Optimización de exponenciaciones modulares en el criptosistema RSA mediante exponentes recodificados

Tesis de licenciatura

Autor: Diego J. Sánchez Navarro

Director: Dr. Hugo D. Scolnik

Departamento de Computación
Facultad de Ciencias Exactas y Naturales
UNIVERSIDAD DE BUENOS AIRES

Diciembre 2002

Índice

Índice **2**
Agradecimientos **3**
1 Introducción **4**
2 El criptosistema RSA **6**
3 Exponenciación modular **8**
3.1 Exponenciaciones modulares directas 8
3.2 Método binario de exponenciación modular 9
3.3 Método m-ario de exponenciación modular 10
3.4 Métodos *m*-arios adaptativos 11
3.5 Algoritmos de ventana deslizante 12
4 Métodos de recodificación **15**
4.1 El algoritmo de Booth y sus modificaciones 16
4.2 Recodificación canónica 17
5 Inversa modular **19**
5.1 La inversa modular de Montgomery 19
5.2 Inversa modular de Montgomery modificada (Savas-Koç) 22
5.3 Inversa modular mediante el algoritmo de MCD extendido de Lehmer 22
5.4 Inversa modular mediante el algoritmo de MCD binario extendido 24
6 Exponenciación modular utilizando exponentes recodificados **25**
6.1 Método binario adaptado para exponentes recodificados 27
6.2 Precomputación en métodos *m*-arios 28
6.3 Método *m*-ario adaptado para exponentes recodificados 30
6.3 Precomputación en métodos de ventana deslizante 31
6.5 Método de ventana deslizante adaptado para exponentes recodificados 34
7 Conclusiones **38**
Bibliografía **41**

Agradecimientos

A mi esposa Natalia, por su completo apoyo e infinita paciencia durante la realización de este trabajo.

A mi familia, por haber hecho de mí quien soy.

A mi director, Dr. Hugo Scolnik, por haberme aceptado como su alumno de tesis y haberme provisto de inestimable guía a través de sus conocimientos y experiencia.

A los docentes y no docentes del Departamento de Computación de la Facultad de Ciencias Exactas y Naturales de la Universidad de Buenos Aires, por la excelente educación recibida durante todos estos años.

A mis compañeros de estudios, a quienes ahora tengo el honor de llamar amigos.

1 Introducción

La exponenciación modular es una operación fundamental en numerosas aplicaciones de la computación, particularmente en lo que respecta a la criptografía de clave pública. Numerosos esfuerzos, tanto en el mundo académico como en el profesional, están siendo dedicados a optimizar y acelerar el cálculo de esta operación, principalmente en el campo de algoritmos paralelos, orientados para su aplicación directa en microprocesadores criptográficos. Sin embargo, la investigación en el campo de los algoritmos tradicionales continúa siendo de gran interés, tanto teórico como práctico.

El presente trabajo se concentra específicamente en la exploración y optimización de métodos aplicables al criptosistema RSA (§2), en el que las exponenciaciones modulares son la base tanto en el proceso de encriptación como en el desencriptación. En particular, se investigará la utilidad de modificar los métodos tradicionales de exponenciación modular (§3), adaptándolos para utilizar exponentes recodificados mediante bits con signo (§4). La necesidad inherente de calcular la inversa modular de la base, tradicionalmente una operación computacionalmente costosa, significó que este tipo de métodos modificados no hayan sido explorados en profundidad en la literatura.

Los principales aportes del presente trabajo consisten en investigar los resultados de una implementación práctica de estos métodos. Se exploran los resultados conseguidos mediante la utilización de algoritmos de cálculo de inversa modular optimizados para su utilización en computadores, tales como la inversa modular de Montgomery modificada, el algoritmo de MCD extendido de Lehmer y el algoritmo de MCD binario extendido (§5).

Se incluye además un análisis completo original de las precomputaciones necesarias en los métodos *m*-arios y de ventana deslizante, tanto para los algoritmos clásicos como para las variaciones que utilizan exponentes recodificados. Dentro de este aparte, se analiza en profundidad un método de precomputación óptimo para los algoritmos modificados, presentando una fórmula para la cantidad exacta de multiplicaciones a realizar (§6). Esto permite contrastar de manera precisa los ahorros conseguidos en la exponenciación propiamente dicha versus la precomputación adicional necesaria.

Los resultados obtenidos muestran que este tipo de algoritmos modificados pueden generar ahorros en el tiempo necesario para el cálculo de las exponenciaciones modulares, dependiendo del tamaño de los operandos y el método a utilizar. En ambientes donde existe un espacio acotado y se buscan métodos con precomputaciones reducidas o nulas, los métodos modificados para utilizar exponentes recodificados pueden ofrecer una alternativa atractiva para acelerar las exponenciaciones modulares. Un ejemplo de este tipo de situación puede encontrarse en la computación de encriptaciones y desencriptaciones RSA en una *smart card*.

El trabajo se organiza de la siguiente manera: En §2 se describe a grandes rasgos el criptosistema RSA, mostrando el rol fundamental de la exponenciación modular. En §3 se describen varios algoritmos de exponenciación modular, especialmente aptos para ser utilizados en el ambiente del criptosistema RSA. El concepto de recodificación se introduce en §4, seleccionando un método óptimo para ser implementado en todos los algoritmos modificados. Los algoritmos de inversa modular a ser implementados se

describen y analizan en §5. Los algoritmos modificados se describen en §6, presentando junto a estos últimos los resultados obtenidos en dicha implementación y un completo análisis. Por último, la conclusión en §7 resume los conceptos desarrollados, presentando posibles áreas de mayor investigación y problemas abiertos de interés.

2 El criptosistema RSA

El algoritmo criptográfico RSA fue introducido a finales de la década del 70 y debe su nombre a sus inventores, R. L. Rivest, A. Shamir y L. Adleman [1]. Este algoritmo es generalmente considerado como el trabajo seminal en el área de la criptografía de clave pública. Los revolucionarios conceptos planteados sentaron las bases para numerosos criptosistemas, en particular el denominado criptosistema RSA. Dicho criptosistema basa su seguridad en la dificultad inherente en la factorización de números "grandes", problema que se presume (pero no existe una prueba formal) entra en la categoría de los problemas NP-completos. De hecho, existe un algoritmo propuesto de tiempo polinomial para factorización en computadoras cuánticas [32], pero este tipo de máquinas pertenecen actualmente al mundo teórico.

A continuación se detallan los pasos a seguir para definir y poner en práctica un criptosistema RSA.

Dados dos números primos (suficientemente *grandes*) elegidos al azar, denominados p y q, se define el módulo n como el producto de estos dos números: $n = p \cdot q$. La función ϕ de Euler aplicada a n está dada por la fórmula $\phi(n) = (p-1)(q-1)$.

El siguiente paso consiste en seleccionar un entero impar e, $1 < e < \phi(n)$, tal que $\text{MCD}(e, \phi(n)) = 1$, y computar $d = e^{-1} \bmod \phi(n)$ utilizando el algoritmo de Euclides extendido [2] o alguna de sus variantes (en §5 se presentan algoritmos alternativos al tradicional de Euclides). El par (e, n) conforma entonces la clave pública, mientras que la clave privada está formada por el par (d, n).

Para evitar que se comprometa la seguridad del criptosistema, es recomendable que los valores originales p, q y $\phi(n)$ sean destruidos una vez creadas las claves pública y privada, ya que es posible que si son capturados por un adversario, los mismos sean utilizados para intentar desencriptar o modificar mensajes sin contar con la clave privada.

El proceso de encriptación de un texto plano M a un texto encriptado C se realiza mediante la siguiente exponenciación modular:

$$C = M^e \bmod(n)$$

La desencriptación para recuperar el texto plano original también consiste en una exponenciación modular:

$$M = C^d \bmod(n)$$

Se puede observar entonces que al realizar la encriptación y subsiguiente desencriptación, se está calculando

$$M = M^{ed} \bmod(n)$$

Se puede probar la corrección del proceso observando el hecho que $e \cdot d \equiv 1 \bmod(\phi(n))$. Esto se puede rescribir como $e \cdot d = 1 + k \cdot \phi(n)$, para algún entero k. Si es el caso que $\text{MCD}(M, p) = 1$, entonces el Teorema de Fermat [3] nos asegura que

$$M^{p-1} \equiv 1 \bmod(p)$$

Elevando ambos lados de esta congruencia al exponente $k \cdot (q-1)$ y luego multiplicando por M, se obtiene

$$M^{1+k(q-1)(p-1)} \equiv M \bmod(p)$$

En el caso complementario en que MCD(M, p) = p, la congruencia arriba detallada se mantiene, ya que ambos lados pasan a ser congruentes a 0 módulo p. Como así se cubren todos los casos, se tiene que

$$M^{ed} \equiv M \bmod(p)$$

Aplicando un razonamiento análogo, pero módulo el primo q, se llega a que

$$M^{ed} \equiv M \bmod(q).$$

Ya que está establecido desde un principio que p y q son primos distintos entre sí, y también es sabido por otro lado que $p \cdot q = n$, entonces se llega justamente al resultado que se quería demostrar:

$$M^{ed} \equiv M \bmod(n)$$

El concepto revolucionario de lograr una comunicación segura entre partes sin un intercambio de claves inicial inauguró una nueva era en la criptografía, con el advenimiento de los criptosistemas y los esquemas de firma de clave pública.

El hecho de que en el caso del criptosistema RSA la operación fundamental tanto para el proceso de encriptación como para el proceso de desencriptación sea la exponenciación modular lleva a investigar la optimización de este cálculo, computacionalmente caro, para intentar realizarlo de la manera más rápida posible y lograr una tasa de encriptación y desencriptación óptima.

3 Exponenciación modular

En el criptosistema RSA, una vez definidos los exponentes e y d y publicada la clave pública (e, n), tanto los emisores como los receptores utilizan la misma operación para encriptar, desencriptar, firmar y verificar: la exponenciación modular.

Es importante notar que esta operación no es utilizada exclusivamente por el criptosistema RSA, sino que es parte integral de varios otros criptosistemas y esquemas de firma digital, tales como el esquema de intercambio de claves de Diffie-Hellman [4], el esquema de firma de El-Gamal [5] y el Digital Signature Standard (DSS) propuesto por el National Institute for Standards and Technology (NIST) [6]. Sin embargo, el proceso de exponenciación que se lleva a cabo en un criptosistema basado en el problema de logaritmo discreto como los mencionados es levemente diferente al tratado en este trabajo: en ellos, la base M y el módulo n están dados previamente y es el exponente e el que varía en cada bloque a encriptar o desencriptar, lo que permitiría utilizar la precomputación de ciertas potencias de la base para acelerar el proceso subsiguiente [7]. En el caso de interés en el presente trabajo, en que buscamos optimizar el proceso de exponenciación dentro del algoritmo RSA, tanto el exponente e como el módulo n son conocidos de manera anticipada, pero no la base M (esto es, el texto plano a ser encriptado o el texto encriptado a ser desencriptado), por lo que este tipo de optimización no resulta aplicable.

La optimización de la exponenciación modular que investigaremos busca tomar ventaja de este conocimiento anticipado del exponente e, y existen variados métodos propuestos en la literatura. A continuación presentaremos los más utilizados por sus características de simplicidad o velocidad, buscando luego modificarlos para acelerar aún más el cálculo.

3.1 Exponenciaciones modulares directas

Si bien el concepto detrás de una exponenciación modular es extremadamente simple (una serie de multiplicaciones modulares hasta arribar al resultado deseado), existen variados métodos para realizar esta operación. La multiplicación modular, operación fundamental en la exponenciación modular, es en general computacionalmente costosa, especialmente en el caso de operandos de precisión múltiple como los considerados en el criptosistema RSA. Por lo tanto, el objetivo general es realizar la menor cantidad de multiplicaciones modulares posibles, de manera que la exponenciación modular se realice a su vez de manera rápida.

El método más intuitivo y a la vez ingenuo de computar $C = M^e \bmod(n)$ es claramente el comenzar con $C := M \bmod(n)$ y luego continuar aplicando repetidamente la multiplicación modular $C := C \cdot M \bmod(n)$ hasta obtener el valor deseado. Este método requiere $e-1$ multiplicaciones modulares, una cantidad de operaciones que puede resultar prohibitiva en el caso en que el valor de e sea lo suficientemente elevado (y en un criptosistema RSA con longitud de clave importante ese será el caso). Por ejemplo, en un sistema con longitud de clave $k = 512$ bits, el exponente e puede ser un número de hasta 155 dígitos decimales, lo que da una idea de lo impracticable de este método.

Sin embargo, no todas las potencias intermedias de M hasta llegar a M^e necesitan ser calculadas, y diversos métodos computacionalmente más económicos (y por lo tanto más veloces) han sido propuestos, comenzando por el denominado *método binario* y avanzando hacia técnicas más complejas, las cuales se describirán a continuación.

3.2 Método binario de exponenciación modular

El *método binario* de exponenciación modular (también conocido como el método de *elevar al cuadrado y multiplicar*) es conocido desde la antigüedad, y es aún muy utilizado debido a su simplicidad de comprensión e implementación. Este algoritmo recorre los bits del exponente de izquierda a derecha (o bien de derecha a izquierda, aunque en este caso se requiere una variable adicional) [2]. Si llamamos k a la cantidad de bits del exponente e, tenemos que $k = 1 + \lfloor \log_2 e \rfloor$ y podemos describir la expansión binaria de e como

$$e = (e_{k-1} e_{k-2} \ldots e_1 e_0) = \sum_{i=0}^{k-1} e_i 2^i \text{ , donde } e_i \in \{0,1\}.$$

El conocido algoritmo binario para calcular $C = M^e \bmod(n)$ se detalla a continuación:

Método binario

Entrada: M, e, n.

