Universidad de la República

Facultad de Ingeniería – Facultad de Ciencias

Test de primalidad y algoritmos de factorización en criptografía: aspectos matemáticos y computacionales

informe del proyecto de grado presentado como requisito para la obtención de los títulos:

Ingeniero en Computación

Licenciado en Matemática

(versión para ingeniería en computación)

Autor:
Bruno Hernández

Supervisores:
Dr. Claudio Qureshi
Dr. Alfredo Viola

Montevideo, diciembre de 2021

Agradecimientos

A mis tutores, Claudio y Tuba. En primer lugar le agradezco a ambos por haberme escuchado atentamente cuando les planteé qué quería hacer en mi Proyecto de Grado, por mostrar un interés genuino en colaborar en un trabajo de esta índole (¡ojalá sea el primero de muchos más!), y por brindarme el apoyo necesario durante el transcurso del mismo. Particularmente, le agradezco a Claudio por todas las veces que se puso a disposición para que yo pudiera lograr hacer un perfil de la Licenciatura en Matemática orientado a las ciencias de la computación. A Tuba le agradezco por compartir conmigo una óptica vanguardista sobre la dinámica universitaria, por haber confiado en mí para integrar su proyecto de investigación en criptografía (que asimismo constituyó mi primer empleo) y por su motivación constante para mi desarrollo tanto académico como profesional.

A los Doctores Eduardo Canale, Franco Robledo y Pablo Romero, porque a pesar del poco tiempo disponible, se presentaron entusiastas por conformar un tribunal evaluador para el día 30 de diciembre de 2021, lo cual permitió cumplir mi objetivo de culminar la carrera de Ingeniería en Computación. De igual manera quisiera agradecerles por sus comentarios que –sin duda alguna– enriquecieron este informe, pero por sobre todo me dejaron enseñanzas para encarar futuros proyectos de investigación.

A mis compañeros de la Facultad de Ingeniería, quienes en diferentes momentos me han ayudado a salir adelante de una u otra manera. Muy particularmente debo agradecer a aquellos que hoy en día puedo llamar *amigos*.

A mis familiares que me han apoyado durante todas mis etapas de formación, porque con una enorme paciencia y confianza en mí, siempre buscaron que tuviera las condiciones adecuadas para poder dedicarme a estudiar lo que me apasiona y alcanzar mis metas.

On the point of "doing the impossible" (...) I'd like to propose this meta-theorem of Cryptography: any apparently contradictory set of requirements can in fact be met with the right mathematical approach.

Ronald L. Rivest
2002 ACM A.M. Turing Award Lecture,
"The Early Days of RSA: History and Lessons"

Resumen

En la década de 1970 Diffie, Hellman y Merkle concibieron un paradigma criptográfico cuya idea revolucionaria fue emplear dos claves, una pública y otra privada, donde se requiere que ambas sean fáciles de generar pero al mismo tiempo no debería ser posible descubrir la privada a partir de la pública. Siguiendo sus pasos, Rivest, Shamir y Adleman publicaron el algoritmo de cifrado "RSA" con la idea de que las claves podrían ser generadas a partir del producto de dos grandes números primos (privados), puesto que todas las técnicas que se conocían para descomponer un número entero de esas características (lo público) eran ineficientes. Actualmente el mayor avance en la materia se debe a Shor, quien descubrió un algoritmo cuántico eficiente con dicho propósito. Sin embargo, ante la escasez de otros avances contundentes en complejidad computacional clásica, y dado que la implementación de computadoras cuánticas de gran porte sigue siendo un desafío, el protocolo RSA goza de plena vigencia. Luego, disponer de buenos test para conseguir números primos, así como conocer qué opciones hay para factorizar un entero e intentar romper RSA por esa vía, resulta un tema de gran importancia.

Esta tesis consiste en un relevamiento profundo de algunos de los test de primalidad y algoritmos de factorización más importantes que se conocen hasta la fecha dentro de la computación clásica. En este sentido, se exploran los aspectos matemáticos y computacionales buscando entender los fundamentos que los sustentan, así como los desafíos que estos imponen. Los test de primalidad más destacados que se incluyen en el relevamiento son: el test de Fermat (el cual ha inspirado numerosos métodos), el test de Miller-Rabin, el test de Goldwasser-Kilian (basado en curvas elípticas) y el test de Agrawal-Kayal-Saxena (AKS). Asimismo, los algoritmos de factorización tratados son: el método rho de Pollard, el método de curvas elípticas de Lenstra y la Criba cuadrática.

El presente *Proyecto de Grado* fue aprobado tanto por el Instituto de Computación de la Facultad de Ingeniería como por el Centro de Matemática de la Facultad de Ciencias, ambas instituciones de la Universidad de la República, como válido para optar a la doble titulación en Ingeniería en Computación y Licenciatura en Matemática. Considerando que el área relevada es extremadamente amplia, profunda y con enfoques matemáticos y computacionales que se complementan, se ha decidido presentar un informe específico para cada carrera priorizando esas diferentes perspectivas. Este documento jerarquiza los aspectos computacionales.

Palabras clave: test de primalidad, factorización de enteros, criptografía, matemática aplicada, ciencias de la computación.

Abstract

In the 1970s Diffie, Hellman and Merkle conceived a cryptographic paradigm which revolutionary idea was to make use of two keys, a private one and a public one, where it is requiered that both can be easily generated but at the same time it should not be possible to figure out the private one from the public one. Following in their footsteps, Rivest, Shamir and Adleman published the "RSA" encryption algorithm and the idea was that keys could be generated from the product of two big prime numbers (private), since every known techniques for decomposing a number like that (public) into its prime factors were inefficient. At the present the major breakthrough in this subject its due to Shor, who discovered an efficient quantum algorithm with that purpose. However, because of the lack of substantial progress in classical computational complexity, and given that the implementation of powerful quantum computers is still an open challenge, the RSA protocol continues to be widely used. Hence, disposing good tests to get prime numbers, as well as know the available options to factorize an integer and on this way try to break RSA, it turns out to be a relevant issue.

The purpose of this thesis is to make a deep research about some of the most important primality tests and factoring algorithms known to date within classical computing. In order to achieve this, both mathematical and computational aspects are explored looking forward to understand their fundamentals, as well as the challenges they impose. The most remarkable primality tests included in this survey are: the Fermat test (which has inspired many other methods), the Miller-Rabin test, the Goldwasser-Kilian test (based on elliptic curves) and the Agrawal–Kayal–Saxena (AKS) test. Likewise, the factoring algorithms treated are: Pollard's rho method, Lenstra's elliptic-curve method and the Quadratic sieve.

This *Final Year Project* was approved by the Computer Science Institute at Faculty of Engineering and by the Mathemathics Department at Faculty of Science, both institutions from the University of the Republic, to qualify for an Engineer's degree in Computer Science and a Bachelor's degree in Mathematics. Considering that the surveyed area is extremely wide, deep and it has both mathematical and computational complementary approaches, it has been decided to present a specific report for each carreer prioritizing those different perspectives. This document emphasises the computational aspects.

Key words: primality tests, integer factorization, cryptography, applied mathematics, computer science.

Lista de símbolos

A continuación se exhiben tres listas con todos los símbolos utilizados a lo largo del documento, junto con una descripción verbal de los mismos.

Conjuntos

$\mathbb{R}$; $\mathbb{R}^+$	Números reales ; reales positivos.
$[a, b]$; (a, b)	Intervalo cerrado de los números reales entre a y b ; intervalo abierto.
$\mathbb{N}$	Números naturales, donde se incluye al 0.
$\mathbb{Z}$; $\mathbb{Z}^+$	Números enteros ; enteros positivos.
$\mathbb{Z}_n$	Residuos módulo n, identificados con los enteros del 0 al $n-1$.
$\mathbb{Z}_n^\times$	Residuos módulo n que tienen inverso según la multiplicación módulo n.
$\mathbb{Z}[x]$	Polinomios con coeficientes en $\mathbb{Z}$.
$\mathbb{P}^2(\mathbb{K})$	Plano proyectivo definido sobre el cuerpo $\mathbb{K}$.
$E(\mathbb{K})$	Curva elíptica definida sobre el cuerpo $\mathbb{K}$.
$\ker(f)$	Núcleo del homomorfismo f.
$\mathrm{Im}(f)$	Imagen del homomorfismo f.
$O\big(f(k)\big)$	Funciones para las cuales la función $f(k)$ es una cota superior asintótica.
$\Omega\big(f(k)\big)$	Funciones para las cuales la función $f(k)$ es una cota inferior asintótica.
P	Clase de complejidad de los problemas resolubles en tiempo polinomial por un algoritmo determinista.
L_n	Mentirosos del test de Fermat (*Fermat liars*).
M_n	Mentirosos del test de Miller-Rabin (*strong liars*).
2^S	Conjunto potencia de un conjunto S (*todos los subconjuntos de* S).
$S^\complement$	Complemento del conjunto S.
S^n	Producto cartesiano del conjunto S consigo mismo n veces.
$\emptyset$	Conjunto vacío.

Funciones matemáticas

$f : S \to T$	Función f definida del conjunto S al conjunto T.
$n!$	Factorial de n.
$\binom{m}{k}$	Coeficiente binomial (*combinaciones de m tomadas de a k*).
$\gcd(a, b)$	Máximo común divisor entre a y b.
$a \pmod{n}$	Resto de la división entera de a entre n.
$\log$; $\log_b$	Logaritmo natural ; logaritmo en base b.
$\mathrm{ord}(a)$	Orden de a como elemento de un grupo.
$\mathrm{ord}_n(a)$	Orden de a como elemento del grupo $\mathbb{Z}_n$.
$\varphi(n)$	Cantidad de enteros coprimos con n entre 1 y n (*función φ de Euler*).
$\pi(x)$	Cantidad de números primos menores o igual a cierto número real x.
máx	Máximo de un conjunto de elementos comparables.
mín	Mínimo de un conjunto de elementos comparables.
$\lvert x \rvert$	Valor absoluto, en el caso de un número x.
$\lvert S \rvert$	Cantidad de elementos (*cardinal*), en el caso de un conjunto S.
$\lceil x \rceil$	Parte entera techo (*el entero más cercano a x por arriba*).
$\lfloor x \rfloor$	Parte entera piso (*el entero más cercano a x por abajo*).

Símbolos lógico-matemáticos

$\forall$	Cuantificador universal (*para todo*).
$\exists$	Cuantificador existencial (*existe*).
$:$	"Se tiene que", "sucede que", "se cumple que".
$/$	"Tal que".
$\wedge$	Conjunción lógica (y).
$\vee$	Disyunción lógica (o).
$\Rightarrow$	Implicancia lógica (*entonces*).
$\Longleftrightarrow$	Equivalencia lógica (*si y solo si*).
$x \leftarrow y$	Asignación (y *se asigna a* x).
$x \xleftarrow{\boxdot} S$	Asignación aleatoria (x *se elige uniformemente del conjunto* S).
$x \in S$; $x \notin S$	Pertenencia a un conjunto (x *pertenece a* S) ; "x no pertenece a S".
$S \subseteq T$	Inclusión de conjuntos (S *está incluido en* T).
$\cup$	Operador de unión de conjuntos.
$\cap$	Operador de intersección de conjuntos.
$\setminus$	Operador de diferencia de conjuntos.
$\sum$	Suma sobre un conjunto indexado (*sumatoria*).
$\prod$	Producto sobre un conjunto indexado (*productoria*).
$a \mid b$; $a \nmid b$	Divisibilidad (a *divide a* b) ; "a no divide a b".
$a \equiv b \pmod{n}$; $a \not\equiv b \pmod{n}$	Congruencia módulo n (a *es congruente con* b *módulo* n) ; "a no es congruente con b módulo n".
$\lim_{x \to \infty} f(x)$	"Límite cuando x tiende a infinito" de la función $f(x)$ con dominio $\mathbb{R}^+$.
∞	Infinito.
$\approx$	Aproximadamente.
$\blacksquare$	Final de una demostración.

Lista de abreviaturas

En este documento se mencionan las siguientes siglas, cuyo significado se detalla a continuación:

AES Advanced Encryption Standard (*algoritmo de encriptación simétrico*).

AKS Agrawal, Kayal y Saxena (*test de primalidad*).

CPU Central Processing Unit (*chip de computadora*).

GIMPS Great Internet Mersenne Prime Search (*proyecto colaborativo*).

NIST National Institute of Standards and Technology (*organismo gubernamental*).

RSA Rivest, Shamir y Adleman (*algoritmo de encriptación asimétrico*).

Índice

Resumen v

Lista de símbolos vii

Lista de abreviaturas x

1. Introducción 1

1.1. Un par de problemas elementales 1

1.2. Ampliando las fronteras 2

1.3. Objetivos del proyecto 4

1.4. Organización del documento 5

2. Marco teórico 7

2.1. Fundamentos matemáticos 7

2.1.1. Teoría de números 7

2.1.2. Álgebra 10

2.1.3. Curvas elípticas 13

2.1.4. Probabilidad 17

2.2. Fundamentos computacionales 19

2.2.1. Algoritmia 19

2.2.2. Costo de algunas rutinas 21

2.2.3. Criptosistema RSA 23

3. En búsqueda de los números primos 27

3.1. Aproximación naíf 27

3.2. Nuevos rumbos 30

3.3. ¿Aleatoriedad o Pseudoaleatoriedad? 32

3.4. Test de Fermat 35

3.5. Reforzando el test de Fermat 40

3.6. Test de Miller-Rabin 49

3.7. Test de Goldwasser-Kilian 56

3.8. La teoría de números ¿al rescate? 62

3.9. El proyecto GIMPS 65

3.10. Test AKS 69

4. Factorización de enteros **73**

4.1. Método rho de Pollard 73

4.2. Método de curvas elípticas de Lenstra 80

4.3. Criba cuadrática 84

5. Conclusiones y trabajo a futuro **87**

Referencias **92**

1. Introducción

1.1. Un par de problemas elementales

El conjunto de los números primos es para un matemático, lo que un balde de LEGOs es para un niño. Así como las piezas de este juguete se encajan unas con otras y le permiten al infante construir cualquier estructura que pueda imaginar con ellas, los números primos se multiplican entre sí y le permiten al matemático construir cualquier otro número entero (mayor a la unidad) que desee. Tal es la importancia que tiene este resultado aritmético para la comunidad matemática, que el mismo ha sido bautizado dentro del campo como *Teorema fundamental*. Es así que los números primos –formalmente, aquellos números naturales que tienen solo dos divisores positivos– son los auténticos *building blocks* de los números enteros.

Sin embargo, esta analogía omite una diferencia fundamental: para que un niño pueda jugar con sus bloquecitos basta con que alguien vaya a una tienda y le compre el recipiente que los contiene, pero para los matemáticos –hacerse de los números primos– es un problema abierto desde hace más de 2.300 años, cuando Euclides en sus *Elementos* dio el primer paso demostrando que estos son infinitos. No obstante, su infinitud no es lo que complica aquí las cosas, ya que muchísimos otros conjuntos de números son infinitos y los matemáticos tienen sus elementos precisamente ubicados en la recta real. Por alguna razón, los números primos se han resistido a todos los intentos de revelar –de una manera conveniente– dónde están desparramados a lo largo de todos los números. Y así se llega hasta el día de hoy, donde en la práctica siguen sin respuesta algunas preguntas tan elementales como «¿cuál es el n-ésimo número primo?» Si bien el poder de la computación ha ayudado enormemente a responder esta pregunta para las primeras decenas de miles de millones de instancias, según [BMST13, p. 324] actualmente no se conoce una fórmula general que responda esta pregunta de manera eficiente *cualquiera* sea el índice n.

Otra pregunta igual de elemental que ha atraído la atención de los matemáticos es la inversa de la anterior: «¿es primo el n-ésimo número natural?» Como se verá más adelante, a lo largo de la historia han surgido diferentes métodos para atacar esto, pero no fue hasta el cercano año 2002 que el problema quedó resuelto de manera contundente. En el contexto de este trabajo, un ***test de primalidad*** hará referencia a un algoritmo cuyo cometido es intentar dar una respuesta a esta cuestión.

Ligado estrechamente con testear primalidad, naturalmente está el problema de saber qué números primos componen a otro el cual se sabe que no es primo (*i. e.*, que es *compuesto*). Tal es el vínculo, que desde el principio el propio Euclides –también en sus *Elementos*– demostró una serie resultados que conducen al Teorema fundamental de la aritmética, igualmente conocido como *Teorema de factorización única*. Es así que en el presente trabajo se le denominará ***algoritmo de factorización*** a cualquier método con este propósito. Desde ya se menciona que, al igual que la interrogante del n-ésimo número primo, inventar un algoritmo de factorización eficiente sigue siendo un desafío abierto. Pero esto que a priori parecería ser una "piedra en el zapato" para el progreso de las matemáticas, ¡resultó ser la **piedra angular** de la seguridad en la inmensa mayoría de las transacciones de información del mundo moderno!

1.2. Ampliando las fronteras

Cuando en 1976 Whitfield Diffie y Martin Hellman publicaron su trabajo *New Directions in Cryptography* [DH76], dieron inicio a lo que más adelante se le llamó *criptografía asimétrica*. Este nuevo paradigma criptográfico traía consigo la solución a la problemática más importante que hasta ese entonces tenían los sistemas criptográficos (luego llamados *simétricos*): asegurar el acuerdo de claves en un canal inseguro. En el fondo, el cambio de paradigma tenía la premisa de que como las comunicaciones digitales serían esenciales en el futuro, todos los resultados fundamentales sobre clasificación de problemas de acuerdo a su complejidad, serían aplicables para resolver este desafío. De acuerdo a la teoría de la complejidad computacional, un algoritmo es *eficiente* en el uso de algún recurso si su consumo en función del tamaño de la entrada tiene un crecimiento polinomial. Típicamente los recursos analizados son el tiempo de ejecución y el espacio de almacenamiento, por eso existe una clase particular de problemas, cuya solución en todas y cada una de las instancias se puede encontrar con un algoritmo *determinista* y *temporalmente eficiente*, llamada clase P [1]. Entonces, el nuevo paradigma de seguridad estaba relacionado con el hecho de que hay problemas para los cuales se desconocen algoritmos eficientes que los resuelvan genéricamente, y se conjetura que no los tengan.

A partir del artículo de Diffie y Hellman, las primitivas criptográficas asimétricas tendrían la particularidad de que todos los datos para acordar una clave serían expuestos públicamente, de tal forma que cualquier atacante con suficiente tiempo (o poder compu-

[1] Para simplificar el discurso, de aquí en adelante el uso del término "eficiente" hará referencia al aspecto temporal, a menos que se especifique lo contrario.

tacional) podría recuperar dicha clave. Pero ¿cuánto es suficiente tiempo? Si uno va a la descripción del protocolo se encuentra con que su seguridad está basada en un problema matemático denominado *logaritmo discreto*, y puesto que desde su publicación hasta la actualidad no se han descubierto técnicas eficientes para resolverlo de manera genérica, se supone que quebrar una instancia aleatoria de tamaño adecuado no es factible. En otras palabras, el paradigma de seguridad está basado en trabajar con "problemas difíciles".

Luego de que Diffie y Hellman marcaran una tendencia al utilizar problemas matemáticos "difíciles de resolver" como paradigma de seguridad, se publicaron otros algoritmos criptográficos de la misma índole. En esa dirección, al año siguiente los criptógrafos Ronald Rivest, Adi Shamir y Leonard Adleman inventaron su famoso algoritmo *RSA* para encriptación [RSA78], cuyo "problema difícil" de base es la **factorización de enteros**. Por tal motivo, el estudio de test de primalidad y algoritmos de factorización ha estado presente desde los inicios de la criptografía moderna.

En relación a los test de primalidad, varios algoritmos probabilísticos se presentaron inmediatamente, con probabilidad de error *negligible* (esto es, exponencialmente chica en el tamaño del problema) y asimismo muy eficientes. Sin embargo, tuvieron que pasar más de dos décadas para que los científicos de la computación Manindra Agrawal, Neeraj Kayal y Nitin Saxena descubrieran el primer algoritmo determinista y de tiempo polinomial, al que llamaron *test AKS* y probaron así que el problema de **testear primalidad está en P**. Es interesante notar que este avance mayúsculo en complejidad no es acompañado por una eficiencia práctica, dado que en los hechos lo más eficiente sigue siendo usar algoritmos probabilísticos, como por ejemplo el *test de Miller-Rabin*.

En relación al problema de factorización, si bien se han realizado grandes avances, lejos se está de encontrar algoritmos eficientes. Actualmente la *Criba general del cuerpo de números* posee el mejor desempeño en términos asintóticos [vzG15, p. 131], sin embargo, la *Criba cuadrática de Pomerance* así como el *método de curvas elípticas de Lenstra* tienen mejor rendimiento para enteros de hasta cierto tamaño. Un detalle no menor omitido hasta el momento es que toda la algoritmia ya mencionada se enmarca dentro de la *computación clásica*, paradigma que se basa en la electrónica digital de ceros y unos. Por otro lado, desde la década de 1980 se ha ido desarrollando un nuevo paradigma de computación basado en la mecánica cuántica, al que se le llamó *computación cuántica*. En este sentido, en 1997 el matemático Peter Shor presentó el primer algoritmo cuántico para factorizar enteros en tiempo polinomial [Sho97], lo que desencadenó toda una nueva área de investigación la cual a la postre se denominó *criptografía postcuántica*. Por este motivo, entender bien los algoritmos clásicos es indispensable para tener bases sólidas en relación

a los fundamentos matemáticos detrás de estos, siendo además un trabajo sustancial para posteriormente comprender propuestas basadas en criptografía postcuántica.

1.3. Objetivos del proyecto

El presente proyecto tiene como objetivo primordial satisfacer simultáneamente los requerimientos de un *Proyecto de Grado* válido para las carreras Ingeniería en Computación (por Facultad de Ingeniería) y Licenciatura en Matemática (por Facultad de Ciencias). Por lo tanto, este se encuentra inmerso en un proyecto de mayores dimensiones, que es **propiciar la integración de las matemáticas con la computación** en la formación universitaria impartida en Uruguay, particularmente en la Universidad de la República. Es importante aclarar que si bien la temática abordada es una sola, se ha elaborado una versión del informe del Proyecto de Grado propia para cada carrera, haciendo énfasis en el contenido inherente a cada una.

Dentro de los objetivos específicos de este proyecto, se encuentran:

1) Relevar **test de primalidad probabilísticos**, haciendo un estudio detallado de al menos uno de ellos. Asimismo, dentro de este objetivo se busca hacer un leve acercamiento a un problema de amplio interés dentro de la criptografía y otras áreas, que es la generación de números pseudoaleatorios.

2) Estudiar detalladamente el **test AKS** mencionado en la subsección anterior, comparándolo con otros test que ya existían pero no poseen alguna de las cualidades que hicieron de este la prueba fehaciente de que "los primos están en P" [2].

3) Estudiar detalladamente algoritmos de factorización basados en distintos objetos matemáticos, como los sistemas dinámicos y las curvas elípticas. También es parte de este objetivo relevar un algoritmo avanzado de cribado, como la **Criba general del cuerpo de números** o bien la **Criba cuadrática**.

4) Si el tiempo lo permite, se buscará implementar algunos de los algoritmos presentados en las secciones relativas a los objetivos anteriores, a efectos de comparar sus tiempos de ejecución con entradas estratégicamente seleccionadas. Sin embargo, este punto no es un requerimiento fundamental para el proyecto.

Al finalizar este proyecto se espera que el estudiante maneje con solvencia el estado del arte de la temática abordada, así como que sea capaz de exponer rigurosamente los

[2] *PRIMES is in* P es el título original de la publicación donde este algoritmo se dio a conocer.

fundamentos matemáticos que dan sustento a los algoritmos descriptos a lo largo del trabajo.

También se espera que el estudiante entienda integralmente la importancia teórica que tuvo el algoritmo AKS, pero simultáneamente conozca otras soluciones que muchas veces se prefieren en la criptografía práctica.

En síntesis, con el presente Proyecto de Grado se busca que su autor comprenda los fundamentos y aplicaciones de los problemas de testear primalidad y la factorización de enteros en el área de la criptografía (particularmente a partir del algoritmo RSA presentado en 1978), y quede capacitado para profundizar en temas de investigación actuales en dicha área.

1.4. Organización del documento

El presente trabajo académico se encuentra dividido en cinco secciones, las cuales –a efectos de facilitar la organización de la información– a su vez se dividen internamente en varias subsecciones.

En la Sección 2 se presentan las definiciones, especificaciones y resultados más elementales que se han requerido para llevar a cabo los objetivos planteados. Dado que esta tesis pretende crear un nexo entre el ámbito matemático y el de las ciencias de la computación, se han definido dos subsecciones de modo que el público objetivo pueda enfocarse en el ámbito que no es su especialidad.

En la Sección 3 se presentan todos los test de primalidad que se decidió abarcar. Esta se divide en diez subsecciones, donde la primera comienza con la aproximación más directa al problema abordado, luego se discute la generación de números aleatorios para dar paso a los test probabilísticos, y finalmente se retoman los algoritmos deterministas para concluir en el test AKS. Se destaca que en la penúltima subsección se examina un proyecto de escala mundial para buscar números primos con cierta forma particular.

En la Sección 4 se presentan todos los algoritmos de factorización de enteros que fueron incluidos en el relevamiento. Asimismo esta se divide en tres subsecciones, donde en la primera se presenta una clásico de la materia, en la segunda se emplean las curvas elípticas para romper la barrera del tiempo exponencial, y en la última se describe uno de los métodos más rápidos conocidos hasta la fecha.

En la Sección 5 se recuerdan los objetivos trazados y en base a ellos se exponen las conclusiones del proyecto. Considerando las limitaciones temporales del mismo, en esta parte también se deja planteado el trabajo a futuro que se ha estimado pertinente.

2. Marco teórico

En esta sección se exhibirán un conjunto de definiciones y resultados matemáticos y computacionales que son elementales para el desarrollo de los temas abordados en la presente tesis.

Dado que dichos resultados son –en cierto sentido– "auxiliares" para los objetivos del proyecto, no es de interés relevar sus demostraciones. No obstante, en cada uno de ellos se dará una referencia bibliográfica donde el lector puede remitirse para comprobar su correctitud u obtener más detalles sobre el mismo.

2.1. Fundamentos matemáticos

En general todo lo expuesto en esta subsección se basa en los libros [Ste09], [Hun14], [Kob94], [vzG15] y [Res14]. En el caso de las proposiciones, lemas y teoremas se detalla en qué páginas el lector podrá encontrarlos con más información al respecto.

2.1.1. Teoría de números

Definición 2.1. Sean a, $b \in \mathbb{Z}$. Se dice que a ***divide*** a b si existe $k \in \mathbb{Z}$ tal que $b = ak$, y la relación se simboliza como $a \mid b$. En este caso también se dice que a es un ***divisor*** de b, o equivalentemente que b es un ***múltiplo*** de a. Finalmente, un divisor de b es ***trivial*** si se trata del 1 o el propio b.

A partir de lo anterior queda claro que 1 divide a cualquier número y asimismo cualquier número divide al 0. Por el primer hecho, tiene sentido hablar de los *divisores comunes* entre un par de enteros y en particular del mayor de ellos.

Definición 2.2. Sean a, $b \in \mathbb{Z}$. El ***máximo común divisor*** de a y b es el mayor $d \in \mathbb{Z}$ tal que $d \mid a \ \wedge \ d \mid b$. Abuso de notación mediante, a este número se lo representa con la notación funcional $\gcd(a, b)$ [1]. Convencionalmente se toma $\gcd(0, 0) = 0$, y cuando $\gcd(a, b) = 1$ se dice que a y b son ***coprimos***, *primos entre sí* o también *primos relativos*.

[1] Es decir, sabiendo que el máximo común divisor existe, se puede considerar la función $\gcd : \mathbb{Z}^2 \to \mathbb{N}$ cuyo resultado es eso mismo. En la Subsección 2.2.2 se trata la computabilidad de esta función.

Notar que si un número negativo divide a otro, entonces también lo divide su opuesto. Por lo tanto, el máximo de los divisores va a ser un número positivo. En consecuencia, si alguno de los números considerados es negativo, es lo mismo calcular el máximo común divisor empleando su contraparte positiva. Observar también que $\gcd(a, b) = \gcd(b, a)$.

Proposición 2.3. [Ste09, p. 4] *Sean a, b y n enteros cualesquiera. Se cumple que* $\gcd(a, b) = \gcd(a, b - an)$.

Según sus divisores, los números pueden clasificarse de la siguiente manera:

Definición 2.4. Un número natural $n > 1$ es ***primo*** si tiene solo dos divisores positivos. Por el contrario, si n no es primo, se dice que es ***compuesto***.

Evidentemente en la definición de *número primo* los divisores en cuestión son el 1 y el propio n. Consecuentemente las definiciones de *número compuesto* y *número primo* son equivalentes, respectivamente, a tener o no divisores no triviales positivos.

Un resultado ampliamente utilizado en teoría de números que vincula divisibilidad con primalidad es el siguiente:

Lema 2.5. [Ste09, p. 7] ***Lema de Euclides:*** *Sean $a, b \in \mathbb{Z}$ y p un número primo. Si $p \mid ab$, entonces $p \mid a \ \vee \ p \mid b$.*

Y tres hechos fundamentales en la teoría son los próximos tres teoremas:

Teorema 2.6. [Ste09, p. 10] ***Teorema fundamental de la aritmética*** o ***teorema de factorización única:*** *Cualquier número natural mayor que 1 puede ser escrito como un producto de números primos, donde esta factorización es única salvo por el orden* [2].

Teorema 2.7. [Ste09, p. 11] *El conjunto de los números primos es infinito.*

Teorema 2.8. [Ste09, p. 16] ***Teorema de los números primos:*** *Sea la función $\pi : \mathbb{R}^+ \to \mathbb{N} \ / \ \pi(x) = |\{p \in \mathbb{N} \ / \ p \leq x \ \wedge \ p \text{ es primo}\}|$. Se cumple que $\pi(x)$ es equivalente asintóticamente a $\frac{x}{\log(x)}$, en el sentido de que $\lim_{x\to\infty} \frac{\pi(x)}{x/\log(x)} = 1$.*

Continuando con cuestiones de divisibilidad se tiene lo siguiente:

[2] Por convención se considera que los números que en sí son primos corresponden a un producto unitario.

Teorema 2.9. [Ste09, p. 4] ***Teorema de la división entera:*** *Sean $a, b \in \mathbb{Z}$ con $b \neq 0$. Entonces existen únicos $q, r \in \mathbb{Z}$ que verifican $a = bq + r$ y $0 \leq r < |b|$. El número q se denomina **cociente** y al número r se le dice **resto** o **residuo**. Al resto r de la división entera también se lo representará con la notación «$a \pmod b$».*

Definición 2.10. Sean $a, b \in \mathbb{Z}$ y $n \in \mathbb{Z}^+$. Se dice que a es ***congruente*** con b ***módulo n*** si ambos tienen el mismo resto al dividirlos entre n. A esta relación se la simboliza como $a \equiv b \pmod n$, y es equivalente a decir que $n \mid a - b$.

Teorema 2.11. [Ste09, p. 29] ***Teorema chino del resto****: Sean a y b enteros cualesquiera y $m, n \in \mathbb{Z}^+$ tales que $\gcd(m, n) = 1$. Luego, el sistema de congruencias* $\begin{cases} x \equiv a \pmod m \\ x \equiv b \pmod n \end{cases}$ *determina una única solución x_0 módulo mn.*

Un caso destacable del teorema previo es cuando $b = a$, ya que aquí (manteniendo la hipótesis de m y n coprimos) vale que $\begin{cases} x \equiv a \pmod m \\ x \equiv a \pmod n \end{cases} \iff x \equiv a \pmod{mn}$.

De hecho, tanto este caso como la formulación general pueden extenderse cualquiera sea la cantidad de congruencias siempre que todos los módulos sean coprimos entre ellos. Esta última formulación quedaría como: *Dados $a_1, \ldots, a_k$ enteros cualesquiera, si $m_1, \ldots, m_k$ son enteros positivos coprimos dos a dos, entonces el sistema* $\begin{cases} x \equiv a_1 \pmod{m_1} \\ \quad \vdots \\ x \equiv a_k \pmod{m_k} \end{cases}$ *determina una única solución x_0 módulo $m_1 \cdots m_k$.*

Finalmente se introduce la denominada ***función φ de Euler*** como $\varphi : \mathbb{Z}^+ \to \mathbb{N}$ / $\varphi(n) = |\{a \in \mathbb{N} \ / \ 1 \leq a \leq n \ \wedge \ \gcd(a, n) = 1\}|$. Luego, mediante una aproximación progresiva que parte del caso trivial cuando p es primo $\big(\varphi(p) = p - 1\big)$, continua con la expresión para potencias de p primo $\big(\varphi(p^k) = (p-1)p^{k-1}\big)$ y termina con la prueba de que φ es una función multiplicativa $\big($si $\gcd(m, n) = 1 \ \Rightarrow \ \varphi(mn) = \varphi(m)\varphi(n)\big)$, uno llega a la fórmula general en función de la factorización del número:

Proposición 2.12. [Ste09, p. 31] *Si $n = {p_1}^{\alpha_1} \ldots {p_s}^{\alpha_s}$ siendo los p_i números primos distintos y los α_i exponentes positivos, entonces $\varphi(n) = \prod_{i=1}^{s} (p_i - 1){p_i}^{\alpha_i - 1}$.*

Y un hecho remarcable que involucra a esta función es el:

Teorema 2.13. [Ste09, p. 26] ***Teorema de Euler:*** *Sean $n \in \mathbb{Z}^+$ y $a \in \mathbb{Z}$ coprimos. Se cumple que $a^{\varphi(n)} \equiv 1 \pmod n$.*

2.1.2. Álgebra

Proposición 2.14. [Hun14, pp. 26-29] *La congruencia módulo n es una relación de equivalencia sobre $\mathbb{Z}$, esto es, cumple las propiedades reflexiva, simétrica y transitiva. Es un hecho que el conjunto $\mathbb{Z}_n = \{0, \ldots, n-1\}$ contiene todos los representantes de las clases de equivalencia de esta relación.*

Definición 2.15. Sea G un conjunto y $*$ una operación binaria en los elementos de G. Se dice que el par $(G, *)$ es un ***grupo*** si cumple con las siguientes propiedades:

1) Cerradura: $\forall\, a,\, b \in G \,:\, a * b \in G$

2) Asociatividad: $\forall\, a,\, b,\, c \in G \,:\, (a * b) * c = a * (b * c)$

3) Neutro: $\exists\, e \in G \;/\; \forall a \in G \,:\, a * e = e * a = a$

4) Inversos: $\forall\, a \in G \,:\, \exists\, x \in G \;/\; a * x = x * a = e$

La palabra "inverso" (*x es el inverso de a*) se emplea cuando la operación del grupo se denota como algún tipo de multiplicación, y cuando dicha operación es referida como algún tipo de suma, la palabra empleada es "opuesto" (*x es el opuesto de a*). En el primer caso al elemento x se lo denota a^{-1}, mientras que en el segundo se escribiría $-a$. Ocasionalmente se referirá al grupo solo con el conjunto si la operación es sobreentendida.

