Criptografía RSA: fundamentos y desarrollo

Teresa Joven

Trabajo fin de Grado en Matemáticas

Universidad de Zaragoza

Abstract

Information and communication technologies have enabled a digital environment where billions of social, cultural and economic interactions between entities located all over the world take place every day.

The soundness of the foundations of such an environment relies on its ability to provide adequate responses to security threats. Frequently, such interactions imply transmitting sensitive information; its disclosure to a non-legitimate agent would jeopardise commercial or even personal interests. Moreover, digital transactions must be as secure as “physical.ºnes; although the involved parties cannot see their counterpart’s face, shake their hands or witness how they sign a contract, they need to have the same level of assurance they enjoyed in the “classical"transactions. Thus confidentiality, authenticity and non-repudiation are essential to the digital security.

Cryptography and digital signatures provide a response to such needs. Since the pioneering work by Diffie and Hellman [8], many solutions have been proposed. Very likely, the most widely known of them is the RSA public-key cryptosystem [14]. Other fascinating techniques are the Zero Knowledge Proofs of knowledge, where an entity proves its identity by demonstrating the knowledge of a secret. For an example of Zero Knowledge Proof authentication scheme, see [9].

The soundness of the above-mentioned techniques is supported by number theory. Many of such techniques exploit the well-known fact that no one has yet found an algorithm which can factor a large integer (with hundreds of digits) in a reasonable amount of time. Their strength relies on the ability to compute large primes.

In this paper place ourselves in this context, from the computational point of view. We focus con the analysis and design of solutions, as well as with algorithmics. We have made the effort of consulting the original papers (some of the most relevant articles in the subject), such as they were published.

Se lo agradezco a mi director
José Carlos Ciria Cosculluela

Índice general

Capítulo 1

Introducción

1.1. Objetivos

Este trabajo trata de profundizar en aspectos relacionados con las asignaturas de Informática, en especial, en análisis y diseño de una aplicación informática. También se profundizará en el estudio de la complejidad de los algoritmos diseñados.
Además, pretende enseñarnos a navegar entre la bibliografía, acudiendo siempre a la búsqueda y lectura de las fuentes originales (los artículos pioneros).

El contexto en que hemos hecho esto es el de Criptografía RSA que describimos en la siguiente sección.

1.2. Contexto

Las tecnologías de la información y las comunicaciones han hecho posible un entorno digital donde a diario tienen lugar millones de interacciones de todo tipo (sociales, culturales, comerciales).

Dicho entorno está expuesto a serios problemas de seguridad. Su solidez se basa en su capacidad para dar respuestas a esos problemas de seguridad. Con frecuencia esas interacciones suponen la transmisión de información sensible; su revelación podría poner en riesgo intereses comerciales y personales. Más aún, las transacciones digitales deben ser tan seguras como las "físicas"; aun cuando las partes implicadas no tengan contacto físico entre sí (no se ven las caras, no pueden estrecharse las manos ni ver cómo firman contratos), necesitan tener el mismo nivel de seguridad que las clásicas. La confidencialidad, la autenticidad y el no repudio son esenciales para la seguridad digital.

La criptografía y las firmas digitales dan respuesta a esas necesidades. Desde el trabajo pionero de Diffie y Hellman [8], se han propuesto numerosas soluciones en esos ámbitos. Quizá la más conocida de ellas es el sistema criptográfico RSA [14].

Otras técnicas, igualmente fascinantes, son las Pruebas de Conocimiento Cero (*ZKP*, *Zero Knowledge Proofs*), mediante las cuales una entidad demuestra su identidad probando que conoce un secreto. En esquemas basados en contraseñas, la entidad que desea autenticarse y el verificador comparten un secreto (la contraseña). Al autenticarse, la entidad comunica su contraseña al verificador. Esa comunicación puede ser interceptada. El interceptor (o peor aún, el verificador) podrían usar indebidamente la contraseña para hacerse pasar por la entidad. En los esquemas *ZKP*, la entidad mantiene su contraseña secreta, sin compartirla con nadie. Es capaz de demostrar que conoce su contraseña a cualquier verificador escéptico, sin filtrar ningún tipo de información que permita reconstruir su secreto. En [9] puede encontrarse un ejemplo de esquema de autenticación basado en *ZKP*.

Las técnicas arriba mencionadas se apoyan en la Teoría de Números. Muchas de esas técnicas explotan el conocido hecho de que todavía no se ha encontrado un algoritmo que permita factorizar un entero grande (con cientos de dígitos) en un tiempo razonable. Su fortaleza reside en la capacidad de computar números primos grandes.

Por último, comentamos la estructura de este trabajo:
-El Capítulo 2 incluye los fundamentos matemáticos en que se basa el sistema RSA y todas las nociones necesarias para entender los resultados principales del siguiente capítulo.
-El Capítulo 3 contiene:
Una primera sección en la que se presenta el diseño del esquema RSA y del esquema de autenticación ZKP haciendo uso del lenguaje de modelización UML (diagrama de clases y de secuencias). Como la parte más delicada es la generación de primos, dedicamos la siguiente sección a hacer una recopilación de métodos para ello. Como también merece especial interés la proposición de Solovay-Strassen, dedicamos otra sección a su demostración. En la última sección de este capítulo exponemos la complejidad de los algoritmos definidos.
-El anexo es el desarrollo detallado del análisis de las complejidades expuestas en la última sección del Capítulo 3.

Capítulo 2

Definiciones y Referencias

2.1. Definiciones y Referencias

2.1.1. Teoría de números

Definición 2.1.1. *Dados dos enteros a, b y un natural n, se dice que a es congruente con b módulo n si y sólo si n es un divisor de (a-b).*

$$a \equiv b \pmod{n} \Leftrightarrow n|(a-b)$$

Propiedades 2.1.2. *Sean a, b, c y n enteros con n>0.*

- *La congruencia de números es una relación de equivalencia, es decir, es reflexiva, simétrica y transitiva ya que se verifica:*
  1. $a \equiv a \pmod{n}$
  2. $a \equiv b \pmod{n}$, *entonces* $b \equiv a \pmod{n}$
  3. $a \equiv a \pmod{n}$ *y* $b \equiv c \pmod{n}$, *entonces* $a \equiv c \pmod{n}$
- *Si a es congruente con b módulo n, mcd(a, n)=mcd(b, n). En particular, si a es coprimo con n, b también lo es.*
- *Si* $a \equiv b \pmod{n}$ *entonces también se cumple:*
  1. $a+c \equiv b+c \pmod{n}$
  2. $a \cdot c \equiv b \cdot c \pmod{n}$
  3. $a^c \equiv b^c \pmod{n}$ *con* $c > 0$

Definición 2.1.3. *(9.1 en [2]).*

Dado un número entero a y un primo p, se define el símbolo de Legendre, $\left(\frac{a}{p}\right)$, *como:*

$$\left(\frac{a}{p}\right) = \begin{cases} 0 & \text{si } p \text{ divide a } a \\ 1 & \text{si } a \text{ es residuo cuadrático módulo } p \\ -1 & \text{si } a \text{ no es residuo cuadrático módulo } p \end{cases} \tag{2.1}$$

donde a es o no residuo cuadrático módulo p si la congruencia $x^2 \equiv a \pmod{p}$ *tiene o no solución.*

Definición 2.1.4. *(9.2 en [2]).*

Sea a un número entero y n un número natural impar con n>2 cuya factorización viene dada por $n = p_1^{\alpha_1} p_2^{\alpha_1} \dots p_r^{\alpha_r}$*, se define el símbolo de Jacobi como*

$$\left(\frac{a}{n}\right) = \left(\frac{a}{p_1}\right)^{\alpha_1} \left(\frac{a}{p_2}\right)^{\alpha_2} \cdots \left(\frac{a}{p_r}\right)^{\alpha_r}$$

donde $\left(\frac{a}{p_i}\right), i = 1 \dots r$*, es el símbolo de Legendre de (2.1).*
Si n es primo el símbolo de Jacobi se reduce al de Legendre.

Propiedades del símbolo de Jacobi

- $\left(\frac{a}{n}\right) = 0$ si $mcd(a,n) \neq 1$
- $\left(\frac{a\,b}{n}\right) = \left(\frac{a}{n}\right)\left(\frac{b}{n}\right)$
- $\left(\frac{a}{m\,n}\right) = \left(\frac{a}{m}\right)\left(\frac{a}{n}\right)$
- $\left(\frac{2}{n}\right) = (-1)^{\frac{n^2-1}{8}}$
- $\left(\frac{a}{n}\right) = \left(\frac{n}{a}\right)(-1)^{\frac{a-1}{2}\frac{n-1}{2}}$
- $\left(\frac{a}{n}\right) = \left(\frac{b}{n}\right) \Leftrightarrow a \equiv b \pmod{n}$

Definición 2.1.5. $\mathbb{Z}_n$ *es el conjunto de las clases de equivalencia de los enteros módulo n. Como cada clase módulo n admite un representante canónico en [0,n-1], se puede efectuar la siguiente identificación:*

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

Se llama grupo de las unidades de $\mathbb{Z}_n$ *al conjunto de sus elementos invertibles, y se denota por* $\mathbb{Z}_n^*$*. Por definición, sus elementos son:*

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

Si p es primo $\Rightarrow \mathbb{Z}_p$ *es cíclico de orden p y* $\mathbb{Z}_{p^e}^*$ *es cíclico de orden* $p^{e-1}(p-1)$*(ver Sección 2.10 en [7]).*

Teorema 2.1.6. *(Teorema de Lagrange (teoría de grupos), 2.1 en [7]).*

Sea G un grupo finito y H un subgrupo de G, entonces el orden de H es un divisor del orden de G.

Teorema 2.1.7. *(Teorema de Lagrange para congruencias polinómicas, 5.21 en [2]).*

Dado un primo p, sea

$$f(x) = c_0 + c_1 x + \cdots + c_n x^n$$

un polinomio de grado n con coeficientes enteros tales que $c_n \not\equiv 0 \pmod{p}$. *Entonces la congruencia polinómica*

$$f(x) \equiv 0 \pmod{p}$$

tiene a lo sumo n soluciones.

Teorema 2.1.8. *(Teorema Chino del Resto, 5.26 en [2]).*

Sean $n_1, n_2, \cdots, n_r (r > 2) \in \mathbb{N}$ *tales que* $mcd(n_i, n_j) = 1$ *para* $i \neq j$. *Entonces dados cualesquiera* $a_1, a_2, \cdots, a_r \in \mathbb{Z}$, *existe un x tal que:*

$$\begin{cases} x \equiv a_1 \pmod{n_1} \\ x \equiv a_2 \pmod{n_2} \\ \cdots \\ x \equiv a_r \pmod{n_r} \end{cases} \tag{2.2}$$

tiene una solución única módulo $n = n_1 n_2 \cdots n_r$.
Además, si y es otro entero que satisface las congruencias de (2.2) se cumple que $y \equiv x \pmod{n_1 n_2 \cdots n_r}$.

Proposición 2.1.9. *Sean n,* $m \in \mathbb{Z}^+$. *Entonces:*

- *Si m|n, mcd (n,m)=m.*
- *Sino, mcd (n,m)=mcd (m,n %m)*

Definición 2.1.10. *(Sección 2.3 en [2]).*

Sea n un número entero positivo, la función $\varphi(n)$ *de Euler se define como el número de enteros positivos menores o iguales a n y coprimos con n. Equivalentemente:*

$$\varphi(n) = \{m \in \mathbb{N} | m \leq n \wedge mcd(n,m) = 1\}$$

Para números primos p,

$$\varphi(p) = p - 1$$

Teorema 2.1.11. *(Teorema de Euler-Fermat, 5.17 en [2]).*

Sean dos enteros positivos, a y n, t.q. mcd(a,n) = 1. Entonces:

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

Corolario 2.1.12. *(Pequeño Teorema de Fermat 5.18 en [2]).*
Si p es un número primo, para cada número entero a se verifica:

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

y si mcd(a,p) = 1 entonces,

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

Corolario 2.1.13. *(VI. The Underlying Mathematics en [14]).*
Sea un entero n tal que $n = p \cdot q$ con p, q primos y distintos entre sí, entonces

$$\varphi(n) = \varphi(p) \cdot \varphi(q) = (p-1)(q-1)$$

Sea un entero d coprimo con $\varphi(n)$, existe un entero e en $\mathbb{Z}_{\varphi(n)}$ tal que $e \cdot d \equiv 1 \pmod{\varphi(n)}$. Entonces,

$$m^{e \cdot d} \equiv m \pmod{n}$$

2.1.2. Nociones de complejidad

Un algoritmo se diseña para resolver problemas que pueden tener una colección infinita de ejemplares o casos. Se podrá considerar como bueno en términos de tiempo de computación dependiendo de su eficiencia, de su rapidez de ejecución.
Caracterizamos el tamaño de un ejemplar como una medida para el número de componentes que tenga y la complejidad como la cantidad de recursos necesarios para cada algoritmo como función del tamaño de los casos considerados. En nuestro caso, el recurso más importante es el tiempo de computación.
Por ejemplo, se nos plantea el problema de la suma de dos números enteros que tiene como ejemplares 23+45, 123+292 y 1295423+849, entre muchos otros. El tamaño de este problema sería el número de dígitos del sumando mayor considerando como operación elemental la suma de dos dígitos.

