_{p \leq \sqrt{x}} \left(1 - \frac{1}{p}\right). \quad (4.8)$$

Más aún, como dice [95, p. 66]:

“La probabilidad de que un número entero grande x elegido al azar sea primo es $1/\ln x$. Por otro lado, el número x es primo si y solo si no es divisible por ninguno de los primos $\leq \sqrt{x}$. Por tanto, podría esperarse que

$$\prod_{2 \leq p \leq \sqrt{x}} \left(1 - \frac{1}{p}\right)$$

fuerá aproximadamente igual a $1/\ln x$ para x suficientemente grande, lo cual está claramente en discrepancia con el teorema de Mertens. Esta discrepancia tiene que surgir del hecho de que existen sutilezas en la distribución de los primos hasta \sqrt{x} , que influyen en los primos en el entorno de x y de las que no puede dar cuenta el sencillo modelo estadístico utilizado, en el que se considera independiente la divisibilidad por distintos primos. O, dicho de otro modo, la criba de Eratóstenes es especial en el sentido de que criba los números de manera más eficiente que una criba ‘aleatoria’. En efecto, cuando usamos la criba de Eratóstenes con todos los primos $p \leq \sqrt{x}$, el teorema de los números primos nos asegura que en un entorno de x quedarán $1/\ln x$ números sin cribar, mientras que usando la criba aleatoria, la fórmula de Mertens (4.5) nos dejaría la fracción $2e^{-\gamma}/\ln x \approx 1,123/\ln x$ números sin cribar, que es un poco más alta”.

Lo anterior justifica que aceptemos la fórmula (4.8) como función densidad de probabilidad, de modo que podemos estimar el número de primos menores o iguales que x por la siguiente expresión:

$$\pi(x) \sim \int_2^x W(t) dt = \int_2^x \frac{1}{\ln t} dt,$$

con lo que hemos alcanzado nuestro objetivo de justificar la fórmula de Gauss a partir del teorema de los números primos.

4.2.2 Funciones recuento para primos seguros

Definición 4.19 Sean $\pi_k^+, \pi_k^- : [1, \infty) \rightarrow \mathbb{R}$ las funciones recuento definidas por:

$$\begin{aligned} \pi_k^+(x) &= \# \{p \in \mathbb{P}_k^+ : p \leq x\}, \\ \pi_k^-(x) &= \# \{p \in \mathbb{P}_k^- : p \leq x\}. \end{aligned}$$

Esto es, $\pi_k^+(x)$ (resp. $\pi_k^-(x)$) es el número de primos de signatura $\varepsilon_1 = \dots = \varepsilon_k = +1$ (resp. $\varepsilon_1 = \dots = \varepsilon_k = -1$) que no exceden x .

Estamos interesados en extender la función recuento de los primos ordinarios a las funciones recuento para primos seguros. Hemos obtenido heurísticamente unas fórmulas asintóticas para esas funciones recuento. Dado el buen resultado obtenido con los presupuestos que nos permitieron llegar a la función recuento para primos ordinarios, extenderemos esos mismos razonamientos a las funciones recuento de primos seguros, $\pi_k^+(x)$ y $\pi_k^-(x)$.

4.2.3 Fórmula heurística asintótica para π_1^+

Comencemos con la siguiente

Notación 4.20 Denotaremos por $W_1^+(x)$ la probabilidad de que $x \in \mathbb{P}_1^+$.

Por definición, tenemos $\pi_1^+(x) = \#\{p \in \mathbb{P}_1^+ : p \leq x\}$, con $x \in [1, \infty)$. Ahora bien, $p \in \mathbb{P}_1^+$, significa, según la Definición 4.1, que $p = 2p_1 + 1$, con p_1 primo. Por consiguiente, para que un número cualquiera x esté en \mathbb{P}_1^+ , ha de verificar simultáneamente lo siguiente:

1. x es primo,
2. $x_1 = \frac{1}{2}(x - 1)$ también es primo.

El teorema de los números primos nos induce a pensar que la probabilidad de que un entero x elegido al azar sea primo es del orden de $1/\ln x$. Ahora bien, para que el número x sea primo ha de pasar el “test” del primo q , que podemos enunciar en el siguiente lema:

Lema 4.21 Un número x es primo si y sólo si para todo $q \in \mathbb{P}$, tal que $q \leq \sqrt{x}$, se verifica que $x \not\equiv 0 \pmod{q}$.

Demostración Es inmediata, a partir de la definición de primo. ■

Está claro entonces que la condición de ser primo se puede expresar diciendo que x no debe pertenecer a la clase 0 entre las q clases de equivalencia para cada primo q . Si suponemos que las distintas clases de equivalencia módulo el primo q están estadísticamente distribuidas según una distribución equiprobable, entonces podemos escribir que la probabilidad de que x sea primo es:

$$\prod_{q \leq \sqrt{x}} \frac{q-1}{q}. \quad (4.9)$$

Ahora bien, si se han de cumplir las dos condiciones exigidas para que x esté en \mathbb{P}_1^+ , entonces tiene que ocurrir simultáneamente lo siguiente:

1. $x \equiv 1 \pmod{2}$, es decir, x es un número impar;
2. $x \not\equiv 0 \pmod{q}$ y $x \not\equiv 1 \pmod{q}$ para todo $q \in \mathbb{P}$, tal que $q \leq \sqrt{x_1}$;
3. $x \not\equiv 0 \pmod{q}$, para todo $q \in \mathbb{P}$, tal que $\sqrt{x_1} < q \leq \sqrt{x}$.

La primera condición y la primera parte de la segunda condición son necesarias para que se cumpla que $x_1 = \frac{1}{2}(x - 1)$ sea primo; la segunda parte de la segunda condición y la tercera condición son necesarias para que lo sea x .

Vamos a calcular la probabilidad de que un número x cumpla cada una de las tres condiciones. Obviamente, la probabilidad de que se cumpla la primera condición es $\frac{1}{2}$. La probabilidad de que, para un q determinado, x verifique la segunda condición es

$$\frac{q-2}{q}.$$

Ahora bien, el caso $q = 2$ ha de ser considerado aparte porque anula la probabilidad. Así pues hemos de considerar el productorio desde $q = 3$ imponiendo que x_1 sea un número impar, lo que equivale a multiplicar el productorio por $\frac{1}{2}$. Aplicando a la tercera condición la fórmula (4.9), se concluye finalmente que la probabilidad de que x esté en \mathbb{P}_1^+ es

$$\frac{1}{4} \prod_{3 \leq q \leq \sqrt{x_1}} \frac{q-2}{q} \prod_{\sqrt{x_1} < q \leq \sqrt{x}} \frac{q-1}{q}.$$

Recordando la fórmula (4.7) y el razonamiento allí presentado, nos sentimos también aquí impulsados a corregir el valor propuesto aplicando dos veces, una por cada primo, el factor $\frac{1}{2}e^\gamma$. Con ello, conjeturamos que el valor de la probabilidad $W_1^+(x)$ es:

$$W_1^+(x) = \left(\frac{1}{2}e^\gamma\right)^2 \frac{1}{4} \prod_{3 \leq q \leq \sqrt{x_1}} \frac{q-2}{q} \prod_{\sqrt{x_1} < q \leq \sqrt{x}} \frac{q-1}{q}. \quad (4.10)$$

Vamos a tratar de obtener una expresión más compacta basándonos de nuevo en la fórmula de Mertens y en la constante de los primos gemelos. Transformemos cada productorio por separado. En cuanto al primero, se tiene:

$$\begin{aligned} \frac{e^\gamma}{2} \prod_{3 \leq q \leq \sqrt{x_1}} \frac{q-2}{q} &= \frac{e^\gamma}{2} \prod_{3 \leq q \leq \sqrt{x_1}} \frac{q-2}{q} \left(\frac{q-1}{q}\right)^2 \left(\frac{q}{q-1}\right)^2 \\ &\simeq \frac{2}{e^\gamma} \prod_{3 \leq q \leq \sqrt{x_1}} \left(1 - \frac{1}{(q-1)^2}\right) \frac{4}{(\ln x_1)^2}. \end{aligned} \quad (4.11)$$

En cuanto al segundo productorio, se tiene:

$$\begin{aligned} \frac{e^\gamma}{2} \prod_{\sqrt{x_1} < q \leq \sqrt{x}} \frac{q-1}{q} &= \frac{e^\gamma}{2} \frac{\prod_{2 \leq q \leq \sqrt{x}} \frac{q-1}{q}}{\prod_{2 \leq q \leq \sqrt{x_1}} \frac{q-1}{q}} \\ &\simeq \frac{e^\gamma \ln x_1}{2 \ln x}. \end{aligned}$$

Calculando ahora el producto total para obtener $W^+(x)$, tenemos:

$$\begin{aligned} W^+(x) &\simeq \frac{1}{4} \frac{e^\gamma}{2} \frac{2}{e^\gamma} \prod_{3 \leq q \leq \sqrt{x_1}} \left(1 - \frac{1}{(q-1)^2}\right) \frac{4}{(\ln x_1)^2} \frac{\ln x_1}{\ln x} \\ &= \prod_{3 \leq q \leq \sqrt{x_1}} \left(1 - \frac{1}{(q-1)^2}\right) \frac{1}{\ln x_1 \cdot \ln x} \end{aligned}$$

Aquí es donde entra en juego la constante C de los primos gemelos. Con este nombre se conoce el siguiente productorio infinito:

$$C = 2 \prod_{p \geq 3} \left(1 - \frac{1}{(p-1)^2} \right) \simeq 1,320323.$$

Para una explicación más detallada acerca de esta constante se pueden consultar, por ejemplo, [94, p. 147] o [115, p. 123]. Teniendo en cuenta que cada término de este productorio es menor que la unidad, está claro que se tiene

$$\prod_{p \geq 3} \left(1 - \frac{1}{(p-1)^2} \right) \leq \prod_{3 \leq p \leq \sqrt{x_1}} \left(1 - \frac{1}{(p-1)^2} \right).$$

Pero, obviamente,

$$\lim_{x \rightarrow \infty} \frac{\prod_{3 \leq p \leq \sqrt{x_1}} \left(1 - \frac{1}{(p-1)^2} \right)}{\prod_{p \geq 3} \left(1 - \frac{1}{(p-1)^2} \right)} = 1,$$

con lo que es claro que, para valores suficientemente grandes de x_1 se puede aproximar el valor

$$\prod_{3 \leq q \leq \sqrt{x_1}} \left(1 - \frac{1}{(q-1)^2} \right)$$

por la constante

$$\frac{C}{2} = \prod_{p \geq 3} \left(1 - \frac{1}{(p-1)^2} \right).$$

En efecto, ya para todo $x_1 > 1000$, se tiene:

$$\frac{\prod_{3 \leq p \leq \sqrt{x_1}} \left(1 - \frac{1}{(p-1)^2} \right)}{\prod_{p \geq 3} \left(1 - \frac{1}{(p-1)^2} \right)} = \frac{1}{\prod_{p > \sqrt{x_1}} \left(1 - \frac{1}{(p-1)^2} \right)} \simeq 1,0064,$$

valor muy próximo a la unidad.

Resumiendo todo el razonamiento: podemos aproximar con bastante exactitud la probabilidad de que un número cualquiera x esté en \mathbb{P}_1^+ por:

$$W_1^+(x) \simeq \frac{C}{2} \frac{1}{\ln x \cdot \ln \frac{x-1}{2}}. \quad (4.12)$$

Si, como antes en la sección 4.2.1, aceptamos ahora este valor como densidad de probabilidad de que $x \in \mathbb{P}_1^+$, nos quedará que la función recuento para primos 1-seguros de firma positiva es

$$\pi_1^+(x) \simeq \int_5^x W_1^+(t) dt = \frac{C}{2} \int_5^x \frac{dt}{\ln t \cdot \ln \frac{t-1}{2}}. \quad (4.13)$$

Observación 4.22 Hemos evaluado el número exacto de primos 1-seguros hasta 2^{31} utilizando la criba de Eratóstenes. Para el rango comprendido entre 2^{31} y 2^{32} , nos hemos servido del test probabilístico de primalidad de Miller-Rabin con un parámetro de seguridad $t = 10$, que asegura una probabilidad de éxito del algoritmo de

$$1 - 0,25^{10} = 0,99999904632568359375.$$

Combinando estos dos métodos, hemos obtenido la lista exacta de primos 1-seguros p en el rango $5 \leq p < 2^{32}$.

Observación 4.23 Se puede comprobar experimentalmente que la relación

$$B_1 = \frac{\pi_1^+(x)}{\int_5^x (\ln t \cdot \ln \frac{t-1}{2})^{-1} dt}$$

entre el valor exacto de $\pi_1^+(x)$ y la integral aproximada tiende muy rápidamente a $C/2$. Utilizando la tabla, cuya confección explicamos en la Observación 4.22, evaluamos el número exacto de primos 1-seguros $\leq n$ con $n \in \mathbb{N}$ en el rango $5 \leq n < 4 \cdot 10^9$; calculando por métodos numéricos la integral anterior, se obtiene la gráfica de la figura 4.1, donde se ha representado los valores de la relación B_1 en función de n . Es muy notable la rapidez con que estabiliza hacia el valor $C/2 \simeq 0,66$.

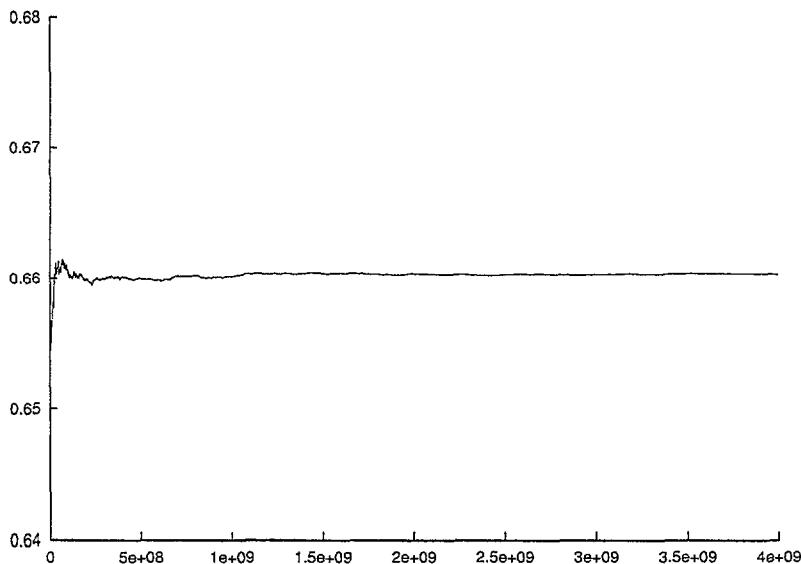


Figura 4.1: Valor de la constante B_1

Observación 4.24 Es de notar que el establecimiento riguroso del resultado anterior (esto es, que el segundo miembro representa realmente un desarrollo asintótico del primero) supondría una demostración de la existencia de infinitos primos 1-seguros, ya que la integral del segundo miembro tiende a infinito. En efecto, se tiene:

$$\frac{C}{2} \int_5^x \frac{dt}{\ln t \cdot \ln \frac{t-1}{2}} \geq \frac{C}{2} \int_5^x \frac{dt}{(\ln t)t} = \frac{C}{2} (\ln \ln x - \ln \ln 5).$$

Observación 4.25 La existencia de infinitos primos 1-seguros no ha sido establecida todavía rigurosamente, si bien los records obtenidos apoyan la veracidad de esta hipótesis. Recuérdense, a este respecto, los grandes tamaños de los primos seguros presentados al principio de la sección 4.1.2.

Observación 4.26 La hipótesis de la infinitud de primos 1-seguros no es trivial, ya que su densidad en el conjunto de los primos ordinarios tiende a cero. En efecto, en virtud del teorema de los números primos, se tiene:

$$\lim_{x \rightarrow \infty} \frac{\ln x}{x} \pi(x) = 1.$$

Por tanto,

$$\lim_{x \rightarrow \infty} \frac{\pi_1^+(x)}{\pi(x)} = 0.$$

Observación 4.27 En la referencia [37] se pueden encontrar publicados algunos resultados parciales acerca de la densidad de los primos 1-seguros.

4.2.4 Fórmula heurística asintótica para π_1^-

Notación 4.28 Denotaremos por $W_1^-(x)$ la probabilidad de que $x \in \mathbb{P}_1^-$.

El trabajo ahora consiste en replicar el mismo esquema usado para la función recuento π_1^+ . En efecto, igual que antes, por definición,

$$\pi_1^-(x) = \# \{ p \in \mathbb{P}_1^- : p \leq x \},$$

con $x \in [1, \infty)$. Ahora bien, si $p \in \mathbb{P}_1^-$, significa, según la Definición 4.1, que $p = 2p_1 - 1$, con p_1 primo. Por consiguiente, para que un número cualquiera x esté en \mathbb{P}_1^- , ha de verificar simultáneamente que:

1. x es un primo,
2. $x_1 = \frac{1}{2}(x + 1)$ también es primo.

Ahora bien, si se han de cumplir las dos condiciones exigidas para que x esté en \mathbb{P}_1^- , entonces tiene que ocurrir simultáneamente lo siguiente:

1. $x \equiv 1 \pmod{2}$, es decir, x es un número impar;
2. $x \not\equiv 0 \pmod{q}$ y $x \not\equiv -1 \pmod{q}$ para todo $q \in \mathbb{P}$, tal que $q \leq \sqrt{x_1}$;
3. $x \not\equiv 0 \pmod{q}$, para todo $q \in \mathbb{P}$, tal que $\sqrt{x_1} < q \leq \sqrt{x}$.

La primera condición y la primera parte de la segunda condición son necesarias para que se cumpla que $x_1 = \frac{1}{2}(x + 1)$ sea primo; la segunda parte de la segunda condición y la tercera condición son necesarias para que lo sea x .

Razonando de forma completamente análoga al caso π_1^+ , podemos concluir que la probabilidad de que x esté en \mathbb{P}_1^- es también la misma, es decir,

$$W_1^-(x) \simeq \prod_{3 \leq q \leq \sqrt{x_1}} \left(1 - \frac{1}{(q-1)^2}\right) \frac{1}{\ln x_1 \cdot \ln x},$$

de donde

$$W_1^-(x) \simeq \frac{C}{2} \frac{1}{\ln x \cdot \ln \frac{x+1}{2}},$$

Consideraremos, como antes, que $W_1^-(x)$ es la función densidad de probabilidad de que $x \in \mathbb{P}_1^-$, con lo que quedará para la función recuento de los primos 1-seguros de signatura negativa que:

$$\pi_1^-(x) \simeq \int_3^x W_1^-(t) dt = \frac{C}{2} \int_3^x \frac{dt}{\ln t \cdot \ln \frac{t+1}{2}}.$$

Este resultado da cuenta y razón de los resultados experimentales obtenidos en las tablas del Ejemplo 4.14 en donde se ponía de manifiesto que el número de primos 1-seguros de signatura positiva era sensiblemente igual al de primos 1-seguros de signatura negativa. Recordemos, por ejemplo, que se tiene

$$\begin{aligned} \#\mathbb{P}_1^+(7500000) &= 21970, \\ \#\mathbb{P}_1^-(7500000) &= 21966, \end{aligned}$$

de modo que sólo se diferencian en 4 unidades.

4.2.5 El trabajo de Y. Cai

Y. Cai realizó un trabajo en 1994 (véase la referencia [21]) en donde conjetura una distribución para los números primos seguros utilizando para ello el método que el profesor C. Pan ya había usado para tratar el tema de la Conjetura de Goldbach en [84]. En realidad, los primos que este autor denomina seguros son los conocidos como primos de Sophie Germain, que definimos en la sección 4.1.2.

Antes de exponer el trabajo, comencemos con las dos siguientes definiciones:

Definición 4.29 La función de Von Mangoldt, $\Lambda : \mathbb{N} \rightarrow \mathbb{R}$, viene dada por

$$\Lambda(n) = \begin{cases} \ln p, & \text{si } n = p^\nu, \text{ con } \nu \geq 1 \text{ y } p \text{ primo,} \\ 0, & \text{en el resto de casos.} \end{cases}$$

Definición 4.30 La función de Möbius está dada por

1. $\mu(1) = 1$,
2. y si $n > 1$, y $n = p_1^{a_1} \cdots p_k^{a_k}$ es su descomposición en factores primos, entonces
 - a) $\mu(n) = (-1)^k$, si $a_1 = a_2 = \dots = a_k = 1$,
 - b) $\mu(n) = 0$, en cualquier otro caso.

Para ver propiedades interesantes de la función de Möbius, véase por ejemplo [3, §2.7].

Siguiendo su nomenclatura, transcribimos las definiciones que el autor presenta inicialmente:

$$S(x) = \sum_{n \leq x} \Lambda(n) \Lambda(2n+1),$$

$$Q(x) = \frac{x^{\frac{1}{2}}}{(\ln x)^{20}},$$

$$\sigma = 2 \prod_{p>2} \left(1 - \frac{1}{(p-1)^2} \right).$$

Observemos que la constante σ es la constante de los primos gemelos, a la que nosotros denominamos C a lo largo de nuestro trabajo. El autor afirma que la función $S(x)$ así definida es esencialmente el número de primos de Sophie Germain que no exceden x . Aplicando entonces el método del círculo (véase, por ejemplo, [6]), demuestra que

$$S(x) = \sigma x + R(x),$$

donde

$$R(x) = R_1(x) + R_2(x) + R_3(x) + O\left(\frac{x}{\ln x}\right)$$

y

$$R_1(x) = \sum_{n \leq x} \left(\sum_{\substack{d_1|n \\ d_1 \leq Q(x)}} a(d_1) \right) \left(\sum_{\substack{d_2|2n+1 \\ d_2 > Q(x)}} a(d_2) \right),$$

$$R_2(x) = \sum_{n \leq x} \left(\sum_{\substack{d_1|n \\ d_1 > Q(x)}} a(d_1) \right) \left(\sum_{\substack{d_2|2n+1 \\ d_2 \leq Q(x)}} a(d_2) \right),$$

$$R_3(x) = \sum_{n \leq x} \left(\sum_{\substack{d_1|n \\ d_1 > Q(x)}} a(d_1) \right) \left(\sum_{\substack{d_2|2n+1 \\ d_2 > Q(x)}} a(d_2) \right),$$

con

$$a(m) = -\mu(m) \ln m,$$

donde μ en la función de Möbius (véase Definición 4.30). El autor demuestra que

$$R_1(x) = O\left(\frac{x}{\ln x}\right),$$

$$R_2(x) = O\left(\frac{x}{\ln x}\right).$$

Sin embargo, $R_3(x)$ resulta difícil de estimar y el autor conjetura que se verifica:

$$R_3(x) = O\left(\frac{x}{\ln x}\right).$$

Por ello, el resultado final es también una conjetura; el autor propone que el número de primos de Sophie Germain que no exceden x vale

$$\pi_s(x) = \frac{S(x)}{(\ln x)^2} \left(1 + O\left(\frac{\ln^2 x}{\ln x}\right) \right) + O\left(\frac{x}{(\ln x)^3}\right).$$

Concluye afirmando, sin ulteriores demostraciones, que una conjetura más afinada es la siguiente:

$$\pi_s(x) = \sigma \frac{x}{(\ln x)^2} + O\left(\frac{x \ln^2 x}{(\ln x)^3}\right). \quad (4.14)$$

Observación 4.31 Observemos que si q es un primo de Sophie Germain, ello implica que $p = 2q + 1$ es primo, es decir, p es un primo 1-seguro según la definición ordinaria y, por tanto, el número de primos de Sophie Germain que no exceden q son exactamente el número de primos 1-seguros que no exceden p . Puesto que $p \simeq 2q$, se debe cumplir que $\pi_s(x) \simeq \pi_1^+(2x)$.

Este resultado concuerda muy bien con el obtenido por nosotros. En efecto, de la ecuación (4.13), se tiene

$$\pi_1^+(x) \simeq \frac{C}{2} \int_5^x \frac{dt}{\ln t \cdot \ln \frac{t-1}{2}} \simeq \frac{C}{2} \frac{x}{\ln x \cdot \ln \frac{x-1}{2}} \simeq \frac{C}{2} \frac{x}{(\ln x)^2}.$$

En efecto, con esta aproximación, se tiene

$$\pi_1^+(2x) \simeq C \frac{x}{(\ln 2x)^2} = C \frac{x}{(\ln x + \ln 2)^2} \simeq C \frac{x}{(\ln x)^2}.$$

Comparando con la ecuación (4.14), y recordando que $C = \sigma$, se ve que $\pi_s(x) \simeq \pi_1^+(2x)$.

4.3 Densidad de primos 2-seguros

4.3.1 Fórmula heurística asintótica para π_2^+

Comencemos con la siguiente

Notación 4.32 Denotaremos por $W_2^+(x)$ la probabilidad de que $x \in \mathbb{P}_2^+$.

Por definición, tenemos $\pi_2^+(x) = \# \{p \in \mathbb{P}_2^+ : p \leq x\}$, con $x \in [11, \infty)$. Ahora bien, $p \in \mathbb{P}_2^+$, significa, según la Definición 4.1, que $p = 2p_1 + 1$, con p_1 primo y además $p_1 = 2p_2 + 1$, con p_2 también primo. Por consiguiente, para que un número cualquiera x esté en \mathbb{P}_2^+ , ha de verificar simultáneamente lo siguiente:

1. x es primo,
2. $x_1 = \frac{1}{2}(x - 1)$ es primo, y
3. $x_2 = \frac{1}{2}(x_1 - 1)$ también es primo.

Argumentando de manera análoga al caso de los 1-seguros, podemos ahora expresar las condiciones para que x esté en \mathbb{P}_2^+ en forma de los siguientes sucesos que se han de cumplir simultáneamente:

1. $x \equiv 3 \pmod{4}$;
2. $x \not\equiv 0 \pmod{q}$, $x \not\equiv 1 \pmod{q}$, $x \not\equiv 3 \pmod{q}$, para todo $q \in \mathbb{P}$, tal que $q \leq \sqrt{x_2}$;
3. $x \not\equiv 0 \pmod{q}$, $x \not\equiv 1 \pmod{q}$, con $\sqrt{x_2} < q \leq \sqrt{x_1}$, $q \in \mathbb{P}$;
4. $x \not\equiv 0 \pmod{q}$, para todo $q \in \mathbb{P}$, tal que $\sqrt{x_1} < q \leq \sqrt{x}$.

La primera condición se sigue del hecho de que $x = 2x_1 + 1 = 4x_2 + 3$. La primera parte de las restantes condiciones nos garantiza que x sea primo; la segunda parte de la segunda y tercera condiciones nos garantizan que x_1 sea primo; y por último, la tercera parte de la segunda condición nos asegura que también lo sea x_2 . Tratemos ahora de calcular la probabilidad de que cada una de estas condiciones se cumpla.

Obviamente, la probabilidad de que se cumpla la primera condición es $\frac{1}{4}$. La probabilidad de que, para un q determinado, x verifique la segunda condición es

$$\frac{q-3}{q}, \quad (4.15)$$

puesto que ha de estar excluido de 3 clases de equivalencia para cada primo q . Por tanto, para calcular la probabilidad de que x verifique la segunda condición hay que efectuar el producto de las fracciones (4.15) para todos los primos $q \leq \sqrt{x_2}$. Los casos $q = 2$ y $q = 3$ han de ser considerados aparte porque hacen negativa o anulan la fracción (4.15). Así pues, hay que efectuar el productorio desde $q = 5$ imponiendo además que ninguno de los x, x_1, x_2 sea múltiplo de 2 ni de 3.

Lema 4.33 Para que ninguno de los x, x_1, x_2 sea múltiplo de 2 ni de 3, es necesario y suficiente tomar x_2 impar y $x_2 \equiv 2 \pmod{3}$.

Demostración Si x, x_1, x_2 no son múltiplos de 2 ni de 3, obviamente, x_2 es impar. Ahora bien si $x_2 \equiv 1 \pmod{3}$, o sea, $x_2 = 1 + 3k$, entonces $x_1 = 3(k+1)$; luego x_1 sería divisible por 3, contra la hipótesis.

Recíprocamente, si x_2 es impar, ciertamente x_1 y x también lo son, y si $x_2 \equiv 2 \pmod{3}$, o sea, $x_2 = 2 + 3k$, entonces $x_1 = 5 + 6k$; luego $x_1 \equiv 2 \pmod{3}$, de donde x_1 no es divisible por 3. Análogamente se comprueba para x . ■

Por último, las condiciones tercera y cuarta son las mismas que se han considerado en el caso \mathbb{P}_1^+ . Por consiguiente, concluimos que la probabilidad de que x esté en \mathbb{P}_2^+ es

$$\frac{1}{2} \cdot \frac{1}{3} \cdot \frac{1}{4} \prod_{5 \leq q \leq \sqrt{x_2}} \frac{q-3}{q} \prod_{\sqrt{x_2} < q \leq \sqrt{x_1}} \frac{q-2}{q} \prod_{\sqrt{x_1} < q \leq \sqrt{x}} \frac{q-1}{q}.$$

Recordando la fórmula (4.7) y el razonamiento allí presentado, nos sentimos también aquí impulsados a corregir el valor propuesto aplicando ahora tres veces,

una por cada primo, el factor $\frac{1}{2}e^\gamma$. Con ello, conjeturamos que el valor de la probabilidad $W_2^+(x)$ es:

$$W_2^+(x) = \left(\frac{1}{2}e^\gamma\right)^3 \frac{1}{24} \prod_{5 \leq q \leq \sqrt{x_2}} \frac{q-3}{q} \prod_{\sqrt{x_2} < q \leq \sqrt{x_1}} \frac{q-2}{q} \prod_{\sqrt{x_1} < q \leq \sqrt{x}} \frac{q-1}{q}. \quad (4.16)$$

Acudiendo a la fórmula de Mertens, podemos ir reduciendo cada productorio por separado. En cuanto al primero,

$$\begin{aligned} \frac{e^\gamma}{2} \prod_{5 \leq q \leq \sqrt{x_2}} \frac{q-3}{q} &= \frac{e^\gamma}{2} \prod_{5 \leq q \leq \sqrt{x_1}} \frac{q-3}{q} \left(\frac{q-1}{q}\right)^3 \left(\frac{q}{q-1}\right)^3 \\ &\simeq \left(\frac{2}{e^\gamma}\right)^2 \left(\frac{2}{1} \frac{3}{2}\right)^3 \frac{1}{(\ln x_2)^3} \prod_{5 \leq q \leq \sqrt{x_2}} \frac{q^2(q-3)}{(q-1)^3}. \end{aligned}$$

En cuanto al segundo productorio, se tiene:

$$\begin{aligned} \frac{e^\gamma}{2} \prod_{\sqrt{x_2} < q \leq \sqrt{x_1}} \frac{q-2}{q} &= \frac{\frac{e^\gamma}{2} \prod_{3 \leq q \leq \sqrt{x_1}} \frac{q-2}{q}}{\prod_{3 \leq q \leq \sqrt{x_2}} \frac{q-2}{q}} \\ &\simeq \frac{e^\gamma}{2} \frac{\prod_{3 \leq q \leq \sqrt{x_1}} \frac{q(q-2)}{(q-1)^2} \frac{4}{(\ln x_1)^2}}{\prod_{3 \leq q \leq \sqrt{x_2}} \frac{q(q-2)}{(q-1)^2} \frac{4}{(\ln x_2)^2}} \\ &= \frac{e^\gamma}{2} \prod_{\sqrt{x_2} < q \leq \sqrt{x_1}} \frac{q(q-2)}{(q-1)^2} \frac{(\ln x_2)^2}{(\ln x_1)^2}. \end{aligned}$$

En cuanto al tercer productorio, se tiene:

$$\begin{aligned} \frac{e^\gamma}{2} \prod_{\sqrt{x_1} < q \leq \sqrt{x}} \frac{q-1}{q} &= \frac{e^\gamma}{2} \frac{\prod_{2 \leq q \leq \sqrt{x}} \frac{q-1}{q}}{\prod_{2 \leq q \leq \sqrt{x_1}} \frac{q-1}{q}} \\ &\simeq \frac{e^\gamma}{2} \frac{\ln x_1}{\ln x} \end{aligned}$$

Calculando ahora el producto total para obtener $W_2^+(x)$, tenemos:

$$\begin{aligned} W_2^+(x) &\simeq \frac{1}{24} \left(\frac{2}{e^\gamma}\right)^2 \left(\frac{2}{1} \frac{3}{2}\right)^3 \frac{1}{(\ln x_2)^3} \prod_{5 \leq q \leq \sqrt{x_2}} \frac{q^2(q-3)}{(q-1)^3} \\ &\quad \frac{e^\gamma}{2} \prod_{\sqrt{x_2} < q \leq \sqrt{x_1}} \frac{q(q-2)}{(q-1)^2} \frac{(\ln x_2)^2}{(\ln x_1)^2} \cdot \frac{e^\gamma}{2} \frac{\ln x_1}{\ln x} \\ &= \frac{9}{8} \prod_{5 \leq q \leq \sqrt{x_2}} \frac{q^2(q-3)}{(q-1)^3} \prod_{\sqrt{x_2} < q \leq \sqrt{x_1}} \frac{q(q-2)}{(q-1)^2} \frac{1}{\ln x \ln x_1 \ln x_2}. \end{aligned}$$

Como

$$\lim_{x_2 \rightarrow \infty} \frac{\prod_{5 \leq q \leq \sqrt{x_2}} \frac{q^2(q-3)}{(q-1)^3}}{\prod_{5 \leq q} \frac{q^2(q-3)}{(q-1)^3}} = 1,$$

es claro que, para valores de x suficientemente grandes, el productorio

$$\prod_{5 \leq q \leq \sqrt{x_2}} \frac{q^2(q-3)}{(q-1)^3}$$

puede aproximarse por

$$C_2 = \prod_{q \geq 5} \frac{q^2(q-3)}{(q-1)^3}. \quad (4.17)$$

Por lo que respecta al segundo productorio, se tiene

$$\prod_{\sqrt{x_2} < q \leq \sqrt{x_1}} \frac{q(q-2)}{(q-1)^2} = \frac{\prod_{3 \leq q \leq \sqrt{x_1}} \frac{q(q-2)}{(q-1)^2}}{\prod_{3 \leq q \leq \sqrt{x_2}} \frac{q(q-2)}{(q-1)^2}}.$$

Por lo tanto, para valores suficientemente grandes de x , este productorio puede aproximarse por 1. Luego

$$W_2^+(x) \simeq \frac{9}{8} \frac{C_2}{\ln x \ln \frac{x-1}{2} \ln \frac{x-3}{4}}. \quad (4.18)$$

Así pues, el problema queda reducido básicamente al cálculo del valor numérico de la constante C_2 de (4.17). Para ello, necesitaremos algunas definiciones.

