DISEÑO, IMPLEMENTACIÓN Y OPTIMIZACIÓN DE ALGORITMOS CRIPTOGRÁFICOS DE GENERACIÓN DE ALEATORIOS Y FACTORIZACIÓN DE ENTEROS

Pedro María ALCOVER GARAU

Tesis propuesta para la obtención del
Grado de Doctor de Informática

Departamento de Ingeniería y Tecnología
de Computadores
Universidad de Murcia

Septiembre de 2003

Dirigida por

D. José Manuel GARCÍA CARRASCO

Universidad de Murcia

DISEÑO, IMPLEMENTACIÓN Y OPTIMIZACIÓN DE ALGORITMOS CRIPTOGRÁFICOS DE GENERACIÓN DE ALEATORIOS Y FACTORIZACIÓN DE ENTEROS

Pedro María ALCOVER GARAU

Tesis propuesta para la obtención del
Grado de Doctor de Informática

Departamento de Ingeniería y Tecnología
de Computadores

Septiembre de 2003

Dirigida por

D. José Manuel GARCÍA CARRASCO

Tú sabes porqué.

... Y tú —que lo sabes—, comprendes que no tenga nada más que decir.

ÍNDICE GENERAL

c

CUADROS

Capítulo 2

Capítulo 4

Capítulo 6

a

ALGORITMOS

Capítulo 4

Capítulo 5

Capítulo 6

t

TABLAS

Capítulo 4

Capítulo 5

Capítulo 7

g GRÁFICAS

Capítulo 5

Capítulo 6

Capítulo 7

a AGRADECIMIENTOS

Me alegra haber terminado la tesis, y he disfrutado mucho estudiando todo lo que he tenido que aprender para llegar a poner por escrito lo que poca gente va a animarse a leer. La tesis ha sido un proceso, no una finalidad. He sido feliz con ella; no gracias a ella.

Mi padre tiene mucho que ver con esta tesis. Creo que mientras vivió en esta tierra jamás le interesó la aritmética modular. Cosas más interesantes le ocupaban. Pero tiene mucho que ver con mi vida universitaria. Él es uno de los tres a quienes va dirigida la dedicatoria. Él sabe porqué.

Mi madre me cuenta a veces lo mucho que desea que termine la tesis: dice que la espera se le hace muy larga. Dice que no quiere morirse sin ver terminada esta tesis: que Dios no lo permita. Una de las principales alegrías que he tenido al terminar este trabajo ha sido liberarla de su ansiada espera. Ella es otra de las tres personas a quienes va dirigida mi encriptada dedicatoria.

El tercer protagonista de la dedicatoria me acompaña a donde quiero que vaya. Él sabe cómo factorizar números grandes pero todavía no me ha contado cómo se hace: le gusta verme buscar. Si me muero sin saberlo, tiene toda la eternidad para contármelo.

Una vez aclarados los destinatarios de la dedicatoria (quise que fuese cifrada) debería quizá ahora dar las gracias a todos los que en este tiempo me han ayudado en la confección de la tesis: A José Manuel GARCÍA, que aceptó dirigir mi tesis cuando yo no sabía ni lo que era el lenguaje C, que ha marcado la pauta de mi trabajo en todo este tiempo, y me ha dedicado las horas que le he requerido; que es, después de mi madre, la segunda persona que se ha leído mi tesis. A Llorenç HUGUET, hasta hace unos meses Rector de la UIB, que ha tenido tiempo para mí siempre que he viajado a Palma donde residen mi madre y mis hermanos; siempre me ha atendido con verdadera elegancia y con una muy amena conversación. A Juan TENA, que se ha leído todo lo que sobre números primos he escrito; que me ha orientado en su estudio y me ha aconsejado cuando he acudido a él con una prontitud que siempre me ha sorprendido. A Luis HERNÁNDEZ, que se ha leído cinco veces el capítulo sobre los generadores de bits aleatorios: Todas sus contestaciones han llegado en menos de 48 horas; puedo asegurar que no se ha leído nada en diagonal, al juzgar por sus varios folios de contestación en cada nueva versión, anotando erratas en tal página, en tal línea; sugiriendo modificaciones de redacción o de fondo, facilitando nueva documentación que mejora lo expuesto; que me dedicó tres horas en el Pabellón de Congresos Príncipe de Asturias en Oviedo. A Amparo FUSTER que me envió por correo apuntes suyos de varios cursos de criptografía, que me han sido realmente de mucha utilidad: un correo mío a quien aún no conocía y a las 72 horas tenía un paquete de más de 100 folios en mi despacho. A mi hermano Guillermo ALCOVER, que ha sido el encargado de lograr que no desviase el tiro: si he logrado terminar la tesis en estos años que le he dedicado, también ha sido gracias a su invariable mensaje, machaconamente reiterado y repetido: tesis, tesis, tesis.