Definimos operación elemental como aquella cuyo tiempo de ejecución se puede acotar por una constante que no depende ni del tamaño ni de los parámetros del ejemplar que se esté considerando. Siempre buscaremos descomponer el algoritmo en operaciones elementales para su posterior análisis.
Si consideramos *f(n)* el número de operaciones elementales requeridas para resolver un ejemplar de tamaño *n*, podemos representar *el orden de f(n)*, *O(f(n))*, como el conjunto de todas las funciones *t:*$\mathbb{N} \to \mathbb{R}^+$ tales que $t(n) \leq cf(n)$ para todo $n \geq n_0$ para una constante positiva real c y para un umbral entero n_0 donde f es una función arbitraria de los números naturales en los reales no negativos (3.2 en [5]),

$$O(f(n)) = \{t:\mathbb{N} \to \mathbb{R}^+ \mid \exists\, c \in \mathbb{R}^+, \forall n \geq n_0,\ n_0 \in \mathbb{N}\ t.q.\ t(n) \leq cf(n)\}$$

Esta notación nos proporciona cotas superiores sobre la cantidad de recursos requeridos, pero cuando analizamos un algoritmo preferimos que su tiempo de ejecución esté acotado tanto superior como inferiormente. Por este motivo, presentamos la notación Theta de *f(n)* y el orden exacto de *f(n)* (3.3 en [5]),

$$\Theta(f(n)) = \{t:\mathbb{N} \to \mathbb{R}^+ \mid \exists\, c,d \in \mathbb{R}^+, \forall n \geq n_0,\ n_0 \in \mathbb{N}\ t.q.\ df(n) \leq t(n) \leq cf(n)\}$$

También el análisis de muchos algoritmos se simplificará de forma significativa cuando sea posible calcular *f(n)* mediante el método de la sentencia barómetro. Éste consiste en localizar una instrucción elemental que se ejecute por lo menos con tanta frecuencia como cualquier otra del algoritmo. Siempre que el tiempo requerido por cada instrucción esté acotado por una constante, el tiempo requerido por el algoritmo completo es del orden exacto del número de veces que se ejecuta la instrucción barómetro.

Capítulo 3

Resultados principales

3.1. Diseño del algoritmo RSA

La figura 3.1 muestra los tres algoritmos del sistema RSA: generarClave, encriptar y desencriptar. En la generación se crean la clave pública (n,e) y la privada (n,d). El corolario 2.1.13 garantiza que $(M^e \,\%n)^d \,\%n = M$ para todo entero M menor que n. La encriptación de un texto consta de dos pasos. En el primero, se codifica el texto, transformándolo en un entero M. Esto puede hacerse, por ejemplo, sustituyendo cada carácter por su código ASCII. A continuación se calcula $m = M^e \,\%n$, donde (n,e) es la clave pública. El desencriptado consiste en realizar los pasos inversos: se calcula $M' = m^d \,\%n$ y se decodifica M', reconstruyéndose el texto original.

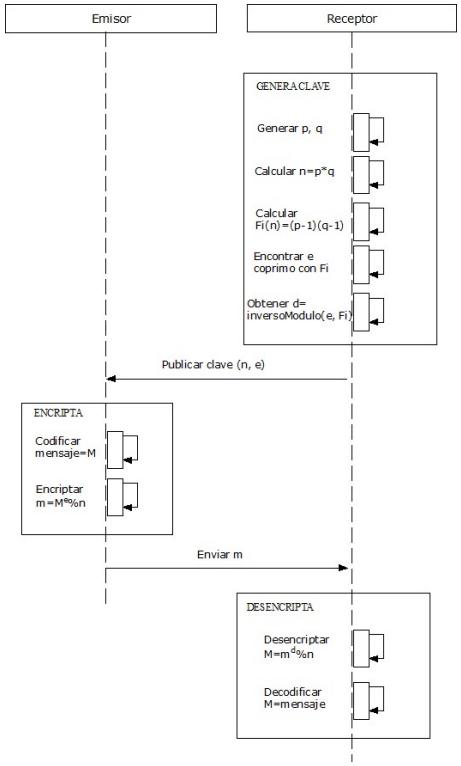

Figura 3.1: Sistema RSA

La figura 3.2 muestra el diagrama de clases que modeliza las entidades con que trabajamos, y los métodos que son capaces de ejecutar. El Desencriptador genera una terna (n,e,d), de la cual (n,e) es la clave pública, accesible a todas las entidades. Las distintas entidades pueden intercambiarse mensajes; una sucesión de mensajes entre dos entidades es un diálogo. Un ejemplo de diálogo es un proceso de autenticación, en el que una entidad demuestra quién es al responder a una serie de preguntas.

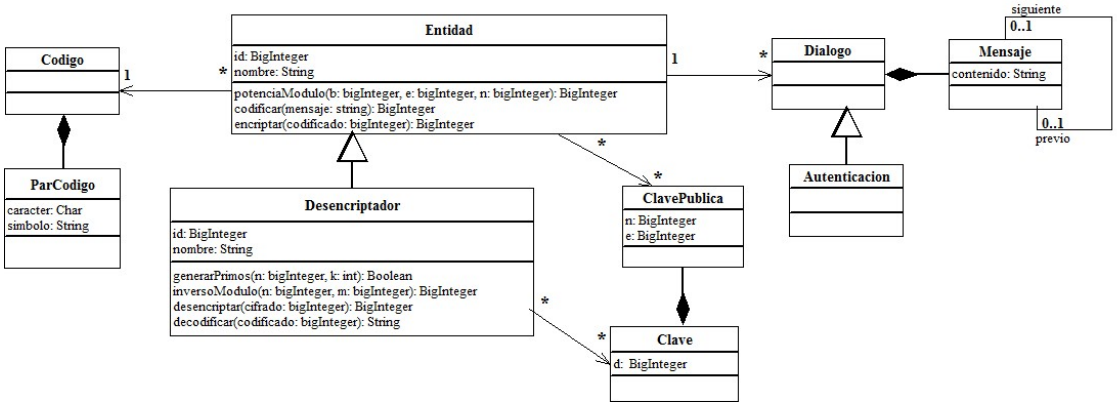

Figura 3.2: Clases del sistema RSA

En la figura 3.3 mostramos un ejemplo de autenticación basado en la criptografía RSA. Para demostrar quién es, la entidad debe demostrar que conoce su clave privada (d). El verificador sólo conoce la clave pública (n,e). El verificador le envía una serie de mensajes M a la entidad. La entidad calcula $M^d \%n$ y le devuelve el resultado al verificador. Este comprueba si $(M^d \%n)^e \%n$ coincide con su mensaje original, y puede repetir las pruebas todas las veces que sea necesario.

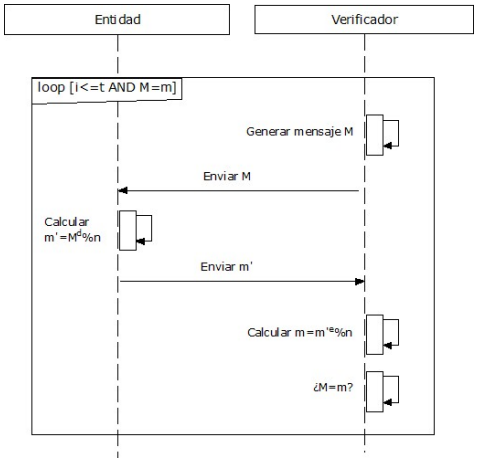

Figura 3.3: Autenticación con RSA

Este modelo de autenticación tiene problemas. Toda entidad tiene que ser capaz de generar sus propias claves. Una solución más asequible para entidades con pocos recursos de computación se presenta en [9]. Un Trusted Center genera un entero $n = p \cdot q$, con p, q primos. Todas las entidades del sistema comparten ese n, y ninguna conoce su factorización. Cada entidad genera una serie de enteros S_i que guarda en secreto. A partir de ellos genera $I_i = \pm 1/S_i^2$. Esos I_i son su identificador público. Observamos que para calcular los S_i a partir de los I_i hay que calcular la raíz cuadrada de estos módulo n. Hacer eso sin conocer la factorización de n es inviable para n grande. El modo de demostrar su identidad es demostrar que conoce S_i. Eso lo hace a través de un diálogo como el representado en la Figura 3.4. En [9] se demuestra que en este diálogo el verificador queda convencido de que la entidad conoce las S_i, pero la información que recibe no le permite reconstruir esos valores."

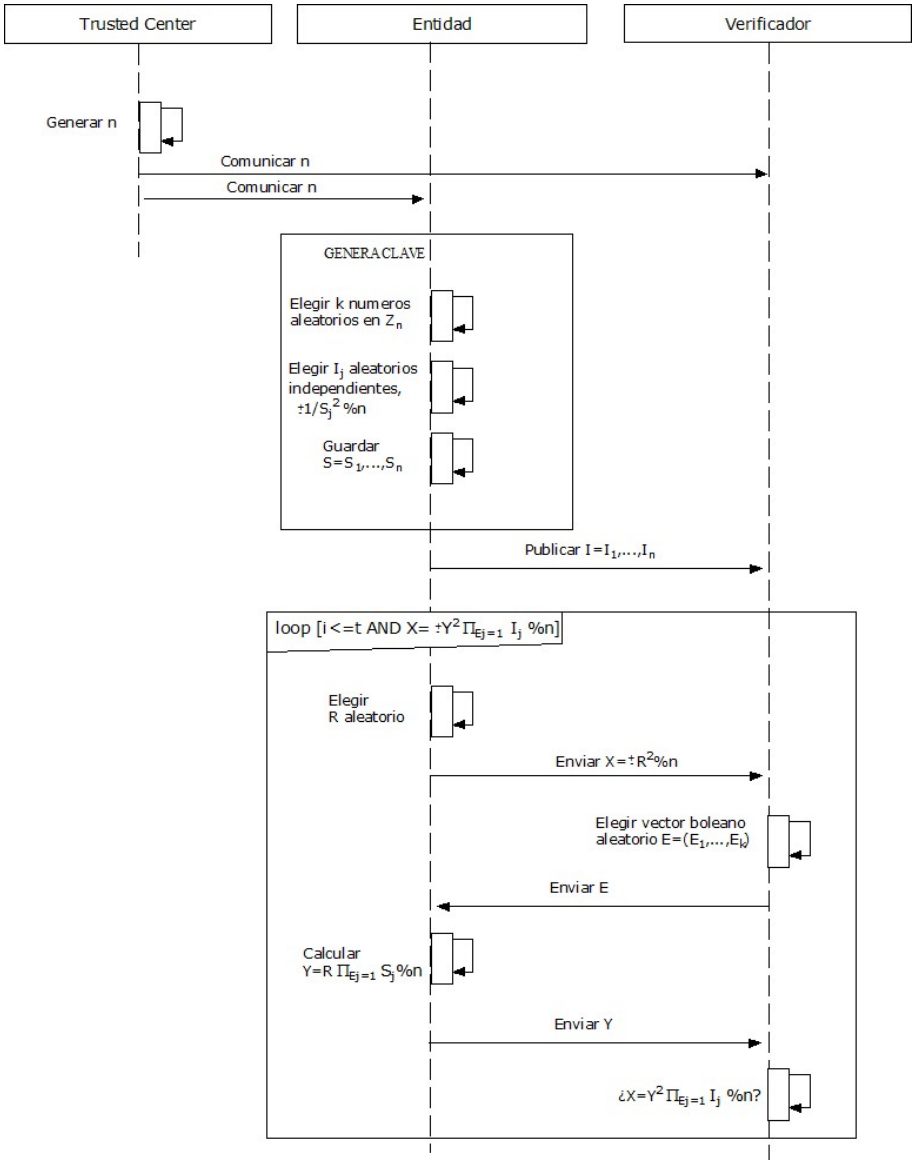

Figura 3.4: Autenticación con ZKP

3.2. Generación de primos

La fortaleza de las soluciones descritas anteriormente reside en la capacidad de computar números primos grandes. El problema es ¿Cómo generar números primos arbitrariamente grandes?