Cálculo de la constante C_2

Se tiene:

$$\ln C_2 = \sum_{q \geq 5} \left\{ \ln \left(1 - \frac{3}{q} \right) - 3 \ln \left(1 - \frac{1}{q} \right) \right\}. \quad (4.19)$$

Utilizando el desarrollo asintótico del logaritmo neperiano, válido para $|x| < 1$, a saber,

$$\ln(1-x) = - \left(x + \frac{x^2}{2} + \frac{x^3}{3} + \dots \right), \quad (4.20)$$

y sustituyéndolo en la fórmula (4.19), se sigue

$$\begin{aligned} \ln C_2 &= \sum_{n=1}^{\infty} \sum_{q \geq 5} \frac{3 - 3^n}{n} \frac{1}{q^n} \\ &= - \sum_{n=2}^{\infty} \frac{3^n - 3}{n} \sum_{q \geq 5} q^{-n}. \end{aligned}$$

Consideremos ahora la llamada función ‘zeta de los primos’, que se define como sigue:

$$P(n) = \sum_{p \in \mathbb{P}} p^{-n}.$$

Está claro, entonces, que se verifica:

$$\ln C_2 = - \sum_{n=2}^{\infty} \frac{3^n - 3}{n} (P(n) - 2^{-n} - 3^{-n}),$$

con lo que nos interesa poder calcular $P(n)$. Aquí entra en juego la función ζ de Riemann, que está relacionada con ella, y se define como:

$$\begin{aligned} \zeta(s) &= \sum_{n=1}^{\infty} \frac{1}{n^s} \\ &= \prod_{p \in \mathbb{P}} \frac{1}{1 - p^{-s}}. \end{aligned}$$

La segunda igualdad no es inmediata; de hecho, fue descubierta por Euler.

Tomando logaritmos y usando de nuevo (4.20), se tiene

$$\begin{aligned} \ln \zeta(n) &= - \sum_{p \in \mathbb{P}} \ln(1 - p^{-n}) \\ &= \sum_{p \in \mathbb{P}} \sum_{k=1}^{\infty} \frac{p^{-kn}}{k} \\ &= \sum_{k=1}^{\infty} \frac{1}{k} \sum_{p \in \mathbb{P}} p^{-kn} \\ &= \sum_{k=1}^{\infty} \frac{P(kn)}{k}. \end{aligned}$$

De acuerdo con [95, p. 65], la fórmula de inversión de Möbius nos da

$$P(n) = \sum_{k=1}^{\infty} \frac{\mu(k)}{k} \ln \zeta(kn).$$

Aquí, $\mu(k)$ es la función de Möbius de la Definición 4.30. La función ζ es bien conocida; de hecho, existen fórmulas explícitas para argumentos enteros. Para $k \geq 2$ y par, se tiene:

$$\zeta(k) = \frac{|B_k| 2^{k-1} \pi^k}{k!}.$$

Para argumentos impares, se han de distinguir dos casos; si $k \equiv 3 \pmod{4}$, entonces

$$\begin{aligned} \zeta(k) &= \frac{2^{k-1} \pi^k}{(k+1)!} \sum_{0 \leq n \leq \frac{k+1}{2}} (-1)^{n-1} \binom{k+1}{2n} B_{k+1-2n} B_{2n} \\ &\quad - 2 \sum_{n \geq 1} \frac{1}{n^k (e^{2\pi n} - 1)}. \end{aligned}$$

En el caso $k \equiv 1 \pmod{4}$, $k \geq 5$, se tiene:

$$\zeta(k) = \frac{(2\pi)^k}{(k+1)!(k-1)} \sum_{0 \leq n \leq \frac{k-1}{4}} (-1)^n (k+1-4n) \binom{k+1}{2n} B_{k+1-2n} B_{2n} \\ - 2 \sum_{n \geq 1} \frac{e^{2\pi n}(1+4\pi n/(k-1)) - 1}{n^k (e^{2\pi n} - 1)^2}.$$

En estas fórmulas, B_n representa el n -ésimo número de Bernoulli; reciben este nombre los coeficientes del desarrollo en serie de Taylor de la función $x/(e^x - 1)$ (véase [25, 95, p. 45]). Gracias a estas fórmulas, se puede calcular $P(n)$ con una aproximación varias decenas de cifras decimales y, por tanto, también la constante C_2 .

Si se desea menos aproximación, se puede acudir a un método más directo, procediendo numéricamente a partir de la definición de C_2 dada en la fórmula (4.17), y usando MAPLE. En efecto, sea (p_n) la sucesión de los números primos. Si

$$C_2(n) = \prod_{q=p_3}^{p_n} \frac{q^2(q-3)}{(q-1)^3},$$

entonces

$$\lim_{n \rightarrow \infty} C_2(n) = C_2.$$

Pues bien, mediante el programa de MAPLE que se adjunta en el Apéndice I, hemos calculado la siguiente tabla de valores de $C_2(n)$ con $3 \leq n \leq 100000$, que proporciona un valor de $C_2(n)$ cada vez que n es múltiplo de 1000:

n	1000	2000	3000	4000	5000
$C_2(n)$	0,6351905636	0,6351765810	0,6351725732	0,6351707403	0,6351696992
n	6000	7000	8000	9000	10000
$C_2(n)$	0,6351690367	0,6351685904	0,6351682555	0,6351680058	0,6351678147
n	11000	12000	13000	14000	15000
$C_2(n)$	0,6351676574	0,6351675260	0,6351674156	0,6351673332	0,6351672537
n	16000	17000	18000	19000	20000
$C_2(n)$	0,6351671912	0,6351671329	0,6351670824	0,6351670382	0,6351669969
n	21000	22000	23000	24000	25000
$C_2(n)$	0,6351669599	0,6351669285	0,6351668920	0,6351668685	0,6351668438
n	26000	27000	28000	29000	30000
$C_2(n)$	0,6351668171	0,6351668011	0,6351667598	0,6351667116	0,6351666724
n	31000	32000	33000	34000	35000
$C_2(n)$	0,6351666324	0,6351665953	0,6351665574	0,6351665197	0,6351664880
n	36000	37000	38000	39000	40000
$C_2(n)$	0,6351664608	0,6351664312	0,6351664069	0,6351663927	0,6351663875
n	41000	42000	43000	44000	45000
$C_2(n)$	0,6351663584	0,6351663372	0,6351663157	0,6351662998	0,6351662798

n	46000	47000	48000	49000	50000
$C_2(n)$	0,6351662598	0,6351662388	0,6351662336	0,6351662279	0,6351662200
n	51000	52000	53000	54000	55000
$C_2(n)$	0,6351661986	0,6351661859	0,6351661697	0,6351661589	0,6351661505
n	56000	57000	58000	59000	60000
$C_2(n)$	0,6351661450	0,6351661354	0,6351661179	0,6351661112	0,6351660991
n	61000	62000	63000	64000	65000
$C_2(n)$	0,6351660882	0,6351660777	0,6351660689	0,6351660597	0,6351660535
n	66000	67000	68000	69000	70000
$C_2(n)$	0,6351660438	0,6351660338	0,6351660273	0,6351660188	0,6351660131
n	71000	72000	73000	74000	75000
$C_2(n)$	0,6351660065	0,6351660007	0,6351659976	0,6351659938	0,6351659872
n	76000	77000	78000	79000	80000
$C_2(n)$	0,6351659782	0,6351659740	0,6351659690	0,6351659641	0,6351659513
n	81000	82000	83000	84000	85000
$C_2(n)$	0,6351659449	0,6351659341	0,6351659280	0,6351659226	0,6351659191
n	86000	87000	88000	89000	90000
$C_2(n)$	0,6351659181	0,6351659151	0,6351659175	0,6351659121	0,6351659055
n	91000	92000	93000	94000	95000
$C_2(n)$	0,6351659024	0,6351659064	0,6351659053	0,6351659165	0,6351659109
n	96000	97000	98000	99000	100000
$C_2(n)$	0,6351659164	0,6351658988	0,6351658902	0,6351658740	0,6351658579

de donde se deduce que una buena aproximación de la constante es

$$C_2 \simeq 0,635166. \quad (4.21)$$

La expresión final para la función recuento será, pues,

$$\pi_2^+(x) = \frac{9}{8} C_2 \int_{11}^x \frac{dt}{\ln t \cdot \ln \frac{t-1}{2} \cdot \ln \frac{t-3}{4}}. \quad (4.22)$$

Observación 4.34 Se puede comprobar experimentalmente que la relación

$$B_2 = \frac{\pi_2^+(x)}{\int_{11}^x (\ln t \cdot \ln \frac{t-1}{2} \cdot \ln \frac{t-3}{4})^{-1} dt}$$

entre el valor exacto de $\pi_2^+(x)$ y la integral aproximada tiende muy rápidamente a C_2 . Evaluando el número exacto de primos 2-seguros $\leq n$ con $n \in \mathbb{N}$ en el rango $11 \leq n < 4 \cdot 10^9$ y calculando por métodos numéricos la integral anterior, se obtiene la gráfica de la figura 4.2, donde se ha representado los valores de la relación B_2 en función de n . La curva estabiliza sensiblemente hacia el valor $\frac{9}{8}C_2 \simeq 0,714562$, representado por la línea horizontal.

Observación 4.35 En las referencias [36, 37] se pueden encontrar publicados algunos resultados parciales acerca de la densidad de los primos 2-seguros.

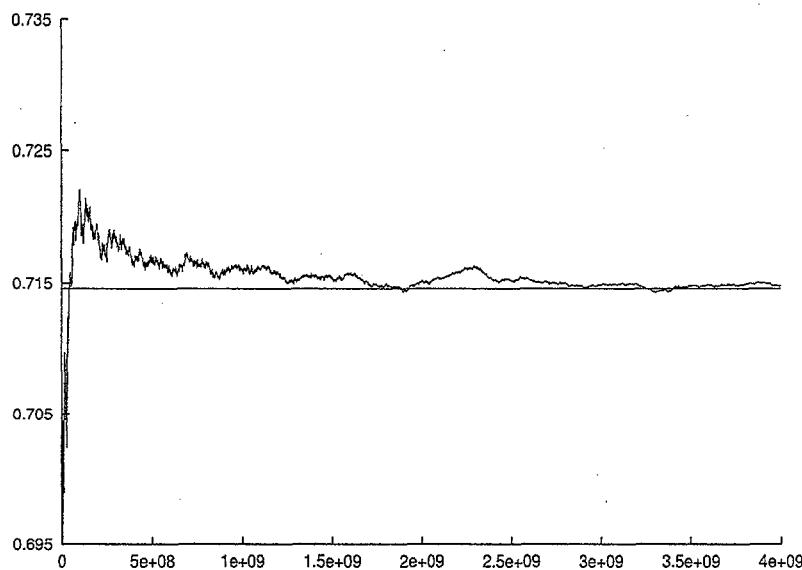


Figura 4.2: Valor de la constante B_2

4.3.2 Fórmula heurística asintótica para π_2^-

En completa analogía al caso π_2^+ , comenzemos con la siguiente

Notación 4.36 Denotaremos por $W_2^-(x)$ la probabilidad de que $x \in \mathbb{P}_2^-$.

Por definición, tenemos $\pi_2^-(x) = \# \{p \in \mathbb{P}_2^- : p \leq x\}$, con $x \in [5, \infty)$. Ahora bien, $p \in \mathbb{P}_2^-$, significa, según la Definición 4.1, que $p = 2p_1 - 1$, con p_1 primo y además $p_1 = 2p_2 - 1$, con p_2 también primo. Por consiguiente, para que un número cualquiera x esté en \mathbb{P}_2^- , ha de verificar simultáneamente lo siguiente:

1. x es primo,
2. $x_1 = \frac{1}{2}(x + 1)$ es primo, y
3. $x_2 = \frac{1}{2}(x_1 + 1)$ también es primo.

Argumentando de manera análoga al caso anterior, podemos ahora expresar las condiciones para que x esté en \mathbb{P}_2^- en forma de los siguientes sucesos que se han de cumplir simultáneamente:

1. $x \equiv 1 \pmod{4}$;
2. $x \not\equiv 0 \pmod{q}$, $x \not\equiv -1 \pmod{q}$, $x \not\equiv -3 \pmod{q}$, para todo $q \in \mathbb{P}$, tal que $q \leq \sqrt{x_2}$;
3. $x \not\equiv 0 \pmod{q}$, $x \not\equiv -1 \pmod{q}$, con $\sqrt{x_2} < q \leq \sqrt{x_1}$, $q \in \mathbb{P}$;
4. $x \not\equiv 0 \pmod{q}$, para todo $q \in \mathbb{P}$, tal que $\sqrt{x_1} < q \leq \sqrt{x}$.

La primera condición se sigue del hecho de que $x = 2x_1 - 1 = 4x_2 - 3$. La primera parte de las restantes condiciones nos garantiza que x sea primo; la segunda parte de la segunda y tercera condiciones nos garantizan que x_1 sea primo; y por último, la tercera parte de la segunda condición nos asegura que también lo sea x_2 . Puesto que las condiciones son totalmente análogas a las del caso π_2^+ , podemos concluir directamente que

$$W_2^-(x) \simeq \frac{9}{8} \frac{C_2}{\ln x \ln \frac{x+1}{2} \ln \frac{x+3}{4}}$$

donde C_2 toma el mismo valor que antes. Por consiguiente,

$$\pi_2^-(x) = \frac{9}{8} C_2 \int_5^x \frac{dt}{\ln t \ln \frac{t+1}{2} \ln \frac{t+3}{4}}.$$

Igual que en la pareja π_1^+ y π_1^- , la expresión de π_2^+ y de π_2^- es muy similar, lo que indica que el número de primos 2-seguros con signatura positiva es muy similar al de primos 2-seguros con signatura negativa. Este hecho encuentra confirmación en el Ejemplo 4.14, donde puede verse que $\#\mathbb{P}_2^+(7500000) = 1800$ y $\#\mathbb{P}_2^-(7500000) = 1807$.

4.4 Densidad de primos k -seguros

4.4.1 Fórmula asintótica generalizada para π_k^\pm

Las fórmulas heurísticas encontradas para los casos π_1^\pm son generalizables, siguiendo la misma línea argumental, al caso de la función recuento π_k^\pm . Existen, sin embargo, ciertos resultados en la literatura que son de aplicación y nos permiten simplificar notablemente la exposición.

Observación 4.37 Las únicas funciones recuento que vamos a considerar son las que corresponden a los conjuntos \mathbb{P}_k^\pm , pues, tal como se estableció en la Proposición 4.11, los conjuntos de signaturas alternadas son finitos y conocidos para $k < 5$ y están vacíos para $k \geq 5$.

Notación 4.38 En analogía a los casos anteriores, definiremos

$$\pi_k^\pm(x) = \# \{p \in \mathbb{P}_k^\pm : p \leq x\}$$

la función recuento de los primos k -seguros.

Recogemos a continuación un resultado, presentado como Lema 3 en la referencia [9], y dice lo siguiente:

Lema 4.39 Sean $f_1, f_2, \dots, f_s \in \mathbb{Z}[x]$, polinomios irreducibles y con el coeficiente del término de mayor grado positivo. Sea F el producto de todos ellos. Sea también $Q_F(N)$ el número de enteros positivos $j \in [1, N]$ tales que $f_1(j), f_2(j), \dots, f_s(j)$ son números primos. En este caso, se tiene,

$$Q_F(N) \leq 2^s s! C(F) N (\ln N)^{-s} + o(N (\ln N)^{-s}),$$

donde

$$C(F) = \prod_p \left(1 - \frac{1}{p}\right)^{-s} \left(1 - \frac{\omega(p)}{p}\right)$$

estando extendido el producto a todos los primos, y $\omega(p)$ denota el número de soluciones de la congruencia

$$F(X) \equiv 0 \pmod{p}.$$

La demostración puede consultarse en la citada referencia. Este lema dio origen a una conjetura que puede leerse en [7] y establece lo siguiente:

Conjetura 4.40 Con las mismas condiciones y notación que en el Lema 4.39, se tiene

$$Q_F(N) \sim h_1^{-1} h_2^{-1} \cdots h_s^{-1} C(F) \int_2^N (\ln u)^{-s} du, \quad (4.23)$$

donde h_1, h_2, \dots, h_s representan los grados de los polinomios f_1, \dots, f_s .

Esta conjetura no ha sido demostrada hasta la fecha pero sí ha recibido una abrumadora confirmación numérica experimental, empezando por la que los propios autores aportan en [7] para el caso particular de los polinomios $x, x^2 + x + 1$.

Es claro que este resultado resulta de aplicación para el caso de los primos k -seguros. En efecto, recordemos que para que $x \in \mathbb{P}_k^\pm$, el siguiente conjunto de condiciones simultáneas han de verificarse:

- (1) x es un primo;
- (2) $x_1 = \frac{1}{2}(x \mp 1)$ es primo;
- (3) $x_2 = \frac{1}{2}(x_1 \mp 1) = \frac{1}{4}(x \mp 3)$ es primo;
- ...
- ($k+1$) $x_k = \frac{1}{2}(x_{k-1} \mp 1) = \frac{1}{2^k}(x \mp (2^k - 1))$ es primo.

Es claro que podemos expresar las anteriores condiciones de manera que la fórmula (4.23) sea de aplicación. En efecto, si en el anterior conjunto de condiciones hacemos $x_k = y$ y despejamos los sucesivos valores de x_i en función de y obtenemos el siguiente conjunto de polinomios:

$$f_1(y) = y, f_2(y) = 2y \pm 1, f_3(y) = 4y \pm 3, \dots, f_{k+1}(y) = 2^k y \pm (2^k - 1).$$

En tal caso, $x = 2^k y \pm (2^k - 1)$ estará en \mathbb{P}_k^\pm si todos los polinomios $f_i(y)$ toman un valor primo para algún valor de y .

A partir de sus respectivas definiciones, podemos relacionar la función Q_F , definida en el Lema 4.39, con la función recuento de los primos k -seguros de la siguiente manera:

$$\pi_k^\pm(2^k y \pm (2^k - 1)) = Q_F(y),$$

o bien, en función de x (recordemos que hemos hecho $x_k = y$)

$$Q_F\left(\frac{1}{2^k}(x \mp (2^k - 1))\right) = \pi_k^\pm(x).$$

Aplicando entonces la fórmula (4.23), tendremos

$$\pi_k^\pm(x) = Q_F \left(\frac{1}{2^k} (x \mp (2^k - 1)) \right) \sim C(F) \int_2^{\frac{1}{2^k}(x \mp (2^k - 1))} (\ln u)^{-k-1} du.$$

Es conveniente hacer el siguiente cambio de variable:

$$u = \frac{1}{2^k} (t \mp (2^k - 1)), \\ du = \frac{1}{2^k} dt,$$

con lo que finalmente obtenemos:

$$\pi_k^\pm(x) \sim \frac{1}{2^k} C(F) \int_{2^{k+1} \pm 2^k - 1}^x (\ln \frac{1}{2^k} (t \mp (2^k - 1)))^{-k-1} dt.$$

Queda calcular el valor de la constante $C(F)$. Según su definición, tendremos para nuestro caso,

$$C(F) = \prod_p \left(1 - \frac{1}{p} \right)^{-k-1} \left(1 - \frac{\omega(p)}{p} \right).$$

Ahora necesitamos calcular $\omega(p)$ para los distintos valores de p . Definamos el siguiente conjunto

$$V(k) = \{j : p|2^j - 1, \quad 1 < j \leq k, \quad 2 < p \leq 2^k - 1\}.$$

La solución ha de ser considerada caso a caso de la siguiente manera:

$$\omega(p) = \begin{cases} 1, & p = 2 \\ p - \#V(k), & 2 < p \leq 2^k - 1 \\ k + 1, & p > 2^k - 1 \end{cases}$$

Como ejemplo, podemos considerar el caso $k = 2$, que correspondería a los 2-seguros. Se tiene que $V(2) = \{1\}$. Con ello, el valor de $\omega(p)$ sería:

$$\omega(p) = \begin{cases} 1, & p = 2 \\ 2, & p = 3 \\ 3, & p > 3 \end{cases}$$

con lo que tendríamos para $C(F)$

$$C(F) = ((1 - \frac{1}{2})(1 - \frac{1}{3}))^{-3} (1 - \frac{1}{2})(1 - \frac{2}{3}) \prod_{p>3} (1 - \frac{1}{p})^{-3} (1 - \frac{3}{p}) \\ = \frac{9}{2} \prod_{p>3} \frac{p^2(p-3)}{(p-1)^3}.$$

Consiguiertemente, obtendríamos para la función recuento

$$\pi_2^+(x) \sim \frac{9}{8} \prod_{p>3} \frac{p^2(p-3)}{(p-1)^3} \int_{11}^x (\ln \frac{t-3}{4})^{-3} dt.$$

Observemos que se ha llegado a un resultado muy similar al obtenido en la ecuación (4.22). De hecho, tan solo varía la integral, pero es fácil ver que

$$\lim_{x \rightarrow \infty} \frac{\int_{11}^x (\ln \frac{1}{4}(t-3))^{-3} dt}{\int_{11}^x (\ln t \cdot \ln \frac{t-1}{2} \cdot \ln \frac{t-3}{4})^{-1} dt} = 1,$$

y, por lo tanto, ambos resultados son asintóticamente equivalentes.

4.5 Generación de primos seguros

Comenzaremos esta sección explicando el algoritmo para la generación de primos 1-seguros. Después lo extenderemos a la generación de 2-seguros.

4.5.1 Generación de primos 1-seguros

Algoritmo 4.41 Dados $n, t \in \mathbb{N}$, este algoritmo consigue un primo 1-seguro de n bits. El test de primalidad se realiza apoyándose en el algoritmo de Miller-Rabin con valor t para el parámetro de seguridad.

LLAMADA: `Primo1Seguro(n, t);`

ENTRADA: Un entero n y un parámetro de seguridad t .

SALIDA: Un número primo aleatorio de n bits 1-seguro con probabilidad $1 - 2^{-2t}$.

1. [Inicialización]

Se elige la semilla para el generador aleatorio BBS.

2. [Lazo]

```
while (encontrado == NO)
```

```
{
```

```
    p = NumeroAleatorioBBS(n);
```

```
    if (MillerRabin(p, t))
```

```
{
```

$$q = \frac{p-1}{2};$$

```
    if (MillerRabin(q, t))
```

```
{
```

```
        encontrado = SI;
```

```
        return p;
```

```
}
```

```
}
```

```
}
```

Para la generación de números aleatorios empleamos el generador pseudoaleatorio de Blum, Blum y Shub que describimos en la sección 2.7.3, y que está basado en la congruencia $x^2 \pmod{N}$, donde N es el producto de dos primos. Recordemos que, según [117], se pueden considerar aleatorios $\log_2 \log_2 N$ bits de los producidos en cada iteración. El algoritmo que implementa este generador está descrito en la sección 3.4.1. Sin embargo, se puede recurrir a otro generador aleatorio de números, como el de Lehmer (véase sección 3.4.2) o el de Tausworthe (véase sección 3.4.3).

Por último, la comprobación de primalidad se realiza de acuerdo al algoritmo de Miller-Rabin modificado que explicamos en la sección 3.2.3.

El Algoritmo 4.41 se ha implementado en el lenguaje de programación C utilizando la biblioteca de multiprecisión GMP (véase [48]) que explicamos en la sección 1.3.1; el código se ofrece en el Apéndice I. El tiempo de ejecución de este algoritmo se explica en la Proposición 6.8.

4.5.2 Generación de primos 2-seguros

Para la generación de primos 2-seguros, se emplea una generalización del Algoritmo 4.41.

Algoritmo 4.42 Dados $n, t \in \mathbb{N}$, este algoritmo consigue un primo 2-seguro de n bits. El test de primalidad se realiza apoyándose en el algoritmo de Miller-Rabin con valor t para el parámetro de seguridad.

LLAMADA: `Primo2Seguro(n, t);`
 ENTRADA: Un entero n y un parámetro de seguridad t .
 SALIDA: Un número primo aleatorio de n bits 2-seguro
 con probabilidad $1 - 2^{-2t}$.

1. [Inicialización]

Se elige la semilla para el generador aleatorio BBS.

2. [Lazo]

```
while (encontrado == NO)
```

```
{
```

```
    p = NumeroAleatorioBBS(n);
    if (MillerRabin(p, t))
    {
        q1 =  $\frac{p-1}{2}$ ;
        if (MillerRabin(q1, t))
        {
            q2 =  $\frac{q1-1}{2}$ ;
            if (MillerRabin(q2, t))
            {
                encontrado = SI;
                return p;
            }
        }
    }
}
```

Los algoritmos utilizados para la generación de números aleatorios y para la comprobación de primalidad son los mismos que para el Algoritmo 4.41. El tiempo de ejecución para este algoritmo está descrito en la Proposición 6.16.

Capítulo 5

Primos robustos

Resumen del capítulo

Se revisan las distintas definiciones de primo robusto, discutiendo la necesidad e interés de su uso en los criptosistemas de clave pública. Se cuantifica la noción de primo robusto, introduciendo la función σ . Se aporta la noción de primo robusto óptimo, y se da una conjetura acerca de su función recuento y distribución, adjuntando datos experimentales. Finalmente se presentan distintos algoritmos clásicos para generar primos robustos junto con uno original que permite generar primos robustos óptimos.

5.1 Diversas definiciones de primo robusto

5.1.1 La definición estándar

Hemos visto en la sección 2.4.4, al estudiar el criptoanálisis de p y q en el criptosistema RSA, que tales primos deben satisfacer las condiciones que en esa sección se referían como (3) y (4); esto es, $\text{mcd}(p-1, q-1)$ debe ser “pequeño” y $p-1$ y $q-1$ deben contener un factor primo “grande”, por las razones allí explicadas.

Como ya se avanzó en la referida sección 2.4.4, estas condiciones motivaron la introducción de los llamados primos robustos, que se presentaron en la Definición 2.14 (véase también [73, §4.4.2]).

Observación 5.1 Obsérvese que si p y q verifican ambos la condición (a) de la Definición 2.14, entonces automáticamente verifican la condición (4) de la sección 2.4.4 y, si se eligen mediante un adecuado proceso probabilístico, también verifican (3), pues la probabilidad de que el factor primo grande de $p-1$ y el de $q-1$ coincidan, es despreciable.

Comentemos brevemente el sentido de las condiciones (a)-(c) de la Definición 2.14.

1. La condición (a) es necesaria para evitar el *algoritmo de factorización de Pollard* (véase [89]) que es eficiente cuando n contiene un factor primo p

tal que todos los factores primos de $p - 1$ son “pequeños”; esto es, $p - 1$ es B -uniforme (véase Definición 2.1).

2. La condición (b) es necesaria para evitar el *algoritmo de factorización de Williams* (véase [122]), que es eficiente si todos los factores primos de $p + 1$ son pequeños. Ambos algoritmos han sido explicados en las secciones 3.3.1 y 3.3.3.
3. La condición (c) es necesaria para evitar los *ataques cíclicos* de Simmons-Norris (véase [107]), que, básicamente, consisten en lo siguiente. Sea $c = m^e \pmod{n}$ un mensaje cifrado. Sea k un entero positivo tal que $c^{e^k} \equiv c \pmod{n}$. Tal entero tiene que existir porque el cifrado es una permutación de \mathbb{Z}_n en sí mismo; luego $c^{e^{k-1}} \equiv m \pmod{n}$. Un adversario calcula las iteraciones de la función de cifrado, c^e, c^{e^2}, \dots , hasta que obtenga c de nuevo. La penúltima iteración es el mensaje m . En cualquier caso, se puede iterar la función de cifrado hasta obtener un u tal que $f = \text{mcd}(c^{e^u} - c, n) > 1$. Entonces, o bien $f = n$ (y $u = k$), o bien f es uno de sus factores. En consecuencia, se trata de garantizar que u sea lo suficientemente grande como para hacer inviable un ataque cíclico. Ello se consigue con la condición (c). En efecto, si suponemos $f = p$, entonces $c^{e^u} \equiv c \pmod{p}$. El caso en que c sea un múltiplo de p o de q hay que descartarlo por razones obvias, de modo que podemos suponer $\text{mcd}(c, n) = 1$, con lo cual la congruencia anterior equivale a $c^{e^u-1} \equiv 1 \pmod{p}$. Un caso en el que tal relación se satisface es aquel en que u es el orden de e módulo $p - 1$. En efecto:

$$\begin{aligned} e^u &\equiv 1 \pmod{p-1} \\ \Rightarrow e^u - 1 &= l(p-1) \\ \Rightarrow c^{e^u-1} &= (c^{p-1})^l \equiv 1 \pmod{p}, \end{aligned}$$

en virtud de la congruencia de Fermat. Ahora bien, en este caso u divide a $\varphi(p-1)$, porque $\text{mcd}(e, p-1) = 1$, ya que por construcción $\text{mcd}(e, \varphi(n)) = 1$, y, por tanto, se aplica el teorema de Euler; pero, por hipótesis, se tiene $p - 1 = r \cdot r'$. Luego $\varphi(p-1) = \varphi(r) \cdot \varphi(r') = (r-1)\varphi(r')$. Así pues, si todos los factores primos de $r - 1$ fuesen pequeños, también lo sería u , y el ataque cíclico podría tener éxito.

Observación 5.2 Los primos robustos que satisfacen las condiciones de la Definición 2.14 se llaman también primos robustos de 3 vías.

5.1.2 La definición de Ogiwara

M. Ogiwara publicó en 1990 un interesante artículo (véase [83]) en donde presenta una definición más generalizada de lo que puede considerarse un primo robusto, con la idea de reforzar el criptosistema RSA y aquellos otros que utilicen también como él la exponenciación módulo un número compuesto n . Presentamos ahora la definición generalizada de Ogiwara, siguiendo sus propias notaciones:

Definición 5.3 Un primo impar p se denomina robusto si satisface las siguientes condiciones:

- (1) existe un primo grande, q_1 , tal que $q_1 \mid p - 1$;
- (2) existe un primo grande, q_2 , tal que $q_2 \mid p + 1$;
- (3) existe un primo grande, r_1 , tal que $r_1 \mid q_1 - 1$;
- (4) existe un primo grande, r_2 , tal que $r_2 \mid q_1 + 1$;
- (5) existe un primo grande, r_3 , tal que $r_3 \mid q_2 - 1$;
- (6) existe un primo grande, r_4 , tal que $r_4 \mid q_2 + 1$.

Observación 5.4 Obsérvese que las condiciones (1)-(3) son las mismas que en la definición estándar mientras que las (4)-(6) son las que amplían y refuerzan la definición de robusto en el sentido de este autor.

Observación 5.5 Los primos robustos en el sentido descrito por este autor se denominan también primos robustos de 6 vías.

A continuación, el autor describe una propuesta de algoritmo que permite encontrar primos que satisfacen las condiciones de la Definición 5.3. Este algoritmo está descrito en la sección 5.3.2.

5.1.3 El análisis de Rivest

En la literatura se ha puesto de manifiesto la falta de acuerdo respecto a la necesidad o conveniencia del uso de la clase de primos robustos. Algunos autores —notablemente [99]— argumentan que no son necesarios, básicamente por dos razones:

- (i) Usar un par de primos que satisfagan las condiciones (a)-(c) de la Definición 2.14 aumenta muy poco la seguridad del criptosistema con respecto a una elección puramente aleatoria, ya que, si bien cumplir esas condiciones evita los algoritmos de Pollard y Williams, no consigue sin embargo esquivar el algoritmo de las curvas elípticas de Lenstra [64], sin contar con el moderno algoritmo de la criba del cuerpo de números del que se afirma (ver, por ejemplo, [20]) que tiene éxito casi con seguridad en menos tiempo que cualquiera de los otros métodos.
- (ii) Los ataques cíclicos son lentos porque al menos requieren $\sqrt[3]{p}$ pasos en general y, por tanto, no son buenos métodos de factorización. Además, su probabilidad de éxito es extremadamente baja, según se afirma en una reciente serie de artículos; véase [24, 43, 44].

Pero ambos puntos admiten cierta crítica.

1. En cuanto al punto (i), cabe decir que Rivest no estima la probabilidad de que las condiciones (a)-(c) de la Definición 2.14 se cumplan en una elección puramente aleatoria. Además, es un hecho cierto que los algoritmos de Pollard y Williams son muy eficientes en las clases de primos para las que

están diseñados, de modo que si se tiene la “mala suerte” de elegir p, q en dichas clases, el criptosistema puede romperse. Por otro lado, la eficacia del algoritmo de las curvas elípticas tiende a disminuir para los números del tipo RSA; esto es, producto de dos primos del mismo tamaño. De hecho, en la actualidad el algoritmo de Lenstra se recomienda si los factores primos de n tienen alrededor de 40 cifras decimales, una longitud 4 veces inferior a la recomendada en la actualidad para el RSA.