Salida: $C = M^e \bmod(n)$

1. **Si** $e_{k-1} = 1$ **entonces** $C \leftarrow M$ **si no** $C \leftarrow 1$

2. **Desde** $i = k-2$ **hasta** 0

2.1 $C \leftarrow C \cdot C \bmod(n)$

2.2 **Si** $e_i = 1$ **entonces** $C \leftarrow C \cdot M \bmod(n)$

3. **Devolver** C

Realizando un análisis este algoritmo, podemos notar que por cada bit del exponente se realiza un cuadrado (en definitiva, una multiplicación modular), y si el bit es igual a 1, entonces tenemos una multiplicación modular adicional en ese paso. Más formalmente, para un exponente arbitrario e de k bits donde $H(e)$ es el *peso de Hamming* de e (definido como la cantidad de bits iguales a 1 en la representación binaria de e), podemos observar que el método binario requiere de $k-1$ cuadrados (en el paso **2.1**) y $H(e)-1$ multiplicaciones (en el paso **2.2**). Por lo tanto, asumiendo que $e > 0$, $0 \le H(e)-1 \le k-1$ y $e_{k-1} = 1$, podemos analizar la cantidad de multiplicaciones en el mejor caso (donde hay sólo un bit igual a 1 en la expansión binaria de e, justamente en la posición $k-1$), el peor caso (donde todos los bits son igual a 1) y el caso promedio (donde la mitad de los bits son igual a 1):

Peor caso: $(k-1)+(k-1) = 2(k-1)$

Mejor caso: $(k-1)+0 = k-1$

Caso promedio: $(k-1)+\frac{1}{2}(k-1) = \frac{3}{2}(k-1)$

3.3 Método m-ario de exponenciación modular

El método binario se puede generalizar de manera tal que los bits del exponente e se recorran de a $\log_2 m$ bits a la vez, basándose en la denominada expansión m-aria del exponente [2]. Cuando m es una potencia de 2, la implementación en una computadora es considerablemente simple, ya que e se particiona simplemente agrupando los bits de la expansión binaria de e. Por ejemplo, agrupar los bits de a dos genera el llamado método cuaternario (ya que $\log_2 4 = 2$), agruparlos de a tres genera el método octal ($\log_2 8 = 3$), y así sucesivamente.

Formalmente, siendo $e = (e_{k-1}e_{k-2}...e_1e_0)$ la expansión binaria del exponente, entonces esta representación se particionará en s bloques de r bits cada uno, donde $sr = k$ (si es el caso que r no divide exactamente a k, entonces el exponente se completará a la izquierda con a lo sumo $r-1$ bits en 0, lo que claramente no modifica su valor). Definimos también el valor de la expansión binaria de cada bloque de r bits:

$$F_i = (e_{ir+r-1}e_{ir+r-2}..e_{ir}) = \sum_{j=0}^{r-1} e_{ir+j}2^j$$

Se puede observar que $0 \le F_i \le m-1$ y también que $e = \sum_{i=0}^{s-1} F_i 2^{ir}$. El método m-ario realiza una precomputación inicial de $M^w \bmod(n)$ para $w = 2,3,...,m-1$, que son todos los valores posibles para F_i. Se puede observar que la cantidad de multiplicaciones modulares a realizar será $m-2$, mediante la siguiente cadena:

$$M \cdot M = M^2$$
$$M^2 \cdot M = M^3$$
$$M^3 \cdot M = M^4$$
$$\vdots$$
$$M^{m-2} \cdot M = M^{m-1}$$

Una vez realizada la precomputación, se recorren los bits de e de a grupos de r bits, de izquierda a derecha, y a cada paso el resultado parcial se eleva a 2^r y se multiplica por M^{F_i} módulo n, donde F_i es el valor no-nulo de la sección de bits siendo considerada.

El algoritmo en detalle es el siguiente:

Método *m*-ario

Entrada: M, e, n.

Salida: $C = M^e \bmod(n)$

1. Calcular y almacenar $M^w \bmod(n)$ para $w = 2,3,...,m-1$.
2. Partir e en secciones de r bits F_i para $i = 0,1,...,s-1$.
3. $C \leftarrow M^{F_{s-1}} \bmod(n)$
4. **Desde** $i = s-2$ **hasta** 0

4.1 $C \leftarrow C^{2^r} \bmod(n)$

4.2 **Si** $F_i \neq 0$ **entonces** $C \leftarrow C \cdot M^{F_i} \bmod(n)$

5. **Devolver** C

A continuación analizamos el número promedio de multiplicaciones y cuadrados requeridos por este método, definiendo el valor $2^r = m$ y asumiendo sin pérdida de generalidad que $\frac{k}{r}$ es un entero:

- Multiplicaciones en precomputación (paso **1**): $m - 2 = 2^r - 2$
- Cuadrados (paso **4.1**): $(\frac{k}{r} - 1) \cdot r = k - r$
- Multiplicaciones (paso **4.2**): $(\frac{k}{r} - 1)(1 - \frac{1}{m}) = (\frac{k}{r} - 1)(1 - 2^{-r})$

Tomando todo esto en consideración, podemos afirmar que en promedio, la cantidad combinada de multiplicaciones y cuadrados que requiere el método *m*-ario es de:

$$2r - 2 + k - r + \left(\frac{k}{r} - 1\right)\left(1 - 2^{-r}\right)$$

En esta fórmula, si se substituye a r por 1 y a m por 2, se consigue precisamente el número promedio de multiplicaciones en el método binario, que como se detalló más arriba es de $\frac{3}{2}(k-1)$. También es interesante notar que existe un r óptimo correspondiente a cada longitud de exponente k, al que podemos llamar r^*, de modo tal que la cantidad promedio de multiplicaciones requeridas por el método *m*-ario es mínimo. Dicho r^* puede ser encontrado mediante simple enumeración [8].

En general, se puede expresar el valor asintótico de la cantidad de operaciones ahorradas mediante el uso del método *m*-ario, que es cercana al 33%. Se arriba a este valor computando el siguiente límite:

$$\lim_{k\to\infty} \frac{2r - 2 + k - r + (\frac{k}{r} - 1)(1 - 2^{-r})}{\frac{3}{2}(k-1)} = \frac{2}{3}\left(1 + \frac{1 - 2^{-r}}{r}\right) \approx \frac{2}{3}.$$

3.4 Métodos *m*-arios adaptativos

Es importante notar que es muy probable que durante el paso 1 del método *m*-ario, sobre todo a medida que m se incrementa, se precalculen valores de M^w que luego no serán utilizados porque esos w en particular no aparecen en la expansión binaria particionada del exponente e. Para evitar estos cálculos superfluos, se puede modificar el algoritmo para sólo calcular los M^w que luego serán necesarios en los pasos subsiguientes, generando lo que se denomina un *algoritmo adaptativo*.

Si lo que se decide es intentar reducir las multiplicaciones precalculadas, se puede modificar cualquier método *m*-ario para, una vez que se particiona la representación binaria del exponente en grupos de d bits cada uno, precomputar $M^w \bmod(n)$ sólo para los valores de w que aparezcan en la expansión binaria. La cantidad de multiplicaciones

que pueden ahorrarse de este modo en el mejor caso es $m-2=2^d-2$, cuando ocurre que cada partición de d bits es igual a 1 (esto es, cada partición presenta un 1 en el bit menos significativo, seguido por $d-1$ bits iguales a cero a la izquierda), lo que significa no tener que realizar ninguna precomputación y utilizar simplemente el valor de M. Claramente, esto ocurrirá en muy raras ocasiones, y en general se necesita calcular $M^w \bmod(n)$ para una importante cantidad de valores. Si esta cantidad de valores es sensiblemente menor a la cardinalidad del conjunto $\{2,3,...,2^d-1\}$, entonces se pueden lograr ahorros. El problema, sin embargo, se traslada en ese caso a lograr la precomputación de dichos valores mediante la menor cantidad posible de multiplicaciones. Este problema, análogo al de encontrar una *cadena de adiciones* de largo mínimo, resulta ser NP-completo [9], y excede el propósito del presente trabajo.

3.5 Algoritmos de ventana deslizante

Este tipo de algoritmo adaptativo utiliza un acercamiento diferente para lograr ahorros en la cantidad de multiplicaciones a ser realizadas en una exponenciación modular. En un método m-ario puro como los arriba mencionados, la probabilidad de que al particionar la representación binaria del exponente e en grupos de d bits, se logre un bloque conformado por todos bits en 0 es de 2^{-d}, asumiendo que los bits 0 y 1 son equiprobables. Como en el paso **4.2** del algoritmo m-ario se saltea una multiplicación cuando el valor del bloque es 0, resulta evidente que a medida de que la longitud de bloque d es más grande, la probabilidad de evitar esta multiplicación es menor. Pero por otro lado, la cantidad total de multiplicaciones a realizar se incrementa a medida que el valor de d disminuye. Los algoritmos de ventana deslizante buscan lograr un punto medio entre estos dos objetivos, buscando utilizar valores de d relativamente grandes, a la vez que el número promedio de bloques en 0 busca incrementarse también.

Como regla general, los algoritmos de ventana deslizante buscan descomponer la expansión binaria del exponente e en bloques (o ventanas, de ahí el nombre dado a este algoritmo) a los que denominaremos F_i, de tamaño $L(F_i)$ (de manera análoga a la notación utilizada en los métodos m-arios). Para mayor claridad, denominaremos *ventana cero* a aquellas cuyos bits son todos nulos ($F_i = 0$), y *ventana no-cero* a las que contengan por lo menos un bit distinto a 0 ($F_i \neq 0$). El número de ventanas puede cambiar dependiendo del exponente siendo tratado, y es probable que exponentes con la misma cantidad de bits generen distinto número de ventanas. Más aún, no se requiere que las ventanas sean todas del mismo tamaño.

Para mayor claridad, llamemos d a la longitud de la mayor ventana. Una característica favorable es que como las ventanas se generan particionando el exponente de derecha a izquierda (es decir, desde el bit menos significativo de e hacia el más significativo), y no hay razón para comenzar las ventanas no-cero con un bit distinto a 1, entonces siempre el bit menos significativo en cada ventana será igual a 1. Esto se traduce en que los valores de todas las ventanas no-cero serán siempre impares, lo que implica a su vez que la cantidad de multiplicaciones en la precomputación será aproximadamente la mitad que en un algoritmo m-ario común, ya que será necesario computar M^w sólo para los w impares.

En general, podemos describir el algoritmo de la siguiente manera:

Método de ventana deslizante

Entrada: M, e, n, d

Salida: $C = M^e \bmod(n)$

1. Computar y almacenar $M^w \bmod(n)$ para $w = 3,5,7,...,2^d - 1$
2. Particionar e en ventanas cero y no-cero F_i de largo $L(F_i) \leq d$ para $i = 0,1,2,...,p-1$
3. $C \leftarrow M^{F_{k-1}} \bmod(n)$
4. **Desde** $i = p-2$ **hasta** 0
   - 4.1 $C \leftarrow C^{2^{L(F_i)}} \bmod(n)$
   - 4.2 **Si** $F_i \neq 0$ **entonces** $C \leftarrow C \cdot M^{F_i} \bmod(n)$
5. **Devolver** C

Existen dos estrategias diferentes con respecto al método de ventana deslizante. La primera de ellas, denominada *ventanas no-cero de largo constante*, fue propuesta por Knuth [2] y genera ventanas cero de largo arbitrario, mientras que las ventanas no-cero son siempre de un largo fijo d. El método no permite ventanas cero adyacentes, ya que las mismas son concatenadas en una ventana cero más grande. Sin embargo, es posible que dos ventanas no-cero coexistan de manera adyacente. El número promedio de multiplicaciones que genera este método en particular fue estudiado en [10], modelando el proceso de partición como una cadena de Markov. De la misma manera que en los métodos m-arios, existen valores óptimos de d para cada k. En general, esta estrategia reduce el número de multiplicaciones entre 3% y 7% para $128 \leq k \leq 2048$. Es interesante notar que los tiempos generados por el proceso de partición son en general despreciables, con un número de operaciones a nivel bit proporcional a k.

La segunda alternativa, propuesta por Bos y Coster [11], permite ventanas de tamaño variable, tanto si las mismas son cero como si son no-cero. Existen aquí dos parámetros a ser considerados: el tamaño máximo de una ventana no-cero d y la cantidad mínima de ceros q para cambiar a una ventana cero. Esta estrategia puede llegar a producir ventanas no-cero adyacentes, pero necesariamente la ventana en la posición menos significativa tendrá d bits. Al analizar esta estrategia en [10], se constató que para valores óptimos de d y q este método produce ahorros de entre 5% y 8% en la cantidad de multiplicaciones a realizar durante una exponenciación modular, para $128 \leq k \leq 2048$. Debido a esta mejora, este método es considerado óptimo entre los conocidos en la literatura, y será el método de ventana deslizante implementado en el presente trabajo.

La estrategia para particionar los bits del exponente que utilizaremos es una variante del método de ventana deslizante de tamaño variable, que genera ventanas de tamaño máximo d y tamaño mínimo 1, cuyos bits más significativo y menos significativo son siempre no nulos (pueden coincidir en el caso de tamaño unitario). El algoritmo resulta de muy simple implementación, y en cada paso se encuentra formando una ventana cero (llamamos a este estado VC) o ventana no-cero (estado VNC), comenzando desde el bit menos significativo del exponente y procediendo de la siguiente manera:

VC: Chequear el valor del próximo bit. Si es 0, mantenerse en VC. En caso contrario, pasar a VNC.