Estamos sujetos a las siguientes restricciones:

- No se conoce ningún algoritmo genérico que permita obtener directamente números primos arbitrariamente grandes.
- Los números primos están distribuidos inhomogéneamente. Al inspecionar interalos pe-

queños de números, la distribución de primos es altamente irregular: no sabemos dónde encontrar el siguiente.

- La seguridad se basa en el secreto. Los primos que utilizamos para construir nuestras claves deben mantenerse en secreto; más aún, un tercero no debería tener ninguna pista que le permita adivinarlos: nuestros primos deberían ser impredecibles.

Una posible solución compatible con los requisitos previos consiste en ir generando números aleatorios hasta encontrar un primo.

A continuación damos una serie de pautas seguidas en la generación de primos.

3.2.1. Candidatos Descartables

Cuanto mayores sean los primos (y por tanto, más seguros), más larga se hará la búsqueda. Según el teorema de los números primos, la probabilidad de que un entero aleatorio con un máximo de n_{Dig} dígitos sea primo es del orden de $1/log(10^{n_{Dig}})$, donde log es el logaritmo neperiano. Por ejemplo, para $n_{Dig} = 100$ haría falta generar, en promedio, 230 enteros antes de encontrar un primo; si $n_{Dig} = 1000$, en promedio harían falta 2300 intentos.

Para abordar el problema anterior, es posible identificar familias de números cuyo carácter de compuestos sea fácil de comprobar. Si es posible, evitarlos. Si no, buscar un modo rápido de transformarlos en otros candidatos más prometedores. Algunos ejemplos de aplicación de este principio son:

- Es inmediato que los enteros pares y los que terminan en 5 son compuestos (excepto, por supuesto, el 2 y el 5). Esta condición es inmediata de verificar: basta mirar su último dígito. Descartándolos de antemano nos quedan 2/5 de los posibles candidatos iniciales (aquellos n tales que $n \equiv 1,3,7,9 \ (mod\ 10)$). En promedio, la cantidad de enteros que requerirán un análisis más concienzudo ser redude a 92 (si $n_{Dig} = 100$), o a 920 (si $n_{Dig} = 1000$).

- Comprobar si un entero n es múltiplo de 3 es rápido. Si n es múltiplo de 3, y no de 2 ni 5, es posible transformarlo en otro número que no sea múltiplo de 2, 3 ni 5 (por ejemplo, haciendo la suma $n+10$). Aplicando esta regla nos quedan $4/15$ de los candidatos iniciales (aquellos n tales que $n \equiv 1,7,11,13,17,19,23,29 \ (mod\ 30)$.

- Es posible comprobar si un número n es una potencia perfecta (es decir, si existen $base$, $expo$ tales que $base^{expo} = n$) en $log^3\ n \cdot log\ log\ n$ pasos (ver Tabla 3.4). Este resultado fue utilizado por Miller [12] para identificar y descartar rápidamente esos números.

3.2.2. Búsqueda de Testigos

No cononocemos una forma genérica, eficiente y directa de caracterizar números primos. Existen caracterizaciones “directas", que sólo dependen de n. Mostramos algunas en la tabla 3.1, junto con la razón por la que no son viables en nuestro contexto.

Nombre	Caracterización	Inconveniente
Wilson [2]	n es primo sii $(n-1)! \equiv -1 \ (mod\ n)$	El cálculo de $(n-1)!$ es inviable para enteros grandes.
Pepin [6]	F_k es primo sii $3^{(F_k-1)/2} \equiv -1 \ (mod\ F_k)$	Sólo aplicable a números de Fermat: $F_k = 2^{2^k}+1, k \geq 1$
Lucas-Lehmer [6]	M_p es primo sii $v_{p-2} \equiv 0 \ (mod\ M_p)$ donde $v_0 = 4$, $v_{k+1} = v_k^2 - 2$	Sólo aplicable a los enteros de Mersenne, $M_p = 2^p - 1$, con p primo

Cuadro 3.1: Caracterizaciones “directas"de primos (dependen sólo de n)

Una solución pasa por buscar testigos de primalidad (composición) de n. Esta solución requiere:

- un dominio D donde buscar ese testigo.
- una condición $C(n,w)$ que identifique a w como testigo. C puede expresarse como una condición booleana, cuyo valor es *cierto* si w es un testigo válido.

En resumen, este patrón se basa un resultado del tipo: n es primo (compuesto) si existe $w \in D$ tal que $C(n,w) = cierto$.

En las Tablas 3.2 y 3.3 mostramos el dominio D y la condición $C(n,w)$ para distintos métodos para determinar si n es primo.

Hacemos notar que el test de Proth [6] sólo es aplicable a los enteros de Proth: $n = A \cdot 2^s + 1$, con A impar y $A < 2^s$.

3.2.3. Dominio Restringido

El dominio de búsqueda es tan grande que recorrerlo puede resultar inviable. Una posible solución pasa por restringir el dominio de búsqueda. Ejemplos de soluciones de este tipo son:

- Si n es compuesto, tiene un divisor menor o igual que $\sqrt{n}$. Eso reduce el dominio de búsqueda a $[2, \sqrt{(n)}]$.
- Más aún, en criptografía trabajamos con primos impares, que no son múltiplos de 2. Podemos restringir nuestro dominio a los impares en $[2, \sqrt{(n)}]$.

Algoritmo	Dominio $D(n)$	$\#(D(n))$	$\#(D(10^{200}))$	$C(n,w)$	Tipo
Naïve 01	[2, n-1]	n	10^{200}	C_{Na} OR C_{Nb}	C
Naïve 02	$D_2 = [2, \sqrt{n}]$	$\sqrt{n}$	10^{100}	C_{Na} OR C_{Nb}	C
Naïve 03	$\{2\} \bigcup \{w \in D_2, w \text{ impar}\}$	$\sqrt{n}/2$	$5 \cdot 10^{99}$	C_{Na} OR C_{Nb}	C
Naïve 03	$\{w \in D_2, w \text{ primo}\}$	$\sqrt{n}/log(\sqrt{n})$	$\sim 10^{98}$	C_{Na} OR C_{Nb}	C
Lucas [6]	$[2, n-1]$	n	10^{200}	C_{Lucas}	P
Proth [6]	$[2, n-1]$	n	10^{200}	C_{Proth}	P
Solovay-Strassen (det)	$[2, 2 \cdot log^2 n]$	$2 \cdot log^2 n$	$8 \cdot 10^4$	$C_{SolStrass}$	C
Miller (det)	$[2, 2 \cdot log^2 n]$	$2 \cdot log^2 n$	$8 \cdot 10^4$	C_{Miller}	C

Cuadro 3.2: Ejemplos de aplicaciones del patrón *Busca Testigo*. Mostramos el dominio D donde buscar el testigo, la condición $C(n,w)$ y el tipo de testimonio (primalidad/composición). Para aclarar la intuición de la cardinalidad de D (el número de testigos que, en el peor de los casos, debemos comprobar) mostramos la cardinalidad de D si n tiene 200 dígitos. En esta tabla consideramos el algoritmo de Miller y la versión determinista del algoritmo de Solovay-Strassen. Como comparación, el número de átomos del Universo se estima del orden de 10^{82}

- Podríamos restringir D a los primos en $[2, \sqrt{(n)}]$. Según el teorema de los números primos, la cardinalidad del nuevo dominio es $\sqrt{(n)} \cdot ln(\sqrt{(n)})$. Esta solución sigue la misma idea que la criba de Eratóstenes. Sin embargo, la complejidad de computar los primos en ese dominio (y los requisitos de memoria para almacenarlos) hacen que la solución sea inviable para números grandes.

- Diseñando una condición más sofisticada, Miller [12] restringió el dominio de búsqueda a $c \cdot log^2 n$, donde c es un factor constante. En [3] se dio un valor preciso para c: $c \leq 2$. El mismo dominio puede aplicarse al test de Solovay-Strassen [16]. Como contrapartida, estas soluciones se apoyan en la Conjetura Extendida de Riemann.

3.2.4. Aceptar la Incertidumbre

Pese a todo, el coste computacional sigue siendo demasiado alto, especialmente para dispositivos con capacidades limitadas de memoria y procesamiento. Todo el esfuerzo dedicado a generar y descartar un testigo fallido se pierde. ¿Es posible extraer información de los testigos descartados?

Hemos de tener en cuenta que:

- La certeza completa no existe en el mundo real.

  Una certeza absoluta sólo es posible en el reino de las Matemáticas, pero no en el mundo real. Aun cuando un algoritmo se base en un teorema demostrable y se haya programado

Nombre	C (n,w)	Coste computacional
C_{Na}	$n \% w \neq 0$	$log\ n$
C_{Nb}	$mcd(n,w) \neq 1, n$	$log^2\ n$
C_{Lucas}	$w^{n-1} \equiv 1 \pmod{n}$ AND	
	$w^{(n-1)/p} \not\equiv 1 \pmod{n}$ para todo primo p divisor of $n-1$	$F(n-1)$
C_{Proth}	$w^{(n-1)/2} \equiv -1 \pmod{n}$	$log\ n \cdot M(n)$
$C_{SolStrass}$	$mcd(w,n) \neq 1$ OR	
	$w^{(n-1)/2} \not\equiv (w\|n) \pmod{n}$	$log\ n \cdot M(n)$
C_{Miller}	w divide a n OR	
	$w^{n-1} \not\equiv 1 \pmod{n}$ OR	
	$mcd((w^{(n-1)/2^k} \% n) - 1, n) \neq 1, n$	
	para algún k tal que 2^k divide a $n-1$	$log\ n \cdot M(n)$

Cuadro 3.3: Condiciones de los algoritmos presentados en la Tabla 3.2. $F(n-1)$ representa el coste de factorizar $n-1$, que crece exponencialmente con el tamaño de n. $M(n)$ es el coste computacional de multiplicar $n \cdot n$

sin fallos, la máquina donde se ejecuta está sometida a errores.

- Apetito de riesgo

  La necesidad de una solución es tan acuciante que estaríamos dispuestos a aceptar una solución, aun cuando no fuera perfecta, siempre y cuando el riesgo se mantuviera bajo control.

Una posible solución es usar un algoritmo probabilista, que equilibra certidumbre y eficiencia.

Ejemplos de este tipo de solución son: Solovay-Strassen [16] y Rabin [13] propusieron tests probabilistas, en que cada prueba nos da información. Si $C(w,n) = 0$ para un cierto w, estamos completamente convencidos de que n es compuesto. Si $C(w,n) = 1$, la probabilidad de que n no sea primo es menor de $1/2$ (en el caso de Solovay-Strassen) y $1/4$ (en el caso de Miller-Rabin).

Supongamos que consideramos n aceptable como primo si la probabilidad de error (de que n sea compuesto, pese a no haber encontrado ningún testigo) sea menor de 2^{-50} ($9 \cdot 10^{-14}$ %. Con el test probabilista de Solovay-Strassen, sería suficiente probar con 50 posibles testigos. Con el método Miller-Rabin bastarían 25.

3.3. Solovay-Strassen

- n es primo $\iff \left(\frac{a}{n}\right) \equiv a^{\frac{n-1}{2}} \pmod{n}$, $\forall a \in [1, n-1] \wedge \text{mcd}(a,n) = 1$
- Si n es compuesto, al menos la mitad de los enteros menores de n son testigos de que es compuesto.
- Si la hipótesis extendida de Riemann, ERH, es cierta, existe un testigo en $[1, log^2 n]$.

Demostración

- $\Rightarrow$) Probaremos que si n es primo, entonces $\left(\frac{a}{n}\right) \equiv a^{\frac{n-1}{2}} \pmod{n}$, $\forall a$. (nótese que este resultado es más general que el del enunciado, ya que, se aplica también para a no coprimos con n).
  Supongamos que n es primo. Entonces $\left(\frac{a}{n}\right)$ es el símbolo de Legendre (2.1.3).
  $\mathbb{Z}_n$ es cíclico (2.1.5). Sea g su generador.