2. En cuanto al punto (ii), hay que reconocer que la estimación de probabilidad de Rivest es sólida si se eligen p, q “en general”, pero si se eligen p, q “sin cuidado” de manera que todos los factores primos de $r - 1$ sean pequeños, entonces el ataque cíclico se convierte en eficiente.

Otros autores, si bien comparten la idea de que no hay razones definitivas para usar los primos robustos, siguen recomendándolos pues ofrecen una seguridad adicional con un coste muy bajo. Gordon estima en [47] que encontrar un primo robusto cuesta sólo un 19 % más de tiempo que encontrar uno cualquiera. Existen otras propuestas para generar primos dotados de ciertas propiedades; véanse, por ejemplo, [59, 83, 105, 106], donde se aportan también datos sobre eficiencia y tiempos de computación.

5.2 Primos robustos óptimos

En esta sección vamos a tratar de introducir una noción precisa acerca de cuándo se pueden calificar como óptimos los primos de tipo robusto, pues la Definición 2.14 presentada más arriba es más bien cualitativa.

5.2.1 La noción de primo robusto óptimo

Comencemos con la siguiente

Proposición 5.6 Sea p un primo impar que verifica:

- (a) $p - 1 = ra$, siendo r un primo impar,
- (b) $p + 1 = sb$, siendo s un primo impar,
- (c) $r - 1 = tc$, siendo t un primo impar.

En estas condiciones, se tiene:

$$\sigma = \frac{p-1}{r} + \frac{p+1}{s} + \frac{r-1}{t} \geq 12.$$

Demostración Obsérvese en primer lugar que σ es par porque es una suma de 3 números pares. Afirmamos que $\sigma \geq 8$. En efecto, si $\sigma < 8$, entonces debe ser $\sigma \leq 6$, y puesto que cada una de las fracciones es ≥ 2 , se concluye $\sigma = 6$; es decir, se tiene: $p - 1 = 2r$, $p + 1 = 2s$, $r - 1 = 2t$. Ahora bien, ello es imposible porque en este caso sería $p = r + s$, lo cual es contradictorio ya que p, r y s son los tres impares.

Según esto, se sigue que $\sigma \geq 8$; recordando que σ es par quedan sólo dos posibilidades: $\sigma = 8$ y $\sigma = 10$. Vayamos por orden, suponiendo en principio que $\sigma = 8$. Tenemos entonces los siguientes casos:

1. $a = 4, b = 2, c = 2$.

Esto es, $p = 4r + 1, p = 2s - 1, r = 2t + 1$. Restando miembro a miembro las dos primeras igualdades, se tiene que $s = 2r + 1$. Teniendo en cuenta la segunda igualdad, se deduce que $p \in \mathbb{P}(-1, +1)$. Según se demostró en la Proposición 4.11, el único elemento de ese conjunto es 13, con lo que $r = 3$, luego t no puede existir. Por tanto este caso no puede darse.

2. $a = 2, b = 4, c = 2$.

Esto es, $p = 2r + 1, p = 4s - 1, r = 2t + 1$. De nuevo restando miembro a miembro las dos primeras igualdades, se obtiene que $r = 2s - 1$, y teniendo a la vista la primera igualdad, se deduce que $p \in \mathbb{P}(+1, -1)$. Acudiendo de nuevo a la Proposición 4.11, el único elemento de ese conjunto es 11, con lo que $r = 5$, luego t no puede existir ni este caso darse.

3. $a = 2, b = 2, c = 4$.

Tenemos ahora $p = 2r + 1, p = 2s - 1, r = 4t + 1$. Sumando miembro a miembro las dos primeras ecuaciones se obtiene que $p = r + s$. Ahora bien, esto es contradictorio pues los tres números se suponen impares, luego este caso tampoco puede darse.

Se concluye de la discusión precedente que $\sigma > 8$. Nos queda entonces por tratar el segundo supuesto, es decir, sea ahora $\sigma = 10$. Entonces, se tienen los siguientes casos:

1. $a = 6, b = 2, c = 2$.

Esto es, $p = 6r + 1, p = 2s - 1, r = 2t + 1$. Comparando las dos primeras igualdades se tiene: $s = 3r + 1$, luego s es par. Contradicción.

2. $a = 4, b = 4, c = 2$.

Esto es, $p = 4r + 1, p = 4s - 1, r = 2t + 1$. Comparando las dos primeras igualdades se tiene: $2(s - r) = 1$. Contradicción.

3. $a = 2, b = 6, c = 2$.

Esto es, $p = 2r + 1, p = 6s - 1, r = 2t + 1$. Comparando las dos primeras igualdades se tiene: $r = 3s - 1$, luego r es par. Contradicción.

4. $a = 2, b = 2, c = 6$.

Esto es, $p = 2r + 1, p = 2s - 1, r = 6t + 1$. Comparando las dos primeras igualdades se tiene: $r - s = 1$, luego o bien r o bien s es par. Contradicción.

5. $a = 4, b = 2, c = 4$.

Esto es, $p = 4r + 1, p = 2s - 1, r = 4t + 1$. Comparando las dos primeras igualdades se tiene $s = 2r + 1$; teniendo en cuenta, además, la segunda

igualdad, se deduce que $p \in \mathbb{P}(-1, +1)$. Según se demostró en la Proposición 4.11, el único elemento de ese conjunto es 13, con lo que $r = 3$, luego t no puede existir. Por tanto este caso tampoco puede darse.

6. $a = 2, b = 4, c = 4$.

Esto es, $p = 2r + 1, p = 4s - 1, r = 4t + 1$. Comparando las dos primeras igualdades se tiene $r = 2s - 1$; teniendo en cuenta, además, la segunda igualdad, se deduce que $p \in \mathbb{P}(+1, -1)$. Análogamente al caso anterior, según lo demostrado en la Proposición 4.11, el único elemento de ese conjunto es 11, con lo que $r = 5$, luego t no puede existir ni este caso darse.

Finalmente se desprende que σ tampoco puede ser igual a 10 en ningún caso. Recordando de nuevo que σ ha de ser par, forzosamente se tiene que $\sigma \geq 12$. ■

El resto de los casos (es decir, cuando alguno de los valores r, s , ó t sea par) están cubiertos por la siguiente

Proposición 5.7 Si p es un primo impar que no verifica la hipótesis de la Proposición 5.6, entonces o bien p es un primo de Fermat o de Mersenne, o bien p es tal que todos los factores primos impares de $p - 1$ son de Fermat.

Demostración El primo p no cumple la hipótesis de la Proposición 5.6 si no verifica alguna de las condiciones (a), (b) o (c).

Si p no verifica (a), entonces $p - 1$ ha de ser de la forma $p - 1 = 2^\alpha$; luego p es un primo de Fermat, pues es bien conocido (véase, por ejemplo [94]) que cuando un entero n es de la forma $n = 1 + 2^\alpha$ sólo es primo si lo es de Fermat.

Si p no verifica (b), entonces $p + 1$ ha de ser de la forma $p + 1 = 2^\beta$, luego p es un primo de Mersenne en este caso.

Supongamos que p no verifica (c) pero sí verifica (a). Esto significa que todos los factores primos impares de $p - 1$ son de Fermat, es decir, $p - 1 = 2^\alpha \pi_1^{\alpha_1} \pi_2^{\alpha_2} \dots \pi_k^{\alpha_k}$ donde cada π_i es un primo de Fermat. ■

Definición 5.8 Decimos que un primo robusto es óptimo si los enteros r, s , y t en la Proposición 5.6 son tan grandes como sea posible, o equivalentemente si el valor

$$\sigma = a + b + c = \frac{p - 1}{r} + \frac{p + 1}{s} + \frac{r - 1}{t}$$

es tan pequeño como sea posible.

Para un primo dado p que satisfaga las condiciones (a)-(c) de la Proposición 5.6, aun cuando r, s ó t no sean necesariamente impares, la suma $a + b + c$ toma su valor mínimo cuando r, s, t se escogen de manera que sean los factores primos mayores de $p - 1, p + 1, r - 1$, respectivamente.

Recordemos de la Definición 2.14 que un primo se considera robusto si los valores de r, s y t son “grandes”. Así pues, si los elegimos de manera que sean “lo más grandes posible”, obtendremos un “buen” primo de esta clase, el mejor posible. Así se justifica la elección que hemos hecho del valor σ en la presente Definición.

Definición 5.9 Sea $S(n)$ el factor primo mayor del entero n si $n \geq 2$ y $S(1) = 1$. Se define la función

$$\sigma: \mathbb{N} \setminus \{1, 2\} \rightarrow \mathbb{N}$$

por la fórmula:

$$\sigma(n) = \frac{n-1}{S(n-1)} + \frac{n+1}{S(n+1)} + \frac{S(n-1)-1}{S(S(n-1)-1)}.$$

Observación 5.10 Usualmente, a $S(n)$ se le conoce como la ‘smoothness’ de n ; véase [100].

Ejemplo 5.11 Se tiene:

$$\begin{aligned} S(32) &= 2, \\ S(75) &= 5, \\ S(30053021) &= 6007. \end{aligned}$$

Véase también la referencia [34].

5.2.2 Caracterización de los primos robustos óptimos

Teorema 5.12 Para todo primo $p \geq 23$ se verifica $\sigma(p) \geq 12$.

Demostración Si p verifica las hipótesis de la Proposición 5.6, entonces el enunciado del teorema se sigue inmediatamente de esa Proposición. Si p no verifica tal hipótesis, entonces, en virtud de la Proposición 5.7, p cae en uno de los siguientes casos:

- (1) Es un primo de Fermat: $p = 1 + 2^{2^a}$. Se tiene $S(p-1) = 2$, luego

$$\frac{p-1}{S(p-1)} = 2^{2^a-1}.$$

De ahí:

$$\sigma(p) \geq 2^{2^a-1} + 2 = \frac{p-1}{2} + 2 = \frac{p+3}{2} \geq 12,$$

con tal que $p \geq 23$. Hemos tenido en cuenta que a, b, c tienen como valor mínimo 1 y por lo tanto la suma de dos de ellos ha de valer al menos 2.

- (2) Es un primo de Mersenne: $p = 2^b - 1$. Se tiene, en este caso, $S(p+1) = 2$, de donde

$$\sigma(p) \geq 2^{b-1} + 2 = \frac{p+1}{2} + 2 = \frac{p+5}{2} \geq 12,$$

con tal que $p \geq 19$.

- (3) Por último, si $p-1$ es tal que todos sus factores primos impares son de Fermat, entonces necesariamente $S(p-1) = 1 + 2^{2^c}$. Por lo tanto, $S(S(p-1)-1) = 2$ y

$$\frac{S(p-1)-1}{S(S(p-1)-1)} = \frac{2^{2^c}}{2} = 2^{2^c-1}.$$

Puesto que el mínimo valor de p que verifica este apartado es 3, es evidente que

$$\frac{p+1}{S(p+1)} \geq 2,$$

y podemos escribir

$$\sigma(p) \geq \frac{p-1}{S(p-1)} + 2 + 2^{2^c-1} \geq 12$$

Esta desigualdad se cumplirá siempre que $c \geq 3$. Los casos en que $c < 3$ podemos tratarlos individualmente:

a) $c = 0$; esto implica $S(p-1) = 1 + 2^{2^0} = 3$. Por tanto

$$\sigma(p) \geq \frac{p-1}{3} + 2 + 1 \geq 12$$

con tal que $p \geq 29$.

b) $c = 1$; esto implica $S(p-1) = 1 + 2^{2^1} = 5$. Por tanto

$$\sigma(p) \geq \frac{p-1}{5} + 2 + 2 \geq 12$$

con tal que $p \geq 41$.

c) $c = 2$; esto implica $S(p-1) = 1 + 2^{2^2} = 17$. Por tanto

$$\sigma(p) \geq \frac{p-1}{17} + 2 + 8 \geq 12$$

con tal que $p \geq 37$.

Se completa este caso con la siguiente tabla:

$p :$	23	29	31	37
$\sigma(p) :$	12	12	24	15

Corolario 5.13 Un primo p es robusto óptimo cuando verifica todas las condiciones de la Proposición 2.14 y además $\sigma(p)$ toma su valor mínimo, es decir, $\sigma(p) = 12$.

Demostración Se deduce directamente de la Proposición 5.6. ■

Teorema 5.14 Un primo $p > 29$ es un primo robusto óptimo si y sólo si se satisfacen las siguientes condiciones:

(i) $\frac{p-1}{6}$ es 1-seguro,

(ii) $S(p-1) = \frac{p-1}{6}$,

(iii) $S(p+1) = \frac{p+1}{4}$.

Demostración Como $p > 29$, es claro que siempre es posible elegir r y s de tal manera que $r > 3$ y $s > 3$. Para demostrar esto, supongamos que, por el contrario, o bien $p - 1 = 3^\alpha$ o bien $p - 1 = 2^\beta$, con $\alpha \geq 3$, $\beta \geq 5$. En cualquiera de los dos casos, tendremos la siguiente cota para $\sigma(p)$:

$$\sigma(p) \geq \frac{p-1}{S(p-1)} + 4 = 3^{\alpha-1} + 4 \geq 3^2 + 4 = 13,$$

o bien

$$\sigma(p) \geq \frac{p-1}{S(p-1)} + 4 = 2^{\beta-1} + 4 \geq 2^4 + 4 = 20.$$

Pero entonces también es claro que un primo p que admite tal factorización no puede ser un primo robusto óptimo. A la misma conclusión se llega si consideramos para el factor s que o bien $p+1 = 3^\alpha$ o $p+1 = 2^\beta$ con $\alpha \geq 3$ y $\beta \geq 5$. Por lo tanto, podemos suponer que $r > 3$ y $s > 3$.

Por otro lado, se tiene que $p^2 - 1 = 24k = (p-1)(p+1) = rsab$, como se sigue del hecho de que 24 divide a $p^2 - 1$, según se demostró en la Proposición 4.9. Puesto que tanto r como s son primos mayores que 3, es obvio que $ab = 24$. Recordando que a, b, c son todos pares y que $\sigma(p) = a+b+c = 12$, se ve que o bien $a = 4, b = 6, c = 2$, o bien $a = 6, b = 4, c = 2$. Sin embargo el primer caso ha de ser descartado. En efecto, supongamos que se tiene

$$p = 1 + 4r, \quad p = -1 + 6s, \quad r = 1 + 2t.$$

Sustituyendo,

$$\begin{aligned} p &= 1 + 4r = 1 + 4(1 + 2t) \\ &= 5 + 8t \\ &= -1 + 6s, \end{aligned}$$

y, por tanto, $8t = 6(s-1)$. Si $t = 3$, entonces $s = 5$ y $p = -1 + 6s = 29$, lo cual es imposible, por hipótesis; luego $t > 3$ lo que nos llevaría a contradicción, puesto que r es 1-seguro. ■

Corolario 5.15 Obtener primos robustos óptimos es equivalente a encontrar un entero t tal que

$$t, 2t+1, 3t+2, 12t+7$$

sean simultáneamente primos impares.

Demostración Si p es un primo robusto óptimo, en virtud de la demostración del Teorema 5.14, se tiene:

1. $p = 1 + 6r,$
2. $p = -1 + 4s,$
3. $r = 1 + 2t,$

siendo r, s, t tres números primos. Del punto 3, es evidente que r es un primo 1-seguro. Restando las expresiones de p en los puntos 1 y 2, se obtiene

$$s = \frac{1 + 3r}{2} = 2 + 3t.$$

Por último,

$$p = 1 + 6r = 7 + 12t.$$

Por tanto, si p es robusto óptimo, se verifica que $t, 2t + 1, 3t + 2$ y $12t + 7$ son simultáneamente primos.

Recíprocamente, si hacemos

$$\begin{aligned} r &= 2t + 1, \\ s &= 3t + 2, \\ p &= 12t + 7, \end{aligned}$$

entonces r es un primo 1-seguro; además, $p = 1 + 6r$ verifica las condiciones del Teorema 5.14, con lo que se concluye. ■

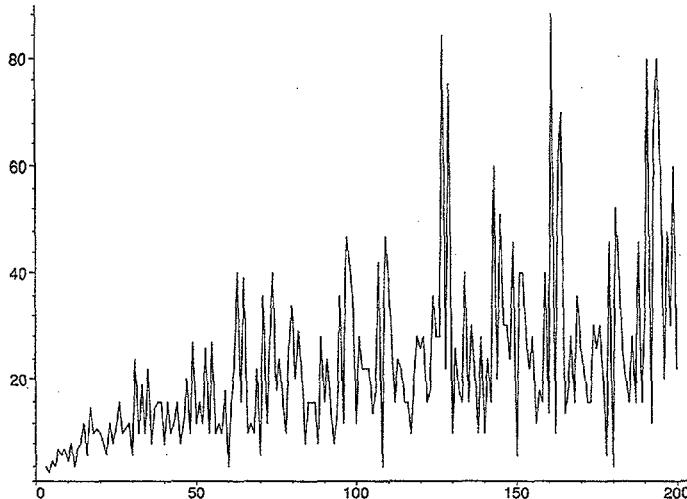


Figura 5.1: Gráfica de la función $n \mapsto \sigma(n)$

Observación 5.16 Existen muchos valores enteros n para los cuales $\sigma(n) < 12$. Es sencillo comprobar que si q es un primo 2-seguro, esto es, $q = 2r + 1, r = 2s + 1$, con r y s primos, entonces el número $n = 2q - 1$ verifica $\sigma(n) = 8$. Por este motivo, parece razonable suponer que existen infinitos. En la figura 5.1 se muestra la gráfica de la función $n \mapsto \sigma(n)$, en el rango $3 \leq n \leq 200$. Se aprecia que existen valores de n para los cuales $\sigma(n)$ es inferior a 12.

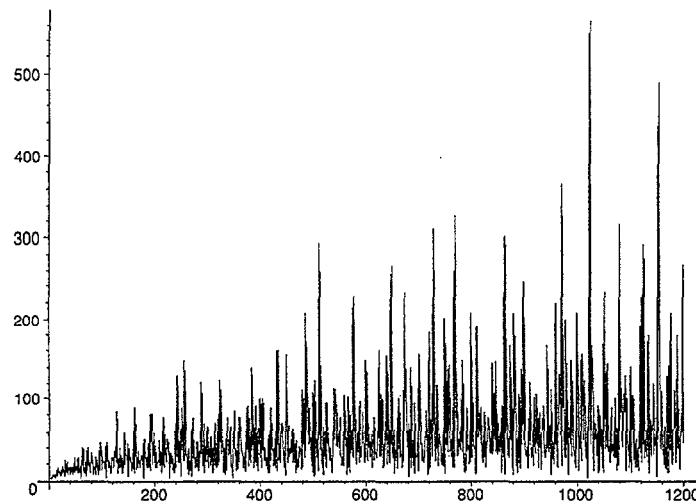


Figura 5.2: Gráfica de la función $n \mapsto \sigma(n)$

Observación 5.17 Análogamente, la figura 5.2 muestra la gráfica de $n \mapsto \sigma(n)$, en el rango $3 \leq n \leq 1200$. Es de destacar que la mayoría de sus puntos son inferiores a 100. De hecho, el porcentaje de valores de n tales que $\sigma(n) \leq 100$, es 86,25 %. A pesar de ello, siguen apareciendo “picos” en la curva, que son precisamente el extremo opuesto a los valores óptimos.

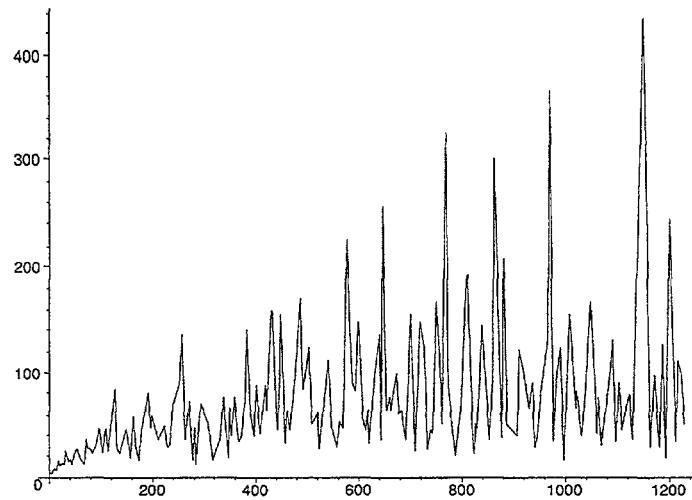


Figura 5.3: Gráfica de la función $p \mapsto \sigma(p)$

Observación 5.18 Comentarios similares pueden aplicarse a la gráfica de la

función anterior cuando su argumento recorre los números primos. En la figura 5.3 se muestra la gráfica de $p \mapsto \sigma(p)$, siendo p un número primo, en el rango $3 \leq p \leq 1200$. En este caso, el porcentaje de puntos inferiores a 100 es 78,97 %.

5.2.3 Función recuento

Las gráficas experimentales que se acaban de presentar en la sección 5.2.1 parecen prometer un número infinito de primos robustos óptimos. En esta sección vamos a tratar de justificar una conjetura acerca de su distribución y densidad, aunque no llegaremos a demostrar teóricamente de forma rigurosa la existencia de un número infinito de tales primos. Comencemos con la siguiente

Definición 5.19 Definimos la función $\pi_\sigma: [0, +\infty) \rightarrow \mathbb{N}$ para cada número real $x \geq 0$ como el número de primos robustos óptimos p tales que $p \leq x$. Es decir,

$$\pi_\sigma(x) = \#\{p \text{ primo robusto óptimo} : p \leq x\}.$$

La denominamos, como de costumbre, función recuento.

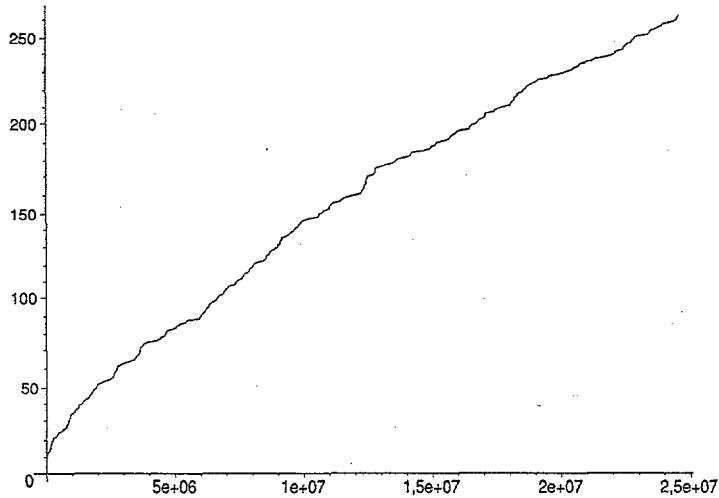


Figura 5.4: Función recuento π_σ

Observación 5.20 En la figura 5.4 representamos la función recuento π_σ en el rango $3 \leq x \leq 25 \cdot 10^6$. Aunque la gráfica deja claro que la densidad de los primos robustos óptimos es más bien baja, la curva mantiene un perfil creciente que da motivos para esperar que existan infinitos primos robustos óptimos.

De acuerdo con el Corolario 5.15, generar un primo robusto óptimo es equivalente a encontrar un número t , tal que $t, 2t+1, 3t+2, 12t+7$ sean simultáneamente primos. En este caso, podemos aplicar de inmediato los resultados que aparecen en [7] y que ya se ofrecieron en la sección 4.4. Los polinomios que entran en juego

en este caso son $f_1(x) = x$, $f_2(x) = 2x + 1$, $f_3(x) = 3x + 2$, $f_4(x) = 12x + 7$. Por tanto, para este caso se tiene

$$Q_\sigma(y) \sim C_\sigma \int_2^y \frac{du}{(\ln u)^4}, \quad (5.1)$$

donde

$$\begin{aligned} C_\sigma &= \left((1 - \frac{1}{2})(1 - \frac{1}{3})(1 - \frac{1}{5})(1 - \frac{1}{7}) \right)^{-4} \\ &\quad \cdot (1 - \frac{1}{2})(1 - \frac{2}{3})(1 - \frac{4}{5})(1 - \frac{3}{7}) \\ &\quad \cdot \prod_{p>7} \left(1 - \frac{1}{p} \right)^{-4} \left(1 - \frac{4}{p} \right) \\ &= \frac{1}{3} \frac{5^3 \cdot 7^3}{2^{11}} \prod_{p>7} \left(1 - \frac{1}{p} \right)^{-4} \left(1 - \frac{4}{p} \right) \\ &= \frac{42875}{6144} \prod_{p>7} \frac{p^3(p-4)}{(p-1)^4}. \end{aligned} \quad (5.2)$$

Es claro que existe la siguiente relación

$$Q_\sigma(y) = \pi_\sigma(12y + 7),$$

de acuerdo a sus respectivas definiciones. Así pues, haciendo $x = 12y + 7$, nos queda finalmente

$$\pi_\sigma(x) \sim C_\sigma \int_2^{(x-7)/12} \frac{du}{(\ln u)^4}.$$

Realicemos ahora el siguiente cambio de variable

$$\begin{aligned} u &= \frac{v-7}{12}, \\ du &= \frac{dv}{12}. \end{aligned}$$

Con ello,

$$\pi_\sigma(x) \sim \frac{1}{12} C_\sigma \int_{31}^x \frac{dv}{(\ln \frac{v-7}{12})^4}. \quad (5.3)$$

El valor aproximado de la constante C_σ ha de ser calculado por métodos numéricos, como ya se hizo en la sección 4.3 para la constante C_2 .

Cálculo de la constante C_σ

Sea (p_n) la sucesión de los números primos. Puesto que

$$C_\sigma(n) = \prod_{q=p_5}^{p_n} \frac{q^3(q-4)}{(q-1)^4},$$

entonces

$$\frac{42875}{6144} \cdot \lim_{n \rightarrow \infty} C_\sigma(n) = C_\sigma.$$

Pues bien, mediante el programa de MAPLE que se adjunta en el Apéndice I, hemos calculado la siguiente tabla de valores de $C_\sigma(n)$ con $5 \leq n \leq 100000$, que proporciona una entrada cada vez que n es múltiplo de 1000:

n	1000	2000	3000	4000	5000
$C_\sigma(n)$	0,7932142111	0,7931794156	0,7931694225	0,7931648402	0,7931622512
n	6000	7000	8000	9000	10000
$C_\sigma(n)$	0,7931606035	0,7931594698	0,7931586465	0,7931580231	0,7931575356
n	11000	12000	13000	14000	15000
$C_\sigma(n)$	0,7931571456	0,7931568268	0,7931565614	0,7931563375	0,7931561460
n	16000	17000	18000	19000	20000
$C_\sigma(n)$	0,7931559812	0,7931558374	0,7931557110	0,7931555991	0,7931554995
n	21000	22000	23000	24000	25000
$C_\sigma(n)$	0,7931554102	0,7931553298	0,7931552570	0,7931551909	0,7931551305
n	26000	27000	28000	29000	30000
$C_\sigma(n)$	0,7931550753	0,7931550245	0,7931549776	0,7931549343	0,7931548940
n	31000	32000	33000	34000	35000
$C_\sigma(n)$	0,7931548566	0,7931548218	0,7931547893	0,7931547588	0,7931547303
n	36000	37000	38000	39000	40000
$C_\sigma(n)$	0,7931547035	0,7931546783	0,7931546545	0,7931546321	0,7931546108
n	41000	42000	43000	44000	45000
$C_\sigma(n)$	0,7931545907	0,7931545717	0,7931545535	0,7931545363	0,7931545199
n	46000	47000	48000	49000	50000
$C_\sigma(n)$	0,7931545044	0,7931544895	0,7931544753	0,7931544617	0,7931544487
n	51000	52000	53000	54000	55000
$C_\sigma(n)$	0,7931544363	0,7931544244	0,7931544129	0,7931544020	0,7931543915
n	56000	57000	58000	59000	60000
$C_\sigma(n)$	0,7931543813	0,7931543716	0,7931543622	0,7931543532	0,7931543445
n	61000	62000	63000	64000	65000
$C_\sigma(n)$	0,7931543361	0,7931543281	0,7931543202	0,7931543127	0,7931543054
n	66000	67000	68000	69000	70000
$C_\sigma(n)$	0,7931542983	0,7931542915	0,7931542849	0,7931542785	0,7931542723
n	71000	72000	73000	74000	75000
$C_\sigma(n)$	0,7931542663	0,7931542604	0,7931542548	0,7931542493	0,7931542439
n	76000	77000	78000	79000	80000
$C_\sigma(n)$	0,7931542387	0,7931542337	0,7931542288	0,7931542241	0,7931542194
n	81000	82000	83000	84000	85000
$C_\sigma(n)$	0,7931542149	0,7931542105	0,7931542062	0,7931542021	0,7931541980
n	86000	87000	88000	89000	90000
$C_\sigma(n)$	0,7931541941	0,7931541902	0,7931541865	0,7931541828	0,7931541792

n	91000	92000	93000	94000	95000
$C_\sigma(n)$	0,7931541757	0,7931541723	0,7931541689	0,7931541657	0,7931541625
n	96000	97000	98000	99000	100000
$C_\sigma(n)$	0,7931541594	0,7931541563	0,7931541534	0,7931541505	0,7931541476

De todo ello se ve que una buena aproximación para esta constante es

$$C_\sigma = \frac{42875}{6144} \cdot \lim_{n \rightarrow \infty} C_\sigma(n) = 5,53491.$$

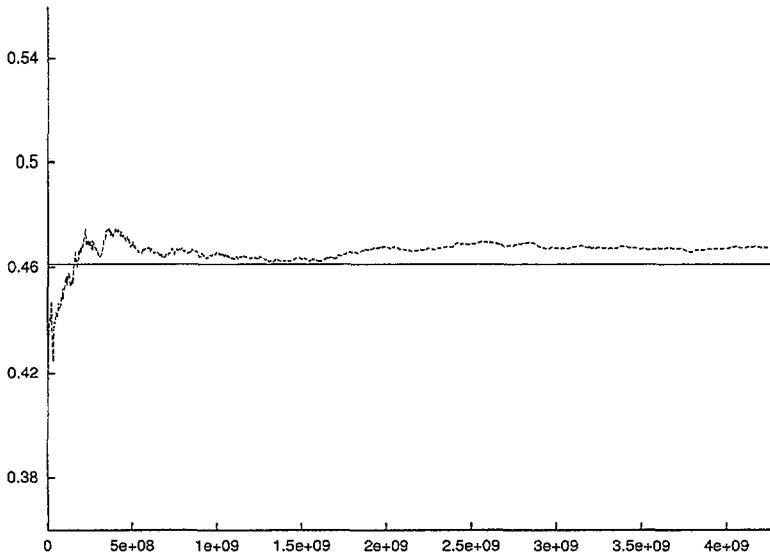


Figura 5.5: Valor de la constante B_σ

Observación 5.21 Se puede comprobar experimentalmente que se obtienen mejores resultados si, en vez de la ecuación (5.3), se toma la siguiente:

$$\pi_\sigma(x) \sim \frac{1}{12} C_\sigma \int_{31}^x \frac{dv}{\ln v \cdot \ln \frac{v-1}{6} \cdot \ln \frac{v+1}{4} \cdot \ln \frac{v-7}{12}}. \quad (5.4)$$

De hecho, la relación

$$B_\sigma(x) = \frac{\pi_\sigma(x)}{\int_{31}^x (\ln v \cdot \ln \frac{v-1}{6} \cdot \ln \frac{v+1}{4} \cdot \ln \frac{v-7}{12})^{-1} dv}$$

entre el valor exacto de $\pi_\sigma(x)$ y la integral aproximada se acerca al valor teórico $C_\sigma/12 = 0,46124$. En la figura 5.5 hemos representado los valores de la relación B_σ en función de x para $31 \leq x < 4,5 \cdot 10^9$, junto con el valor exacto de la constante. El perfil que presenta la curva resulta muy aceptable.

Véanse también las referencias [34, 35].

5.3 Algoritmos para la generación de primos robustos

Presentamos a continuación varios algoritmos para la obtención de primos robustos que aparecen en la literatura y el que se ha construido en esta memoria.

5.3.1 Algoritmo de J. Gordon

En la referencia [47], su autor, J. Gordon, propone una técnica sencilla para generar primos robustos. El autor expone su algoritmo argumentando que con ella se consigue construir un primo p dotado de las propiedades deseadas, es decir, que sea robusto, con un coste computacional económico. En efecto, según se demuestra en el artículo, el algoritmo consigue su objetivo usando tan solo un 19 % más del tiempo de computación necesario para calcular un primo convencional. Antes de describir los pasos del algoritmo, recogeremos un teorema que al autor enuncia y demuestra en [47, pp. 220–221].

Teorema 5.22 Si r y s son primos impares, entonces p satisface

$$\begin{aligned} p &\equiv 1 \pmod{2r}, \\ p &\equiv (s-1) \pmod{2s}, \end{aligned}$$

si y solo si p es de la forma $p = p_0 + 2krs$, donde

$$p_0 = \begin{cases} u(r, s) & \text{si } u(r, s) \text{ es impar} \\ u(r, s) + rs & \text{si } u(r, s) \text{ es par} \end{cases}$$

y $u(r, s) = (s^{r-1} - r^{s-1}) \pmod{rs}$.