VNC: Colectar d bits. Si el último bit colectado es 0, volver atrás un bit, reduciendo el tamaño de la ventana. Repetir este chequeo hasta arribar a un bit distinto a 0, quedando conformada la ventana no-cero. Si el próximo bit es 0, pasar a VC. En caso contrario, mantenerse en VNC.

Como ejemplo, mostramos las particiones generadas con diferentes valores de d del siguiente exponente e, expresado en base binaria:

$$e = 11000101100010001010111010001001010100010110110111 01$$

La partición generada cuando $d = 5$ es la siguiente:

$$e = \underline{11000}\underline{10110}\underline{00010}\underline{00101}\underline{01110}\underline{10001}\underline{00101}\underline{01010}\underline{00101}\underline{10110}\underline{11101}$$

Si $d = 7$, la partición generada es:

$$e = \underline{1100010}\underline{1100010}\underline{0010}\underline{1011}\underline{1010001}\underline{0010}\underline{1010}\underline{0010}\underline{1101}\underline{1011101}$$

En la práctica, al igual que en los métodos m-arios, la partición no constituye un proceso separado, sino que se intercala con el proceso de exponenciación.

Queda claro que la selección de una longitud de ventana d óptima que minimice la cantidad de multiplicaciones y a la vez maximice la cantidad y tamaño de ventanas cero depende de varios factores, tales como la longitud k del exponente e y la distribución en particular de los bits no-nulos en el exponente e siendo recodificado. En §6 se listan los valores óptimos de longitud de ventana deslizante para distintas longitudes del exponente e, obtenidos mediante experimentación y enumeración.

4 Métodos de recodificación

Se ha establecido que, para cualquiera de los métodos anteriormente mencionados, $k-1$ es el límite inferior para la cantidad de cuadrados requeridos para computar M^e, donde k es el número de dígitos o bits en la representación binaria de e. Sin embargo, es posible reducir la cantidad de multiplicaciones subsiguientes, las cuales dependen de la cantidad de bits distintos de 0 en la representación binaria de e, mediante una *recodificación* del exponente [8, 12, 13, 14] utilizando un dígito especial al que denominaremos *dígito con signo*. Estas técnicas de recodificación utilizan la siguiente identidad:

$$2^{i+j-1}+2^{i+j-2}+\ldots+2^{i}=2^{i+j}-2^{i}$$

Lo que se busca al recodificar el exponente es lograr una representación alternativa que minimice la cantidad de bits distintos a 0, por lo tanto minimizando la cantidad de multiplicaciones al aplicar el método binario, m-ario o de ventana deslizante para el cálculo de una exponenciación modular. Los dígitos a utilizar para representar el exponente recodificado serán, por lo tanto, {1, 0, -1} (el dígito –1 será escrito como $\bar{1}$ para evitar confusiones). Por ejemplo, el número decimal 30 se representa en base binaria como

$$(11110)=2^4+2^3+2^2+2^1$$

Utilizando la identidad arriba detallada, el número 30 se puede recodificar en esta nueva representación como

$$(100\bar{1}0)=2^5-2^1$$

Aquí se puede apreciar la ventaja de la recodificación. A pesar de exceder la longitud original (lo que significa una operación de cuadrado adicional), se logra reducir la cantidad de dígitos no nulos de 4 a 2, lo que llevaría la cantidad de multiplicaciones modulares a realizar en el clásico método binario de 7 a 6.

Podemos formalizar esto con respecto al método binario observando que computar M^e utilizando este método conlleva $k-1$ cuadrados (donde k es el número de bits en la representación binaria de e) y $(w(e)-1)$ multiplicaciones, donde $w(e)$ es el *peso de Hamming* de e, o lo que es lo mismo, la cantidad de bits no-nulos en e.

La desventaja es que será necesario contar de antemano con M^{-1} (o en su defecto calcularlo). Para enfrentar este problema en particular, investigaremos más adelante alternativas al clásico algoritmo de Euclides extendido, tales como la inversa modular de Montgomery, el algoritmo de MCD extendido de Lehmer y el algoritmo de MCD binario extendido (§5). Por el momento nos concentraremos en la variedad de métodos propuestos para lograr una representación del exponente utilizando *bits con signo*, ya que podemos observar que existen infinitas representaciones posibles.

Formalizando la notación, denominaremos como una *representación con bits con signo de longitud l(e)* para un entero positivo e a la secuencia $s_{l(e)-1}, s_{l(e)-2}, \ldots, s_0$ tal que $e=\sum_{i=0}^{l(e)-1} s_i 2^i$, donde $s_i \in \{-1,0,1\}$ y $s_{l(e)-1}=1$. La secuencia de bits con signo s_i se escribirá normalmente sin comas, y utilizaremos $m=l(e)-1$. Como podemos reemplazar

siempre a 2^m por $2^{m+1}-2^m$, y esto puede repetirse indefinidamente, queda claro existen infinitas representaciones para e, con longitud arbitraria lo suficientemente grande. Teniendo en mente el objetivo de optimización, en donde una menor cantidad de bits implica una menor cantidad de multiplicaciones modulares, es deseable encontrar una representación corta, y además con el menor peso de Hamming posible (o lo que es equivalente, la menor cantidad de dígitos no-nulos posibles). Llamaremos a esta representación *óptima* si tiene el menor peso posible, y si entre todas las representaciones con dicho peso, tiene la menor longitud posible. Es claro ver que la longitud de la representación binaria original de e presenta un límite inferior para la longitud de la representación con bits con signo. Otra observación interesante es que una representación óptima puede no ser única, como podemos ver en el siguiente ejemplo en donde presentaremos dos representaciones óptimas del número $11=(1011)_2$:

$$(1011)=2^3+2+1$$

$$(110\bar{1})=2^3+2^2-1$$

A continuación presentamos una variedad de algoritmos propuestos para recodificar un exponente e en una representación con bits con signo, optando en definitiva por uno de ellos (el algoritmo de recodificación canónico) por sus características óptimas.

4.1 El algoritmo de Booth y sus modificaciones

El algoritmo de recodificación de Booth [15], quien introdujo originalmente la noción de recodificación mediante bits con signo como una mejora para los algoritmos de multiplicación, recorre los bits del número binario $e=(e_{k-1}e_{k-2}\ldots e_1e_0)$ de derecha a izquierda, y obtiene los dígitos del número recodificado f utilizando la siguiente tabla de verdad (se considera $e_{-1}=0$ para iniciar el algoritmo):

e_i	e_{i-1}	f_i
0	0	0
0	1	1
1	0	$\bar{1}$
1	1	0

Por ejemplo, la recodificación de $e=(111001111)$ mediante el algoritmo de Booth resulta en $f=(100\bar{1}01000\bar{1})$, que resulta en un menor peso de Hamming, con sólo 4 dígitos no-cero, comparado con 7 del exponente original.

Sin embargo, el algoritmo de Booth puede llegar a generar exponentes recodificados con mayor peso de Hamming que el exponente original, ya que al encontrar secuencias del par de bits (01), son recodificadas como secuencias de $(1\bar{1})$. Un ejemplo patológico como $e=(101010101)$, que es recodificado como $f=(1\bar{1}1\bar{1}1\bar{1}1\bar{1}1)$, sirve como muestra de que la recodificación mediante este algoritmo puede resultar contraproducente en ciertos casos. Un inconveniente adicional, en cierta manera relacionado con el anterior, es que si dos secuencias de bits en 1 están separadas por un único bit en 0, este algoritmo no realiza la recodificación óptima posible. Por ejemplo, $e=(11101111)$ se

recodifica como $f = (100\bar{1}1000\bar{1})$, mientras que la recodificación alternativa $f' = (1000\bar{1}000\bar{1})$ resulta de menor peso, ya que $(\bar{1}1) = -2+1 = -1 = (0\bar{1})$.

Varias modificaciones alternativas basadas en el algoritmo de Booth fueron propuestas para contrarrestar estas deficiencias [8, 13], donde los bits se recorren de a un mayor número por vez, intentando evitar introducir dígitos no-nulos innecesarios al número recodificado. En particular, la alternativa propuesta por Reitweisner en [20], denominada *recodificación canónica*, reúne ciertas propiedades óptimas, y se describe en detalle en la siguiente sección.

4.2 Recodificación canónica

Un exponente recodificado f cumple con la denominada propiedad de *no-adyacencia* cuando no contiene dos bits no-nulos contiguos. Más formalmente, si la representación recodificada es $f = (f_k f_{k-1} \cdots f_1 f_0)$, se cumple la propiedad de *no-adyacencia* si $f_i \cdot f_{i-1} = 0$, para $0 < i \leq k$.

Si el exponente recodificado f además de cumplir con esta propiedad tiene longitud mínima entre todas las posibles, entonces se lo denomina como un *exponente recodificado canónico*. Si consideramos a la representación binaria del exponente original e como completada a izquierda por un bit en 0, entonces se prueba que existe un único exponente recodificado f canónico para e, y este exponente recodificado es óptimo en el sentido que presenta la mínima cantidad de dígitos no-cero entre todos los infinitos exponentes recodificados que representan a e [20] (a diferencia del algoritmo original de Booth y algunas de las modificaciones propuestas).

El *algoritmo de recodificación canónico* [20, 21, 22] computa el exponente recodificado $f = (f_k f_{k-1} \cdots f_1 f_0)$ comenzando por el bit menos significativo, utilizando un vector adicional c como variable adicional. Luego de inicializar $c_0 = 0$, se examinan los bits de e de a dos por vez, generando los dígitos recodificados f_i y la variable auxiliar c_{i+1} mediante la siguiente tabla de verdad:

c_i	e_{i+1}	e_i	c_{i+1}	f_i
0	0	0	0	0
0	0	1	0	1
0	1	0	0	0
0	1	1	1	$\bar{1}$
1	0	0	0	1
1	0	1	1	0
1	1	0	1	$\bar{1}$
1	1	1	1	0

Se prueba [15] que, en promedio, uno de cada tres bits en una recodificación canónica será no-nulo, comparado con uno de cada dos en una representación binaria clásica. Debido a lo deseables que resultan los bits nulos en los algoritmos de exponenciación detallados, esta característica resulta muy ventajosa. Por caso, la probabilidad de una

cadena de n bits no-nulos en una representación binaria clásica será de $\left(\frac{1}{2}\right)^n$, mientras que esta misma probabilidad en una recodificación canónica es de $\left(\frac{2}{3}\right)^n$.

Utilizando lenguajes formales para modelar la remodificación canónica mediante cadenas de Markov, se ha probado en [14] que si se cuenta con el valor de $M^{-1} \bmod(n)$ junto con el valor de M entonces el promedio de ahorros en el número de cuadrados y multiplicaciones necesarias para calcular $C = M^e \bmod(n)$ mediante el método binario es de aproximadamente un 11%, propiedad que se comprueba más adelante en los resultados experimentales (§6).

En general, cualquier algoritmo de exponenciación modular entre los mencionados arriba puede adaptarse de manera muy simple para manejar exponentes recodificados con signo, siempre bajo la premisa de que se cuente tanto con la base M como con su inversa $M^{-1} \bmod(n)$. Si consideramos los métodos *m*-arios o de ventana deslizante, está claro que al tomar particiones (o ventanas) del exponente recodificado, los valores w de las mismas pueden ser negativos, dependiendo de la cantidad y la posición de los bits iguales a $\bar{1}$ en la partición siendo considerada. Por lo tanto, durante la etapa de precomputación, puede ser necesario calcular $M^w \bmod(n)$ para ciertos $w < 0$, lo cual no presenta inconvenientes adicionales si se cuenta de manera anticipada con el valor $M^{-1} \bmod(n)$, ya que es posible utilizar el hecho de que $M^{-w} \bmod(n) = (M^{-1})^w \bmod(n)$.

Antes de adentrarnos en la adaptación de los métodos tradicionales para hacer uso de los exponentes recodificados, analizaremos las mejores alternativas existentes para el cálculo de la inversa modular $M^{-1} \bmod(n)$.

5 Inversa modular

Una de las principales razones por las cuales los métodos de exponenciación modular que utilizan exponentes recodificados no han resultado de utilidad es que en general el tiempo necesario para realizar el cálculo de $M^{-1} \bmod(n)$ mediante los algoritmos de inversa modular clásicos excedía el tiempo ahorrado al utilizar exponentes recodificados.

El método más popular para calcular una inversa modular es utilizar el clásico algoritmo de Euclides extendido. Este algoritmo, dados dos números enteros x e y, calcula los valores a, b y v, donde $ax+by=v$ y $MCD(x,y)=v$. Para calcular la inversa modular $M^{-1} \bmod(n)$, basta con aplicar el algoritmo tomando $x=M$ y $y=n$. Si es el caso que $MCD(M,n)=1$, entonces el valor de a que retorne el algoritmo será la inversa modular deseada. En caso de que $MCD(M,n)\neq 1$, entonces M no es inversible módulo n.

Si bien el algoritmo de Euclides extendido puede utilizarse de manera muy simple para calcular esta inversa, tiene la desventaja de requerir costosas divisiones de precisión múltiple en el caso de que los enteros x e y de entrada sean de precisión múltiple, justamente el caso de interés en el presente trabajo. Para contrarrestar este inconveniente, existen métodos alternativos que resultan más apropiados para ser utilizados en un ambiente de precisión múltiple. A continuación seleccionamos tres de ellos, que serán investigados como las mejores alternativas para el cálculo de $M^{-1} \bmod(n)$ en los métodos de exponenciación con exponente recodificado.