Definición 2.16. Un ***grupo*** $(G, *)$ se dice ***conmutativo*** o ***abeliano*** si además de las cuatro anteriores, cumple con la siguiente propiedad:

5) Conmutatividad: $\forall\, a,\, b \in G \,:\, a * b = b * a$

Proposición 2.17. [Hun14, p. 177] *$\mathbb{Z}_n$ y $+ : \mathbb{Z}_n \to \mathbb{Z}_n \;/\; a + b = (a + b) \pmod{n}$ (siendo $+$ la suma usual de los enteros) forman un grupo conmutativo llamado los **enteros módulo n**, donde el neutro del grupo es el 0. De aquí en más se utilizará el símbolo $+$ para representar la suma del grupo, quedando a cargo del lector deducir su contexto de aplicación.*

De manera análoga, también es posible definir una multiplicación para los elementos de $\mathbb{Z}_n$ como $\cdot : \mathbb{Z}_n \to \mathbb{Z}_n \;/\; a \cdot b = (ab) \pmod{n}$. Al igual que con la multiplicación usual de los enteros se omitirá el símbolo $\cdot$ para representar esta operación, quedando nuevamente a cargo del lector deducir su contexto de aplicación.

Lema 2.18. [Hun14, p. 40] *Sean $a \in \mathbb{Z}$ y $n \in \mathbb{Z}^+$. La congruencia $ax \equiv 1 \pmod{n}$ tiene una única solución $x \in \mathbb{Z}_n$ si y solo si* $\gcd(a, n) = 1$.

Proposición 2.19. [Hun14, p. 179] *El conjunto* $\mathbb{Z}_n^\times = \{a \in \mathbb{Z}_n \ / \ \gcd(a,n) = 1\}$ *y la multiplicación en* $\mathbb{Z}_n$ *definida anteriormente forman un grupo conmutativo llamado los* ***invertibles módulo n****.*

Observar que la función φ de Euler cuantifica el tamaño del grupo de los invertibles módulo n, esto es, $\forall\, n \in \mathbb{Z}^+ \,:\, |\mathbb{Z}_n^\times| = \varphi(n)$.

Definición 2.20. Dado un grupo $(G, *)$, un ***subgrupo*** de G es un conjunto $H \subseteq G$ que cumple con las siguientes propiedades:

1) Cerrado por la operación: $\forall\, a,\, b \in H \,:\, a * b \in H$

2) Contiene al neutro: $e \in H$

3) Cerrado por inversos: $\forall\, a \in H \,:\, a^{-1} \in H$

En otras palabras, $H \subseteq G$ es un subgrupo de $(G, *)$ si es un grupo con la operación $*$. [3]

Un resultado destacado que vincula grupos y subgrupos es lo que se establece a continuación.

Teorema 2.21. [Hun14, p. 241] ***Teorema de Lagrange para grupos finitos:*** *Si* $(G, *)$ *es un grupo finito y* H *un subgrupo de* G*, entonces* $|H| \,|\, |G|$.

Si $(G, *)$ es un grupo y g un elemento del mismo, uno puede definir las potencias de g como $g^0 = e$ (siendo e el neutro del grupo), y para $k \in \mathbb{Z}^+$ que $g^k = \underbrace{g * g * \ldots * g}_{k \text{ veces } g}$ y $g^{-k} = \underbrace{g^{-1} * g^{-1} * \ldots * g^{-1}}_{k \text{ veces } g^{-1}}$. De esta forma se puede probar que valen las reglas usuales sobre los exponentes, y además $\forall\, g,\, h \in G \,:\, (gh)^{-1} = h^{-1}g^{-1}$. Luego, a partir de este concepto se puede establecer la siguiente:

Definición 2.22. Dado un grupo $(G, *)$ y un elemento $g \in G$, el ***orden*** de g se define como $\operatorname{ord}(g) = \begin{cases} \infty & , \text{ si } \ \forall\, k \in \mathbb{Z}^+ : g^k \neq e \\ \min\{k \in \mathbb{Z}^+ \ / \ g^k = e\} & , \text{ si no} \end{cases}$.

En el caso de $G = (\mathbb{Z}_n^\times, \cdot)$ la notación se hará más sugestiva, siendo esta « $\operatorname{ord}_n(g)$ ».

Proposición 2.23. [Hun14, p. 200] *Sean* $(G, *)$ *un grupo finito,* $g \in G$ *y* $k \in \mathbb{Z}^+$*. Se cumple que* $g^k = e \iff \operatorname{ord}(g) \,|\, k$.

[3] Notar que la asociatividad de la operación del grupo G es heredada por H.

Definición 2.24. Dado $(G, *)$ un grupo y $g \in G$, el ***subgrupo generado*** por g es $\langle g \rangle = \{g^k \ / \ k \in \mathbb{Z}\}$. Si $\exists\, g \in G \ / \ G = \langle g \rangle$, se dice que G es un grupo ***cíclico*** y que g es un ***generador*** del grupo.

Luego, a partir de lo anterior se tiene:

Teorema 2.25. [Hun14, pp. 206-207] *Si $(G, *)$ un grupo, entonces $\forall\, g \in G : \langle g \rangle$ es un subgrupo de G. Además si $\text{ord}(g) = n$, entonces $\langle g \rangle = \{g^0, g^1, \ldots, g^{n-1}\}$ con lo cual $|\langle g \rangle| = \text{ord}(g)$.*

Observar que si ahora se aplica el *Teorema de Lagrange para grupos finitos*, se puede deducir que para cualquier grupo G finito y para cualquier elemento g en el mismo, el $\text{ord}(g) \,|\, |G|$ y en particular $g^{|G|} = e$.

Un caso de particular interés es el grupo de los invertibles módulo n, pues si $\mathbb{Z}_n^\times$ es cíclico, a sus generadores se los denomina ***raíces primitivas módulo n***. El siguiente teorema responde cuándo es seguro que existen raíces primitivas.

Teorema 2.26. [Ste09, pp. 42-43] *Para todo número primo p se tiene que $\mathbb{Z}_p^\times$ es cíclico. Más aún, $\mathbb{Z}_n^\times$ es cíclico si y solo si $n = 2$, 4, p^k o $2p^k$ con p primo impar y $k \in \mathbb{Z}^+$.*

Definición 2.27. Sean $(G_1, *)$ y $(G_2, \bullet)$ dos grupos. Una función $f : G_1 \to G_2$ es un ***homomorfismo*** si $\forall\, a, b \in G_1 \ : \ f(a * b) = f(a) \bullet f(b)$.

Los homomorfismos de grupos determinan dos conjuntos que, como se verá en próximo el teorema, poseen una importante cualidad.

Definición 2.28. Dado un homomorfismo de grupos $f : G_1 \to G_2$, el ***núcleo*** o ***kernel*** de f es el conjunto $\ker(f) = \{a \in G_1 \ / \ f(a) = e_2\}$, donde e_2 es el neutro de G_2. Por su lado, la ***imagen*** de f es el conjunto $\text{Im}(f) = \{b \in G_2 \ / \ b = f(a) \text{ para algún } a \in G_1\}$.

Teorema 2.29. [Hun14, pp. 221 y 264] Para cualquier homomorfismo de grupos $f : G_1 \to G_2$ se tiene que el $\ker(f)$ es un subgrupo de G_1, mientras que la $\text{Im}(f)$ es un subgrupo de G_2.

Una estructura algebraica que generaliza la noción de *grupo* es la que se define a continuación.

Definición 2.30. Sea R un conjunto dotado con dos operaciones $+$ y $\cdot$ binarias en los elementos de R. Se dice que la tripleta $(R, +, \cdot)$ es un ***anillo*** [4] si satisface las siguientes propiedades:

1) $(R, +)$ es un grupo abeliano, con su elemento neutro denotado por $\mathbf{0}$

2) Cerradura para el producto: $\forall\, a,\, b \in R \,:\, a \cdot b \in R$

3) Asociatividad para el producto: $\forall\, a,\, b,\, c \in R \,:\, (a \cdot b) \cdot c = a \cdot (b \cdot c)$

4) Propiedades distributivas con respecto a la suma:
$\forall\, a,\, b,\, c \in R \,:\, a \cdot (b + c) = a \cdot b + a \cdot c \;\;\wedge\;\; (a + b) \cdot c = a \cdot c + b \cdot c$

5) Neutro para el producto: $\exists\, \mathbf{1} \in R \setminus \{\mathbf{0}\} \;/\; \forall a \in R \,:\, a \cdot \mathbf{1} = \mathbf{1} \cdot a = a$

Cuando $\forall\, a,\, b \in R \,:\, a \cdot b = b \cdot a$ se dice que el anillo es ***conmutativo***, y cuando un anillo sea conmutativo y además todo elemento diferente del $\mathbf{0}$ sea invertible, se estará en presencia de la siguiente estructura:

Definición 2.31. Sea $(\mathbb{K}, +, \cdot)$ un anillo conmutativo. Se dice que $\mathbb{K}$ es un ***cuerpo*** si $\forall\, a \in \mathbb{K} \setminus \{\mathbf{0}\} \,:\, \exists\, x \in \mathbb{K} \;/\; a \cdot x = \mathbf{1}$. En este caso se denota a x por a^{-1}.

De esta manera, un *cuerpo* es una estructura algebraica en la cual se pueden efectuar con naturalidad todas las operaciones de la aritmética básica: adición, substracción (sumar opuestos), producto y división (multiplicar inversos).

Lema 2.32. [Hun14, p. 49] Si p es un número primo, entonces $(\mathbb{Z}_p, +, \cdot)$ es un cuerpo.

Definición 2.33. La ***característica*** de un anillo $(R, +, \cdot)$ es el menor $n \in \mathbb{N}$ para el cual $n\mathbf{1} = \underbrace{\mathbf{1} + \cdots + \mathbf{1}}_{n \text{ veces } \mathbf{1}} = \mathbf{0}$. Si tal n no existe, entonces se dice que R tiene ***característica cero***.

2.1.3. Curvas elípticas

Esta subsección está basada en la presentación del tema que se hace en los libros [Kob94] (Capítulo VI) y [vzG15] (Capítulo 5).

Si $\mathbb{K}$ es un cuerpo con neutros 0 y 1 (para la suma y el producto), se define $\mathbb{P}^2(\mathbb{K})$ como el *plano proyectivo sobre* $\mathbb{K}$, que consiste en las clases de equivalencia $(x : y : z)$ de una relación en la que, para una tripleta $(x, y, z) \in \mathbb{K}^3$ no nula, sus equivalentes son los

[4] Formalmente, lo que se define con las siguientes reglas es un *anillo con unidad*.

múltiplos escalares de ella. En otras palabras, los puntos que conforman este plano se llaman *puntos proyectivos* y se los denota por $(x : y : z) = \{\lambda(x, y, z) \ / \ \lambda \in \mathbb{K} \ \wedge \ \lambda \neq 0\}$.

Si $z \neq 0$ entonces $(x : y : z) = \left(\frac{x}{z} : \frac{y}{z} : 1\right)$, por lo tanto, sería válido enunciar que $\mathbb{P}^2(\mathbb{K}) \ = \ \{(x : y : 1) \ / \ x, \ y \in \mathbb{K}\} \ \cup \ \{(x : y : 0) \ / \ x, \ y \in \mathbb{K} \ \ \wedge \ (x, y) \neq (0, 0)\}$. En esta descomposición del plano proyectivo, el primer conjunto se identifica con el "plano afín" y el segundo se conoce como "puntos en el infinito".

En geometría proyectiva, a cada polinomio $F(x, y) = \sum a_{ij} x^i y^j$ de grado d con coeficientes $a_{ij} \in \mathbb{K}$ para el cual la ecuación $F(x, y) = 0$ describe una curva $\mathcal{C}$, se le asocia su versión homogénea $\tilde{F}(x, y, z) = \sum a_{ij} x^i y^j z^{d-i-j}$, porque lo que interesa estudiar es la *ecuación proyectiva* $\tilde{F}(x, y, z) = 0$. Esta determina la *curva proyectiva* $\mathcal{C}(\mathbb{K}) = \{(x : y : z) \in \mathbb{P}^2(\mathbb{K}) \ / \ \tilde{F}(x, y, z) = 0\}$, que resulta de mayor provecho pues en el desarrollo de la teoría será necesario aplicar el *Teorema de Bézout* (el mismo establece que dos curvas proyectivas sin componentes comunes cuyos respectivos grados son m y n, se intersectan en mn puntos contando multiplicidades). Así como el plano proyectivo se podía dividir en dos partes, la versión proyectiva de la curva también puede verse como $\mathcal{C}(\mathbb{K}) = \underbrace{\{(x : y : 1) \in \mathbb{P}^2(\mathbb{K}) \ / \ F(x, y) = 0\}}_{\text{"los puntos afines de } \mathcal{C}\text{"}} \cup \underbrace{\{(x : y : 0) \in \mathbb{P}^2(\mathbb{K}) \ / \ \tilde{F}(x, y, 0) = 0\}}_{\text{"los puntos en el infinito de } \mathcal{C}\text{"}}$.

Luego, en este contexto de la geometría proyectiva, si $\mathbb{K}$ es un cuerpo con característica distinta de 2 y 3, una ***curva elíptica*** sobre $\mathbb{K}$ es la versión proyectiva de una curva E definida por la ecuación $\boldsymbol{y^2 = x^3 + ax + b}$, donde los coeficientes a, $b \in \mathbb{K}$ y además verifican que $4a^3 + 27b^2 \neq 0$ [5]. La homogeneización del polinomio en la igualdad anterior deja la ecuación proyectiva $y^2 z = x^3 + axz^2 + bz^3$, entonces para ver sus puntos en el infinito, tomando $z = 0$ queda que $x^3 = 0$ y esto se satisface para $x = 0$. Sin embargo, la única clase de equivalencia que tiene $x = z = 0$ es la del $(0 : 1 : 0)$, por lo tanto, las curvas elípticas tienen un solo punto en el infinito el cual se denota por $\boldsymbol{\mathcal{O} = (0 : 1 : 0)}$. Así se llega a que una curva elíptica definida sobre un cuerpo es

$$E(\mathbb{K}) \ = \ \{(x : y : 1) \in \mathbb{P}^2(\mathbb{K}) \ / \ y^2 = x^3 + ax + b\} \ \cup \ \{\mathcal{O}\}$$

Estos objetos geométricos tienen la particularidad de que sus puntos forman un grupo abeliano, y de acuerdo a [vzG15, p. 207] las operaciones en este grupo se pueden llevar a cabo de manera eficiente, con el agregado de que en aplicaciones criptográficas permiten emplear claves con un tamaño reducido en comparación a otros grupos como $\mathbb{Z}_p^\times$ (lo cual es especialmente útil en contextos de poca memoria o ancho de banda).

[5] Esta es una condición para que la curva sea *no singular*, es decir, que la recta tangente en todos sus puntos esté bien definida.

La operación asociada a $E(\mathbb{K})$ para que ambos formen una estructura de grupo es aditiva (entonces será representada con el símbolo $+$), y se define así para $P, Q \in E(\mathbb{K})$:

- El opuesto de un punto afín corresponde a su simetría axial respecto al eje x, o sea, si $P = (x, y) \Rightarrow -P = (x, -y)$. Para el punto en el infinito se establece que $-\mathcal{O} = \mathcal{O}$. Un ejemplo se provee en la Figura 2.1.
- Si se tienen dos puntos P y Q, el concepto genérico para hallar $P + Q$ es considerar la recta que pasa por ambos y acto seguido tomar el opuesto del punto de corte R de dicha recta con E [6]. O sea, se define $P + Q = -R$; ver un ejemplo en la Figura 2.2. Aquí se distinguen tres casos particulares:
  - Si $Q = P$ entonces la recta que pasa por "ambos" es la recta tangente en P, y a este caso se le llama *duplicación* de P. Un ejemplo se muestra en la Figura 2.3.
  - Si $Q = \mathcal{O}$ se considera que la recta que conecta a ambos es la línea vertical que pasa por P. Luego, esta corta a E en $-P$, con lo cual $P + \mathcal{O} = -(-P)$. Es decir, se define $P + \mathcal{O} = P$.
  - Si $Q = -P$, de vuelta la recta que une a ambos es una vertical cuyo "punto de corte" con E es $\mathcal{O}$. Por lo tanto, $P + (-P) = -\mathcal{O}$, esto es, se establece que $P - P = \mathcal{O}$.

Las tres figuras que siguen ilustran la operación de suma en el caso ordinario de $\mathbb{K} = \mathbb{R}$:

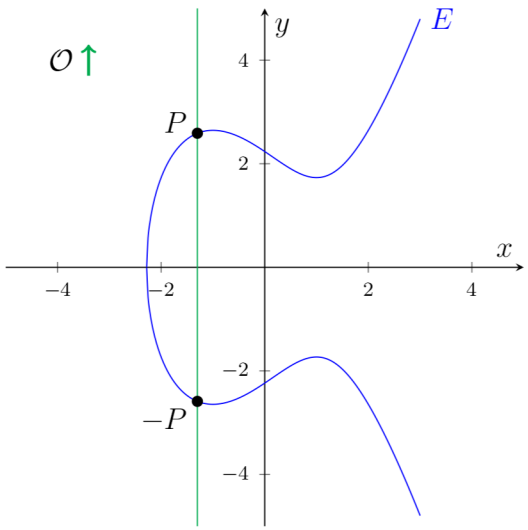

Figura 2.1: Interpretación geométrica del **opuesto** de un punto sobre una curva elíptica

[6] Observar que R está bien definido gracias al Teorema de Bézout: $E(\mathbb{K})$ es una curva proyectiva de grado 3 y la recta (incluso en su versión proyectiva) tiene grado 1, entonces ambas se intersectan en $3 \times 1 = 3$ puntos, dos de los cuales son P y Q.

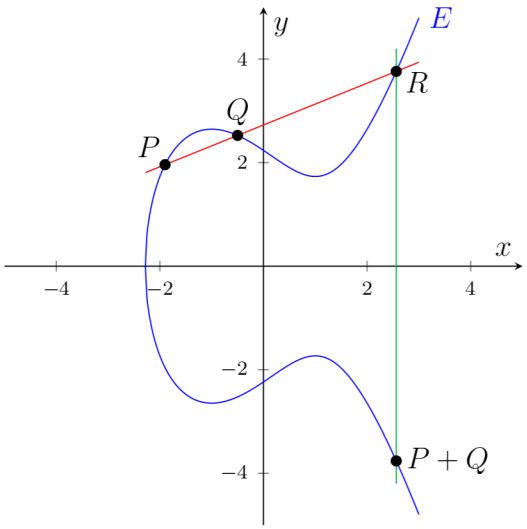

Figura 2.2: Interpretación geométrica de **sumar** dos puntos sobre una curva elíptica

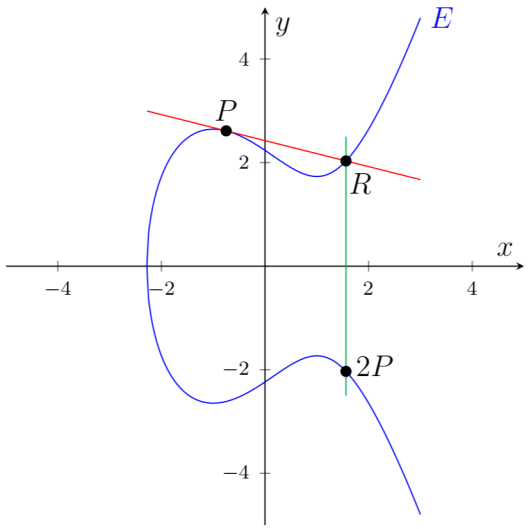

Figura 2.3: Interpretación geométrica de **duplicar** un punto sobre una curva elíptica

De esta manera uno puede demostrar que la curva elíptica $E(\mathbb{K})$ con dicha operación de suma cumple los axiomas de grupo conmutativo con $\mathcal{O}$ como neutro (siendo la propiedad asociativa la más laboriosa de demostrar). Como es estándar en los grupos aditivos, sumar $k \in \mathbb{N}$ veces un punto $P \in E(\mathbb{K})$ con sí mismo se representa con la notación kP, definiéndose $0P = \mathcal{O}$ y $-kP = -(kP)$.

A continuación se muestran las fórmulas explícitas para el cálculo de la suma de puntos afines sobre una curva elíptica $E(\mathbb{K})$ cuya ecuación es $y^2 = x^3 + ax + b$. Por simplicidad, para un elemento uv^{-1} del cuerpo $\mathbb{K}$ se empleará la notación de cociente $\frac{u}{v}$.

Sean $P = (x_1, y_1)$ y $Q = (x_2, y_2)$ tales que $x_1 \neq x_2$ en $\mathbb{K}$. Luego, considerando la ecuación de la recta que pasa por dichos puntos, uno puede trabajar con tal ecuación y la que define a la curva para deducir que $P + Q = (x_3, y_3)$ donde

$$x_3 = \left(\frac{y_2 - y_1}{x_2 - x_1}\right)^2 - x_1 - x_2 \qquad \text{y} \qquad y_3 = \left(\frac{y_2 - y_1}{x_2 - x_1}\right)(x_1 - x_3) - y_1$$

y si $x_1 = x_2$ en $\mathbb{K}$ se tiene $P + Q = \mathcal{O}$ (pues sería $Q = -P$).

Si $P = (x_1, y_1)$ con $y_1 \neq 0$, similarmente se puede inferir que $2P = (x_3, y_3)$ con

$$x_3 = \left(\frac{3\,{x_1}^2 + a}{2\,y_1}\right)^2 - 2\,x_1 \qquad \text{y} \qquad y_3 = \left(\frac{3\,{x_1}^2 + a}{2\,y_1}\right)(x_1 - x_3) - y_1$$

y si $y_1 = 0$ se tiene $2P = \mathcal{O}$ (pues la recta tangente en P sería vertical).

Finalmente se concluye la presente subsección con un resultado importante relacionado con el tamaño de la curva elíptica cuando $\mathbb{K}$ es finito.

Teorema 2.34. [Kob94, p. 174] ***Cota de Hasse:*** *Si N es la cantidad de puntos de una curva elíptica definida sobre un cuerpo finito de q elementos, entonces $|N-(q+1)| \leq 2\sqrt{q}$.*

Concretamente, la cantidad de puntos de una curva elíptica definida sobre un cuerpo finito puede calcularse de manera eficiente (polinomial en el tamaño del cuerpo finito), empleando el ***algoritmo de Schoof*** (cf. [vzG15, pp. 227-229]).

2.1.4. Probabilidad

Definición 2.35. Dado un conjunto $\Omega \neq \emptyset$, un determinado conjunto $\mathcal{B} \subseteq 2^{\Omega}$ no vacío es una ***σ-álgebra*** sobre Ω si cumple las siguientes propiedades:

1) **Contiene a todo el espacio:** $\Omega \in \mathcal{B}$
2) **Cerrado por complementos:** $\forall\, A \in \mathcal{B} \,:\, A^{\complement} \in \mathcal{B}$
3) **Cerrado por uniones numerables:** Para cualquier colección numerable de subconjuntos $\{A_i\}_{i\geq 0} \subseteq \mathcal{B}$, se tiene que $\big(\bigcup_{i=0}^{\infty} A_i\big) \in \mathcal{B}$

El concepto anterior, que proviene de un área de las matemáticas llamada *teoría de la medida*, es la base para construir la teoría de la probabilidad.

Definición 2.36. El ***espacio muestral*** de un experimento es un conjunto $\Omega \neq \emptyset$ compuesto por todas las posibles salidas del experimento. Establecida una σ-álgebra $\mathcal{B}$ sobre Ω, un ***evento*** de dicho espacio es cualquier subconjunto $A \in \mathcal{B}$.

Luego:

Definición 2.37. Un ***espacio de probabilidad*** es una tripleta $(\Omega, \mathcal{B}, \mathcal{P})$ donde Ω es el espacio muestral, $\mathcal{B}$ es una σ-álgebra sobre Ω, y se cuenta con una ***función de probabilidad*** $\mathcal{P} : \mathcal{B} \to [0, 1]$ que satisface lo siguiente:

1) $\mathcal{P}(\Omega) = 1$
2) Para cualquier colección numerable de eventos disjuntos dos a dos $\{A_i\}_{i\geq 0} \subseteq \mathcal{B}$, se tiene que $\mathcal{P}\big(\bigcup_{i=0}^{\infty} A_i\big) = \sum_{i=0}^{\infty} \mathcal{P}(A_i)$

Inmediatamente de lo anterior se deducen estas propiedades:

Proposición 2.38. [Res14, p. 30] *Sea* $(\Omega, \mathcal{B}, \mathcal{P})$ *un espacio de probabilidad. Entonces:*

- ***Prob. de que no suceda un evento:*** $\forall\, A \in \mathcal{B} \,:\, \mathcal{P}\big(A^{\complement}\big) = 1 - \mathcal{P}(A)$
- ***Prob. del evento vacío:*** $\mathcal{P}(\emptyset) = 0$
- ***Prob. de la unión de dos eventos:*** $\forall\, A, B \in \mathcal{B} \,:\, \mathcal{P}(A \cup B) = \mathcal{P}(A) + \mathcal{P}(B) - \mathcal{P}(A \cap B)$

En lo que concierne a este trabajo será suficiente contar con el ***modelo finito*** de [Res14, p. 41] donde: $\Omega = \{\omega_1, \ldots, \omega_n\},\ \mathcal{B} = 2^{\Omega}$ y $\mathcal{P}(A) = \sum\limits_{\omega_i \in A} p_i$, siendo p_i un número no negativo asociado a cada $\omega_i \in \Omega$ que entre todos verifican $\sum_{i=1}^{n} p_i = 1$.

Si se establece $p_i = \frac{1}{n}$ para todo $\omega_i \in \Omega$, es claro que se cumple la condición anterior sobre todos los p_i. En este caso se dice que el espacio muestral es ***equiprobable***, sus elementos se denominan "casos posibles" y dado un evento A del mismo, a los $\omega_i \in A$ se les llama "casos favorables". Luego, $\mathcal{P}(A) = \sum\limits_{\omega_i \in A} \frac{1}{n} = \frac{1}{n}|A|$, y en consecuencia la función de probabilidad queda definida como $\mathcal{P}(A) = \frac{|A|}{|\Omega|}$, más conocida como "casos favorables sobre casos posibles".

Si S es un conjunto finito, se puede considerar el experimento de sortear (elegir aleatoriamente) un elemento x de S. De esta forma el propio S configura un espacio muestral, y considerando el modelo finito anterior puede decirse que –cuando se lleve adelante el experimento– "x se elige de manera uniforme del conjunto S", acción que se representa con la notación $x \xleftarrow{\boxdot} S$.

Finalmente, cuando la ocurrencia de un evento no condicione de ninguna manera la ocurrencia de otro, se hará referencia al siguiente concepto:

Definición 2.39. Dado un espacio de probabilidad $(\Omega, \mathcal{B}, \mathcal{P})$, dos eventos $A,\ B \in \mathcal{B}$ son ***independientes*** si $\mathcal{P}(A \cap B) = \mathcal{P}(A)\,\mathcal{P}(B)$. En el caso de $k > 2$ eventos, la independencia se da si al considerar todas las intersecciones posibles de los k eventos, en todos los casos sucede que la probabilidad de la intersección coincide con el producto de las probabilidades de ocurrencia de los eventos involucrados en la misma [7].

En particular, lo anterior implica que para cualesquiera k eventos independientes, a saber $A_1, \ldots, A_k \in \mathcal{B}$, siempre se verificará que la probabilidad de ocurrencia simultánea de todos ellos es $\mathcal{P}\big(\bigcap_{i=1}^{k} A_i\big) = \prod_{i=1}^{k} \mathcal{P}(A_i)$.

[7] Por ejemplo, si $k = 3$ obviamente hay que chequear $\mathcal{P}(A \cap B \cap C) = \mathcal{P}(A)\,\mathcal{P}(B)\,\mathcal{P}(C)$, pero también hay que constatar que $\mathcal{P}(A \cap B) = \mathcal{P}(A)\,\mathcal{P}(B),\ \mathcal{P}(A \cap C) = \mathcal{P}(A)\,\mathcal{P}(C)$ y $\mathcal{P}(B \cap C) = \mathcal{P}(B)\,\mathcal{P}(C)$. La definición aquí presentada es una adaptación de la que se encuentra en [PM08].

2.2. Fundamentos computacionales

En general todo lo expuesto en esta subsección se basa en los libros [BB97] y [vzG15]. Para las proposiciones, lemas y teoremas se detalla en qué páginas el lector podrá encontrarlos con más información al respecto.

2.2.1. Algoritmia

En este trabajo se especifican los ***algoritmos*** o ***programas*** de acuerdo a un lenguaje *imperativo* [8] en el que –al igual que otros de la misma índole– se manejan los conceptos de *variable* y su estado, *asignación* (representado con la notación $\leftarrow$) y bloques de control del tipo IF y WHILE capaces de evaluar sentencias matemáticas que involucren a las variables, así como los del tipo FOR. Se considera también la instrucción BREAK que interrumpe el flujo del bloque de control más cercano para así escapar de este. Además se tiene la *asignación aleatoria* (representada con $\xleftarrow{\boxdot}$) que consiste en asignar a una variable un elemento extraído de manera uniforme de un conjunto. Todos los algoritmos tendrán una etiqueta de Entrada en la que se declaran variables (y su dominio) cuyo estado se asume definido antes de que el mismo ejecute su primera instrucción, así como una etiqueta de Salida en la que se especifica qué puede decir el algoritmo cuando este termina en relación a sus variables de entrada. En este lenguaje de programación un algoritmo termina tras ejecutar una instrucción que comienza con la palabra RETURN. Por último, será asumido que se cuenta con una biblioteca de funciones matemáticas para realizar operaciones aritméticas en las que pueden intervenir las variables.

Como se verá en las secciones posteriores, para resolver los problemas abordados en esta tesis existen múltiples algoritmos, donde asimismo es de interés poder hacer comparaciones objetivas entre ellos. Para lograr esto se necesita cuantificar de alguna manera el *costo* de ejecutar un algoritmo en su totalidad, entonces lo primero que hay que hacer es definir la unidad de costo más pequeña. Citando textualmente a [BB97]:

Definición 2.40. Una ***operación elemental*** es aquella cuyo tiempo de ejecución se puede acotar superiormente por una constante que solamente dependerá de la implementación particular usada.

Luego, estas operaciones se considerarán con costo *unitario* cuando sea analizada

[8] Esto es, se utiliza una notación que consiste en una lista numerada de *instrucciones* a seguir en orden. Cuando se completa la tarea que se especifica en una instrucción, se dice que la misma fue *ejecutada*.

la complejidad temporal de un algoritmo. [9]

Por otro lado, los problemas computacionales tienen cierto *dominio de definición*, el cual trasladado a los algoritmos que los resuelven viene dado –naturalmente– por el dominio de sus variables de entrada. Así pues, para realizar el análisis del tiempo de ejecución de un algoritmo, la idea es cuantificar la cantidad de operaciones elementales que se llevan a cabo, en función del *tamaño* de los ejemplares plausibles de ocupar el estado de **al menos una** de sus variables de entrada. Formalmente el tamaño de un ejemplar se mide en la cantidad de bits necesarios para representarlo, sin embargo, también se admite considerar como "tamaño" cualquier número que describa la cantidad de "componentes" de cierto ejemplar.

Como frecuentemente sucede, en los problemas tratados en este proyecto existen infinitos ejemplares que podrían llegar a ser la entrada de un algoritmo. Por tal motivo, en la práctica uno se encuentra con que no todas las posibles entradas de un algoritmo, a pesar de tener el mismo tamaño, consumen el mismo tiempo de ejecución. Esto conduce a que el análisis de costo para cada tamaño de entrada se centre en el *peor caso*, que consiste en aquellos ejemplares para los cuales el algoritmo realiza la mayor cantidad de operaciones elementales, o en otras palabras, los casos en los que el algoritmo tarda más tiempo. En este sentido, si –en relación a un algoritmo– se dice algo como "se ejecuta en un tiempo *del orden* de $t(k)$", el mismo debe ser capaz de resolver **todas** las instancias de tamaño k en **a lo sumo** $c \,.\, t(k)$ operaciones elementales (siendo c una constante real positiva).

La noción anterior del orden del tiempo de ejecución está asociada a un concepto llamado ***notación asintótica***:

Definición 2.41. Sean dos funciones t, $f : \mathbb{N} \to (\mathbb{R}^+ \cup \{0\})$. Se dice que f es una ***cota superior asintótica*** de t si $\exists\, c \in \mathbb{R}^+,\ \exists\, k_0 \in \mathbb{N}\ /\ \forall\, k \geq k_0\ :\ t(k) \leq c \,.\, f(k)$. El conjunto de todas las funciones como t que tienen a f por cota superior asintótica se denota por $O\big(f(k)\big)$.

[9] Por ejemplo, hallar el resto de dividir un número natural entre 2 podría considerarse una operación elemental, ya que basta con devolver el último bit de la representación binaria del número. También podría considerarse una operación elemental la misma tarea pero siendo otro el valor del divisor, si uno sabe de antemano que los números involucrados serán suficientemente pequeños como para poder afirmar que la implementación de dicha tarea siempre puede efectuarse en a lo sumo t_r segundos. Sin embargo, si los números en cuestión son considerablemente grandes, la misma operación que hasta hace un momento podía considerarse elemental, ahora sería más sensato mirarla detalladamente y ver qué otras operaciones *más* elementales esta requiere.

Si $t \in O\big(f(k)\big)$, además de la que el lector pueda estar subvocalizando, existen varias maneras de referirse a esto, a saber: «*t es del orden de* $f(k)$», «*t es orden* $f(k)$», o simplemente «*t es O* $f(k)$». Entonces si $t(k)$ representa el tiempo de ejecución de un algoritmo, las expresiones anteriores también se pueden usar cambiando "t" por "el tiempo de ejecución del algoritmo".