Observación 5.23 Las condiciones del Teorema 5.22 vienen derivadas de la Definición 2.14 de primo robusto. En efecto, si $p \equiv 1 \pmod{2r}$, es claro que, entonces, $p-1 \equiv 0 \pmod{r}$, luego r es un factor “grande” de $p-1$. Si, además, $p \equiv (s-1) \pmod{2s}$, también es claro que $p+1 \equiv 0 \pmod{s}$, luego s es un factor “grande” de $p+1$.

Algoritmo 5.24 (de Gordon) Este algoritmo proporciona un primo robusto de n bits. Se utiliza el algoritmo de Miller-Rabin modificado con un parámetro de seguridad z , que deber ser suministrado como entrada.

LLAMADA: $\text{GordonStrong}(n, z)$;

ENTRADA: Un entero n y un entero parámetro de seguridad t .

SALIDA: Un número primo de n bits robusto
con probabilidad $1 - 2^{-2z}$.

1. [Inicialización]

Se seleccionan aleatoriamente $a, b \in \mathbb{Z}$, que llamaremos *semillas*.

2. [Generación de s, t]

Buscar el primer primo $s \geq a$.

Buscar el primer primo $t \geq b$.

3. [Generación de r]

for ($L \in [0, \infty)$)

{

```

 $r = 2Lt + 1;$ 
if (MillerRabin( $r$ ,  $z$ ))
    goto [Cálculo de  $p_0$ ];
}

4. [Comprobación del número de bits]
if (tamaño( $2rs$ ) no es  $\simeq n - \log_2 n$ )
    goto [Inicialización];

5. [Cálculo de  $p_0$ ]
 $u = (s^{r-1} - r^{s-1}) \pmod{rs};$ 
if (EsPar( $u$ ))
     $p_0 = u + rs;$ 
else
     $p_0 = u;$ 

6. [Generación de  $p$ ]
for ( $L \in [0, \infty)$ )
{
     $p = p_0 + 2Lrs;$ 
    if (MillerRabin( $p$ ,  $z$ ))
        return  $p$ ;
}

```

Expliquemos verbalmente cada uno de los pasos del algoritmo:

1. Se seleccionan en primer lugar dos enteros aleatoriamente, a y b , que llamaremos *semillas*.
2. A partir de esas dos *semillas*, se generan dos primos s y t . Para generar s (resp. t), utiliza el simple procedimiento de buscar el primer primo mayor o igual que a (resp. b).
3. A partir de t se construye otro primo, r . Este va a ser de la forma $r = 2Lt + 1$. La técnica es explorar el espacio $2Lt + 1$ para sucesivos valores de L , de 0 en adelante, hasta encontrar un primo que llamaremos r . El autor propone un conjunto de estrategias para controlar el tamaño en bits de r .
4. Comprobamos si el número de bits de $2rs$ es adecuado: en caso afirmativo, seguimos adelante; si no, volvemos al paso 1.
5. Construimos p_0 de acuerdo al Teorema 5.22.
6. Conocidos r , s y p_0 , construimos p . Esto se reduce a explorar el espacio $p_0 + 2Lrs$ para sucesivos valores de L , de 0 en adelante, hasta encontrar un primo, que será el primo p buscado.

Observación 5.25 El tamaño de p generado por el algoritmo es unos pocos bits mayor que el tamaño de $2rs$. Así pues, para controlar con precisión el tamaño de p elegimos un valor de $2rs$ adecuado, por ejemplo, $n - \log_2 n$ bits, siendo n el número de bits que se desean para p .

Se ha implementado el algoritmo de Gordon utilizando MAPLE. Ofrecemos en el Apéndice I el código de este algoritmo escrito en el lenguaje propio de MAPLE.

Para números no muy grandes, y utilizando una plataforma PC estándar, se ha obtenido una lista de primos robustos en pocos minutos, que transcribimos a continuación. En ella, se da cada primo p obtenido junto con el valor de la función σ (introducida en Definición 5.9) aplicada a él.

$$\begin{array}{cccccc} p : & 6128819 & 11234213 & 11429879 & 16861609 & 19068383 \\ \sigma(p) : & 15352 & 22230 & 9418 & 1258 & 30832 \end{array}$$

$$\begin{array}{cccccc} p : & 20510989 & 20647201 & 26070767 & 27617951 & 28429397 \\ \sigma(p) : & 25456 & 19382 & 15604 & 27670 & 21772 \end{array}$$

$$\begin{array}{cccccc} p : & 28918759 & 32877071 & 42144847 & 44562191 & 44570363 \\ \sigma(p) : & 54170 & 34294 & 19996 & 48232 & 2754 \end{array}$$

$$\begin{array}{cccccc} p : & 92679101 & 98786927 & 110854253 & 112548727 & 113451257 \\ \sigma(p) : & 89002 & 48172 & 70000 & 25002 & 139738 \end{array}$$

$$\begin{array}{cccccc} p : & 116978249 & 124238453 & 132696449 & 143372149 & 148300549 \\ \sigma(p) : & 36232 & 85756 & 67542 & 37548 & 122184 \end{array}$$

$$\begin{array}{cccccc} p : & 148384463 & 154465411 & 159518657 & 171770447 & 174913337 \\ \sigma(p) : & 88524 & 90628 & 202048 & 178120 & 7174 \end{array}$$

$$\begin{array}{cccccc} p : & 183006137 & 186178523 & 187861613 & 193229983 & 214999027 \\ \sigma(p) : & 64708 & 119338 & 21808 & 353450 & 113600 \end{array}$$

$$\begin{array}{cccccc} p : & 219345877 & 225453139 & 257541187 & 261455849 & 261699967 \\ \sigma(p) : & 8232 & 221268 & 20542 & 349552 & 111930 \end{array}$$

$$\begin{array}{cccccc} p : & 262587077 & 282277759 & 299330987 & 311958037 & 323141993 \\ \sigma(p) : & 72172 & 78506 & 68028 & 30064 & 76578 \end{array}$$

$$\begin{array}{cccccc} p : & 325051681 & 348920813 & 358708403 & 360850729 & 382437893 \\ \sigma(p) : & 72400 & 81352 & 165772 & 97000 & 93940 \end{array}$$

$$\begin{array}{cccccc} p : & 399103711 & 421265123 & 421273189 & 446977967 & 481027243 \\ \sigma(p) : & 61160 & 216106 & 189148 & 14560 & 86276 \end{array}$$

$$\begin{array}{cccccc} p : & 504440851 & 508178137 & 508226387 & 523226059 & 524397637 \\ \sigma(p) : & 201424 & 190132 & 96526 & 143622 & 69198 \end{array}$$

$$\begin{array}{cccccc} p : & 532319831 & 626691643 & 673276763 & 676250557 & 784871371 \\ \sigma(p) : & 343338 & 340098 & 119734 & 220268 & 58962 \end{array}$$

$$\begin{array}{cccccc} p : & 799088093 & 822756421 & 863265539 & 868016983 & 873406591 \\ \sigma(p) : & 162330 & 25280 & 93574 & 216688 & 500320 \end{array}$$

$$\begin{array}{cccccc} p : & 881098781 & 896977079 & 922277809 & 987885779 & 1024107067 \\ \sigma(p) : & 75922 & 524968 & 240168 & 267382 & 18800 \end{array}$$

$$\begin{array}{cccccc} p : & 1044607909 & 1067010743 & 1096743157 & 1098831893 & 1146917693 \\ \sigma(p) : & 71888 & 122202 & 82950 & 234952 & 336670 \end{array}$$

$$\begin{array}{cccccc} p : & 1286408261 & 1350012737 & 1394335433 & 1429403177 & 1445717107 \\ \sigma(p) : & 323188 & 1400554 & 61132 & 246718 & 145618 \end{array}$$

$$\begin{array}{cccccc} p : & 1474408129 & 1519134541 & 1521089387 & 1530303799 & 1536206897 \\ \sigma(p) : & 602092 & 404556 & 17099710 & 333280 & 72442 \end{array}$$

$$\begin{array}{cccccc} p : & 1590534863 & 1638876427 & 1691189693 & 1750935161 & 1797462493 \\ \sigma(p) : & 380584 & 227652 & 257722 & 296058 & 563922 \end{array}$$

$$\begin{array}{cccccc} p : & 1818099887 & 1864478299 & 1919090893 & 1932128293 & 1935537811 \\ \sigma(p) : & 242338 & 405688 & 476182 & 235794 & 304936 \end{array}$$

$$\begin{array}{cccccc} p : & 1935609947 & 1952507287 & 1989872359 & 1990547549 & 2013172583 \\ \sigma(p) : & 116448 & 94362 & 741098 & 479392 & 249310 \end{array}$$

$$\begin{array}{cccccc} p : & 2029062677 & 2055891931 & 2057826851 & 2083376461 & 2088288283 \\ \sigma(p) : & 316698 & 73904 & 444844 & 831152 & 1649928 \end{array}$$

$$\begin{array}{ccccccc} p : & 2212867147 & 2256296137 & 2280670993 & 2284620827 & 2354812133 \\ \sigma(p) : & 298940 & 770992 & 444628 & 388360 & 34914 \end{array}$$

$$\begin{array}{ccccccc} p : & 2489255927 & 2528691169 & 2623910899 & 2635586509 & 2727294749 \\ \sigma(p) : & 325744 & 89088 & 449884 & 167800 & 4268638 \end{array}$$

$$\begin{array}{ccccccc} p : & 2822973683 & 3027386543 & 3052250807 & 3115315171 \\ \sigma(p) : & 359454 & 628936 & 993220 & 51402 \end{array}$$

$$\begin{array}{ccccccc} p : & 3120843073 & 3293086513 & 3725036129 & 3822293537 \\ \sigma(p) : & 917528 & 73424 & 383482 & 746820 \end{array}$$

$$\begin{array}{ccccccc} p : & 3852474889 & 3869803547 & 4051345297 & 4095656941 \\ \sigma(p) : & 870822 & 647224 & 1817474 & 63016 \end{array}$$

$$\begin{array}{ccccccc} p : & 4121156513 & 4150148071 & 4943724497 & 5120069807 \\ \sigma(p) : & 2623054 & 1962512 & 174592 & 835194 \end{array}$$

$$\begin{array}{ccccccc} p : & 5250851741 & 5268808733 & 5285335823 & 5545408741 \\ \sigma(p) : & 1644504 & 1134376 & 927286 & 797846 \end{array}$$

$$\begin{array}{ccccccc} p : & 5596137293 & 5597156489 & 5661673211 & 5792075147 \\ \sigma(p) : & 624016 & 2236552 & 864522 & 271618 \end{array}$$

$$\begin{array}{ccccccc} p : & 5859699563 & 5899668721 & 6104144309 & 6966706519 \\ \sigma(p) : & 58720 & 443042 & 1090780 & 265216 \end{array}$$

$$\begin{array}{ccccccc} p : & 7071846451 & 7072248307 & 7116820153 & 7484401961 \\ \sigma(p) : & 1451298 & 966764 & 2530298 & 1354198 \end{array}$$

$$\begin{array}{ccccccc} p : & 7726901639 & 7898186951 & 8220624931 & 8428689469 \\ \sigma(p) : & 848818 & 4388656 & 2140020 & 183614 \end{array}$$

$$\begin{array}{ccccccc} p : & 9044907029 & 9716766529 & 11969437009 & 12261772639 \\ \sigma(p) : & 1086532 & 1550462 & 1232828 & 1938998 \end{array}$$

$$\begin{array}{ccccccc} p : & 13335796561 & 13799132749 & 14497700683 & 15712927081 \\ \sigma(p) : & 607662 & 4104680 & 1198468 & 1455392 \end{array}$$

$$\begin{array}{ccccccc} p : & 16969537961 & 17310126011 & 17849490979 & 18985523933 \\ \sigma(p) : & 3917256 & 2495416 & 408856 & 512440 \end{array}$$

$$\begin{array}{ccccccc} p : & 19874601959 & 20166413383 & 20317457717 & 29319966001 \\ \sigma(p) : & 3517662 & 2549018 & 773268 & 931100 \end{array}$$

$p :$	34037075593	38017681501	40178744887	47185039597
$\sigma(p) :$	1127178	4929460	12586420	911842

Los valores calculados de $\sigma(p)$ para cada p demuestran claramente cómo este algoritmo, aunque eficaz en tiempo de computación, produce sin embargo primos robustos muy lejanos de la optimalidad, cuando los medimos en términos de la función σ . En la tabla anterior, por ejemplo, puede verse cómo el valor mínimo de σ es 1258, muy lejano del valor óptimo, que es 12 como ya se demostró más arriba.

En la referencia [73] se recoge el algoritmo de Gordon, con algunas modificaciones, que ofrecemos a continuación.

Algoritmo 5.26 Este algoritmo proporciona un primo robusto p . Se utiliza el algoritmo de Miller-Rabin modificado con un parámetro de seguridad z , que deber ser suministrado como entrada.

LLAMADA: `GordonStrong2(n, z);`

ENTRADA: Un entero n y un entero parámetro de seguridad z .

SALIDA: Un número primo de n bits robusto
con probabilidad $1 - 2^{-2z}$.

1. [Inicialización]

Se eligen aleatoriamente dos primos distintos, s y t , de aproximadamente el mismo tamaño en bits y enteros i_0, j_0 .

2. [Generación de r]

```
for ( $i \in [i_0, \infty)$ )
{
     $r = 2it + 1;$ 
    if (MillerRabin( $r, z$ ))
        goto [Cálculo de  $p_0$ ];
}
```

3. [Cálculo de p_0]

$$p_0 = (2s^{t-2}(\text{mod } r))s - 1;$$

4. [Generación de p]

```
for ( $i \in [j_0, \infty)$ )
{
     $p = p_0 + 2jrs;$ 
    if (MillerRabin( $p, z$ ))
        return  $p$ ;
}
```

Veamos que realmente esta variante del algoritmo de Gordon también genera primos robustos. Observemos en primer lugar que, en virtud del teorema de Fermat, si $r \neq s$ se verifica que $s^{r-1} \equiv 1 \pmod{r}$, con lo que $p_0 \equiv 1 \pmod{r}$ y $p_0 \equiv -1 \pmod{s}$. Así pues,

- (a) $p-1 = p_0 + 2jrs - 1 \equiv 0 \pmod{r}$, por lo que $p-1$ tiene r como factor primo;
- (b) $p+1 = p_0 + 2jrs + 1 \equiv 0 \pmod{s}$, por lo que $p+1$ tiene s como factor primo; y
- (c) $r-1 = 2it \equiv 0 \pmod{t}$, por lo que $r-1$ tiene t como factor primo.

Sin embargo, también es sencillo comprobar que $\sigma(p)$ va a ser, con mucha probabilidad, un valor muy alto pues, de la Proposición 5.6, se tiene que

$$\sigma = a + b + c = \frac{p-1}{r} + \frac{p+1}{s} + \frac{r-1}{t}$$

y, por tanto, para nuestro caso

1. $a = \frac{p-1}{r} = \frac{p_0 + 2jrs - 1}{r} = 2js + \frac{(2s^{r-2} \pmod{r})s - 2}{r},$
2. $b = \frac{p+1}{s} = \frac{p_0 + 2jrs + 1}{s} = 2jr + 2s^{r-2} \pmod{r},$
3. $c = \frac{r-1}{t} = 2i.$

Si, como es de esperar, $j \neq 0$, entonces σ va a ser del orden de $2j(r+s)$, que será un valor en principio muy alto, pues justamente un primo es robusto cuando los valores de r y s son grandes. Conseguir un robusto óptimo con este algoritmo implicaría imponer las siguientes condiciones:

1. asegurar que $i = 1$ y que $j = 0$,
2. asegurar que $\frac{(2s^{r-2} \pmod{r})s - 2}{r} = 6$,
3. asegurar que $2s^{r-2} \pmod{r} = 4$.

5.3.2 Algoritmo de M. Ogiwara

Describiremos el algoritmo siguiendo al autor, para lo que introducimos en primer lugar algunas definiciones.

Definición 5.27 Decimos que un primo q es un “factor primo minus” de un primo p (abreviadamente, q es un fm de p) si q divide a $p-1$. Análogamente, decimos que un primo q es un “factor primo plus” de un primo p (abreviadamente, q es un fp de p) si q divide a $p+1$.

Definición 5.28 Se llamarán primos de nivel 1 a los producidos a la salida de este algoritmo; se denominarán primos de nivel 2 a los factores primos plus o minus de los primos de nivel 1 y primos de nivel 3 a los factores primos plus o minus de los primos de nivel 2.

Con estas definiciones y convenios, se construyen los primos de cada nivel de la siguiente manera.

Para empezar, se construyen primos de nivel 3 a partir de primos pequeños tabulados. Usando una tabla de primos pequeños, en el rango $[x, cx]$, generamos números pares, parcialmente factorizados, tales que las partes factorizadas sean mayores que $\sqrt[3]{cx}$. Aquí, c es una constante, a la que más tarde se asignará el valor 2. Esta condición garantiza la aplicabilidad del algoritmo probabilístico de primalidad que el autor desarrolla en secciones posteriores de la misma referencia.

A continuación, conseguidos dos primos de nivel 3, r_1 y r_2 , se construye un primo q_1 tal que r_1 y r_2 sean, respectivamente, un fm y un fp de q_1 . Conseguídos otros dos primos, r_3 y r_4 , se procede análogamente para obtener q_2 .

Por último, se construye un primo p de nivel 1 a partir de los dos primos de nivel 2, q_1 y q_2 del párrafo anterior, de modo que sean, respectivamente, un fm y un fp del primo p .

La idea para generar los primos a partir de su fm y su fp es debida a Gordon. Sean s y t el fm y el fp, respectivamente y supongamos que $s \neq t$. Usando el algoritmo de Euclides, se pueden encontrar dos enteros positivos, A y B , tales que

$$Bt - As = 1.$$

Si un primo p satisface que $p \equiv 2Bt - 1 \pmod{2st}$, entonces se tiene

$$p = 2(B + ks)t - 1 = 2(A + kt)s + 1$$

para algún entero k , de donde p tendrá a s y a t como su fm y su fp respectivamente. Además, $2Bt - 1$ (o, lo que es lo mismo, $2As + 1$) es primo con respecto a $2st$. Por lo tanto hay infinitos primos p que satisfacen $p \equiv 2Bt - 1 \pmod{2st}$, de acuerdo con los resultados de Dirichlet acerca de los primos en progresiones aritméticas (véase, por ejemplo, [94]). Sólo necesitamos dar valores a k y comprobar la primalidad de $p = 2(B + ks)t - 1$. Para controlar el tamaño del primo p de nivel 1 basta establecer una cota para k . Del primo p así generado diremos que cumple la condición de Ogiwara.

Resumamos esta explicación en los dos algoritmos siguientes:

Algoritmo 5.29 El algoritmo tiene como entrada dos números primos, p_1 y p_2 y el parámetro de seguridad z . Proporciona como salida un primo p que cumple la condición de Ogiwara con parámetro de seguridad z .

LLAMADA: $\text{Ogiwara}(p_1, p_2, z);$

ENTRADA: Dos primos p_1 y p_2 y un entero, el parámetro de seguridad z .

SALIDA: Un número primo p que cumple la condición de Ogiwara con probabilidad $1 - 2^{-2z}$.

1. [Euclides extendido]

Aplicamos este algoritmo para obtener A y B :

```
EuclidesExt(A, B, u, p1, p2); /* u=1, p1, p2 son primos */
```

```
M = 2p1p2;
```

2. [Prepara valor inicial de p]

```
if (signo(A) es positivo)
```

```
{
```

```
    p = 2Ap1;
```

```
}
```

```
else if (signo(B) es positivo)
```

```
{
```

```
    p = 2Bp2;
```

```
}
```

```
else
```

```
{
```

```
    return "Error: ni A ni B son positivos.";
```

```
}
```

3. [Lazo]

```
while (MillerRabin(p, z) == NO)
```

```
{
```

```
    p = p + M;
```

```
}
```

```
return p;
```

Utilizando este algoritmo, construimos el siguiente:

Algoritmo 5.30 El algoritmo tiene como entrada el número de bits n y el parámetro de seguridad z para el algoritmo de primalidad de Miller-Rabin; proporciona como salida un primo robusto de 6 vías (de Ogiwara) con tamaño n bits y con parámetro de seguridad z .

LLAMADA: `OgiwaraStrong(n, z);`

ENTRADA: Un entero n y un entero parámetro de seguridad z .

SALIDA: Un número primo de n bits robusto de Ogiwara con probabilidad $1 - 2^{-2z}$.

1. [Generación de cuatro primos de nivel 3]

Generamos r_1, r_2, r_3, r_4 , mediante el algoritmo `GeneraPrimoAleatorio`.

```
r1 =GeneraPrimoAleatorio(n/4, z);
```

```
r2 =GeneraPrimoAleatorio(n/4, z);
```

```

 $r_3 = \text{GeneraPrimoAleatorio}(n/4, z);$ 
 $r_4 = \text{GeneraPrimoAleatorio}(n/4, z);$ 

```

2. [Generación de dos primos de nivel 2]

Usando los primos de nivel 3, generamos dos, de nivel 2.

```

 $q_1 = \text{Ogiwara}(r_1, r_2, z);$ 
 $q_2 = \text{Ogiwara}(r_3, r_4, z);$ 

```

3. [Generación de un primo de nivel 1]

Usando los primos de nivel 2, generamos el primo pedido.

```
 $p = \text{Ogiwara}(q_1, q_2, z);$ 
```

Observación 5.31 Este algoritmo permite generar primos con un tamaño pre-determinado. En efecto, los tamaños de los factores q_i y r_i son del orden de \sqrt{p} y $\sqrt[4]{p}$, respectivamente. Como p se construye a partir de los factores r_i y q_i , basta elegirlos del tamaño apropiado para obtener el tamaño deseado para p . Por esta razón, los primos de nivel 3 se generan con un tamaño en bits de la cuarta parte del requerido, puesto que los factores r_i son del orden de $\sqrt[4]{p}$.

Observación 5.32 Para generar el par de primos de nivel 3 hemos recurrido al Algoritmo 3.27 en vez del propuesto por el autor. La razón es que no utilizamos su test de primalidad, sino el Algoritmo 3.8 de Miller-Rabin.

Hemos implementado el algoritmo de Ogiwara en una plataforma PC utilizando lenguaje C, con el nombre `OgiwaraStrong` (véase Apéndice I). El algoritmo es muy rápido y hemos podido obtener fácilmente la siguiente tabla en que presentamos los valores de primos robustos —en el sentido de este autor— obtenidos junto con los valores de la función σ para cada uno de ellos.

Tabla de primos robustos de Ogiwara

p robusto de Ogiwara	$\sigma(p)$
3168589466903640955006383757	251885989801938
4459449750221298923822200019	711480897289908
11134583520062100706426087189	1818390092428714
2256193406695142553489791779	616353752371222
1799164282413848715111434629	266512583446736
7045925859647918595203710289	544209830703754
2919789642653167383722862137	3899899586872746
115719223055921216414866075559	3572669824565752
1391968522714397775765098809	189344654650048
1013006681154448660333225500953	34472621604871252
34360939631553696804372209561	1761159652585702
113296882156814348251182250321	2895825746966156
11897741191815225983766435130031	67457862115652782
822844810916177943352542622673	13902465409149088

p robusto de Ogiwara	$\sigma(p)$
272821914877353530745360967699	5493003384794248
20238809365203924657747613268453	15426897103186684
18356806267520589706462571594411	48734993063005696
40164519507109565837620484570023	118261067602776708
88146756701888027195373152921027	375301280284643302
116571109989235901391462454893697	126506275901393820
1780760021192609102183222194243303	951766478160057038
185357275918218277765851936783713	385183532542923130
19523169476914103764135028771819	54446261730009940
296187854296394647453201118552017	105817515132457988
7182284643751974969826511031021383	816300423693629602
3041331874215688277579303717675279	249150615430310362
217295416330164231300596020886842079	14312579477616516388
77672696287283483818803499416162109	4481049881553579968
434453635171974219629152313063063	246527963764537044
27748706097864025358843809665517783	3065452787990554636
535034180842477380077437806415781197	27022723466408926838
1258784349396761445679714041884907091	14393741128363283972
306289600218519443639648762081172497	36179252018179463442
162699359032406161479120386667966457	3160040372373903802
225462658738238502755283709217123351	20822804663423834608
18298561326438383795237520761816674691	96064501335189457624
4478384803032597395861668805818074893	7697889907919013664
19044750008911317490512741429044975167	180216531571017986390
53308560724213194688322767571261016887	229362471694432775812
777451051201782771795394982555379439	34406409724709888352
230313837499199280915191068368999904129	884983275849693794236
16905042528697203763138668059666839013	18017574976371452554
13007962366058070540252348292917739171	23316654634362450628
42336567907024317415299540432143277127	178201647164448036718
24789628453689688988951936645507791427	114327713641164211300
899624371669084649361402284208386512339	237626070583104635328
2094362718576249623537880520449372071819	92799333390258024630
46395109470999630257675199174472886239	28425430546348095592
368895226491449926142791673176823191673	608899276869830351332
719564971454868072327260530902394884883	475800325018028015038
29472881110356734215204219283940043845433	4736303606038403783516
9903976455501627798773824102811710668253	1922592949478213568440
81561229056136726635019393285770982615183	3075344145631227688658
5205101707621975781276639276272533254353	153910079219079444110
3144258313792062816862616573573513823031	654706148078105054190
18458305269832320642897492160906322922702143	145732822156762383017286
125774068102883845314832964363194472455193	13668955899805114202146
250394412495487751824965430391770463582561	6940267003095785515714
9432011978049226352364652260201064000327	2150433885161368562980
103828782796825353326867490105762113033881	5020570197176431886878

p robusto de Ogiwara	$\sigma(p)$
4358980420683931748548150447763951727490747	24782432428729824608510
11455335641107561941552495191587204569954113	97590722389736568289128
3187859597893032354955995947272111487264569	101807235781755824790360
2509101050764416135266337904591226194433143	11286388409074236361216
763691437281886294164900801881488197820483	18086317922704489070210
10515731671013281397485967540381096301677629	86902367812182098624428
17822810038869886928109080604047051058184577	149663602654826467293052
880090083267657990857296370500841091568607513	1001922481198933815591418
3044678753700606757131521478576215456981683937	1389026049716481874523512
2044885807111518164924335940600179111451345969	877605751166924790541122
24368182775567239210947832472137921873775447753	816452941982507018786176
428506539461163371844910249164417588555230803	599633395874048562970582
2173058955278860040718819318993101587415065139	738326706906250648021222
5440503885931911645869927718762055054603668221	925883913298776521729778
150783501613179441771667817024950818667609523	363092252296398148877710

También como en el caso del algoritmo de Gordon, vemos que los valores de σ son muy altos. Estos primos están muy lejanos de la optimalidad.

5.3.3 Algoritmo para primos robustos óptimos

Presentamos ahora un método operativo que permite generar en la práctica primos robustos óptimos y que está basado en la caracterización dada en el Corolario 5.15.

Algoritmo 5.33 El algoritmo tiene como entrada el número de bits n y el parámetro de seguridad z para el algoritmo de primalidad de Miller-Rabin; proporciona como salida un primo robusto óptimo de n bits con parámetro de seguridad z .

LLAMADA: `StrongOptimo(n , z);`

ENTRADA: Un entero n y un entero parámetro de seguridad z .

SALIDA: Un número primo de n bits robusto óptimo
con probabilidad $1 - 2^{-2z}$.

1. [Inicialización]

Se elige la semilla para el generador aleatorio BBS.

2. [Lazo]

`while (TRUE)`

`{`

`$t = \text{NumeroAleatorioBBS}(n);$`

`if (MillerRabin(t , z))`

`{`

`$r = 2t + 1;$`

`if (MillerRabin(r , z))`

```

{
    s =  $\frac{1+3r}{2}$ ;
    if (MillerRabin(s, z))
    {
        p = 1 + 6r;
        if (MillerRabin(p, z))
            return p;
    }
}
}
}

```

Se comprueba fácilmente que el algoritmo devuelve un primo robusto óptimo realmente. En efecto, por la propia construcción, se deduce que $(r - 1)/t = 2$, $(p - 1)/r = 6$ y $(p + 1)/s = 4$, con lo que

$$\sigma = \frac{p-1}{r} + \frac{p+1}{s} + \frac{r-1}{t} = 12.$$

Por construcción, el número de bits de t es n , es decir, $\log_2 t = n$. Puesto que $p = 12t + 7$, es claro que $\log_2 p \leq 4 + \log_2 t = 4 + n$.

Observación 5.34 Un parámetro de importancia al generar primos especiales del tipo que sea es controlar su tamaño en bits. Una ventaja del algoritmo descrito es que controlar el tamaño en bits de p se reduce a controlar el de t .

Resultados experimentales

Presentamos ahora una tabla de primos robustos óptimos, con tamaños comprendidos entre aproximadamente 24 y 123 bits obtenidos utilizando el algoritmo 5.33. Se da un primo para cada tamaño.

Tabla de primos robustos óptimos

23381923,	45947443,
52491883,	131720683,
253773643,	570089803,
999290443,	2890729363,
6072231883,	12492535603,
18629892283,	46185983203,
92751619963,	111281540323,
300011563003,	444770052043,
1433852941603,	2787188880523,
5562410861803,	7142186272603,

16844802967723, 28965341018443,
92599988802403, 180498982277443,
413236049235403, 634351994196043,
1492746766145563, 2438811419265643,
4238310170718643, 6927563529158683,
24021820058804923, 35267637957986443,
104963053052422723, 186255441540169843,
279415594561000243, 683569721417515123,
989778489252112003, 2986640326986915163,
5832769788266237203, 7455813930838589683,
14955446493467176603, 40626747728628193723,
79191507328571868403, 153230884208038637563,
265804795012562810923, 618180170045765011843,
1492038142274522288443, 2973723738199212505603,
3667009025038464618763,
12267227374064027737843,
18341944861693514195803,
54739039385298556523083,
73029240787537761920923,
163798321922511671604883,
359997054375653024994283,
774365919656271083578363,
1215465071089189244613403,
3175229537824509499501723,
6584567773263909956947723,
14196010615172494910369923,
28883086812692846497914763,
57635428355002560323215363,
66232378369652880177258043,
229857182981185878569071723,
421478186323222482361491523,
637814834882013197004112003,
1752168595378961655030436843,
3328640992605045616985247403,
4098847089957889900927921123,
10518944435725278633014908003,
20131417360761582391420753363,
51686902561196178085116921763,
73659781742051637807699562843,
228125438800730282755742503603,
283303764101814653035863661243,
928139300474696421447509713003,
1701215293157722568280451516123,
3241349023722332480645503551403,
3900001270182574279176036007483,
12638057547178317537271017910003,
18795233039803613549001474230443,

53224715598697771282966075851163,
114370547621408049706207231543843,
150911919819022606212781659277723,
417827644272592375490465511004483,
506113502211052211745307039026163,
1823573972026258621020709888482643,
2355018964733172146698286322453643,
4108098316452124749606199408600003,
8632743199737398021299963647725563,
21897364442116892030041089978740803,
54513047376880901765901313509290323,
121231726097586290926170685055860243,
166957473658505310713837682116365603,
302980791645347966782712606154349723,
815038786864112107292653994138795683,
1268048809995834414879728947863344803,
3019337135068115799619521504285812683,
6764648382041294679901712524875587683,
12755032395140828426853528812742287683

Capítulo 6

Aplicaciones criptográficas de los primos especiales

Resumen del capítulo

Se describen las aplicaciones criptográficas para las que es conveniente el uso de los primos especiales descritos en los capítulos anteriores. En particular, se explica su aplicación a los sistemas RSA y BBS y se introduce el problema del cambio de clave. Se revisan los algoritmos para obtener los primos especiales y se presentan los resultados experimentales referidos al esfuerzo computacional necesario para obtenerlos.

6.1 Introducción

En este capítulo vamos a mostrar cómo diversos tipos de primos especiales tienen utilidad (y, de hecho, aparecen de modo natural) en la construcción de los criptosistemas de clave pública más difundidos (véase capítulo 2) y cómo las propiedades de estos primos afectan esencialmente a la seguridad de los criptosistemas.

Por otra parte, desde un punto de vista práctico o de ingeniería, es necesario estudiar los costes que la utilización de un determinado criptosistema comporta. Entre ellos está como factor de no poco interés el *coste computacional*, que podemos considerar tanto referido a la operación del sistema como a la generación de sus parámetros de funcionamiento, especialmente las claves.

Así pues, utilizando el material desarrollado en los capítulos previos, presentaremos también resultados que permiten predecir en la práctica los tiempos de computación de los primos especiales considerados.

6.2 Aplicaciones al criptosistema RSA

6.2.1 Uso de primos 1-seguros

Recordemos que al tratar en el capítulo 2 el criptoanálisis de los factores p y q del módulo de RSA, se requería que, en particular, esos factores cumplieran las

siguientes condiciones (véase sección 2.4.4):

- (3) $\text{mcd}(p - 1, q - 1)$ debe ser “pequeño”;
- (4) $p - 1$ y $q - 1$ deben contener un factor primo “grande”.

Proposición 6.1 Las condiciones (3) y (4) enunciadas para los primos p, q del criptosistema RSA se satisfacen óptimamente si p y q son primos 1-seguros.

Demostración En efecto, si $p = 2p_1 + 1$, $q = 2q_1 + 1$, siendo p_1, q_1 primos, entonces $\text{mcd}(p - 1, q - 1) = \text{mcd}(2p_1, 2q_1) = 2$, que es el mínimo valor posible ya que $p - 1$ y $q - 1$ son pares. Análogamente, $p - 1$ (resp. $q - 1$) contiene el factor primo más grande posible cuando es 1-seguro. ■

Observación 6.2 Queda claro, pues, que es muy recomendable utilizar este tipo de primos como factores para el módulo de RSA.