5.1 La inversa modular de Montgomery

P. L. Montgomery propuso en [23] un algoritmo eficiente para realizar el cálculo de una multiplicación modular $R=a\cdot b \bmod n$, en donde a, b y n son números binarios de k bits. La estructura del algoritmo está dirigida directamente a su aplicación eficiente en computadores, ya que reemplaza toda división por n por divisiones por una potencia de 2, intrínsecamente rápidas en microprocesadores ya que se traducen en la lógica binaria como simples operaciones de desplazamiento.

Suponiendo que el módulo n es un número de k bits (esto es, $2^{k-1} \leq n < 2^k$), definamos el número $r=2^k$. El algoritmo de Montgomery requiere que n y r sean coprimos, lo que siempre se cumple si n es impar ya que el único factor en r es el 2. Si n es par, sin embargo, existe una técnica simple y directa para poder utilizar este método, factorizando $n=q\cdot 2^j$ mediante un desplazamiento a derecha hasta que el bit menos significativo sea igual a 1, y luego utilizando propiedades del Teorema Chino del Resto para lograr el resultado final [24].

La idea detrás del algoritmo de Montgomery se puede resumir de manera relativamente simple. Dado un entero $a<n$, su n-residuo con respecto a r se define como

$$\bar{a}=a\cdot r \bmod(n)$$

Se puede observar que el conjunto $\{i\cdot r \bmod n \mid 0\leq i\leq n-1\}$ es un sistema completo de residuos y por lo tanto contiene a todos los números entre 0 y $n-1$. De esto se deduce

que existe una correspondencia 1 a 1 entre el rango $\{0,1,...,n-1\}$ y el conjunto arriba mencionado. El algoritmo utiliza ventajosamente esta propiedad, mediante una rutina de multiplicación sensiblemente más rápida, que computa el n-residuo del producto de dos enteros cuyos n-residuos son dados como dato. Se define entonces el *producto de Montgomery* como el siguiente n-residuo:

$$\overline{R} = \overline{a} \cdot \overline{b} \cdot r^{-1} \bmod(n).$$

Aquí r^{-1} es simplemente la inversa de r módulo n.

Podemos confirmar que $\overline{R}$ es efectivamente el n-residuo de $R = a \cdot b \bmod(n)$, ya que se cumple que

$$\overline{R} = \overline{a} \cdot \overline{b} \cdot r^{-1} \bmod(n) = a \cdot r \cdot b \cdot r \cdot r^{-1} \bmod(n) = a \cdot b \cdot r \bmod(n)$$

Para describir el funcionamiento del algoritmo de Montgomery es necesario contar con una cifra adicional, a la que denominaremos n', que cumple con la siguiente propiedad:

$$r \cdot r^{-1} - n \cdot n' = 1.$$

Por su definición, tanto r^{-1} como n' pueden computarse mediante el algoritmo de Euclides extendido o alguna de sus variantes, pero existen implementaciones efectivas en donde se realizan importantes ahorros evitando el cálculo explícito de estos valores [25].

A continuación se presenta el algoritmo de Montgomery:

Método de Montgomery

Entrada: $\overline{a}, \overline{b}$

Salida: $a \cdot b \cdot r \bmod(n)$

1. $t \leftarrow \overline{a} \cdot \overline{b}$
2. $m \leftarrow t \cdot n' \bmod r$
3. $u \leftarrow (t + m \cdot n)/r$
4. **Si** $u \geq n$ **entonces devolver** $u - n$**. Si no, devolver** u

La ventaja fundamental de este algoritmo es que todas las operaciones involucradas corresponden a multiplicaciones módulo r y divisiones por r, y ambas resultan ideales para ser implementadas en microprocesadores por el hecho de que r es una potencia de 2, lo que resultará en su posible implementación como *shifts* o desplazamientos binarios a izquierda o derecha.

Resta mostrar cómo este algoritmo de Montgomery puede utilizarse para computar el producto de dos enteros cualesquiera a y b módulo n (siempre bajo la condición de que n sea impar). A continuación, describimos el algoritmo que realiza justamente la operación de interés:

Multiplicación modular de Montgomery

Entrada: a, b, n

Salida: $a \cdot b \bmod(n)$

1. $\overline{a} \leftarrow a \cdot r \bmod n$
2. $\overline{b} \leftarrow b \cdot r \bmod n$

3. $x \leftarrow$ Montgomery $(\bar{a}, \bar{b})$
4. $x \leftarrow$ Montgomery $(x,1)$
5. **Devolver** x

Es interesante notar cómo se obtiene el resultado final x a partir de $\bar{x}$ en el paso **5**, mediante la utilización de la siguiente propiedad del algoritmo de Montgomery:

$$\text{Montgomery}(\bar{x},1) = \bar{x} \cdot 1 \cdot r^{-1} = x \cdot r \cdot r^{-1} = x \bmod n$$

Existe un algoritmo aún más optimizado, que utiliza la propiedad abajo detallada:

$$\text{Montgomery}(\bar{a}, b) = (a \cdot r) \cdot b \cdot r^{-1} = a \cdot b \bmod n,$$

Esto genera un algoritmo de multiplicación con una menor cantidad de pasos, como detallamos a continuación:

Multiplicación modular de Montgomery optimizada

Entrada: a, b, n
Salida: $a \cdot b \bmod(n)$

1. $\bar{a} \leftarrow a \cdot r \bmod(n)$
2. $x \leftarrow$ Montgomery $(\bar{a}, b)$
3. **Devolver** x

El algoritmo de Montgomery tiene el atractivo adicional de permitir calcular la inversa de un entero a módulo un entero n de una manera alternativa al tradicional algoritmo de Euclides Extendido. Para esto, Kaliski [26] definió la denominada inversa de Montgomery como

$$\text{InvMontgomery}(a) = a^{-1} r \bmod(n)$$

Kaliski también definió en [26] un algoritmo para computar la inversa de Montgomery. Este algoritmo puede dividirse en dos fases. En la *fase I*, se calcula el entero u tal que $u = a^{-1} 2^v \bmod(n)$, donde $k \le v \le 2k$. Este resultado se corrige durante la *fase II*, para obtener la inversa de Montgomery $a^{-1} r \bmod(n)$.

Esta inversa de Montgomery puede ser utilizada para calcular la inversa modular clásica. Una de las maneras de lograrlo es computar la inversa de Montgomery del entero inicialmente, y luego utilizar la multiplicación de Montgomery para obtener el resultado final, de la siguiente manera:

$$b \leftarrow \text{InvMontgomery}(a) = a^{-1} r \bmod(n)$$

$$x \leftarrow \text{Montgomery}(b,1) = (a^{-1} r) \cdot 1 \cdot r^{-1} \bmod(n) = a^{-1} \bmod(n)$$

Alternativamente, se puede revertir el orden de estas operaciones, llegando al mismo resultado por otro camino. El cálculo de la inversa modular pasa a ser de la siguiente manera:

$$b \leftarrow \text{Montgomery}(a, r^2) = a \cdot (r^2) r^{-1} \bmod(n) = a \cdot r \bmod(n)$$

$$x \leftarrow \text{InvMontgomery}(b) = (a \cdot r)^{-1} r \bmod(n) = a^{-1} \bmod(n)$$

5.2 Inversa modular de Montgomery modificada (Savas-Koç)

En [27], Savaş y Koç propusieron una modificación a la inversa de Montgomery originalmente propuesta por Kaliski, en donde la raíz en la que se basa el sistema se amplía de 2^k (donde k es la cantidad de bits de n) a 2^m (donde m es un entero múltiplo del tamaño w de palabra del procesador utilizado por la computadora en donde se implemente el algoritmo), lo que lleva a una mejora en su eficiencia.

El cálculo de la inversa modular tradicional se realiza entonces mediante el siguiente algoritmo:

Inversa modular de Savas-Koç

Entrada: a, n, k y m, donde $a \in [1, 2^{m-1}]$

Salida: $x = a^{-1} \bmod(n)$

1. **Calcular** u **y** v **mediante la fase I de Kaliski, donde** $u = a^{-1}2^v \bmod(n)$ **y** $k \le v \le m + k$
2. **Si** $v > m$ **entonces**
   - **2.1** $u \leftarrow \text{Montgomery}(u,1) = (a^{-1}2^v)(2^{-m}) = a^{-1}2^{v-m} \bmod(n)$
   - **2.2** $v \leftarrow v - m < m$
3. **Devolver** $x \leftarrow \text{Montgomery}(u, 2^{m-v}) = (a^{-1})(2^v)(2^{m-v})(2^{-m}) = a^{-1} \bmod(n)$

Se puede notar analizando este algoritmo que la inversa modular clásica se calcula directamente luego de aplicar la *fase I* de Kaliski, mediante 1 o 2 productos de Montgomery, evitando calcular la inversa de Montgomery completa. Debido a esta característica deseable, seleccionaremos esta modificación de la inversa modular de Montgomery para ser implementada en nuestra experimentación.

5.3 Inversa modular mediante el algoritmo de MCD extendido de Lehmer

D. H. Lehmer propuso en [27] un algoritmo dirigido a optimizar el cálculo del máximo común divisor (MCD) de dos enteros positivos de múltiple precisión utilizando mayormente operaciones de precisión simple, lo que permite utilizar en la mayor parte de los casos operaciones internas del procesador. Este algoritmo puede extenderse de manera directa a un algoritmo de MCD extendido, aplicable para encontrar la inversa modular de un entero, y resulta considerablemente eficiente.

El algoritmo original de Euclides para calcular el MCD d de dos enteros a y b puede resumirse en pocas palabras. Si $b = 0$, entonces el algoritmo se detiene, devolviendo $d = a$. En caso contrario, se computa $a = b \cdot w + r$, donde $0 \le r < b$, y se itera con b tomando el lugar de a y con r tomando el lugar de b.

La idea de Lehmer en [27] surge de la observación que cuando a y b tienen un tamaño similar, entonces el cociente entero w en cada paso es usualmente de sólo un dígito. Además, observó que el proceso subyacente en el algoritmo de Euclides es la aplicación de sucesivas transformaciones lineales:

$$\begin{pmatrix} u \\ v \end{pmatrix} \rightarrow \begin{pmatrix} A & B \\ C & D \end{pmatrix} \begin{pmatrix} u \\ v \end{pmatrix} = \begin{pmatrix} Au + Bv \\ Cu + Dv \end{pmatrix}$$

El enfoque propuesto por Lehmer es el de encontrar repetidamente este w mientras se mantenga "chico" (es decir, de un solo dígito en la base elegida), manteniendo un registro de las operaciones involucradas mediante una matriz. Cuando no es posible proceder, entonces se aplica dicha matriz a los datos originales y se intenta de nuevo. Ocasionalmente, se puede llegar a un punto en que una división (o reducción modular, en este caso) resulta inevitable.

El algoritmo de Lehmer se presenta en detalle a continuación:

Algoritmo de MCD de Lehmer

Entrada: $x, y, x \leq y$, *expresados en base* b

Salida: $MCD(x, y)$

1. **Mientras** $y \geq b$ **hacer**

1.1 **Asignar a** $\bar{x}$ **e** $\bar{y}$ **los dígitos más significativos de** x **e** y

1.2 $A \leftarrow 1; B \leftarrow 0; C \leftarrow 0; D \leftarrow 1$

1.3 **Mientras** $(\bar{y} + C) \neq 0$ **y** $(\bar{y} + D) \neq 0$**, hacer**

1.3.1 $q \leftarrow \lfloor (\bar{x} + A)/(\bar{y} + C) \rfloor;\ q' \leftarrow \lfloor (\bar{x} + B)/(\bar{y} + D) \rfloor$

1.3.2 **Si** $q \neq q'$ **entonces ir al paso 1.4**

1.3.3 $t \leftarrow A - qC; A \leftarrow C; C \leftarrow t; t \leftarrow B - qD; B \leftarrow D; D \leftarrow t$

1.3.4 $t \leftarrow \bar{x} - q\bar{y}; \bar{x} \leftarrow \bar{y}; \bar{y} \leftarrow t$

1.4 **Si** $B = 0$ **entonces**

1.4.1 $T \leftarrow x \bmod y; x \leftarrow y; y \leftarrow T$

Si no,

1.4.2 $T \leftarrow Ax + By; u \leftarrow Cx + Dy; x \leftarrow T; y \leftarrow u$

2. **Computar** $v = MCD(x, y)$ **mediante el algoritmo de Euclides**

3. **Devolver** v

Se puede observar que en el paso 1.3 y todos sus pasos interiores se intentan simular divisiones de múltiple precisión mediante operaciones de precisión simple mucho menos costosas. En cada paso dentro de la iteración de 1.3 las operaciones son de precisión simple, donde la cantidad de iteraciones a realizar depende de la base b.

Sin embargo, existe una operación de precisión múltiple, y es la reducción modular en 1.4.1. Las operaciones en 1.4.2 son también de precisión múltiple, pero el hecho de que los multiplicadores son de precisión simple hace que puedan realizarse en tiempo lineal.

Las variables A, B, C y D son las que a lo largo del algoritmo almacenan los coeficientes que generarán los valores a y b, donde $ax + by = v$ y $MCD(x, y) = v$, análogamente al algoritmo de Euclides extendido clásico.