Proposición 2.42. [BB97, p. 94] ***Regla del máximo:*** *Si se tienen dos funciones* $f, g : \mathbb{N} \to (\mathbb{R}^+ \cup \{0\})$*, entonces* $O\big(f(k) + g(k)\big) = O\big(\text{máx}\,\big(f(k), g(k)\big)\big)$.

Más aún, esta regla se puede generalizar para una suma en la cual intervengan cualquier cantidad finita de funciones: el orden de la suma siempre será igual al orden de la función maximal. En otras palabras, cuando la función que describe el tiempo de ejecución de un algoritmo es una suma de otras funciones, se tiene que el término de mayor orden *domina* al resto y es el que prevalece en la notación asintótica.

Proposición 2.43. [BB97, p. 96] *La relación "ser del orden de" es transitiva.*

Por lo tanto, si se tiene algo como $t(k) \in O\Big(\frac{\lceil \log(k) \rceil}{7} + 21\Big)$ y $\lceil \log(k) \rceil \in O\big(\log(k)\big)$, por la regla del máximo, por la propia definición de cota superior asintótica y por transitividad, la notación asintótica se simplifica a $t(k) \in O\big(\log(k)\big)$.

Definición 2.44. Sean dos funciones t, $f : \mathbb{N} \to (\mathbb{R}^+ \cup \{0\})$. Se dice que f es una ***cota inferior asintótica*** de t si $\exists\, c \in \mathbb{R}^+,\ \exists\, k_0 \in \mathbb{N}\ /\ \forall\, k \geq k_0\ :\ c\,.\,f(k) \leq t(k)$. El conjunto de todas las funciones como t que tienen a f por cota inferior asintótica se denota por $\Omega\big(f(k)\big)$.

Para este concepto análogo al del orden no se emplearán tantas alternativas para referirse al hecho de que $t \in \Omega\big(f(k)\big)$, sino que simplemente se dirá que «*t es Omega* $f(k)$». De la misma manera que antes se extiende la expresión al tiempo de ejecución de un algoritmo.

2.2.2. Costo de algunas rutinas

Como el lector podrá observar a lo largo de este trabajo, existen ciertas tareas que son recurrentes al resolver diferentes problemas computacionales, pero algunas de estas no son lo suficientemente elementales como para que en la mayoría de los casos uno pueda asumir que tienen costo $O(1)$. Cuando son implementadas mediante un programa, dichas tareas se conocen como *rutinas*. Y a efectos de realizar el análisis del tiempo de

ejecución de un algoritmo que emplee alguna de ellas, resulta preciso conocer cuál es la complejidad de cada una de estas tareas.

Para esta subsección se consideran las operaciones de la aritmética básica (suma, resta, multiplicación y división entera) como elementales.

En primer lugar considerar la tarea de hallar el máximo común divisor entre dos naturales. Sorprendentemente, por más grandes que sean los números considerados, calcular su máximo común divisor resulta sencillo en términos computacionales. Esto es gracias al ***algoritmo de Euclides***, que no requiere hallar los divisores de los números en cuestión. Suponer que se desea hallar el $\gcd(a, b)$ para $a \geq b > 0$. Por el **Teorema de la división entera**, $\exists\, q,\, r \in \mathbb{Z} \;/\; a = bq + r$ con $0 \leq r < b$, y por la **Proposición 2.3** $\gcd(b, a) = \gcd(b, a - bq)$. Luego, como $\gcd(a, b) = \gcd(b, a)$ y $r = a - bq$, se tiene que $\gcd(a, b) = \gcd(b, r)$. Esta última ecuación implica que el desafío de hallar el máximo común divisor se puede trasladar hacia el resto de dividir a entre b. Más aún, si $r \neq 0$, entonces como $b > r$ ahora b podría tomar el rol de a y r el rol de b, para así trasladar el problema a un residuo aún más pequeño, *i. e.*, $\gcd(b, r) = \gcd\big(r,\, b \pmod r\big)$. Si se procede de la misma manera sucesivamente, dado que los restos forman una secuencia decreciente de enteros no negativos, es seguro que en algún momento se obtiene un residuo nulo y allí el algoritmo termina, pues $\gcd(\tilde{r}, 0) = \tilde{r}$ [10].

Evidentemente, la cantidad de operaciones elementales que realiza el algoritmo de Euclides queda determinada por la cantidad de veces que se aplica el paso de calcular un nuevo resto, o dicho de otra forma, por cuántas veces el mismo ejecuta el bucle donde efectúa una división entera mientras **no** se obtenga un resto nulo. Como se menciona en [Sha94, p. 410], Émile Léger, en 1837, habría sido el primero en reconocer que el peor caso para el algoritmo de Euclides es cuando a y b son *números de Fibonacci* consecutivos [11]. Sin embargo, "Léger no dio una prueba rigurosa de sus afirmaciones", en comparación a –por ejemplo– la prueba disponible en [Knu98, pp. 356-360]. De acuerdo a [Sha94, p. 413], Pierre J. É. Finck desarrolló en 1841 el primer análisis del algoritmo de Euclides y probó que a lo sumo se requieren $2\log_2(b) + 1$ pasos en los que se hace una división entera, lo que en notación moderna se expresaría como $O\big(\log(b)\big)$. Tres años después, Gabriel Lamé mejoró la cota a cinco veces la cantidad de dígitos decimales de b [Sha94, p. 403]. Una cota aún más precisa es probada por Knuth en [Knu98, p. 360]: si $0 \leq b < n$, entonces la cantidad de divisiones efectuadas es a lo sumo $\big\lfloor \log_\phi\big((3 - \phi)n\big)\big\rfloor$ (donde $\phi = \frac{1+\sqrt{5}}{2}$, el *número áureo*). Por lo tanto, el tiempo de ejecución de este algoritmo

[10] Es decir, el $\gcd(a, b)$ termina siendo $\tilde{r}$, el último resto no nulo obtenido en el proceso.

[11] Los números de Fibonacci son $F_0 = 0$, $F_1 = 1$ y los demás se calculan como $F_n = F_{n-1} + F_{n-2}$.

tiene una cota superior asintótica *lineal* según el tamaño de la menor entrada.

Proposición 2.45. [Knu98, p. 360] *Sean a y b dos enteros positivos y n un natural de k bits, tales que $b \leq a < n$. El cómputo del* $\gcd(a, b)$ *se puede efectuar en $O(k)$ operaciones elementales en el peor caso.*

Con una ligera modificación de las operaciones elementales efectuadas dentro del bucle "mientras no se obtenga un residuo nulo", se tiene un nuevo proceso cuyo tiempo de ejecución mantiene la misma cota superior asintótica de la proposición anterior. Este se denomina ***algoritmo de Euclides extendido*** y como tal, además de computar el $\gcd(a, b)$, permite hallar un par de valores x, $y \in \mathbb{Z}$ tales que $ax + by = \gcd(a, b)$. Esta ecuación se conoce como *identidad de Bézout*, y a los multiplicadores de a y b se los llama *coeficientes de Bézout*.

Lo siguiente a considerar es la tarea de realizar una exponenciación modular, es decir, calcular x^n en $\mathbb{Z}_m$. Como se explica en [vzG15, pp. 770-771], el método más simple para resolver esta tarea eficientemente es el llamado ***exponenciación por cuadrados***, *binaria* o también *rápida* (en inglés, *repeated squaring*). Si la representación binaria de n consta de k bits $b_{k-1} \cdots b_0$, entonces $n = \sum_{i=0}^{k-1} b_i 2^i$. Luego, $x^n = \prod_{i=0}^{k-1} \left(x^{2^i}\right)^{b_i}$. Observar que en la fórmula anterior el bit b_i indica simplemente si el factor x^{2^i} interviene o no en el producto de lo que se pretende calcular, por ende, la cantidad de multiplicaciones queda determinada por la cantidad de unos en la representación binaria de n [12]. Más aún, notar que $\left(x^{2^{i-1}}\right)^2 = x^{2^i}$, con lo cual partiendo desde x ($i = 0$) y elevando al cuadrado el valor del paso anterior ($i \geq 1$), se obtienen todos los factores necesarios para hallar x^n. Por lo tanto, el método de exponenciación modular por cuadrados requiere efectuar $k - 1$ cuadrados con una reducción módulo m en cada vez, y a lo sumo $k - 1$ productos con reducción módulo m en cada uno. Así se concluye la siguiente:

Proposición 2.46. [vzG15, p. 771] *Sean n un natural de k bits, m un entero positivo y $x \in \mathbb{Z}$. El cálculo de $x^n \pmod m$ se puede efectuar en $O(k)$ operaciones elementales en el peor caso.*

2.2.3. Criptosistema RSA

En esta parte se presentan los aspectos formales del protocolo RSA, inventado por Rivest, Shamir y Adleman en 1977 y publicado en [RSA78] a principios del año siguiente.

[12] Esto último implica que el peor caso se da cuando $n = 2^k - 1$, o sea, cuando los k bits son todos 1.

En términos históricos fue la primera materialización del paradigma asimétrico concebido por Diffie y Hellman en 1976, y al día de hoy continúa siendo uno de los protocolos más usados en las comunicaciones digitales. De todas formas, es importante destacar que tanto RSA como el resto de los sistemas asimétricos, en la práctica se emplean para autenticar y cifrar la transmisión de la clave de un sistema simétrico (*e. g.*, AES [13]), debido a que –con las implementaciones actuales– los criptosistemas simétricos gozan de una velocidad de cómputo considerablemente mayor.

Dado que RSA forma parte de la familia de protocolos de criptografía **asimétrica**, hay cuatro conceptos que el mismo debe especificar precisamente:

1) **Clave privada:** es el elemento secreto del criptosistema, en el sentido de que **solo debe ser conocido por la entidad que lo genera**, puesto que es la llave del descifrado de los mensajes recibidos.

2) **Clave pública:** es un elemento asociado a la entidad propietaria de la clave privada, pero a diferencia de la misma, esta clave **debe darse a conocer a cualquier entidad** que desee enviar información, porque se trata de la llave para cifrar sus mensajes.

3) **Función de cifrado:** [14] es una función matemática que toma como entrada un mensaje *plano* (inteligible) codificado numéricamente, y aplicando la clave **pública** devuelve otro elemento que representa al mensaje encriptado.

4) **Función de descifrado:** [15] es una función matemática que recibe la representación de un mensaje encriptado, y empleando la clave **privada** retorna el número que codifica al mensaje plano. En otras palabras, es la función inversa de la de cifrado.

Por motivos obvios de seguridad, adicionalmente se exige que para la función de encriptado sea **difícil de computar su inversa** valiéndose únicamente de la información pública, tarea que –como se detalla más abajo– en el caso de RSA está teóricamente relacionada con el problema de factorización de enteros.

Para generar la clave pública, en RSA primero se deben elegir **dos números primos** p y q **grandes** y **distantes** a efectos de calcular su producto $\boldsymbol{n = pq}$. A la fecha del presente trabajo, el *Instituto Nacional de Estándares y Tecnología* (NIST, por su sigla en inglés) del Departamento de Comercio de los Estados Unidos de América

[13] El *estándar de encriptación avanzado*, conocido por su sigla en inglés "AES", es el estándar a nivel mundial de criptografía **simétrica**.

[14] También llamada *de encriptado* o bien *de encriptación*.

[15] También conocida como *de desencriptado*.

especifica en su reporte más reciente de recomendaciones sobre gestión de claves [Bar20], que el menor tamaño de n considerado seguro es de 2048 bits [16]. Por ende, esto implica que hoy en día al decir que p y q han de ser "grandes", uno se refiere a que ambos sean de al menos 1024 bits. En otras palabras, actualmente el rango para elegir los números primos iría del 2^{1023} al $2^{1024}-1$. En cuanto a qué se espera con que p y q sean "distantes", lo mínimo sería que resista el clásico ataque llamado *método de factorización de Fermat*, el cual se basa en escribir n como la diferencia de dos cuadrados: si $n = t^2 - s^2$ con $t = \frac{q+p}{2}$ y $s = \frac{q-p}{2}$, entonces es posible obtener la factorización de n como $pq = (t-s)(t+s)$. Luego, iterando en s hasta que $\sqrt{n+s^2}$ dé un entero, en teoría uno podría conseguir el valor de t. No obstante, visto que el intervalo recomendado contiene $(2^{1024}-1) - 2^{1023} + 1 = 2^{1023}$ enteros, este es suficientemente extenso como para que p y q puedan escogerse con una distancia inviable de ser recorrida de manera lineal.

Una vez que se tiene decidido el par de números primos, se procede calculando –de acuerdo a la **Proposición 2.12**– el valor de $\varphi(n) = (p-1)(q-1)$, para así terminar eligiendo un número entero e entre 2 y $\varphi(n) - 2$ coprimo con $\varphi(n)$. Finalmente, el protocolo RSA establece que **la clave pública es la dupla (e, n)**.

Llegado a este punto queda clara la importancia que posee para la "puesta en escena" del criptosistema RSA, tener la capacidad de buscar números primos y asimismo poder hacerlo de manera eficiente. He aquí la motivación de la Sección 3.

Por otro lado, RSA establece que **la clave privada es la dupla (d, n)**, siendo d el elemento de $\mathbb{Z}_{\varphi(n)}$ que satisface $de \equiv 1 \pmod{\varphi(n)}$. Notar que gracias a que e se toma con $\gcd\big(e, \varphi(n)\big) = 1$, por el **Lema 2.18** la existencia y unicidad de tal d está garantida. De hecho, la congruencia anterior es equivalente a la ecuación $de - 1 = \varphi(n)\, k$ con $k \in \mathbb{Z}$, donde a su vez esta se puede reescribir como $ed + \varphi(n)\, \tilde{k} = \gcd\big(e, \varphi(n)\big)$ con $\tilde{k} = -k$. Por lo tanto, en esta última se identifican d y $\tilde{k}$ con el rol de coeficientes de Bézout indeterminados, que como se mencionó al final de la subsección anterior, se pueden hallar de manera eficiente con el algoritmo de Euclides extendido.

Finalmente, en el protocolo RSA **la función de cifrado es $E(x) \equiv x^e \pmod{n}$** y la **función de descifrado es $D(y) \equiv y^d \pmod{n}$**. Observar que nuevamente estas operaciones se pueden computar eficientemente, en este caso empleando el método de exponenciación por cuadrados introducido previamente. En cuanto a la correctitud del

[16] Si bien todavía no se han factorizado números de 1024 bits con esta forma, el ejemplo más reciente fue dado por [BGG+20] y se trata del número "RSA-250" que consta de 829 bits. Por lo tanto, el NIST ya considera deprecados todos los valores de n de 1024 bits.

descifrado, el asunto se puede tratar según si x es o no coprimo con n.

Por un lado, se tiene que $D\big(E(x)\big) \equiv x^{de} \equiv x^{\varphi(n)\,k+1} \equiv \big(x^{\varphi(n)}\big)^k\, x \pmod{n}$. Luego, cuando x es coprimo con n, por el **Teorema de Euler** $x^{\varphi(n)} \equiv 1 \pmod{n}$, con lo cual $\big(x^{\varphi(n)}\big)^k \equiv 1 \pmod{n}$. Entonces $D\big(E(x)\big) \equiv x \pmod{n}$.

Por otro lado, si x no es coprimo con n, la descomposición en factores primos de este último determina que $\gcd(x, n) \in \{p, q, n\}$. Si $n \,|\, x \;\Rightarrow\; x \equiv 0 \pmod{n}$, y es claro que $D\big(E(0)\big) \equiv 0 \pmod{n}$. Si $n \nmid x$, la condición sobre el $\gcd(x, n)$ implica que $p \,|\, x \;\wedge\; q \nmid x$, o viceversa. Suponer sin pérdida de generalidad que vale el caso donde $p \,|\, x$, entonces $x \equiv 0 \pmod{p}$. Luego, vale que $D\big(E(x)\big) \equiv x \pmod{p}$. Observar que también $D\big(E(x)\big) \equiv x^{de} \equiv x^{\varphi(n)\,k+1} \equiv x^{\varphi(p)\,\varphi(q)\,k+1} \equiv \big(x^{\varphi(q)}\big)^{\varphi(p)\,k} x \pmod{q}$, y aquí ocurre que $\big(x^{\varphi(q)}\big)^{\varphi(p)\,k} \equiv 1 \pmod{q}$ porque como el factor primo $q \nmid x$, por el **Teorema de Euler** $x^{\varphi(q)} \equiv 1 \pmod{q}$. Entonces $D\big(E(x)\big) \equiv x \pmod{q}$. Así se ve que la congruencia que avala la correctitud del descifrado, se cumple tanto en módulo p como en módulo q. Luego, por el **Teorema chino del resto**, como $\gcd(p, q) = 1$ esta también se cumple en módulo pq, es decir, $D\big(E(x)\big) \equiv x \pmod{n}$.

Por lo tanto, la forma más directa de computar la función inversa de la de cifrado a partir de la clave pública (e, n) recae en hallar los factores primos de n (p y q), calcular $\varphi(n)$ como la simple multiplicación de $(p-1) \times (q-1)$ y luego resolver la congruencia $de \equiv 1 \pmod{\varphi(n)}$ para d (vía el eficiente algoritmo de Euclides extendido). Luego, si el desafío de factorización se torna suficientemente difícil mientras la información protegida tiene validez, el criptosistema RSA resulta seguro. Por eso, conocer el estado del arte de este problema que sustenta la seguridad de uno de los protocolos más célebres, es la motivación de la Sección 4.

3. En búsqueda de los números primos

3.1. Aproximación naíf

Situándose en lo más elemental del asunto –la definición de número primo– el algoritmo más obvio para determinar si un número $n > 1$ es primo consiste en comprobar su divisibilidad con respecto a cada uno de los naturales mayores que la unidad que son anteriores a n. Este método en realidad constituiría una variante la ***criba de Eratóstenes*** (cuyo propósito es conseguir *todos* los números primos hasta n inclusive), diseñada hace algo más de 2.000 años por el matemático griego Eratóstenes de Cirene [vzG15, p. 118].

Observar que si n es un primo impar, entonces en la recorrida antedicha alcanza con pasar solamente por los números impares (dicho de otro modo, si n fuera divisible por un número par, entonces el propio n sería par). Por ende, con algo de astucia se puede reducir a la mitad el tamaño del rango de búsqueda de divisores.

Más aún, con un poco de perspicacia uno puede reducir el rango de búsqueda de divisores hasta la raíz cuadrada de n:

Proposición 3.1. *Para todo $n \in \mathbb{N}$: si n es compuesto, entonces existe $d > 1$ divisor de n tal que $d \leq \sqrt{n}$.*

Demostración:

Si n es compuesto, existen $x, y \in \mathbb{N}$ tales que $n = xy$, $1 < x < n$ y $1 < y < n$. Suponer, por el contrario, que para todo $d > 1$ divisor de n sucede que $d > \sqrt{n}$. De esa forma –particularmente– se tiene que $x > \sqrt{n}$ y $y > \sqrt{n}$. Pero entonces sería $xy > n$, cuando en realidad $xy = n$.

Luego, debe existir $d > 1$ divisor de n tal que $d \leq \sqrt{n}$.

■

Por lo tanto, el algoritmo *naíf* para decidir si un número es primo, consiste en descartar el caso trivial donde n es par y luego chequear divisibilidad con los números impares hasta $\sqrt{n}$:

Algoritmo 3.1. Test naíf

Entrada: $n > 1$ y natural.

Salida: `PRIMO` o `COMPUESTO`.

```
1  IF  n (mód 2) = 0  THEN
2      IF  n = 2  THEN
3          RETURN PRIMO
4      ENDIF
5      RETURN COMPUESTO
6  ENDIF
7  i ← 3
8  WHILE  i ≤ √n  DO
9      IF  n (mód i) = 0  THEN
10         RETURN COMPUESTO
11     ENDIF
12     i ← i + 2
13 ENDWHILE
14 RETURN PRIMO
```

Teorema 3.2. *Para todo natural $n > 1$: n es primo si y solo si el Algoritmo 3.1 devuelve `PRIMO`.*

Demostración:

Si n es primo, entonces el Algoritmo 3.1 devuelve `PRIMO`

Si $n = 2$, entonces la condición del IF de la línea 1 se verifica dando paso al IF de la línea 2, que también se verifica. Este último habilita entonces la línea 3, donde se devuelve `PRIMO`.

Si $n \neq 2$, entonces n es un primo impar, con lo cual debe ser $n \equiv 1 \pmod 2$. Por ende, el bloque IF que comienza en la línea 1 es omitido, siguiendo el programa en la línea 7. Aquí se inicia un iterador i con el primer número impar (mayor que uno) porque –como ya se había mencionado– para n impar no es necesario testear divisibilidad con números pares. Luego, en la línea 8 se tiene un bucle WHILE que durará mientras el iterador sea menor o igual que $\sqrt{n}$.

Se observa que para los siguientes números primos, $n = 3$, 5 y 7, la condición del WHILE no se verifica cuando $i = 3$, por ende, en esos casos el bucle se omite, continuando

el programa en la línea 14 donde se devuelve `PRIMO`.

Suponer ahora que el primo $n > 7$, de modo que la condición del WHILE se verifica al menos una vez y se habilita la línea 9. En estos casos, la condición del IF en esa línea nunca será verdadera (su validez implicaría que i es un divisor no trivial de n, contradiciendo la hipótesis de que n es primo), por lo tanto, el programa ejecutará solamente la línea 12 mientras permanezca en el bucle, donde el iterador se mueve al siguiente número impar. Evidentemente el avance ascendente del iterador provocará que en algún momento la condición del WHILE deje de cumplirse, por tanto, cuando n es un primo mayor que 7, el programa llegará a la línea 14 donde se retorna `PRIMO`.

Si n es compuesto, entonces el Algoritmo 3.1 devuelve `COMPUESTO`

Si n es un compuesto par, entonces se cumple la condición del IF en la línea 1 pero no la del IF en la línea 2, con lo cual el programa llega a la línea 5, donde se retorna `COMPUESTO`.

Si n es un compuesto impar, entonces –por las mismas razones que en la prueba anterior– el programa llega al WHILE de la línea 8 con el iterador i valiendo 3.

Se observa que el primer número compuesto impar es 9, donde la condición del WHILE se cumple siendo $i = 3$. Por ende, para todo $n \geq 9$ compuesto, el programa entra al menos una vez al bloque WHILE. Por la **Proposición 3.1** (combinada con el hecho de que en estos casos los divisores deben ser impares), deberá existir $i > 1$ divisor impar de n tal que $i \leq \sqrt{n}$. Entonces es un hecho que la condición del IF de la línea 9 se verificará en algún momento **antes** de que iterador desborde la condición de permanencia en el bucle. Cuando eso suceda, el programa ejecutará la línea 10 donde se devuelve `COMPUESTO`.

■

Con lo anterior ha quedado demostrado que el test naíf es un algoritmo determinista y general, incluso muy fácil de describir y entender. Pero como se verá a continuación, lamentablemente su complejidad computacional está lejos de ser eficiente:

Teorema 3.3. *Si $n > 1$ es un natural de k bits, entonces el tiempo de ejecución del Algoritmo 3.1 en el peor caso es $O\left(\sqrt{2}^{k}\right)$.*

Demostración:

Para este análisis se considera que las operaciones elementales son el cálculo de

$n \pmod{i}$, comparación de $=$ o $\leq$, raíz cuadrada, sumas y asignaciones.

Bajo este criterio, es claro que cuando n es 2 o un número compuesto par, el programa termina en las líneas 2 o 5 habiendo hecho una cantidad constante de operaciones elementales. También se ve que cuando $n = 3$, 5 y 7, el algoritmo termina habiendo ejecutado una cantidad constante de tales instrucciones. Más aún, cuando $n \geq 9$ es compuesto se vio que el programa termina antes de salir del WHILE, por lo tanto, el peor caso es cuando $n \geq 9$ y primo, porque ahí el programa ejecuta el bucle y termina en una instrucción posterior. Además si n tiene k bits, entonces el valor máximo posible para n es $2^k - 1$ (que puede ser primo, *e. g.*, con $k = 5$).

De esta forma se deduce que la complejidad en el peor caso depende exclusivamente de la cantidad de veces que se ejecute el bloque WHILE. Dado que este se ejecutará para todo i impar tal que $3 \leq i \leq \sqrt{n}$, la cantidad exacta de veces será $\left\lceil \frac{\lfloor \sqrt{n} \rfloor - 3 + 1}{2} \right\rceil = \left\lceil \frac{\left\lfloor \sqrt{2^k - 1} \right\rfloor - 2}{2} \right\rceil$. Luego, en términos asintóticos, el tiempo de ejecución es $O\left(\sqrt{2^k}\right)$. Y como $\sqrt{2^k} = \sqrt{2}^k$, se concluye que la cantidad de operaciones elementales hechas en el peor caso es $O\left(\sqrt{2}^k\right)$.

■

En síntesis, la aproximación al problema "por definición" es una opción certera y bien sencilla de programar, pero al tener un tiempo de ejecución exponencial se vuelve inviable para números grandes (recordar de la Subsección 2.2.3 que hoy en día se buscan primos de al menos 1024 bits, lo cual daría aproximadamente $\sqrt{2}^{1024} \approx 1.34 \times 10^{154}$ operaciones elementales).

3.2. Nuevos rumbos

Tal como se acaba de ver, encarar el problema por su definición resultaría ingenuo para números de tamaño considerable, por eso, la comunidad matemática se vio obligada a buscar otras rutas que condujeran a la factibilidad práctica del problema. La primera alternativa que surgió para acelerar las cosas fue quitarle generalidad a las entradas de los test, es decir, crear algoritmos para decidir si números de cierta forma son primos; un ejemplo de esto –cuyo origen se remonta a finales de los años 1870– se expondrá en la Subsección 3.9. Sin embargo, el foco de este proyecto en cuanto a los test de primalidad está puesto en los métodos genéricos, dado que el criptosistema RSA no especifica que los números primos a emplear deban pertenecer a determinada categoría.

El siguiente enfoque que se tomó buscando mantener la generalidad a bajo costo fue sacrificar la cualidad determinista de los test, esto es, se introdujo la idea de que el algoritmo podría tomar alguna opción aleatoria en algunas de sus instrucciones. Naturalmente surge la inquietud de qué contrapartidas podría tener introducir eventos aleatorios en la algoritmia, y de acuerdo a [BB97, p. 367] básicamente hay dos: el programa podría dar resultados erróneos (cuando el azar conduce a emitir una sentencia falsa), o bien podría devolver respuestas vacías al respecto [1] (si la elección aleatoria condujo a un camino por el cual nada se puede concluir) pero cuando sí hay un resultado este siempre es correcto. Ejemplos de ambos tipos se desarrollan a lo largo de la presente sección, pero lo que el lector habría de notar en este momento es que –en cualquiera de los casos– un mismo algoritmo probabilístico puede generar **resultados distintos** cuando se aplica dos veces **para la misma entrada**.

Considerar el algoritmo probabilístico más fácil de todos para intentar obtener un número primo: *dado $x \in \mathbb{Z}^+$, elíjase aleatoriamente un número entero dentro del intervalo* $[1, x]$. ¿Qué probabilidad de éxito tendría? Si $\pi(x)$ denota la cantidad de números primos menores o igual a x, entonces su ratio entre los primeros x enteros es $\frac{\pi(x)}{x}$. Esta expresión por sí misma, que de hecho representa la probabilidad buscada, en realidad no aporta demasiada sustancia al asunto puesto que $\pi(x)$ no se está dando en función de x [2]. Luego, y recordando que los números primos deseados siempre son grandes en tamaño (cantidad de bits) y enormes en valor numérico, lo más conveniente (para concluir si el método descripto es adecuado) sería mirar el comportamiento *asintótico* de la proporción. Para concretar esto, el protagonista aquí es el **Teorema de los números primos**, el cual sentencia que $\pi(x)$ es equivalente con $\frac{x}{\log(x)}$. Entonces:

$$\begin{aligned}
\lim_{x\to\infty} \frac{\pi(x)}{x} &= \lim_{x\to\infty} \frac{\pi(x)}{\frac{x}{\log(x)} \cdot \log(x)} \\
&= \lim_{x\to\infty} \frac{\pi(x)}{x/\log(x)} \cdot \frac{1}{\log(x)} \\
&= \lim_{x\to\infty} 1 \cdot \frac{1}{\log(x)} && [\ \textbf{Teorema de los números primos}\ ] \\
&= 0 && \left[\ \lim_{x\to\infty} \log(x) = \infty\ \right]
\end{aligned}$$

Por lo tanto, se ve que **la probabilidad de elegir un número aleatorio hasta x y que este sea primo, tiende a cero** conforme aumenta x. Luego, la probabilidad

[1] Por ejemplo, retornando `NULL`.

[2] Es decir, para calcular la probabilidad de encontrar un número primo en el intervalo $[1, x]$ para un x en concreto, habría que consultar los valores tabulados de $\pi(x)$.

de éxito del algoritmo planteado es escasa, haciendo que el método resulte para nada confiable.

El precedente razonamiento induce la pregunta de cuál tendría que ser, entonces, la componente aleatoria en un test de primalidad para que el mismo resulte útil. Como se verá a lo largo de la presente sección, el caso típico de instrucción aleatoria consistirá simplemente en sortear un número en un rango dado, para luego emplear dicho elemento en comprobaciones sobre la entrada. En la siguiente subsección se estudiará cómo tal tarea puede ser implementada.

Sobre los test que con un número compuesto podrían retornar "es primo" por error, en este trabajo se ha preferido utilizar el término `PSEUDO-PRIMO` como resultado para todas aquellas entradas que logren escurrirse entre los embates del test. Así es que si la salida de un algoritmo es `PSEUDO-PRIMO`, entonces uno tendrá la disyuntiva de ejecutarlo t veces más para ver si se trató de un error, o confiar en que la entrada realmente era un número primo. En la práctica, la confianza buscada siempre será la que otorga llevar la probabilidad de error por debajo de la de una falla material en un equipo ejecutando un algoritmo determinista.

Finalmente, cabe mencionar que el primer test de primalidad general y eficiente fue uno probabilístico publicado por Robert Solovay y Volker Strassen en [SS77], apenas un mes antes de que Rivest, Shamir y Adleman inventaran su famoso sistema de encriptación. No obstante, a pesar de su importancia histórica por haber servido como garante de que el protocolo RSA sería realizable en la práctica, este test cayó en desuso con el advenimiento de otro algoritmo probabilístico también fundado en resultados de la teoría de números: el *test de Miller-Rabin*. Dada esta circunstancia de la realidad, y considerando que el test basado en curvas elípticas de la Subsección 3.7 ya aportaría variedad en cuanto a las distintas subáreas de las matemáticas alcanzadas, se decidió excluir del relevamiento el *test de Solovay-Strassen*.

3.3. ¿Aleatoriedad o Pseudoaleatoriedad?

Anteriormente se mencionó que la tarea de obtener un número aleatorio dentro de un rango dado es de relevancia para los test de primalidad, entonces –y en vista de los objetivos trazados en el proyecto– se ha decidido dedicar la presente subsección a dicha operación recurrente en los algoritmos probabilísticos que se tratan en esta tesis.

Intuitivamente, la cualidad deseada en esta tarea parece contradictoria con que la misma vaya a ser ejecutada por un dispositivo determinista (*e. g.*, una computadora). Sin embargo, sí es posible *recabar* aleatoriedad a partir de distintas fuentes, las cuales según [vzG15, pp. 448-449] se clasifican en dos grandes categorías. Por un lado se tienen las fuentes *basadas en software*, siendo los ejemplos típicos el reloj del sistema, ciertos registros del CPU [3] y señales emitidas por distintos periféricos (*e. g.*, ratón, teclado, adaptadores de red, etc.) que son detectadas por un procedimiento de sondeo. Del otro lado se encuentran las fuentes *basadas en hardware*, como pueden ser mediciones de radiactividad en el ambiente o ruido térmico en diferentes circuitos y dispositivos electrónicos (*e. g.*, capacitores, diodos, osciladores de anillo, etc.).

Desafortunadamente, si se empleara cualquiera de las fuentes anteriores cada vez que se necesita un número aleatorio, la tarea resultaría inconveniente, costosa o incluso potencialmente insegura (de acuerdo a la aplicación). Por lo tanto, el truco utilizado en la práctica consiste en sacrificar "verdadera" aleatoriedad para darle el protagonismo al concepto de ***pseudoaleatoriedad***. Esta noción se relaciona con la intuición planteada al comienzo del párrafo anterior: dado que para un dispositivo determinista es imposible *generar* aleatoriedad, la idea es recabar una pequeña cantidad de aleatoriedad "real" (con las fuentes antes mencionadas) y en base a esos datos efectuar operaciones deterministas que produzcan una gran cantidad de elementos que, en su aplicación práctica, se comportan *como si fueran* aleatorios.

Formalmente hablando, un ***generador pseudoaleatorio*** es un algoritmo determinista que toma como entrada elementos de un conjunto finito X formado a partir una fuente de aleatoriedad, los elementos de su salida conforman otro conjunto Y aun finito pero más numeroso que X, y además la ejecución de este algoritmo ha de ser *indistinguible* del proceso de elegir un elemento de Y de manera uniforme [vzG15, p. 452]. Aquí la palabra "indistinguible" se refiere a que a la larga las sucesiones de elementos generados *parezcan* tener las propiedades de una secuencia aleatoria, para lo cual existen una gran cantidad de test estadísticos que, cuando se aplican sobre una secuencia que "realmente" es aleatoria, sus resultados son conocidos de antemano ya que se pueden expresar en términos propios de la teoría de la probabilidad. Como ejemplo de esto se

[3] La *unidad central de procesamiento*, más conocida por su sigla en inglés "CPU", es el hardware que se encarga de ejecutar las instrucciones de los programas de computadora. Dentro de esta pieza se encuentran unas memorias de altísima velocidad pero de muy limitada capacidad (todo en comparación con cualquier otra memoria o dispositivo de almacenamiento que integre la computadora), llamadas *registros*. Esta decisión de diseño que hace a las computadoras un producto asequible, tiene como consecuencia que los bits almacenados en los registros se renueven constantemente.

tiene el artículo [BRS+10] publicado por el NIST, que presenta una batería de quince test destinados a buscar la presencia o ausencia de patrones que indicarían que la sucesión emitida por un generador pseudoaleatorio no se comporta como una aleatoria.