6.2.2 Uso de primos robustos

En el capítulo 2, sección 2.4.4, tratamos también del criptoanálisis del módulo de RSA. Decíamos que algunos autores recomiendan que los primos p y q sean de los llamados robustos para garantizar que las propiedades (3) y (4) enunciadas en la mencionada sección 2.4.4 se verifiquen y se soslayen los ataques mediante los algoritmos $p - 1$ de Pollard y $p + 1$ de Williams, que se explicaron en las secciones 3.3.1 y 3.3.3, respectivamente. En efecto, recordemos aquí la noción de primo robusto que proporcionamos en la Definición 2.14: un primo impar p se dice que es robusto si verifica las tres siguientes condiciones:

- (a) $p - 1$ tiene un factor primo grande r ;
- (b) $p + 1$ tiene también un factor primo grande s ;
- (c) $r - 1$ tiene también un factor primo grande t .

Recordemos que los algoritmos de Pollard y Williams son eficientes cuando los factores primos de $p - 1$ (resp. $p + 1$) son “pequeños”, o, de modo más preciso, $p - 1$ (resp. $p + 1$) es un número M -uniforme (véase la Definición 2.1), donde M puede ser, por ejemplo, del orden de 10^6 o 10^7 . Véase la sección 5.1.1 para una explicación detallada.

Proposición 6.3 Las condiciones para evitar el criptoanálisis mediante los algoritmos de Pollard y Williams se satisfacen óptimamente si se utilizan primos robustos óptimos como factores del módulo de RSA.

Demostración En virtud del Teorema 5.14, se deduce directamente que

- (a) el factor r de $p - 1$ es el más grande posible, y vale $(p - 1)/6$;
- (b) el factor s de $p + 1$ es el más grande posible, y vale $(p + 1)/4$;
- (c) el factor t de $r - 1$ es el más grande posible y vale $(r - 1)/2$.

■

Observación 6.4 El uso de primos robustos óptimos proporciona la máxima seguridad con respecto al criptoanálisis del módulo de RSA utilizando los algoritmos de tipo $p \pm 1$ de Pollard y Williams.

6.3 Aplicaciones al generador BBS

Hemos visto en el capítulo 2, en la sección 2.7.3, una descripción del generador de números pseudo-aleatorios de Blum, Blum y Shub, abreviadamente llamado BBS.

Ya observamos allí cómo el problema práctico de este generador tal y como sus autores lo propusieron en [13] es la dificultad que presenta la elección de una semilla que proporcione órbitas de la máxima longitud posible; o, al menos, garantice una longitud mínima a esas órbitas.

Este problema práctico ha sido atacado y resuelto en [54], tal como hemos descrito en la Proposición 2.50. Los resultados ahí descritos permiten describir un nuevo criptosistema que mejora el tradicional de Blum-Goldwasser. Los detalles se describen en la sección 2.7.4.

6.3.1 Uso de primos 1-seguros

Resumimos a continuación las condiciones presentadas en la Proposición 2.50 que permiten garantizar las órbitas de periodo máximo. Sean $p = 2p' + 1$ y $q = 2q' + 1$ dos primos 1-seguros y hagamos $n = p \cdot q$. En estas condiciones, existen órbitas de la función $x^2 \pmod{n}$ con ciclo máximo igual a $\frac{1}{8}(p-3)(q-3)$ si se satisfacen las condiciones (a) o (b) y (c) siguientes:

- (a) 2 es un generador de $\mathbb{Z}_{p'}^*$, y o bien 2 o -2 es generador de $\mathbb{Z}_{q'}^*$;
- (b) 2 es generador de $\mathbb{Z}_{q'}^*$, y o bien 2 o -2 es generador de $\mathbb{Z}_{p'}^*$;
- (c) $\text{mcd}(p'-1, q'-1) = 2$.

Con estas premisas, se demuestra en [54] (véase Proposición 2.51) que toda semilla $x \not\equiv 1, -1, 0 \pmod{p}$, $x \not\equiv 1, -1, 0 \pmod{q}$, da una órbita de periodo máximo, cuyo valor también se presenta en la Proposición 2.51.

6.3.2 Uso de primos 2-seguros

Las condiciones que necesitan los primos p y q exigidas en la Proposición 2.50 se satisfacen directamente si p y q son ambos primos 2-seguros, con $p, q > 11$ y o bien $p' \equiv 3 \pmod{8}$ o $q' \equiv 3 \pmod{8}$. Una demostración de ello puede verse en [54].

6.4 El problema del cambio de clave

Hemos visto en las secciones precedentes la conveniencia de utilizar en los criptosistemas reseñados los primos especiales de los que nos ocupamos en los capítulos previos.

Estos primos aparecen esencialmente en la elección de las claves de operación de los sistemas. En efecto, tal como hemos visto, en el caso del RSA se trata de elegir una pareja de primos para construir el módulo que formará parte de la clave pública, junto con el exponente de cifrado (que en algunas implementaciones prácticas se encuentra incluso fijado); esa pareja puede ser un par de primos 1-seguros, o bien de primos robustos. Para el generador BBS hemos visto la conveniencia de utilizar para el módulo del sistema una pareja de primos que sean 1-seguros o 2-seguros.

Estos requisitos suscitan la cuestión del coste que supone generar esos primos especiales, coste que tiene no solamente una repercusión económica en el sentido de cuánto dinero va a costar el recurso computacional necesario, sino que puede influir decisivamente en la viabilidad del propio sistema. En efecto, si se demuestra que obtener una clave que satisfaga un determinado conjunto de requisitos es inabordable incluso con el mejor de los computadores existentes, el uso práctico del sistema es imposible.

En este contexto se sitúa el problema que hemos querido resumir en el título: el problema del cambio de clave. En la referencia [63] se trata el problema de seleccionar el tamaño apropiado para la clave, dando un repaso exhaustivo a los diversos criptosistemas existentes. Sin embargo, no tenemos noticia de que exista hasta ahora ninguna referencia que aborde el problema de conocer cuál es el esfuerzo o coste computacional que implica generar un nuevo conjunto de claves que satisfagan los requisitos típicos para los criptosistemas actuales. Como se ha destacado antes, esos requisitos sí son satisfechos por los primos especiales de los que venimos hablando.

Conviene no olvidar, no obstante, que en los criptosistemas de clave pública, el cambio de clave no es una operación tan frecuente como el cambio de clave en los sistemas de clave simétrica, de modo que es aceptable que el tiempo necesario para el cambio de clave sea más largo en aquéllos que en éstos.

Así pues, en las secciones que siguen vamos a introducir los resultados prácticos más interesantes de la memoria, derivados del material presentado en los capítulos precedentes. Aportaremos los tiempos de computación necesarios para calcular primos especiales, justificándolos teóricamente y completándolos con datos obtenidos en experimentos numéricos reales.

Se trata, en definitiva, de contestar las preguntas que invariablemente se formulan quienes tratan de implantar en la práctica alguno de estos sistemas: ¿qué necesidades de recursos computacionales se van a tener si se utilizan claves de un determinado tamaño? ¿Cuánto tiempo se invertirá en un cambio de clave, dados estos recursos computacionales concretos? ¿Cómo influirá en el coste computacional el cambio de tamaño de un parámetro del sistema, como puede ser la clave?

Las siguientes secciones tratarán de dar contestación a estas o parecidas cuestiones.

6.5 Tiempo de ejecución para un primo 1-seguro

Examinamos en esta sección dos aspectos del tiempo de computación y apor-tamos datos experimentales.

En primer lugar, damos una estimación teórica acompañada de algunos ejem-plos reales, acerca del número de tiradas necesarias para obtener un primo 1-seguro. Recordemos que una tirada no es más que la obtención por métodos aleatorios de un número candidato con un tamaño en bits especificado.

En segundo lugar, obtendremos una estimación teórica acerca del tiempo de computación necesario para obtener en la práctica un primo 1-seguro. Hemos seleccionado el generador de números aleatorios BBS, para el que disponemos de un soporte teórico y práctico que se puede consultar en las secciones 2.7.3 y 3.4.1.

Por último, presentamos datos experimentales acerca del tiempo de compu-tación que ha sido necesario en la práctica para obtener primos 1-seguros de diferentes tamaños.

6.5.1 Estimación del número de tiradas

Comencemos la primera parte con la siguiente

Proposición 6.5 El número promedio de tiradas necesarias para obtener un pri-mo 1-seguro p es

$$\frac{2}{C} \cdot \ln p \cdot \ln \frac{p-1}{2}, \quad (6.1)$$

donde C es la constante de los primos gemelos, introducida en la sección 4.2.3.

Demostración Se deduce directamente de la fórmula (4.12), que da la densidad de los primos 1-seguros. ■

Ejemplo 6.6 Para obtener un primo 1-seguro de 32 bits, se requiere efectuar un promedio de

$$\begin{aligned} \frac{2}{C} \cdot \ln 2^{32} \cdot \ln \frac{2^{32}-1}{2} &\simeq \frac{2}{C} \cdot 32 \cdot 31 \cdot (\ln 2)^2 \\ &\simeq 722 \end{aligned}$$

tiradas.

Ejemplo 6.7 Para las aplicaciones actuales, se recomienda utilizar primos 1-seguros de hasta 512 bits. Según la Proposición 6.5, para un primo 1-seguro de 512 bits se requiere efectuar teóricamente un promedio de

$$\begin{aligned} \frac{2}{C} \cdot \ln 2^{512} \cdot \ln \frac{2^{512}-1}{2} &\simeq \frac{2}{C} \cdot 512 \cdot 511 \cdot (\ln 2)^2 \\ &\simeq 190411 \end{aligned}$$

tiradas. Este ejemplo no queda cubierto por los resultados presentados en la figura 4.1, que sólo alcanza hasta 2^{32} . Para contrastar este dato, adjuntamos los resul-tados de un experimento numérico en que hemos calculado el número promedio de tiradas necesario para obtener un primo 1-seguro en un entorno de 2^{512} .

El método seguido para efectuar este experimento es obtener números de 512 bits mediante el generador de números pseudoaleatorios BBS a partir de una semilla y contabilizar cuántos es necesario generar para que aparezca un primo 1-seguro. Finalmente se promedian los resultados obtenidos para distintas semillas. Para comprobar la primalidad de los números generados se utiliza el algoritmo de Miller-Rabin con un parámetro de seguridad $t = 10$, que asegura una probabilidad de éxito del algoritmo de

$$1 - 0,25^{10} = 0,99999904632568359375.$$

A continuación presentamos como ejemplo tres primos, p_1 , p_2 y p_3 así generados junto con el número de tiradas, t_1 , t_2 , t_3 , necesarias:

$p_1 =$	67822 45532 03870 50645 55364 68971 85037 88589 00919 17234 81074 59020 75759 63536 33294 62498 01962 72094 31196 74791 75843 18570 82390 98557 96500 44229 72001 16297 34051 68787 9723
$t_1 =$	351870
$p_2 =$	87567 43650 35344 01466 32384 07333 99722 84861 80353 66599 50210 97311 63175 95004 90343 10413 46142 26078 44320 66652 52549 70147 69722 55296 14405 50939 95172 61431 66194 57810 2863
$t_2 =$	246683
$p_3 =$	79956 40825 22936 67047 66892 36522 79559 62030 21538 14919 11000 68379 13898 70885 79486 92966 14240 59426 26711 74046 16874 87822 98874 73930 77980 11814 98211 22675 65502 39229 4507
$t_3 =$	223687

En el experimento se generaron un total de 100 primos 1-seguros y el promedio de tiradas fue de 245949, que es del orden del conjeturado. Conviene reseñar, no obstante, que la desviación típica $\sigma = 298950,3475$ es muy alta, lo que indica una gran dispersión de la variable 'número de tiradas'. En efecto, si la tabla de 100 primos se descompone en 5 de 20 cada una, se obtienen los siguientes valores para la media μ y la desviación típica σ :

μ	σ
206804,7000	226287,5087
219029,0000	298819,4130
205372,5500	182882,5678
220190,3500	177091,5102
378350,3500	468458,1094

Si bien la dispersión sigue siendo alta, es de notar que precisamente en los casos en que la media se acerca más al valor conjeturado, la desviación típica es de las más bajas.

6.5.2 Tiempo de ejecución teórico

Pasamos ahora a deducir el tiempo teórico de ejecución del algoritmo 4.41 explicado en la sección 4.5.1 que permite obtener primos 1-seguros. Comencemos con la siguiente

Proposición 6.8 El algoritmo 4.41 efectúa $O(n^4)$ operaciones bit para obtener un primo 1-seguro p de n bits.

Demostración Del enunciado de la proposición se sigue que $p \simeq 2^n$, es decir, $n \simeq \log_2 p$. Observemos en primer lugar que el número promedio de tiradas necesarias para encontrar un primo 1-seguro es, de acuerdo con la fórmula (6.1), $O(n^2)$. Por lo tanto, ése será el número de veces que, en promedio, será necesario iterar el lazo del algoritmo.

El lazo del algoritmo consta de la generación de un número aleatorio, usando el generador BBS que se explicó en la sección 3.4.1, y dos comprobaciones de primalidad, utilizando el algoritmo de Miller-Rabin modificado, explicado en la sección 3.2.3. Recordemos que el número de operaciones bit del generador es, de acuerdo con lo dicho en sección 3.4.1, $O(n)$ y recordemos también que la comprobación de primalidad toma $O(n^3)$ operaciones bit si p es primo y $O(n^2)$ si p es compuesto, según se explicó en la sección 3.2.3. Para calcular la contribución debida a la comprobación de primalidad, debemos distinguir los casos siguientes:

- (a) p no es primo;
- (b) p es primo, pero $(p - 1)/2$ no lo es;
- (c) p es primo y $(p - 1)/2$ también lo es.

Es claro que la contribución del caso (c) al tiempo de computación es despreciable, pues justamente es la condición de terminación del lazo y, por lo tanto, sólo se ejecuta una vez; hemos de considerar más bien los casos (a) y (b). De acuerdo con el teorema de los números primos, la probabilidad de que se dé el caso (b) es $1/\ln p$, esto es, $O(n)^{-1}$. Por lo tanto, de las $O(n^2)$ iteraciones necesarias, el caso (b) se dará en $O(n)$ de ellas, y la contribución al tiempo será

$$O(n)(O(n) + O(n^3) + O(n^2)),$$

en donde aparecen las contribuciones del generador y de la realización de las dos comprobaciones de primalidad, una con éxito (es decir, el candidato resultó ser primo) y otra sin él.

El caso (a) se dará en el resto de las iteraciones, que serán $O(n^2) - O(n) = O(n^2)$ por lo que se puede escribir

$$O(n^2)(O(n) + O(n^2)),$$

donde aparecen de nuevo las contribuciones del generador y de una comprobación de primalidad sin éxito. Resumiendo las contribuciones de los casos (a) y (b), se tiene finalmente

$$\begin{aligned} O(n)(O(n) + O(n^3) + O(n^2)) + \\ O(n^2)(O(n) + O(n^2)) = O(n^4) = O((\log_2 p)^4). \end{aligned} \quad (6.2)$$

Observación 6.9 Naturalmente, el tiempo de ejecución teórico no resuelve por completo el problema de la obtención de primos 1-seguros en la práctica, pues no proporciona el tiempo de ejecución real de la implementación del algoritmo en una determinada plataforma. Sin embargo, el resultado de la Proposición 6.8 es importante porque establece una cota superior del tiempo de ejecución, que es independiente de la implementación que exista en cada instante.

Observación 6.10 Como ya quedó dicho, en la fórmula (6.2) aparecen tres contribuciones al tiempo de ejecución:

1. Tiempo debido a la ejecución del generador aleatorio, correspondiente al término $O(n)$.
2. Tiempo debido a la comprobación de primalidad, correspondiente al término $O(n^3)$ u $O(n^2)$, según los casos.
3. Tiempo debido a la iteración del lazo del algoritmo 4.41, correspondiente al término $O(n^2)$.

Es obvio que de estas tres contribuciones, la tercera es inamovible, pues depende de la densidad de primos 1-seguros que exista en el intervalo considerado: se podría decir del término $O(n^2)$ que es “duro”. Sin embargo, los otros dos términos sí pueden ser susceptibles de mejora, si, por ejemplo, se consigue un método más eficiente de comprobación de primalidad, pues éste es, de los dos, el tiempo que domina.

6.5.3 Datos experimentales

Presentamos a continuación los datos de un experimento numérico en el que tratamos de establecer el tiempo real de ejecución y comprobar el ajuste de la predicción teórica a esos tiempos de computación.

El experimento consiste en obtener 10 primos 1-seguros cuya longitud en bits está comprendida entre 375 y 625 bits, con un paso de 5 bits, lo que da un total de 510 primos 1-seguros. Hemos elegido este rango porque está centrado justamente en 512 bits, que es la longitud requerida actualmente, por razones de seguridad, para los primos empleados en los criptosistemas de clave pública (véase [63]).

El equipo utilizado ha sido una plataforma PC estándar con las siguientes características:

1. Sistema operativo: Microsoft Windows XP, versión 2002, service pack 1.

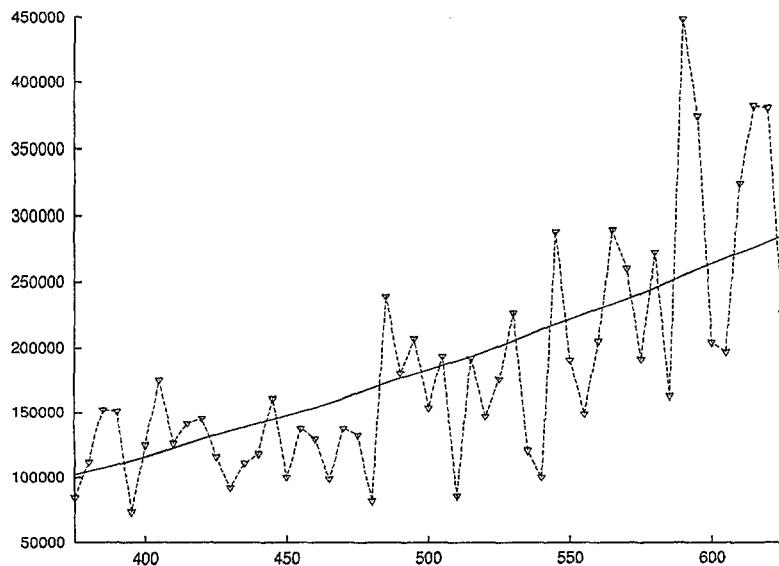


Figura 6.1: Tiradas T_1 frente a número de bits

2. Biblioteca Cygwin (véase [93]), que permite emular el entorno de programación y desarrollo de Unix. Da soporte a la mayoría de los servicios declarados en el estándar POSIX.
3. Procesador: Intel Pentium III, a 731 MHz.
4. Memoria: 256 MB.

Desarrollamos un programa escrito en C y apoyado en la biblioteca de programación en multiprecisión GMP (véase [48]). Ofrecemos el código original en el Apéndice I.

El primer dato de interés es justamente el número de tiradas que ha sido necesario, en promedio, para obtener un primo 1-seguro de un determinado número de bits. Teóricamente, el número de tiradas $T_1(n)$ necesarias se puede obtener a partir de la fórmula (6.1), de forma que para un 1-seguro de n bits, se tendría

$$T_1(n) \simeq \frac{2}{C} \cdot \ln 2^n \cdot \ln \frac{2^n - 1}{2} \simeq \frac{2}{C} (\ln 2)^2 n(n - 1),$$

es decir, T_1 es aproximadamente cuadrática en n . Para contrastar esto, presentamos en la gráfica 6.1 el número promedio de tiradas frente al número de bits, en línea discontinua, junto con la gráfica de la función αn^2 , en línea continua.

En teoría el valor de α debe ser igual a la relación

$$\alpha = \frac{T_1(n)}{n(n - 1)} \simeq \frac{2}{C} (\ln 2)^2 \simeq 0,7277810264. \quad (6.3)$$

Para comprobarlo, presentamos también la gráfica 6.2 que representa, con línea discontinua, los valores de la relación (6.3) para cada punto experimental y, en línea continua, el valor teórico α de la misma ecuación (6.3). Se observa que la concordancia es buena.

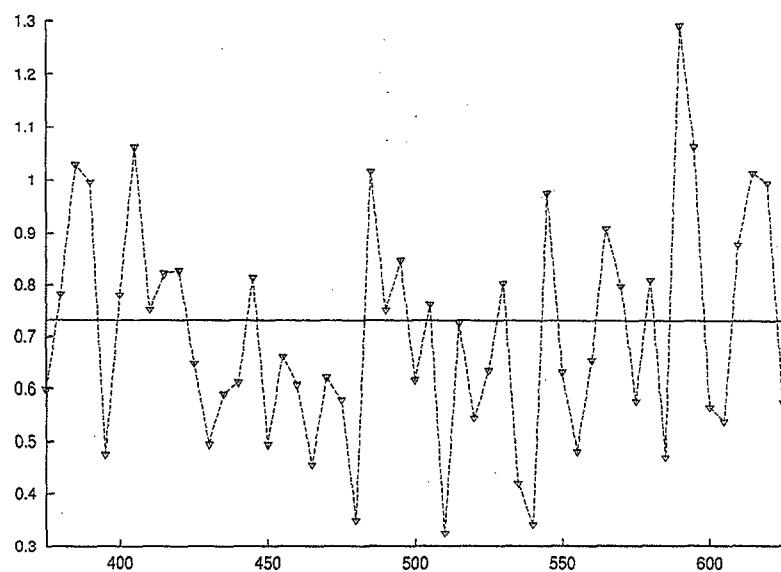


Figura 6.2: Valores teórico y experimental de la razón α

El siguiente paso es estudiar los tiempos experimentales de computación. A continuación, presentamos una tabla en que se puede ver el tiempo promedio en segundos necesario para obtener un primo 1-seguro de un número determinado de bits en el presente experimento.

Número de bits	Tiempo en segundos
375	157,297
380	208,955
385	299,254
390	300,222
395	149,159
400	258,051
405	349,47
410	251,564
415	288,547
420	304,064
425	250,674
430	196,653
435	238,882
440	252,246
445	352,292
450	238,161
455	327,687
460	314,97
465	248,589
470	351,644
475	344,775
480	215,553

Número de bits	Tiempo en segundos
485	662,678
490	504,575
495	583,178
500	429,768
505	522,796
510	224,815
515	516,104
520	393,62
525	474,39
530	621,283
535	335,547
540	277,961
545	817,831
550	527,796
555	415,966
560	576,698
565	832,835
570	782,226
575	561,93
580	908,173
585	523,501
590	1378,35
595	1132,4
600	615,429
605	600,703
610	1017,91
615	1209,35
620	1220,81
625	725,191

En la figura 6.3 presentamos gráficamente estos mismos datos. Proporcionamos dos gráficas: por un lado están los resultados numéricos, representados por la línea quebrada discontinua, con triángulos en los puntos experimentales; la línea de trazo continuo representa una aproximación dada por el siguiente polinomio:

$$t = an^4 + b, \quad (6.4)$$

donde

$$a = 0,623983875909405907 \cdot 10^{-8}$$

$$b = 54,91383871$$

y n representa el número de bits. Las aproximaciones se han realizado utilizando el módulo para ajuste parabólico **LeastSquares**, del programa de computación simbólica **MAPLE**. De acuerdo con la Proposición 6.8, la aproximación de grado 4 es la que prescribe la teoría y, como se aprecia claramente en la figura 6.3, resulta muy ajustada, lo que viene a confirmar la validez del resultado teórico.

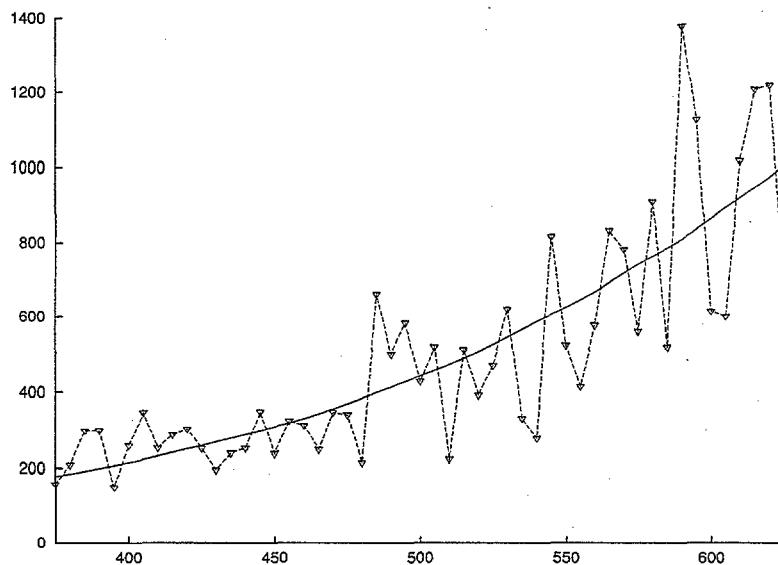


Figura 6.3: Tiempo de ejecución en segundos frente a número de bits

Para resumir, podemos dar ya una estimación del tiempo real necesario para obtener un primo 1-seguro en el intervalo $I = [500, 520]$, que es un valor práctico usado hoy día, de acuerdo a los resultados del experimento:

1. Valor promedio en I de acuerdo a la tabla experimental: 417,421 segundos.
2. Valor máximo en I de acuerdo a la tabla experimental: 522,796 segundos.
3. Valor según el polinomio (6.4) evaluado en $n = 510$: 477,051 segundos.

Como conclusión, se puede decir que para obtener un primo 1-seguro en un entorno de 512 bits se necesitan como máximo de 7 a 8 minutos de computación en una plataforma estándar utilizando el algoritmo 4.41.

6.5.4 Algoritmo mejorado para primos 1-seguros

El algoritmo 4.41 admite una mejora inmediata. Se tiene la siguiente

Proposición 6.11 Si $p > 5$ es un primo 1-seguro, entonces $p \equiv 3(\text{mod } 4)$.

Demostración De acuerdo con la Definición 4.1 y la Notación 4.2, p es un primo 1-seguro si y sólo si $p \in \mathbb{P}(+1)$. Aplicando entonces el resultado de la Proposición 4.3, se concluye inmediatamente. ■

Esta proposición nos permite mejorar el algoritmo simplemente forzando a que los candidatos a ensayar siempre cumplan la condición anterior. Pero esto es inmediato de conseguir, pues ello significa que los dos bits menos significativos de su representación binaria han de ser siempre 1. El algoritmo quedaría así:

Algoritmo 6.12 Dados $n, t \in \mathbb{N}$, este algoritmo consigue un primo 1-seguro de n bits. El test de primalidad se realiza apoyándose en el algoritmo de Miller-Rabin con valor t para el parámetro de seguridad.

LLAMADA: Primo1Seguro(n, t);
 ENTRADA: Un entero n y un parámetro de seguridad t .
 SALIDA: Un número primo aleatorio de n bits 1-seguro con probabilidad $1 - 2^{-2t}$.

1. [Inicialización]

Se elige la semilla para el generador aleatorio BBS.

2. [Lazo]

```
while (encontrado == NO)
{
    p = NumeroAleatorioBBS(n - 2);
    p = 4p + 3;
    if (MillerRabin(p, t))
    {
        q =  $\frac{p - 1}{2}$ ;
        if (MillerRabin(q, t))
        {
            encontrado = SI;
            return p;
        }
    }
}
```

Observemos que la diferencia consiste en generar un número aleatorio con dos bits menos de lo necesario que después aparecen tras multiplicarlo por 4 y sumarle 3. Con esto, el número de ensayos necesarios para obtener un primo 1-seguro queda disminuido a la cuarta parte, pues el espacio muestral se reduce a los números que están en la clase de equivalencia del 3 módulo 4.

6.6 Tiempo ejecución para un primo 2-seguro

El caso de los primos 2-seguros se puede tratar análogamente al de los 1-seguros. En la primera parte aportamos brevemente los datos referentes al número de tiradas y el valor teórico del tiempo de ejecución. En la segunda, proporcionamos datos experimentales referidos al tiempo de ejecución necesario para obtener para primos 2-seguros de diversos tamaños.

6.6.1 Estimación del número de tiradas

Comencemos la primera parte con la siguiente

Proposición 6.13 El número promedio de tiradas necesarias para obtener un primo 2-seguro p es

$$\frac{8}{9C_2} \cdot \ln p \cdot \ln \frac{p-1}{2} \cdot \ln \frac{p-3}{4}, \quad (6.5)$$

donde C_2 es la constante definida en la fórmula (4.17). Su valor aproximado es (véase fórmula (4.21))

$$C_2 \simeq 0,635166.$$

Demostración Se deduce directamente de la fórmula (4.18), que da la densidad de los primos 2-seguros. ■

Ejemplo 6.14 Para obtener un primo 2-seguro de 32 bits, se requiere efectuar un promedio de

$$\frac{8}{9C_2} \cdot \ln 2^{32} \cdot \ln \frac{2^{32}-1}{2} \cdot \ln \frac{2^{32}-3}{4} \simeq 13870$$

tiradas.

Ejemplo 6.15 Para las aplicaciones actuales, el tamaño recomendado para los primos 2-seguros llega hasta 512 bits. Según la Proposición 6.13, para un primo 2-seguro de 512 bits se requiere efectuar teóricamente un promedio de

$$\frac{8}{9C_2} \cdot \ln 2^{512} \cdot \ln \frac{2^{512}-1}{2} \cdot \ln \frac{2^{512}-3}{4} \simeq 62186721$$

tiradas.

Con este ejemplo se ve claro que encontrar números primos 2-seguros de gran tamaño, como puede ser 512 ó 1024 bits, es una tarea muy difícil, lo que debe ser tenido en cuenta a la hora de diseñar el sistema y elegir sus parámetros.

6.6.2 Tiempo de ejecución teórico

Proposición 6.16 El algoritmo 4.42 explicado en la sección 4.5.2 efectúa $O(n^5)$ operaciones bit para obtener un primo 2-seguro p de n bits.

Demostración Es completamente análoga a la demostración de la Proposición 6.8, a la que nos remitimos. En primer lugar, observemos que, del enunciado de la proposición se sigue que $p \simeq 2^n$, es decir, $n \simeq \log_2 p$. Además, de acuerdo con la fórmula (6.5), es claro que el número promedio de tiradas necesarias para encontrar un primo 2-seguro es $O(n^3)$. Por lo tanto, ése será el número de veces que, en promedio, será necesario iterar el lazo del algoritmo. En este caso, el lazo del algoritmo consta de la generación de un número aleatorio, usando el generador BBS que se explicó en la sección 3.4.1, y tres comprobaciones de primalidad, utilizando el algoritmo de Miller-Rabin modificado, explicado en la sección 3.2.3. Recordemos nuevamente que el número de operaciones bit del generador es, de

acuerdo con lo dicho en sección 3.4.1, de $O(n)$ y recordemos también que la comprobación de primalidad toma $O(n^3)$ operaciones bit si p es primo y $O(n^2)$ si p es compuesto, según se explicó en la sección 3.2.3. Para calcular la contribución debida a la comprobación de primalidad, podemos distinguir los casos siguientes:

- (a) p no es primo;
- (b) p es primo, pero $(p - 1)/2$ no lo es;
- (c) p es primo, $(p - 1)/2$ también lo es, pero $(p - 3)/4$ no lo es;
- (d) p es primo, $(p - 1)/2$ también lo es y $(p - 3)/4$ también.

La contribución del caso (d) al tiempo de computación es despreciable, pues justamente es la condición de terminación del lazo y, por lo tanto, sólo se ejecuta una vez; hemos de considerar más bien los casos (a), (b) y (c). De acuerdo con nuestros resultados acerca de la densidad de primos 1-seguros (véase la fórmula (4.12)), la probabilidad de que se dé el caso (c) es del orden de $1/(\ln p)^2$, esto es, $O(n)^{-2}$. Por lo tanto, de las $O(n^3)$ iteraciones necesarias, el caso (c) se dará en $O(n)$ de ellas, y la contribución al tiempo será

$$O(n)(O(n) + 2 \cdot O(n^3) + O(n^2)),$$

en donde aparecen las contribuciones del generador y de la realización de tres comprobaciones de primalidad, dos con éxito (es decir, el candidato resultó ser primo) y una tercera sin él.

El caso (b) se dará, de acuerdo con el teorema de los números primos, con una probabilidad de $1/\ln p$, esto es, $O(n)^{-1}$. Igual que antes, de las $O(n^3)$ iteraciones necesarias, el caso (b) se dará en $O(n^2)$ de ellas, por lo que la contribución al tiempo será

$$O(n^2)(O(n) + O(n^3) + 2 \cdot O(n^2)),$$

en donde aparecen las contribuciones del generador y de la realización de tres comprobaciones de primalidad, una con éxito y dos sin él.

Por último, en el resto de las iteraciones, que serán $O(n^3) - O(n^2) - O(n) = O(n^3)$ podemos escribir como contribución temporal

$$O(n^3)(O(n) + O(n^2)),$$

donde aparecen de nuevo las contribuciones del generador y de una comprobación de primalidad sin éxito. Resumiendo las contribuciones de los distintos casos, (a), (b) y (c), se tiene finalmente

$$\begin{aligned} & O(n)(O(n) + 2 \cdot O(n^3) + O(n^2)) + \\ & O(n^2)(O(n) + O(n^3) + 2 \cdot O(n^2)) + \\ & O(n^3)(O(n) + O(n^2)) = O(n^5) \end{aligned}$$



Observación 6.17 Son de aplicación las observaciones hechas para el caso de los primos 1-seguros: nuevamente la contribución más “dura” es la que procede de la densidad de los primos 2-seguros.