5.4 Inversa modular mediante el algoritmo de MCD binario extendido

El algoritmo de MCD binario toma una ruta alternativa a la euclidiana. Si bien requiere una cantidad mayor de pasos que el algoritmo de Euclides, este algoritmo elimina las divisiones (computacionalmente muy caras, sobre todo en el caso de números de precisión múltiple) y las reemplaza por sumas y *shifts* o desplazamientos, de veloz implementación en microprocesadores. El algoritmo de MCD binario extendido es la extensión natural del algoritmo de MCD binario, y se describe a continuación:

Algoritmo de MCD binario extendido

Entrada: x, y
Salida: a, b y v, donde $ax + by = v$ y $MCD(x, y) = v$

1. $g \leftarrow 1$
2. **Mientras** x e y sean ambos pares, **hacer**
  2.1 $x \leftarrow x/2$; $y \leftarrow y/2$; $g \leftarrow 2g$
3. $u \leftarrow x$; $v \leftarrow y$; $A \leftarrow 1$; $B \leftarrow 0$; $C \leftarrow 0$; $D \leftarrow 1$
4. **Mientras** u sea par, **hacer**
  4.1 $u \leftarrow u/2$
  4.2 **Si** A y B son ambos pares, **entonces**
    4.2.1 $A \leftarrow A/2$; $B \leftarrow B/2$
    Si no,
    4.2.2 $A \leftarrow (A + y)/2$; $B \leftarrow (B - x)/2$
5. **Mientras** v sea par, **hacer**
  5.1 $v \leftarrow v/2$
  5.2 **Si** C y D son ambos pares, **entonces**
    5.2.1 $C \leftarrow C/2$; $D \leftarrow D/2$
    Si no,
    5.2.2 $C \leftarrow (C + y)/2$; $D \leftarrow (D - x)/2$
6. **Si** $u \geq v$, **entonces**
  6.1 $u \leftarrow u - v$; $A \leftarrow A - C$; $B \leftarrow B - D$
  Si no,
  6.2 $v \leftarrow v - u$; $C \leftarrow C - A$; $D \leftarrow D - B$
7. **Si** $u = 0$, **entonces**
  7.1 $a \leftarrow C$; $b \leftarrow D$; $v \leftarrow g \cdot v$
  7.2 **Devolver** a, b, **y** g
  Si no,
  7.3 **Ir al paso 4.**

Existen en la literatura opiniones encontradas con respecto a las ventajas de cada uno de estos algoritmos sobre los demás. Por ejemplo, Jebelean [27] sugiere que el algoritmo de Lehmer es superior, mientras que Sorenson [28] sostiene una mayor eficiencia del algoritmo de MCD binario. Es por estas discrepancias que decidimos investigar el comportamiento de cada uno de los tres en los algoritmos adaptados para utilizar exponentes recodificados.

6 Exponenciación modular utilizando exponentes recodificados

Una vez introducidos los distintos métodos clásicos de cálculo de la exponenciación modular, la noción de exponentes recodificados y las distintas maneras de calcular la inversa modular, pasamos a presentar los resultados originales conseguidos al implementar las modificaciones a los algoritmos clásicos para que hagan uso de los exponentes recodificados.

Los algoritmos fueron implementados utilizando C++ (en particular, *Microsoft© Visual C++ 6.0™*) en un equipo con un procesador *Intel© Celeron* trabajando a una frecuencia de 766 MHz. Debido al manejo de enteros de varios cientos de dígitos, se decidió utilizar la librería GMP 4.1, publicada bajo la licencia GNU de código *open source*, que provee un excelente manejo optimizado de aritmética de precisión múltiple [30].

Con la idea de mantener las comparaciones lo más generales posibles, evitando depender del equipo en que se implementaron los distintos métodos, los cuadros comparativos presentarán los resultados no en términos de tiempos de ejecución, sino en términos de cantidad de multiplicaciones modulares de precisión múltiple. Sin perder generalidad, podemos dividir conceptualmente los algoritmos en etapas disjuntas (es importante notar que en algunos algoritmos ciertas etapas pueden no existir): *recodificación* del exponente, *partición* del exponente, *inversión* modular de la base, *precomputación* de valores y *exponenciación* propiamente dicha. Como ya se indicó, las etapas de recodificación y partición conllevan un tiempo y esfuerzo computacional que puede despreciarse en comparación con una multiplicación modular, por lo que los cuadros comparativos tomarán en cuenta sólo las etapas de inversión, precomputación y exponenciación.

Las etapas de precomputación y exponenciación están conformadas de manera prácticamente exclusiva por multiplicaciones modulares, por lo que presentar la cantidad de multiplicaciones modulares promedio que implican es una tarea directa. Sin embargo, las alternativas de inversa modular exploradas en el presente trabajo involucran esencialmente operaciones de desplazamiento a nivel bit en lugar de multiplicaciones o divisiones modulares, en aras de conseguir mayor velocidad.

Por lo tanto, para poder adaptarlas al modelo de comparación de multiplicaciones modulares elegido, optamos por medir el tiempo promedio requerido para el cálculo de la inversa para las diferentes longitudes k de operandos, y equiparar esos tiempos con su equivalente en multiplicaciones modulares. Para lograr esto, primero medimos los tiempos requeridos para realizar una multiplicación modular, presentados en la siguiente tabla:

bits	Tiempo multiplicación modular
512	0.07 ms
1024	0.22 ms
1536	0.46 ms
2048	0.78 ms
2560	1.10 ms
3072	1.60 ms
3584	2.13 ms
4096	2.60 ms

A continuación, tomando como base estos tiempos, presentamos los tiempos promedios para realizar la inversa modular mediante cada uno de los tres métodos estudiados, junto con su equivalente en multiplicaciones modulares para las diferentes longitudes de k.

	Inversa de Montgomery		MCD Extendido de Lehmer		MCD binario extendido	
bits	**Tiempo**	**Multiplicaciones modulares equivalentes**	**Tiempo**	**Multiplicaciones modulares equivalentes**	**Tiempo**	**Multiplicaciones modulares equivalentes**
512	0.61	8.71	0.31	4.43	0.24	3.43
1024	1.88	8.55	0.84	3.82	0.67	3.05
1536	3.90	8.48	1.86	4.04	1.44	3.13
2048	7.03	9.01	2.91	3.73	2.44	3.13
2560	10.11	9.19	4.44	4.04	3.97	3.61
3072	14.58	9.11	5.96	3.73	4.90	3.06
3584	19.28	9.05	8.31	3.90	7.54	3.54
4096	23.87	9.18	10.52	4.05	9.17	3.53

Podemos notar que, en nuestra implementación en particular, los mejores resultados a la hora de calcular la inversa modular corresponden al algoritmo de MCD binario extendido, seguido de cerca por el algoritmo de MCD extendido de Lehmer, mientras que la inversa de Montgomery modificada resultó con un desempeño algo peor. Es importante notar que debido a la naturaleza de estos algoritmos, su comportamiento puede depender mucho del tipo de computador en que se implemente, así como de la programación y compilación del algoritmo en sí. En el presente trabajo se intentó realizar implementaciones fieles a los algoritmos originales, pero a la vez se tomó ventaja de todas las posibles optimizaciones ofrecidas por el software utilizado.

Estos valores serán entonces los que se agregarán a los correspondientes a la etapa de precomputación (si existe) y de exponenciación. Específicamente, los resultados presentados corresponderán a la ecuación:

```
mult. modulares totales = mult. modulares equivalentes para inversa +
                          mult. modulares en precomputación +
                          mult. modulares en exponenciación
```

Es importante notar que existe la probabilidad de que la base M no sea inversible con respecto al módulo n (como $n = p \cdot q$, entonces p o q serán en ese caso factores de M). En ese caso particular, los algoritmos recodificados no serán aplicables, y toda aplicación que implemente este tipo de algoritmo debe considerar esta posibilidad. La solución más directa es que el paso de obtener la inversa modular sea el primero de todos. Si este paso falla, indicando que la base M no es inversible módulo n, entonces cualquiera de los métodos clásicos puede utilizarse. En nuestras pruebas, este caso se dio en un porcentaje sensiblemente menor al 0.1%, lo que indica la baja probabilidad de este tipo situación.

Los resultados obtenidos corresponden a aplicar los distintos algoritmos implementados para calcular la $M^e \bmod(n)$ a 100 grupos de enteros M, e y n de la misma longitud binaria, generados al azar. Una vez calculadas las cien exponenciaciones, los resultados fueron promediados. Para mantener la comparación entre algoritmos justa, exactamente los mismos datos de entrada fueron utilizados para cada método distinto.

Se generaron ocho distintos grupos de operandos para las experimentaciones, de 512, 1024, 1536, 2048, 2560, 3072, 3584 y 4096 dígitos binarios. La elección de estas

longitudes se basa en las longitudes de clave (o módulo) utilizadas o sugeridas actualmente para el criptosistema RSA, y en la posible expansión de las mismas en el futuro para contrarrestar los avances en el poder de cómputo de los microprocesadores. RSA Laboratories [24] mantiene una lista de longitudes recomendadas para criptosistemas RSA, que en el momento de publicación de este trabajo sugiere la utilización de claves de 1024 bits para uso corporativo y de 2048 bits para claves extremadamente importantes, tales como las correspondientes a una autoridad certificadora (CA).

A continuación presentamos una descripción de cada método adaptado a la utilización de exponentes recodificados, junto con los resultados obtenidos en su implementación, contrastándolos con los métodos clásicos. En los casos en que una etapa de precomputación es necesaria (método *m*-ario y método de ventana deslizante), presentamos un estudio detallado de la cantidad de precomputaciones a realizar. En el resto del presente trabajo, en aras de mayor claridad, se utilizará un abuso de notación, mencionando en ocasión a los métodos adaptados para utilizar exponentes recodificados como *métodos recodificados*.

6.1 Método binario adaptado para exponentes recodificados

La adaptación del método binario para hacer uso de exponentes recodificados es relativamente directa, ya que no es necesario ningún tipo de precomputación, exceptuando el cálculo de la inversa modular de la base. La única modificación adicional a realizar en el algoritmo ocurre en el paso en que se multiplica el resultado parcial por la base M al encontrar un bit no nulo. En el caso de un exponente recodificado, este bit puede ser igual tanto a 1 como a -1 (excepto en el caso del bit más significativo, que será siempre igual a 1). En el caso negativo, el resultado parcial debe multiplicarse por la inversa de la base, previamente calculada. El algoritmo completo, al que denominamos *método binario recodificado*, se puede describir de la siguiente manera:

Método binario recodificado

Entrada: M, M^{-1}, e, n.

Salida: $C = M^e \bmod(n)$.

1. Obtener el exponente recodificado con signo f a partir de e.
2. **Si** $f_k = 1$ **entonces** $C := M$ **si no** $C := 1$
3. **Desde** $i = k-1$ **hasta** 0
   - **3.1** $C := C \cdot C \bmod(n)$
   - **3.2** **Si** $f_i = 1$ **entonces** $C := C \cdot M \bmod(n)$
   - **3.3** **Si** $f_i = \bar{1}$ **entonces** $C := C \cdot M^{-1} \bmod(n)$
4. **Devolver** C

A continuación se presentan los resultados obtenidos al comparar el método binario clásico con el método binario recodificado:

	Método Binario	Método binario recodificado					
		Inv. Montgomery		MCD Ext. Lehmer		MCD Binario Ext.	
bits	Mult Mod	Mult Mod	Diferencia	Mult Mod	Diferencia	Mult Mod	Diferencia
512	773.74	696.71	-9.95%	692.43	-10.51%	691.43	-10.64%
1024	1544.64	1381.55	-10.56%	1376.82	-10.86%	1376.05	-10.91%
1536	2298.17	2053.48	-10.65%	2049.04	-10.84%	2048.13	-10.88%
2048	3122.05	2788.01	-10.70%	2782.73	-10.87%	2782.13	-10.89%
2560	3826.49	3419.19	-10.64%	3414.04	-10.78%	3413.61	-10.79%
3072	4605.25	4117.11	-10.60%	4111.73	-10.72%	4111.06	-10.73%
3584	5369.95	4784.05	-10.91%	4778.90	-11.01%	4778.54	-11.01%
4096	6142.68	5475.18	-10.87%	5470.05	-10.95%	5469.53	-10.96%

Se puede apreciar que los ahorros obtenidos mediante la utilización del método binario recodificado son muy apreciables, hecho que se basa directamente en el menor peso de Hamming del exponente recodificado con respecto al exponente original. Esto repercute directamente en la cantidad de multiplicaciones que realiza el método binario, que además de los cuadrados realiza una multiplicación por cada dígito binario (o binario recodificado) no nulo del exponente. Los ahorros logrados se encuentran en el rango de entre el 10% y el 11%, aproximadamente, ratificando el resultado teórico detallado en §4.2.

6.2 Precomputación en métodos *m*-arios

Al adaptar los métodos *m*-arios para hacer uso de los exponentes recodificados, es importante notar que, al existir ahora bits negativos, el valor de los bloques particionados F_i puede ser menor a cero. Esto lleva a ampliar la precomputación de valores para incluir los M^{-w} para todos los valores negativos $-w$ que puedan ocurrir en la expansión particionada del exponente e. Sin embargo, podemos utilizar la ventaja de conocer que el exponente recodificado canónicamente cumple con la propiedad de no-adyacencia para evitar calcular valores superfluos.