  $\rightarrow$ Si $a \equiv 0 \pmod{n}$, es inmediato que $a^{\frac{n-1}{2}} \equiv 0 \pmod{n}$.
  $\rightarrow$ Si a es residuo cuadrático $\Longrightarrow$ existe b tal que $b^2 \equiv a \pmod{n}$. A su vez, $b = g^\alpha$ para cierto $\alpha \Longrightarrow a \equiv b^2 \equiv g^{2\alpha} \pmod{n}$.
  Por tanto, $a^{\frac{n-1}{2}} \equiv (g^{2\alpha})^{\frac{n-1}{2}} \equiv g^{\alpha(n-1)} \equiv (g^{n-1})^\alpha \equiv 1^\alpha \equiv 1 \pmod{n}$.
  $\rightarrow$ Si a no es residuo cuadrático $\Longrightarrow a = g^\alpha$ con α impar. Entonces, $a^{\frac{n-1}{2}} = g^{\alpha(\frac{n-1}{2})} = g^{\frac{\alpha(n-1)}{2}}$. Además, sabemos que $g^{n-1} \equiv 1 \pmod{n}$. Así que, tomando $h = g^{\frac{(n-1)}{2}}$ obtenemos $h^2 \equiv 1 \pmod{n}$, que tiene como mucho dos soluciones (2.1.7). Pero $h \not\equiv 1 \pmod{n}$ porque $\frac{n-1}{2} < n-1$ y como $g^{n-1} \equiv 1 \pmod{n}$, no puede ser $g^{\frac{n-1}{2}} \equiv 1 \pmod{n}$. Por consiguiente, $h \equiv -1 \pmod{n}$ y entonces $a^{\frac{n-1}{2}} = h^\alpha \equiv -1^\alpha \equiv -1 \pmod{n}$ (2.1.2).

  $\Leftarrow$) Si $a \equiv 0 \pmod{n}$, la igualdad es trivial.
  Nos centramos en el caso $a \not\equiv 0 \pmod{n}$ (ver [16]).
  Veremos en primer lugar que n es libre de cuadrados, es decir, su factorización en primos es $n = \prod_i p_i$ con $p_i \neq p_j$ para $i \neq j$.

  Dado que a es coprimo con n, se tiene que $\left(\frac{a}{n}\right) = \pm 1$. Por tanto, $a^{n-1} = (a^{\frac{n-1}{2}})^2 \equiv 1 \pmod{n}$. Sea p un primo divisor de n, e la máxima potencia tal que $p^e | n$. Entonces, $n = p^e \cdot d$ donde el mcd(p^e, d)=1. Como p es primo, $\mathbb{Z}^*_{p^e}$ es cíclico de orden $p^{e-1}(p-1)$ (2.1.5). Sea g el generador de $\mathbb{Z}^*_{p^e}$. Buscamos b tal que

  $$b \equiv g \pmod{p^e}, \quad b \equiv 1 \pmod{d}$$

  Dicho b existe por el Teorema Chino del Resto (2.1.8). Además, mcd(b, p^e)=mcd(b, d)=1 (2.1.9) $\Rightarrow$ mcd(b, n)=1. Por tanto,

  $$b^{n-1} \equiv 1 \pmod{n} \Rightarrow b^{n-1} \equiv 1 \pmod{p^e} \Rightarrow g^{n-1} \equiv 1 \pmod{p^e}$$

Como g es el generador, $p^{e-1}(p-1)|p^e \cdot d-1$. Esto implica que existe α tal que $\alpha \cdot p^{e-1}(p-1) = p^e \cdot d-1$; y despejando, $1 = p^{e-1}(p \cdot d - \alpha \cdot p + \alpha) \Leftrightarrow p^{e-1} = 1 \Leftrightarrow e = 1$.

Ahora probaremos que n no puede ser producto de más de un primo. Lo veremos por reducción al absurdo.
Supongamos $n = \prod_i p_i$. Existe b que no es raíz cuadrática módulo p_1. Tomamos a tal que

$$a \equiv b \pmod{p_1}, \quad a \equiv 1 \pmod{p_i}\ i \neq 1$$

Existe a por el Teorema Chino del Resto (2.1.8). Además, mcd(a, p_1)=mcd(a, p_i)=1 $\forall i \Rightarrow$ mcd(a, n)=1. Por tanto, $\left(\frac{a}{n}\right) \equiv a^{\frac{n-1}{2}} \pmod{n}$. Como $\left(\frac{a}{n}\right)$ es el símbolo de Jacobi (2.1.4), $\left(\frac{a}{n}\right) = \left(\frac{a}{p_1}\right) \cdot \left(\frac{a}{p_2}\right) \cdots \left(\frac{a}{p_r}\right) = -1$. Es decir, $a^{\frac{n-1}{2}} \equiv -1 \pmod{n} \Rightarrow a^{\frac{n-1}{2}} \equiv -1 \pmod{p_i}\ \forall i$. Pero esto se contradice con que $a \equiv 1 \pmod{p_i}$. Así n es primo.

- Consideramos el grupo

$$G = \left\{ a+(n) | a \in \mathbb{Z} \wedge \text{mcd}(a,n) = 1 \wedge \left(\frac{a}{n}\right) \equiv a^{\frac{n-1}{2}} \pmod{n} \right\}$$

Todo elemento fuera de $\mathbb{Z}_n^*$ que está fuera de G es testigo de que n es compuesto. Por el punto anterior, si n es compuesto, G es un subgrupo de $\mathbb{Z}_n^*$. Así que, por el Teorema de Lagrange (2.1.6), si $G \neq \mathbb{Z}_n^*$, entonces $|G(n)| \leq \frac{|\mathbb{Z}_n^*|}{2} \leq \frac{n-1}{2}$. Por tanto, como mínimo 1/2 de los números entre 1 y n-1 son testigos de que n es compuesto.

- La complejidad de los algoritmos de Miller y Solovay-Strassen (Determinista) depende de la rapidez para encontrar un número fuera de un subgrupo no trivial del grupo multiplicativo de enteros módulo n. Esta cuestión es comentada por Ankeny en [1] y también por Bach en su artículo [3], quien dice: *"Sea G un subgrupo propio del grupo multiplicativo de enteros módulo n. Entonces, asumiendo la Hipótesis Extendida de Riemann (ERH), el menor entero positivo fuera de G es* $O(log^2 n)$*"*. Todos los números de $\mathbb{Z}_n$ que estén fuera de G son testigos de primalidad. En particular, demuestra que ese *"menor entero positivo"* está acotado por $2log^2 n$.

3.4. Algoritmos y Complejidades

Para simplificar la descripción de nuestros algoritmos, tenderemos a omitir las declaraciones de magnitudes escalares (variables y parámetros de funciones). Por defecto y mientras no se diga explícitamente lo contrario, sobreentendemos que las variables, parámetros y resultados devueltos por las funciones son enteros positivos.

Del mismo modo, mientras no se especifique otra cosa los algoritmos descritos son independientes del sistema de numeración usado (deberían ser válidos tanto para base 2 como para base 10, por ejemplo). Denotaremos por *base* la base en que estemos trabajando. Ello no afecta a la complejidad de los algoritmos considerados. En efecto, para toda base existe una constante positiva c_{base} tal que $log_{base} n = c_{base} \cdot log_{10} n$ (más precisamente, $c_{base} = log_{base}\ 10$. En consecuencia, para todo $\alpha \in \mathbb{R}^+$ se tiene que $O\left(log^{\alpha}_{base} n\right) = O\left(log^{\alpha}_{10} n\right)$. Por tanto, al describir la complejidad de un algoritmo omitiremos la base.

Haremos uso del lenguaje matemático para conseguir mayor claridad en nuestros algoritmos, incluyendo símbolos tales como $\lfloor\rfloor$, $/$ y $\%$ que presentamos a continuación:
Si x es un número real, $\lfloor x \rfloor$ reperesenta el mayor entero que no es mayor que x. Nótese que el número de dígitos de un número n en base *base* viene dado por $\lfloor log_{base}\ n \rfloor + 1$. $\lceil x \rceil$ representa el menor entero que no es menor que x.
Si n y m son dos enteros positivos $m \div n$ denota el resultado de dividir m por n, lo cual no es necesariamente un entero. Denotamos el cociente entero mediante $/$ y el resto de la división entera por $\%$. Obsérvese que $n/m = \lfloor n \div m \rfloor$. Dado un número n, con la notación $n[i]$ representamos el i-ésimo dígito de n (así, $n = \sum_{i=0}^{\lfloor log_{base}\ n \rfloor} n[i]\ base^i$).

Consideraremos operaciones elementales:

- sumar, restar, multiplicar o dividir dos dígitos (dos números en el intervalo $[0, base - 1]$).
- multiplicar un número por la base del sistema de numeración (añadir un 0 a la derecha del número).
- calcular el cociente obtenido al dividir un número por la base del sistema de numeración (cortar el último dígito del número).
- calcular el resto obtenido al dividir un número por la base del sistema de numeración (tomar el último dígito del número).
- asignar valor a un dígito.
- acceder al dígito i-ésimo de un número n ($n[i]$).

Para simplificar la notación, al escribir *la complejidad del algoritmo* queremos decir *La complejidad del tiempo de ejecución del algoritmo*.

Algoritmo	**Notación**	**Complejidad**
Suma (A.1.1)	$n+m$	$\Theta(\ log\ máx(\|n\|,\|m\|)\ )$
Resta (A.1.1)	$n-m$	$\Theta(\ log\ máx(\|n\|,\|m\|)\ )$
Multiplicación (A.1.2)	$n \cdot m$	$\Theta(log\ n\ \cdot log\ m)$
División (A.1.3)	$n \div m$	$O(log\ n\ \cdot log\ m)$
Método de Reducción de Barrett (A.1.5)	$restoBarrett(a,n)$	$M(n)$
Potencia (A.1.4)	b^e	$O(e^2 \cdot log^2\ b)$
Potencia Módulo (A.1.6)	$b^e\ \%n$	$O(log\ e\ \cdot M(n))$
Potencia Perfecta (A.1.8)	$esPotenciaPerfecta(n)$	$O(log^3 n \cdot log\ log\ n)$
Euclides (A.1.9)	$euclides(n,m)$	$O(log\ n\ \cdot\ M(n))$
Inverso Módulo (Euclides) (A.1.10)	$inversoModuloEuclides(n,m)$	$O(log\ n\ \cdot\ M(n))$
Máximo Común Divisor (Binario) (A.1.11)	$mcdBin(n,m)$	$O(log^2\ n)$
Inverso Módulo (Binario) (A.1.12)	$inversoModuloBinario(n,m)$	$O(log^2\ n)$
Símbolo de Jacobi (A.1.13)	$(\frac{a}{n})$	$O(log\ a\ \cdot\ M(n))$
Condición de Miller (A.1.14)	$condicionMiller(a,n)$	$O(log^2\ n\ \cdot\ M(n))$
Condición de Miller Mejorada (A.1.17)	$condicionMillerMejorada(a,n)$	$O(log\ n\ \cdot\ M(n))$
Test de Miller (A.1.18)	$miller(n)$	$O(log^3\ n\ \cdot\ M(n))$
Test de Miller-Rabin (A.1.19)	$millerRabin(n,t)$	$O(t\ \cdot log\ n\ \cdot\ M(n))$
Condición de Solovay-Strassen (A.1.20)	$condicionSolovayStrassen(a,n)$	$O(log\ n\ \cdot\ M(n))$
Test de Solovay-Strassen (determinista) (A.1.21)	$solovayDeterminista(n)$	$O(log^3 n \cdot\ M(n))$
Test de Solovay-Strassen (probabilista) (A.1.22)	$solovayProbabilista(n,t)$	$O(t\ \cdot log\ n\ \cdot\ M(n))$

Cuadro 3.4: Complejidad de los algoritmos considerados. $M(n)$ denota la complejidad del producto $n \cdot n$; su valor para distintos algoritmos se muestra en la tabla 3.4

Algoritmo	$M(n)$
multiplicación naïve (A.1.2)	$log^2\ n$
Karatsuba ([11])	$log^{1,59} n$
Schönhage- Strassen ([15])	$log(n)log(log(n))log(log(log(n))))$

Cuadro 3.5: Complejidad de distintos algoritmos de multiplicación.

Computacionalmente, los algoritmos de multiplicación y división mostrados en la Tabla 3.4 son caros (están en $O(log^2 n)$), lo cual ha motivado una intensa investigación para proponer otros más eficientes.

Existen algoritmos más eficientes para multiplicar enteros, como el de Karatsuba [11], que está en $O(log^{1,59}n)$, y el de Schönhage- Strassen [15], que está en

$$O(log(n)log(log(n))log(log(log(n)))))$$

Al valorar la eficiencia de dichos algoritmos es necesario tener en cuenta el peso de las constantes ocultas. En [10] se comparan los tiempos de ejecución de los algoritmos para distintos procesadores. Para enteros de menos de 2^9 dígitos binarios (500 dígitos decimales) es más eficiente el algotritmo naïve; a partir de los 2^{10} dígitos binarios empieza a ser más eficiente el algoritmo de Karatsuba, y por encima de los $2^{15} - 2^{17}$ dígitos binarios (32000 dígitos decimales) se impone el de Schönhage-Strassen.