Observación 6.18 Observemos también que la contribución más importante al tiempo viene de que la mayoría de las $O(n^3)$ comprobaciones resultan no tener éxito (el candidato no era primo), por lo que reducir el tiempo de la comprobación resulta crucial para acelerar el algoritmo.

6.6.3 Datos experimentales

Al igual que para el caso de los primos 1-seguros, hemos realizado un experimento numérico en el que tratamos de establecer el tiempo real de ejecución y comprobar el ajuste de la predicción teórica a esos tiempos de computación.

El experimento consiste en obtener 10 primos 2-seguros cuya longitud en bits está ahora comprendida entre 125 y 335 bits, con un paso de 5 bits, lo que da un total de 430 primos 2-seguros. Hemos elegido este rango porque está próximo a 256 bits.

El equipo utilizado ha sido el mismo que en el caso anterior, una plataforma PC estándar con las siguientes características:

1. Sistema operativo: Microsoft Windows XP, versión 2002, service pack 1.
2. Biblioteca Cygwin (véase [93]), que permite emular el entorno de programación y desarrollo de Unix. Da soporte a la mayoría de los servicios declarados en el estándar POSIX.
3. Procesador: Intel Pentium III, a 731 MHz.
4. Memoria: 256 MB.

Desarrollamos un programa escrito en C y apoyado en la biblioteca de programación en multiprecisión GMP (véase [48]). Ofrecemos el código original en el Apéndice I.

Comenzamos con el número de tiradas que ha sido necesario, en promedio, para obtener un primo 2-seguro de un determinado número de bits. Teóricamente, el número de tiradas $T_2(n)$ necesarias se puede obtener a partir de la fórmula (6.5), de forma que para un primo 2-seguro de n bits, se tendría

$$T_2(n) \simeq \frac{8}{9C_2} \cdot \ln 2^n \cdot \ln \frac{2^n - 1}{2} \cdot \ln \frac{2^n - 3}{4} \simeq \frac{8}{9C_2} (\ln 2)^3 n(n-1)(n-2),$$

es decir, T_2 es aproximadamente cúbica en n . Para contrastar esto, presentamos en la gráfica 6.4 el número promedio de tiradas frente al número de bits, en línea discontinua, junto con la gráfica de la función βn^3 , en línea continua.

En teoría, el valor de β debe ser igual a la relación

$$\beta = \frac{T_2(n)}{n(n-1)(n-2)} \simeq \frac{8}{9C_2} (\ln 2)^3 \simeq 0,4660544061. \quad (6.6)$$

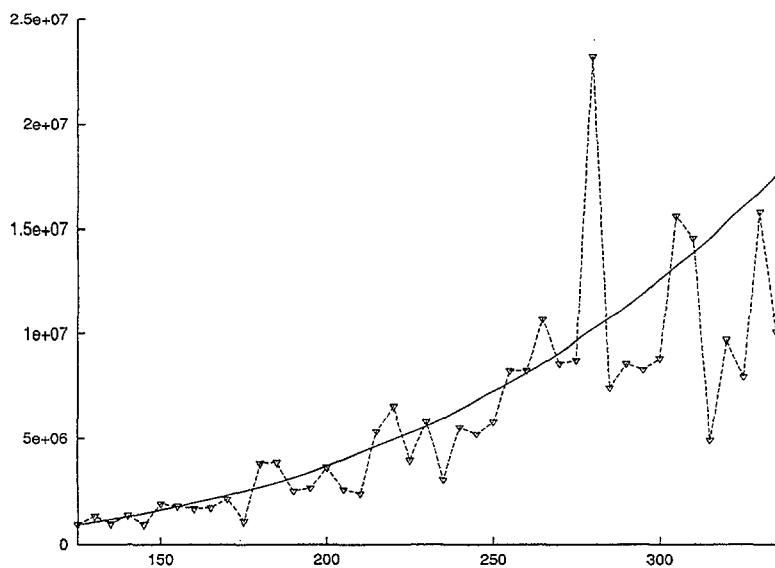


Figura 6.4: Tiradas T_2 frente a número de bits

Para comprobarlo, presentamos también la gráfica 6.5 que representa, con línea discontinua, los valores de la relación (6.6) para cada punto experimental y, en línea continua, el valor teórico β de la misma ecuación (6.6). Se observa también una notable coincidencia.

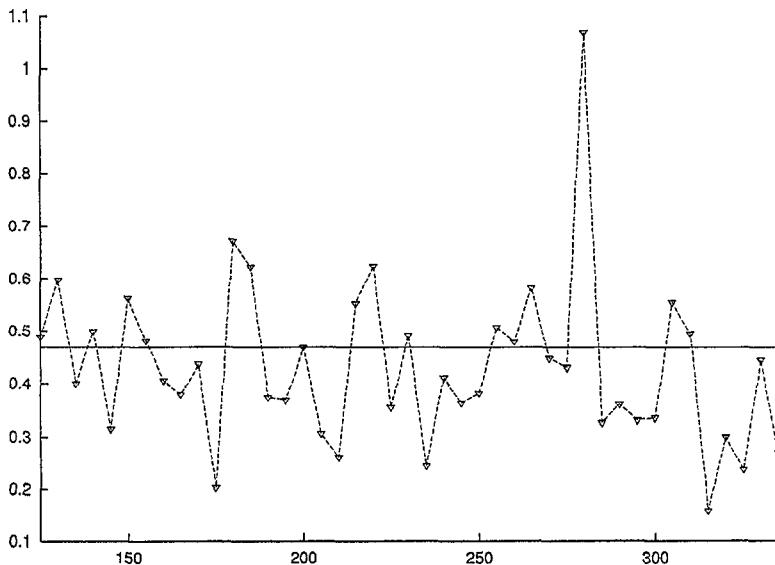


Figura 6.5: Valores teórico y experimental de la razón β

El siguiente paso es estudiar los tiempos experimentales de computación. A continuación, presentamos una tabla en que se puede ver el tiempo promedio en segundos necesario para obtener un primo 2-seguro de un número determinado de bits en el presente experimento.

Número de bits	Tiempo en segundos
125	623,7589
130	805,0486
135	625,0307
140	889,4409
145	639,0148
150	1278,5234
155	1250,2968
160	1243,6873
165	1302,1694
170	1654,2386
175	858,3572
180	3227,8133
185	3829,0739
190	2375,4337
195	2717,0739
200	4008,3707
205	2703,5635
210	2690,1862
215	6116,3188
220	7612,3319
225	4769,4291
230	6809,2372
235	3749,2942
240	6534,7374
245	6110,7528
250	7598,1786
255	11590,0056
260	17085,5577
265	16466,3174
270	13675,5314
275	14329,2123
280	35975,8947
285	12852,8094
290	15092,3407
295	15302,7602
300	16577,4371
305	29140,8464
310	28756,6059
315	9817,8093
320	19687,6855
325	16691,7024
330	33300,2143
335	21788,3451

En la figura 6.6 presentamos gráficamente estos mismos datos. Proporcionamos dos gráficas: por un lado están los resultados numéricos, representados por la línea

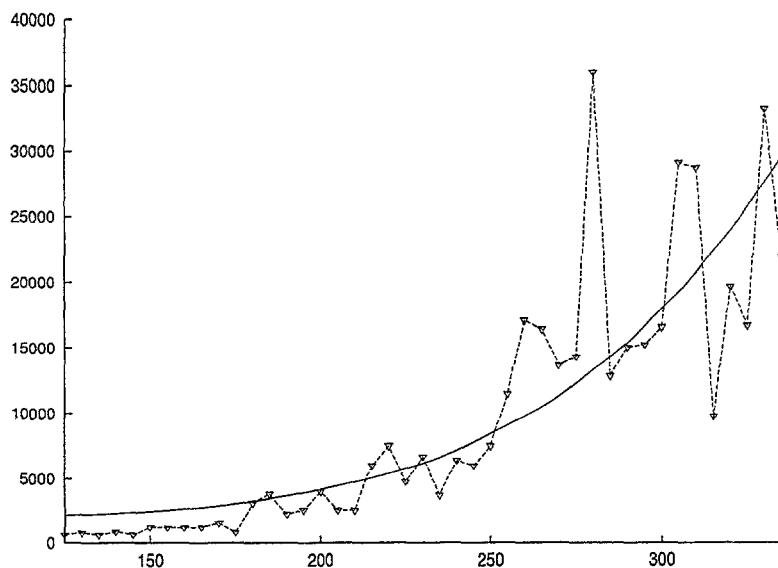


Figura 6.6: Tiempo de ejecución en segundos frente a número de bits

quebrada discontinua, con triángulos en los puntos experimentales; la línea de trazo continuo representa una aproximación dada por el siguiente polinomio:

$$t = an^5 + b, \quad (6.7)$$

donde

$$a = 0,650835993829727869 \cdot 10^{-8}$$

$$b = 2101,168124$$

y n representa el número de bits. Las aproximaciones también se han realizado utilizando el módulo para ajuste parabólico LeastSquares, del programa de computación simbólica MAPLE. De acuerdo con la Proposición 6.16, la aproximación de grado 5 es la que prescribe la teoría y, como se aprecia claramente en la figura 6.6, resulta muy ajustada, lo que viene a confirmar la validez del resultado teórico.

Para resumir, podemos dar ya una estimación del tiempo real necesario para obtener un primo 2-seguro en el intervalo $I = [240, 260]$, de acuerdo a los resultados del experimento:

1. Valor promedio en I de acuerdo a la tabla experimental: 9783,846 segundos.
2. Valor máximo en I de acuerdo a la tabla experimental: 17085,557700 segundos.
3. Valor según el polinomio (6.7) evaluado en $n = 250$: 8456,988 segundos.

Como conclusión, se puede decir que para obtener un primo 2-seguro en un entorno de 256 bits se necesitan como máximo 2,5 horas de computación en una plataforma estándar utilizando el algoritmo 4.42.

6.6.4 Algoritmo mejorado para primos 2-seguros

En totalanalogía al caso de los primos 1-seguros, el algoritmo 4.42 admite también una mejora inmediata, basada en idéntico principio. Se tiene la siguiente

Proposición 6.19 Si $p > 5$ es un primo 2-seguro, entonces $p \equiv 7 \pmod{8}$.

Demostración De acuerdo con la Definición 4.1 y la Notación 4.2, p es un primo 2-seguro si y sólo si $p \in \mathbb{P}(+1, +1)$. Aplicando entonces el resultado de la Proposición 4.3, se concluye inmediatamente. ■

Esta proposición nos permite mejorar el algoritmo simplemente forzando a que los candidatos a ensayar siempre cumplan la condición anterior. Pero esto es inmediato de conseguir, pues ello significa que los tres bits menos significativos de su representación binaria han de ser siempre 1. El algoritmo quedaría así:

Algoritmo 6.20 Dados $n, t \in \mathbb{N}$, este algoritmo consigue un primo 2-seguro de n bits. El test de primalidad se realiza apoyándose en el algoritmo de Miller-Rabin con valor t para el parámetro de seguridad.

LLAMADA: `Primo2Seguro(n, t);`

ENTRADA: Un entero n y un parámetro de seguridad t .

SALIDA: Un número primo aleatorio de n bits 2-seguro con probabilidad $1 - 2^{-2t}$.

1. [Inicialización]

Se elige la semilla para el generador aleatorio BBS.

2. [Lazo]

```
while (encontrado == NO)
```

```
{
```

```
    p = NumeroAleatorioBBS(n - 3);
```

```
    p = 8p + 7;
```

```
    if (MillerRabin(p, t))
```

```
{
```

$$q_1 = \frac{p-1}{2};$$

```
    if (MillerRabin(q_1, t))
```

```
{
```

$$q_2 = \frac{q_1-1}{2};$$

```
    if (MillerRabin(q_2, t))
```

```
{
```

```
        encontrado = SI;
```

```
    return p;
```

```
}
```

```

    }
}

}

```

Ahora la diferencia consiste en generar un número aleatorio con tres bits menos de lo necesario que después aparecen tras multiplicarlo por 8 y sumarle 7. Con esto, el número de ensayos queda disminuido a la octava parte, pues el espacio muestral se reduce a los números que están en la clase de equivalencia del 7 módulo 8. Por consiguiente, es de esperar que también se reduzca a una octava parte el tiempo necesario para la ejecución del algoritmo.

6.7 Tiempo de ejecución para un primo robusto óptimo

En analogía con los casos anteriores, vamos, en primer lugar, a dar una estimación teórica acompañada de algunos ejemplos reales, acerca del número de tiradas necesarias para obtener un primo robusto óptimo.

En segundo lugar, obtendremos una estimación teórica acerca del tiempo de computación necesario para obtener en la práctica un primo robusto óptimo. En este caso hemos utilizado el generador pseudoaleatorio de Lehmer, que hemos introducido en la sección 3.4.2 como algoritmo 3.21.

Por último, presentamos datos experimentales acerca del tiempo de computación que ha sido necesario en la práctica para obtener primos robustos óptimos de diferentes tamaños.

6.7.1 Estimación del número de tiradas

Proposición 6.21 El número promedio de tiradas necesarias para obtener un primo robusto óptimo p es

$$\frac{12}{C_\sigma} \cdot \left(\ln \frac{p-7}{12} \right)^4, \quad (6.8)$$

donde C_σ es la constante introducida en la fórmula (5.2) de la sección 5.2.3.

Demostración Se deduce directamente de la fórmula (5.3), que da la densidad de los primos robustos óptimos. ■

Ejemplo 6.22 Para obtener un primo robusto óptimo de n bits será necesario efectuar un promedio de

$$\frac{12}{C_\sigma} \cdot \left(\ln \frac{2^n - 7}{12} \right)^4 \simeq \frac{12}{C_\sigma} (\ln 2)^4 n^4 \simeq 0,5004636359 \cdot n^4$$

tiradas. Para $n = 256$, se tiene

$$0,5004636359 \cdot 256^4 \simeq 2,15 \cdot 10^9$$

tiradas.

Observación 6.23 Como se ha visto en la sección 5.2.3, la densidad π_σ de los primos robustos óptimos es muy baja, por lo que el número de tiradas es alto cuando se requiere un primo robusto de un tamaño significativo.

Ejemplo 6.24 Como ejemplo, damos tres primos robustos óptimos junto con el número de tiradas necesarias para obtenerlos.

<i>p</i> robusto óptimo	Número de tiradas
258892550563	51414
25764555802960401403	284551
11625406576232098882640943094056507883	26475400

6.7.2 Tiempo de ejecución teórico

Se tiene la siguiente

Proposición 6.25 El algoritmo 5.33 efectúa $O(n^6)$ operaciones bit para obtener un primo robusto óptimo de n bits.

Demostración Del enunciado de la proposición se sigue que $p \simeq 2^n$, es decir, $n \simeq \log_2 p$. Observemos en primer lugar que el número promedio de tiradas necesarias para encontrar un primo robusto óptimo es, de acuerdo con la fórmula (6.1), $O(n^4)$. Por lo tanto, ése será el número de veces que, en promedio, será necesario iterar el lazo del algoritmo.

Ahora bien, dentro del lazo se genera un número aleatorio, usando el generador de Lehmer, y se llegan a realizar hasta cuatro comprobaciones de primalidad utilizando el algoritmo de Miller-Rabin modificado. Sin embargo, la mayor parte de las veces, de hecho $O(n^4)$, lo que ocurre es que se realiza una única comprobación de primalidad, la primera, tal que el candidato no es primo, es decir, sin éxito. Recordemos que el generador pseudoaleatorio corre en $O(n)$ (véase Proposición 3.22 de la sección 3.4.2) y que las comprobaciones de primalidad sin éxito corren en $O(n^2)$ (véase sección 3.2.3). Por lo tanto, el tiempo de computación esperado será

$$O(n^4)(O(n) + O(n^2)) = O(n^6).$$

6.7.3 Datos experimentales

Para comprobar nuestros resultados, hemos diseñado un experimento numérico que hace uso del algoritmo 5.33 descrito en 5.3.3. Hemos calculado primos robustos óptimos para diferentes tamaños en bits.

Como se ha dicho más arriba, para generar el número aleatorio t con el tamaño en bits requerido con que comienza el algoritmo, nos hemos servido el generador de números aleatorios de Lehmer; como test de primalidad hemos usado el test probabilístico de Miller-Rabin con valor del parámetro de seguridad igual a 10 igual que en los casos anteriores. Recordemos que el algoritmo genera números t sobre los que aplica sus condiciones, hasta obtener un primo robusto óptimo

$p = 12t + 7$. Para nuestro experimento, hemos elegido generar valores de t en el rango desde 21 a 120 bits, para ensayar con ellos hasta obtener un total de 20 primos robustos óptimos para cada longitud en bits. Observemos que el tamaño de cada primo robusto óptimo va a ser del orden de tres bits más largo que el de t , puesto que $p = 12t + 7$.

El equipo utilizado ha sido el mismo que en el caso anterior, una plataforma PC estándar con las siguientes características:

1. Sistema operativo: Microsoft Windows XP, versión 2002, service pack 1.
2. Biblioteca Cygwin (véase [93]), que permite emular el entorno de programación y desarrollo de Unix. Da soporte a la mayoría de los servicios declarados en el estándar POSIX.
3. Procesador: Intel Pentium III, a 731 MHz.
4. Memoria: 256 MB.

Usando la biblioteca GMP hemos desarrollado en lenguaje C el algoritmo, cuyo código se puede encontrar en el Apéndice I. Como de costumbre, el primer dato de interés es el número de tiradas que, en promedio, es necesario realizar para obtener un primo robusto óptimo de un determinado número de bits. A partir de la fórmula (6.8), se tiene que el número de tiradas teóricas T_s para obtener un primo robusto óptimo p de n bits es

$$T_s(n) = \frac{12}{C_\sigma} \cdot \left(\ln \frac{2^n - 7}{12} \right)^4 \simeq \frac{12}{C_\sigma} (\ln 2)^4 n^4,$$

es decir, es un polinomio de grado 4 en n .

Ahora bien, estudiando detenidamente el algoritmo 5.33, se ve que funciona de una manera distinta a como lo hacían los que hemos utilizado para los primos seguros. En efecto, mientras que éstos son puramente *heurísticos* (es decir, se limitan a buscar al candidato), el algoritmo de los primos robustos óptimos es *constructivo*, es decir, trata de construir un número que vaya satisfaciendo las condiciones impuestas. En este caso, el algoritmo 5.33 comprueba sucesivamente la primalidad sobre los siguientes números:

$$\begin{array}{c} t \\ 2t + 1 \\ 3t + 2 \\ 12t + 7 \end{array}$$

Si pasa los cuatro ensayos, el número $p = 12t + 7$ es considerado primo robusto óptimo. Ahora bien, ello indica que no estamos ensayando las condiciones para ser robusto óptimo sobre cualquier número p , sino sobre aquellos que sean de la forma $p = 12t + 7$ o, dicho de otro modo, que $p \equiv 7 \pmod{12}$. Evidentemente sólo hay 1/12 enteros en esta situación y, por tanto, la cantidad de ensayos, o tiradas

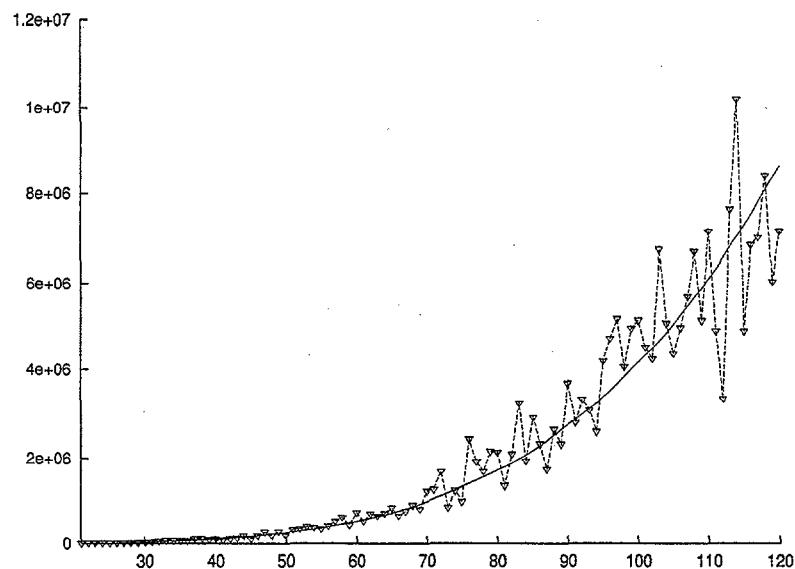


Figura 6.7: Tiradas T_s frente a tamaño en bits

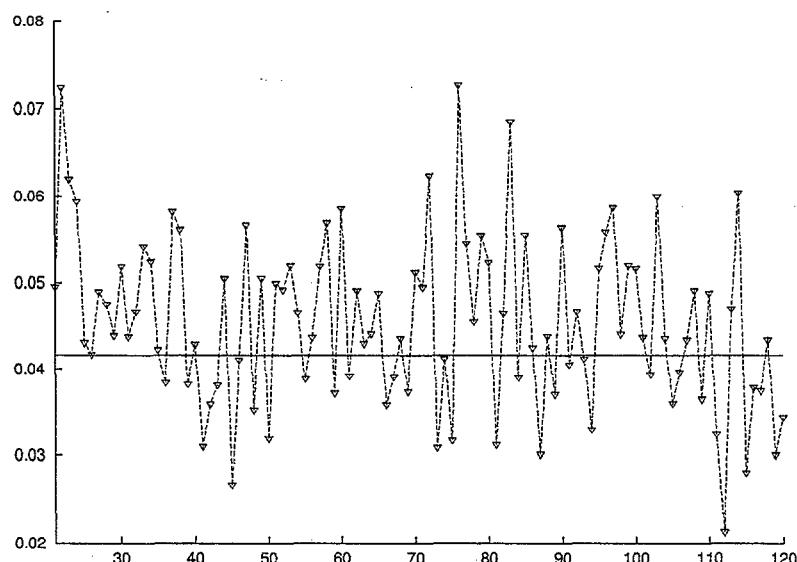


Figura 6.8: Valores teórico y experimental de la razón γ

que serán necesarias se verá reducida en igual proporción. Así pues, el número esperado de tiradas para que el algoritmo 5.33 tenga éxito será

$$T_s(n) \simeq \frac{(\ln 2)^4}{C_\sigma} n^4. \quad (6.9)$$

En la gráfica de la figura 6.7, se presentan, con línea discontinua, los valores medios del número de tiradas T_s frente al tamaño en bits n pretendido; así, cada punto representa una aproximación experimental al número de tiradas necesario para conseguir un primo de estas características con un tamaño en bits dado. Con linea continua representamos la función γn^4 .

En teoría, de acuerdo con la fórmula (6.9), el valor de γ debiera ser

$$\gamma = \frac{T_s(n)}{n^4} \simeq \frac{(\ln 2)^4}{C_\sigma} \simeq 0,0417053. \quad (6.10)$$

Para comprobarlo, presentamos también la gráfica 6.8 que representa, con línea discontinua, los valores de la relación (6.10) para cada punto experimental y, en línea continua, el valor teórico γ de la misma ecuación (6.10). Se observa que la concordancia es bastante aceptable.

El siguiente paso es, como en los casos anteriores, estudiar los tiempos experimentales de computación. A continuación, presentamos una tabla en que se puede ver el tiempo promedio en segundos necesario para obtener un primo robusto óptimo de un número determinado de bits para t en el presente experimento.

Número de bits de t	Tiempo en segundos
21	4,786
22	8,301
23	8,422
24	11,666
25	10,144
26	11,095
27	16,563
28	17,124
29	18,947
30	27,830
31	24,925
32	31,865
33	52,565
34	55,810
35	53,436
36	54,248
37	87,866
38	91,812
39	68,368
40	84,431
41	68,298
42	87,265
43	103,068

Número de bits de t	Tiempo en segundos
44	145,899
45	84,511
46	141,623
47	214,999
48	146,450
49	232,354
50	161,021
51	270,208
52	287,253
53	327,530
54	317,806
55	286,381
56	349,662
57	448,995
58	528,590
59	373,256
60	628,193
61	452,580
62	619,090
63	583,268
64	633,460
65	913,203
66	725,112
67	844,394
68	992,346
69	918,560
70	1317,414
71	1344,953
72	1796,843
73	946,110
74	1328,700
75	1079,892
76	2628,780
77	2089,524
78	1823,772
79	2379,050
80	2352,032
81	1496,892
82	2359,342
83	3614,797
84	2148,779
85	3188,885
86	2558,048
87	1903,867
88	2903,485
89	2571,758

Número de bits de t	Tiempo en segundos
90	4106,665
91	3140,175
92	3785,763
93	3524,578
94	2971,653
95	4891,774
96	5711,222
97	7278,175
98	5909,857
99	7353,603
100	7645,924
101	6754,362
102	6269,014
103	10390,580
104	7443,763
105	6501,859
106	7448,961
107	8502,926
108	10016,923
109	7773,848
110	10819,327
111	7445,205
112	5091,401
113	11752,419
114	15502,691
115	7529,747
116	10462,354
117	10812,717
118	13025,489
119	9433,464
120	11141,440

En la figura 6.9 presentamos gráficamente estos mismos datos. Proporcionamos dos gráficas: por un lado están los resultados numéricos, representados por la línea quebrada discontinua, con triángulos en los puntos experimentales; la línea de trazo continuo representa una aproximación dada por el siguiente polinomio:

$$t = an^6 + b, \quad (6.11)$$

donde

$$\begin{aligned} a &= 0,451662078782578726 \cdot 10^{-8} \\ b &= 568,9976028 \end{aligned}$$

y n representa el número de bits. Las aproximaciones también se han realizado utilizando el módulo para ajuste parabólico LeastSquares, del programa de computación simbólica MAPLE. De acuerdo con la Proposición 6.25, la aproximación

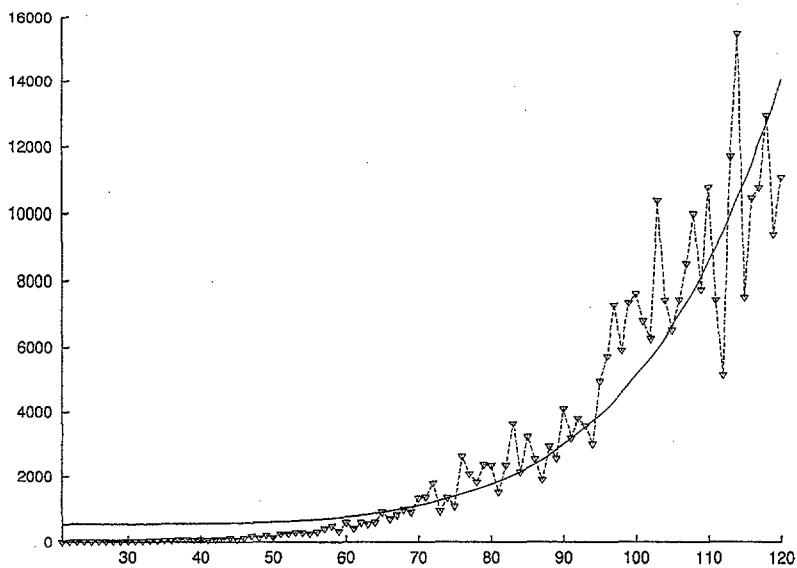


Figura 6.9: Tiempo de ejecución en segundos frente a número de bits

de grado 6 es la que prescribe la teoría y, como ve en la figura 6.9, el resultado parece bastante aceptable, en consonancia con lo predicho teóricamente.

Para resumir, podemos dar ya una estimación del tiempo real necesario para obtener un primo robusto óptimo en el intervalo $I = [100, 120]$, de acuerdo a los resultados del experimento:

1. Valor promedio en I de acuerdo a la tabla experimental: 10274,205 segundos.
2. Valor máximo en I de acuerdo a la tabla experimental: 15502,69100 segundos.
3. Valor según el polinomio (6.11) evaluado en $n = 110$: 8570,466843 segundos.

A la vista de los datos, se puede obtener un primo robusto óptimo de 110 bits en un tiempo aproximado de 3 horas en una plataforma estándar utilizando el algoritmo 5.33 descrito en la sección 5.3.3.

6.7.4 Conjetura sobre la función recuento π_σ^*

Como se ha visto en la sección 5.2.3, la densidad π_σ de los primos robustos óptimos es muy baja. Nos planteamos la siguiente cuestión: ¿cuál será la densidad de primos robustos óptimos en el conjunto de los primos 1-seguros? La pregunta tiene sentido, pues recordando las condiciones del Corolario 5.15, si p es un primo robusto óptimo, entonces $r = (p - 1)/6$ resulta ser un primo 1-seguro.

Para tratar de contestar a esta pregunta, hemos acudido a la experiencia numérica. Realizamos una computación en que se examina el porcentaje de primos 1-seguros r tales que $p = 6r + 1$ resulta ser un primo robusto óptimo.

Hemos elegido unos rangos de bits representativos desde 10 a 120, estructurando las tablas en los recorridos especificados en cada una de ellas; para cada tabla se han generado un total de 1000 primos 1-seguros.

Porcentajes de primos robustos óptimos

Recorrido	entre 10 y 20 bits
Promedio	2,0000 %
Tiradas	50
t_{inicial}	1031
t_{final}	1048576

Recorrido	entre 30 y 40 bits
Promedio	0,8150 %
Tiradas	123
t_{inicial}	1073741827
t_{final}	1099511627776

Recorrido	entre 50 y 60 bits
Promedio	0,2368 %
Tiradas	422
t_{inicial}	576460752303423619
t_{final}	1152921504606846976

Recorrido	entre 79 y 80 bits
Promedio	0,1309 %
Tiradas	764
t_{inicial}	604462909807314587353111
t_{final}	1208925819614629174706176

Recorrido	entre 99 y 100 bits
Promedio	0,0849 %
Tiradas	1178
t_{inicial}	633825300114114700748351602943
t_{final}	1267650600228229401496703205376

Recorrido	entre 119 y 120 bits
Promedio	0,0612 %
Tiradas	1635
t_{inicial}	664613997892457936451903530140172297
t_{final}	1329227995784915872903807060280344576

Observación 6.26 Esta tabla ha de interpretarse en el sentido de que en cada rango de bits considerado, se obtiene el porcentaje de primos robustos óptimos indicado en la columna “Promedio” por cada 100 primos 1-seguros obtenidos en ese rango; así, por ejemplo, para números entre 10 y 20 bits, se obtienen, en promedio, 2 primos robustos óptimos por cada 100 primos 1-seguros comprobados en ese rango.

Observación 6.27 Como se ve, los tiempos de computación de primos robustos óptimos son muy altos; así, por ejemplo, de la última tabla anterior se sigue que para obtener un primo robusto óptimo de aproximadamente 120 bits hay que

realizar un promedio de $100/0,0612 \simeq 1635$ tiradas en el conjunto de los primos 1-seguros, tal y como se recoge en la tabla. Y no olvidemos que el rango de 120 bits resulta hoy en día pequeño para los tamaños requeridos en las aplicaciones criptográficas de clave pública reales, donde se manejan tamaños de hasta varios miles de bits.

Observación 6.28 De la fórmula (5.4) en la sección 5.2.3, y de las observaciones que nos han conducido a la fórmula (6.9) de arriba, se deduce que el número necesario de tiradas para obtener un primo robusto óptimo próximo a $n = 12t + 7$ es del orden de

$$\frac{1}{C_\sigma} \ln(12t + 7) \cdot \ln(2t + 1) \cdot \ln(3t + 2) \cdot \ln t.$$

Por otro lado, de la fórmula (4.12) de la sección 4.2.3 se sigue que el número necesario de tiradas para obtener un primo 1-seguro cercano a $(n - 1)/6 = 2t + 1$ es del orden de

$$\frac{1}{C} \ln t \cdot \ln(2t + 1).$$

en donde hemos tenido en cuenta que los valores aleatorios son para t y por lo tanto, hacen falta la mitad de tiradas que las predichas por la fórmula (4.12). Parece, pues, razonable suponer que el cociente entre ambas cantidades debiera estar próximo al número necesario de tiradas para obtener un primo robusto óptimo próximo a $n \in \mathbb{P}_1^+$. Ese cociente vale

$$q_n = \frac{C}{C_\sigma} \ln n \cdot \ln\left(\frac{n+1}{4}\right). \quad (6.12)$$

Aplicando esta fórmula a los valores de n considerados en las tablas anteriores, se obtienen los siguientes resultados:

n	q_n	Tiradas	Tiradas/ q_n
2^{35}	132	123	0,93
2^{55}	334	422	1,26
2^{80}	715	764	1,07
2^{100}	1123	1178	1,05
2^{120}	1623	1635	1,01

Observación 6.29 Está claro a la vista de estos datos que el número de tiradas predichas, q_n , concuerda sensiblemente bien con el realmente necesario. Parece, pues, que aun moviéndonos en el terreno de las conjeturas, los datos experimentales se ajustan notablemente bien a lo predicho y no es arriesgado afirmar que el número de tiradas promedio para obtener un primo robusto óptimo es $\mathcal{O}((\ln n)^4)$.

Observación 6.30 Cabe preguntarse, llegados a este punto, cuál sería la formulación teórica que podría conducirnos a una justificación de la fórmula (6.12). En efecto, planteado teóricamente, el problema consiste en deducir cuál es la densidad de primos robustos óptimos existentes en el conjunto \mathbb{P}_1^+ de los primos 1-seguros, y qué valor tendría una función recuento $\pi_\sigma^*: \mathbb{P}_1^+ \rightarrow \mathbb{R}$ definida como

$$\pi_\sigma^*(p) = \#\{q \text{ robusto óptimo} : q \leq p, q \in \mathbb{P}_1^+\},$$

o, dicho con palabras, cuál es el número de primos robustos óptimos de entre todos los primos 1-seguros menores que uno dado. De modo totalmente heurístico, podríamos conjutar, a la vista de los resultados, que tal cantidad se puede estimar en

$$\pi_\sigma^*(p) \sim \frac{C_\sigma}{C} \int_{31}^p \frac{du}{\ln \frac{u+1}{4} \cdot \ln u}.$$

Capítulo 7

Conclusiones, aportaciones y desarrollos futuros

7.1 Conclusiones

El objetivo de esta memoria ha sido el estudio de las propiedades de varias clases de números primos especiales y su aplicación a diversos criptosistemas de clave pública.