Por ejemplo, veamos los valores posibles de los bloques de 3 bits correspondientes al método octal:

$001 = 1$	$010 = 2$	$100 = 4$
$00\bar{1} = -1$	$0\bar{1}0 = -2$	$\bar{1}00 = -4$
		$101 = 5$
		$10\bar{1} = 3$
		$\bar{1}01 = -3$
		$\bar{1}0\bar{1} = -5$

Analizando en forma general las precomputaciones necesarias en un método *m*-ario. Los posibles valores de la ventana incluyen todos los enteros (tanto positivos como negativos) desde 1 hasta el máximo valor posible del bloque de *m* bits, al que denominaremos w_{MAX}. Para las ventanas de longitud impar, el máximo valor posible (considerando la propiedad de no-adyacencia) ocurre cuando todos los bits en posición par son no-nulos, y todos los

bits en posición impar son nulos. Presentamos como ejemplo los valores máximos de ventana para algunos valores impares de m:

m	w_{MAX}
3	$(101)_2 = 5$
5	$(10101)_2 = 21$
7	$(1010101)_2 = 85$
9	$(101010101)_2 = 341$

En general para cualquier valor impar de m, este valor será igual a:

$$w_{MAX} = \sum_{i=0}^{m-1} 2^i \cdot ((i+1) \bmod (2))$$

Para las ventanas de longitud par, el máximo valor posible corresponderá a la ventana cuyos bits en posición impar son todos no-nulos, mientras que todos los bits en posiciones pares son nulos. Ahora presentamos como ejemplo los valores máximos de ventana para algunos valores pares de m:

m	w_{MAX}
4	$(1010)_2 = 10$
6	$(101010)_2 = 42$
8	$(10101010)_2 = 170$
10	$(1010101010)_2 = 682$

En general para cualquier valor par de m, este valor será igual a:

$$w_{MAX} = \sum_{i=0}^{m-1} 2^i \cdot (i \bmod (2))$$

La precomputación entonces significará calcular $M^w \bmod(n)$ y $M^{-w} \bmod(n)$ para $w = 2,3,4,\ldots,w_{MAX}$, lo que genera una cadena de $w_{MAX} - 1$ multiplicaciones modulares. La propiedad de que $M^{-w} \bmod(n) = (M^{-1})^w \bmod(n)$ nos da la pauta de que se necesitará la misma cantidad de precomputaciones para valores positivos y para valores positivos de los bloques de longitud m. Esto significa que la misma estrategia de precomputación se puede utilizar para los valores positivos y los negativos, tomando como base de las exponenciaciones a M en el primer caso y M^{-1} en el segundo. Por lo tanto, duplicar la cantidad de precomputaciones para los valores de bloque positivos equivaldrá a calcular la cantidad de precomputaciones para los valores positivos y negativos. Este valor será igual a

$$2 \cdot (w_{MAX} - 1), \text{ donde } w_{MAX} = \begin{cases} \sum_{i=0}^{m-1} 2^i \cdot ((i+1) \bmod (2)), si\ m\ es\ par \\ \sum_{i=0}^{m-1} 2^i \cdot (i \bmod (2)), si\ m\ es\ impar \end{cases}$$

6.3 Método *m*-ario adaptado para exponentes recodificados

El algoritmo *m*-ario de exponenciación utilizando exponentes recodificados puede describirse entonces de la siguiente manera:

Método *m*-ario recodificado

Entrada: M, e, n

Salida: $C = M^e \bmod(n)$

1. Calcular y almacenar $M^w \bmod(n)$ y $M^{-w} \bmod(n)$ para $w = 2,3,...,m-1$.
2. Partir e en secciones de r bits F_i para $i = 0,1,...,s-1$.
3. $C \leftarrow M^{F_{s-1}} \bmod(n)$
4. **Desde** $i = s-2$ **hasta** 0
   - 4.1 $C \leftarrow C^{2^r} \bmod(n)$
   - 4.2 **Si** $F_i \neq 0$ **entonces** $C \leftarrow C \cdot M^{F_i} \bmod(n)$
5. **Devolver** C

Se presentan ahora los resultados obtenidos al comparar el método cuaternario clásico y el método octal clásico con sus contrapartes adaptadas para exponentes recodificados:

	Método cuaternario	Método cuaternario recodificado					
		Inv. Montgomery		MCD Ext. Lehmer		MCD Binario Ext.	
Bits	Mult Mod	Mult Mod	Diferencia	Mult Mod	Diferencia	Mult Mod	Diferencia
512	710.45	695.71	-2.07%	691.43	-2.68%	690.43	-2.82%
1024	1418.07	1380.55	-2.65%	1375.82	-2.98%	1375.05	-3.03%
1536	2109.26	2055.48	-2.55%	2051.04	-2.76%	2050.13	-2.80%
2048	2864.02	2787.01	-2.69%	2781.73	-2.87%	2781.13	-2.89%
2560	3511.07	3418.19	-2.65%	3413.04	-2.79%	3412.61	-2.80%
3072	4225.03	4116.11	-2.58%	4110.73	-2.71%	4110.06	-2.72%
3584	4923.82	4786.05	-2.80%	4780.90	-2.90%	4780.54	-2.91%
4096	5635.86	5475.18	-2.85%	5470.05	-2.94%	5469.53	-2.95%

	Método octal	Método octal recodificado					
		Inv. Montgomery		MCD Ext. Lehmer		MCD Binario Ext.	
Bits	Mult Mod	Mult Mod	Diferencia	Mult Mod	Diferencia	Mult Mod	Diferencia
512	668.03	671.71	0.55%	667.43	-0.09%	666.43	-0.24%
1024	1333.79	1330.55	-0.24%	1325.82	-0.60%	1325.05	-0.66%
1536	1984.98	1973.48	-0.58%	1969.04	-0.80%	1968.13	-0.85%
2048	2694.71	2676.01	-0.69%	2670.73	-0.89%	2670.13	-0.91%
2560	3302.19	3279.19	-0.70%	3274.04	-0.85%	3273.61	-0.87%
3072	3974.60	3943.11	-0.79%	3937.73	-0.93%	3937.06	-0.94%
3584	4627.85	4590.05	-0.82%	4584.90	-0.93%	4584.54	-0.94%
4096	5295.25	5254.18	-0.78%	5249.05	-0.87%	5248.53	-0.88%

Los ahorros obtenidos son mucho menores que en el caso binario recodificado, sobre todo a medida que los bloques que se toman en cuenta son mayores, como se puede observar comparando los métodos cuaternario y octal.

La primera razón para este comportamiento es la cantidad de precomputaciones. El caso cuaternario clásico ($m-2=4-2=2$ precomputaciones) y el cuaternario recodificado ($2\cdot(w_{MAX}-1)=2.1=2$ precomputaciones) presentan una cantidad equivalente de precomputaciones, pero en el caso octal clásico (6 precomputaciones) y octal recodificado (8 precomputaciones) ya existe una diferencia.

Además, para reducir la cantidad de multiplicaciones durante el proceso de exponenciación, deben generarse mayor cantidad de bloques de valor cero en el método recodificado que en el método clásico. Claramente, a medida que los bloques considerados son de mayor longitud, la posibilidad de un bloque de valor cero disminuye.

Los experimentos no incluyen valores de m mayores porque el método de ventana deslizante presenta una alternativa más atractiva y optimizada, como se puede observar si se compara el método octal con el método de ventana deslizante tamaño 3 de la próxima sección.

6.3 Precomputación en métodos de ventana deslizante

Los métodos de ventana deslizante se adaptan fácilmente para ser aplicados a exponentes recodificados. La única modificación importante, de manera análoga a los métodos *m*-arios, es que la precomputación debe contemplar la posibilidad de bits negativos, lo que puede significar que el valor de las ventanas puede ser a su vez negativo. Sin embargo, veremos que esto no implica que la cantidad de valores a precalcular sea el doble, debido la naturaleza de los exponentes recodificados canónicos.

Se mencionó anteriormente que en el método de ventana deslizante original debía de calcularse $M^w \bmod(n)$ para $w=3,5,7,\ldots,2^d-1$, es decir, para todos los valores impares posibles (ya que el valor del bit menos significativo es siempre no nulo) de una ventana de longitud máxima d. Sin embargo, recordemos que una de las características de la recodificación canónica es la *no-adyacencia*, lo que implica que no se encontrarán dos bits no nulos adyacentes. Esto reduce la cantidad de precomputaciones a realizar, ya que no será necesario calcular $M^w \bmod(n)$ para ciertos w que no son posibles en un exponente que cumple con la propiedad de no-adyacencia.

Como ejemplo, supongamos ventanas deslizantes de tamaño máximo $d=5$. En un método de ventana deslizante aplicado a un exponente binario tradicional, los valores posibles de la ventana serán todos los números impares entre 1 y $2^5-1=31$, un total de 16 valores (2 posibilidades para cada bit de la ventana, exceptuando el menos significativo, que es siempre 1), y por lo tanto existirán 16 precomputaciones a realizar. Si consideramos que los exponentes recodificados cumplen con la propiedad de no-adyacencia, entonces podemos listar las posibles ventanas con sus correspondientes valores decimales w:

1	$=1$	101	$=5$	1001	$=9$	10001	$=17$
$\bar{1}$	$=-1$	$10\bar{1}$	$=3$	$100\bar{1}$	$=7$	$1000\bar{1}$	$=15$
		$\bar{1}01$	$=-3$	$\bar{1}001$	$=-7$	$\bar{1}0001$	$=-15$
		$\bar{1}0\bar{1}$	$=-5$	$\bar{1}00\bar{1}$	$=-9$	$\bar{1}000\bar{1}$	$=-17$
						10101	$=21$
						$1010\bar{1}$	$=19$
						$\bar{1}0101$	$=-11$
						$\bar{1}010\bar{1}$	$=-13$
						$10\bar{1}01$	$=13$
						$10\bar{1}0\bar{1}$	$=11$
						$\bar{1}0\bar{1}01$	$=-19$
						$\bar{1}0\bar{1}0\bar{1}$	$=-21$

Podemos notar que a pesar de contar con bits con signo, que introducen la posibilidad de ventanas con valor negativo, los valores precomputados a calcular con una ventana de tamaño máximo $d = 5$ son 22, una cantidad apenas superior a las 16 precomputaciones que encontramos en el algoritmo de ventana deslizante clásico.

Realizaremos ahora un análisis general para las precomputaciones a realizar si se considera una ventana de tamaño máximo d. Podemos observar que los posibles valores de la ventana incluyen todos los enteros impares (tanto positivos como negativos) desde 1 hasta el máximo valor posible de la ventana, al que denominaremos w_{MAX}. Para las ventanas de longitud impar, el máximo valor posible (considerando la propiedad de no-adyacencia) ocurre cuando todos los bits en posición par son no-nulos, y todos los bits en posición impar son nulos. Presentamos como ejemplo los valores máximos de ventana para algunos valores impares de d:

d	w_{MAX}
3	$(101)_2 = 5$
5	$(10101)_2 = 21$
7	$(1010101)_2 = 85$
9	$(101010101)_2 = 341$

En general para cualquier valor impar de d, este valor será igual a:

$$w_{MAX} = \sum_{i=0}^{d-1} 2^i \cdot ((i+1) \bmod (2))$$

Para las ventanas de longitud par, el máximo valor posible corresponderá a la ventana con todos los bits en posición impar entre las posiciones $d-1$ y 3 no-nulos, y todos los bits en posiciones pares en ese mismo rango nulos. El bit menos significativo (posición 0) será siempre no-nulo, por las características de las ventanas, y los bits en posición 1 y 2 serán nulos. Ahora presentamos como ejemplo los valores máximos de ventana para algunos valores pares de d:

d	w_{MAX}
4	$(1001)_2 = 9$
6	$(101001)_2 = 41$
8	$(10101001)_2 = 169$
10	$(1010101001)_2 = 681$

En general para cualquier valor par de d, este valor será igual a:

$$w_{MAX} = 1 + \sum_{i=3}^{d-1} 2^i \cdot (i \bmod(2))$$

Ahora bien, queda por determinar la cantidad de multiplicaciones modulares que serán necesarias para realizar la precomputación. Podemos definir esta precomputación como el cálculo y almacenamiento de $M^w \bmod(n)$ y $M^{-w} \bmod(n)$ para $w = 3,5,\ldots,w_{MAX}$. Se puede observar que se necesitará la misma cantidad de precomputaciones tanto para los valores de ventana positivos como para los negativos, utilizando la propiedad de que $M^{-w} \bmod(n) = (M^{-1})^w \bmod(n)$. Esto significa que la misma estrategia de precomputación se puede utilizar para los valores positivos y los negativos, tomando como base de las exponenciaciones a M en el primer caso y M^{-1} en el segundo. Por lo tanto, calcular la cantidad de precomputaciones para los valores de ventana positivos y luego duplicar este número es equivalente a calcular la cantidad de precomputaciones para los valores positivos y negativos.