Análogamente, el resto de la división de dos enteros puede calcularse mediante el algoritmo de Barrett, que consiste básicamente en una multiplicación y un máximo de cuatro restas (A.1.5). Su complejidad es la misma que la del algoritmo usado para multiplicar.

La multiplicación y la división son más costosas que la suma y la resta. Ello explica la diferencia de las complejidades de los algoritmos *euclides* y *mcdBinario* (análogamente, *inversoModuloEuclides* e *inversoModuloBinario*): mientras el primero calcula cocientes, el segundo sólo necesita restas.

Apéndice A

A.1. Algoritmos y Complejidades

Algoritmo A.1.1. **SUMA**

función $sumar(n,m)$
$i \leftarrow 0$; $llevo \leftarrow 0$; $suma \leftarrow 0$
mientras $n \neq 0$ *OR* $m \neq 0$ **hacer**
    $n_1 \leftarrow n\%2$; $m_1 \leftarrow m\%2$
    $sumaDigito \leftarrow n_1 + m_1 + llevo$
    $llevo \leftarrow sumaDigito/2$
    $suma[i] \leftarrow sumaDigito\%2$
    $i \leftarrow i+1$; $n \leftarrow n/2$; $m \leftarrow m/2$
fin
devolver *suma*

Comentarios: Si los enteros están representados en binario mediante el sistema del complemento a dos, el algoritmo es válido independientemente de si son negativos o positivos. Sin pérdida de generalidad, supongamos $|n| \geq |m|$. El número de bits requeridos para representar a n y m es $\lfloor log_2 |n| \rfloor + 2$ (un máximo de $\lfloor log_2 |n| \rfloor + 1$ para los dígitos del módulo, y un dígito extra para el signo).

La complejidad del algoritmo A.1.1 está en $\Theta(\ log\ máx(|n|,|m|)\ )$

Demostración. El algoritmo de la suma anteriormente descrito consta solamente de operaciones elementales. Por tanto, cualquiera de las operaciones del bucle sirve como barómetro, y la complejidad del algoritmo viene dada por el número de iteraciones del bucle.

En cada división $n/2, m/2$ se elimina un dígito de n, m. Por tanto, el número de iteraciones del bucle es $\lfloor log_2 n \rfloor + 2$. Por tanto, la complejidad está en $\Theta(\ \lfloor log_2 n \rfloor + 2\ ) = \Theta(\ log\ n\ )$

Ese mismo cálculo es aplicable a la resta, ya que la resta $n-m$ equivale a la suma $n+(-m)$. En la representación considerada, $-m$ es el complemento a 2 de m, y su cálculo supone un coste extra de $\lfloor log_2\, n \rfloor + 2$ operaciones.

Algoritmo A.1.2. MULTIPLICACIÓN

función $multiplicar(n,m)$
$i \leftarrow 0;\ producto \leftarrow 0$
mientras $n > 0$ **hacer**
  $n_1 \leftarrow n\,\%base;\ mAux \leftarrow m;\ j \leftarrow i;\ llevo \leftarrow 0;$
  mientras $mAux > 0$ **hacer**
    $m_1 \leftarrow mAux\,\%base;$
    $productoDigito \leftarrow n_1 \cdot m_1 + llevo + producto[j]$
    $llevo \leftarrow productoDigito/base$
    $producto[j] \leftarrow productoDigito\,\%base$
    $j \leftarrow j+1;\ \ mAux \leftarrow mAux/base$
  fin
  $producto[j] \leftarrow llevo$
  $i \leftarrow i+1;\ n \leftarrow n/base;$
fin
devolver $producto$

La complejidad del algoritmo A.1.2 está en $\Theta(log\ n\ \ \cdot log\ m)$

Demostración.

El algoritmo anteriormente descrito consta solamente de operaciones elementales. Por tanto, cualquiera de las operaciones del bucle interno sirve como barómetro, y la complejidad del algoritmo viene dada por el número de iteraciones de ese bucle.

- El número de iteraciones del bucle externo (en n) es $\lfloor log_{base} n \rfloor + 1$ (en cada iteración se considera uno de sus dígitos).
- Análogamente, para cada iteración del bucle externo, el bucle interno (en m) consta de $\lfloor log_{base} m \rfloor + 1$ iteraciones.

Por tanto, el número total de veces que se ejecuta la sentencia barómetro es
$(\lfloor log_{base} n \rfloor + 1) \cdot (\lfloor log_{base} m \rfloor + 1) < 4 \cdot \lfloor log_{base} n \rfloor \cdot \lfloor log_{base} m \rfloor \leq 4 \cdot log\ n\ \cdot\ log\ m.$

Dado que nos interesa el comportamiento asintótico, en el segundo paso hemos considerado $n, m > base$, y por tanto $1 < \lfloor log\ n \rfloor\ \ ,\ \ \lfloor log\ m \rfloor$.

Algoritmo A.1.3. **DIVISION**

función $dividir(n,m)$
$i \leftarrow \lfloor log_{base}\, n \rfloor$; $resto \leftarrow 0$; $cociente \leftarrow 0$
mientras $i \geq 0$ **hacer**
    $resto \leftarrow resto * base + n[i]$
    $contador \leftarrow 0$
    mientras $resto \geq m$ **hacer**
        $resto \leftarrow resto - m$
        $contador \leftarrow contador + 1$
    fin
    $cociente \leftarrow cociente * base + contador$
    $i \leftarrow i - 1$
fin
devolver $cociente, resto$

La complejidad de A.1.3 está en $O(log\ n\ \cdot log\ m)$

Demostración.

El cálculo de $\leftarrow \lfloor log_{base}\, n \rfloor$ puede hacerse en $\leftarrow \lfloor log_{base}\, n \rfloor$ operaciones elementales (por ejemplo, dividiendo n sucesivamente por $base$ hasta llegar a un cociente menor que $base$).

El número de iteraciones del bucle externo es $\lfloor log_{base}\, n \rfloor + 1$. Es fácil comprobar que $resto < m$ y $contador < base$ son invariantes del bucle (son condiciones ciertas al inicio y al final de cada iteración). Por tanto, la operación $resto * base + n[i]$ arroja un número menor que $base \cdot m$, y para cada valor de i el bucle interno tiene menos de $base$ iteraciones. Su cuerpo contiene dos operaciones (una suma y una resta). La suma es una operación elemental ($contador$ consta de un único dígito. Por A.1.1, existe una constante a tal que el número de operaciones elementales para la resta es menor de $2 \cdot a \cdot log_{base}\, m$. El resto de operaciones del algoritmo son elementales ($resto * base + n[i]$ es una multiplicación por la base, seguida de la asignación del valor $n[i]$ al primer dígito -el menos significativo- del producto, y lo mismo ocurre con $cociente * base + contador$). Por tanto, el número de operaciones elementales requeridas es, como máximo,

$$(\lfloor log_{base}\, n \rfloor + 1) \cdot 2 \cdot a \cdot log_{base}\, m < 4 \cdot a \cdot log_{base}\, n \cdot log_{base}\, m$$

Dado que nos interesa el comportamiento asintótico, en el segundo paso hemos considerado $n > base$, y por tanto $1 < \lfloor log\, n \rfloor$.

Algoritmo A.1.4. **POTENCIA**

Seguiremos el algoritmo de exponenciación binaria. Lo describimos a continuación:

$$b^e = \begin{cases} 1 & si\ e = 0 \\ b^{e/2} \cdot b^{e/2} & si\ e\ par \\ b \cdot\ b^{e-1} & si\ e\ impar \end{cases} \tag{A.1}$$

La complejidad del algoritmo A.1.4 está en $O(e^2 \cdot log^2\ b)$

Demostración.
Se demuestra por inducción.
El caso base ($e = 0$) no requiere ninguna operación. Ahora distinguimos entre el caso par y el caso impar:
El cálculo de b^e para e par implica:

- calcular $b^{e/2}$. Por hipótesis de inducción, requiere un máximo de $\alpha \cdot \frac{e^2}{4} \cdot log^2\ b$ operaciones elementales, donde α es una constante.
- multiplicar $b^{e/2} \cdot b^{e/2}$. Por A.1.2 el número de operaciones elementales requeridas es, como máximo, $\beta \cdot log^2\ b^{e/2} = \beta \cdot \frac{e^2}{4} \cdot log^2\ b$.

Tomando $\alpha \geq \beta$, el número total de operaciones está acotado por $\frac{1}{2}\alpha \cdot e^2 \cdot log^2\ b$.

El cálculo de b^e para e impar implica:

- calcular b^{e-1}. Por hipótesis de inducción, requiere un máximo de $\alpha \cdot (e-1)^2 \cdot log^2\ b$ operaciones elementales, donde α es una constante.
- multiplicar $b \cdot b^{e-1}$. Por A.1.2 el número de operaciones elementales requeridas es, como máximo, $\beta \cdot (e-1) \cdot log^2\ b$.

Tomando $\alpha \geq \beta$, el número total de operaciones está acotado por $\alpha \cdot (e^2 - e) \cdot log^2\ b < \alpha \cdot e^2 \cdot log^2\ b$.

Algoritmo A.1.5. MÉTODO DE REDUCCIÓN DE BARRETT

En [4] se propone un algoritmo para calcular el resto $a\,\%n = a - n \cdot a/n$ más eficiente que el presentado previamente en A.1.3. En adelante intentaremos dar una idea intuitiva del mismo.

En primer lugar, hacemos notar que para todo entero λ se cumple que

$a\,\%n \equiv a - \lambda \cdot n (mod\ n)$. La propuesta consiste en encontrar un λ tal que $a - \lambda \cdot n$ sea muy próximo a $a\,\%n$.

La idea para encontrar ese λ se inspira en que $a\,\%n = a - n \cdot (a \cdot R)$, donde $R = 1 \div n$. Obviamente, R es un número real (en general mucho menor que 1), con una gran cantidad de decimales. La propuesta de Barrett consiste en "aproximar y escalar R, esto es, multiplicar R por alguna potencia de la base [del sistema de numeración en que trabajemos] y redondear para tomar la parte entera"(a esta parte entera la designaremos r). Un aspecto esencial es con qué precisión se redondea R: cuantos más dígitos se tomen, el cálculo será más preciso pero más costoso. La solución de compromiso es tomar $r \leftarrow base^{n_{Dig}}/n$. Hacemos notar que r tiene un máximo de $n_{Dig}+1$ dígitos, donde $n_{Dig} = \lfloor log_{base}\, n \rfloor + 1$ es el número de dígitos de n.

Las ventajas del método se basan en tres puntos:

- En el esquema criptográfico RSA, el divisor (la componente n de la clave pública) es siempre el mismo. r se puede calcular una única vez, en el mismo momento de conocer la clave.
- En el esquema RSA, cada vez que se requiere calcular un resto $a\,\%n$, a está acotado (es como mucho n^2).
- Existen algoritmos eficientes para la multiplicación.

función $restoBarrett(a,n)$
(suponemos ya calculados $n_{Dig} = \lfloor log_{base}\, n \rfloor + 1$, $r = base^{2\,n_{Dig}}/n$)
$y \leftarrow a/base^{n_{Dig}-1}$
$x \leftarrow a - n \cdot (r \cdot y/base^{nDig+1})$
mientras $x \geq n$ **hacer**
    $x \leftarrow x - n$
fin
devolver x

Si $a < n^2$, Barrett demuestra que el primer valor hallado de x es menor que $3n-1$: hacen falta como máximo 2 restas adicionales para obtener el resto módulo n. Más aún, en el 90 % de los casos el primer valor de x es ya menor que n: no se requieren restas adicionales. Sólo en el 1 % de los casos el primer valor de x es mayor que $2n$ (y se requieren, por tanto, dos restas adicionales).

Dejando aparte el coste de calcular r (se hace una única vez, por ejemplo al conocer la clave pública), el cálculo del resto implica esencialmente dos productos, dos divisiones por un múltiplo de la base y un máximo de cuatro restas. Los números implicados tienen un máximo de $n_{Dig}+1$ dígitos. Dado que la operación más costosa es el producto (ver las tablas 3.4 y 3.4), la complejidad del algoritmo viene dada por $M(n)$, la complejidad del producto de dos números del orden de n.