En el capítulo 2 hemos realizado un repaso de los criptosistemas de clave pública más importantes y utilizados en la actualidad, incluyendo también algunas variantes menos frecuentes. Hemos presentado una comparación con los tradicionales criptosistemas de clave secreta.

De este estudio se concluye que los criptosistemas de clave pública complementan muy adecuadamente a los sistemas tradicionales de clave secreta aportando soluciones a los problemas que éstos plantean, como la carencia de firma digital o la dificultad de la distribución de claves; presentan, sin embargo, otros problemas, como la lentitud de operación y la larga longitud de las claves, por lo que no los sustituyen totalmente.

Los criptosistemas de clave pública se basan en la dificultad de resolver ciertos “problemas matemáticos”, como la factorización de enteros grandes o la extracción del logaritmo discreto y requieren el uso de números primos para su funcionamiento. Estos números primos suelen formar parte de las “claves” (entendido este término en forma genérica) que necesita cada sistema. Ahora bien, a lo largo de los años, también se han desarrollado ciertas técnicas que resuelven en parte esos “problemas”. Se concluye que es necesaria la utilización de primos dotados de ciertas propiedades especiales para frustrar esas técnicas.

Del estudio de las propiedades especiales requeridas y las técnicas existentes que presentamos en el capítulo 3, se concluye que las clases de primos interesantes como objeto de estudio son las siguientes:

1. Los primos 1-seguros;
2. los primos 2-seguros;
3. los primos k -seguros;

4. los primos robustos.
5. los primos robustos óptimos.

Llegamos así al problema de si es posible obtener en la práctica primos de las clases mencionadas. Lo acometemos a lo largo de los capítulos 4 y 5.

Como observaciones comunes a los resultados obtenidos para todas las clases de primos estudiadas, podemos afirmar:

- hemos tratado de justificar razonablemente los resultados teóricos que hemos obtenido, pero sin aportar en la mayoría de los casos demostraciones matemáticas totalmente rigurosas. De hecho, muchos resultados comúnmente admitidos no están rigurosamente probados, como, por ejemplo, la existencia de infinitos números primos 1-seguros.
- los resultados que hemos obtenido se fundamentan en conjeturas clásicas, avaladas por el tiempo;
- hemos contrastado esos resultados con la experimentación numérica, y hemos encontrado en todos los casos una precisión muy aceptable.

En el caso de los primos seguros, hemos extendido su definición, de forma que se puede hablar de primos k -seguros de cierta signatura. Como primera conclusión cabe destacar que las signaturas alternadas no son interesantes, pues resultan vacías si $k \geq 5$ y son conjuntos muy pequeños si $k < 5$. Así pues, centramos nuestro estudio en las signaturas o bien positivas o bien negativas.

Comenzamos con el estudio de la densidad de primos 1-seguros que se apoya en conjeturas clásicas bien establecidas y trabajos anteriores. Para ellos, aportamos la función recuento y su densidad, lo que permite predecir cuál es el esfuerzo computacional para obtener uno de ellos de un tamaño fijado. A la vista de esas funciones, destacamos como conclusión que todo apunta a la existencia de infinitos primos 1-seguros. También queda claro que la densidad de primos 1-seguros de signatura positiva es sustancialmente idéntica a los de signatura negativa.

Pasamos a continuación al estudio de los primos 2-seguros. Usando conjeturas clásicas junto con razonamientos propios, obtenemos unas fórmulas para la función recuento y densidad, que apuntan a idénticas conclusiones que en el caso anterior, es decir, la existencia de infinitos primos 2-seguros, por un lado, y, por otro, que la densidad correspondiente a las signaturas positiva y negativa es virtualmente idéntica. Igualmente, se hace posible predecir el esfuerzo computacional para obtener un primo 2-seguro. Para uno y otro casos, todo ello viene avalado por experimentos numéricos *ad hoc*.

El estudio general de los primos k -seguros indica que se puede conjeturar razonablemente la existencia de una infinidad de ellos aunque, al aumentar el valor de k , la densidad se enrarece. Se puede concluir, sin embargo, que esa densidad nunca es tan pequeña como para impedir el uso criptográfico de esta clase de primos.

Al estudiar los primos robustos, la primera conclusión a la que se llega es que la definición estándar es deficiente, por ser meramente cualitativa. Además no

existe un acuerdo unánime en la comunidad de criptógrafos respecto al interés de su uso. Nosotros pensamos que pueden mantener su interés si conseguimos aportar una definición cuantitativa y evidenciar que su densidad es lo suficiente alta como para permitir su uso criptográfico.

Para complementar la noción de primo robusto, hemos introducido la función σ , que mide en cierto modo la “robustez” del candidato a primo robusto. Esta función nos ha permitido dar un paso más, presentando la noción de *primo robusto óptimo*, aquel para el que la función σ presenta un mínimo. Estudiando dos clásicos algoritmos de generación de primos robustos, llegamos a la conclusión de que los primos con ellos generados distan mucho de ser óptimos y pasamos a proponer un algoritmo que sí los genera.

Análogamente al caso de los primos seguros, estudiamos la función recuento y densidad de los primos robustos óptimos. Concluimos que, si bien esta densidad es más bien baja, no impide en absoluto su uso criptográfico y razonablemente invita a pensar que existen una infinidad de ellos. Además, las fórmulas aportadas permiten predecir el esfuerzo computacional requerido para generar uno de ellos con tamaño fijado. Estas conclusiones son, como siempre, conjeturales, pero vienen avaladas por los resultados numéricos de los experimentos diseñados para confirmarlas.

Como conclusión final, se puede afirmar que es posible y razonablemente económica la utilización de primos de las clases estudiadas en el rango de bits exigido por las aplicaciones actuales de los criptosistemas de clave pública. Para el caso de los primos robustos óptimos, esta afirmación debe ser matizada, pues resultan mucho más caros en tiempo de cálculo que los robustos ordinarios, como se explicó en el capítulo 5.

7.2 Aportaciones

Las aportaciones de esta memoria pueden dividirse en dos partes diferenciadas: teóricas y prácticas.

Las aportaciones más teóricas han sido:

- El estudio sistemático de la clase de primos seguros, estimando sus funciones recuento y, por tanto, sus densidades para los primos 1-, 2- y k -seguros. Estos resultados se derivan a partir de ciertas conjeturas clásicas de la teoría de números.
- El estudio sistemático de la clase de primos robustos, estimando su función recuento y, por tanto, su densidad. Estos resultados también se derivan a partir de conjeturas clásicas de la teoría de números.
- La definición de la función de argumento entero σ que permite cuantificar la robustez de un primo robusto y establecer comparaciones. La caracterización de esta función, estableciendo en particular su valor mínimo y para qué valores de su argumento se obtiene.
- La mejora de la definición de primo robusto, aportando un matiz cuantitativo derivado del uso de la función σ .

- La noción de primo robusto óptimo, como aquel para el que la función σ proporciona su valor mínimo.

Las aportaciones más prácticas, basadas en las anteriores, pueden resumirse así:

- Implementaciones reales de los algoritmos que permiten obtener las clases de primos especiales tratadas. Estas implementaciones se han realizado usando como herramienta la biblioteca de multiprecisión GMP, aunque es relativamente sencillo utilizar cualquier otra.
- Confirmación, mediante experimentos numéricos, de la exactitud de los resultados teóricos antes descritos.
- Datos estadísticos acerca del tiempo de computación necesario para obtener primos especiales sobre una plataforma de computación estándar. Esto permite estimar con precisión en la práctica el tiempo necesario para obtener uno de ellos.

Se ofrece también en el Apéndice I el código fuente de programación para todos los algoritmos presentados, tanto de los originales como de los implementados.

7.3 Desarrollos futuros

Como ya hemos comentado más arriba, los desarrollos en la teoría de números se han llevado a cabo lentamente y bastantes conjeturas resisten impertérritas todos los intentos por demostrarlas. Ciento que el camino recorrido es mucho y, para algunas de ellas, la demostración final está cercana.

Con este breve exordio queremos justificar por qué no es tan sencillo traducir en demostraciones rigurosas los resultados teóricos que hemos presentado en esta memoria, basados principalmente en razonamientos heurísticos y conjeturales.

Teniendo en cuenta las anteriores precisiones, sería deseable proseguir la investigación que condujera a una demostración rigurosa de algunas de las conjeturas:

- como se dijo en la sección 2.5.1, aunque es claro que factorizar el módulo es una forma de infringir el criptosistema RSA, no está demostrado que la recíproca sea cierta, es decir, que el sistema se infringe sólo mediante la factorización del módulo;
- asimismo, en la sección 2.1.3, se explicó que la solución del problema del logaritmo discreto implica la solución del problema de Diffie-Hellman, pero no se sabe si, en general, el recíproco es cierto. En principio, podría caber la posibilidad de que un criptoanalista calculara α^{ab} por otro método que no fuera el de resolver la ecuación logarítmica.

Como se explicó en las secciones correspondientes, no está probada la existencia de que existan infinitos primos de las clases especiales consideradas, es decir, seguros y robustos. Sería interesante cualquier avance en este sentido.

Como resultado concreto, de acuerdo a la Observación 4.17, se podría investigar si existen cadenas de primos seguros de longitud k arbitraria, o equivalentemente si para todo $k \in \mathbb{N}$ los conjuntos $\mathbb{P}_k^+, \mathbb{P}_k^-$ son no vacíos.

Puesto que los grandes sistemas se basan fundamentalmente en los dos problemas considerados, el problema de la factorización y el problema del logaritmo discreto, sería interesante obtener resultados más fuertes relativos a los algoritmos que resuelven esos problemas.

Los desarrollos futuros más interesantes son los ligados con la generación práctica de los primos especiales. Así como existen desarrollos (por ejemplo, [70]) que son capaces de generar primos probados con una distribución aproximadamente uniforme de forma mucho más eficiente que el algoritmo 3.27, sería deseable desarrollar algoritmos más eficaces para generar primos especiales de las clases consideradas, es decir, los seguros y los robustos.

Otra línea de trabajo es conseguir bajar los tiempos de computación necesarios para la generación de los primos especiales. Esta línea admite dos caminos paralelos:

- Mejorar el tiempo de computación del algoritmo de comprobación de primalidad. Como vimos a lo largo del capítulo 6, este tiempo es una de las principales contribuciones al tiempo total de generación; además, el algoritmo se utiliza repetitivamente, por lo que una pequeña mejora en su tiempo de ejecución repercute muy favorablemente en el tiempo de ejecución total.
- Investigar la existencia de zonas en que la densidad de primos especiales sea más alta. En el presente trabajo, hemos realizado un estudio de las densidades “a gran escala”. Sería interesante descender al detalle de la distribución “a pequeña escala”, que pudiera permitir seleccionar zonas de búsqueda donde la densidad sea mayor y, por tanto, el esfuerzo computacional menor.

Apéndice I

Resumen del capítulo

Se incluyen implementaciones reales de la mayoría de los algoritmos utilizados o referenciados a lo largo de la memoria. Utilizando las bibliotecas o aplicaciones referenciadas, estos programas funcionan tal como están.

I.1 Propósito

En este Apéndice se recogen implementaciones para la mayoría de los algoritmos que se han presentado a lo largo de la memoria.

Casi todos ellos se han implementado en lenguaje C, utilizando la biblioteca de multiprecisión GMP. En algunos casos, se ha utilizado el lenguaje propio de programación de la aplicación MAPLE, descrita en la sección 1.3.2.

Cuando se trata de implementaciones en C, damos por separado el cuerpo del algoritmo y la definición de su interfaz. Hemos empleado los siguientes símbolos para indicar el sentido de entrada/salida de las variables:

- => Indica variable de entrada, leída por la función.
- <= Indica variable de salida, escrita por la función.
- <=> Indica variable de entrada/salida, leída y escrita por la función.

I.2 Algoritmo 1.9 (Euclides)

Este algoritmo está implementado en C, utilizando la biblioteca GMP.

Definición de interfaz: euclid.h

```
# include <stdio.h>
# include "gmp.h"

void Euclides(
    mpz_t d,      /* <= Máximo común divisor de a y b */
    mpz_t a,       /* => Valor de a */
    mpz_t b,       /* => Valor de b */
);
```

Cuerpo del programa: euclid.c

```

void Euclides(mpz_t d,          /* Salida */
              mpz_t a, mpz_t b)  /* Entradas */
{
    mpz_t r;

    /* Si a es menor que b, se llama a sí mismo */
    if (mpz_cmp(a, b) < 0)
    {
        Euclides(d, b, a);
        return;
    }

    mpz_init(r);

    while (mpz_cmp_si(b, 0) != 0)
    {
        mpz_mod(r, a, b);
        mpz_set(a, b);
        mpz_set(b, r);
    }
    mpz_set(d, a);
    mpz_clear(r);
    return;
}

```

I.3 Algoritmo 1.14 (EuclidesExt)

Este algoritmo está implementado en C, utilizando la biblioteca GMP.

Definición de interfaz: euclidext.h

```

#include <stdio.h>
#include "gmp.h"

void EuclidesExt(
    mpz_t u,           /* <= Valor de u */
    mpz_t v,           /* <= Valor de v */
    mpz_t d,           /* <= Máximo común divisor de a y b */
    mpz_t a,           /* => Valor de a */
    mpz_t b            /* => Valor de b
);

```

Cuerpo del programa: euclidext.c

```

void EuclidesExt(mpz_t u, mpz_t v, mpz_t d, /* Salidas */
                  mpz_t a, mpz_t b)      /* Entradas */
{

```

```

mpz_t q, v1, v3, t1, t3;

mpz_set_ui(u, 1);
mpz_set(d, a);
if (mpz_cmp_si(b, 0) == 0)
{
    mpz_set_ui(v, 0);
    return;
}

mpz_init(q);
mpz_init(v1); mpz_init(v3);
mpz_init(t1); mpz_init(t3);

mpz_set_ui(v1, 0);
mpz_set(v3, b);

while (mpz_cmp_si(v3, 0) != 0)
{
    mpz_fdiv_q(q, d, v3);
    mpz_mod(t3, d, v3);
    mpz_mul(t1, q, v1); mpz_sub(t1, u, t1);
    mpz_set(u, v1);
    mpz_set(d, v3);
    mpz_set(v1, t1);
    mpz_set(v3, t3);
}
}

mpz_mul(v, u, a); mpz_sub(v, d, v); mpz_divexact(v, v, b);
mpz_clear(q);
mpz_clear(v1); mpz_clear(v3);
mpz_clear(t1); mpz_clear(t3);
return;
}

```

I.4 Algoritmo 1.44 (Jacobi)

Este algoritmo está implementado en C, utilizando la biblioteca GMP.

Definición de interfaz: jacobi.h

```

#include <stdio.h>
#include "gmp.h"

int Jacobi(
    mpz_t a,      /* => Valor de a */
    mpz_t n       /* => Valor de n */

```

);

Cuerpo del programa: jacobi.c

```
int Jacobi(mpz_t a, mpz_t n) /* Entradas */
{
    unsigned long int e = 0;
    int             s = 0;
    mpz_t           a1;
    mpz_t           n1;

    if (mpz_cmp_ui(a, 0) == 0)
        return 0;

    if (mpz_cmp_ui(a, 1) == 0)
        return 1;

    mpz_init_set(a1, a);
    mpz_init(n1);

    e = mpz_scan1(a, 0L);
    if (e > 0)
        mpz_tdiv_q_2exp(a1, a1, e);

    if (e%2)
    { /* e es impar */
        switch(mpz_fdiv_ui(n, 8))
        {
            case 1:
            case 7:
                s = 1;
                break;

            default:
                s = -1;
                break;
        }
    }
    else
    { /* e es par */
        s = 1;
    }

    if (mpz_fdiv_ui(n, 4) == 3 &&
        mpz_fdiv_ui(a1, 4) == 3)
        s = -s;
    mpz_mod(n1, n, a1);
```

```

if (mpz_cmp_ui(a1, 1) == 0)
{
    mpz_clear(a1);
    mpz_clear(n1);
    return s;
}
else
    return s*Jacobi(n1, a1);
}

```

I.5 Algoritmo 3.5 (TestMillerRabin)

Este algoritmo está implementado en C, utilizando la biblioteca GMP.

Definición de interfaz: testmillerrabin.h

```

#include <stdio.h>
#include "gmp.h"

int TestMillerRabin(
    mpz_t n,           /* => Número a comprobar */
    unsigned long int t /* => Parámetro de seguridad */
);

```

Cuerpo del programa: testmillerrabin.c

```

static int MR(mpz_t n, mpz_t nm1, mpz_t a,
             mpz_t b, mpz_t q,   unsigned long int s)
{
    unsigned long int i;

    mpz_powm(b, a, q, n);

    if (mpz_cmp_ui(b, 1) == 0 || mpz_cmp(b, nm1) == 0)
        return 1;

    for (i = 1; i < s; i++)
    {
        mpz_powm_ui(b, b, 2, n);
        if (mpz_cmp(b, nm1) == 0)
            return 1;

        if (mpz_cmp_ui(b, 1) == 0)
            return 0;
    }
    return 0;
}

```

```

}

int TestMillerRabin(mpz_t n, unsigned long int t)
{
    unsigned long int s      = 0;
    int             es_primo = 1;
    gmp_randstate_t rstate;
    mpz_t           nm1;
    mpz_t           q;
    mpz_t           a;
    mpz_t           b;

    mpz_init_set(nm1, n);
    mpz_sub_ui(nm1, n, 1);
    mpz_init_set(q, nm1);
    mpz_init(a);
    mpz_init(b);

    gmp_randinit_default(rstate);

    s = mpz_scan1(nm1, 0);
    if (s > 0)
        mpz_tdiv_q_2exp(q, nm1, s);

    for (; t > 0 && es_primo == 1; t--)
    {
        do
        {
            mpz_urandomb(a, rstate, mpz_sizeinbase(n, 2) - 1);
        } while (mpz_cmp_ui(a, 1) <= 0);

        es_primo = MR(n, nm1, a, b, q, s);
    }

    gmp_randclear(rstate); /* Corresponde a un */
    mpz_clear(nm1);       /* generador tipo LC */
    mpz_clear(q);         /*  $x(i+1) = (ax(i)+c) \bmod 2^m$  */
    mpz_clear(a);
    mpz_clear(b);
    return es_primo;
}

```

I.6 Algoritmo 3.8 (MillerRabin)

Este algoritmo está implementado en C, utilizando la biblioteca GMP.

Definición de interfaz: millerrabin.h

```
# include <stdio.h>
# include "gmp.h"
# include "testmillerrabin.h"

int MillerRabin(
    mpz_t n,           /* => Número a comprobar      */
    unsigned long int t /* => Parámetro de seguridad */
);
```

Cuerpo del programa: millerrabin.c

```
int MillerRabin(mpz_t n, unsigned long int t) /* Entradas */
{
    mpz_t b;
    mpz_t nm1;
    mpz_t r;
    int i;

    static unsigned long int
    PrimosBajos[] = { 2, 3, 5, 7, 11, 13, 17,
                      19, 23, 29, 31, 37, 41, 43,
                      47, 53, 59, 61, 67, 71, 73,
                      79, 83, 89, 97, 101, 103, 107,
                      109, 113, 127, 131, 137, 139, 149,
                      151, 157, 163, 167, 173, 179, 181,
                      191, 193, 197, 199, 211, 223, 227,
                      229, 233, 239, 241, 251, 0};

    /* Por si acaso es par... */
    if (mpz_tstbit(n, 0) == 0)
        return 0;

    /* Ensayo de divisiones. Empezamos por el 3 */
    for (i = 1; PrimosBajos[i] != 0; i++)
    {
        if (mpz_cmp_ui(n, PrimosBajos[i]) == 0)
            return 1;

        if (mpz_fdiv_ui(n, PrimosBajos[i]) == 0)
            return 0;
    }

    /* Ensayo de Fermat */
    mpz_init_set_ui(b, 2*3*5*7);
    mpz_init_set(nm1, n); mpz_sub_ui(nm1, nm1, 1);
    mpz_init(r);
    mpz_powm(r, b, nm1, n);
    if (mpz_cmp_ui(r, 1) != 0)
```

```

    return 0;

mpz_clear(b);
mpz_clear(nm1);
mpz_clear(r);

/* Ensayo de Rabin-Miller propiamente dicho */
return TestMillerRabin(n, t);
}

```

I.7 Algoritmo 3.18 (NumeroAleatorioBBS)

Este algoritmo está implementado en C, utilizando la biblioteca GMP.

Definición de interfaz: bbs.h

```

#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include "gmp.h"

#define YES 1
#define NO 0

void NumeroAleatorioBBS(
    mpz_t r, /* <= Número devuelto */
    unsigned long int bits /* => Tamaño en bits */
);

```

Cuerpo del programa: bbs.c

```

static int FirstTime = YES;
static mpz_t s;
static mpz_t n;
static mpz_t p;
static mpz_t q;

static unsigned long int TakeSeed()
{
    return((unsigned long int)time((time_t *)NULL));
}

void NumeroAleatorioBBS(mpz_t r, unsigned long int bits)
{
    unsigned long int i;

```

```

if (FirstTime == YES)
{
    unsigned long int seed;

    mpz_init_set_str(p, "45011503705880468327", 10);
    mpz_init_set_str(q, "186395645564449848119", 10);
    mpz_init(n);
    mpz_init(s);
    mpz_mul(n, p, q);
    do
    {
        seed = TakeSeed();
        mpz_set_ui(s, seed);
    } while (mpz_gcd_ui(NULL, n, seed) != 1);
    mpz_clear(p);
    mpz_clear(q);
    FirstTime = NO;
}

mpz_set_ui(r, 1);

for (i = 1; i < bits; i++)
{
    mpz_powm_ui(s, s, 2, n);
    mpz_mul_2exp(r, r, 1);
    if (mpz_get_ui(s) & 1)
        mpz_add_ui(r, r, 1);
}
}

```

Observación I.1 En esta implementación, hemos utilizado dos primos prefijados, $p = 45011503705880468327$ y $q = 186395645564449848119$, que son congruentes con 3 módulo 4 tal como necesita el generador BBS. Además, sólo se toma un bit de cada iteración, si bien podría optimizarse para usar hasta $\log_2 \log_2(pq)$ bits (véase [117]).

Observación I.2 Como semilla, se utiliza la hora, medida en segundos desde el 1 de enero de 1970, que el sistema reporte en el momento de ejecución. Esta semilla no se puede considerar criptográficamente segura, no por falta de aleatoriedad, sino porque es sencillo ensayarla y es fácil de deducir. Para nuestros propósitos, es suficiente.

I.8 Algoritmo 3.21 (*NumeroAleatorioLehmer*)

Este algoritmo está implementado en C, utilizando la biblioteca GMP.

Definición de interfaz: **lehmer.h**

```
# include "gmp.h"

# define YES 1
# define NO 0

void NumeroAleatorioLehmer(
    mpz_t r,           /* <= Número aleatorio generado */
    unsigned long int bits /* => Tamaño en bits para el número */
);
```

Cuerpo del programa: lehmer.c

```
# include <time.h>
# include "lehmer.h"

static unsigned long int TakeSeed()
{
    return((unsigned long int)time((time_t *)NULL));
}

void NumeroAleatorioLehmer(mpz_t r, unsigned long int bits)
{
    static int FirstTime = YES;
    static mpz_t X;
    static mpz_t Xc;
    static mpz_t K;
    static mpz_t m;
    unsigned long int b;
    long int diff;

    if (FirstTime == YES)
    {
        /* K = 14^29 (mod 2^31 - 1) */
        mpz_init_set_ui(K, 630360016);

        /* m = 2^31 - 1 */
        mpz_init_set_ui(m, 2147483647);

        mpz_init_set_ui(X, TakeSeed());
        mpz_init(Xc);
        FirstTime = NO;
    }

    b = 0;
    mpz_set_ui(r, 0);
    while ((diff = bits - b) > 0)
    {
```

```

mpz_mul(X, K, X);
mpz_mod(X, X, m);
b += mpz_sizeinbase(X, 2);
mpz_set(Xc, X);

if (b > bits)
    mpz_tdiv_q_2exp(Xc, Xc, b - bits);

if (mpz_cmp_ui(r, 0) != 0)
    mpz_mul_2exp(Xc, Xc, mpz_sizeinbase(r, 2));
    mpz_add(r, r, Xc);
}
return;
}

```

Observación I.3 Al igual que en el caso anterior, se utiliza como semilla la hora, medida en segundos desde el 1 de enero de 1970, que el sistema reporte en el momento de ejecución. Son válidas idénticas precauciones que antes en cuanto a la seguridad criptográfica de esta semilla.

I.9 Algoritmo 3.23 (NumeroAleatorioTausworthe)

Este algoritmo está implementado en C, utilizando la biblioteca GMP.

Definición de interfaz: tausworthe.h

```

#include "gmp.h"

#define YES 1
#define NO 0

void NumeroAleatorioTausworthe(
    mpz_t r, /* <= Número aleatorio generado */
    unsigned long int bits /* => Tamaño en bits para el número */
);

```

Cuerpo del programa: tausworthe.c

```

#include <time.h>
#include "tausworthe.h"

static unsigned long int TakeSeed()
{
    return((unsigned long int)time((time_t *)NULL));
}

static unsigned long taus88(void)

```

```
{  
static unsigned long int s1;  
static unsigned long int s2;  
static unsigned long int s3;  
static unsigned long int b;  
static int FirstTime = YES;  
unsigned long int t = TakeSeed();  
  
if (FirstTime == YES)  
{  
    s1 = 2*t;  
    s2 = 8*t;  
    s3 = 16*t;  
    FirstTime = NO;  
}  
  
b = (((s1 << 13)^s1) >> 19);  
s1 = (((s1 & 4294967294U) << 12)^b);  
b = (((s2 << 2)^s2) >> 25);  
s2 = (((s2 & 4294967288U) << 4)^b);  
b = (((s3 << 3)^s3) >> 11);  
s3 = (((s3 & 4294967280U) << 17)^b);  
  
return(s1^s2^s3);  
}  
  
void NumeroAleatorioTausworthe(mpz_t r, unsigned long int bits)  
{  
static mpz_t X;  
static int FirstTime = YES;  
unsigned long int b;  
long int diff;  
  
if (FirstTime == YES)  
{  
    mpz_init(X);  
    FirstTime = NO;  
}  
  
b = 0;  
mpz_set_ui(r, 0);  
while ((diff = bits - b) > 0)  
{  
    mpz_set_ui(X, taus88());  
    b += mpz_sizeinbase(X, 2);  
  
    if (b > bits)
```

```

        mpz_tdiv_q_2exp(X, X, b - bits);

        if (mpz_cmp_ui(r, 0) != 0)
            mpz_mul_2exp(X, X, mpz_sizeinbase(r, 2));
        mpz_add(r, r, X);
    }
    return;
}

```

Observación I.4 También se utiliza aquí como semilla la hora, medida en segundos desde el 1 de enero de 1970, que el sistema reporte en el momento de ejecución.

I.10 Algoritmo 3.27 (GeneraPrimoAleatorio)

Este algoritmo está implementado en C, utilizando la biblioteca GMP.

Definición de interfaz: `primoaleatorio.h`

```

#include "gmp.h"
#include "bbs.h"
#include "millerrabin.h"

void GeneraPrimoAleatorio(
    mpz_t p, /* <= Primo generado */
    unsigned long int bits, /* => Tamaño en bits del primo */
    unsigned long int z /* => Parámetro de seguridad */
);

```

Cuerpo del programa: `primoaleatorio.c`

```

void GeneraPrimoAleatorio(mpz_t p, unsigned long int bits,
                           unsigned long int z)
{
    for (NumeroAleatorioBBS(p, bits);
         ;
         NumeroAleatorioBBS(p, bits))
    {
        if (MillerRabin(p, z))
            break;
    }
}

```

I.11 Algoritmo 4.41 (Primo1Seguro)

Este algoritmo está implementado en C, utilizando la biblioteca GMP.

Definición de interfaz: primo1seguro.h

```
# include "gmp.h"
# include "bbs.h"
# include "millerrabin.h"

void Primo1Seguro(
    mpz_t r,           /* <= Primo 1-seguro devuelto */
    unsigned long int bits, /* => Tamaño en bits deseado */
    unsigned long int z      /* => Parámetro de seguridad */
);
```

Cuerpo del programa: primo1seguro.c

```
void Primo1Seguro(mpz_t p, unsigned long bits, unsigned long z)
{
    mpz_t r;

    mpz_init(r);

    for (NumeroAleatorioBBS(p, bits);
         ;
         NumeroAleatorioBBS(p, bits))
    {
        if (MillerRabin(p, z))
        {
            mpz_sub_ui(r, p, 1);
            mpz_tdiv_q_ui(r, r, 2);
            if (MillerRabin(r, z))
                break;
        }
    }

    mpz_clear(r);
}
```

I.12 Algoritmo 4.42 (Primo2Seguro)

Este algoritmo está implementado en C, utilizando la biblioteca GMP.

Definición de interfaz: primo2seguro.h

```
# include "gmp.h"
# include "bbs.h"
# include "millerrabin.h"

void Primo2Seguro(
```

```
mpz_t r,           /* <= Primo 2-seguro devuelto */
unsigned long int bits, /* => Tamaño en bits deseado */
unsigned long int z   /* => Parámetro de seguridad */
);
```

Cuerpo del programa: primo2seguro.c

```
void Primo2Seguro(mpz_t p, unsigned long bits, unsigned long z)
{
mpz_t r;
mpz_t s;

mpz_init(r);
mpz_init(s);

for (NumeroAleatorioBBS(p, bits);
     ;
     NumeroAleatorioBBS(p, bits))
{
    if (MillerRabin(p, z))
    {
        mpz_sub_ui(r, p, 1);
        mpz_tdiv_q_ui(r, r, 2);
        if (MillerRabin(r, z))
        {
            mpz_sub_ui(s, r, 1);
            mpz_tdiv_q_ui(s, s, 2);
            if (MillerRabin(s, z))
                break;
        }
    }
}

mpz_clear(r);
mpz_clear(s);
}
```

I.13 Algoritmo 5.24 (GordonStrong)

Presentamos la implementación del algoritmo 5.24, escrita en el lenguaje propio de MAPLE, para calcular primos robustos.

El programa tiene varias partes. Definimos, en principio, un generador aleatorio de números comprendidos entre 0 y 10000. Para ello utilizamos estas dos sentencias:

```
> randomize();
> pa:=rand(10000):
```

A continuación comenzamos con el lazo, que repetimos 200 veces según indexa la variable n lo que nos permitirá obtener esa misma cantidad de primos robustos. Comenzamos generando números aleatorios que almacenamos en las variables a y b, mediante las cuales obtenemos los primos s y t. Abrimos otro lazo para encontrar el primer primo igual a $2Lt + 1$ que almacenamos en la variable r.

```
> for n from 1 to 200
> do
>   a:=pa():
>   b:=pa():
>   s:=nextprime(a):
>   t:=nextprime(b):
>   for L from 1 while not isprime(2*L*t+1)
>   do
>     end do:
>   r := 2*L*t+1:
```

En la siguiente parte computamos el valor $u(r, s) = (s^{r-1} - r^{s-1}) \pmod{rs}$.

```
> u := (s&^(r-1) - r&^(s-1)) mod r*s;
```

Siguiendo el Teorema 5.22, calculamos p_0 cuyo valor depende de si u es par o impar.

```
> if u mod 2 = 0 then
>   p0 := u + r*s:
> else
>   p0 := u:
> end if:
```

Obtenido p_0 , exploramos el espacio $p_0 + 2krs$ haciendo variar dentro de un lazo la variable k y utilizando los valores de p_0 , r y s obtenidos en los pasos anteriores.

```
> for k from 1 while not isprime(p0 + 2*k*r*s)
> do
>   end do:
>   p := p0 + 2*k*r*s:
>   print(p):
> end do:
```

El contenido de la variable p será un primo robusto de acuerdo a los resultados del método de Gordon. Cerramos el lazo que nos permitirá repetir el experimento tantas veces cuantas lo permita el rango de la variable n (en este ejemplo, 200).

I.14 Algoritmo 5.26 (GordonStrong2)

Presentamos la implementación del algoritmo 5.26, escrita en el lenguaje propio de MAPLE, para calcular primos robustos por el método alternativo propuesto en [73].

Definimos, en principio, un generador aleatorio de números comprendidos entre 2^n y 2^{n+1} . Fijemos para nuestro ejemplo, $n = 10$. Utilizamos estas dos sentencias:

```
> randomize();
> n:=10;
> pa:=rand(2^n..2^(n+1));
```

Generamos aleatoriamente dos primos, s y t , de tamaño similar. Para nuestro ejemplo, estos primos van a ser del orden de 2^{10} .

```
> s:=pa();
> while not isprime(s) do
>   s:=pa();
> end do;
> t:=pa();
> while not isprime(t) do
>   t:=pa();
> end do;
```

Exploramos los valores de la sucesión $2it + 1$ con $i = 1, 2, 3, \dots$ hasta encontrar uno que sea primo. Llaremos $r = 2it + 1$ a ese primo.

```
> for i from 1 while not isprime(2*i*t+1)
> do
> end do;
> r:=2*i*t+1;
```

A continuación se computa el valor $p_0 = (2s^{r-2} \pmod{r})s - 1$.

```
> p0:=(2*s^(r-2) mod r)*s-1;
```

Exploramos los valores de la sucesión $p_0 + 2jrs$ con $j = 0, 1, 2, \dots$ hasta encontrar uno que sea primo. Llaremos p a ese primo. Se devuelve p como resultado.

```
> for j from 0 while not isprime(p0+2*j*r*s)
> do
> end do;
> p:=p0+2*j*r*s;
```

El contenido de la variable p es el resultado pedido.