Asumiendo un valor de ventana máximo w_{MAX}, la cadena de multiplicaciones a realizar para calcular todos los valores necesarios será:

$$\begin{aligned} M \cdot M &= M^2 \\ M^2 \cdot M &= M^3 \\ M^3 \cdot M^2 &= M^5 \\ &\vdots \\ M^{w_{MAX}-2} \cdot M^2 &= M^{w_{MAX}} \end{aligned}$$

Se puede observar que el valor de M^2, si bien no es una precomputación necesaria para el algoritmo de ventana deslizante, sí resulta fundamental para cada cálculo subsiguiente en la cadena. En general, el número de precomputaciones para los valores de ventana positivos será de $(w_{MAX}+1)/2$. Por lo tanto, duplicando este número, arribamos a la cantidad de multiplicaciones necesarias para realizar la precomputación en cualquier método de ventana deslizante con ventana de tamaño máximo d:

$$w_{MAX} + 1, \text{ donde } w_{MAX} = \begin{cases} \sum_{i=0}^{d-1} 2^i \cdot ((i+1) \bmod(2)), \textit{si } d \textit{ es par} \\ 1 + \sum_{i=3}^{d-1} 2^i \cdot (i \bmod(2)), \textit{si } d \textit{ es impar} \end{cases}$$

Presentamos a continuación una comparación de la cantidad de multiplicaciones modulares necesarias para realizar la precomputación en los algoritmos de ventana deslizante, tanto el clásico como el modificado para utilizar exponentes recodificados, para longitudes máximas de ventana $3 \le d \le 10$:

d	Ventana deslizante clásico	Ventana deslizante recodificado	Diferencia	Porcentaje
3	4	6	2	+50.00%
4	8	10	2	+25.00%
5	16	22	6	+37.50%
6	32	42	10	+31.25%
7	64	86	22	+34.38%
8	128	170	42	+32.81%
9	256	342	86	+33.59%
10	512	682	170	+33.20%

6.5 Método de ventana deslizante adaptado para exponentes recodificados

El algoritmo de ventana deslizante modificado para utilizar exponentes recodificados puede describirse entonces de la siguiente manera:

Método de ventana deslizante recodificado

Entrada: M, e, n, d

Salida: $C = M^e \bmod(n)$

1. Computar y almacenar $M^w \bmod(n)$ y $M^{-w} \bmod(n)$ para $w = 3,5,\ldots,w_{MAX}$
2. Particionar e en ventanas cero y no-cero F_i de largo $L(F_i)$, donde $L(F_i) \le d$
3. $C \leftarrow M^{F_{k-1}} \bmod(n)$
4. **Desde** $i = p-2$ **hasta** 0
   - 4.1 $C \leftarrow C^{2^{L(F_i)}} \bmod(n)$
   - 4.2 **Si** $F_i \neq 0$ **entonces** $C \leftarrow C \cdot M^{F_i} \bmod(n)$
5. **Devolver** C

A continuación presentamos los resultados obtenidos para diferentes tamaños de ventana, desde 3 hasta 10, comparando el método tradicional con el método que utiliza exponentes recodificados:

	Ventana tamaño 3	Ventana tamaño 3 recodificada					
		Inv. Montgomery		MCD Ext. Lehmer		MCD Binario Ext.	
bits	Mult Mod	Mult Mod	Diferencia	Mult Mod	Diferencia	Mult Mod	Diferencia
512	648.54	644.71	-0.59%	640.43	-1.25%	639.43	-1.41%
1024	1291.08	1273.55	-1.36%	1268.82	-1.72%	1268.05	-1.78%
1536	1921.76	1889.48	-1.68%	1885.04	-1.91%	1884.13	-1.96%
2048	2608.85	2561.01	-1.83%	2555.73	-2.04%	2555.13	-2.06%
2560	3193.49	3141.19	-1.64%	3136.04	-1.80%	3135.61	-1.81%
3072	3846.48	3773.11	-1.91%	3767.73	-2.05%	3767.06	-2.06%
3584	4479.39	4392.05	-1.95%	4386.90	-2.06%	4386.54	-2.07%
4096	5125.65	5027.18	-1.92%	5022.05	-2.02%	5021.53	-2.03%

	Ventana tamaño 4	Ventana tamaño 4 recodificada					
		Inv. Montgomery		MCD Ext. Lehmer		MCD Binario Ext.	
bits	Mult Mod	Mult Mod	Diferencia	Mult Mod	Diferencia	Mult Mod	Diferencia
512	626.17	631.71	0.89%	627.43	0.20%	626.43	0.04%
1024	1243.37	1243.55	0.01%	1238.82	-0.37%	1238.05	-0.43%
1536	1849.47	1844.48	-0.27%	1840.04	-0.51%	1839.13	-0.56%
2048	2508.75	2500.01	-0.35%	2494.73	-0.56%	2494.13	-0.58%
2560	3075.51	3063.19	-0.40%	3058.04	-0.57%	3057.61	-0.58%
3072	3695.79	3680.11	-0.42%	3674.73	-0.57%	3674.06	-0.59%
3584	4307.44	4285.05	-0.52%	4279.90	-0.64%	4279.54	-0.65%
4096	4927.30	4896.18	-0.63%	4891.05	-0.74%	4890.53	-0.75%

	Ventana tamaño 5	Ventana tamaño 5 recodificada					
		Inv. Montgomery		MCD Ext. Lehmer		MCD Binario Ext.	
bits	Mult Mod	Mult Mod	Diferencia	Mult Mod	Diferencia	Mult Mod	Diferencia
512	617.89	626.71	1.43%	622.43	0.73%	621.43	0.57%
1024	1217.48	1222.55	0.42%	1217.82	0.03%	1217.05	-0.04%
1536	1806.88	1805.48	-0.08%	1801.04	-0.32%	1800.13	-0.37%
2048	2446.20	2442.01	-0.17%	2436.73	-0.39%	2436.13	-0.41%
2560	2998.94	2988.19	-0.36%	2983.04	-0.53%	2982.61	-0.54%
3072	3603.85	3585.11	-0.52%	3579.73	-0.67%	3579.06	-0.69%
3584	4196.49	4177.05	-0.46%	4171.90	-0.59%	4171.54	-0.59%
4096	4798.60	4771.18	-0.57%	4766.05	-0.68%	4765.53	-0.69%

	Ventana tamaño 6	Ventana tamaño 6 recodificada					
		Inv. Montgomery		MCD Ext. Lehmer		MCD Binario Ext.	
bits	Mult Mod	Mult Mod	Diferencia	Mult Mod	Diferencia	Mult Mod	Diferencia
512	620.88	636.71	2.55%	632.43	1.86%	631.43	1.70%
1024	1209.73	1220.55	0.89%	1215.82	0.50%	1215.05	0.44%
1536	1785.64	1794.48	0.49%	1790.04	0.25%	1789.13	0.20%
2048	2412.20	2419.01	0.28%	2413.73	0.06%	2413.13	0.04%
2560	2953.60	2957.19	0.12%	2952.04	-0.05%	2951.61	-0.07%
3072	3546.24	3545.11	-0.03%	3539.73	-0.18%	3539.06	-0.20%
3584	4126.99	4124.05	-0.07%	4118.90	-0.20%	4118.54	-0.20%
4096	4716.34	4710.18	-0.13%	4705.05	-0.24%	4704.53	-0.25%

	Ventana tamaño 7	Ventana tamaño 7 recodificada					
		Inv. Montgomery		MCD Ext. Lehmer		MCD Binario Ext.	
bits	Mult Mod	Mult Mod	Diferencia	Mult Mod	Diferencia	Mult Mod	Diferencia
512	643.67	671.71	4.36%	667.43	3.69%	666.43	3.54%
1024	1222.38	1247.55	2.06%	1242.82	1.67%	1242.05	1.61%
1536	1790.20	1813.48	1.30%	1809.04	1.05%	1808.13	1.00%
2048	2408.42	2427.01	0.77%	2421.73	0.55%	2421.13	0.53%
2560	2939.57	2958.19	0.63%	2953.04	0.46%	2952.61	0.44%
3072	3522.79	3537.11	0.41%	3531.73	0.25%	3531.06	0.23%
3584	4096.47	4107.05	0.26%	4101.90	0.13%	4101.54	0.12%
4096	4674.97	4685.18	0.22%	4680.05	0.11%	4679.53	0.10%

	Ventana tamaño 8	Ventana tamaño 8 recodificada					
		Inv. Montgomery		MCD Ext. Lehmer		MCD Binario Ext.	
bits	Mult Mod	Mult Mod	Diferencia	Mult Mod	Diferencia	Mult Mod	Diferencia
512	700.15	749.71	7.08%	745.43	6.47%	744.43	6.32%
1024	1272.34	1318.55	3.63%	1313.82	3.26%	1313.05	3.20%
1536	1833.70	1877.48	2.39%	1873.04	2.15%	1872.13	2.10%
2048	2442.47	2486.01	1.78%	2480.73	1.57%	2480.13	1.54%
2560	2968.57	3010.19	1.40%	3005.04	1.23%	3004.61	1.21%
3072	3543.19	3582.11	1.10%	3576.73	0.95%	3576.06	0.93%
3584	4108.52	4147.05	0.94%	4141.90	0.81%	4141.54	0.80%
4096	4682.08	4720.18	0.81%	4715.05	0.70%	4714.53	0.69%

	Ventana tamaño 9	Ventana tamaño 9 recodificada					
		Inv. Montgomery		MCD Ext. Lehmer		MCD Binario Ext.	
bits	Mult Mod	Mult Mod	Diferencia	Mult Mod	Diferencia	Mult Mod	Diferencia
512	822.03	915.71	11.40%	911.43	10.88%	910.43	10.75%
1024	1389.70	1479.55	6.46%	1474.82	6.12%	1474.05	6.07%
1536	1943.37	2033.48	4.64%	2029.04	4.41%	2028.13	4.36%
2048	2547.68	2636.01	3.47%	2630.73	3.26%	2630.13	3.24%
2560	3067.19	3154.19	2.84%	3149.04	2.67%	3148.61	2.65%
3072	3636.25	3722.11	2.36%	3716.73	2.21%	3716.06	2.19%
3584	4198.53	4281.05	1.97%	4275.90	1.84%	4275.54	1.83%
4096	4765.32	4846.18	1.70%	4841.05	1.59%	4840.53	1.58%

	Ventana tamaño 10	Ventana tamaño 10 recodificada					
		Inv. Montgomery		MCD Ext. Lehmer		MCD Binario Ext.	
bits	Mult Mod	Mult Mod	Diferencia	Mult Mod	Diferencia	Mult Mod	Diferencia
512	1074.56	1251.71	16.49%	1247.43	16.09%	1246.43	15.99%
1024	1635.15	1811.55	10.79%	1806.82	10.50%	1806.05	10.45%
1536	2186.44	2360.48	7.96%	2356.04	7.76%	2355.13	7.72%
2048	2783.19	2958.01	6.28%	2952.73	6.09%	2952.13	6.07%
2560	3300.27	3473.19	5.24%	3468.04	5.08%	3467.61	5.07%
3072	3865.08	4036.11	4.43%	4030.73	4.29%	4030.06	4.27%
3584	4422.14	4591.05	3.82%	4585.90	3.70%	4585.54	3.70%
4096	4985.80	5152.18	3.34%	5147.05	3.23%	5146.53	3.22%

Analizando los resultados dentro de cada longitud de ventana individual, se puede apreciar que los ahorros son mayores a medida que se consideran operandos de mayor longitud binaria k.

La primera razón para este comportamiento es que a medida que los operandos considerados sean mayores, la proporción de multiplicaciones modulares requeridas por la precomputación será mucho menor en comparación a las multiplicaciones modulares requeridas por el proceso de exponenciación. Como un ejemplo ilustrativo de este hecho, tomemos los resultados de la ventana de tamaño 5: con operandos de 512 bits, la precomputación (22 multiplicaciones modulares) representa alrededor del 3.5% del total, mientras que si se consideran operandos de 4096 bits, este porcentaje disminuye al 0.46%.

El segundo factor de influencia, mucho menor que el primero, es que a medida que las longitudes de operando son mayores, la posibilidad de generar un mayor porcentaje

relativo de ventanas de tamaño cero se incrementa. Tomando la misma situación del ejemplo anterior, observamos que con operandos de 512 bits el proceso de exponenciación en el método recodificado realiza alrededor de 0.83% menos multiplicaciones modulares que el método clásico. Este porcentaje de ahorros se eleva a 0.88% en operandos de 4096 bits.

Por otro lado, si se observa el comportamiento de los métodos recodificados a medida que se incrementa el tamaño de ventana, su desempeño tiende a empeorar, al punto de no registrarse ahorros para ninguna longitud de operando cuando las ventanas tienen una longitud máxima de 7 bits o más. Lo que explica esta degradación es que el aumento en la cantidad de precomputaciones a realizar con ventanas de mayor longitud contrarresta cualquier tipo de ahorro logrado mediante la recodificación del exponente.

No se realizaron pruebas con tamaños de ventana superiores a 10, porque es evidente al analizar los resultados que cuando se alcanza este valor el tiempo necesario para las precomputaciones supera ampliamente los ahorros conseguidos mediante la recodificación y la partición en ventanas.

A continuación, basados en estos resultados, presentamos los tamaños de ventana óptimos para cada longitud de operando en la exponenciación modular:

Bits	Tamaños óptimos de ventana Método clásico	Tamaño óptimo de ventana Método recodificado
512	5	5
1024	6	6
1536	6	6
2048	7	6
2560	7	6
3072	7	7
3584	7	7
4096	7	7

7 Conclusiones

A la luz de los resultados obtenidos, creemos que los métodos recodificados presentan una alternativa de interés para la optimización de exponenciaciones modulares en el ámbito del criptosistema RSA. Si bien los métodos clásicos de ventana deslizante continúan siendo la mejor opción en términos de velocidad, los métodos recodificados pueden ofrecer un mejor desempeño que sus contrapartes análogas clásicas si se trabaja en un contexto de espacio computacional restringido en donde la precomputación de numerosos valores sea prohibitiva (tales como *smart cards* o ambientes similares) o se opta por métodos en que la precomputación se mantenga en un mínimo por cuestiones de seguridad (un adversario con acceso a los valores precomputados podría fácilmente recuperar el texto plano).