Algoritmo A.1.6. POTENCIA MÓDULO

Tomamos módulos en el algoritmo de exponenciación binaria:

$$b^e\,\%n = \begin{cases} 1 & si\ e = 0 \\ (\ (b^{e/2}\,\%n)\ (b^{e/2}\,\%n)\ )\,\%n & si\ e\ par \\ (\ b\ (b^{e-1}\,\%n)\ )\,\%n & si\ e\ impar \end{cases} \tag{A.2}$$

Suponemos $b < n$. Para el análisis de la complejidad de este algoritmo designaremos $M(n)$ a la complejidad del producto $n \cdot n$.

La complejidad del algoritmo A.1.6 está en $O(log\ e\ \cdot M(n))$

Demostración.
A diferencia del caso anterior, al tomar módulo n en todas las operaciones en todo momento estamos trabajando con números menores o iguales que n. Por tanto, el número de operaciones elementales requeridas para cada multiplicación está acotada por $\beta \cdot M(n)$.

El cálculo de $b^e\,\%n$ para e par implica:

- calcular $b^{e/2}\,\%n$. Por hipótesis de inducción, requiere un máximo de $\alpha \cdot log\ (e/2)\ \cdot M(n)$ operaciones elementales, donde α es una constante.
- multiplicar dos números menores de n, lo que requiere un máximo de $\beta \cdot M(n)$ operaciones elementales, con β constante.
- calcular el resto de dividir el producto anterior por n. Si se utiliza el algoritmo de Barrett (A.1.5), el número de operaciones elementales requerido es $\gamma M(n)$.

En total, el número de operaciones elementales es $\alpha \cdot log\ e \cdot M(n) + (\beta + \gamma - \alpha log\ 2) \cdot M(n)$. Basta tomar $\alpha \geq (\beta+\gamma)/(log 2)$ para obtener el resultado.

El cálculo de b^e para e impar implica:

- calcular $b^{e-1}\,\%n = (b^{(e-1)/2}\,\%n)\cdot(b^{(e-1)/2}\,\%n)\,\%n$. Por hipótesis de inducción, se requieren un máximo de:
  - $\alpha\cdot log\ (e-1)\ \cdot\ M(n)-\alpha\ \cdot\ log\ 2\ \cdot\ M(n) < \alpha\cdot log\ e\ \cdot\ M(n)-\alpha\cdot log\ 2\ \cdot\ M(n)$ operaciones para calcular $b^{(e-1)/2}\,\%n$,
  - $\beta M(n)$ operaciones para el producto,
  - $\gamma M(n)$ operaciones para el resto (ver A.1.5).
- multiplicar $b\cdot b^{e-1}\,\%n$, que requiere como máximo $\beta M(n)$ operaciones.
- calcular el resto, que requiere como máximo $\gamma M(n)$ operaciones.

En total, el número de operaciones elementales es menor de $\alpha\cdot log\,e\cdot M(n)+(2\beta+2\gamma-\alpha log\ 2)\cdot M(n)$. Basta tomar $\alpha\geq 2(\beta+\gamma)/(log 2)$ para obtener el resultado.

El siguiente resultado será utilizado en el análisis de los algoritmos que vienen a continuación.

Proposición A.1.7. *Sea $a_0, a_1 \ldots$ una sucesión de enteros no negativos. Si para todo $i>0$ se cumple que $a_i \leq a_{i-1}/2$, entonces la sucesión tiene un máximo de $1+\lfloor log_2\, a_0\rfloor$ términos.*

Demostración. *Es inmediato que la sucesión tiene un número finito de términos. Sea a_F el último (hay, por tanto, $F+1$ términos). De las propiedades de la sucesión se sigue que $1\leq a_F \leq a_0/2^F \Rightarrow F\leq log_2\, a_0$. Dado que F es entero, $F\leq \lfloor log_2 a_0\rfloor$.*

Algoritmo A.1.8. ES POTENCIA PERFECTA

Este algoritmo que resuelve la siguiente cuestión: ¿un número dado n es potencia perfecta? Es decir, ¿existen números b y α para los cuales se cumple que $b^\alpha = n$?.

Hacemos notar que existe una cota superior para los posibles valores de α: dado que $b\geq 2$, $\alpha \leq log_2\, n$.

La primera parte de nuestro programa es la siguiente:

función $esPotenciaPerfecta$
$\alpha \leftarrow 2$
mientras $\alpha \leq log_2\, n$ **hacer**
  si *existe b t.q.* $b^\alpha = n$ **entonces**
    devolver SI
  fin
  $\alpha \leftarrow \alpha + 1$
fin
devolver NO

En la segunda parte del algoritmo se responde a la pregunta: dados α y n, ¿existe un b tal que $b^\alpha = n$?. Para tratar de identificar ese valor de b seguimos el algoritmo de la búsqueda binaria. Hacemos notar que, de existir tal valor, está acotado por $b \leq 2^{\lceil log_2\, n/\alpha \rceil}$

$iMin \leftarrow 2$
$iMax \leftarrow 2^{\lceil \frac{log_2\, n}{\alpha} \rceil}$
mientras $iMax > iMin$ **hacer**
  $iMed \leftarrow \frac{iMin+iMax}{2}$
  calculo $iMed^\alpha$
  si $iMed^\alpha < n$ **entonces**
    $iMin \leftarrow iMed + 1$
  en otro caso
    $iMax \leftarrow iMed$
  fin
fin
si $iMin^\alpha = n$ **entonces**
  devolver SI
en otro caso
  devolver NO
fin

La complejidad del algoritmo A.1.8 está en $O(log^3 n \cdot log\ log\ n)$

Demostración.

El número de iteraciones de la búsqueda binaria es, como máximo, $log_2 iMax_0 + 1 = \lceil \frac{log_2 n}{\alpha} \rceil + 1$ ($iMax_0$ es el valor inicial de $iMax$). Para probarlo, consideremos la secuencia de enteros $\{a_i\}$, donde $a_i = iMax - iMin$ al inicio de la i-ésima iteración; es fácil de comprobar que $a_{i+1} \leq a_i/2$, con lo que se puede aplicar A.1.7.

En cada una de las iteraciones de esta búsqueda, la operación más costosa es el cálculo de $iMed^\alpha$; por A.1.4 sabemos que el cálculo de esta potencia requiere, como máximo, del orden de $\alpha^2 \cdot log_2^2\ iMed < \alpha^2 \cdot \lceil \frac{log_2 n}{\alpha} \rceil^2$ operaciones elementales.

Por tanto, para un valor determinado de α se requieren, como máximo, del orden de $\frac{1}{\alpha} \cdot log_2^3\ n$.

Pero como tenemos que probar con todos los valores de α, el número de operaciones elementales viene acotado por $log_2^3\ n \cdot \sum_{\alpha=2}^{log_2 n} \frac{1}{\alpha}$.

Podemos encontrar una cota superior para el sumatorio en α: $\sum_{\alpha=2}^{log_2 n} \frac{1}{\alpha} \leq \int_2^{log_2 n} \frac{1}{\alpha} d\alpha = ln\ log_2 n - ln\ 2 < ln\ log_2 n = ln\ 2 \cdot log_2\ log_2 n$, con lo que se obtiene el resultado.

Algoritmo A.1.9. EUCLIDES

El máximo común divisor de n y m, $mcd(n,m)$, es el mayor entero que divide exactamente tanto a uno como al otro. Existe un algoritmo eficiente para calcular el máximo común divisor conocido con el nombre de Algoritmo de Euclides.

Se basa en la siguiente propiedad: $mcd(n,m) = mcd(m, n\%m)$. Esta propiedad es sencilla de deducir: utilizando la relación $n\%m = n - n/m \cdot m$, es inmediato que un número es divisor común de (n,m) si y sólo si lo es de $(m, n\%m)$; si $d = mcd(n,m)$ y $e = mcd(m, n\%m)$, d no puede ser mayor que e y e no puede ser mayor que d.

El algoritmo de Euclides va tomando pares sucesivos (n_i, m_i), donde $(n_0, m_0) = (n,m)$, $n_{i+1} = m_i$, $m_{i+1} = n_i\%m_i$ (por tanto $mcd(n,m) = mcd(n_i, m_i)$ para todo i). En cada paso, obtenemos un par de números más sencillos hasta llegar al caso base ($(n_F, m_F = 0)$), cuyo máximo común divisor es inmediato (n_F).

función $Euclides(n,m)$
mientras $m > 0$ **hacer**
    $t \leftarrow m$
    $m \leftarrow n\,\%m$
    $n \leftarrow t$
fin
devolver n

La complejidad del algoritmo A.1.9 está en $O(log\ n\ \cdot\ M(n))$

Demostración.
Suponemos sin pérdida de generalidad que $m \geq n$ puesto que en caso contrario la primera pasada por el bucle intercambia n y m. Empezamos por probar que siempre es cierto que $n\,\%m \leq n/2$:

- Si $m \leq n/2$ entonces $n\,\%m < m \leq n/2$
- Si $m > n/2$ entonces el cociente $n/m = 1$ lo cual implica que $n\,\%m = n-(n/m)m = n-m < n/2$

Definimos la secuencia de enteros $\{a_i\}$, donde a_i es el producto $n \cdot m$ al inicio de la i-ésima iteración del bucle. Del resultado anterior se sigue que $a_{i+1} \leq a_i/2$; aplicando A.1.7, el número de iteraciones está acotado por $log_2\ (n \cdot m) \leq 2\ log_2\ n$. En cada iteración, la sentencia más costosa es el cálculo del resto; según hemos visto en A.1.5, la complejidad de esta operación está dada por $M(n)$.

Algoritmo A.1.10. INVERSO MÓDULO EUCLIDES

Existe inverso modular de n módulo M si $n^{-1}n \equiv 1 \pmod{M}$. De hecho existe si y sólo si $mcd(n,M) = 1$.

Es posible extender el algoritmo de Euclides para calcular el inverso de m módulo n. Recordemos que el algoritmo genera una sucesión (n_i, m_i), donde $n_{i+1} = m_i$, $m_{i+1} = n_i\,\%m_i$. La idea es generar además otra sucesión de coeficientes, t_i, tales que $n_i \equiv t_i \cdot m\ (mod\ n)$. El máximo común divisor de (n,m) es el último valor de n_i, n_F, con lo que $n_F = mcd(n,m) \equiv t_F \cdot m\ (mod\ n)$. Si n y m son coprimos, t_F es el inverso de m módulo n. Para calcular t_i podemos fundir las relaciones de recurrencia de n_i, m_i en una sola: $n_0 = n$, $n_1 = m$; para $i \geq 2$, $n_i = m_{i-1} = n_{i-2}\,\%m_{i-2} = n_{i-2}\,\%n_{i-1} = n_{i-2} - n_{i-2}/n_{i-1} \cdot n_{i-1}$. De ahí es inmediato obtener la relación de recurrencia de las t_i: $t_0 = 0$, $t_1 = 1$; para $i \geq 2$, $t_i = t_{i-2} - c_{i-1} \cdot t_{i-1}$, donde $c_{i-1} = n_{i-2}/n_{i-1}$.

función *inversoModuloEuclides*(n,m)
$n_0 \leftarrow n$
$\{t_{i-1}, c_{i-1}\} \leftarrow \{1,0\}$
$\{t_i, c_i\} \leftarrow \{0,0\}$
mientras $m > 0$ **hacer**
    $\{c_{i-1}, c_i\} \leftarrow \{c_i, n/m\}$
    $auxT \leftarrow t$;
    $t_i \leftarrow t_i - c_{i-1} \cdot t_{i-1}$;
    $t_{i-1} \leftarrow auxT$;
    $\{n,m\} \leftarrow \{m, n\,\%m\}$
fin
devolver t_i

La complejidad del algoritmo A.1.10 está en $O(log\, n \cdot M(n))$

Demostración.

El algoritmo se forma añadiendo al de Euclides las sentencias requeridas para calcular la secuencia t_i. De ellas, la más costosa es el producto de dos números menores que n: $c_{i-1} \cdot t_{i-1}$, donde $c_{i-1} = n_{i-2}/n_{i-1} \leq n$. La complejidad esta operación es $M(n)$ (para el caso del resto, ver A.1.5), la misma que la de $n\,\%m$ cuya complejidad es $M(n)$. Por tanto, los cálculos añadidos no modifican la complejidad del algoritmo.

Al finalizar el bucle se obtiene un valor t_F cuyo módulo es menor que n. Para obtener un resultado no negativo, es preciso hacer, como máximo, una suma extra.