De todas formas, es importante mencionar que en ciertas aplicaciones criptográficas, la propiedad fundamental de la secuencia pseudoaleatoria es que la misma no pueda ser distinguida de una "verdaderamente" aleatoria mediante un algoritmo **eficiente** (por ejemplo, esto es crucial en los esquemas de identificación). En contraste con la noción previa de *aleatoriedad estadística*, von zur Gathen dedica la mayor parte del undécimo capítulo de [vzG15] a la teoría de la *pseudoaleatoriedad computacional*, que resulta el enfoque adecuado para tales aplicaciones.

Una gama de generadores pseudoaleatorios muy populares (sino los más) son los denominados ***generadores de congruencia lineal***. Estos se basan en recurrencias modulares de la siguiente manera: dado un entero positivo n, a partir de un número $x_0 \in \mathbb{Z}_n$ aleatorio (llamado *semilla*) se generan nuevos valores $x_i \in \mathbb{Z}_n$ empleando la fórmula $x_i \equiv ax_{i-1} + b \pmod{n}$, para valores de a y b fijos tales que $1 < a < n$ y $0 \leq b < n$.

Cuando a y b se eligen apropiadamente, el *período* de un generador [4] de este tipo es suficientemente largo como para que sortee algunos test estadísticos. No obstante, como se muestra en [vzG15, p. 454], observando pocos valores consecutivos de la secuencia generada uno puede hacer suposiciones muy importantes sobre los parámetros a, b y n, las cuales permitirían predecir los próximos valores aún no emitidos [5]. Luego, esta familia de generadores pseudoaleatorios no son buenos para la criptografía práctica.

Afortunadamente para los propósitos de la presente tesis, con que se cumpla lo primero y así la sucesión luzca aleatoria, es suficiente para emplearlos en la construcción de test de primalidad. Más aún, a partir de su sencilla expresión algebraica, es claro que el costo de generar nuevos valores pseudoaleatorios queda determinado por el costo de la aritmética, que en la mayoría de los análisis se asume como $O(1)$ o bien $O(k^{1+\varepsilon})$ para algún $\varepsilon > 0$ [6].

[4] Esto es, la cantidad de números que se pueden producir antes de entrar en un ciclo. Recordar que como por definición el conjunto de elementos generados debe ser finito, los ciclos inevitablemente existen.

[5] Si se conoce un valor x_ℓ de la sucesión (no necesariamente la semilla), aplicando "hacia adelante" la fórmula recursiva del generador se llega a que $x_{\ell+i} \equiv a^i x_\ell + (a^{i-1} + \cdots + 1)b \pmod{n}$. O sea, la fórmula cerrada para el i-ésimo número que será generado a partir de x_ℓ es $x_{\ell+i} \equiv a^i x_\ell + \left(\frac{a^i - 1}{a-1}\right) b \pmod{n}$.

[6] De hecho, la implementación de la aritmética se puede optimizar, por ejemplo, si n se elije como 2^k. En tal caso, la operación de módulo sería un simple truncamiento de los k bits menos significativos.

3.4. Test de Fermat

Llegado a este punto es momento de presentar el primer test de primalidad **probabilístico** del relevamiento. Si bien este algoritmo hace uso del azar, que el mismo ocupe el primer lugar dentro de este trabajo no se trata de un hecho aleatorio, sino que la decisión de que sea el primero pasa principalmente por dos razones. Por un lado, ocurre que el mismo está basado en un resultado clásico de la teoría de números (el *pequeño teorema de Fermat*), entonces resulta muy interesante ver cómo un teorema tan elemental puede repercutir en un algoritmo con muy buenas propiedades. Por otro lado, comprendiendo sus limitaciones es posible seguir un camino progresivo en el cual si este test es dotado con unas mejoras, se llega a otro que es de los más usados en la práctica (en particular para generar las claves de RSA de manera eficiente y confiable).

Hecha esta breve introducción, se presenta el resultado matemático que dará sustento al test desarrollado en la presente subsección.

Teorema 3.4. ***Pequeño teorema de Fermat:*** *Si n es un número primo, entonces para todo $a \in \mathbb{Z}$ coprimo con n se tiene que $a^{n-1} \equiv 1 \pmod{n}$.*

Demostración:

Una demostración que emplea resultados elementales de divisibilidad puede consultarse en [Kob94, p. 20].

No obstante, dado que por hipótesis a es coprimo con n, el camino más directo pasa por aplicar el **Teorema de Euler**: aquí se cumple que $a^{\varphi(n)} \equiv 1 \pmod{n}$. Y por la hipótesis de que n es primo, con la **Proposición 2.12** se concluye que $\varphi(n) = n - 1$.

■

Pensando en el contrarrecíproco del resultado anterior, si existe $a \in \mathbb{Z}$ coprimo con n tal que $a^{n-1} \pmod{n} \neq 1$, el número n debe ser compuesto. Luego, si se quisiera deducir por esta vía que un entero $n > 1$ dado es compuesto, vista la propiedad a testear claramente no es necesario buscar el a coprimo con n en todo $\mathbb{Z}$, alcanza con buscarlo entre los residuos $\{1, \ldots, n-1\}$.

Con un poco de ilusión, uno podría esperar que si al elegir un elemento aleatorio a de $\{1, \ldots, n-1\}$ resulta que $\gcd(a, n) = 1$ y además $a^{n-1} \pmod{n} = 1$, por lo mencionado anteriormente el número n tiene chances de ser primo. Más aún, si este

experimento se lleva a cabo de manera independiente "muchas" veces dando siempre el mismo resultado, la *sensación* de "no existencia" (técnicamente, la ausencia de prueba) de un a que permita concluir por contrarrecíproco del Teorema 3.4 que n es compuesto, aumentaría cada vez más las expectativas de que n sea primo. Y es esta corazonada la base del ***test de Fermat***.

Algoritmo 3.2. Test de Fermat [vzG15, p. 111]

Entrada: $n > 1$ y natural.

Salida: `PSEUDO-PRIMO` o `COMPUESTO`.

```
1  a ⚄← {1, ..., n − 1}
2  IF gcd(a, n) ≠ 1 THEN
3      RETURN COMPUESTO
4  ENDIF
5  IF a^{n−1} (mód n) ≠ 1 THEN
6      RETURN COMPUESTO
7  ENDIF
8  RETURN PSEUDO-PRIMO
```

Teorema 3.5. *Para todo natural $n > 1$: si el Algoritmo 3.2 devuelve `COMPUESTO`, entonces n es compuesto.*

Demostración:

En primer lugar se observa que el a elegido en la línea 1 es menor a n, con lo cual el valor máximo posible para $\gcd(a, n)$ es el propio a, en caso de que a divida a n. Si se cumple la condición del IF de la línea 2, el algoritmo procede a ejecutar la línea 3 donde se retorna `COMPUESTO`. Y en este caso, siendo $\gcd(a, n) \neq 1$, se tiene que existe un entero $x > 1$ tal que $x \mid n$ y –por la observación inicial– también $x < n$. De esta forma, x resulta ser un divisor no trivial de n, con lo cual n es compuesto.

Si se verifica que $\gcd(a, n) = 1$, entonces el programa omite el primer bloque IF y pasa a evaluar el siguiente en la línea 5. Si aquí resulta que $a^{n-1} \pmod{n} \neq 1$, entonces la condición de este nuevo IF se verifica y se habilita la línea 6, donde se devuelve `COMPUESTO`. Y en este caso, las sentencias cumplidas hasta el momento permiten concluir –por contrarrecíproco del **Teorema 3.4**– que n es compuesto.

Por último, si $\gcd(a, n) = 1$ pero $a^{n-1} \pmod{n} = 1$, el programa saltea el bloque IF de la línea 5 y termina en la línea 8 retornando `PSEUDO-PRIMO`. De esta forma, se

comprueba que en todos los casos donde el algoritmo devuelve `COMPUESTO`, efectivamente n es compuesto.

■

Corolario 3.6. *Para todo natural $n > 1$: si n es primo, entonces el Algoritmo 3.2 devuelve `PSEUDO-PRIMO`.*

Luego, cuando la entrada sea un número primo, de ninguna manera el test podría reportar que es compuesto. Por ende, queriendo testear primalidad, queda claro que el test de Fermat tiene la propiedad deseable de **no reportar falsos negativos**, o en otras palabras, cuando n es primo el test de Fermat nunca falla.

Por el contrario, con lo probado hasta el momento el test **sí puede reportar falsos positivos**. Es decir, para un número compuesto en la entrada, el test de Fermat podría fallar en detectar que es compuesto y consecuentemente devolver `PSEUDO-PRIMO`. Entonces la gran pregunta es: ¿qué tan probable sería obtener un falso positivo?

Para responder esta interrogante lo primero será definir un par de conceptos. Cuando $n \in \mathbb{N}$ es **compuesto**, según [vzG15, p. 112] los elementos $a \in \mathbb{Z}_n^\times$ (*i. e.*, los $a \in \{0, \ldots, n-1\}$ con $\gcd(a, n) = 1$) que verifican $a^{n-1} \pmod n \neq 1$ se denominan ***testigos de Fermat*** (o *Fermat witnesses*, por el inglés), pues de acuerdo al contrarrecíproco del Teorema 3.4, *testifican* que n es compuesto. Y a los complementarios $\big($cuando $a^{n-1} \equiv 1 \pmod n\big)$ se los conoce como ***mentirosos de Fermat*** (o en inglés, *Fermat liars*), ya que provocan que el Algoritmo 3.2 retorne `PSEUDO-PRIMO` [7].

Sea $L_n = \{a \in \mathbb{Z}_n^\times \ / \ a^{n-1} \equiv 1 \pmod n\}$, el conjunto de los mentirosos de Fermat asociados a un n compuesto. Luego, por las comprobaciones que hace el Algoritmo 3.2 en las líneas 2 y 5, es claro que L_n es el conjunto de los números que si son elegidos en la línea 1, el resultado es un falso positivo. Se tiene entonces que el tamaño de L_n determina la probabilidad de que el test falle, y para ahondar en esto se presentan los siguientes resultados:

Proposición 3.7. *Para todo $n \in \mathbb{N}$ compuesto: L_n es un subgrupo de $\mathbb{Z}_n^\times$.*

Demostración:

Es claro que la función $f : \mathbb{Z}_n^\times \to \mathbb{Z}_n^\times \ / \ f(a) = a^{n-1} \pmod n$ constituye un homomorfismo del grupo conmutativo $\mathbb{Z}_n^\times$ en sí mismo. Luego, observando que –por su

[7] Por ejemplo, 1 siempre es mentiroso, y como se verá en la Proposición 3.7, es importante que lo sea.

propia definición– L_n resulta ser el $\ker(f)$, por el **Teorema 2.29** se concluye que L_n es un subgrupo de $\mathbb{Z}_n^\times$.

■

Lema 3.8. *Para todo $n \in \mathbb{N}$ compuesto : $|L_n| = |\mathbb{Z}_n^\times|$ o $|L_n| \leq \frac{|\mathbb{Z}_n^\times|}{2}$.*

Demostración:

Por el **Teorema de Lagrange para grupos finitos**, al ser L_n un subgrupo de $\mathbb{Z}_n^\times$ debe cumplirse que $|L_n| \,|\, |\mathbb{Z}_n^\times|$, *i. e.*, ha de existir un $k \in \mathbb{Z}^+$ tal que $|\mathbb{Z}_n^\times| = |L_n|\, k$.

Si $k = 1$ se tiene el primer caso de la tesis. Y si $k > 1$, es decir, si $k \geq 2$, ocurre que $|L_n|\, k \geq 2\,|L_n|$. Esto es $|\mathbb{Z}_n^\times| \geq 2\,|L_n|$, con lo cual $|L_n| \leq \frac{|\mathbb{Z}_n^\times|}{2}$.

■

Por lo tanto, según el lema anterior o bien todos los elementos de $\mathbb{Z}_n^\times$ son mentirosos de Fermat (en otras palabras, no hay testigos de Fermat), o bien a lo sumo la mitad de estos elementos son mentirosos.

Naturalmente surge la pregunta de si habrá algún $n \in \mathbb{N}$ compuesto para el cual ocurre la situación extrema de que $L_n = \mathbb{Z}_n^\times$, y por tanto en la mayoría de las veces el test de Fermat falla y reporta un falso positivo [8]. Lamentablemente para el test, la respuesta a esta pregunta es afirmativa: por exhaución se puede demostrar que para el compuesto $n = 561 = 3 \,.\, 11 \,.\, 17$ se tiene que $\forall\, a \in \mathbb{Z}_n^\times \;:\; a^{n-1} \equiv 1 \pmod{n}$. Y de hecho, este es el primero de una familia de números compuestos caracterizados por dicha propiedad, llamada ***números de Carmichael*** [vzG15, p. 112]. Para más desgracia del test, se ha demostrado en [AGP94] que esta familia de números es infinita, con lo cual estos números "inconvenientes" no son tan raros como a priori se podría creer.

De esta forma se llega al siguiente teorema que responde (parcialmente) la pregunta de qué tan probable es la ocurrencia de un falso positivo:

Teorema 3.9. *Para todo $n \in \mathbb{N}$: si n es un número compuesto pero no de Carmichael, entonces la probabilidad de que el Algoritmo 3.2 devuelva `PSEUDO-PRIMO` es a lo sumo $1/2$.*

Demostración:

Como se observó anteriormente, L_n es el conjunto de números para los cuales el

[8] Salvo por los pocos $a \in \{1, \ldots, n-1\}$ con $\gcd(a, n) \neq 1$, que en la línea 2 del Algoritmo 3.2 habilitan a que el mismo devuelva `COMPUESTO`.

Algoritmo 3.2 devuelve `PSEUDO-PRIMO` cuando n es compuesto. Por tanto, recordando que a se elige aleatoriamente del conjunto $\{1, \ldots, n-1\}$, se tiene que la probabilidad buscada es $\frac{|L_n|}{|\{1,\ldots,n-1\}|} = \frac{|L_n|}{n-1}$.

Dado que n es un número compuesto pero no *de Carmichael*, por el **Lema 3.8** $|L_n| \leq \frac{|\mathbb{Z}_n^\times|}{2}$, entonces $\frac{|L_n|}{n-1} \leq \frac{|\mathbb{Z}_n^\times|}{2(n-1)}$. A su vez se tiene que $\mathbb{Z}_n^\times \subseteq \{1, \ldots, n-1\}$, con lo cual $|\mathbb{Z}_n^\times| \leq |\{1, \ldots, n-1\}| = n-1$ y en consecuencia $\frac{|\mathbb{Z}_n^\times|}{2(n-1)} \leq \frac{n-1}{2(n-1)} = \frac{1}{2}$.

Luego, por transitividad, queda demostrado que la probabilidad buscada es menor o igual a $1/2$.

■

Corolario 3.10. *Para todo $n \in \mathbb{N}$: si n es un número compuesto pero no de Carmichael para el cual el Algoritmo 3.2 se ejecuta t veces de manera independiente, entonces la probabilidad de que el mismo retorne `PSEUDO-PRIMO` en todas las instancias es, a lo sumo, 2^{-t}.*

Por lo tanto, excluyendo el caso de los números de Carmichael, el test de Fermat adquiere una **probabilidad de error exponencialmente chica** conforme este se repite con la misma entrada. Así es, por ejemplo, que si se instanciara el test de Fermat con un número compuesto que no es de Carmichael unas 128 veces, y en todas las veces uno observó que este retornó `PSEUDO-PRIMO`, entonces uno habría experimentado un evento cuya probabilidad de ocurrir era menor a 3×10^{-39}. Visto desde otro ángulo, algún suceso con una probabilidad de *una* en *cien sextillones* era más propenso a ocurrir...

Este resultado tan alentador (al menos para los números que no son de Carmichael) conduce inmediatamente a la siguiente pregunta: ¿qué tan costoso resulta ejecutar 128 veces el Algoritmo 3.2, o en general, t veces? Afortunadamente, el tiempo de ejecución del test de Fermat es polinomial:

Teorema 3.11. *Si $n > 1$ es un natural de k bits, entonces el tiempo de ejecución del Algoritmo 3.2 en el peor caso es $O(k)$.*

Demostración:

Claramente el peor caso para el algoritmo es uno donde este llega a evaluar la condición del IF en la línea 5 (sea la valuación verdadera o falsa, posteriormente el algoritmo termina), esto es, uno donde n es coprimo con el a elegido previamente.

Además también es evidente que en tal caso el tiempo de ejecución del algoritmo estará dominado asintóticamente por el máximo entre: elegir a aleatoriamente del conjunto $\{1, \ldots, n-1\}$ (línea 1), calcular el máximo común divisor con n (línea 2) y la exponenciación $a^{n-1} \pmod{n}$ (línea 5). Asumiendo coste unitario en las operaciones aritméticas, entonces por lo visto en la Subsección 3.3 y en las **Proposiciones 2.45** y **2.46**, se concluye que el orden del tiempo de ejecución del Algoritmo 3.2, en términos de operaciones elementales, es lineal en k.

■

De esta forma, se podría decir que ejecutar t veces el Algoritmo 3.2 para un $n \in \mathbb{N}$ de k bits tiene un costo asintótico $O(t \cdot k)$, con lo cual el test de Fermat es una opción **eficiente** en la práctica. En la siguiente subsección se ahondará sobre cómo resolver el tema de los números de Carmichael sin comprometer esta importante cualidad.

3.5. Reforzando el test de Fermat

Antes de avanzar sobre los resultados que harán posible detectar a los números de Carmichael, lo primero será disponer de la siguiente proposición que permite clasificar las soluciones de la ecuación $x^2 = 1$ en $\mathbb{Z}_{p^\alpha}$.

Proposición 3.12. *Si p es un número primo impar y $\alpha \in \mathbb{Z}^+$, se verifica que $x^2 \equiv 1 \pmod{p^\alpha} \iff x \equiv \pm 1 \pmod{p^\alpha}$ [9]. En otras palabras, 1 y $p^\alpha - 1$ son las únicas "raíces cuadradas" de 1 en $\mathbb{Z}_{p^\alpha}$.*

Demostración:

Se procederá a demostrar la implicancia directa en la fórmula, ya que la recíproca es claro que se cumple. Para esto, considerar $x \in \mathbb{Z}_{p^\alpha}$ una solución cualquiera de la ecuación en congruencia. Luego, $p^\alpha \mid x^2 - 1 = (x+1)(x-1)$.

De acuerdo a la **Proposición 2.3**, el $\gcd(x+1,\, x-1) = \gcd(x+1,\, 2)$, siendo 1 o 2 los valores posibles según la paridad de x. Pero como p es un primo impar, en cualquier caso va a suceder que $p \nmid \gcd(x+1,\, x-1)$. De aquí se concluye que p no puede dividir a ambos $x+1$ y $x-1$ simultáneamente. Por lo tanto, o bien $p^\alpha \mid x+1$, o bien $p^\alpha \mid x-1$, esto es, $x \equiv \pm 1 \pmod{p^\alpha}$.

■

[9] La notación « $X \equiv \pm 1 \pmod{N}$ » debe interpretarse como « $X \equiv 1 \pmod{N} \vee X \equiv -1 \pmod{N}$ ».

Entonces ahora que el enunciado de esta proposición se puede dar por válido, se procederá a atacar propiamente el problema heredado de la subsección anterior. En este sentido, es conveniente mencionar el ***criterio de Korselt*** (cf. [AGP94]) que establece una caracterización de los números de Carmichael alternativa a la definición:

Teorema 3.13. [vzG15, p. 115] *Para todo natural $n > 1$ compuesto, n es de Carmichael si y solo si n es libre de cuadrados y para todo primo $p \mid n$ se tiene que $p - 1 \mid n - 1$.*

De aquí se desprende inmediatamente que **los números de Carmichael son impares**, ya que en caso contrario $n - 1$ sería impar y entonces la condición de que $p - 1 \mid n - 1$ no podría cumplirse para otro divisor primo impar (el cual necesariamente ha de existir, dado que n es compuesto pero libre de cuadrados). Luego, restringir el dominio del test que se pretende crear a los enteros impares (mayores que uno) a priori no sería contraproducente. [10]

Recordando que el sustento del Algoritmo 3.2 está en el Teorema 3.4, con el siguiente teorema se pretende dar una "versión mejorada" de dicho resultado. En otras palabras, el teorema probado a continuación constituye la base de la optimización deseada sobre el test de Fermat.

Teorema 3.14. *Sea $n \geq 3$ un número primo. Escribiendo $n - 1 = 2^e\, m$ con $e \geq 1$ y m impar, se tiene que para todo $a \in \mathbb{Z}$ coprimo con n se cumple que $a^m \equiv 1 \pmod{n}$ o $\exists\, i \in \{0, \ldots, e-1\} \;/\; a^{2^i m} \equiv -1 \pmod{n}$.*

Demostración: (Debida a [Ste09, p. 38])

Suponer, por el contrario, que existe $a \in \mathbb{Z}$ coprimo con n tal que $a^m \not\equiv 1 \pmod{n}$ y asimismo $\forall\, i \in \{0, \ldots, e-1\} : a^{2^i m} \not\equiv -1 \pmod{n}$.

Dado que n es primo, por el **Teorema 3.4** se cumple que $a^{n-1} \equiv 1 \pmod{n}$, o equivalentemente $a^{2^e m} \equiv 1 \pmod{n}$. Puesto que $e \geq 1$, ocurre que $\frac{2^e m}{2} = 2^{e-1} m \in \mathbb{Z}$, entonces sobre esta última congruencia es válido tomar la "raíz cuadrada" de 1 módulo n, para así concluir por la **Proposición 3.12** que $a^{2^{e-1} m} \equiv \pm 1 \pmod{n}$. Sin embargo, por lo establecido en el párrafo anterior ha de descartarse el caso -1, entonces en realidad $a^{2^{e-1} m} \equiv 1 \pmod{n}$.

Si $e - 1$ es todavía mayor que 0, entonces nuevamente se podría tomar "raíz

[10] Si bien *en teoría* esta decisión le quitaría generalidad al test, *en la práctica* resolver la primalidad de un número par es trivial, con lo cual bastaría con implementar una rutina previa para esos casos.

cuadrada" en $\mathbb{Z}_n$ y mediante la **Proposición 3.12** deducir que $a^{2^{e-2}m} \equiv \pm 1 \pmod{n}$, para luego descartar el caso -1 por la suposición del párrafo inicial. O sea, se obtendría que $a^{2^{e-2}m} \equiv 1 \pmod{n}$. Así sucesivamente, combinando la **Proposición 3.12** para tomar "raíz cuadrada" de 1 junto con la suposición que permite descartar la solución -1, se llega a que $a^{2^0 m} \equiv 1 \pmod{n}$, *i. e.*, $a^m \equiv 1 \pmod{n}$. Como se observa, esto contradice el otro supuesto que hasta ahora no se había empleado.

Luego, ha de cumplirse la negación de la hipótesis asumida, con lo cual el teorema queda demostrado.

■

Corolario 3.15. *Sea $n \geq 3$ un natural impar. Si al escribir $n - 1 = 2^e\, m$ (con $e \geq 1$ y m impar) se verifica que existe $a \in \mathbb{Z}$ coprimo con n tal que $a^m \pmod{n} \neq 1$ y $\forall\, i \in \{0, \ldots, e-1\} : a^{2^i m} \pmod{n} \neq n - 1$, entonces n es compuesto.*

De esta forma, a la luz del contrarrecíproco del teorema que se acaba de probar, uno ya puede ir vislumbrando cuáles serían las cuestiones a implementar en el algoritmo de testeo para concluir que $n \geq 3$ e impar es compuesto. Inicialmente, notar que alcanza con buscar la base a directamente en el subconjunto $\{1, \ldots, n-1\}$ en vez de en todo $\mathbb{Z}$ (porque al igual que antes, los cálculos se hacen módulo n), entonces lo que era el primer paso del Algoritmo 3.2 se mantiene. Asimismo se conserva su segundo paso de calcular el $\gcd(a, n)$, ya que aquí –para poder avanzar– también a debe ser coprimo con n. Llegado a este punto se deben hallar los enteros e y m que intervienen en la factorización de $n-1$ mencionada por el teorema [11], porque, si antes el tercer paso para distinguir que el número dado fuera compuesto consistía en efectuar la exponenciación $a^{n-1} \pmod{n}$, en el nuevo procedimiento habrán de calcularse las potencias $a^m \pmod{n}$ y $a^{2^i m} \pmod{n}$ para todo $i \in \{0, \ldots, e-1\}$. Pero si durante el cómputo de alguna de estas potencias ocurriera que el resultado no es el esperado por el corolario, entonces a se estaría comportando tal como lo haría para un número primo, con lo cual en una situación así el test "apostará" por el n vertido en la entrada y habrá de retornar `PSEUDO-PRIMO`.

Si bien en la próxima subsección se mostrará detalladamente, también es perceptible que las nuevas instrucciones del algoritmo diseñado no varían su complejidad en relación al test de Fermat. Luego, lo único que resta por preguntarse es cómo de esta manera, aún eficiente, los números de Carmichael pueden ser detectados.

[11] Ver en la próxima subsección cómo dicha tarea se puede realizar simplemente mirando la representación binaria de $n - 1$.

En respuesta a tal interrogante, para un número n **compuesto e impar** se define

$$M_n = \left\{ a \in \mathbb{Z}_n^\times \;/\; a^m \equiv 1 \pmod{n} \;\; \vee \;\; \exists\, i \in \{0, \ldots, e-1\} \text{ tal que } a^{2^i m} \equiv -1 \pmod{n} \right\}$$

que, de acuerdo a lo que se acaba de explicar, sería el conjunto de las bases a que si salen sorteadas lograrían engañar al test cuando n no es primo. En otras palabras, M_n será el conjunto de *mentirosos* del nuevo test, así como L_n lo era para el de Fermat. Sin embargo, a diferencia de L_n que podía ser todo $\mathbb{Z}_n^\times$ (justamente para los números de Carmichael), en el teorema que viene a continuación se demuestra que M_n es a lo sumo la cuarta parte de $\mathbb{Z}_n^\times$. Esto implica que los números de Carmichael sí tienen *testigos* en el nuevo test, con lo cual se confirma que los cambios propuestos tienen el efecto deseado.

Teorema 3.16. *Dado un entero $n > 9$ compuesto e impar, se tiene que $\frac{|M_n|}{\varphi(n)} \leq \frac{1}{4}$.*

Demostración: (Debida a [Sch08, pp. 102–104])

En virtud de que n es impar, n solo es divisible por números **primos impares**. Luego, considerando el conjunto $S_p = \{k \in \mathbb{N} \;/\; 2^k \,|\, p-1\}$ para todo divisor primo p de n, resulta que $1 \in S_p$ para todo p. Se define entonces $\ell = \max \bigcap_{p \,|\, n} S_p$, como el mayor exponente tal que 2^ℓ divide a $p-1$ para todo primo p que divide a n. Claramente todos los S_p son finitos pues $p - 1 \neq 0$, con lo cual ℓ está bien definido y además $\boldsymbol{\ell \geq 1}$.

Escribiendo $n - 1 = 2^e\, m$ con $e \geq 1$ y m impar, a partir de lo anterior queda claro que $2^{\ell-1} \in \mathbb{N}$ y entonces es pertinente considerar el conjunto

$$B_n = \{a \in \mathbb{Z}_n^\times \;/\; a^{2^{\ell-1} m} \equiv \pm 1 \pmod{n}\}$$

el cual será de ayuda para probar lo deseado. En este sentido, el primer hecho a demostrar es que $\boldsymbol{M_n \subseteq B_n}$, naturalmente partiendo de un $a \in M_n$ genérico. En primera instancia, si a es tal que $a^m \equiv 1 \pmod{n}$, entonces elevando ambos lados de la congruencia a la $2^{\ell-1}$ se ve claramente que $a \in B_n$. Luego, para el otro caso atender a lo que sigue.

Si $a^{2^i m} \equiv -1 \pmod{n}$ para algún $i \in \{0, \ldots, e-1\}$, entonces la congruencia también vale para cualquier divisor de n. Ergo, si p es uno de los primos que dividen a n, se cumple que $a^{2^i m} \equiv -1 \pmod{p}$.

Elevando al cuadrado esta última congruencia se deduce que $a^{2^{i+1} m} \equiv 1 \pmod{p}$, con lo cual $\mathrm{ord}_p(a) \,|\, 2^{i+1} m$ y así $2^{i+1} m = \mathrm{ord}_p(a) h$ para cierto $h \in \mathbb{Z}$. Observar que si este h fuera par, entonces la ecuación previa puede dividirse entre 2 y reescribirse como $2^i\, m = \mathrm{ord}_p(a) \tilde{h}$, donde $h = 2\tilde{h}$. De esta forma, debería ser $a^{2^i m} \equiv a^{\mathrm{ord}_p(a)\tilde{h}} \pmod{p}$. Pero

mientras que $a^{\mathrm{ord}_p(a)\tilde{h}} \equiv \left(a^{\mathrm{ord}_p(a)}\right)^{\tilde{h}} \equiv 1 \pmod{p}$, aquí era $a^{2^i m} \equiv -1 \pmod{p}$. Luego, visto que $p \neq 2$, se tendría el absurdo $-1 \equiv 1 \pmod{p}$. Se concluye entonces que el h antedicho es impar, y ahora sabiendo esto, es claro que en la factorización en números primos de $\mathrm{ord}_p(a)h$ los factores 2 (de haberlos) solo pueden provenir del $\mathrm{ord}_p(a)$. Por igual razón, como m es impar este no aporta factores 2 en la expresión $2^{i+1}m$. Luego, puesto que era $2^{i+1}m = \mathrm{ord}_p(a)h$, gracias al **Teorema de factorización única** se puede inferir que $i+1$ debe ser el mayor exponente para 2 en la factorización de $\mathrm{ord}_p(a)$. En otras palabras, $2^{i+1} \,|\, \mathrm{ord}_p(a)$.

Recordando que $\mathrm{ord}_p(a) \,|\, |\mathbb{Z}_p^\times|$, por transitividad se llega a que $2^{i+1} \,|\, p-1$. Entonces ocurre que $i+1 \in S_p$, y como esto sucede para cualquiera de los primos p que dividen a n, se tiene que $i+1 \in \bigcap_{p \,|\, n} S_p$. Luego, $\ell \geq i+1$ y entonces $2^{\ell-i-1} \in \mathbb{N}$, con lo cual se podría exponenciar con tal valor la congruencia que determina este caso, esto es, $\left(a^{2^i m}\right)^{2^{\ell-i-1}} \equiv (-1)^{2^{\ell-i-1}} \pmod{n}$. Por lo tanto, se concluye que $a^{2^{\ell-1}m} \equiv \pm 1 \pmod{n}$ según si $\ell - i - 1 > 0$ o $\ell - i - 1 = 0$. Consecuentemente, de este caso también se desprende que $a \in B_n$ y así finaliza la prueba de la inclusión de M_n en B_n.

El siguiente gran paso de la demostración consiste en determinar el tamaño del conjunto auxiliar B_n. Observar que por su definición

$$|B_n| \;=\; |\{a \in \mathbb{Z}_n^\times \;/\; a^{2^{\ell-1}m} \equiv 1 \pmod{n}\}| \;+\; |\{a \in \mathbb{Z}_n^\times \;/\; a^{2^{\ell-1}m} \equiv -1 \pmod{n}\}|$$

con lo cual se puede reducir el problema a cada uno de los casos que lo definen.

En relación al primero de los subconjuntos, por el **Teorema chino del resto** se tiene que $a^{2^{\ell-1}m} \equiv 1 \pmod{n} \iff \left\{a^{2^{\ell-1}m} \equiv 1 \pmod{p^{\alpha_p}}\right\}_{\boldsymbol{p \,|\, n}}$, siendo p^{α_p} la mayor potencia que divide a n del divisor primo p de n. Luego, si A_p denota al conjunto de "soluciones parciales" que surgen de la congruencia módulo p^{α_p}, para resolver la congruencia módulo n han de considerarse todas las combinaciones posibles que surgen del producto cartesiano $\prod_{p \,|\, n} A_p$. De aquí se desprende que $\boldsymbol{|\{a \in \mathbb{Z}_n^\times \;/\; a^{2^{\ell-1}m} \equiv 1 \pmod{n}\}| = \prod_{p \,|\, n} |A_p|}$.

Para determinar cuántas "soluciones parciales" tiene cada congruencia módulo p^{α_p}, recuérdese del **Teorema 2.26** que –como p es impar– $\mathbb{Z}_{p^{\alpha_p}}^\times$ es cíclico, *i. e.*, existe una raíz primitiva $g \in \mathbb{Z}_{p^{\alpha_p}}^\times$. Así es (y teniendo en cuenta el **Teorema 2.25**) que resolver la ecuación $a^{2^{\ell-1}m} \equiv 1 \pmod{p^{\alpha_p}}$ para $a \in \mathbb{Z}_{p^{\alpha_p}}$ es equivalente a resolver la ecuación $\left(g^k\right)^{2^{\ell-1}m} \equiv 1 \pmod{p^{\alpha_p}}$ para $k \in \{0, \ldots, \varphi(p^{\alpha_p}) - 1\}$.

$\left(g^k\right)^{2^{\ell-1}m} \equiv 1 \pmod{p^{\alpha_p}} \iff g^{2^{\ell-1}mk} \equiv 1 \pmod{p^{\alpha_p}} \iff \mathrm{ord}_{p^{\alpha_p}}(g) \,|\, 2^{\ell-1}mk \iff$

$\operatorname{ord}_{p^{\alpha_p}}(g) \,|\, dk$, siendo $d = \gcd\left(\operatorname{ord}_{p^{\alpha_p}}(g),\, 2^{\ell-1}m\right)$. Como en esta última relación obviamente $d \,|\, \operatorname{ord}_{p^{\alpha_p}}(g)$, la cadena de equivalencias anterior puede continuar de la forma $\operatorname{ord}_{p^{\alpha_p}}(g) \,|\, dk \iff \frac{\operatorname{ord}_{p^{\alpha_p}}(g)}{d} \mid k \iff k = \frac{\operatorname{ord}_{p^{\alpha_p}}(g)}{d}u$, con $u \in \mathbb{N}$. Así se ve que las soluciones en $\mathbb{Z}_{p^{\alpha_p}}$ serían: $1,\ g^{\frac{\operatorname{ord}_{p^{\alpha_p}}(g)}{d}},\ \ldots,\ g^{\frac{\operatorname{ord}_{p^{\alpha_p}}(g)}{d}(d-1)}$. [12] Por lo tanto, se concluye que hay d soluciones a la ecuación para k, y como un generador del grupo tiene $\operatorname{ord}_{p^{\alpha_p}}(g) = \varphi(p^{\alpha_p}) = (p-1)p^{\alpha_p-1}$, esto implica que $\boldsymbol{|A_p| = \gcd\left((p-1)p^{\alpha_p-1},\, 2^{\ell-1}m\right)}$.