I.15 Algoritmo 5.30 (OgiwaraStrong)

Este algoritmo está implementado en C, utilizando la biblioteca GMP.

Definición de interfaz: `ogiwara.h`

```
# include <stdio.h>
# include "gmp.h"
# include "euclidesext.h"
# include "bbs.h"

void OgiwaraStrong(
    mpz_t p,           /* <= Primo robusto generado */
    unsigned long int bits, /* => Número de bits deseados */
    unsigned long int z      /* => Parámetro de seguridad */
);
```

Cuerpo del programa: ogiwara.c

```
static void GetLevel2Prime(mpz_t p, unsigned long int bits,
                           unsigned long int z)
{
    mpz_t p1;
    mpz_t p2;
    mpz_t a;
    mpz_t b;
    mpz_t uno;
    mpz_t M;
    unsigned long int k = 0;

    mpz_init(p1);
    mpz_init(p2);
    mpz_init(a);
    mpz_init(b);
    mpz_init(uno);
    mpz_init(M);

    do
    {
        GeneraPrimoAleatorio(p1, bits, z);
        GeneraPrimoAleatorio(p2, bits, z);

    } while (mpz_cmp(p1, p2) == 0); /* Los dos primos */
                                    /* son iguales */

    EuclidesExt(a, b, uno, p1, p2);

    mpz_mul_ui(M, p1, 2);
    mpz_mul(M, M, p2);

    if (mpz_sgn(a) == 1)
    {
```

```
    mpz_mul_ui(p, a, 2);
    mpz_mul(p, p, p1);
    mpz_sub_ui(p, p, 1);
}
else if (mpz_sgn(b) == 1)
{
    mpz_mul_ui(p, b, 2);
    mpz_mul(p, p, p2);
    mpz_sub_ui(p, p, 1);
}
else
{
    fprintf(stderr, "ERROR: Ni A ni B son positivas.\n");
    exit(1);
}

do
{
    k++;
    mpz_add(p, p, M);
} while (MillerRabin(p, z) == 0); /* Lazo mientras */
                                /* no sea primo */

    mpz_clear(p1);
    mpz_clear(p2);
    mpz_clear(a);
    mpz_clear(b);
    mpz_clear(uno);
    mpz_clear(M);
}

static void GetLevel1Prime(mpz_t p, unsigned long int bits,
                           unsigned long int z)
{
    mpz_t p1;
    mpz_t p2;
    mpz_t a;
    mpz_t b;
    mpz_t uno;
    mpz_t M;
    unsigned long int k = 0;

    mpz_init(p1);
    mpz_init(p2);
    mpz_init(a);
    mpz_init(b);
    mpz_init(uno);
```

```

mpz_init(M);

do
{
    GetLevel2Prime(p1, bits, z);
    GetLevel2Prime(p2, bits, z);

} while (mpz_cmp(p1, p2) == 0); /* Los dos primos */
                                /* son iguales */

EuclidesExt(a, b, uno, p1, p2);

mpz_mul_ui(M, p1, 2);
mpz_mul(M, M, p2);

if (mpz_sgn(a) == 1
{
    mpz_mul_ui(p, i, 2);
    mpz_mul(p, p, p1);
    mpz_sub_ui(p, p, i);
}
else if (mpz_sgn(b) == 1)
{
    mpz_mul_ui(p, b, 2);
    mpz_mul(p, p, p2);
    mpz_sub_ui(p, p, i);
}
else
{
    fprintf(stderr, "ERROR: Ni A ni B son positivas.\n");
    exit(1);
}

do
{
    k++;
    mpz_add(p, p, M);
} while (MillerRabin(o, z) == 0); /* Lazo mientras */
                                /* no sea primo */

mpz_clear(p1);
mpz_clear(p2);
mpz_clear(a);
mpz_clear(b);
mpz_clear(uno);
mpz_clear(M);
}

```

```

void OgiwaraStrong(mpz_t p,
                    unsigned long int bits,
                    unsigned long int z)
{
    GetLevel1Prime(p, bits/4, z);
}

```

I.16 Algoritmo 5.33 (StrongOptimo)

Este algoritmo está implementado en C, utilizando la biblioteca GMP.

Definición de interfaz: strong.h

```

#include <stdlib.h>
#include <stdio.h>
#include "gmp.h"
#include "millerrabin.h"
#include "bbs.h"

void StrongOptimo(
    mpz_t p,          /* <= Primo robusto generado */
    unsigned long bits, /* => Bits deseados para el primo */
    unsigned long z    /* => Parámetro de seguridad */
);

```

Cuerpo del programa: strong.c

```

void StrongOptimo(mpz_t p, unsigned long bits, unsigned long z)
{
    mpz_t r;
    mpz_t s;
    mpz_t t;

    mpz_init(r);
    mpz_init(s);
    mpz_init(t);

    for (NumeroAleatorioBBS(t, bits);
         ;
         NumeroAleatorioBBS(t, bits))
    {
        if (MillerRabin(t, z))
        {
            mpz_mul_ui(r, t, 2);
            mpz_add_ui(r, r, 1);
            if (MillerRabin(r, z))

```

```

{
    mpz_mul_ui(s, r, 3);
    mpz_add_ui(s, s, 1);
    mpz_tdiv_q_ui(s, s, 2);
    if (MillerRabin(s, z))
    {
        mpz_mul_ui(p, r, 6);
        mpz_add_ui(p, p, 1);
        if (MillerRabin(p, z))
            break;
    }
}
}

mpz_clear(r);
mpz_clear(s);
mpz_clear(t);
}

```

I.17 Cálculo de la constante C_2 (véase 4.3.1)

El siguiente programa está escrito en el lenguaje propio de MAPLE. Nuestro objetivo es calcular aproximadamente el valor de la constante C_2 referenciada en la sección 4.3.1. Para ello, se tiene la siguiente sucesión

$$C_2(n) = \prod_{q=p_3}^{p_n} \frac{q^2(q-3)}{(q-1)^3},$$

tal que

$$\lim_{n \rightarrow \infty} C_2(n) = C_2.$$

La variable c acumulará el valor sucesivo del productorio, es decir, la n -ésima vuelta del lazo efectúa

$$c \leftarrow c \cdot \frac{p_n^2(p_n - 3)}{(p_n - 1)^3}.$$

La variable p almacena el valor del primo indexado por la variable n . Obsérvese que, obviamente, $p_n > 3$ para que el productorio no se anule; por tanto, hemos de comenzar el programa en el primer primo que cumpla esa condición, es decir, $p_3 = 5$. Esto explica los valores iniciales de las variables p y n . Generamos un dato de salida cada vez que la variable n es un múltiplo de 1000, con lo que este programa nos proporciona una tabla de valores $C_2(1000), C_2(2000), C_2(3000), \dots, C_2(100000)$. El programa queda, pues, como sigue:

```

> c:=1:
> p:=5:
> for n from 3 to 100000 do

```

```
> c:=evalf(c*p^2*(p-3)/(p-1)^3):
> p:=nextprime(p):
> if n mod 1000 = 0 then
>   print(c):
> end if:
> end do:
```

I.18 Cálculo de la constante C_σ (véase 5.2.3)

El siguiente programa está escrito en el lenguaje propio de MAPLE. Nuestro objetivo es calcular aproximadamente el valor de la constante C_σ referenciada en la sección 5.2.3. Para ello, se tiene la siguiente sucesión

$$C_\sigma(n) = \prod_{q=p_5}^{p_n} \frac{q^3(q-4)}{(q-1)^4},$$

tal que

$$\lim_{n \rightarrow \infty} C_\sigma(n) = C_\sigma.$$

La variable c acumulará el valor sucesivo del productorio, es decir, la n -ésima vuelta del lazo efectúa

$$c \leftarrow c \cdot \frac{p_n^3(p_n-4)}{(p_n-1)^4}.$$

La variable p almacena el valor del primo indexado por la variable n . El primo por el que se ha de comenzar es p_5 , como quedó explicado en la sección referenciada, lo cual explica los valores iniciales para las variables p y n . Generamos un dato de salida cada vez que la variable n es un múltiplo de 1000, con lo que este programa nos proporciona una tabla de valores $C_\sigma(1000), C_\sigma(2000), C_\sigma(3000), \dots, C_\sigma(100000)$. El programa queda, pues, como sigue:

```
> c:=1:
> p:=11:
> for n from 5 to 100000 do
>   c:=evalf(c*p^3*(p-4)/(p-1)^4):
>   p:=nextprime(p):
>   if n mod 1000 = 0 then
>     print(c):
>   end if:
> end do:
```


Referencias

- [1] L.M. Adleman, J. Demarrais, *A Subexponential Algorithm for Discrete Logarithms over All Finite Fields*, Mathematics of Computation **61** (1993), 1–15.
- [2] W. Alexi, B. Chor, O. Goldreich, C.P. Schnorr, *RSA and Rabin functions: certain parts are as hard as the whole*, SIAM Journal on Computing **17** (1988), 194–209.
- [3] T.M. Apostol, *Introduction to Analytic Number Theory*, Springer-Verlag, New York, 1976, 1997, 4th corrected edition.
- [4] A.O.L. Atkin, F. Morain, *Elliptic Curves and Primality Proving*, Mathematics of Computation **61** (1993), 29–68.
- [5] E. Bach, J. Shallit, *Algorithmic Number Theory, Vol. I: Efficient Algorithms*, The MIT Press, Cambridge, MA, 1996.
- [6] R. Balasubramanian, *The Circle Method and its Implications*, Journal of the Indian Institute of Science Special Issue (1987), 39–44.
- [7] P.T. Bateman, R.A. Horn, *A Heuristic Asymptotic Formula Concerning the Distribution of Prime Numbers*, Mathematics of Computation **16** (1962), 363–367.
- [8] P.T. Bateman, R.A. Horn, *Primes Represented by Irreducible Polynomials in One Variable*, en *Proceedings of Symposia in Pure Mathematics VIII*, pp. 119–132, Providence, RI, 1965, American Mathematical Society.
- [9] P.T. Bateman, R.M. Stemmler, *Waring’s Problem for Algebraic Number Fields and Primes of the Form $(p^r - 1)/(p^d - 1)$* , Illinois Journal of Mathematics **6** (1962), 142–156.
- [10] M. Bellare, P. Rogaway, *Optimal asymmetric encryption*, en *Advances in Cryptology - Proceedings of EUROCRYPT ’94*, volumen 950 de *Lecture Notes in Computer Science*, pp. 92–111, Berlin, 1994, Springer-Verlag.
- [11] D.J. Bernstein, A.K. Lenstra, *A general number field sieve implementation*, en A.K. Lenstra, H.W. Lenstra, Jr., editores, *The Development of the Number Field Sieve*, volumen 1554 de *Lecture Notes in Mathematics*, pp. 103–126, Springer-Verlag, 1993.

- [12] G.R. Blakley, I. Borosh, *Rivest-Shamir-Adleman Public Key Cryptosystems Do Not Always Conceal Messages*, Computers & Mathematics with Applications **5** (1979), 169–178.
- [13] L. Blum, M. Blum, M. Shub, *A Simple Unpredictable Pseudo-Random Number Generator*, SIAM Journal on Computing **15** (1986), 364–383.
- [14] M. Blum, S. Goldwasser, *An Efficient Probabilistic Public Key Encryption Scheme Which Hides All Partial Information*, en *Advances in Cryptology - Proceedings of CRYPTO '84*, volumen 196 de *Lecture Notes in Computer Science*, pp. 289–299, Berlin, 1985, Springer-Verlag.
- [15] B. den Boer, *Diffie-Hellman is as Strong as Discrete Log for Certain Primes*, en S. Goldwasser, editor, *Advances in Cryptology - CRYPTO '88*, volumen 403 de *Lecture Notes in Computer Science*, pp. 530–539, Springer-Verlag, 1988.
- [16] D. Boneh, G. Durfee, *Cryptanalysis of RSA with Private Key d less than N^{0,292}*, en Jacques Stern, editor, *Advances in Cryptology - EUROCRYPT '99*, volumen 1592 de *Lecture Notes in Computer Science*, pp. 1–11, Jacques Stern, 1999.
- [17] D. Boneh, R. Venkatesan, *Breaking RSA May Not Be Equivalent to Factoring*, en *Proceedings of EUROCRYPT '98*, volumen 1233 de *Lecture Notes in Computer Science*, pp. 59–71, Berlin, 1998, Springer-Verlag.
- [18] J. Brillhart, D.H. Lehmer, J.L. Selfridge, *New Primality Criteria and Factorizations of 2^m + 1*, Mathematics of Computation **29** (1975), 620–647.
- [19] J.A. Buchmann, *Introduction to Cryptography*, Springer-Verlag, New York, 2001.
- [20] J.P. Buhler, H.W. Lenstra, Jr., C. Pomerance, *Factoring integers with the number field sieve*, en A.K. Lenstra, H.W. Lenstra, Jr., editores, *The Development of the Number Field Sieve*, volumen 1554 de *Lecture Notes in Mathematics*, pp. 50–94, Springer-Verlag, 1993.
- [21] Y. Cai, *On the Distribution of Safe-Primes*, Journal of Shandong University **29** (1994), 388–392.
- [22] S. Cavallar, B. Dodson, A.K. Lenstra, W. Lioen, P.L. Montgomery, B. Murphy, H. te Riele, K. Aardal, J. Gilchrist, G. Guillerm, P. Leyland, J. Marchand, F. Morain, A. Muffett, C. Putnam, P. Zimmermann, *Factorization of a 512-Bit Modulus*, en Bart Preneel, editor, *Advances in Cryptology - EUROCRYPT '00*, volumen 1807 de *Lecture Notes in Computer Science*, pp. 1–18, Berlin, 2000, Springer-Verlag.
- [23] B.A. Cipra, *Safe Against Cycling: Researchers Confirm Invulnerability of RSA Cryptosystem*, SIAM News **34** (2001), 1.
- [24] H. Cohen, *High Precision Computation of Hardy-Littlewood Constants*, <http://www.math.u-bordeaux.fr/~cohen/hardylw.dvi>.

- [25] H. Cohen, *A Course in Computational Algebraic Number Theory*, Springer-Verlag, Berlin, 1993.
- [26] D. Coppersmith, *Fast Evaluation of Logarithms in Fields of Characteristic Two*, IEEE Transactions on Information Theory **30** (1984), 587–594.
- [27] R. Crandall, C. Pomerance, *Prime Numbers. A Computational Perspective*, Springer-Verlag, New York, 2001.
- [28] I.B. Damgård, *A Design Principle for Hash Functions*, en G. Brassard, editor, *Advances in Cryptology - CRYPTO '89*, volumen 435 de *Lecture Notes in Computer Science*, pp. 416–427, Springer-Verlag, 1990.
- [29] L.E. Dickson, *A New Extension of Dirichlet's Theorem on Primes Numbers*, Messenger of Mathematics **33** (1904), 155–161.
- [30] W. Diffie, M.E. Hellman, *New Directions in Cryptography*, IEEE Transactions on Information Theory **22** (1976), 644–654.
- [31] H. Dubner, *Large Sophie Germain Primes*, Mathematics of Computation **65** (1996), 393–396.
- [32] R. Durán Díaz, L. Hernández Encinas, J. Muñoz Masqué, *Ataques a DES y módulos factorizados de RSA*, Ágora SIC **20** (2000), 1–4.
- [33] R. Durán Díaz, J. Muñoz Masqué, *How strong are strong primes? A proposal of test*, en *Proceedings of the 5th World Multiconference Cybernetics and Informatics, Orlando, Florida, 22–25 de julio de 2001*, volumen VII, pp. 496–499, 2001.
- [34] R. Durán Díaz, J. Muñoz Masqué, *Un criterio de optimalidad para primos strong*, en *Actas del XVI Simposium Nacional de la Unión Científica Internacional de Radio (URSI), Madrid, 19–21 de septiembre de 2001*, pp. 227–228, 2001.
- [35] R. Durán Díaz, F. Montoya Vitini, J. Muñoz Masqué, *Densidad de primos seguros*, en *Actas del XIV Simposium Nacional de la Unión Científica Internacional de Radio (URSI), Santiago de Compostela, 8–10 de septiembre de 1999*, pp. 86–87, 1999.
- [36] R. Durán Díaz, F. Montoya Vitini, J. Muñoz Masqué, *Safe primes density and cryptographic applications*, en *IEEE Proceedings of the 33rd Annual International Carnahan Conference on Security Technology, Madrid, 5–7 de octubre de 1999*, pp. 363–367, 1999.
- [37] G. Durfee, *Cryptanalysis of RSA Using Algebraic and Lattice Methods*, tesis doctoral, Department of Computer Science, Stanford University, 2002.
- [38] H. Eberle, *A High-speed DES Implementation for Network Applications*, en E.F. Brickell, editor, *Advances in Cryptology - CRYPTO '92*, volumen 740 de *Lecture Notes in Computer Science*, pp. 521–539, Springer-Verlag, 1993.

- [39] T. ElGamal, *A Public Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms*, IEEE Transactions on Information Theory **31** (1985), 469–472.
- [40] T. ElGamal, *A Subexponential-Time Algorithm for Computing Discrete Logarithms over $GF(p^2)$* , IEEE Transactions on Information Theory **31** (1985), 473–481.
- [41] T. Forbes, *Prime Clusters and Cunningham Chains*, Mathematics of Computation **68** (1999), 1739–1747.
- [42] J.B. Friedlander, C. Pomerance, I.E. Shparlinski, *Period of the Power Generator and Small Values of Camichael's Function*, Mathematics of Computation **70** (2001), 1591–1605.
- [43] J.B. Friedlander, I.E. Shparlinski, *On the Distribution of the Power Generator*, Mathematics of Computation **70** (2001), 1575–1589.
- [44] S. Goldwasser, S. Micali, *Probabilistic Encryption*, Journal of Computer and System Sciences **28** (1984), 270–299.
- [45] S.W. Golomb, *Shift Register Sequences*, AEGEAN PARK PRESS, Laguna Hills, CA, 1982.
- [46] J. Gordon, *Strong Primes are Easy to Find*, en G. Goos, J. Hartmanis, editores, *Advances in Cryptology - Proceedings of EUROCRYPT '84*, volumen 209 de *Lecture Notes in Computer Science*, pp. 216–223, Berlin, 1984, Springer-Verlag.
- [47] T. Granlund, *The GNU Multiprecision Arithmetic Library*, <http://www.swox.com/gmp>.
- [48] R.K. Guy, *Unsolved Problems in Number Theory*, Springer-Verlag, Berlin, 1994.
- [49] M. Gysin, J. Seberry, *Generalised Cycling Attacks on RSA and Strong RSA Primes*, en J. Pieprzyk, editor, *Information security and privacy. 4th Australian conference, ACISP '99, Wollongong, NSW, Australia, April 7–9, 1999*, volumen 1587 de *Lecture Notes in Computer Science*, pp. 149–163, Berlin, 1999, Springer-Verlag.
- [50] G.H. Hardy, E. Littlewood, *Some Problems of 'Partitio Numerorum'; III: On the Expression of a Number as a Sum of Primes*, Acta Mathematica **44** (1922), 1–70.
- [51] J. Hastad, *On Using RSA with Low Exponent in a Public Key Network*, en H.C. Williams, editor, *Advances in Cryptology - CRYPTO '85*, volumen 218 de *Lecture Notes in Computer Science*, pp. 403–408, Berlin, 1986, Springer-Verlag.
- [52] J. Hastad, *Solving Simultaneous Modular Equations of Low Degree*, SIAM Journal on Computing **17** (1988), 336–341.

- [53] L. Hernández Encinas, F. Montoya Vitini, J. Muñoz Masqué, A. Peinado Domínguez, *Maximal Period of Orbits of the BBS Generator*, en *Proceedings of CISC'98*, pp. 70–80, Seoul, 1998, Korea Institute of Information Security & Criptology.
- [54] L. Hernández Encinas, J. Muñoz Masqué, F. Montoya Vitini, G. Álvarez Mañañón, A. Peinado Domínguez, *Algoritmo de Cifrado con Clave Pública mediante una Función Cuadrática en el Grupo de los Enteros módulo n*, en J. Tena Ayuso, M.F. Blanco Martín, editores, *Actas de la IV Reunión Española sobre Criptología, Valladolid, 9–11 de septiembre de 1996*, pp. 101–108, Valladolid, 1996, Universidad de Valladolid.
- [55] D. Kahn, *The Code Breakers, the Story of Secret Writing*, MacMillan Publishing Co., New York, 1967.
- [56] D.E. Knuth, *The Art of Computer Programming*, volumen 2 - Seminumerical Algorithms, Addison-Wesley Publishing Co., Reading, MA, 1968, 1980, 2nd edition.
- [57] K. Kurosawa, T. Ito, M. Takeuchi, *Public Key Cryptosystem Using a Reciprocal Number with the Same Intractability as Factoring a Large Number*, *Cryptologia* **12** (1988), 225–233.
- [58] C.S. Laih, W.C. Yang, C.H. Chen, *Efficient Method for Generating Strong Primes with Constraint of Bit Length*, *Electronics Letters* **27** (1991), 1807–1808.
- [59] P. L'Ecuyer, *Maximally Equidistributed Combined Tausworthe Generators*, *Mathematics of Computation* **65** (1996), 203–213.
- [60] D.H. Lehmer, *An Extended Theory of Lucas' Functions*, *Annals of Mathematics* **31** (1930), 419–448.
- [61] A.K. Lenstra, H.W. Lenstra, Jr., M.S. Manasse, J.M. Pollard, *The Number Field Sieve*, en A.K. Lenstra, H.W. Lenstra, Jr., editores, *The Development of the Number Field Sieve*, volumen 1554 de *Lecture Notes in Mathematics*, pp. 11–42, Springer-Verlag, 1993.
- [62] A.K. Lenstra, E.R. Verheul, *Selecting Cryptographic Key Sizes*, *Journal of Cryptology* **14** (2001), 255–293.
- [63] H.W. Lenstra, Jr., *Factoring Integers with Elliptic Curves*, *Annals of Mathematics* **126** (1987), 649–673.
- [64] J.H. Loxton, D.S.P. Khoo, G.J. Bird, J. Seberry, *A Cubic RSA Code Equivalent to Factorization*, *Journal of Cryptology* **5** (1992), 139–150.
- [65] J.L. Massey, *Shift-register synthesis and BCH decoding*, *IEEE Transactions on Information Theory* **15** (1969), 122–127.

- [66] U.M. Maurer, *Some Number-Theoretic Conjectures and their Relation to the Generation of Cryptographic Primes*, en Chris Mitchell, editor, *Cryptography and Coding II*, pp. 173–191, Oxford, 1992, Clarendon Press.
- [67] U.M. Maurer, *Towards the Equivalence of Breaking the Diffie-Hellman Protocol and Computing Discrete Logarithms*, en Y. Desmedt, editor, *Advances in Cryptology - CRYPTO '94*, volumen 839 de *Lecture Notes in Computer Science*, pp. 271–281, Springer-Verlag, 1994.
- [68] U.M. Maurer, *Fast Generation of Prime Numbers and Secure Public-Key Cryptographic Parameters*, *Journal of Cryptology* **8** (1995), 123–155.
- [69] U.M. Maurer, S. Wolf, *The Relationship Between the Diffie-Hellman Protocol and Computing Discrete Logarithms*, *SIAM Journal on Computing* **28** (1999), 1689–1721.
- [70] U.M. Maurer, S. Wolf, *The Diffie-Hellman Protocol*, *Designs, Codes and Cryptography* **19** (2000), 147–171.
- [71] A.J. Menezes, T. Okamoto, S.A. Vanstone, *Reducing Elliptic Curve Logarithms to Logarithms in Finite Field*, *IEEE Transactions on Information Theory* **39** (1993), 1639–1646.
- [72] A.J. Menezes, P.C. van Oorschot, S.A. Vanstone, *Handbook of Applied Cryptography*, CRC Press, Inc., Boca Raton, FL, 1997.
- [73] R.C. Merkle, *A Fast Software One-Way Hash Function*, *Journal of Cryptology* **3** (1990), 43–58.
- [74] S. Micali, C.P. Schnorr, *Efficient, Perfect Polynomial Random Number Generators*, *Journal of Cryptology* **3** (1991), 157–172.
- [75] P. Mihailescu, *Fast Generation of Provable Primes Using Search in Arithmetic Progressions*, en *Advances in Cryptology - CRYPTO '94*, volumen 839 de *Lecture Notes in Computer Science*, pp. 282–293, Berlin, 1994, Springer-Verlag.
- [76] G. Miller, *Riemann's Hypothesis and Tests for Primality*, *Journal of Computation and System Sciences* **13** (1976), 300–317.
- [77] R.A. Mollin, *RSA and PUBLIC-KEY CRYPTOGRAPHY*, CHAPMAN & HALL/CRC, Boca Raton, FL, 2003.
- [78] F. Morain, *Courbes Elliptiques et Tests de Primalité*, tesis doctoral, Université Claude Bernard - Lyon I, 1990.
- [79] W. Narkiewicz, *Polynomial Mappings*, Springer-Verlag, Berlin, 1995.
- [80] NIST, *The Digital Signature Standard, Proposed by NIST*, *Communications of the ACM* **35** (1992), 36–54, National Institute for Standards and Technology.

- [81] A.M. Odlyzko, *Discrete Logarithms in Finite Fields and their Cryptographic Significance*, en *Advances in Cryptology - Proceedings of EUROCRYPT '84*, volumen 209 de *Lecture Notes in Computer Science*, pp. 224–314, Berlin, 1985, Springer-Verlag.
- [82] M. Ogiwara, *A Method for Generating Cryptographically Strong Primes*, IEICE Transactions on Communications, Electronics, Information, and Systems **E73** (1990), 985–994.
- [83] P. Paillier, *Public-Key Cryptosystems Based on Composite Degree Residuosity Classes*, en Jacques Stern, editor, *Advances in Cryptology - EUROCRYPT '99*, volumen 1592 de *Lecture Notes in Computer Science*, pp. 223–238, Berlin, 1999, Springer-Verlag.
- [84] C. Pan, *A New Attempt on Goldbach Conjecture*, Chinese Annals of Mathematics **3** (1982), 555–560.
- [85] W.H. Payne, J.R. Rabung, T.P. Bogyo, *Coding the Lehmer Pseudo-Random Number Generator*, Communications of the ACM **12** (1969), 85–86.
- [86] A. Peinado Domínguez, *Órbitas de las funciones cuadráticas sobre cuerpos finitos. Aplicaciones a la generación de números pseudoaleatorios y al diseño de criptosistemas*, tesis doctoral, Universidad Politécnica de Madrid, Facultad de Informática, Departamento de Lenguajes y Sistemas Informáticos e Ingeniería del Software, 1997.
- [87] H.C. Pocklington, *The Determination of the Prime or Composite Nature of Large Numbers by Fermat's Theorem*, Proceedings of Cambridge Philosophical Society **18** (1914), 29–30.
- [88] S.C. Pohlig, M.E. Hellman, *An Improved Algorithm for Computing Logarithms over $GF(p)$ and its Cryptographic Significance*, IEEE Transactions on Information Theory **24** (1978), 106–110.
- [89] J.M. Pollard, *Theorems on Factorization and Primality Testing*, Mathematical Proceedings of the Cambridge Philosophical Society **76** (1974), 521–528.
- [90] J.J. Quisquater, C. Couvreur, *Fast Decipherment Algorithm for RSA Public-Key Cryptosystem*, Electronics Letters **18** (1982), 905–907.
- [91] M. Rabin, *Probabilistic Algorithms for Testing Primality*, Journal of Number Theory **12** (1980), 128–138.
- [92] M.O. Rabin, *Digital Signatures and Public Key Functions as Intractable as Factorization*, Informe Técnico TM-212, Lab. for Computer Science, Massachusetts Inst. of Technology, 1979.
- [93] Red Hat, Inc., <http://www.cygwin.com>.
- [94] P. Ribenboim, *The little book of big primes*, Springer-Verlag, New York, 1991.

- [95] H. Riesel, *Prime Numbers and Computer Methods of Factorization*, Birkhäuser, Boston, 1994.
- [96] T. Ritter, *The Efficient Generation of Cryptographic Confusion Sequences*, Cryptologia **15** (1991), 81–139.
- [97] R.L. Rivest, *The MD₄ Message Digest Algorithm*, en A.J Menezes, S.A. Vanstone, editores, *Advances in Cryptology - CRYPTO '90*, volumen 537 de *Lecture Notes in Computer Science*, pp. 303–311, Springer-Verlag, 1991.
- [98] R.L. Rivest, A. Shamir, L. Adleman, *A Method for Obtaining Digital Signatures and Public-Key Cryptosystems*, Communications of the ACM **21** (1978), 120–126.
- [99] R.L. Rivest, R.D. Silverman, *Are ‘Strong Primes’ Needed for RSA?*, <http://eprint.iacr.org/>.
- [100] L. Ronyai, *Factoring Polynomials Modulo Special Primes*, Combinatorica **9** (1989), 199–206.
- [101] A. Salomaa, *Public Key Cryptography*, Springer-Verlag, Berlin, 1990.
- [102] A. Schinzel, *A Remark on a Paper of Bateman and Horn*, Mathematics of Computation **17** (1963), 445–447.
- [103] C.P. Schnorr, *Efficient identification and signatures for smart cards*, en G. Brassard, editor, *Proceedings of CRYPTO '89*, volumen 435 de *Lecture Notes in Computer Science*, pp. 239–252, Berlin, 1990, Springer-Verlag.
- [104] M.R. Schroeder, *Number Theory in Science and Communication*, Springer-Verlag, Berlin, 1984, 1997, 3d edition.
- [105] J. Shawe-Taylor, *Generating Strong Primes*, Electronics Letters **22** (1986), 875–877.
- [106] J. Shawe-Taylor, *Proportion of Primes Generated by Strong Primes Methods*, Electronics Letters **28** (1992), 135–137.
- [107] G.J. Simmons, M.J. Norris, *Preliminary Comments on the M.I.T. Public-Key Cryptosystem*, Cryptologia **1** (1977), 406–414.
- [108] M.E. Smid, D.K. Branstad, *Response to comments on the NIST proposed digital signature standard*, en E.F. Brickell, editor, *Advances in Cryptology - CRYPTO '92*, volumen 740 de *Lecture Notes in Computer Science*, pp. 76–88, Springer-Verlag, 1993.
- [109] R. Solovay, V. Strassen, *A fast Monte-Carlo test for primality*, SIAM Journal on Computing **6** (1977), 84–85.
- [110] R. Solovay, V. Strassen, *Erratum for “A fast Monte-Carlo test for primality”*, SIAM Journal on Computing **7** (1978), 118.

- [111] D.R. Stinson, *Cryptography. Theory and Practice*, CRC Press, Boca Raton, FL, 1995.
- [112] T. Takagi, *Fast RSA-Type Cryptosystems Using N-Adic Expansion*, en B. Kaliski, editor, *Advances in Cryptology - CRYPTO '97*, volumen 1294 de *Lecture Notes in Computer Science*, pp. 372–384, Berlin, 1997, Springer-Verlag.
- [113] T. Takagi, *Fast RSA-Type Cryptosystem Modulo $p^k q$* , en H. Krawczyk, editor, *Advances in Cryptology - CRYPTO '98*, volumen 1462 de *Lecture Notes in Computer Science*, pp. 318–326, Berlin, 1998, Springer-Verlag.
- [114] R.C. Tausworthe, *Random Numbers Generated by Linear Recurrence Modulo Two*, Mathematics of Computation **19** (1965), 201–209.
- [115] G. Tenenbaum, M. Mendès-France, *The Prime Numbers and Their Distribution*, American Mathematical Society, Providence, RI, 2000.
- [116] E. Teske, H.C. Williams, *A Note on Shanks' Chains of Primes*, University of Waterloo, 2000.
- [117] U.V. Vazirani, V.V. Vazirani, *Efficient and Secure Pseudo-Random Number Generator*, en *Proceedings of the IEEE 25th Annual Symposium on Foundations of Computer Science*, pp. 458–463, 1984.
- [118] Waterloo Maple, Inc., <http://www.maplesoft.com>.
- [119] D. Welsh, *Codes and Cryptography*, Oxford University Press, Oxford, 1988.
- [120] M.J. Wiener, *Cryptanalysis of Short RSA Secret Exponents*, IEEE Transactions on Information Theory **36** (1990), 553–558.
- [121] H.C. Williams, *A Modification of the RSA Public-Key Encryption Procedure*, IEEE Transactions on Information Theory **26** (1980), 726–729.
- [122] H.C. Williams, *A $p+1$ Method of Factoring*, Mathematics of Computation **39** (1982), 225–234.
- [123] H.C. Williams, *Some Public-Key Crypto-Functions as Intractable as Factorization*, Cryptologia **9** (1985), 223–237.
- [124] Wolfram Research, Inc., <http://www.wolfram.com>.
- [125] S. Yates, *Sophie Germain Primes*, en G.M. Rassias, editor, *The mathematical heritage of C.F. Gauss*, pp. 882–886, World Scientific Publ. Co., River Edge, NJ, 1991.