El estilo de trabajo de cada uno de los arriba citados me ha deslumbrado. Su prontitud para atenderme siempre (sí: ¡ siempre!) me ha enseñado mucho. Doy gracias a Dios por no haberme acostumbrado al trato que todos ellos me han dispensado. Me encandila la perspectiva de llegar a ser, alguna vez, una persona del talante y de la categoría humana y profesional de José Manuel, de Llorenç, de Juan, de Luis, de Amparo, de mi hermano Guillermo. Quisiera poder demostrar con mi actitud futura en el mundo universitario, que he aprendido a servir como ellos. Ésa es la gratitud que quisiera brindarles.

Me gustaría hacer mención nominal de los compañeros del Departamento de Tecnología de la Información y las Comunicaciones, de la Universidad Politécnica de Cartagena, donde tengo la suerte de trabajar; son tantos que pienso que, si lo hiciera, estas líneas de agradecimiento podrían confundirse con las páginas de una guía telefónica.

Quiero dar la enhorabuena a los que trabajan conmigo (o yo con ellos) en el Club ESTAY: han sabido atender mis explicaciones sobre los generadores, la factorización, el Linux, los números primos,... E incluso a veces han logrado hacerme dudar si les interesaba o no.

r

RESUMEN

Estudio. Análisis. Diseño. Implementación. Optimización. Y en todo momento deslumbramiento. Estas cinco palabras, y esta actitud de fondo, logran resumir muy escuetamente el trabajo de esta tesis.

Estudio de la aritmética modular; de las propiedades de los enteros; de la distribución de los números primos y de los modos de qué disponemos para su identificación; de los sistemas criptográficos más extendidos: especialmente del criptosistema de clave pública RSA; de los generadores de secuencias de bits aleatorios y de los generadores de las secuencias de bits pseudoaleatorios; de los diferentes algoritmos de factorización, especialmente de los algoritmos basados en la estrategia de FERMAT de buscar dos cuadrados congruentes con el módulo el número a factorizar; y de las características de la arquitectura de los computadores, especialmente de aquellas que más directamente influyen en la velocidad de ejecución de instrucciones.

Análisis de diferentes implementaciones disponibles para el uso y manejo de enteros de gran longitud; de los diferentes tests de primalidad, y selección del de MILLER–RABIN, que hemos considerado el mejor; de los diferentes generadores de secuencias de bits pseudoaleatorios, y selección del que hemos considerado criptográficamente más seguro: BBS; de las diferentes implementaciones y mejoras que paulatinamente han ido surgiendo para el algoritmo de factorización basado en la técnica de las fracciones continuas, de los valores de sus parámetros óptimos para su mejor rendimiento, y de las principales semejanzas entre ese algoritmo y los posteriores de Carl POMERANCE (QS) y Arjen K. LENSTRA (NFS); de las diferentes condiciones que se debe exigir al criptosistema RSA para lograr su uso alejado de ataques y trampas; y un largo proceso de análisis de la interacción entre nuestro código y nuestra máquina, buscando siempre el modo de reducir tiempos.

Diseño de un nuevo modelo de entero largo, con su definición de dominio o rango de valores posibles codificables y de sus operadores; de algoritmos varios matemáticos, criptográficos; de un generador de secuencias de bits aleatorios por entrada de teclado; de un protocolo de actuación para desarrollar con orden y sistema una tarea de optimización de código.

Implementación de todas las herramientas necesarias para que nuestro modelo de entero largo resulta operativo en todas las necesidades de cálculo (operadores a nivel de bit, relacionales, aritméticos, funciones matemáticas), del generador de secuencias de bits de aleatorios diseñado y del generador de secuencias de bits pseudoaleatorios BBS; de los algoritmos para los test de primalidad; de todos los procesos necesarios para lograr factorizar enteros largos producto de dos primos: bibliotecas de funciones que son requeridas por el algoritmo CFRAC y programas para factorizar innumerables enteros (varios millones hemos factorizado en diferentes máquinas); replica de todas las implementaciones en forma de macro para lograr programas más largos pero, sobre todo, más veloces; y también de todas las herramientas necesarias para lograr analizar la interacción entre software y hardware.