Algoritmo A.1.11. MÁXIMO COMÚN DIVISOR BINARIO

Un método alternativo para calcular el máximo común divisor es el método binario que utiliza sólo resta y división por 2. Se basa en las siguientes propiedades del máximo común divisor:

$$mcd(n,m) = mcd(n-m,m) \tag{A.3}$$

$$mcd(n,m) = \begin{cases} 2 \cdot mcd(n/2, m/2) & si\ ambos\ son\ pares \\ mcd(n/2, m) & si\ sólo\ n\ es\ par \\ mcd(n, m/2) & si\ sólo\ m\ es\ par \end{cases} \tag{A.4}$$

El algoritmo es el siguiente:

función $mcdBinario(n,m)$
$d \leftarrow 1$
mientras $n\%2 = 0\ AND\ m\%2 = 0\ AND\ n \neq 0\ AND\ m \neq 0$ **hacer**
    $\{n,m,d\} \leftarrow \{n/2, m/2, 2 \cdot d\}$
fin
mientras $n\%2 = 0\ AND\ n \neq 0$ **hacer**
    $n \leftarrow n/2$
fin
mientras $m\%2 = 0\ AND\ m \neq 0$ **hacer**
    $m \leftarrow m/2$
fin
mientras $n > 0$ **hacer**
    si $n\%2 = 0$ **entonces**
        $n \leftarrow n/2$
        en otro caso
            $ordena(n,m)$
            $n \leftarrow (n-m)/2$
        fin
    fin
fin
devolver $d \cdot m$

La complejidad del algoritmo A.1.11 está en $O(log^2\ n)$

Demostración. Sin pérdida de generalidad, suponemos $n > m$. El algoritmo consta de varios bucles, cada una de cuyas iteraciones recibe un par (n_i, m_i) y genera un nuevo par (n_{i+1}, m_{i+1}). Sea $a_i = n_i \cdot m_i$. Es inmediato que $a_{i+1} \leq a_i/2$. Por A.1.7, el número máximo de iteraciones es $1 + \lfloor log\ (n \cdot m) \rfloor < 1 + \lfloor 2 \cdot log\ n \rfloor$. En cada iteración se realiza una división por 2 (es una operación elemental) o una resta, cuya complejidad viene dada por $log(n)$. Por tanto, el número máximo de operaciones elementales es un múltiplo de $O(log^2\ n)$.

Algoritmo A.1.12. INVERSO MÓDULO BINARIO

De modo análogo a lo hecho con el algoritmo de Euclides, es posible extender el del máximo común divisor binario para calcular el inverso de m módulo n. Presentamos un algoritmo aplicable cuando n es impar (lo que ocurre siempre en la criptografía RSA). En el algoritmo A.1.11, al final del primer bucle tenemos un par $(n_i = n_0/2^l, m_i = m_0/2^l)$, donde l es la máxima potencia de 2 que es divisor común de n_0 y m_0. En ese punto introducimos dos nuevas variables, t y v, que cumplen los siguientes invariantes: $n_i \equiv 2^l \cdot t \cdot m_0\ (mod\ n_0)$, $m_i \equiv 2^l \cdot v \cdot m_0\ (mod\ n_0)$. Dado que el máximo común divisor de (n_0, m_0) es $2^l \cdot m_F$, t es el inverso de m_0 módulo n_0.

En el algoritmo utilizaremos la función $medio(a,n)$, que devuelve la mitad de a módulo n para n impar: $2 \cdot medio(a,n) \equiv a\ (mod\ n)$, y se define:

$$medio(a,n) = \begin{cases} a/2 & si\ a\ es\ par \\ (a+n)/2 & si\ a\ es\ impar \end{cases} \tag{A.5}$$

La complejidad del algoritmo A.1.12 está en $O(log^2\ n)$

Demostración. Las operaciones añadidas al algoritmo A.1.11 son de complejidad análoga a las ya existentes. Por tanto, la complejidad total no varía.

función $inversoModuloBinario(n,m)$
$d \leftarrow 1$
mientras $n\%2 = 0\ AND\ m\%2 = 0\ AND\ n \neq 0\ AND\ m \neq 0$ **hacer**
    $\{n,m,d\} \leftarrow \{n/2, m/2, 2 \cdot d\}$
fin
$\{t,v\} \leftarrow \{0,1\}$
mientras $n\%2 = 0\ AND\ n \neq 0$ **hacer**
    $n \leftarrow n/2$
    $t \leftarrow medio(t,n_0)$
fin
mientras $m\%2 = 0\ AND\ m \neq 0$ **hacer**
    $m \leftarrow m/2$
    $t \leftarrow medio(t,n_0)$
fin
si $n\%2 = 0$ **entonces**
    $\{n,m,t,v\} \leftarrow \{m,n,v,t\}$
fin
mientras $n > 0$ **hacer**
    si $n\%2 = 0$ **entonces**
        $n \leftarrow n/2$
        $t \leftarrow medio(t,n_0)$
        en otro caso
            si $n < m$ **entonces**
                $\{n,m,t,v\} \leftarrow \{m,n,v,t\}$
            fin
            $n \leftarrow (n-m)/2$
            $t \leftarrow medio((t-v),n_0$
        fin
    fin
fin
$mcd \leftarrow d \cdot m$
devolver t

Algoritmo A.1.13. **SÍMBOLO DE JACOBI**

En una sección previa ya hemos introducido el símbolo de Jacobi. A continuación presentamos un algoritmo para su cálculo, basado en sus propiedades:

función $jacobi(a, n\ impar)$
$contador \leftarrow 0$
$a \leftarrow a\%n$
si $a = 0$ **entonces**
  devolver 0
fin
mientras $a\%2 = 0$ **hacer**
  $a \leftarrow a/2$
  $contador \leftarrow contador + 1$
fin
devolver $(-1)^{\frac{n^2-1}{8} contador}(-1)^{\frac{a-1}{2}\frac{n-1}{2}} jacobi(n,a)$

La complejidad del algoritmo A.1.13 está en $O(log\ a \cdot M(n))$

Demostración.

Sin pérdida de generalidad, suponemos $a > n$. El algoritmo se reduce al cálculo del símbolo para pares (a_i, n_i), que cumplen la siguiente condición:

$$\begin{cases} a_{i+1} = n_i \\ n_{i+1} < \frac{a_i}{2} \end{cases} \tag{A.6}$$

Definiendo $b_i = a_i \cdot n_i$, obtenemos que $b_{i+1} < b_i/2$. Por A.1.7, la función se invoca un máximo de $1 + \lfloor log\ (a \cdot n) \rfloor < 1 + \lfloor 2\ \cdot\ log\ a \rfloor$ veces. En cada invocación, se ejecutan las siguientes sentencias:

- El resto $a\%n$, cuya complejidad está en $M(n)$ (ver A.1.5).
- La división $a/2$, que se repite un máximo de $1 + \lfloor log_2\ a \rfloor$ veces. Cada división es una operación elemental.
- El cálculo de $(-1)^{\frac{a-1}{2}\frac{n-1}{2}}$. Notamos que el algoritmo garantiza que tanto a como n son impares en el momento en que se hace ese cálculo. Ese valor es -1 si $\frac{a-1}{2}$ y $\frac{n-1}{2}$ son impares, 1 en otro caso. Determinar esa paridad puede considerarse una sentencia elemental, como razonaremos a continuación.

  Dado un número impar m, $\frac{m-1}{2}$ es par si $m \equiv 1\ (mod\ 4)$, e impar si $m \equiv 3\ (mod\ 4)$. Por tanto, para determinar la paridad de $\frac{m-1}{2}$ basta tomar sus dos dígitos menos significativos (suponemos base 2), operación que puede considerarse elemental.

- Por último, $(-1)^{\frac{n^2-1}{8}} = 1$ si $n \equiv 1$ *o* $7 \pmod 8$, y -1 si $n \equiv 3$ *o* $5 \pmod 8$. Es decir, calcular $(-1)^{\frac{n^2-1}{8}}$ es equivalente a calcular $n\%8$ (tomar sus tres dígitos menos significativos, si estamos en base 2) y comparar el resto con 1, 3, 5 ó 7. Este cálculo puede considerarse elemental.

Por tanto, el número de operaciones elementales está acotado por un múltiplo constante de $O(log\, a \cdot M(n))$.

Algoritmo A.1.14. CONDICIÓN DE MILLER

Dados a, n devuelve 0 si a es testigo de que n es compuesto (con el criterio de Miller), 1 en otro caso.

función $condicionMiller(a, n\ impar)$
si $n\%a = 0$ **entonces**
  devolver 0
  en otro caso
    si $a^{n-1}\%n \neq 1$ **entonces**
      devolver 0
      en otro caso
        $k \leftarrow 1$
        mientras 2^k *es divisor de* $(n-1)$ **hacer**
          si $mcd((a^{\frac{n-1}{2^k}}\%n) - 1, n) \neq 1, n$ **entonces**
            devolver 0
          fin
          $k \leftarrow k + 1$
        fin
      fin
    fin
  fin
fin
devolver 1

La complejidad del algoritmo A.1.14 está en $O(log^2\, n \cdot M(n))$

Demostración.

El cociente $a\%n$ conlleva, como máximo, del orden de $M(n)$ operaciones elementales (ver A.1.5).

El cálculo $a^{n-1}\,\%n$ conlleva, como máximo, del orden de $log\, n \cdot M(n)$ operaciones elementales (ver A.1.6).

El bucle tiene un máximo de $log_2\, n$ iteraciones. En cada una de ellas se calcula una potencia módulo n y un máximo común divisor ($log^2\, n$ operaciones, ver A.1.11). Por tanto supone, como máximo, del orden de $log^2\, n \cdot M(n)$ operaciones elementales.

Es posible diseñar un algoritmo más eficiente que el anterior para evaluar la condición de Miller. Este nuevo algoritmo no requiere el cálculo del máximo común divisor, y la única potencia que usa es el cuadrado. Se apoya en los siguientes resultados:

Proposición A.1.15. *Dados dos enteros, x, n:*

i) $mcd(x \bmod n - 1, n) = mcd(x-1, n)$

ii) $mcd(x-1,n) = n \Leftrightarrow x \equiv 1 \pmod n$

iii) *Si n es impar y* $x \equiv -1 \pmod n$, *entonces* $mcd(x-1,n) = 1$.

Demostración.

i) *Basta tener en cuenta que un entero es divisor común de* (m,n) *sii es divisor común de* $(m - a \cdot n, n)$ *para cualquier a.*

ii) $mcd(x-1,n) = n$ *sii n divide a* $x-1$*; esa es precisamente la definición de congruencia.*

iii) *Por hipótesis, x es congruente con* $n-1$. *Por tanto,* $mcd(x-1,n) = mcd(n-2,n) = d$. *Dado que n es impar, d debe serlo también. Existen enteros* a,b *tales que* $n-2 = a \cdot n$ *y* $n = b \cdot n$, *y por tanto* $(b-a) \cdot d = 2$. *El único entero impar d que satisface esa condición es 1.*

Proposición A.1.16. *Sean n impar y una sucesión* $x_0, \ldots x_k$ *tal que* $x_{i+1} = x_i^2$. *Sea j el primer índice para el cual* $x_{j+1} \equiv 1 \pmod n$ *(si todos los términos son congruentes con 1, tomamos* $j = -1$*). Entonces:*

i) *Para todo* $i > j$ *se cumple que* $mcd(x_i \bmod n - 1, n) = n$.

ii) *Si* $x_j \equiv -1 \pmod n$, *entonces* $mcd(x_i \bmod n\ -1, n) = 1$ *para todo* $i \leq j$.

iii) *Si* $x_j \not\equiv -1 \pmod n$, *entonces* $mcd(x_j \bmod n\ -1, n) = 1$

Demostración.

i) *Se sigue inmediatamente de los puntos i y ii de A.1.15.*

ii) El caso $i = j$ se sigue inmediatamente de A.1.15.iii. Para los demás índices $i < j$, hacemos notar que $x_j - 1$ es múltiplo de $x_i - 1$. En efecto, $x_j = x_i^{\alpha}$ (concretamente, $\alpha = 2 \cdot (j-i)$) y, por tanto, $x_j - 1 = x_i^{\alpha} - 1 = (x_i - 1) \cdot (\sum_{l=0}^{\alpha-1} x_i^l)$. Por tanto,

$$mcd(x_i \bmod n - 1, n) = mcd(x_i - 1, n) \leq mcd(x_j - 1, n) = mcd(x_j \bmod n - 1, n) = 1.$$

iii) Por definición de j, $x_{j+1} = x_j^2 \equiv 1 \ (mod\ n)$. Esto es, $(x_j + 1) \cdot (x_j - 1)$ es múltiplo de n. Probaremos el resultado por reducción al absurdo; supongamos $mcd(x_j - 1, n) = 1$. Por el lema de Euclides, n es divisor de $(x_j + 1)$, y por tanto $x_j \equiv -1 \ (mod\ n)$, lo cual es una contradicción.