El método binario recodificado, por citar un ejemplo en que no existe ningún tipo de precomputación, ofrece un muy apreciable ahorro en velocidad cercano al 11% con respecto al método binario clásico, a cambio de una recodificación de exponente que resulta despreciable tanto en esfuerzo computacional como en espacio requerido.

Los métodos m-arios recodificados también ofrecen un desempeño superior a sus contrapartes clásicas, lográndose ahorros de entre 2.70% y 3% en el caso cuaternario (en el que la cantidad de precomputaciones es igual para el método clásico y para el recodificado) y entre 0.25% y 0.95% en el caso octal (a pesar de que el método octal clásico realiza menos precomputaciones que el método octal recodificado).

Con respecto a los métodos de ventana deslizante, las adaptaciones a exponentes recodificados presentan mejores desempeños en prácticamente todos los casos con tamaños de ventana entre 3 y 5, y en ciertos casos para ventanas de tamaño 6.

A la luz de estos resultados, creemos que los métodos recodificados son firmes contendientes de los métodos clásicos para aplicaciones en que se manejen grandes longitudes de clave, y en ambientes en que la simplicidad y el ahorro en espacio de precomputación sean esenciales. Optar por una leve degradación en rendimiento puede significar un ahorro importante en espacio de almacenamiento de valores precomputados.

Por ejemplo, si se opta por el método de ventana deslizante adaptado para exponentes recodificados, utilizando el algoritmo de MCD binario extendido para el cálculo de inversa modular, con un tamaño máximo de ventana de 5 bits, los tiempos logrados son apenas entre un 0.2% y un 1% peores que un método de ventana deslizante clásico con tamaño máximo de ventana de 6 bits. Sin embargo, las precomputaciones necesarias serán 22 en el primer caso, mientras que en el segundo serán 32, requiriendo aproximadamente un 45% más de espacio para su almacenamiento.

A continuación presentamos un análisis más detallado, para cada longitud de operando considerada, de las diferencias en precomputaciones y multiplicaciones modulares entre utilizar un método de ventana deslizante clásico con ventana máxima de tamaño d^* (en donde d^* es el valor óptimo de ventana para cada operando, como se detallan en §6) y utilizar un método de ventana deslizante modificado para utilizar exponentes recodificados con ventana máxima de tamaño d^*-1 y tamaño d^*-2. El algoritmo de inversa modular considerado es el GCD binario extendido, que es el que mejores resultados ofrece entre los implementados.

	Ventana deslizante clásico			Ventana deslizante recodificado			Diferencia	
bits	***d****	**Mult Mod**	**Precomp.**	**d*-1**	**Mult Mod**	**Precomp**	**Mult Mod**	**Precomp**
512	5	617.89	16	4	626.43	10	*1.38%*	*-37.50%*
1024	6	1209.73	32	5	1217.05	22	*0.61%*	*-31.25%*
1538	6	1785.64	32	5	1800.13	22	*0.81%*	*-31.25%*
2048	7	2408.42	64	6	2413.13	42	*0.20%*	*-34.38%*
2560	7	2939.57	64	6	2951.61	42	*0.41%*	*-34.38%*
3072	7	3522.79	64	6	3539.06	42	*0.46%*	*-34.38%*
3584	7	4096.47	64	6	4118.54	42	*0.54%*	*-34.38%*
4096	7	4674.97	64	6	4704.53	42	*0.63%*	*-34.38%*

	Ventana deslizante clásico			Ventana deslizante recodificado			Diferencia	
bits	***D****	**Mult Mod**	**Precomp.**	**d*-2**	**Mult Mod**	**Precomp**	**Mult Mod**	**Precomp**
512	5	617.89	16	3	639.43	6	*3.49%*	*-62.50%*
1024	6	1209.73	32	4	1238.05	10	*2.34%*	*-68.75%*
1538	6	1785.64	32	4	1839.13	10	*3.00%*	*-68.75%*
2048	7	2408.42	64	5	2436.13	22	*1.15%*	*-65.63%*
2560	7	2939.57	64	5	2982.61	22	*1.46%*	*-65.63%*
3072	7	3522.79	64	5	3579.06	22	*1.60%*	*-65.63%*
3584	7	4096.47	64	5	4171.54	22	*1.83%*	*-65.63%*
4096	7	4674.97	64	5	4765.53	22	*1.94%*	*-65.63%*

Al analizar estos resultados vemos que optar por utilizar un método de ventana deslizante adaptado para exponentes recodificados con un tamaño máximo de ventana menor al óptimo en su contraparte clásica genera un lógico deterioro en su desempeño, pero los ahorros en la cantidad de precomputaciones a realizar son comparativamente mucho mayores.

Puede observarse que los ahorros en precomputaciones serían aún mayores si se utilizara un método de ventana deslizante clásico con longitudes de ventana reducidas en lugar del método recodificado, llegando al 50% al reducir la ventana en un bit y al 75% si se la recorta en dos bits. Sin embargo, la degradación en desempeño sería en general más pronunciada, ya que los métodos recodificados se comportan en general mejor que los clásicos cuando se consideran longitudes de ventana de entre 3 y 6 bits. Consideramos que los métodos recodificados proveen de un interesante término medio si se adopta esta visión, generando importantes ahorros en espacio de precomputación a cambio de leves deterioros en tiempo, y merecen ser tomados en cuenta a la hora de seleccionar el método de exponenciación en un criptosistema RSA.

Los métodos de cálculo de inversa, recodificación y exponenciación considerados en este trabajo no son las únicas variantes existentes, sino que fueron seleccionados por ciertas características deseables. A continuación se mencionarán otras opciones de interés.

Existen otros métodos para el cálculo de inversas modulares que pueden ser explorados, tales como el propuesto por Gordon en [27] o la versión *k*-aria del algoritmo de GCD binario de Sorenson [28]. Sin embargos, los que se investigan en el presente trabajo son considerados entre los más eficientes en la literatura.

Un acercamiento alternativo a la recodificación con signo se encuentra en las representaciones con reemplazo de cadenas, introducidas por Gollmann, Han y Mitchell en [29], en donde se reemplazan cadenas de bits no nulos de longitud máxima SR por

dígitos decimales que representan su valor. Por ejemplo, si tomamos SR=3, tenemos la siguiente recodificación posible:

$$(110111110011101)_2 = (110007110000701)_{SR3}$$

En general, pueden existir varias representaciones de la misma cadena, lo que dificulta su implementación. Sin embargo, argumentos heurísticos muestran una reducción de dígitos no nulos de alrededor del 50%, presentando un área merecedora de mayor exploración.

Numerosos otros acercamientos al problema de optimizar las exponenciaciones modulares en RSA alternativos a la utilización de exponentes recodificados pueden encontrarse en la literatura, y aquí nombraremos algunos de ellos que pueden resultar de interés.

El *método de factorización* propuesto por Knuth [2] se basa en la factorización del exponente $e = r \cdot s$, donde r es el menor factor primo de e y $s > 1$, y logra mejores resultados que el algoritmo binario clásico. Lamentablemente, el hecho de factorizar el exponente e resulta una complicación en el método RSA con grandes longitudes de clave. Una solución para esto sería construir los exponentes de manera en que esta factorización resulte simple, pero cuidando de no dañar la seguridad del criptosistema.

Otro método propuesto por Knuth en [2], conocido como *power tree*, utiliza heurísticas para construir un árbol de exponenciaciones precomputadas, para ahorrar cálculos en computaciones subsiguientes. Pero la complejidad de armar este tipo de árbol lo hace más apto para criptosistemas basados en el problema del logaritmo discreto, en donde el árbol podría mantenerse igual a lo largo de todo el proceso de encriptación y desencriptación, y no generarse para cada bloque a tratar, como sería el caso en el criptosistema RSA.

Por último, mencionamos que todos los algoritmos clásicos mencionados en el presente trabajo pueden ser vistos como métodos para generar las denominadas *cadenas de adición*, una secuencia de enteros $a_0, a_1, a_2, ..., a_r$ en que $a_0 = 1$ y para todo k entonces existen índices $i, j < k$ tal que $a_k = a_i + a_j$. Si éste concepto de cadena de adición se aplica a un exponente $e = a_r$, entonces la cadena provee de un algoritmo para el cálculo de M^e con una cantidad de multiplicaciones igual a r. Claramente, lo que se busca es que estas cadenas de adición sean lo más cortas posibles, lo que se ha probado ser un problema NP-completo [9]. Esto implica que para encontrar la cadena de adición más corta para e será necesario calcular todas las cadenas posibles. Se cuenta con un límite inferior para el largo de estas cadenas, formulado por Schönhage [31], que es igual a $\log_2 e + \log_2 H(e) + 2.13$. Muchas heurísticas alternativas y métodos estadísticos para lograr cadenas de adición lo más cercanas posible a este límite han sido propuestas, y el trabajo en esta área continúa siendo de gran interés.

Bibliografía

[1] R. L. Rivest, A. Shamir, y L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120-126, Febrero 1978.

[2] D. E. Knuth. *The Art of Computer Programming: Seminumerical Algorithms,* volumen 2. Reading, MA: Addison-Wesley, Second edition, 1981.

[3] A. J. Menezes, P. C. van Oorschot y S. A. Vanstone. *Handbook of Applied Cryptography*, página 69. CRC Press LLC, 1997.

[4] W. Diffie y M. E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22:644-654, Noviembre 1976.

[5] T. ElGamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Transactions on Information Theory*, 31(4):469-472, Julio 1985.

[6] National Institute for Standards and Technology. Digital Signature Standards (DSS). *Federal Register*, 56:169, Agosto 1991.

[7] E. F. Brickell, D. M. Gordon, K. S. McCurley, y D. B. Wilson. Fast exponentiation with precomputation. R. A. Rueppel, editor, *Advances in Cryptology – EUROCRYPT 92*, Lecture Notes in Computer Science, No. 658, pages 200-207. New York, NY: Springer-Verlag, 1992.

[8] Ç. K. Koç. High-radix and bit-recoding techniques for modular exponentiation. *International Journal of Computer Mathematics*, 40(3+4):139-156, 1991.

[9] P. Downey, B. Leony, y R. Sethi. Computing sequences with addition chains. *SIAM Journal on Computing*, 3:638-696, 1981.

[10] Ç. K. Koç. Analysis of sliding windows techniques for exponentiation. *Computers and Mathematics with Applications*, 30(10):17-24, 1995.

[11] J. Bos y M. Coster. Addition chain heuristics. G. Brassard, editor, *Advances in Cryptology – CRYPTO 89, Proceedings*, Lecture Notes in Computer Science, No. 435, páginas 400-407. New York, NY : Springer-Verlag, 1989.

[12] J. Jedwab y C. J. Mitchell. Minimum weight modified signed-digit representations and fast exponentiation. *Electronic Letters*, 25(17) :1771-1772, Agosto 1989.

[13] F. Morain and J. Olivos. Speeding up the computations on an elliptic curve using addition-substraction chains. *Rapport de recherche 983*, INRIA, Marzo 1989.

[14] Ö. Eğecioğlu y Ç. K. Koç. Fast Modular Exponentiation. E. Arkan, editor, *Communication, Control and Signal Processing : Proceedings of 1990 Bilkent International Conference on New Trends in Communication, Control and Signal Processing*, páginas 188-194, Bilkent University, Ankara, Turquía, Julio 1990.

[15] W. Bosma. Signed bits and fast exponentiation. Department of Mathematics, University of Nijmegen, Holanda.

[16] A. D. Booth. A signed binary multiplication technique. *Q. J. Mech. Appl. Math.*, 4(2):236-240, 1951.

[20] G. W. Reitweisner. Binary arithmetic. *Advances in Computers*, 1:231-308, 1960.

[21] K. Hwang. *Computer Arithmetic, Principles, Architecture and Design*. New York, NY: John Wiley & Sons, 1979.

[22] I. Koren. *Computer Arithmetic Algorithms*. Englewood Cliffs, NJ: Prentice-Hall, 1993.

[23] P. L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44(170):519-521, Abril 1985.

[24] RSA Laboratories. www.rsasecurity.com

[25] S. R. Dussé and B. S. Kaliski, Jr. A crpyotgraphic library for the Motorola DSP56000. I. B. Damgård, editor, *Advances in Cryptology – EUROCRYPT 90,* Lecture Notes in Computer Science, No. 473, páginas 230-244. New York, NY: Springer-Verlag, 1990.

[26] B. S. Kaliski, Jr. The Montgomery inverse and its applications. *IEEE Transactions on Computers*, 44(8):1064-1065, Agosto 1995.

[27] J. Gordon. Fast multiplicative inverse in modular arithmetic, H. Beker y F. Piper, editors, *Cryptography and Coding*, Institute of Mathematics & Applications (IMA), páginas 269-279. Clarendon Press, 1989.

[28] J. Sorenson. Two fast GCD algorithms, *Journal of Algorithms*, 16(1994), páginas 110-144.

[29] D. Gollmann, Y. Han y C. Mitchell. Redundant integer representations and fast exponentiation, *Designs, Codes and Cryptography*, 7(1996), páginas 135-151.

[30] GNU MP. www.swox.com/gmp

[31] A. Schönhage. A lower bound for the length of addition chains, *Theoretical Computer Science*, 1:1-12, 1975.

[32] P. W. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer, *SIAM Journal of Computing* 26, páginas 1484-1509, 1997.