Ahora bien, dado que p es primo, cualquier divisor de p^{α_p-1} es de la forma p^{α} con $0 \leq \alpha \leq \alpha_p - 1$. Pero salvo para el caso de $\alpha = 0$, siempre $p^{\alpha} > p-1$, con lo cual no podría ser un divisor de este otro número. Luego, sucede que $\gcd(p-1, p^{\alpha_p-1}) = 1$. Entonces para estos valores la función gcd se comporta de manera multiplicativa, es decir, $\gcd\left((p-1)p^{\alpha_p-1},\, 2^{\ell-1}m\right) = \gcd(p-1, 2^{\ell-1}m).\gcd(p^{\alpha_p-1}, 2^{\ell-1}m)$. Análogamente, como m es impar, vale que $\gcd(p-1, 2^{\ell-1}m) = \gcd(p-1, 2^{\ell-1}).\gcd(p-1, m)$ y que $\gcd(p^{\alpha_p-1}, 2^{\ell-1}m) = \gcd(p^{\alpha_p-1}, 2^{\ell-1}).\gcd(p^{\alpha_p-1}, m)$. Entonces se tiene la descomposición $\gcd\left((p-1)p^{\alpha_p-1},\, 2^{\ell-1}m\right) = \gcd(p-1, 2^{\ell-1}).\gcd(p-1, m).\gcd(p^{\alpha_p-1}, 2^{\ell-1}).\gcd(p^{\alpha_p-1}, m)$.

Por la definición del propio ℓ, ocurre que $\gcd(p-1, 2^{\ell-1}) = 2^{\ell-1}$. Memorando que p es impar, es claro también que $\gcd(p^{\alpha_p-1}, 2^{\ell-1}) = 1$. Suponer por absurdo que $p \,|\, m$; puesto que $m \,|\, n-1$, también sucedería que $p \,|\, n-1$, o sea, sería $n \equiv 1 \pmod{p}$. Sin embargo, p es un divisor de n, con lo cual en realidad $n \equiv 0 \pmod{p}$. Luego, $p \nmid m$ y por su primalidad se concluye que $\gcd(p^{\alpha_p-1}, m) = 1$.

De esta forma se deduce que $\boldsymbol{|A_p| = \gcd\left((p-1)p^{\alpha_p-1},\, 2^{\ell-1}m\right) = \gcd(p-1, m)\, 2^{\ell-1}}$, y así la fórmula cerrada para el cardinal del primer subconjunto de B_n es:

$$\left|\{a \in \mathbb{Z}_n^{\times} \;/\; a^{2^{\ell-1}m} \equiv 1 \pmod{n}\}\right| \;=\; \prod_{p \,|\, n} \gcd(p-1, m)\, 2^{\ell-1}$$

En relación al segundo de los subconjuntos, al igual que antes se tiene por el **Teorema chino del resto** que $a^{2^{\ell-1}m} \equiv -1 \pmod{n} \Leftrightarrow \left\{a^{2^{\ell-1}m} \equiv -1 \pmod{p^{\alpha_p}}\right\}_{\boldsymbol{p \,|\, n}}$ Entonces denotando por $\widehat{A_p}$ al conjunto de "soluciones parciales" de la congruencia módulo p^{α_p}, la congruencia módulo n tiene $\prod_{p \,|\, n} \left|\widehat{A_p}\right|$ soluciones.

Con la misma técnica de más arriba, se prueba que la cantidad de soluciones a la ecuación congruencial $a^{2^{\ell}m} \equiv 1 \pmod{p^{\alpha_p}}$ es $d' = \gcd\left(\operatorname{ord}_{p^{\alpha_p}}(g),\, 2^{\ell}\, m\right) = \gcd(p-1, m)\, 2^{\ell}$.

[12] Observar que para $u = d$ quedaría $g^{\operatorname{ord}_{p^{\alpha_p}}(g)}$, pero $g^{\operatorname{ord}_{p^{\alpha_p}}(g)} = g^{\left|\mathbb{Z}_{p^{\alpha_p}}^{\times}\right|}$ puesto que g es un generador, donde a su vez $g^{\left|\mathbb{Z}_{p^{\alpha_p}}^{\times}\right|} = g^{\varphi(p^{\alpha_p})} \equiv 1 \pmod{p^{\alpha_p}}$ por el **Teorema de Euler**. Entonces para valores de $u \geq d$ la secuencia de potencias de g ya descripta se repite.

Luego, definiendo $x = a^{2^{\ell-1}m}$, la congruencia resulta equivalente a $x^2 \equiv 1 \pmod{p^{\alpha_p}}$, que por la **Proposición 3.12** tiene soluciones $x \equiv 1 \pmod{p^{\alpha_p}}$ y $x \equiv -1 \pmod{p^{\alpha_p}}$. Por lo tanto, $|\{a \in \mathbb{Z}_n^\times \;/\; a^{2^\ell m} \equiv 1 \pmod{p^{\alpha_p}}\}| = |A_p| + |\widehat{A_p}|$, lo cual se traduce en $\gcd(p-1,m)\,2^\ell = \gcd(p-1,m)\,2^{\ell-1} + |\widehat{A_p}|$. Despejando, de aquí se puede inferir que $\boldsymbol{|\widehat{A_p}| = \gcd(p-1,m)\,2^{\ell-1}}$, o sea, $|\widehat{A_p}| = |A_p|$ y en consecuencia la fórmula cerrada para el cardinal del segundo subconjunto de B_n es la misma que la del primero.

Entonces se concluye que

$$\boldsymbol{|B_n| \;=\; 2\prod_{p\,|\,n} \gcd(p-1,m)\,2^{\ell-1}}$$

Si ahora se multiplica y divide cada factor en la productoria por su correspondiente $\varphi(p^{\alpha_p})$, se observa que entre todos conforman $\varphi(n)$:

$$\begin{aligned}\prod_{p\,|\,n} \gcd(p-1,m)\,2^{\ell-1} &= \prod_{p\,|\,n} \frac{\gcd(p-1,m)\,2^{\ell-1}\,(p-1)p^{\alpha_p-1}}{(p-1)p^{\alpha_p-1}} \\ &= \prod_{p\,|\,n}(p-1)p^{\alpha_p-1} \cdot \prod_{p\,|\,n} \frac{\gcd(p-1,m)\,2^{\ell-1}}{(p-1)p^{\alpha_p-1}} \\ &= \varphi(n) \prod_{p\,|\,n} \frac{\gcd(p-1,m)\,2^{\ell-1}}{(p-1)p^{\alpha_p-1}}\end{aligned}$$

Por ende, se tiene que la proporción

$$\boldsymbol{\frac{|B_n|}{\varphi(n)} \;=\; 2\prod_{p\,|\,n} \frac{\gcd(p-1,m)\,2^{\ell-1}}{(p-1)p^{\alpha_p-1}}}$$

Recuérdese del enunciado del teorema que lo que se desea probar es que el ratio $\frac{|M_n|}{\varphi(n)} \leq \frac{1}{4}$. Para esto, ahora **supóngase lo contrario, *i. e.*, que $\boldsymbol{\frac{|M_n|}{\varphi(n)} > \frac{1}{4}}$.**

En vista de que $M_n \subseteq B_n$, se tiene que $|M_n| \leq |B_n| \;\Rightarrow\; \frac{|M_n|}{\varphi(n)} \leq \frac{|B_n|}{\varphi(n)}$. Luego, por transitividad también $\frac{|B_n|}{\varphi(n)} > \frac{1}{4}$ y así

$$\frac{1}{4} \;<\; 2\prod_{p\,|\,n} \frac{\gcd(p-1,m)\,2^{\ell-1}}{(p-1)p^{\alpha_p-1}} \tag{3.1}$$

Puesto que m es impar, $\gcd(p-1, 2^\ell m) = \gcd(p-1, 2^\ell).\gcd(p-1, m)$. Además, por la definición de ℓ, el $\gcd(p-1, 2^\ell) = 2^\ell$. Entonces $\gcd(p-1, 2^\ell m) = \gcd(p-1,m)\,2^\ell$. Evidentemente $\gcd(p-1, 2^\ell m) \,|\, p-1$, ergo, $\gcd(p-1,m)\,2^\ell \,|\, p-1$. En consecuencia, como $\ell \geq 1$ y $p-1$ es par para un número primo p de los que dividen a n, se puede establecer

que $\gcd(p-1,m)\,2^{\ell-1} \,\big|\, \frac{p-1}{2}$. Por lo tanto, $\exists\, v_p \in \mathbb{Z}^+ \;/\; \frac{p-1}{2} = \gcd(p-1,m)\,2^{\ell-1}\,v_p$, de donde se desprende que $\boldsymbol{\frac{\gcd(p-1,m)\,2^{\ell-1}}{p-1} = \frac{1}{2\,v_p}}$. Luego, si se le llama s a la cantidad de divisores primos de n, vale que:

$$\begin{aligned} 2\prod_{p\,|\,n} \frac{\gcd(p-1,m)\,2^{\ell-1}}{(p-1)p^{\alpha_p-1}} &= 2\prod_{p\,|\,n} \frac{1}{2\,v_p\,p^{\alpha_p-1}} \\ &= 2\cdot\frac{1}{2^s}\cdot\prod_{p\,|\,n} \frac{1}{v_p\,p^{\alpha_p-1}} \\ &= 2^{1-s}\prod_{p\,|\,n} \frac{1}{v_p\,p^{\alpha_p-1}} \\ &\leq 2^{1-s} \end{aligned}$$

donde la última desigualdad surge de que todos los factores de la productoria están acotados por 1, entonces el producto en sí de todos ellos es menor o igual a 1. De esta forma –por transitividad con la Desigualdad 3.1– debe satisfacerse que $\frac{1}{4} < 2^{1-s}$, o sea, $2^{1-s} > 2^{-2}$. De aquí se concluye que $1-s > -2$, es decir, $s < 3$ que en $\mathbb{Z}$ se traduce como $\boldsymbol{s \leq 2}$.

Suponer que $\boldsymbol{s = 2}$, *i. e.*, n tiene solo dos factores primos distintos, a saber $n = p^{\alpha_p}q^{\alpha_q}$. Si alguno de ellos, por ejemplo p, fuera tal que $\alpha_p \geq 2 \;\Rightarrow\; \alpha_p - 1 \geq 1 \;\Rightarrow\; p^{\alpha_p-1} \geq p \geq 3$ (p debía ser impar). Además era $v_p \geq 1$, con lo cual $v_p\,p^{\alpha_p-1} \geq 1\,.\,3 \;\Rightarrow\; \frac{1}{v_p\,p^{\alpha_p-1}} \leq \frac{1}{3}$, y de este modo:

$$\begin{aligned} 2\prod_{p\,|\,n} \frac{\gcd(p-1,m)\,2^{\ell-1}}{(p-1)p^{\alpha_p-1}} &= 2\cdot\frac{1}{2\,v_p\,p^{\alpha_p-1}}\cdot\frac{1}{2\,v_q\,q^{\alpha_q-1}} \\ &= \frac{1}{2}\cdot\frac{1}{v_p\,p^{\alpha_p-1}}\cdot\frac{1}{v_q\,q^{\alpha_q-1}} \\ &\leq \frac{1}{2}\cdot\frac{1}{3}\cdot 1 \end{aligned}$$

Por lo tanto, vía transitividad con la Desigualdad 3.1 quedaría que $\frac{1}{4} < \frac{1}{6}$, lo cual es absurdo. Luego, todo divisor primo de n debe tener su exponente en la factorización de n igual a 1, *i. e.*, $\boldsymbol{n = pq}$.

Si $\alpha_p = \alpha_q = 1$, entonces $p^{\alpha_p-1} = q^{\alpha_q-1} = 1$, por ende

$$\begin{aligned} 2\prod_{p\,|\,n} \frac{\gcd(p-1,m)\,2^{\ell-1}}{(p-1)p^{\alpha_p-1}} &= 2\cdot\frac{\gcd(p-1,m)\,2^{\ell-1}}{p-1}\cdot\frac{\gcd(q-1,m)\,2^{\ell-1}}{q-1} \\ &= \frac{\gcd(p-1,m)\,2^{\ell}}{p-1}\cdot\frac{\gcd(q-1,m)\,2^{\ell}}{2(q-1)} \end{aligned}$$

Luego, invirtiendo la Desigualdad 3.1, a partir de lo anterior queda que

$$4 \;>\; \frac{p-1}{\gcd(p-1,m)\,2^{\ell}} \cdot \frac{2(q-1)}{\gcd(q-1,m)\,2^{\ell}}$$

o en forma equivalente

$$\frac{p-1}{\gcd(p-1,m)\,2^{\ell}} \cdot \frac{q-1}{\gcd(q-1,m)\,2^{\ell}} \;<\; 2$$

En el párrafo siguiente a la Desigualdad 3.1 se había visto que para cualquier número primo p tal que $p \,|\, n$, se cumple que $\gcd(p-1,m)2^{\ell} \,|\, p-1$. Por lo tanto, los cocientes en la desigualdad de aquí arriba son en realidad enteros positivos. Más aún, por la cota impuesta del lado derecho, necesariamente ambos deben ser iguales a 1, con lo cual han de ser $p-1 = \gcd(p-1,m)\,2^{\ell}$ y $q-1 = \gcd(q-1,m)\,2^{\ell}$. Renombrando $w_{p-1} = \gcd(p-1,m)$ y $w_{q-1} = \gcd(q-1,m)$, se ve claramente que $p = 1 + w_{p-1}2^{\ell}$ y $q = 1 + w_{q-1}2^{\ell}$, entonces $p \equiv 1 \pmod{w_{p-1}}$ y $q \equiv 1 \pmod{w_{q-1}}$.

Por hipótesis $n - 1 = 2^{e}\,m$, entonces llegado a este punto se podría reescribir dicha igualdad como $pq = 1 + 2^{e}\,m$. Y de aquí, puesto que $w_{p-1} \,|\, m$ y $w_{q-1} \,|\, m$, se infiere que $pq \equiv 1 \pmod{w_{p-1}}$ así como $pq \equiv 1 \pmod{w_{q-1}}$. Considerando la primera de estas congruencias, con la información disponible se tiene que $1.(1 + w_{q-1}2^{\ell}) \equiv 1 \pmod{w_{p-1}}$, por ende $w_{q-1}2^{\ell} \equiv 0 \pmod{w_{p-1}}$. Y ahora, como $w_{p-1} = \gcd(p-1,m)$ es impar (porque m lo es), se cumple que $\gcd(2^{\ell}, w_{p-1}) = 1$ y entonces existe el inverso de 2^{ℓ} módulo w_{p-1}. Luego, de la última congruencia es posible despejar que $w_{q-1} \equiv 0 \pmod{w_{p-1}}$, *i. e.*, $w_{q-1} \,|\, w_{p-1}$.

Considerando que $pq \equiv 1 \pmod{w_{q-1}}$, análogamente se puede deducir que $w_{p-1} \,|\, w_{q-1}$. Por lo tanto, como ambos w_{p-1} y w_{q-1} son enteros positivos, que la relación de divisibilidad sea simétrica implica que en realidad son el mismo elemento. Consecuentemente, se tiene que $p = q$, contradiciendo el hecho de que eran factores primos diferentes.

Luego, la cantidad de números primos que dividen a n debe ser $\boldsymbol{s = 1}$, y –como n es compuesto– ha de ser $\boldsymbol{n = p^{\alpha_p}}$ con p primo y $\alpha_p \geq 2$. En estas condiciones

$$2\prod_{p \,|\, n} \frac{\gcd(p-1,m)\,2^{\ell-1}}{(p-1)p^{\alpha_p-1}} \;=\; \frac{\gcd(p-1,m)\,2^{\ell}}{(p-1)p^{\alpha_p-1}}$$

con lo cual al invertir la Desigualdad 3.1 en esta instancia queda que

$$\frac{(p-1)p^{\alpha_p-1}}{\gcd(p-1,m)\,2^{\ell}} \;<\; 4$$

Recordar que ya se había visto que $\gcd(p-1, m)2^\ell \mid p-1$, ergo, $\frac{(p-1)}{\gcd(p-1,m)\,2^\ell} \in \mathbb{Z}^+$. Entonces si este cociente es mayor o igual que 1, se tiene que $\frac{p-1}{\gcd(p-1,m)\,2^\ell} \cdot p^{\alpha_p - 1} \geq p^{\alpha_p - 1}$. Luego, por transitividad ha de ser $p^{\alpha_p - 1} < 4$.

Finalmente, las únicas instancias de p primo impar y $\alpha_p \geq 2$ que satisfacen lo anterior son, respectivamente, 3 y 2. Esto implica que $n = 3^2$, pero al mismo tiempo se contradice la hipótesis de $n > 9$. Luego, de la suposición de que $\frac{|M_n|}{\varphi(n)}$ *excede* a $\frac{1}{4}$ se deriva una contradicción, con lo cual ha de ser $\frac{|M_n|}{\varphi(n)} \leq \frac{1}{4}$ como se quería demostrar.

■

Las herramientas matemáticas necesarias para reforzar el test de Fermat han quedado constituidas, con lo cual se da por terminada la presente subsección para darle paso a la implementación del nuevo algoritmo.

3.6. Test de Miller-Rabin

En octubre de 1975, Gary Miller expuso en su tesis doctoral dos algoritmos deterministas para decidir si un número –supóngase, de k bits– es primo o compuesto. El primero de ellos tenía un tiempo de ejecución exponencial $O\left(\sqrt[7]{2}^{\,k}\right)$, y si bien este algoritmo por sí mismo ya representaba un avance en la materia [13], realizándole una "ligera" modificación Miller obtuvo un segundo test **determinista** y **eficiente** (pues corría en tiempo $O(k^4)$), pero su **correctitud** estaba **condicionada** por un resultado matemático que –en aquel entonces y al día de hoy– no está demostrado: la llamada "hipótesis generalizada de Riemann".

La investigación de Miller se publicó en [Mil76] a fines de 1976, y en conocimiento de esta, Michael Rabin alteró el test de tiempo polinomial para hacerlo incondicional a cambio de transformarlo en un algoritmo probabilístico. Para alcanzar su objetivo, Rabin consideró los mismos resultados matemáticos demostrados en la subsección anterior, y tras publicar su artículo en [Rab80], en honor al progreso hecho por ambos hoy en día este algoritmo es conocido como el ***test de Miller-Rabin***.

[13] Miller le atribuye a Pollard la mejor cota superior asintótica existente *hasta ese entonces* para testear primalidad de manera determinista; cf. [Mil76, p. 300].

Algoritmo 3.3. Test de Miller-Rabin [vzG15, p. 112]

Entrada: $n \geq 3$ natural e impar.

Salida: `PSEUDO-PRIMO` o `COMPUESTO`.

```
1  a ←⚄ {1, ..., n − 1}
2  IF gcd(a, n) ≠ 1 THEN
3      RETURN COMPUESTO
4  ENDIF
5  e , m ← DESCOMPONER(n − 1)
6  b ← a^m (mód n)
7  IF b = 1 THEN
8      RETURN PSEUDO-PRIMO
9  ENDIF
10 FOR i FROM 0 TO e − 1 DO
11     IF b = n − 1 THEN
12         RETURN PSEUDO-PRIMO
13     ENDIF
14     b ← b^2 (mód n)
15 ENDFOR
16 RETURN COMPUESTO
```

Teorema 3.17. *Para todo entero impar $n \geq 3$: si el Algoritmo 3.3 devuelve `COMPUESTO`, entonces n es compuesto.*

Demostración:

Análogamente a como ya se había explicado en la demostración de la correctitud del test de Fermat, si se cumple la condición del IF de la línea 2 el algoritmo retorna `COMPUESTO`, y esto es coherente dado que el $\gcd(a, n) \neq 1$ y es un divisor no trivial de n como consecuencia del dominio del a sorteado en la línea 1.

Si se verifica que $\gcd(a, n) = 1$, entonces el programa omite el primer bloque IF y pasa a la línea 5 donde, dado que n es impar, puede obtener los enteros $e \geq 1$ y m impar tales que $n - 1 = 2^e\, m$ [14]. En la siguiente instrucción se carga en una variable "b" el valor de $a^m \pmod{n}$, y en la línea 7 un IF evalúa si dicho valor resultó ser 1. Si el caso fuera afirmativo, entonces el programa habría seguido a la línea 8. Sin embargo, allí este

[14] Ver la especificación de la función DESCOMPONER en el Algoritmo 3.4.

termina retornando `PSEUDO-PRIMO`, con lo cual como se había asumido que la salida era `COMPUESTO`, ha de ser $a^m \pmod{n} \neq 1$.

Así la condición del IF no se verifica y por tanto el algoritmo continúa con su ejecución en la línea 10, donde comienza un bucle FOR regido por un iterador "i" que, como $e \geq 1$, se ejecutará al menos una vez para $i = 0$. Varias consideraciones merece lo que sucede dentro de esta estructura. En primer lugar observar que al ingresar a la misma, la variable b todavía tiene cargado el resultado de la exponenciación modular $a^m \pmod{n}$. Segundo, nótese que las únicas maneras de concluir este bucle pasan por: o bien terminarlo tras ejecutar e rondas (o sea, si i pasa por todos los enteros desde 0 a $e-1$), o bien si el algoritmo detecta con el IF de la línea 11 que el valor de b es $n-1$ $\big($en otras palabras, si en alguna instancia de i se cumple que $b \equiv -1 \pmod{n}\big)$. Dado que en las instancias afirmativas el programa quedaría habilitado a retornar `PSEUDO-PRIMO` en la línea 12, recordando que la hipótesis de salida era `COMPUESTO`, esta se debe haber producido porque el bloque FOR llegó a su fin (*i. e.*, todas las instancias del IF en la línea 11 fueron negativas), puesto que acto seguido el algorimo ejecuta su última instrucción (línea 16) donde efectivamente se retorna eso. Finalmente, la tercera observación pasa por cómo se actualiza b cuando el IF de la undécima instrucción es falso: allí el programa sigue en la línea 14 y el estado de b cambia por el cuadrado módulo n del que era hasta ese entonces. Por lo tanto, en la ronda i se deja preparado para la ronda $i+1$ (de haberla) el valor de $(b_0)^{2^{i+1}} \pmod{n}$, siendo b_0 el estado inicial de b (cuando $i = 0$).

Dicho lo anterior, ahora resulta evidente que la primera vez que se ejecuta el IF de la línea 11 ($i = 0$) este determina que $b_0 = a^m \pmod{n} \neq n-1$, y en las sucesivas veces hasta terminar el bucle (para todo i tal que $0 < i \leq e-1$), el IF concluye que $(b_0)^{2^i} = a^{2^i m} \pmod{n} \neq n-1$. Luego, como $n \geq 3$ e impar y en este otro caso de salida `COMPUESTO` también valía que $\gcd(a, n) = 1$ y $a^m \pmod{n} \neq 1$, por el **Corolario 3.15** se infiere que el a elegido en la línea 1 es un testigo de que n efectivamente es compuesto.

■

Corolario 3.18. *Para todo entero impar $n \geq 3$: si n es primo, entonces el Algoritmo 3.3 devuelve `PSEUDO-PRIMO`.*

Queda entonces demostrado que para los números primos, el test de Miller-Rabin conserva (en relación al test de Fermat) la apreciada propiedad de **no reportar falsos negativos**. Por otro lado, cuando un $n \in \mathbb{N}$ sea **compuesto** e **impar**, los elementos $a \in \mathbb{Z}_n^\times$ para lo cuales o bien $a^m \equiv 1 \pmod{n}$, o bien $\exists\, i \in \{0, \ldots, e-1\}$ / $a^{2^i m} \equiv -1 \pmod{n}$, serán llamados ***mentirosos fuertes*** (o *strong liars*, en inglés),

puesto que provocan que el nuevo Algoritmo 3.3 (a pesar de reforzar al test de Fermat) aún retorne `PSEUDO-PRIMO`.

Al igual que como sucede con los mentirosos de Fermat en su respectivo test, el tamaño del conjunto de mentirosos fuertes para un $n \in \mathbb{N}$ compuesto e impar determina la probabilidad de que el test de Miller-Rabin retorne un **falso positivo**. A propósito de esta problemática, ya se vio en la Subsección 3.4 que los denominados "números de Carmichael" son enteros compuestos para los cuales no existe un testigo de Fermat. Sin embargo, en cuanto al test de Miller-Rabin, citando a Brochero *et al.* (p. 332) podría decirse que **no existen "números de Carmichael fuertes"**: recordando que $|\mathbb{Z}_n^\times| = \varphi(n)$ y que el Teorema 3.16 vale para todo número compuesto e impar mayor que nueve, uno puede establecer el lema enunciado aquí abajo para el conjunto $M_n = \left\{a \in \mathbb{Z}_n^\times \;/\; a^m \equiv 1 \pmod{n} \;\vee\; \exists\, i \in \{0, \ldots, e-1\} \text{ tal que } a^{2^i m} \equiv -1 \pmod{n}\right\}$ de los mentirosos fuertes asociados a un n impar y compuesto.

Lema 3.19. *Para todo natural $n > 9$ compuesto e impar: $|M_n| \leq \frac{|\mathbb{Z}_n^\times|}{4}$.*

Por lo tanto, para todo $n \in \mathbb{N}$ en las condiciones anteriores siempre existe un $a \in \mathbb{Z}_n^\times$ que si sale sorteado en la línea 1 del Algoritmo 3.3, el test de Miller-Rabin logrará deducir que n es compuesto. Así se cumple el siguiente teorema y su inmediato corolario.

Teorema 3.20. *Para todo entero $n > 9$ e impar: si n es un número compuesto, entonces la probabilidad de que el Algoritmo 3.3 devuelva `PSEUDO-PRIMO` es a lo sumo $1/4$.*

Demostración:

Visto que M_n son los elementos que, si uno de ellos sale sorteado del conjunto $\{1, \ldots, n-1\}$, el Algoritmo 3.3 devuelve `PSEUDO-PRIMO` cuando n es un compuesto impar, se tiene que la probabilidad buscada es $\frac{|M_n|}{|\{1,\ldots,n-1\}|} = \frac{|M_n|}{n-1}$.

Siendo n un número compuesto e impar, por el **Lema 3.19** vale la desigualdad $|M_n| \leq \frac{|\mathbb{Z}_n^\times|}{4} \;\Rightarrow\; \frac{|M_n|}{n-1} \leq \frac{|\mathbb{Z}_n^\times|}{4(n-1)}$. Asimismo $\mathbb{Z}_n^\times \subseteq \{1, \ldots, n-1\}$, con lo cual $|\mathbb{Z}_n^\times| \leq n-1$ y en consecuencia $\frac{|\mathbb{Z}_n^\times|}{4(n-1)} \leq \frac{n-1}{4(n-1)} = \frac{1}{4}$. Luego, por transitividad, queda demostrado que la probabilidad de elegir aleatoriamente una base mentirosa es menor o igual a $1/4$.

■

Corolario 3.21. *Para todo entero $n > 9$ e impar: si n es un número compuesto para el cual el Algoritmo 3.3 se ejecuta t veces de manera independiente, entonces la probabilidad de que el mismo retorne `PSEUDO-PRIMO` en todas las instancias es, a lo sumo, 4^{-t}.*

Vale la pena observar que $n = 9$ es el primer número compuesto impar, pero para este caso $|M_9| = |\{1, 8\}| = 2$ y $\varphi(9) = (3-1) \times 3 = 6$, con lo cual $\frac{|M_9|}{|\mathbb{Z}_9^\times|} = \frac{1}{3} > \frac{1}{4}$. No obstante, en la práctica no poder testear dicho valor es tan intrascendente como no poder someter a los números pares... ¡pues estos casos se resuelven en $O(1)$! Entonces, a efectos prácticos es viable decir que este es un **test de propósito general**. De hecho, en [Rab80, p. 134] el propio Rabin afirma que antes de aplicar el test, sería más conveniente que para el n dado uno chequeara divisibilidad por los números primos por debajo de cierta cota C (*e. g.*, $C = 1000$ o incluso $C = \log(n)$ que es lineal en la cantidad de bits de n), así se podría descartar rápidamente números compuestos por factores "chicos".

Si bien con el Lema 3.19 se evidencia claramente que el test de Miller-Rabin es más fuerte que el de Fermat [15], en la práctica esta superioridad es aún más perceptible de lo que marca el Teorema 3.20 puesto que, según [Sch08, p. 105], **en la mayoría de los casos la cota de $\frac{1}{4}$ es holgada**. Es decir, comúnmente sucede que la probabilidad de que el test detecte que la entrada es un número compuesto (en caso de que así lo sea) resulta considerablemente mayor al 75 %. De todas formas, es preciso mencionar que hablando genéricamente, un cuarto de $|\mathbb{Z}_n^\times|$ es la mejor cota que se puede dar para $|M_n|$, ya que en [BMST13, pp. 332-334] se prueba que existen números $n > 9$ e impares que por cómo están compuestos vale la igualdad $\frac{|M_n|}{|\mathbb{Z}_n^\times|} = \frac{1}{4}$ [16].

En sintonía con lo anterior, también se ha estudiado que si en vez de sortear la base $a \in \{1, \ldots, n-1\}$ se prueba una por una dentro de un subconjunto más limitado, entonces la cantidad de ejecuciones del test de Miller-Rabin se podría reducir notablemente. Por ejemplo, en [SW17] se demuestra que *para todo* número compuesto que es menor a 317 044 064 679 887 385 961 981, alguno de los primeros trece números primos sirve como testigo de que el número es compuesto. Más aún, en [Bac90] se probó que **si la hipótesis generalizada de Riemann es verdadera, entonces todo número compuesto n tiene un testigo que es como mucho $2\log^2(n)$**. Luego, en un tiempo de orden cuadrático en la cantidad de bits de n, uno podría revisar todos los números $a \in \{2, \ldots, \lfloor 2\log^2(n) \rfloor\}$ para encontrar la base que testifica que n es compuesto, o si no la encuentra **concluir** que n es primo. Así se ve explícitamente cómo esta conjetura permite transformar el test de Miller-Rabin en un algoritmo determinista y eficiente.

[15] Comparar este resultado con el Lema 3.8: no solo se elimina el caso del grupo completo, sino que además en el caso restante la cota se reduce a la mitad.

[16] De acuerdo a esa fuente, los primeros ejemplos serían $n = 15$ y $n = 91$. Para esos números: $|M_{15}| = |\{1, 14\}| = 2$ y $\varphi(15) = (3-1) \times (5-1) = 8$, luego el ratio de mentirosos fuertes en $\mathbb{Z}_{15}^\times$ es $\frac{2}{8}$. $|M_{91}| = |\{1, 9, 10, 12, 16, 17, 22, 29, 38, 53, 62, 69, 74, 75, 79, 81, 82, 90\}| = 18$ y por su parte $\varphi(91) = (7-1) \times (13-1) = 72$, entonces la proporción de mentirosos fuertes en $\mathbb{Z}_{91}^\times$ es $\frac{18}{72}$.

Mientras la hipótesis se mantenga en ese estado, comparando los Corolarios 3.21 y 3.10 se observa que no solo **la probabilidad de error del test de Miller-Rabin es exponencialmente pequeña** luego de t ejecuciones con la misma entrada, sino que también es **el cuadrado** de la tasa de error del test de Fermat. Entonces si empleando este último 128 veces se tenía una probabilidad aproximada a 3×10^{-39} de obtener un falso positivo, ejecutando en su lugar el test de Miller-Rabin dicha probabilidad decae a aproximadamente 9×10^{-78}. Por lo tanto, más allá de que para el Algoritmo 3.2 es prácticamente imposible detectar a los números de Carmichael pero para el Algoritmo 3.3 no lo es, en el resto de las entradas compuestas el segundo resulta mucho más fiable que el primero.

Finalmente, incumbe preguntarse si todas estas mejoras que refuerzan al test de Fermat tienen un impacto en su tiempo de ejecución. Como se demostrará en breve, asintóticamente hablando el tiempo de ejecución sigue siendo polinomial, pero antes de ir a eso vale la pena enunciar algunas palabras sobre la rutina DESCOMPONER que se emplea en la línea 5 del algoritmo a analizar.

Primero, se exhibe el siguiente algoritmo a efectos de probar que la función de descomponer un natural $N > 1$ par como $N = 2^e\, m$ con $e \geq 1$ y m impar, es computable:

Algoritmo 3.4. Función **DESCOMPONER**

Entrada: $N > 1$ natural y par.

Salida: Como par ordenado, los enteros $e \geq 1$ y m impar tales que $N = 2^e\, m$.

```
1  e ← 0
2  m ← N
3  WHILE m (mód 2) = 0 DO
4      e ← e + 1
5      m ← m/2
6  ENDWHILE
7  RETURN e , m
```

Si $N \neq 0$ como se pide en el dominio de la entrada, entonces es obvio que la cantidad de veces que se puede dividir un número entre 2 es limitada (dicho límite se contabiliza con la variable e). Por ende, los valores que toma la variable m al actualizarla con $\frac{m}{2}$ mientras sea par, forman una secuencia descendente que se detiene en un número impar. También se ve fácilmente que el peor caso para este algoritmo se da cuando N es directamente una potencia de 2, *i. e.*, $N = 2^k \cdot 1$. Allí la cantidad de veces que se ejecuta el bloque WHILE es exactamente k, o sea, lineal en la cantidad de bits de N.

Pensando en la representación binaria de m, la operación de m (mód 2) en la tercera línea se puede resolver con costo unitario tomando el bit menos significativo. Más aún, la quinta instrucción de dividir m entre 2 y reasignar el resultado, podría sustituirse por un desplazamiento de los bits de m un lugar hacia la derecha. Por lo tanto, asumiendo que las operaciones a nivel de bit, las sumas y las asignaciones son todas instrucciones de tiempo constante, el costo de ejecutar el Algoritmo 3.4 en el peor caso es $O(k)$.

Hecha esta discusión pertinente, se procede con el estudio del tiempo de ejecución del test de Miller-Rabin.