Algoritmo A.1.17. **CONDICIÓN DE MILLER (ALGORITMO MEJORADO)**

El algoritmo se apoya en la proposición A.1.16. En primer lugar, calcula $k = \#_2(n-1)$ (la máxima potencia de 2 que divide a n-1) y descompone $n - 1 = 2^k \cdot d$. A continuación, considera la sucesión dada por $x_i = a^{2^i \cdot d} = a^{(n-1)/2^{k-i}}$, donde i recorre el intervalo $[0, k-1]$. Observamos que esta sucesión cumple las condiciones de A.1.17. La congruencia de Fermat ($a^{n-1} \equiv 1 \ (mod\ n)$) se satisface sii $j < k$, donde j es el primer índice para el cual $x_{j+1} \equiv 1 \ (mod\ n)$. El algoritmo consiste en buscar ese índice j. Hay tres situaciones posibles:

- $j = -1$: todos los términos satisfacen $x_i \equiv 1 \ (mod\ n)$. Por A.1.16.i, todos los términos de la sucesión satisfacen $mcd(x_i \bmod n - 1, n) = n$
- $0 \leq j < k$ y $x_j \equiv -1 \ (mod\ n)$. Por A.1.16, todos los términos de la sucesión con $i > j$ satisfacen $mcd(x_i \bmod n - 1, n) = n$, y todos los términos con $i \leq j$ satisfacen $mcd(x_i \bmod n - 1, n) = 1$.
- $0 \leq j < k$ y $x_j \not\equiv -1 \ (mod\ n)$. Por A.1.16.iii, x_j incumple la condición de Miller.

función *condicionMillerMejorado*$(a, n\ impar)$
$d \leftarrow n-1$
$k \leftarrow 0$
mientras *d es divisible por 2* **hacer**
    $d \leftarrow d/2$
    $k \leftarrow k+1$
fin
$x \leftarrow a^d \,\%n$
si $x = 1$ *OR* $x = n-1$ **entonces**
    devolver 1 **en otro caso**
        para $i \leftarrow 1$ ***hasta*** $k-1$ ***con paso*** 1 **hacer**
            $x \leftarrow x^2 \,\%n$
            si x = n - *1* **entonces**
                devolver 1
            fin
            si x = *1* **entonces**
                devolver 0
            fin
        fin
        devolver 0
    fin
fin

La complejidad del algoritmo A.1.17 está en $O(log\ n \cdot M(n))$

Demostración. El primer bucle contiene un máximo de $1+\lfloor log_2 n \rfloor$ iteraciones. En cada una de ellas de ejecuta una división por 2 (operación elemental) y una suma, cuyo coste es del orden de $logn$ (ver A.1.1). La complejidad de ese bucle está, pues, en $O(log^2 n)$.

El segundo bucle contiene un máximo de $1+\lfloor log_2 n \rfloor$ iteraciones. En ellas, la operación más costosa es el cálculo $x^2 \,\%n$, cuyo coste computacional es del orden de $M(n)$ (ver A.1.6).

Por tanto, la complejidad de todo el algoritmo está en $O(log\ n \cdot M(n))$.

Algoritmo A.1.18. **TEST DE MILLER (DETERMINISTA)**

El test de Miller, tal como se presenta en [12] y suponiendo la cota $2 \cdot log^2 n$ para el primer testigo dada por Ankeny [1], es:

función *miller*$(n\ impar)$
si *n es potencia perfecta* **entonces**
  devolver *compuesto*
fin
para $a \leftarrow 2$ ***hasta*** $2 \cdot log^2 n$ ***con paso*** 1 **hacer**
  si *condicionMillerMejorada (a,n) = 0* **entonces**
    devolver *compuesto*
  fin
fin
devolver *primo*

La complejidad del algoritmo A.1.18 está en $O(log^3 n \cdot M(n))$

Demostración.
Primero comprobar si n es potencia perfecta costará $O(log^3 n \cdot log\ log\ n)$ operaciones elementales (ver Algoritmo A.1.8).
Por otro lado, el bucle *Para* hace como mucho $2 \cdot log^2 n$ iteraciones. El coste de cada una está en $O(log\ n \cdot M(n))$ (ver A.1.17). En resumen, el número de operaciones viene dado por
$t(n) < \alpha \cdot log^3 n \cdot log\ log\ n + \beta \cdot log^2 n \cdot log\ n \cdot M(n)$ con $\alpha, \beta, k \in \mathbb{R}^+$. Basta tomar un valor de $k > \alpha + \beta$ para tener $t(n) < k \cdot log^3 n \cdot M(n)$.

Algoritmo A.1.19. **TEST DE MILLER-RABIN (PROBABILISTA)**

Si la condición de Miller de este algoritmo mejorado (A.1.17) devuelve 0, está claro que el número n es compuesto. Si devuelve 1, no hay certeza sobre su primalidad. Rabin [13] demuestra que en este último caso la probabilidad de que n sea compuesto es menor de 1/4. Por tanto, si tras repetir el algoritmo t veces obtenemos siempre 1, la probabilidad de error (de que si n es primo el test no lo haya detectado) es menor de 2^{-2t}. Como comenta en su artículo *"Supongamos que tomamos* $k = 30$ *[...]. La probabilidad de error es* $1/2^{60} < 10^{-18}$. *¡Un error esperado en un miliardo de miliardo de pruebas! Es un error pequeño en comparación con la frecuencia de errores debidos al mal funcionamiento de la máquina en este tipo de cálculos"*.

función *millerRabin*$(n\ impar,\ t$ entero positivo menor que n$)$
para $a \leftarrow 2$ ***hasta*** t ***con paso*** 1 **hacer**
  si *condicionMillerMejorada (a,n) = 0* **entonces**
    devolver *compuesto*
  fin
fin
devolver *probablemente primo*

La complejidad del algoritmo A.1.19 está en $O(t \cdot log\ n \cdot M(n))$

Demostración. (ver [13])
El bucle *Para* hace como mucho t iteraciones. La complejidad de cada iteración es la misma que la condición de Miller de este algoritmo mejorado (en esencia, son las mismas operaciones). En resumen,

$$t(n) < t \cdot log\ n \cdot M(n)$$

Algoritmo A.1.20. **CONDICIÓN DE SOLOVAY-STRASSEN**

Dados a,n devuelve 0 si a es testigo de que n es compuesto (con el criterio de Solovay-Strassen, ver 3.3), 1 en otro caso.

función *condicionSolovayStrassen*$(a,n\ impar)$
si $mcd(a,n) = 1\ AND\ a^{\frac{n-1}{2}} \equiv \left(\frac{a}{n}\right)\ (\text{mód } n)$ **entonces**
  devolver 1
fin
en otro caso
  devolver 0
fin

La complejidad del algoritmo A.1.20 está en $O(log\ n \cdot M(n))$

Demostración. (ver [16])
El máximo común divisor supone $log^2\ n$ operaciones (ver A.1.11).
El cálculo $a^{\frac{n-1}{2}}\ \%n$ conlleva, como máximo, del orden de $log\ n \cdot M(n)$ operaciones elementales (ver A.1.6).
El cálculo $\left(\frac{a}{n}\right)$ conlleva, como máximo, del orden de $log\ a \cdot M(n)$ operaciones elementales (ver A.1.13). Por tanto, el cálculo $\left(\frac{a}{n}\right)\ (\text{mód } n)$ conlleva, como máximo, del orden de $log\ a\ log\ n \cdot M(n)$ operaciones elementales (ver A.1.13).
En resumen supone, como máximo, del orden de $log\ n \cdot M(n)$ operaciones elementales.

Algoritmo A.1.21. TEST DE SOLOVAY-STRASSEN (DETERMINISTA)

El test de Solovay-Strassen, tal como se presenta en [16] y suponiendo la cota $2 \cdot log^2 n$ para el primer testigo dada por Ankeny [1], es:

función *solovayDeterminista*(n *impar*)
para $a \leftarrow 2$ ***hasta*** $2 \cdot log^2 n$ ***con paso*** 1 **hacer**
  si *condicionSolovayStrassen (a,n) = 0* **entonces**
    devolver *compuesto*
  fin
fin
devolver *primo*

La complejidad del algoritmo A.1.21 está en $O(log^3 n \cdot M(n))$

Demostración.
El bucle *Para* hace como mucho $2 \cdot log^2 n$ iteraciones. El coste de cada una está en $O(log^2 n \cdot M(n))$ (ver A.1.14). Por tanto, el número de operaciones viene dado por
$t(n) < \alpha \cdot log^2 n \cdot log\, n \cdot M(n)$ con $\alpha \in \mathbb{R}^+$. Es decir, $t(n) < \alpha \cdot log^3 n \cdot M(n)$.

Algoritmo A.1.22. TEST DE SOLOVAY-STRASSEN (PROBABILISTA)

Si la condición de Solovay-Strassen (A.1.20) devuelve 0, está claro que el número n es compuesto. Si devuelve 1, no hay certeza sobre su primalidad. Solovay-Strassen [16] demuestra que en este último caso la probabilidad de que n sea compuesto es menor de 1/2. Por tanto, si tras repetir el algoritmo t veces obtenemos siempre 1, la probabilidad de error (de que si n es primo el test no lo haya detectado) es menor de 2^{-t}.

función *solovayProbabilista*(n impar, t entero positivo menor que n)
para $a \leftarrow 2$ ***hasta*** t ***con paso*** 1 **hacer**
  si *condicionSolovayStrassen (a,n) = 0* **entonces**
    devolver *compuesto*
  fin
fin
devolver *probablemente primo*

La complejidad del algoritmo A.1.22 está en $O(t \cdot log\, n \cdot M(n))$

Demostración. (ver [16])
El bucle *Para* hace como mucho t iteraciones. La complejidad de cada iteración es la misma que la condición de Solovay-Strassen (en esencia, son las mismas operaciones). En resumen,

$$t(n) < t \cdot \log n \cdot M(n)$$

Bibliografía

[1] NC Ankeny. «The least quadratic non residue». En: *Annals of mathematics* (1952), págs. 65-72.

[2] Tom M Apostol. *Introduction to analytic number theory*. Vol. 1. Springer Science & Business Media, 1976.

[3] Eric Bach. «Explicit bounds for primality testing and related problems». En: *Mathematics of Computation* 55.191 (1990), págs. 355-380.

[4] Paul Barrett. «Implementing the Rivest Shamir and Adleman Public Key Encryption Algorithm on a Standard Digital Signal Processor.» En: *Crypto*. Vol. 86. Springer. 1986, págs. 311-323.

[5] Gilles Brassard y Paul Bratley. *Fundamentals of algorithmics*. Vol. 133350681. Prentice Hall Englewood Cliffs, 1996.

[6] Richard Crandall y Carl Pomerance. *Prime numbers: a computational perspective*. Vol. 182. Springer Science & Business Media, 2006.

[7] Sebastián Xambó Descamps, Félix Delgado y Concha Fuertes. *Introducción al álgebra*. Vol. 1. Editorial Complutense, 1993.

[8] Whitfield Diffie y Martin E Hellman. «New directions in cryptography». En: *Information Theory, IEEE Transactions on* 22.6 (1976), págs. 644-654.

[9] Uriel Feige, Amos Fiat y Adi Shamir. «Zero-knowledge proofs of identity». En: *Journal of cryptology* 1.2 (1988), págs. 77-94.

[10] Luis Carlos Coronado Garcıa. *Can Schönhage multiplication speed up the RSA decryption or encryption?* 2007.

[11] Anatolii Karatsuba y Yu Ofman. «Multiplication of multidigit numbers on automata». En: *Soviet physics doklady*. Vol. 7. 1963, pág. 595.

[12] Gary L Miller. «Riemann's hypothesis and tests for primality». En: *Journal of computer and system sciences* 13.3 (1976), págs. 300-317.

[13] Michael O Rabin. «Probabilistic algorithm for testing primality». En: *Journal of number theory* 12.1 (1980), págs. 128-138.

[14] Ronald L Rivest, Adi Shamir y Len Adleman. «A method for obtaining digital signatures and public-key cryptosystems». En: *Communications of the ACM* 21.2 (1978), págs. 120-126.

[15] Doz Dr A Schönhage y Volker Strassen. «Schnelle multiplikation grosser zahlen». En: *Computing* 7.3-4 (1971), págs. 281-292.

[16] Robert Solovay y Volker Strassen. «A fast Monte-Carlo test for primality». En: *SIAM journal on Computing* 6.1 (1977), págs. 84-85.