Y optimización del código estudiado, analizado, diseñado e implementado para factorizar enteros compuestos producto de dos primos grandes. Buscar con un protocolo diseñado, las formas de reducir tiempos de ejecución, analizando los tiempos, número de instrucciones, fallos de caché, instrucciones de salto. Buscar la manera de lograr obtener los factores de un entero, no sólo procurando algoritmos de menor complejidad computacional, sino también procurando implementaciones que renten al máximo las posibilidades de nuestros ordenadores actuales.

Y siempre, deslumbrados ante la esférica perfección de los números. El motor principal de nuestro estudio ha sido la contemplación de la belleza. Que nadie dude de que, en las matemáticas que hemos tenido la suerte de bucear, todo es ... ¿perfecto? ... ¡ increíble!

a

ABSTRACT

Study. Analysis. Design. Optimization. Above all: dazzling. These five words and that particular underlining attitude allow to summarizing very briefly the work done in this Thesis.

Study of: the modular arithmetic; the properties of integers numbers; the prime number distribution and the way to identify them; the most usual cryptographic systems, specially the RSA public key one; the random and pseudorandom bit sequence generators; the factoring algorithms, specially based in the FERMAT method to search two congruent square numbers with the module of the number being factorized; and finally, study of the computer architecture features most directly related to the execution time optimization.

Analysis of: the different available approaches to implement operations with very long integers; the several existing tests for prime condition of numbers, being selected the MILLER-RABIN one as the best in our opinion; the various pseudorandom bit sequence generators, being selected the BBS one as the most secure for cryptography; the alternative implementations and improvements carried out in the factoring algorithm, based in the continued fractions technique, the optimum

parameter value used for better yield and the comparisons of the implemented algorithm with those of Carl POMERANCE (QS) and Arjen K. LENSTRA (NFS); the different conditions that the RSA cryptosystem must fulfill to avoid attacks and tricks; and finally, an exhaustive analysis process regarding the code and computer interaction to achieve a significant reduction of execution time.

Design of: a new model of long integer, defining its domain, that is, its rank of the allowed values for codification and their operators; new mathematical and cryptographic algorithms; a bit sequence random generator; an orderly and systematic code optimization protocol.

Implementation of: every tool needed for optimum calculus regarding either the long integer model or the designed random bit sequence generator or the BBS pseudorandom one; the prime condition tests algorithms; the needed processes to factorize long integers, as a product of two prime numbers, such as function libraries required for the CFRAC algorithm and others programs to factorize millions of integers; macros for friendly interfaces and speedy programs; and also the implementation of any tool needed to analyze software and hardware interaction.

Finally, optimization of the code that has been studied, analyzed, designed and implemented to factorize integers obtained as the product of two large prime numbers. Following an innovated protocol in order to minimize execution time, number of instructions, cache misses and branch instructions. Our objective has always been to simplify computer algorithms and to maximize the performance of current computers.

But in any aspects, being dazzled by the rounded perfection of numbers. The major engine of this study has been beauty contemplation. Nobody can't doubt about mathematics as a whole being…perfect?…amazing!

1 INTRODUCCIÓN

El masivo almacenamiento de información en forma digital mediante soporte electrónico, unido al enorme desarrollo de las comunicaciones, ha propiciado un modo de trabajo y de vida del que difícilmente podríamos ya sustraernos. Los medios para el almacenamiento y la transmisión de información están disponibles a muchos y a un bajo coste. Podemos acceder de modo masivo e inmediato a diferentes bases de datos ubicadas en diferentes puntos del planeta. La sociedad demanda nuevos servicios de comunicación: banco por la red, e-comercio, firma digital con valor mercantil, etc.; se exige confidencialidad, integridad y accesibilidad a la información.

Existen, sin embargo, muchos problemas todavía no del todo resueltos en este universo de las comunicaciones. Existe el peligro de que nuestra información, muchas veces confidencial, pueda ser accedida por personas no deseadas: consultada, modificada, destruida; también podemos ser perjudicados de forma que se desbarate, por un agente externo, los protocolos de autorizaciones, siendo impedidos a acceder a nuestra propia información. La información en tránsito es fácilmente interceptada. Frente al deseo de confidencialidad podemos sufrir un ataque

de interceptación; frente al deseo de autenticación podemos ser suplantados; frente al deseo de integridad de nuestra información podemos sufrir modificaciones e incluso destrucción de nuestra información; frente al deseo de autenticidad podemos sufrir ataques de falsificación; ...