Teorema 3.22. *Si $n \geq 3$ es un natural impar de k bits, entonces el tiempo de ejecución del Algoritmo 3.3 en el peor caso es $O(k)$.*

Demostración:

Obviamente el peor caso para este algoritmo es uno donde n es compuesto y el test logra concluir eso recién en la línea 16. Para que esto suceda, el a elegido en la línea 1 deber ser coprimo con n y además no podría ser un mentiroso fuerte (*i. e.*, $a \notin M_n$).

En tal caso, el tiempo de ejecución del programa estará dominado asintóticamente por el máximo entre: elegir a aleatoriamente del conjunto $\{1, \ldots, n-1\}$ (línea 1), calcular el máximo común divisor con n (línea 2), ejecutar la función DESCOMPONER para $n-1$ (línea 5), la exponenciación $a^m \pmod{n}$ (línea 6) y –finalmente– completar las e rondas del FOR que empieza en la línea 10.

Visto que n es representable con k bits, como se trata de un número impar (*i. e.*, terminado en 1 en binario) la representación binaria de $n-1$ va a seguir ocupando k bits. Así que si $n - 1 = 2^e\, m$ con $e \geq 1$ y m impar, denotando por $[m]$ a la cantidad de bits de m, se tiene que $[m] + e = k$. Luego, de acuerdo a la **Proposición 2.46**, $O(k)$ sería una cota para el tiempo de efectuar $a^m \pmod{n}$. Y asumiendo costo unitario en las operaciones aritméticas, comparaciones de igualdad y asignaciones, se ve que ejecutar cada ronda del FOR es $O(1)$, con lo cual $O(k)$ también es una cota superior para el tiempo de ejecutar las e rondas.

Entonces: por lo visto en la Subsección 3.3, considerando la **Proposición 2.45** y que el tiempo de ejecución de la función DESCOMPONER está acotado asintóticamente por $O(k)$, se infiere que el tiempo de ejecución global del Algoritmo 3.3, en términos de operaciones elementales, es de orden lineal en k.

■

De esta forma, la complejidad del test de Miller-Rabin es equivalente a la del test de Fermat (recordar el Teorema 3.11). Asimismo se podría decir que ejecutar t veces el Algoritmo 3.3 para un entero $n \geq 3$ impar de k bits, tiene un costo asintótico $O(t \cdot k)$, con lo cual es una opción **eficiente** en la práctica.

En efecto, **OpenSSL** (un proyecto de software libre de amplio reconocimiento dentro del ambiente criptográfico) tiene su propia implementación del test. Entre las numerosas versatilidades que esta librería de código abierto ofrece, se incluye un comando llamado `genrsa` el cual permite generar una clave privada RSA. De acuerdo a su manual disponible en [OPA], para obtener los números primos que el protocolo requiere, primero se lleva a cabo un ligero procedimiento de cribado (esto es, descartar divisibilidad con números primos pequeños) y luego se hacen varias iteraciones del test de Miller-Rabin.[17] La cantidad de divisores primos tentativos considerados así como la cantidad de veces que se aplica el test probabilístico, dependen ambos del tamaño de la clave requerido, y esta es una opción que se puede especificar al utilizar el comando (siendo 2048 bits el valor predeterminado).

Como ejemplo de aplicación que emplea la librería antedicha, se tiene otro gran proyecto de software libre llamado **OpenVPN**. Su propósito es que las organizaciones puedan establecer redes privadas sobre la infraestructura que ya existe para la Internet pública, con lo cual todo el tráfico que transite por la red debe estar cifrado. Según el sitio oficial [Ope], para lograr esto se emplean todas las funcionalidades de encriptación, autenticación y certificación provistas por OpenSSL. Luego, el test de Miller-Rabin –gracias a su diminuta tasa de error y alta eficiencia– es usado en el día a día (indirectamente) por grandes instituciones que distribuyen sus operaciones en diferentes lugares geográficos.

3.7. Test de Goldwasser-Kilian

El test que será presentado en esta subsección es un algoritmo probabilístico que está basado en **curvas elípticas**. Pero a diferencia de los test de esa índole ya estudiados, en el que aquí será tratado la cualidad de probabilístico viene dada únicamente por el hecho de que al arranque se debe tomar algo aleatorio, mas no porque una salida posible sea `PSEUDO-PRIMO`. Esto quiere decir que el test o bien con total certeza afirma que la entrada es `PRIMO` o `COMPUESTO`, o bien resulta no concluyente para los elementos

[17] Por lo tanto, en realidad lo que se termina usando son *pseudo*primos.

aleatorios considerados, con lo cual habría que volver a ejecutarlo desde el principio. Entonces, debido a esa última posibilidad, la notación que se había estado empleando hasta la subsección anterior para describir algoritmos no resultará tan práctica en la subsección actual. Luego, el algoritmo será descripto en muy alto nivel tratando de ser lo más imperativo posible, siguiendo como referencia lo expuesto en [Kob94, pp. 187-190].

Sea $n > 1$ un natural el cual se desea confirmar si es primo [18]. Si realmente n fuera primo, entonces $\mathbb{Z}_n$ sería un cuerpo y los puntos que verifican la ecuación $y^2 = x^3 + ax + b$ en $\mathbb{Z}_n$ $\big($*i. e.*, la congruencia $y^2 \equiv x^3 + ax + b \pmod{n}\big)$ junto con el llamado punto en el infinito $\mathcal{O}$, definirían una curva elíptica $E(\mathbb{Z}_n)$. Luego, las fórmulas mencionadas sobre el final de la Subsección 2.1.3 efectivamente representarían cómo operar (sumar y duplicar) con puntos en la curva elíptica, y aplicando el *algoritmo de Schoof* para el conjunto $E(\mathbb{Z}_n)$ se obtendría su cardinal. Por lo tanto, considerando a n como la entrada del algoritmo (o sea, siendo su primalidad todavía no confirmada), en el intento de computar alguna de las cosas antedichas uno podría encontrarse con alguna expresión indefinida por culpa de un elemento no invertible en $\mathbb{Z}_n$ [19]. Pero si esto sucede, entonces es porque el valor problemático **no es coprimo** con n, con lo cual al tomar el máximo común divisor se obtendría un divisor no trivial de n, evidenciando así que n **es compuesto**.

Luego, teniendo presente que al operar como si $\mathbb{Z}_n$ fuera un cuerpo podría detectarse que n es compuesto, cuando en la descripción del algoritmo se emplee la palabra *Calcular*, esta debe interpretarse como una función que o bien devuelve lo esperado, o bien devuelve una excepción. Y en este último caso, el manejo de dicha excepción debe consistir en que el test finalice su ejecución retornando `COMPUESTO`.

Hechas las aclaraciones pertinentes sobre la algoritmia, se procede con el resultado matemático que justifica el funcionamiento del test.

Teorema 3.23. *Sean n un entero positivo, m un entero que posee un divisor primo q tal que $q > (\sqrt[4]{n}+1)^2$, y $E = \{(x, y) \in {\mathbb{Z}_n}^2 \ / \ y^2 \equiv x^3 + ax + b \pmod{n}\} \cup \{\mathcal{O}\}$. Si existe $P \in E$ tal que $mP = \mathcal{O}$ y $\left(\frac{m}{q}\right)P$ está definido con $\left(\frac{m}{q}\right)P \neq \mathcal{O}$, entonces n es primo.*

[18] Típicamente uno ejecutaría el algoritmo que será descripto para números que, como entrada de alguno de los test probabilísticos ya estudiados, la salida fue `PSEUDO-PRIMO`.

[19] También podría pasar que las primeras coordenadas de los puntos involucrados en el cálculo sean iguales módulo n y las segundas coordenadas opuestas módulo n, en cuyo caso el resultado de la cuenta se define como el punto $\mathcal{O}$.

Demostración: (Debida a [Kob94, p. 189])

Suponer, por el contrario, que n es compuesto. Entonces por la **Proposición 3.1** y el **Teorema fundamental de la aritmética**, existe un divisor primo p de n tal que $p \leq \sqrt{n}$.

Luego, se tiene que $\mathbb{Z}_p$ es un cuerpo (finito) y entonces está bien definida la curva elíptica $E'(\mathbb{Z}_p) = \{(x, y) \in {\mathbb{Z}_p}^2 \ / \ y^2 \equiv x^3 + ax + b \ (\text{mód } p)\} \cup \{\mathcal{O}'\}$, con lo cual se denotará por m' a la cantidad de puntos en ella.

Por la **Cota de Hasse**, $m' - (p+1) \leq 2\sqrt{p}$, *i. e.*, $m' \leq p + 2\sqrt{p} + 1 = (\sqrt{p}+1)^2$. Y como $p \leq \sqrt{n}$, se tiene que $\sqrt{p} \leq \sqrt[4]{n}$, con lo cual $(\sqrt{p}+1)^2 \leq (\sqrt[4]{n}+1)^2$. Así se infiere que $m' \leq (\sqrt[4]{n}+1)^2$, y por hipótesis esto último era menor a q. Por lo tanto, vale que $m' < q$.

Ahora recordando que q es primo, lo anterior implica que $\gcd(m', q) = 1$, entonces $\exists\, u \in \mathbb{Z} \ / \ uq \equiv 1 \ (\text{mód } m') \ \Rightarrow \ uq = vm' + 1$, con $v \in \mathbb{Z}$. Se tiene entonces que si R es un punto cualquiera de $E'(\mathbb{Z}_p)$, al multiplicarlo por el escalar uq queda que $uqR = (vm'+1)R = vm'R + R$. Y dado que m' es el tamaño del grupo, ocurre que $m'R = \mathcal{O}' \ \Rightarrow \ vm'R = v\mathcal{O}' = \mathcal{O}'$. Por ende, $uqR = R$.

Considerando el $P' \in E'$ que resulta de reducir las coordenadas de P módulo p, por lo que se ha dicho vale que $uq\left(\frac{m}{q}\right)P' = \left(\frac{m}{q}\right)P'$, esto es, $\left(\frac{m}{q}\right)P' = umP'$. En las hipótesis se tenía que $mP = \mathcal{O}$, lo cual significa que los últimos dos puntos (x_1, y_1) y (x_2, y_2) que intervienen en el cálculo de mP satisfacen que $x_1 \equiv x_2 \ (\text{mód } n)$ y $y_1 \equiv -y_2 \ (\text{mód } n)$. Entonces, visto que $p \,|\, n$, esas equivalencias entre las coordenadas valen incluso reduciéndolas módulo p. Por tanto, en el cálculo de mP' también se tiene que al sumar los últimos dos puntos el resultado es $\mathcal{O}'$, el respectivo punto en el infinito de $E'(\mathbb{Z}_p)$. Luego, $umP' = u\mathcal{O}' = \mathcal{O}'$ y así $\left(\frac{m}{q}\right)P' = \mathcal{O}'$.

Sin embargo, la hipótesis de que $\left(\frac{m}{q}\right)P$ está bien definido y asimismo $\left(\frac{m}{q}\right)P \neq \mathcal{O}$, implica que *ningún* par de puntos involucrados en el cómputo de $\left(\frac{m}{q}\right)P$ podrían tener las condiciones para que su suma sea $\mathcal{O}$ (y mucho menos para que las fórmulas de la suma tengan algún problema de indefinición). Entonces trabajando las sumas módulo p en lugar de módulo n, se llega a que también debe ser $\left(\frac{m}{q}\right)P' \neq \mathcal{O}'$.

Luego, esto contradice lo deducido más arriba, y por tanto se concluye que n debe ser primo.

■

El algoritmo que será descripto a continuación data de 1986, cuando fue presentado por Shafi Goldwasser y Joe Kilian en [GK86]. La entrada es un número natural $n > 1$ para el cual, tras haber ejecutado algún test probabilístico, se desea **certificar** su primalidad.

Paso 1

Elegir aleatoriamente a, x_0, $y_0 \in \mathbb{Z}_n$ y establecer $b = {y_0}^2 - {x_0}^3 - ax_0 \pmod{n}$. Si resulta que $4a^3 + 27b^2 \pmod{n} = 0$, entonces repetir el sorteo aleatorio hasta conseguir una configuración de a y b en la que $4a^3 + 27b^2 \pmod{n} \neq 0$. Una vez que se tengan a y b con la condición deseada, se establece E como el conjunto de puntos que verifican la ecuación $y^2 \equiv x^3 + ax + b \pmod{n}$ y $P = (x_0, y_0)$ de acuerdo a los valores sorteados [20].

Paso 2

Calcular el tamaño de $E(\mathbb{Z}_n)$, por ejemplo, empleando el *algoritmo de Schoof*. Si el cálculo fue posible se establece m como este valor, y en caso contrario [21] retornar `COMPUESTO`.

Paso 3

Intentar descomponer $m = kq$ siendo $k \geq 2$ un entero "pequeño" [22] y por su parte $q > (\sqrt[4]{n} + 1)^2$ un número primo o pseudoprimo [23]. Si esto no fuera posible, entonces el test no puede continuar con la configuración de E sorteada inicialmente, con lo cual se debe volver al **Paso 1** a elegir nuevos parámetros.

Paso 4

Calcular el punto mP. Si el cálculo o bien no fuera posible, o bien resultara ser diferente de $\mathcal{O}$, retornar `COMPUESTO`. [24]

[20] Observar que por construcción $P \in E$.

[21] Por lo que ya se explicó al principio de la subsección.

[22] Para no tener ambigüedad y facilitar la implementación, podría tomarse directamente $k = 2$. De hecho, así es como se hace en el artículo original de Goldwasser y Kilian (cf. [GK86, p. 322]).

[23] Es decir, q es un número que se sabe que es primo, o bien luego de aplicarle un test probabilístico (*e. g.*, el *test de Miller-Rabin*) la salida dio `PSEUDO-PRIMO`.

[24] Si n fuera primo, entonces $E(\mathbb{Z}_n)$ efectivamente es una curva elíptica y de esa forma se tiene que m es el tamaño del grupo. Luego, para todo punto en $E(\mathbb{Z}_n)$, multiplicar por el tamaño del grupo debería dar el elemento neutro, es decir, $\mathcal{O}$.

Paso 5

Calcular el punto kP. Para ello se distinguen tres casos:

- Si el cálculo no es posible de realizar, retornar `COMPUESTO`.
- Si el resultado es $\mathcal{O}$, el test es incapaz de concluir algo, con lo cual se debe volver al **Paso 1** para sortear nuevos parámetros.
- Si el resultado es distinto de $\mathcal{O}$, entonces por el Teorema 3.23 se puede afirmar que n es primo siempre y cuando el q proveniente del **Paso 3** efectivamente sea primo. Por lo tanto, aquí se abren dos caminos:
  - Si se tiene la certeza de que q es primo, retornar `PRIMO`.
  - Si q es pseudoprimo, invocar recursivamente este algoritmo con q como entrada. Si con la recursión se logra deducir que q es primo, entonces retornar `PRIMO`. En caso contrario, el test no puede concluir para n y se debe volver al **Paso 1** a elegir una nueva configuración de E.

Estos son todos los pasos que describen el ***test de Goldwasser-Kilian***.

Como bien se ha explicado en los comentarios al pie de página y en los mismos pasos del algoritmo, su correctitud subyace en el Teorema 3.23.

La complejidad de este algoritmo para producir un certificado de primalidad resulta difícil de estimar por las veces que aparece la instrucción "volver al **Paso 1**" cuando el test no es concluyente (incluso el propio **Paso 1** podría ser reiterativo). Particularmente, de acuerdo a [Kob94, p. 190] no existe teorema alguno que garantice que una curva cuyo tamaño sea expresable como se requiere en el **Paso 3**, pueda encontrarse en una cantidad polinomial de intentos (polinomial en el tamaño de n), aunque de todas formas "hay una conjetura muy razonable que sí lo garantizaría, y en la práctica no sería un problema". Por lo tanto, para simplificar el análisis se permitirá que en los casos donde no es posible continuar sin volver a sortear nuevos parámetros, el algoritmo se detenga [25].

Luego, en un caso exitoso donde el algoritmo logre testificar que n es primo, el costo estará determinado por el máximo entre: sortear los parámetros de la curva elíptica en el **Paso 1**, ejecutar el algoritmo de Schoof en el **Paso 2**, encontrar el (pseudo)primo q en el **Paso 3**, efectuar la multiplicación de P por el entero m en el **Paso 4**, y de ser

[25] Por ejemplo, retornando `NULL`.

necesario, aplicar recursivamente el algoritmo en el **Paso 5** [26].

Como se vio en la Subsección 3.3, el costo de sortear los valores que determinan $E(\mathbb{Z}_n)$ y P en el **Paso 1** es despreciable. Asimismo, en el **Paso 4** mP se puede resolver en una cantidad lineal de duplicaciones (lineal según el tamaño de m), de manera análoga a como la *exponenciación por cuadrados* descripta en la Subsección 2.2.2 se puede resolver en una cantidad lineal de cuadrados. Por la **Cota de Hasse**, el tamaño de m está acotado por el tamaño de n, con lo cual asintóticamente hallar mP sigue siendo lineal en el tamaño de la entrada.

Cualquiera de los test probabilísticos vistos en la presente sección que podrían emplearse en el **Paso 3**, efectúan una cantidad de operaciones en $\mathbb{Z}_q$ que es lineal en el tamaño de q. Luego, si q está lo más cerca posible de $(\sqrt[4]{n}+1)^2$, sería razonable asumir que q es similar a $\sqrt{n}$ en tamaño, o sea, q tendría aproximadamente la mitad de los bits de n. Esto implica que cuando se llama a la recursión en el **Paso 5**, el problema de conseguir un certificado para un número que se tiene la convicción de que es primo, ahora reduce su tamaño a la mitad. Ergo, la cantidad de llamadas recursivas es logarítmica en función del tamaño de q, lo cual en la notación asintótica termina siendo lo mismo que logarítmico en el tamaño de n.

Lo último que faltaría considerar es el costo de ejecutar el algoritmo de Schoof en el **Paso 2**. Remitiéndose a [vzG15, p. 227], se ve que este algoritmo puede ser implementado en $O\big(\log^8(n)\big)$, es decir, polinomial en el tamaño de n. Claramente el costo de este paso supera ampliamente a cualquier otro costo lineal, cuadrático o incluso cúbico (usando la aritmética modular más ineficiente) que pudiera ocurrir en el resto del algoritmo, por tanto, es este paso el que asintóticamente marca el tiempo de ejecución de todo el test. Luego, teniendo esto en consideración y el costo del paso recursivo, se concluye que el tiempo de ejecución del test de Goldwasser-Kilian es $O\big(\log(n)^{8+\varepsilon}\big)$ para algún $\varepsilon > 0$.

De esta manera queda probado que, en teoría, el test de Goldwasser-Kilian es un algoritmo probabilístico **eficiente para certificar primalidad**. Sin embargo, a pesar de que su tiempo de ejecución es polinomial en el tamaño de la entrada, en la práctica resulta demasiado lento. Por eso, según [Kob94, p. 190], A. O. L. Atkin desarrolló una variante del test aquí presentado en el que, para evitar usar el algoritmo de Schoof, se construye una curva elíptica muy especial cuya cantidad de puntos es mucho más fácil de computar en comparación con una curva elíptica construida aleatoriamente.

[26] En este paso también se computa kP, pero como $k < m$ por construcción, el costo de esta operación ya quedaría contemplado al considerar el costo de hallar mP en el **Paso 4**.

3.8. La teoría de números ¿al rescate?

Hasta el momento, en la sección actual se abordó el problema de testear primalidad tanto de manera determinista como probabilística. En cuanto a la primera, se trató de resolver la cuestión empleando la aproximación más directa posible, que es "por su definición". Esto dio como resultado un algoritmo de tiempo exponencial, con lo cual acto seguido se decidió pasar al terreno probabilístico en busca de mejores cotas asintóticas. Y ahora que se ha estudiado lo eficiente que pueden resultar estos otros, en la subsección que aquí comienza se pretende retomar los test deterministas para finalmente llegar al hito que representó el test AKS. Se recuerda lo siguiente a modo de motivación:

En la Subsección 3.4 se introdujo la existencia de un conjunto numérico infinito cuyos elementos casi siempre logran engañar al test de Fermat: los números de Carmichael. Asimismo, en la Subsección 3.5 se presentó la existencia de una equivalencia algo más eficiente que la propia definición (el criterio de Korselt). ¿Acaso existe algo análogo para los números primos? La realidad es que sí, y algunos resultados de la teoría de números podrían ser empleados como test de primalidad genéricos y deterministas, como por ejemplo el ***teorema de Wilson***:

Teorema 3.24. *Para todo natural $n > 1$, n es primo si y solo si* $(n-1)! \equiv -1 \pmod{n}$.

Demostración: (Debida a [Ste09, p. 27])

Cuando $n = 2$ y 3 se comprueba la validez del teorema trivialmente. Luego, para lo que sigue suponer que $n \geq 4$.

Si n es primo, entonces $(n-1)! \equiv -1 \pmod{n}$

Como n es primo, entonces $\forall\, a \in \{1, \ldots, n-1\} : \gcd(a, n) = 1$, con lo cual la congruencia $ax \equiv 1 \pmod{n}$ tiene solución en $\{1, \ldots, n-1\}$, que además es única (en otras palabras, existe el inverso de a módulo n). Pero entre todos esos números, hay dos que se destacan por ser su propio inverso:

Si $aa \equiv 1 \pmod{n}$, entonces $n \,|\, a^2 - 1 = (a-1)(a+1)$. Luego, por el **Lema de Euclides**, la primalidad de n permite inferir que $n \,|\, a-1 \ \vee \ n \,|\, a+1$. De acuerdo al dominio de a, se observa que en el primer caso $a - 1 \in \{0, \ldots, n-2\}$ y en el segundo $a + 1 \in \{2, \ldots, n\}$. Entonces, por un tema de tamaños ($n > 1$ no puede dividir a un entero positivo más chico que sí mismo), en el primer caso solo es posible $a - 1 = 0$ y en el segundo $a + 1 = n$. Luego, los elementos que son su propio inverso son 1 y $n - 1$.

Por lo tanto, se concluye que $\forall\, a \in \{2, \ldots, n-2\}$ la ecuación $ax \equiv 1 \pmod{n}$ tiene una única solución en $\{2, \ldots, n-2\}$. Y puesto el conjunto anterior tiene una cantidad par de elementos (hay $n-3$ y se había asumido n como un primo impar), es claro que emparejando cada número con su inverso modular, no sobra ni falta nada dentro del conjunto. Luego, dado que $(n-2)! = (n-2)\ldots 2$, por conmutatividad uno puede reordenar ese producto de forma tal que aparezcan las $\frac{n-3}{2}$ parejas de números inversos módulo n, con lo cual al ver $(n-2)\ldots 2$ módulo n, se tiene un producto de $\frac{n-3}{2}$ unos. Por ende, $(n-2)! \equiv 1 \pmod{n}$.

Finalmente, multiplicando esta última congruencia por $n-1$, resulta en que $(n-1)(n-2)! \equiv n-1 \pmod{n}$, es decir, $(n-1)! \equiv -1 \pmod{n}$.

Si $(n-1)! \equiv -1 \pmod{n}$, entonces n es primo

Suponer –por el contrario– que n es compuesto, esto es, que existe un divisor d de n tal que $1 < d < n$, o equivalentemente que $2 \leq d \leq n-1$. Por consiguiente, d aparece en el producto $2\ldots(n-1) = (n-1)!$, entonces $d \,|\, (n-1)!$.

Por hipótesis $n \,|\, (n-1)! + 1$, y visto que $d \,|\, n$, por transitividad también debe suceder que $d \,|\, (n-1)! + 1$.

Pero si $d \,|\, (n-1)!$ y $d \,|\, (n-1)! + 1$, entonces $d \,|\, 1$, lo cual es imposible ya que era $d > 1$. Luego, n debe ser primo.

■

Sin embargo, como se explicará a continuación, calcular $(n-1)! \pmod{n}$ "por definición" [27] es peor que el *test naíf*.

Lema 3.25. *Si un entero compuesto $n > 4$, entonces $(n-1)! \equiv 0 \pmod{n}$.*

Demostración: (Debida a [BMST13, p. 44])

Si n es compuesto, entonces existen $x, y \in \mathbb{N}$ tales que $n = xy$, con $1 < x < n$ y

[27] Por ejemplo, para hallar $6! \pmod 7$ se haría:

$$\begin{aligned} 6! &\equiv 6.5.4.3.2 \pmod 7 && \text{// Definición de } 6! \\ &\equiv 2.4.3.2 \pmod 7 && \text{// } 30 \equiv 2 \pmod 7 \\ &\equiv 1.3.2 \pmod 7 && \text{// } 8 \equiv 1 \pmod 7 \\ &\equiv 3.2 \pmod 7 \\ &\equiv 6 \pmod 7 \end{aligned}$$

$1 < y < n$. Luego, la demostración se puede desarrollar en dos casos: o bien si alguno de estos divisores de n es mayor que el otro, o bien si son iguales.

Para el primer caso, asumir sin pérdida de generalidad que $x < y$. Sucede así que $2 \leq x < y \leq n-1$, con lo cual tanto x como y aparecen en el producto $2 \ldots (n-1)$. Entonces en el desarrollo de $(n-1)!$, reordenando los factores, uno puede identificar que aparece xy, ergo, $n = xy \,|\, (n-1)!$.

En el segundo caso se tiene que $n = x^2$. Observar que para $x > 1$ y entero, $x^2 \leq 2x \iff x = 2$ [28]. Sin embargo, este valor de x contradice la hipótesis de $n > 4$. Luego, $n = x^2 > 2x$, que para números naturales es equivalente a decir que $2x \leq n-1$. Obviamente si $x > 1 \Rightarrow x < 2x$, entonces se puede concluir que $2 \leq x < 2x \leq n-1$, con lo cual tanto x como $2x$ se hallan en el producto $2 \ldots (n-1)$. Por tanto, en el desarrollo de $(n-1)!$ uno puede identificar que indirectamente aparece x^2, *i. e.*, $n = x^2 \,|\, (n-1)!$.

■

Teorema 3.26. *Si $n > 1$ es un entero de k bits, entonces el tiempo de calcular por definición $(n-1)!$ (mód n) es, en el peor caso, $\Omega(2^k)$.*

Demostración:

Si n es un "gran" número compuesto (entiéndase $n > 4$), por el **Lema 3.25** el residuo $(n-1)!$ (mód n) $= 0$. Esto implica que en el proceso de computar por definición $(n-1)!$ (mód n), debe aparecer algún residuo igual a 0 que anule todo el producto, entonces uno podría dar por concluido el procedimiento aún cuando queden multiplicaciones por hacer. Luego, el peor caso ocurre cuando n es primo, pues de acuerdo al **Teorema 3.24**, ahí ningún producto intermedio podría ser nulo ya que el resultado deber ser $n-1$ (mód n).

Si en principio se supone que una multiplicación con reducción módulo n es una operación elemental, para hallar $(n-1) \ldots 2$ hay que efectuar $[(n-1)-2+1]-1 = n-3$ multiplicaciones. Luego, si n consta de k bits, el valor más alto posible para n es $2^k - 1$. O sea, en el peor caso se contabilizan $2^k - 1 - 3 = 2^k - 4$ multiplicaciones.

Es por esto que el tiempo de calcular por definición $(n-1)!$ (mód n) es, como mínimo, el tiempo de resolver $2^k - 4$ multiplicaciones. Luego, en términos asintóticos, el

[28] Se cumple $x^2 - 2x = 0 \iff x(x-2) = 0 \iff x = 0 \vee x = 2$. En consecuencia, $x^2 - 2x \leq 0 \iff x \in [0, 2]$, donde los únicos valores enteros en este intervalo son 0, 1 y 2. Pero como debe ser $x > 1$, solo ha de considerarse la opción de $x = 2$.

tiempo de ejecución en el peor caso es $\Omega(2^k)$.

■

Y de hecho, no se conoce un método eficiente para calcular factoriales de manera exacta [BMST13, p. 330], con lo cual no hay alternativas sustancialmente mejores que el cálculo por definición. Por lo tanto, el teorema de Wilson no es en realidad un test de primalidad útil en la práctica, al menos mientras se desconozca una manera rápida de hallar $(n-1)!$ (mód n).

3.9. El proyecto GIMPS

Tras haber estudiado diferentes test de primalidad genéricos a lo largo de la presente sección, previo al cierre de la misma se verá un algoritmo determinista dedicado a un tipo de enteros muy particular. Si bien el foco de este relevamiento se sitúa en los algoritmos generales [29], resulta prudente conocer que en la práctica –cuando los números buscados tienen cierta forma– un test específico puede ser preferible.

El caso que aquí compete es, concretamente, el de los llamados ***números de Mersenne***, que son los de la forma $\mathcal{M}_n = 2^n - 1$ donde n es un natural mayor que uno. Estos fueron nombrados así en honor al monje francés Marin Mersenne, quien dejó constancia de haber tratado con algunos de ellos en el siglo XVII. Según [Knu98, p. 409], el primer procedimiento eficiente para testear primalidad sobre esta clase de enteros fue propuesto por Édouard Lucas en 1878, y más tarde se vio mejorado por Derrick Lehmer en 1930. Por eso, el algoritmo central de esta subsección en la actualidad se denomina ***test de Lucas-Lehmer***. Yendo a la cuestión práctica que los involucra, en Internet existe un proyecto de computación distribuida para buscar números *primos* de Mersenne, cuyo afán es crear una lista exhaustiva de los mismos [30] así como romper el récord del número primo más grande conocido. El mismo se conoce por su acrónimo en inglés **GIMPS** (*Great Internet Mersenne Prime Search*, o en español, *Gran búsqueda de primos de Mersenne por Internet*), y desde ya se deja constancia que la información dada hasta el fin de esta subsección fue tomada de su página oficial [Mer] (en inglés).

El proyecto GIMPS comenzó en 1996, cuando George Woltman puso a disposición del público internauta su programa de computadora *Prime95* para buscar números pri-

[29] Y debido a eso, está por fuera de los objetivos trazados profundizar en todas las demostraciones que validan lo que será expuesto en esta subsección.

[30] Actualmente no está probado que existan infinitos de estos números primos; cf. [BMST13, p. 352].

mos de Mersenne de manera distribuida a escala global. Desde entonces, todas las veces que se batió el récord por concepto del "número primo más grande conocido" fue gracias a algún miembro del proyecto. El último número primo confirmado de esta familia –que a su vez resulta ser el mayor número primo conocido– es $\mathcal{M}_{82\,589\,933}$, el cual consta de casi 25 millones de dígitos decimales y fue descubierto el 7 de diciembre de 2018 [31]. Sin embargo, a la fecha de redacción de este trabajo no se han comprobado todos los candidatos entre $\mathcal{M}_{57\,885\,161}$ y el antedicho, con lo cual podrían haber otros primos de Mersenne por descubrir en tal rango.

Para convertirse en integrante del proyecto, un voluntario debe descargar (gratis) el software *Prime95* e instalarlo en una computadora relativamente moderna que pueda dejar encendida la mayor parte del tiempo. Una vez en ejecución, el proceso tendrá la menor prioridad posible entre todas las demás tareas del sistema, de modo que el usuario no perciba un impacto en el rendimiento de su equipo. Eventualmente el programa utilizará Internet para conectarse con un servidor central del proyecto llamado *PrimeNet*, el cual se encarga de recibir resultados y repartir tareas, donde el trabajo es asignado en función de la velocidad del procesador de la computadora del voluntario. Cabe mencionar que para aquellos participantes que tengan la suerte de encontrar un nuevo número primo de Mersenne, GIMPS ofrecerá un premio monetario.

El proceso para descubrir un nuevo número primo de Mersenne consta de cinco etapas, las cuales serán descriptas brevemente a continuación.

Una propiedad básica de esta familia numérica es que ningún exponente que sea *compuesto* puede generar un número *primo* de Mersenne [32]. Entonces, el primer paso consiste en crear una **lista de exponentes primos** p que determinan los enteros $\mathcal{M}_p$ que eventualmente serán testeados.

La segunda etapa –llamada "trial factoring" por GIMPS– se trata de **chequear divisibilidad con factores chicos** que podrían dividir al $\mathcal{M}_p$ considerado. Para lograrlo, el programa construye una versión modificada de la criba de Eratóstenes (remitirse a la Subsección 3.1) basándose en una condición que poseen los factores primos de los

[31] La prueba de primalidad de este número fue llevada a cabo en una computadora con procesador Intel i5-4590T, la cual estuvo doce días ininterrumpidos dedicada a tales efectos. Para demostrar que no hubo errores en el descubrimiento, el entonces nuevo número primo fue verificado de manera independiente con tres programas diferentes en tres configuraciones de hardware diferentes.

[32] Si n es compuesto, entonces existen x, $y \in \mathbb{N}$ tales que $n = xy$, $1 < x < n$ y $1 < y < n$. Luego, $\mathcal{M}_n = 2^{xy} - 1 = (2^x)^y - 1 = \big(2^x - 1\big)\big((2^x)^{y-1} + (2^x)^{y-2} + \cdots + 1\big)$. Dado que $x > 1$ y $y > 1$, en esta descomposición se tiene que **ambos** factores son **mayores que 1**, por lo tanto $\mathcal{M}_n$ es compuesto.

números de la forma $2^p - 1$ con p primo impar (ver próximo párrafo). Como se mostró en la Subsección 3.1 (precisamente en el Teorema 3.3), hacer este proceso puede llegar a ser sumamente ineficiente, con lo cual la gran pregunta es cuánto "trial factoring" vale la pena hacer. Para determinarlo se acota el costo de esta etapa de la siguiente manera: `costo_trial_factoring` $<$ `prob_encontrar_un_factor` $\times$ `costo_testear_primalidad` Mientras que los costos se computan midiendo tiempos de ejecución, la probabilidad de encontrar un factor entre 2^x y 2^{x+1} ha sido estimada (en base a datos históricos) por $1/x$.

La tercera fase radica en ejecutar el **método $p-1$ de Pollard**, que es un algoritmo de factorización no general (*i. e.*, dedicado a números cuyos factores primos satisfacen determinada condición; cf. [Kob94, p. 192]), o mejor dicho una versión de este método ajustada a la propiedad que posee cualquier factor primo q de los números $\mathcal{M}_p$ que son compuestos para p primo impar: debe ser $q = 2kp+1$ con $k \in \mathbb{Z}^+$ (cf. [BMST13, p. 355]). Al igual que en la etapa previa, el objetivo aquí pasa por descartar números con factores primos que aún podrían ser considerados "pequeños", pero el procedimiento tiene dos partes. En la primera parte se elige una cota "B_1", y el método de Pollard habría de encontrar un factor q siempre y cuando todos los divisores primos de k sean menores a B_1. Para la segunda parte se toma otra cota "B_2", y aquí se va a encontrar un factor q si y solo si k tiene un único divisor primo entre B_1 y B_2, y además todos los otros factores primos son menores a B_1. El método $p-1$ de Pollard también tiene un costo para nada despreciable con entradas de gran tamaño, con lo cual resulta crucial elegir astutamente las cotas B_1 y B_2. Para eso, el programa tratará de *maximizar* la fórmula `prob_encontrar_un_factor` $\times$ `costo_testear_primalidad - costo_método_p-1` probando varios valores de B_1 y B_2 de acuerdo a la memoria disponible.

Si luego de las dos etapas anteriores todavía no hay indicios de que el candidato $\mathcal{M}_p$ fuera compuesto, entonces en la cuarta etapa se procede a aplicarle el ***test de Lucas-Lehmer***. No es difícil reconocer que el Algoritmo 3.5 exhibido más abajo se basa en el siguiente resultado:

Teorema 3.27. [BMST13, p. 353] *Sean $p > 2$ un número primo y la recurrencia* $S_k = \begin{cases} 4 & , \text{ si } i = 0 \\ (S_{k-1})^2 - 2 & , \text{ si } i > 0 \end{cases}$. *Luego, $\mathcal{M}_p$ es primo si y solo si $S_{p-2} \equiv 0 \pmod{\mathcal{M}_p}$.*

Algoritmo 3.5. Test de Lucas-Lehmer

Entrada: $p > 2$ natural y primo.

Salida: `PRIMO` o `COMPUESTO`, de acuerdo a lo que sea $\mathcal{M}_p$.

```
1  M ← 2^p − 1
2  S ← 4
3  FOR i FROM 1 TO p − 2 DO
4      S ← S^2 − 2 (mód M)
5  ENDFOR
6  IF S = 0 THEN
7      RETURN M_p es PRIMO
8  ENDIF
9  RETURN M_p es COMPUESTO
```

Observar que el cálculo de $2^p - 1$ en la línea 1 se resuelve simplemente estableciendo p bits consecutivos en 1 en la variable M, con lo cual el tiempo de ejecución de esta instrucción así como la cantidad de pasos que se efectúan en el bloque FOR de la línea 3, son lineales en el tamaño del número que se desea testear (el $\mathcal{M}_p$). Sin embargo, para la magnitud de los números que se manejan en GIMPS sería ridículo asumir como hipótesis que la instrucción que sucede dentro del FOR lleva tiempo constante. El costo de la tarea en la línea 4 está dominado por el tiempo de realizar un cuadrado módulo un número de p bits (*i. e.*, $\mathcal{M}_p$), con lo cual una posible cota superior asintótica de peor caso es $O(p^2)$, usando el algoritmo clásico de multiplicación. Por lo tanto, el tiempo de ejecución del test de Lucas-Lehmer es $O(p^3)$, lo cual es polinomial en la cantidad de bits de $\mathcal{M}_p$. O sea, se trata de un test de primalidad **eficiente** para los números con la forma requerida.

De todas formas, son tan grandes los números manejados por GIMPS, que si bien teóricamente hacer aritmética modular con costo $O(p^2)$ es eficiente, en la práctica resulta demasiado lento. Luego, uno debería buscar la manera más rápida de elevar al cuadrado módulo $2^p - 1$ para reducir la complejidad de la tarea. Desde fines de la década de 1960, el algoritmo más rápido para elevar al cuadrado grandes números está basado en la transformada rápida de Fourier, y para calcularla en el programa de GIMPS se emplea aritmética de punto flotante. Debido a que los cálculos en punto flotante no son exactos, los valores computados se redondean a enteros. Sin embargo, esto trae aparejado un error de redondeo (la diferencia entre el valor de punto flotante y su entero más cercano), que si supera cierta tolerancia preestablecida indica que el cálculo de la transformada rápida de Fourier debería repetirse con otra configuración.

Finalmente, la quinta fase es una de **doble comprobación** en la que el test de primalidad se vuelve a ejecutar para la misma entrada hasta obtener dos resultados iguales. Por una cuestión de seguridad adicional, en esta segunda ejecución del test los bits del valor inicial con el que arranca la secuencia recurrente ($S_0 = 4$ en el algoritmo expuesto) se desplazan hacia la izquierda en una cantidad aleatoria, pues esto permitiría detectar la ocurrencia de algún error en los cálculos con la transformada de Fourier en caso de haberlos.

A partir del año 2018, GIMPS comenzó a emplear el *test de Fermat* para buscar nuevos (pseudo)primos de Mersenne, dado que Robert Gerbicz encontró una forma de eliminar casi por completo la aparición de un error de hardware al aplicar el mencionado test (no obstante, la última etapa de doble comprobación seguía siendo necesaria). Sin embargo, gracias a un avance muy reciente debido a Krzysztof Pietrzak, se logró que la primera corrida del test de Lucas-Lehmer produzca un archivo `.proof` el cual puede ser verificado con ¡menos del 0.5 % del trabajo requerido por una segunda corrida! Por ende, a partir de 2020 se incorporó al programa de GIMPS la generación de estos archivos, y junto con el test de Fermat (como fase previa del test de Lucas-Lehmer) se convirtió en la forma preferida de buscar nuevos primos de Mersenne.

3.10. Test AKS

Recapitulando los algoritmos que se han tratado hasta este momento:

- En las Subsecciones 3.1 y 3.8 se presentaron dos test de primalidad deterministas ***pero*** sus tiempos de ejecución no son polinomiales.
- En las Subsecciones 3.4 y 3.6 fueron estudiados dos test de primalidad cuyos tiempos de ejecución son polinomiales ***pero*** no son algoritmos deterministas.
- También se vio en la Subsección 3.6 que una ligera modificación al test de tiempo polinomial allí tratado, permitía hacerlo determinista ***pero*** su correctitud quedaba sujeta a un resultado matemático que aún no ha sido demostrado.
- Por último, en la Subsección 3.9 se mostró un test de primalidad determinista y con tiempo de ejecución polinomial, ***pero*** este solo sirve para números de cierta forma.

Luego, a través de este punteo se observa que los test de primalidad presentados a lo largo de este trabajo representan fielmente el problema matemático y computacional que por siglos estuvo abierto: ¿acaso es posible tener un test de primalidad que de manera

simultánea: sea determinista, su tiempo de ejecución sea polinomial, su correctitud esté incondicionada y sea de propósito general?

Ha llegado el momento de ver el algoritmo que confirmó que la respuesta a dicha pregunta ¡es afirmativa! En el año 2002 Manindra Agrawal, Neeraj Kayal y Nitin Saxena presentaron el primer test que reúne todas las condiciones anteriores, en un artículo que posteriormente fue publicado en [AKS04]. De esta forma, marcaron un hito en la teoría de la complejidad computacional, demostrando que el problema de **determinar si un número es primo** pertenece a la célebre **clase P**.

El algoritmo que se describirá más abajo está basado en el siguiente teorema, cuya demostración hace uso de resultados avanzados de álgebra abstracta. En realidad este resultado no es el que publicaron Agrawal, Kayal y Saxena originalmente, sino que se trata de una simplificación observada por Hendrik Lenstra.

Teorema 3.28. [BMST13, p. 341] *Por un lado, sean n, r y v enteros mayores que uno, donde r es una potencia de un número primo. Por otro lado, sea S un conjunto con s elementos. Si se cumple que:*

1) $\gcd(n, r) = 1$ *y* $\operatorname{ord}_r(n) = v$

2) $\forall\, a, b \in S$ *con* $a \neq b$ *se tiene que* $\gcd(n, a - b) = 1$

3) $\forall\, t \mid \varphi(r)$ *que asimismo es múltiplo de* v, *se tiene que* $\binom{s+t-1}{s} \geq n^{\sqrt{t/2}}$

4) $\forall\, a \in S$ *se tiene que* $(x + a)^n \equiv x^n + a \pmod{x^r - 1,\, n}$

Entonces n es una potencia de un número primo.

Resulta pertinente aclarar que la notación $p(x) \equiv q(x) \pmod{x^r - 1,\, n}$ debe interpretarse como una "congruencia generalizada", la cual estaría definida como que $p(x) - q(x) \in \{n.f(x) + (x^r - 1).g(x) \;/\; f(x),\, g(x) \in \mathbb{Z}[x]\}$, siendo $\mathbb{Z}[x]$ el anillo de polinomios con coeficientes en $\mathbb{Z}$.

Como se muestra en [BMST13, p. 344], se puede probar que si $v > \frac{1}{2}\log_2(n)^2$, tomando $S = \{0, 1, \ldots, \ell\}$ con $\ell = \left\lfloor \sqrt{\frac{1}{2}\varphi(r)}\, \log_2(n) \right\rfloor$, se tiene que r y s satisfacen la tercera condición del teorema.

Para entender la correctitud del test AKS, también debe considerarse el próximo:

Lema 3.29. [BMST13, p. 344] *Si $n \geq 9$ es un entero, entonces existe r, una potencia de un número primo, tal que $r < \log_2(n)^5$ y $\operatorname{ord}_r(n) > \frac{1}{2}\log_2(n)^2$.*

Entonces ahora que se han exhibido los fundamentos matemáticos pertinentes, he aquí el ***test AKS***.

Algoritmo 3.6. Test AKS [BMST13, p. 345]

Entrada: $n > 6$ natural.

Salida: `PRIMO` o `COMPUESTO`.

```
1  IF esPotenciaPerfecta(n) THEN
2      RETURN COMPUESTO
3  ENDIF
4  r ← 2
5  WHILE r < log_2(n)^5 DO
6      IF ordenModulo(n, r) > (1/2) log_2(n)^2 THEN
7          BREAK
8      ELSE
9          r ← r + 1
10     ENDIF
11 ENDWHILE
12 FOR a FROM 2 TO r DO
13     IF 1 < gcd(a, n) < n THEN
14         RETURN COMPUESTO
15     ENDIF
16 ENDFOR
17 IF √n < r THEN
18     RETURN PRIMO
19 ENDIF
20 FOR a FROM 2 TO ⌊√((1/2)φ(r)) log_2(n)⌋ DO
21     IF potenciaPolinomio(a, n, r) ≠ x^n + a (mód x^r − 1, n) THEN
22         RETURN COMPUESTO
23     ENDIF
24 ENDFOR
25 RETURN PRIMO
```

Para analizar la complejidad del Algoritmo 3.6 se utilizará la notación $\widetilde{O}(k^i)$ que es equivalente a $O\Big(k^i.P\big(\log(k)\big)\Big)$, siendo P un polinomio.

La rutina **esPotenciaPerfecta(**n**)** determina si la entrada se puede expresar como $n = a^b$ con $a, b > 1$. En [BMST13, p. 346] se muestra un pseudocódigo que implementa esto y tiene orden $\widetilde{O}\big(\log(n)\big)$.

La función **ordenModulo(**n, r**)** calcula las potencias módulo r de n hasta $\lfloor \frac{1}{2} \log_2(n)^2 \rfloor$ inclusive, o bien hasta que alguna de ellas dé 1 como resultado. Luego, el peor caso se da cuando para cualquier $j \leq \lfloor \frac{1}{2} \log_2(n)^2 \rfloor$ sucede que $n^j \pmod r \neq 1$, con lo cual $\text{ord}_r(n) > \lfloor \frac{1}{2} \log_2(n)^2 \rfloor$ y por eso la función devolvería (de manera simbólica) $\lfloor \frac{1}{2} \log_2(n)^2 \rfloor + 1$. De esta forma, se ve que el costo de esta rutina es $\widetilde{O}\big(\log(n)^2\big)$. Puesto que la misma aparece dentro de un WHILE que en el peor caso haría $\lfloor \log_2(n)^5 \rfloor$ iteraciones, se tiene que el costo total de esta parte es $\widetilde{O}\big(\log(n)^7\big)$.

Como se había mencionado en la Subsección 2.2.2, el costo de calcular un máximo común divisor es lineal en el tamaño de la entrada, o sea, $O\big(\log(n)\big)$. Pero como el algoritmo ejecuta $r-1$ veces este paso, se tiene que el mismo aporta un tiempo del orden $O\big(r \log(n)\big)$. Luego, como el peor caso para r es $\lfloor \log(n)^5 \rfloor$, se tiene que la complejidad de esta parte es $O\big(\log(n)^6\big)$.

La comparación de $\sqrt{n} < r$ hecha en la línea 17 requiere un tiempo $O\big(\log(n)\big)$.

Finalmente, la rutina **potenciaPolinomio(**a, n, r**)** calcula $(x+a)^n \pmod{x^r - 1,\ n}$. En [BMST13, p. 348] se encuentra un pseudocódigo que implementa esto en tiempo $\widetilde{O}\big(r \log(n)^2\big)$. Y como el peor caso para r es $\lfloor \log(n)^5 \rfloor$, la complejidad de esta función es $\widetilde{O}\big(\log(n)^7\big)$. En virtud de que la misma aparece dentro de un FOR que efectúa $\left\lfloor \sqrt{\frac{1}{2}\varphi(r)}\ \log_2(n) \right\rfloor - 1$ iteraciones, acotando el tamaño de $\varphi(r)$ por el de r y aplicando el peor caso para r, queda que la cantidad de iteraciones es $\widetilde{O}\big(\log(n)^{\frac{5}{2}+1}\big) = \widetilde{O}\big(\log(n)^{\frac{7}{2}}\big)$. De este modo, el costo total del último paso es del orden $\widetilde{O}\big(\log(n)^{\frac{21}{2}}\big)$.

Es así que el tiempo de ejecución del test AKS está dominado por el costo de realizar su último paso, el cual termina provocando que la complejidad del test sea $\widetilde{O}\big(\log(n)^{\frac{21}{2}}\big)$. Esta cota superior asintótica, en comparación con $\widetilde{O}\big(\log(n)\big)$ de –por ejemplo– el *test de Miller-Rabin*, indica que el tiempo de ejecución resulta tremendamente lento. Por lo tanto, si bien en la teoría de la computación se reconoce el descubrimiento de este test como un gran hito, en la práctica (en particular para RSA) se siguen empleando los test probabilísticos predecesores a este.

4. Factorización de enteros

Dado un natural n impar y compuesto, entre todos los algoritmos vistos en la sección anterior, solamente el *test naíf* podría descubrir la no primalidad de n y al mismo tiempo ser capaz de dar un divisor propio, ya que básicamente lo que hace es probar con todos los números impares hasta la $\sqrt{n}$ para ver si alguno de ellos es un divisor. Esta técnica podría refinarse un poco más y en lugar de comprobar todos los valores impares, podrían emplearse directamente los números primos hasta la $\sqrt{n}$. Asumiendo que n es representable con k bits, el peor caso para ese procedimiento es con el n más grande de todos, *i. e.*, $2^k - 1$. Luego, por el **Teorema de los números primos**, hasta la $\sqrt{2^k - 1}$ se esperaría tener que chequear divisibilidad (con un número primo) una cantidad de veces del orden $O\left(\frac{\sqrt{2}^k}{k}\right) \subset O\left(\sqrt{2}^k\right)$. Por lo tanto, la complejidad del método lo hace ineficiente cuando el tamaño de n es grande.

Todos los test rápidos que se estudiaron en la sección previa son capaces de afirmar que n *debe* tener divisores propios [1], pero no dicen nada sobre cuáles son. Y como se había adelantado en la Subsección 2.2.3, que se mantenga esta dualidad resulta vital para la seguridad del protocolo RSA.

En la presente sección se explorarán técnicas para encontrar divisores, que si se aplican de manera iterativa, permiten descubrir la factorización en números primos de un entero dado. De hecho, cuando $n = pq$ con p y q primos (es decir, se sabe que la entrada es parte de una clave RSA), al encontrar un divisor no trivial –supóngase p– el otro se obtiene inmediatamente dividiendo n entre p [2].

4.1. Método rho de Pollard

El primer algoritmo substancialmente más rápido que el método *naíf* fue presentado en 1975 por John Pollard, bajo el título *A Monte Carlo method for factorization* [Pol75] (por eso este algoritmo también es conocido como "el método Monte Carlo").

A grandes rasgos, esta técnica para factorizar un natural n impar y compuesto se puede describir de la siguiente manera:

[1] Con cualquiera de estos test es que, en primer lugar, uno podría saber que n es compuesto.

[2] En el caso general habría un paso adicional de testear primalidad, para decidir si corresponde seguir buscando factores.

1) Lo primero es **establecer un polinomio $f(x)$** con coeficientes en $\mathbb{Z}_n$, dado que el mismo será evaluado módulo n. A la brevedad se verá que, en teoría, el éxito del algoritmo se basa en que este polinomio sea "aleatorio" en cierto sentido. Sin embargo, de acuerdo a [Kob94, p. 142], la experiencia sugiere que algunos polinomios muy simples como $f(x) = x^2 + 1$ son suficientes.

2) Luego **se elige un valor inicial $x_0 \in \mathbb{Z}_n$**, que al igual que el polinomio podría estar definido de antemano (*e. g.*, tomando $x_0 = 1$ o $x_0 = 2$). No obstante, como el método puede "fallar" en el sentido de que el factor encontrado es el propio n, eligiendo este valor aleatoriamente se podría esperar tener más suerte con una nueva ejecución [3].

3) Finalmente se va computando la secuencia $x_i = f(x_{i-1})$ y en cada paso este valor se compara con x_j, siendo $j = 2^{h-1} - 1$ y h la cantidad de bits de i [4]. Esencialmente lo que buscará el método es averiguar si x_i y x_j están en la **misma clase** módulo algún divisor de n, ya que esto es fácil de hacer calculando el $\gcd(x_i - x_j,\, n)$ [5]. Por ende, un próximo x_{i+1} deberá ser computado si para el x_i actual el $\gcd(x_i - x_j,\, n) = 1$.

Notar que cuando el algoritmo deduzca $x_i \equiv x_j \pmod{d}$ para un $d \mid n$, puesto que por construcción $i > j$, esto implica que x_i sería congruente con uno de los valores ya calculados, *i. e.*, si el método siguiera aplicando f permanecería en un ciclo módulo d.

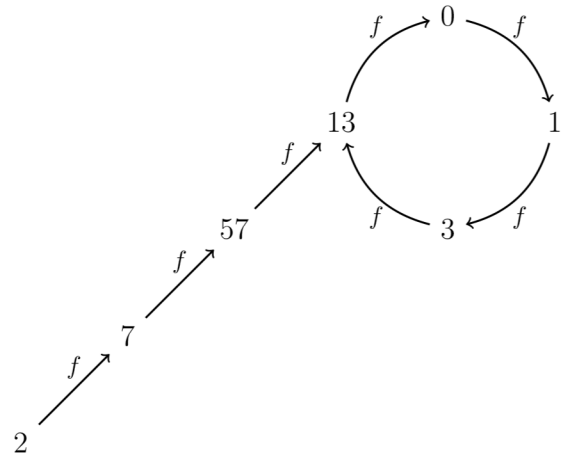

Figura 4.1: Método ρ de Pollard para $n = 4087$, empleando $x_0 = 2$ y $f(x) = x^2 + x + 1$
En este caso $n = 61 \,.\, 67$ y el método detecta un ciclo en módulo 61

[3] Claramente fijando f y x_0, una nueva ejecución del test no va a provocar un resultado diferente; en los casos de "falla" hay que cambiar el polinomio y/o el valor inicial.

[4] Es decir, mientras $2^{h-1} \leq i < 2^h$ hay que recordar el último x_j con j el mayor entero de $h - 1$ bits.

[5] Si $d = \gcd(x_i - x_j,\, n) \;\Rightarrow\; d \mid x_i - x_j \;\Rightarrow\; x_i \equiv x_j \pmod{d}$. Recíprocamente, si d es un divisor no trivial de n y $x_i = x_j$ en $\mathbb{Z}_d$, entonces $\gcd(x_i - x_j,\, n) \geq d$. Luego, es aquí donde el método podría "fallar" si justo la secuencia de valores que se tiene hasta el momento cumple que $\gcd(x_i - x_j,\, n) = n$.

Por lo tanto, el nombre del método proviene de que otra interpretación de lo que hace es –indirectamente– **detectar ciclos** módulo algún divisor de n, y si la secuencia de los x_i necesarios se grafica como aquí arriba [6], parece la letra griega ρ.

Hecha la descripción anterior, una implementación del ***método rho de Pollard*** es la que sigue.

Algoritmo 4.7. Método rho de Pollard

Entrada: $n \in \mathbb{N}$ impar y compuesto.

Salida: Un divisor de n.

```
1   x ←⚄ {0, ..., n − 1}
2   y ← x
3   i ← 0
4   d ← 1
5   WHILE d = 1 DO
6       i ← i + 1
7       x ← f(x) (mód n)          // Polinomio f definido de antemano, por ejemplo f(x) = x^2 + 1
8       d ← gcd(x − y, n)
9       IF esPotenciaDe2Menos1(i) THEN
10          y ← x
11      ENDIF
12  ENDWHILE
13  RETURN d
```

La función **esPotenciaDe2Menos1(i)** simplemente determina si $i = 2^h - 1$ para algún h (es decir, si i es el mayor entero de $h-1$ bits; ver el tercer punto en la descripción general). Esto se puede concluir fácilmente mirando si todos los bits significativos de la variable i están en 1, o bien si la cuenta $\log_2(i+1) - \lfloor \log_2(i+1) \rfloor$ es igual a 0 (*i. e.*, si $\log_2(i+1) \in \mathbb{N}$).

Luego, asumiendo que las operaciones aritméticas y las asignaciones son de costo unitario, la instrucción más costosa del Algoritmo 4.7 es el cálculo del máximo común divisor en la línea 8, que por lo visto en la Subsección 2.2.2 este paso tiene un tiempo de ejecución lineal en el tamaño de n. Dado que dicha tarea se encuentra inmersa en un bloque WHILE, para determinar el costo total del algoritmo resulta preciso contabilizar cuántas iteraciones habrían de efectuarse. En este sentido, a continuación se demostrarán

[6] En el ejemplo de la Figura 4.1 fueron necesarias siete iteraciones, ya que allí $x_3 \equiv 3307 \pmod{4087}$ y $x_7 \equiv 3734 \pmod{4087}$, con lo cual $\gcd(x_7 - x_3, n) = \gcd(427, 4087) = 61$.

un par de resultados que estiman cuán lejos normalmente será necesario llegar con i para que el método active la condición de parada.

Proposición 4.1. *Sean S un conjunto con d elementos y $\lambda \in \mathbb{R}^+$. Si $\ell = 1 + \lceil \sqrt{2\lambda d} \rceil$, entonces se cumple que la proporción de los pares (f, x_0) en los cuales $f : S \to S$, $x_0 \in S$, $x_{i+1} = f(x_i)$ y los valores $x_0, x_1, \ldots, x_\ell$ son distintos dos a dos, entre todos los pares de función de S en S y valor inicial en S, es menor que $e^{-\lambda}$.*

Demostración: (Debida a [Kob94, pp. 139-140])

*Primero se determinará cuántos pares (f, x_0) existen **sin** las condiciones solicitadas.*

Si $f : S \to S$ y no hay ninguna restricción sobre cómo actúa f, entonces para cada $x \in S$ hay $|S|$ opciones diferentes para elegir $f(x)$. Luego, hay $\prod_{x \in S} |S|$ opciones distintas para definir *totalmente* a f. Visto que $|S| = d$, estas son d^d opciones.

Asimismo como $x_0 \in S$ y no es ninguno en particular, hay $|S| = d$ opciones para este elemento. Luego, la cantidad total de "casos posibles" es $d^d \times d = \boldsymbol{d^{d+1}}$.

*Ahora se calculará cuántos pares (f, x_0) pueden existir **con** las condiciones requeridas.*

Considerando que $|S| = d$, existen d opciones para elegir x_0. Puesto que debe ser $x_1 = f(x_0)$ y asimismo $x_1 \neq x_0$, en S quedan $d - 1$ elementos diferentes a x_0 para los cuales $f(x_0)$ podría tomar su valor. Fijado eso, ahora en S hay $d - 2$ elementos distintos a x_0 y a x_1 que podrían ser $f(x_1) = x_2$. Así sucesivamente, se ve que para $f(x_{\ell-1}) = x_\ell$ habrán $d - \ell$ opciones. Por lo tanto, existen $d\,(d-1)(d-2)\cdots(d-\ell)$ posibilidades para definir *parcialmente* una f tal que $x_0 \in S$, $x_{i+1} = f(x_i)$ y los $x_0, x_1, \ldots, x_\ell$ son todos diferentes.

De esta manera, solo resta asignar el valor funcional de los elementos en S sobre los cuales no hay restricción alguna. En el proceso anterior se habrán de definir los valores de $f(x_i)$ para todo $i \in \{0, \ldots, \ell - 1\}$. Luego, teniendo en cuenta que $|S| = d$, los elementos de S se pueden identificar así:

$$\overbrace{\underbrace{0, \ldots, \ell - 1}_{\substack{\text{valor de } f \\ \text{asignado}}}, \underbrace{\ell, \ldots, d - 1}_{\substack{\text{valor de } f \\ \textbf{no} \text{ asignado}}}}^{d \text{ elementos en } S}$$

Por ende, se observa que los elementos por asignar son $(d-1) - \ell + 1 = d - \ell$ en total. Y como sobre estos no aplican condiciones, para cada uno hay d opciones en S, entonces se

tienen $d^{d-\ell}$ posibilidades. Por lo tanto, la cantidad de "casos favorables" termina siendo $d\,(d-1)(d-2)\cdots(d-\ell) \times d^{d-\ell} = \boldsymbol{d^{d-\ell}\prod_{i=0}^{\ell}(d-i)}$.

Por último, se muestra que se cumple la tesis para la proporción deseada.

De acuerdo a lo deducido previamente, la proporción que se trata de acotar es:

$$\begin{aligned} r = \frac{d^{d-\ell}\prod_{i=0}^{\ell}(d-i)}{d^{d+1}} &= d^{d-\ell-(d+1)}\prod_{i=0}^{\ell}(d-i) \\ &= d^{-\ell-1}\prod_{i=0}^{\ell} d\left(1-\frac{i}{d}\right) \\ &= d^{-\ell-1}d^{\ell-0+1}\prod_{i=0}^{\ell}\left(1-\frac{i}{d}\right) \\ &= \prod_{i=0}^{\ell}\left(1-\frac{i}{d}\right) \end{aligned}$$

Nótese que $r < e^{-\lambda} \iff \log(r) < -\lambda$, y esta última cota vale por la siguiente cadena de desigualdades:

$$\begin{aligned} \log(r) = \log\left(\prod_{i=0}^{\ell}\left(1-\frac{i}{d}\right)\right) = \sum_{i=0}^{\ell}\log\left(1-\frac{i}{d}\right) &< \sum_{i=0}^{\ell} -\frac{i}{d} \\ &= -\frac{1}{d}\sum_{i=0}^{\ell} i \\ &= -\frac{1}{d}\cdot\frac{\ell(\ell+1)}{2} \\ &= -\frac{\ell(\ell+1)}{2d} \\ &< -\frac{\ell^2}{2d} \\ &= -\frac{\left(1+\lceil\sqrt{2\lambda d}\,\rceil\right)^2}{2d} \\ &< -\frac{\lceil\sqrt{2\lambda d}\,\rceil^2}{2d} \\ &\leq -\frac{2\lambda d}{2d} \\ &= -\lambda \end{aligned}$$

La primera desigualdad se da por el hecho analítico de que todo $x \in (0,1)$ verifica que

$\log(1-x) < -x$. Por su lado, la segunda desigualdad se debe a que como $\ell > 0$, sucede que $\ell(\ell+1) = \ell^2 + \ell > \ell^2$, con lo cual debe ocurrir que $-\ell(\ell+1) < -\ell^2$.

■

La proposición anterior, llevándola al caso del método donde $S = \mathbb{Z}_d$ y $d > 1$ es un divisor de n, dice que la probabilidad de que el polinomio f y el valor inicial x_0 empleados **no** generen un x_i repetido módulo d para $i \geq 1 + \lceil \sqrt{2\lambda d} \rceil$, es menor a $e^{-\lambda}$. O sea, visto que $\lceil \sqrt{2\lambda d} \rceil \geq \sqrt{2\lambda d}$, la probabilidad de que el primer valor repetido módulo d ocurra con $i \geq 1 + \sqrt{2\lambda d}$ es **exponencialmente chica**.

En otras palabras, con probabilidad mayor a $1 - e^{-\lambda}$ el primer índice i_0 para el cual existe un $j_0 < i_0$ tal que $x_{i_0} \equiv x_{j_0} \pmod{d}$, va a cumplir que $i_0 < 1 + \sqrt{2\lambda d}$. Sin embargo, el índice j que emplea el método rho de Pollard no es uno cualquiera (recordar el tercer punto de la descripción general), con lo cual no necesariamente será en este i_0 donde se active la condición de parada. De todas formas, con el próximo lema se verá que esto no supone mayores dificultades.

Lema 4.2. *Sean n un número compuesto, $d > 1$ un divisor de n y $f : \mathbb{Z}_d \to \mathbb{Z}_d$ / $f(x_k) = x_{k+1}$. En esas condiciones, existe un par de índices i_0 y j_0 tales que $j_0 < i_0$ y $x_{i_0} \equiv x_{j_0} \pmod{d}$, y para cualquier par de índices i y j tales que $i - j = i_0 - j_0$ se tiene que $x_i \equiv x_j \pmod{d}$.*

Demostración: (Debida a [Kob94, p. 140])

La existencia de los índices i_0 y j_0 tales que $j_0 < i_0$ y $x_{i_0} \equiv x_{j_0} \pmod{d}$, es clara por el hecho de que $\mathbb{Z}_d$ es un conjunto finito. Luego, si i y j son tales que $i - j = i_0 - j_0$, de esto último se desprende que $\exists m \in \mathbb{N} \ / \ i = i_0 + m \ \wedge \ j = j_0 + m$. Entonces con la congruencia se puede trabajar de la siguiente manera:

$$\begin{aligned} x_{i_0} \equiv x_{j_0} \pmod{d} &\overset{f}{\Rightarrow} x_{i_0+1} \equiv x_{j_0+1} \pmod{d} \\ &\overset{f}{\Rightarrow} x_{i_0+2} \equiv x_{j_0+2} \pmod{d} \\ &\vdots \\ &\overset{f}{\Rightarrow} x_{i_0+m} \equiv x_{j_0+m} \pmod{d} \\ &\Rightarrow x_i \equiv x_j \pmod{d} \end{aligned}$$

■

Luego, si i_0 es un índice de $h-1$ bits y asimismo es el primero para el cual existe un $j_0 < i_0$ tal que $x_{i_0} \equiv x_{j_0} \pmod{d}$, considerando que el mayor entero de $h-1$ bits

es $j = 2^{h-1} - 1$, por el lema previo se tiene que para $i = j + i_0 - j_0$ también valdrá la congruencia $x_i \equiv x_j \pmod{d}$. Como $i_0 - j_0 > 0$, sucede que $i > j$, entonces i tiene h bits y para cuando el Algoritmo 4.7 llegue a esa iteración va a estar recordando x_j. De esa forma, el método va a hacer el $\gcd(x_i - x_j,\, n)$ y por la última congruencia es seguro que el resultado será al menos $d > 1$, provocando que el mismo termine.

Notar que si i consta de h bits, entonces $i < 2^h = 4 \,.\, 2^{h-2}$. Además si i_0 tiene $h-1$ bits, entonces $2^{h-2} \leq i_0 < 2^{h-1}$. Por lo tanto, $i < 4 \,.\, 2^{h-2} \leq 4\, i_0 \;\Rightarrow\; i < 4\, i_0$.

Así se llega al siguiente teorema que concluye el tiempo de ejecución del método rho de Pollard con alta fiabilidad:

Teorema 4.3. *Sean n un entero de k bits compuesto e impar, d un divisor de n tal que $1 < d < \sqrt{n}$ y $\lambda \in \mathbb{R}^+$. Si un par (f, x_0) que consiste de un polinomio f con coeficientes enteros y de un valor inicial x_0 entero, es elegido de forma tal que se comporta como un par "promedio" en el sentido de la Proposición 4.1, entonces con probabilidad mayor a $1 - e^{-\lambda}$ el Algoritmo 4.7 retorna d en tiempo $O\Big(\sqrt[4]{2}^{\,k} k\Big)$.*

Demostración:

Como ya fue mencionado, por la **Proposición 4.1** se tiene para la configuración (f, x_0) empleada que, con probabilidad mayor a $1 - e^{-\lambda}$, el primer índice i_0 para el cual existe un $j_0 < i_0$ tal que $x_{i_0} \equiv x_{j_0} \pmod{d}$, va a cumplir que $i_0 < 1 + \sqrt{2\lambda d}$.

Adicionalmente, también se ha visto que el índice i que provoca la terminación del bloque WHILE del Algoritmo 4.7, mantiene la relación $i < 4\, i_0$ con respecto al anterior.

Por lo tanto, con probabilidad $1 - e^{-\lambda}$ la cantidad de iteraciones del WHILE se encuentra acotada por $i < 4(1 + \sqrt{2\lambda d}) = 4 + 4\sqrt{2\lambda}\sqrt{d}$. Además por la hipótesis de que $d < \sqrt{n}$, también se cumple que $\sqrt{d} < \sqrt[4]{n}$, con lo cual $i < 4 + 4\sqrt{2\lambda}\sqrt[4]{n}$. Por último, el valor máximo para n de k bits es $2^k - 1$, entonces con la mencionada probabilidad $i < 4 + 4\sqrt{2\lambda}\sqrt[4]{2^k - 1}$.

Luego, con probabilidad $1 - e^{-\lambda}$ la cantidad de iteraciones del WHILE está asintóticamente acotada por $O\Big(\sqrt[4]{2^k}\Big) = O\Big(\sqrt[4]{2}^{\,k}\Big)$. Finalmente, en virtud de que la instrucción más costosa dentro del bloque iterativo es el cálculo de un máximo común divisor que involucra a n, y considerando que dicho cálculo insume tiempo $O(k)$ si la aritmética tiene costo unitario, se concluye que el tiempo de ejecución total, en términos de operaciones elementales, es $O\Big(\sqrt[4]{2}^{\,k} k\Big)$. ■

Como se observa, el método rho de Pollard aún en su versión más optimista lleva tiempo exponencial. Sin embargo, dado que $O\left(\sqrt[4]{2}^k k\right) \subset O\left(\sqrt{2}^k\right)$, este realmente constituye una mejora con respecto a la técnica *naíf*. Asimismo, este método tiene el mérito de haber sido utilizado para factorizar el octavo *número de Fermat* [7], del cual –para entonces– ya se tenía la certeza de que es compuesto [BP81].

En su publicación inicial, Pollard empleó $f(x) \equiv x^2 - 1 \pmod{n}$ suponiendo que este polinomio se comporta como un "mapa aleatorio" (en el sentido de la Proposición 4.1), y sobre el final de su artículo planteó las siguientes conjeturas:

- Los polinomios de la familia $x^2 + b$ *parecen* comportarse como "mapas aleatorios" (en el sentido de la Proposición 4.1), excepto para los casos de x^2 y $x^2 - 2$ (independientemente del x_0).
- Si se sabe que todos los factores primos p de n verifican que $p \equiv 1 \pmod{k}$ para cierto $k > 2$, entonces *podría* resultar mejor utilizar los polinomios de la familia $x^k + b$.

A la fecha del presente trabajo estas heurísticas continúan sin haber sido demostradas, no obstante, las mismas han inspirado numerosas líneas de investigación en el área de sistemas dinámicos sobre cuerpos finitos. En el octavo capítulo de [SW19] el lector podrá encontrar un relevamiento de los principales resultados en esta temática, los cuales fueron motivados en gran parte por la obra de Pollard. Actualmente se cree que la mayoría de los polinomios se comportan "aleatoriamente" (en el sentido de la Proposición 4.1) y pueden utilizarse en el Algoritmo 4.7. Sin embargo, hay familias de polinomios que poseen un alto grado de regularidad probado y su uso no es recomendado, como por ejemplo los monomios x^n y los *polinomios de Chebyshev* ($x^2 - 2$ es uno de estos). Para ver una descripción completa de la dinámica de estos últimos cf. [QP19].

4.2. Método de curvas elípticas de Lenstra

A mediados de la década de 1980 Hendrik Lenstra circuló una prepublicación titulada *Factoring integers with elliptic curves*, en la que por primera vez se presentaba una "aplicación práctica" de las curvas elípticas, concretamente para factorizar enteros. Es así que la técnica que será expuesta en esta subsección definió otro mojón en la historia de los algoritmos de factorización, dado que –como se verá más adelante– no solo mejoró la cota superior asintótica del *método rho de Pollard*, sino que además sirvió de inspiración tanto a Koblitz para inventar la criptografía basada en curvas elípticas [Kob87],

[7] Los números de Fermat son los de la forma $\mathcal{F}_n = 2^{2^n} + 1$, con $n \in \mathbb{N}$.

como a Goldwasser y Kilian para idear el test de primalidad visto en la Subsección 3.7.

Recordar de la Subsección 2.1.3 que una curva elíptica está definida sobre un cuerpo, porque de esa forma los inversos existen y las fórmulas para sumar puntos sobre ella tienen validez. Sin embargo, cuando n es compuesto, en $\mathbb{Z}_n$ hay elementos que no tienen inverso módulo n, con lo cual al trabajar las fórmulas de suma de puntos módulo n la operación podría no ser realizable. En este caso, calculando el máximo común divisor entre n y el valor no invertible (que sería lo que aparece en un denominador en alguna de las fórmulas) se obtendría un divisor de n mayor que uno. Más aún, si a los parámetros a y b que definen la presunta curva elíptica se les pide que $\gcd(4a^3 + 27b^2, n) = 1$, entonces para cualquier primo $p \mid n$ estos en efecto definen una curva elíptica sobre el cuerpo $\mathbb{Z}_p$. Por lo tanto, la esencia del método de factorización basado en curvas elípticas es tratar de hacer cálculos con puntos (x, y) que verifican $y^2 \equiv x^3 + ax + b \pmod{n}$ como si $\mathbb{Z}_n$ fuera un cuerpo, buscando chocar con el hecho de que en realidad no lo es.

Se recuerda que también podría pasar que las primeras coordenadas de los puntos involucrados en una suma sean iguales módulo n y las segundas coordenadas opuestas módulo n, en cuyo caso el resultado de la cuenta se define como el punto $\mathcal{O}$.

Tal como se hizo cuando se presentó el test de primalidad basado en este mismo objeto matemático, el algoritmo será descripto en muy alto nivel tratando de ser lo más imperativo posible, siguiendo como referencia lo expuesto en [Kob94, pp. 195-197].

Paso 1

Elegir aleatoriamente a, x_0, $y_0 \in \mathbb{Z}_n$ y establecer $b = {y_0}^2 - {x_0}^3 - ax_0 \pmod{n}$. Si resulta que $4a^3 + 27b^2 \pmod{n} = 0$, entonces repetir el sorteo aleatorio hasta conseguir una configuración de a y b en la que $4a^3 + 27b^2 \pmod{n} \neq 0$. Asimismo, si se cumple la condición anterior pero $d = \gcd(4a^3 + 27b^2, n) > 1$, entonces d es un divisor propio de n, con lo cual retornar d.

Por el contrario, si para los a y b sorteados se tiene que $4a^3 + 27b^2 \pmod{n} \neq 0$ y $\gcd(4a^3 + 27b^2, n) = 1$, se establece E como el conjunto de puntos que verifican la ecuación $y^2 \equiv x^3 + ax + b \pmod{n}$ y $P = (x_0, y_0)$ [8].

[8] Observar que por construcción $P \in E$.

Paso 2

Elegir una cota $B \in \mathbb{N}$ para la cual se conocen todos los números primos p menores o igual a B. Elegir una cota $C \in \mathbb{N}$ para determinar qué tanto se podrán exponenciar los primos p anteriores.

Paso 3

Definir $k = \prod_{\substack{p \leq B \\ p \text{ primo}}} p^{\alpha_p}$, donde α_p es el mayor entero tal que $p^{\alpha_p} \leq C$.

Paso 4

Intentar calcular kP de la siguiente manera:

Por definición se tiene que $k = 2^{\alpha_2}\, 3^{\alpha_3} \ldots p^{\alpha_p}$, siendo p el mayor número primo tal que $p \leq B$. Entonces, a partir de P, se procede multiplicando el último valor calculado por el escalar primo correspondiente, hasta alcanzar la potencia adecuada:

$$
\begin{array}{lll}
P \Rightarrow 2P & 2^{\alpha_2} P \Rightarrow 3(2^{\alpha_2} P) & \text{Y así sucesivamente} \\
\quad \Rightarrow 2(2P) = 2^2 P & \quad \Rightarrow 3(3\,.\,2^{\alpha_2} P) = 3^2\, 2^{\alpha_2} P & \text{hasta llegar a} \\
\quad \Rightarrow 2(2^2 P) = 2^3 P & \quad \Rightarrow 3(3^2\, 2^{\alpha_2} P) = 3^3\, 2^{\alpha_2} P & \underbrace{p^{\alpha_p} \ldots 3^{\alpha_3}\, 2^{\alpha_2}}_{k} P. \\
\quad \vdots & \quad \vdots & \\
\quad \Rightarrow 2(2^{\alpha_2 - 1} P) = 2^{\alpha_2} P & \quad \Rightarrow 3(3^{\alpha_3 - 1}\, 2^{\alpha_2} P) = 3^{\alpha_3}\, 2^{\alpha_2} P &
\end{array}
$$

En el intento de hallar un $\tilde{k}P$ "parcial" (*i. e.*, el producto hasta cierto $\tilde{k} \leq k$), al querer sumar los últimos dos puntos $R = (x_r, y_r)$ y $S = (x_s, y_s)$ en E tales que $R + S = \tilde{k}P$, primero se requerirá calcular $d = \gcd(\texttt{denominador}, n)$ para saber si `denominador` tiene inverso módulo n, siendo `denominador` igual a $x_r - x_s$ (si $R \neq S$) o igual a $2\, y_r$ (si $R = S$ y se emplea duplicación). Entonces se distinguen tres casos:

- Si $d = 1$, entonces `denominador` es invertible módulo n, con lo cual se puede efectuar la suma parcial. Por lo tanto, $\tilde{k}P$ queda resuelto y se debe continuar la multiplicación de este por el próximo escalar.
- Si $d = n$, entonces vale que `denominador` $(\text{mód } n) = 0$ y de esa forma la suma parcial $R + S = \mathcal{O}$. Por ende, $\tilde{k}P = \mathcal{O}$ y si se procede multiplicando esta igualdad por los números primos restantes, se llega a que $kP = \mathcal{O}$. Es así que en este caso el método puede calcular kP, con lo cual no se detecta un divisor de n. Ergo, volver al **Paso 1** para sortear una nueva configuración.

- Si $1 < d < n$, entonces `denominador` no tiene inverso módulo n, con lo cual la suma no puede ser efectuada. Luego, retornar d que es un divisor no trivial de n.

Paso 5

Haber llegado a esta fase significa que en el paso anterior se obtuvo $d = 1$ para todos los denominadores necesarios, lo cual implica que el proceso de multiplicación llegó a su fin, esto es, se calculó kP. Sin embargo, este valor termina siendo intrascendente, ya que en ningún momento fue posible deducir un divisor de n. Luego, habrá que volver al **Paso 1** para sortear nuevos parámetros.

Estos son todos los pasos que describen el ***método de curvas elípticas de Lenstra***.

En su mismo artículo, Lenstra demostró que en base a una conjetura "razonable" relacionada con la cota B, el tiempo de ejecución esperado para encontrar un divisor no trivial de n es $O\Big(e^{\sqrt{(2+\varepsilon)\log(p)\log(\log(p))}}\Big)$, donde $\boldsymbol{p}$ **es el menor número primo** que divide a n [Kob94, p. 198]. Esto trae aparejado varias implicancias, de las cuales se destacan las mencionadas a continuación.

En primera instancia observar que si p es el *menor* número primo que divide a n, entonces por la Proposición 3.1 se tiene que $p \leq \sqrt{n}$. Luego, $\log(p) \leq \log\left(n^{\frac{1}{2}}\right) = \frac{1}{2}\log(n)$ y entonces también vale que $\log\big(\log(p)\big) \leq \log\left(\frac{1}{2}\log(n)\right) < \log\big(\log(n)\big)$. Por lo tanto, $(2+\varepsilon)\log(p)\log\big(\log(p)\big) < (2+\varepsilon)\frac{1}{2}\log(n)\log\big(\log(n)\big) = (1+\epsilon)\log(n)\log\big(\log(n)\big)$, con $\epsilon = \frac{\varepsilon}{2}$. Consiguientemente se ve que el tiempo de ejecución esperado en función de la entrada es $O\Big(e^{\sqrt{(1+\epsilon)\log(n)\log(\log(n))}}\Big)$, lo cual es de carácter **subexponencial** y contrasta (para mejor) con el tiempo exponencial insumido por el método visto en la subsección previa. No obstante, un orden subexponencial sigue siendo ineficiente para grandes valores de n genéricos.

Sin perjuicio de lo anterior, si n es grande pero está compuesto por factores primos pequeños (en comparación con la $\sqrt{n}$), entonces el algoritmo que aquí se ha presentado posee un desempeño razonable. Esto tiene como consecuencia que el método de curvas elípticas de Lenstra sea empleado como fase preliminar de extracción de factores chicos, para luego pasar a otro algoritmo de factorización asintóticamente más rápido, solo con factores grandes.

4.3. Criba cuadrática

Para concluir con el estudio de los algoritmos de factorización de enteros, en la presente subsección será descripto uno de los más rápidos que actualmente se conocen. De hecho, este solo es superado por la llamada *Criba general del cuerpo de números*. Si bien se verá más adelante que la complejidad de esta nueva técnica es análoga al método de la subsección anterior (subexponencial), la diferencia fundamental con aquel es que el método aquí exhibido es más rápido cuando n solo se compone por factores grandes (en el sentido de que son muy cercanos a la $\sqrt{n}$ [9]).

El algoritmo que aquí compete tratar es conocido como la ***Criba cuadrática***, y fue inventado por Carl Pomerance en 1981 basándose en ideas previas de Kraitchik y Dixon. Desde ya se deja constancia que el contenido de esta subsección está basado íntegramente en el artículo [Lan01], por lo tanto, aquí no se empleará la notación algorítmica que ha sido característica a lo largo de esta tesis, sino que serán descriptas las ideas fundamentales del método.

El objetivo principal de la Criba cuadrática es encontrar dos números x e y tales que $x^2 \equiv y^2 \pmod{n}$ pero con $x \not\equiv \pm y \pmod{n}$. La gracia de esto radica en que si $n \mid x^2 - y^2 = (x-y)(x+y)$ pero al mismo tiempo ni $n \mid x-y$ ni $n \mid x+y$, entonces para que ocurra lo primero tanto $x-y$ como $x+y$ deben aportar factores primos de un modo tal que en su producto aparezcan (al menos) todos los de n. Esto implica que cuando se hayan encontrado los mencionados x e y, calculando el $\gcd(x-y, n)$ o también el $\gcd(x+y, n)$, el resultado que se obtiene es un divisor **no trivial** de n.

El primer paso en este sentido es definir $Q(x) = x^2 - n$, y luego aplicar esta función sobre una serie de valores $\{x_1, \ldots, x_k\}$ que a la brevedad serán especificados.

Luego, entre todos los $\{Q(x_1), \ldots, Q(x_k)\}$ hallados, se seleccionará un subconjunto $\{Q(x_{i_1}), \ldots, Q(x_{i_r})\}$ tal que el producto de todos sus elementos es un cuadrado perfecto, a saber, $y^2 = Q(x_{i_1}) \ldots Q(x_{i_r})$.

En última instancia, dado que para cualquier x genérico $Q(x) \equiv x^2 \pmod{n}$, se tiene que $Q(x_{i_1}) \ldots Q(x_{i_r}) \equiv {x_{i_1}}^2 \ldots {x_{i_r}}^2 \equiv (x_{i_1} \ldots x_{i_r})^2 \pmod{n}$. Entonces los valores que habrán de emplearse al calcular el máximo común divisor con n son $x = x_{i_1} \ldots x_{i_r}$ e $y = \sqrt{Q(x_{i_1}) \ldots Q(x_{i_r})}$, siempre y cuando se cumpla que $x \not\equiv \pm y \pmod{n}$.

[9] Típicamente el caso de las claves RSA.

Con lo anterior ha quedado descripto el funcionamiento general de la Criba. Por lo tanto, ahora se pasará a los detalles de la misma.

Para saber si el producto de algunos $Q(x_i)$ es un cuadrado perfecto, es claro que basta con mirar que en la descomposición factorial de tal producto, todos los exponentes de los factores primos sean pares. Luego, visto que para determinar lo anterior será necesario disponer de la factorización de cada $Q(x_i)$, lo más conveniente [10] será que estos valores sean *pequeños* y asimismo sus divisores primos estén dentro de un conjunto preestablecido de números primos acotados, el cual es llamado *base de factores*.

Llegado a este punto surgen dos preguntas obvias: ¿qué se entiende por $Q(x_i)$ pequeño? y ¿cómo se conforma la base de factores?

Para responder la primera, notar que si x_i es *cercano* a $\sqrt{n}$, entonces $Q(x_i)$ también es *cercano* a $\sqrt{n}^2 - n = 0$, *i. e.*, $Q(x_i)$ es chico. Y para dejar claro qué significa "x_i cercano a $\sqrt{n}$", se define una cota M y se usa $x_i = \lfloor \sqrt{n} \rfloor + i$ para todo $i \in \{1, \ldots, M\}$ [11].

En cuanto a la segunda pregunta, un número primo $p \,|\, Q(x) \iff p \,|\, x^2 - n \iff x^2 \equiv n \pmod{p}$, y esto significa que n es lo que se conoce como un *residuo cuadrático módulo p*. Entonces, para los primos que se vayan a incluir en la base de factores tiene que ocurrir eso. Además, tratando de que la base no sea demasiado exhaustiva, también se pide que los números primos anteriores estén por debajo de cierta cota B [12].

Así es que el proceso de cribado va a descartar aquellos x_i (con la forma que se han definido) para los cuales $Q(x_i)$ tiene algún divisor primo por fuera de la base de factores construida para n.

El siguiente paso consiste en armar una matriz binaria A, que en la celda (i, j) contiene la paridad del exponente del j-ésimo elemento de la base en la factorización de $Q(x_i)$, siendo x_i un "sobreviviente" del proceso de cribado.

[10] Para no agregarle complejidad extra al método.

[11] Según el artículo estudiado, el valor óptimo para M sería aproximadamente $\left(e^{\sqrt{\log(n)\log(\log(n))}}\right)^{\frac{3\sqrt{2}}{4}}$, con lo cual tomando el entero más cercano a eso, M ya queda establecida.

[12] Según el artículo estudiado, el valor óptimo para B sería aproximadamente $\left(e^{\sqrt{\log(n)\log(\log(n))}}\right)^{\frac{\sqrt{2}}{4}}$.

	2	3	13	17	19	29
i	1	1	0	0	1	0

Cuadro 4.1: Ejemplo de la i-ésima fila de la matriz A para $Q(x_i) = 2 \,.\, 3 \,.\, 17^2 \,.\, 19$ y una base de factores $\{2, 3, 13, 17, 19, 29\}$

Dado que se requiere que el producto de los $Q(x_i)$ empleados sea un cuadrado perfecto, la suma de los exponentes de cada elemento en la base de factores tiene que ser par. Luego, para saber con cuáles $Q(x_i)$ uno tendría que quedarse, si $\vec{a_i}$ corresponde a la i-ésima fila de la matriz A, entonces se precisa resolver $\vec{a_1}e_1 + \cdots + \vec{a_k}e_k \equiv 0 \pmod 2$ [13] para $e_i \in \{0, 1\}$ (pues $Q(x_i)$ se incluye en el producto si y solo si en la solución $e_i = 1$).

Finalmente, definiendo el vector incógnita $\vec{e} = (e_1, \ldots, e_k)$, la ecuación congruencial anterior es equivalente a resolver el sistema lineal $\vec{e}A \equiv 0 \pmod 2$. Este sistema se puede atacar con el método de eliminación gaussiana implementado por los algoritmos de Wiedemann y Lanczos (los cuales trabajan sobre cuerpos finitos), y de esa forma obtener un conjunto generador del espacio de soluciones. Luego, cada elemento del conjunto generador determina un subconjunto de valores $Q(x_i)$ cuyo producto es un cuadrado perfecto. Así termina la descripción detallada de la Criba cuadrática.

Por último, en cuanto al tiempo de ejecución de este método, se puede mencionar que el mismo queda determinado por la cota B para el tamaño de la base de factores. Luego, considerando el costo de la eliminación gaussiana, la Criba cuadrática insume tiempo subexponencial $O\left(e^{(1+\varepsilon)\sqrt{\log(n)\log(\log(n))}}\right)$.

[13] La notación significa que cada coordenada del vector suma tiene que ser congruente con 0 módulo 2.

5. Conclusiones y trabajo a futuro

El primer objetivo específico del proyecto era relevar test de primalidad probabilísticos, estudiando detalladamente al menos uno de ellos. En este sentido es posible afirmar que dicha meta se cumplió con creces, dado que en total fueron estudiados tres test de primalidad probabilísticos y todos con una profundidad importante. Específicamente, los test analizados son los de: *Fermat*, *Miller-Rabin* y *Goldwasser-Kilian*. En el estudio de estos algoritmos quedó en evidencia la importancia de poder generar valores aleatorios o pseudoaleatorios con eficiencia, y en la Subsección 3.3 se presentaron los generadores de congruencia lineal como una opción viable. Luego, una segunda parte que tenía este objetivo sobre realizar un leve acercamiento a la generación de números pseudoaleatorios, también se puede dar por cumplida.

Yendo a los resultados concretos de cada test, en cuanto al test de Fermat se puede decir que el mismo es sumamente sencillo de describir, se concluyó que efectúa una cantidad de pasos lineal en el tamaño del número n a testear, y si n no es uno de los llamados "números de Carmichael", la probabilidad de obtener un falso positivo se puede hacer exponencialmente chica (ejecutando el test varias veces con el mismo n).

Por su parte, el test de Miller-Rabin se puede interpretar como un test de Fermat "reforzado", ya que si bien la descripción del mismo no es tan intuitiva a primera vista, en esencia los dos son muy parecidos. Según lo que se ha concluido para este test, su fortaleza radica en que manteniendo el orden lineal en la cantidad de pasos, consigue que la probabilidad de obtener un falso positivo se pueda hacer negligible para *cualquier* entrada. De hecho, incluso es capaz de brindar esta facilidad con menos iteraciones de las que serían necesarias para obtener una confianza similar con el test de Fermat. Como consecuencia de todo lo anterior, se pudo averiguar que algunas célebres bibliotecas criptográficas emplean este test en sus implementaciones del protocolo RSA.

En contraste con los dos algoritmos probabilísticos anteriores que su salida puede ser `PSEUDO-PRIMO`, para el test de Goldwasser-Kilian se probó que cuando el mismo detecta que la entrada es un número primo, efectivamente su salida es `PRIMO`. Por lo tanto, la principal característica de este test probabilístico es que sirve para *certificar* primalidad. Otra diferencia importante observada con respecto a los anteriores, es que mientras los fundamentos de aquellos están en la teoría elemental de números, este tercer test se encuentra basado en las curvas elípticas. Esto tiene dos consecuencias directas: primero que la descripción del algoritmo se vuelve más intrincada, y segundo que la inocente aritmética modular (que para números de enorme tamaño podría decirse que

tiene una complejidad cuadrática) ahora se reemplaza por algoritmos más laboriosos (como el de conteo de puntos, que tiene orden polinomial de grado ocho).

Para tener una idea del costo de la aritmética modular en la práctica, se ejecutó varias veces el comando `genrsa` de OpenSSL con diferentes tamaños de clave, en una computadora con las siguientes especificaciones:

- Modelo: OMEN by HP Laptop 17-an0xx
- Sistema operativo: Windows 10 Home Single Language
- Memoria RAM: 16 GB
- Procesador: Intel Core i7-7700HQ CPU @ 2.80 GHz, 4 núcleos y 8 procesadores lógicos
- Arquitectura: 64 bits
- Virtualización: Tecnología Intel VT-x
- Hipervisor: Windows Subsystem for Linux

Los promedios de los tiempos medidos para diez ejecuciones de dicha función, fueron los siguientes:

2048 bits	**3072 bits**	**4096 bits**	**7680 bits**
0,113 s	0,380 s	0,800 s	7,350 s

Cuadro 5.1: Tiempo promedio en segundos para generar una clave privada de RSA con la función `genrsa`, según el tamaño del módulo indicado

Entonces se observa que hasta 4096 bits, todo el proceso de generar la clave (no solo ejecutar el test de Miller-Rabin) demora menos que un segundo. Luego, en virtud de que el largo de las claves recomendado a la fecha es 2048 bits, se concluye que asumir las operaciones modulares como "elementales" (*i. e.*, de tiempo constante) es razonable.

El segundo objetivo específico de este proyecto consistía en estudiar detalladamente el *test de primalidad AKS*, y compararlo con otros algoritmos con igual propósito que ya existían antes de su publicación. En este caso cuando se dijo que el relevamiento buscaba ser "detallado", además de que obviamente estuviera presente la descripción del algoritmo, se pretendía incluir tanto la demostración "paso a paso" del teorema que garantiza la correctitud del método, como el análisis del tiempo de ejecución. De estas tres cosas no fue posible incluir en este informe la demostración con el nivel de detalle que sí se llegó a comprender. Por ende, se afirma que esta meta fue alcanzada parcialmente; ver más adelante el trabajo a futuro.

Además de los tres algoritmos probabilísticos mencionados más arriba, en este trabajo se ha incluido un par de test deterministas y de propósito general pero con tiempo de ejecución exponencial (Subsecciones 3.1 y 3.8), así como un test determinista y de tiempo polinomial pero específico (Subsección 3.9). También se vio cómo el test de Miller-Rabin podría modificarse para ser determinista, pero a cambio de que su correctitud quede condicionada a un resultado matemático que en realidad no se ha demostrado. Por lo tanto, con todo lo visto en las subsecciones previas a la del test AKS, fue posible concluir que este algoritmo constituyó el primer test de primalidad en satisfacer simultáneamente: ser determinista, su tiempo de ejecución es polinomial, su correctitud está incondicionada y ser de propósito general.

Luego, se deduce que el test AKS marcó un hito en la teoría de la complejidad computacional, pues demostró que el problema de decidir si un número es primo, pertenece a la clase P. Sin embargo, en el análisis de su tiempo de ejecución se vio que el polinomio que acota la cantidad de operaciones en función del tamaño de la entrada, tiene exponente $^{21}/_{2}$. Esto tiene como consecuencia que en la práctica se prefiera seguir usando los algoritmos probabilísticos (como el test de Miller-Rabin) que con muy alta fiabilidad resuelven la tarea a un costo asintótico substancialmente menor.

Finalmente, el tercer objetivo específico que fue definido para este proyecto, se trataba de estudiar detalladamente algoritmos de factorización basados en diferentes objetos matemáticos, y asimismo relevar un algoritmo de cribado avanzado. Visto que se ha explicado la *Criba cuadrática*, y considerando que se ha estudiado el *método rho de Pollard* (que se basa en un sistema dinámico discreto no lineal), así como el *método de Lenstra* (basado en curvas elípticas), es posible sostener que este objetivo fue cumplido.

Sobre el método rho de Pollard, se pudo probar que es capaz de mejorar la complejidad del *método naíf*, pero la misma sigue siendo exponencial. El método de Lenstra fue la primera aplicación práctica de las curvas elípticas, y además de haber motivado a otros investigadores a desarrollar el área, hoy en día es el preferido para factorizar números con factores primos de hasta cierto tamaño por su tiempo de ejecución subexponencial. Por último, se pudo relevar que la Criba cuadrática también posee un tiempo de ejecución subexponencial, pero funciona mejor para números con factores primos de cualquier tamaño.

Dentro del **trabajo a futuro** se puede resaltar lo que sigue.

Demostración "paso a paso" de la correctitud del test AKS

El segundo objetivo específico del proyecto hacía énfasis en estudiar *detalladamente* el test AKS. Si bien es cierto que se llegó a comprender integralmente la demostración de la correctitud de este algoritmo, debido a la extensión de dicha prueba y al hecho de que las herramientas de álgebra abstracta que requiere habrían implicado tener que expandir la sección de fundamento teórico, no fue posible transcribirla dentro del plazo que se disponía para culminar esta versión del informe del Proyecto de Grado, que es dedicada a la carrera Ingeniería en Computación. No obstante, en virtud de que en la fecha actual todavía se dispone de plazo para finalizar la versión del informe que será dedicada a la Licenciatura en Matemática, está planificado incluir esta importante demostración en dicho documento. Luego, al corto plazo este objetivo quedará cumplido "al pie de la letra".

Implementaciones y comparaciones

En el cuarto objetivo específico de este proyecto se había definido implementar algunos de los algoritmos que serían presentados, y luego comparar sus tiempos de ejecución con entradas seleccionadas. Si bien se había dejado claro que esto no era un punto fundamental para este Proyecto de Grado, la intención siempre fue llegar a concretarlo. Por ende, se considera que este ítem no pudo ser cumplido por agotamiento del tiempo pautado. Sin embargo, como este objetivo siempre se tuvo "en la mira" a pesar de no ser prioritario, la gran mayoría de los algoritmos que fueron estudiados han sido exhibidos con un pseudocódigo muy cercano a lo que sería su implementación en un lenguaje de programación real. Luego, se estima que si se retomara esta línea de investigación en el futuro, con el trabajo hecho en esta tesis podría ser fácil concretar este objetivo.

Criba general del cuerpo de números

Actualmente esta criba es el mejor algoritmo de propósito general para factorizar enteros. Entonces, teniendo en cuenta que a partir del relevamiento hecho ya se cuenta con una importante base de métodos cuya complejidad conceptual es menor, en caso de llevar a cabo nuevas investigaciones o relevamientos, sería conveniente que esta avanzada técnica de cribado fuera incluida en tal proyecto.

Generación de números pseudoaleatorios

Como se ha visto a lo largo de este trabajo, poder generar números pseudoaleatorios con eficiencia es de vital importancia para muchos algoritmos probabilísticos que son ampliamente usados en la práctica. Y si bien en el presente proyecto se cumplió el objetivo de hacer un *leve* acercamiento a este tema, el área en sí misma tiene repercusión

mucho más allá de los temas centrales de esta tesis (muy particularmente en aplicaciones criptográficas concretas). Luego, a través de los generadores de números pseudoaleatorios, se abre un camino para continuar investigando en áreas fundamentales.

Software de GIMPS

Dado que en este Proyecto de Grado lo primordial son los algoritmos generales, en ningún momento fue imperioso experimentar con el programa de GIMPS que busca nuevos números primos de Mersenne. Sin embargo, luego de haber dedicado una subsección entera a este proyecto de computación distribuida, es de interés instalar el software necesario para conocer de primera mano este proyecto colaborativo.

Referencias

[AGP94] W. R. Alford, A. Granville, and C. Pomerance. There are infinitely many Carmichael numbers. Annals of Mathematics, 139(3):703–722, 1994.

[AKS04] M. Agrawal, N. Kayal, and N. Saxena. PRIMES is in P. Annals of Mathematics, 160(2):781–793, 2004.

[Bac90] Eric Bach. Explicit bounds for primality testing and related problems. Mathematics of Computation, 55(191):355–380, 1990.

[Bar20] Elaine Barker. Recommendation for Key Management: Part 1 – General. National Institute of Standards and Technology. Special Publication 800-57 Revision 5, 2020.

[BB97] Gilles Brassard and Paul Bratley. Fundamentos de Algoritmia. Pretince Hall. Pearson, 1997.

[BGG+20] F. Boudot, P Gaudry, A. Guillevic, N. Heninger, E. Thomé, and P. Zimmermann. Comparing the Difficulty of Factorization and Discrete Logarithm: a 240-digit Experiment. Advances in Cryptology – CRYPTO 2020, 12172(1):62–91, 2020.

[BMST13] F. E. Brochero, C. G. Moreira, N. C. Saldanha, and E. Tengan. Teoria dos números: um passeio com primos e outros números familiares pelo mundo inteiro. Projeto Euclides. IMPA, 3rd edition, 2013.

[BP81] Richard P. Brent and John M. Pollard. Factorization of the Eighth Fermat Number. Mathematics of Computation, 36(154):627–630, 1981.

[BRS+10] L. E. Bassham, A. L. Rukhin, J. Soto, J. R. Nechvatal, M. E. Smid, S. D. Leigh, M. Levenson, M. Vangel, N. A. Heckert, and D. L. Banks. A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications. National Institute of Standards and Technology. Special Publication 800-22 Revision 1a, 2010.

[DH76] Whitfield Diffie and Martin Hellman. New Directions in Cryptography. IEEE Transactions on Information Theory, 22(6):644–654, 1976.

[GK86] Shafi Goldwasser and Joe Kilian. Almost All Primes Can be Quickly Certified. In Proceedings of the Eighteenth Annual ACM Symposium on Theory of

Computing, STOC '86, pages 316–329. Association for Computing Machinery, 1986.

[Hun14] Thomas W. Hungerford. Abstract Algebra: An Introduction. Brooks/Cole. Cengage Learning, 2014.

[Knu98] Donald Knuth. The Art of Computer Programming, Volume 2: Seminumerical Algorithms. Addison–Wesley, 1998.

[Kob87] Neil Koblitz. Elliptic Curve Cryptosystems. Mathematics of Computation, 48(177):203–209, 1987.

[Kob94] Neil Koblitz. A Course in Number Theory and Cryptography. Graduate Texts in Mathematics 114. Springer-Verlag, 1994.

[Lan01] Eric Landquist. The Quadratic Sieve Factoring Algorithm. MATH 488: Cryptographic Algorithms, 2001.

[Mer] Mersenne Research, Inc. Great Internet Mersenne Prime Search.
`https://www.mersenne.org/`.
Accedido el 8 de diciembre de 2021.

[Mil76] Gary L. Miller. Riemann's Hypothesis and Tests for Primality. Journal of Computer and System Sciences, 13(3):300–317, 1976.

[OPA] The OpenSSL Project Authors. genrsa.
`https://www.openssl.org/docs/man1.1.1/man1/genrsa.html`.
Accedido el 6 de diciembre de 2021.

[Ope] OpenVPN Technologies, Inc. What is OpenVPN?
`https://openvpn.net/faq/what-is-openvpn/`.
Accedido el 6 de diciembre de 2021.

[PM08] Valentín Petrov and Ernesto Mordecki. Teoría de la Probabilidad. DIRAC, 2008.

[Pol75] John M. Pollard. A Monte Carlo method for factorization. BIT Numerical Mathematics, 15(3):331–334, 1975.

[QP19] Claudio Qureshi and Daniel Panario. The graph structure of Chebyshev polynomials over finite fields and applications. Designs, Codes and Cryptography, 87(2):393–416, 2019.

[Rab80] Michael O. Rabin. Probabilistic Algorithm for Testing Primality. Journal of Number Theory, 12(1):128–138, 1980.

[Res14] Sidney I. Resnick. A Probability Path. Modern Birkhäuser Classics. Birkhäuser, 2014.

[RSA78] R. L. Rivest, A. Shamir, and L. Adleman. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. Communications of the ACM, 21(2):120–126, 1978.

[Sch08] René Schoof. Four primality testing algorithms. Mathematical Sciences Research Institute Publications, 44(1):101–126, 2008.

[Sha94] Jeffrey Shallit. Origins of the Analysis of the Euclidean Algorithm. Historia Mathematica, 21(4):401–419, 1994.

[Sho97] Peter Shor. Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer. SIAM Journal of Computing, 26(5):1484–1509, 1997.

[SS77] Robert Solovay and Volker Strassen. A Fast Monte-Carlo Test for Primality. SIAM Journal on Computing, 6(1):84–85, 1977.

[Ste09] William Stein. Elementary Number Theory: Primes, Congruences, and Secrets. Undergraduate Texts in Mathematics. Springer, 2009.

[SW17] Jonathan Sorenson and Jonathan Webster. Strong pseudoprimes to twelve prime bases. Mathematics of Computation, 86(304):985–1003, 2017.

[SW19] Kai-Uwe Schmidt and Arne Winterhof, editors. Combinatorics and Finite Fields: Difference Sets, Polynomials, Pseudorandomness and Applications. De Gruyter, 2019.

[vzG15] Joachim von zur Gathen. CryptoSchool. Springer, 2015.