Las principales herramientas con las que se cuenta hoy en día para lograr una comunicación segura son la criptografía (comunicación cifrada) y la validación de usuarios. Desde luego, el estudio de los sistemas criptográficos resulta de especial interés para lograr avances en el mundo de la comunicación digital.

La **criptografía** es la ciencia y el arte de proteger y custodiar la información digital de forma segura mediante técnicas de cifrado. El **criptoanálisis** es la ciencia y el arte de obtener esa información descifrando el secreto guardado mediante las técnicas de cifrado. Ambas ciencias forman una parte del saber matemático que se llama **criptología**.

La comunicación cifrada pone en medios públicos e inseguros cualquier información que solo podrán lograr entender aquellas personas que dispongan de otra información adicional. Se basa en el recurso a lo que llamamos claves. Mediante un algoritmo de cifrado y una clave, se puede transformar una información (que llamamos plana) perfectamente legible por cualquiera en otra (llamada cifrada) imposible de interpretar. Únicamente aquellas personas que conozcan el algoritmo de cifrado y dispongan de la clave utilizada podrán volver a recuperar la información plana. Estos algoritmos que emplean la misma clave para cifrar la información plana que para obtenerla de nuevo a partir de la cifrada se llaman algoritmos criptográficos de clave simétrica.

Como señalan J. RIFÁ y Ll. HUGUET en [Rifa91], todo sistema criptográfico, o **criptosistema**, consta de cinco componentes: M, C, K, E y D. M es el conjunto de todos los **mensajes a trasmitir**; C es el de todos los **mensajes cifrados**; K es el conjunto de las claves a utilizar, y que llamaremos **espacio de claves**; E es el conjunto de todos los **métodos de cifrado**, $E=\{E_k \mid M \to C, \forall k \in K\}$ y D el de todos los **métodos de descifrado**, $D=\{D_k \mid C \to M, \forall k \in K\}$.

Cada método de cifrado E está definido mediante un algoritmo y una clave $k \in K$, que es la causante de que un mismo algoritmo defina multitud de transformaciones E_k. La misma consideración podemos hacer para las transformaciones de descifrado D_k de D. Para una clave dada k, la transformación D_k es la inversa de E_k; es decir

$D_k(E_k(m))=m$, $\forall m \in M$.

En la comunicación cifrada el conocimiento, por parte de cualquiera, de cuál algoritmo ha sido empleado no hace vulnerable el proceso de comunicación. La seguridad descansa únicamente en que nadie (excepto quienes deben conocer la información) dispone de la clave.

Todo criptosistema debe cumplir al menos los siguientes tres requisitos:

1. Todas las transformaciones de cifrado E_k y D_k han de ser fácilmente calculables.
2. Los algoritmos de las transformaciones E_k y D_k han de ser de fácil implementación.

3. La seguridad del sistema sólo debe depender del secreto de las claves $k \in K$ y no de los algoritmos de las transformaciones E y D.

Además, todo criptosistema debe tener en cuenta los objetivos de seguridad y autenticidad:

1. **Seguridad**: incapacidad, para el criptoanalista, de determinar el texto original a partir del texto cifrado que se haya podido interceptar.

   Este objetivo de seguridad exige dos requerimientos:

   a.- Desde un punto de vista computacional, un criptoanalista no ha de poder determinar la transformación de descifrado D_k a partir del mensaje cifrado c, aún conociendo el mensaje original m.

   b.- Desde un punto de vista computacional, un criptoanalista no ha de poder determinar el mensaje original, sistemáticamente, a partir de la sola interceptación del mensaje cifrado c.

2. **Autenticidad**: incapacidad, para un criptoanalista, de improvisar un texto cifrado falso c' y substituirlo en el lugar del texto cifrado c, sin que el receptor se entere.

   Este objetivo de autenticidad exige también dos requerimientos:

   a.- Desde un punto de vista computacional, un criptoanalista no ha de poder determinar la transformación E_k correspondiente a un mensaje cifrado c, aunque sea conocido el mensaje original m.

   b.- Desde un punto de vista computacional, un criptoanalista no ha de poder determinar un mensaje cifrado c' tal que $D_k(c') \in M$; es decir, un mensaje que el receptor no detecte como extraño

Es evidente que la conjunción de seguridad y la autenticidad nos exigirán la no revelación de las dos transformaciones E_k y D_k al mismo tiempo.

Actualmente se consideran dos tipos de criptosistemas, según la utilización y gerencia de las transformaciones de cifrado y descifrado:

1. El criptosistema clásico o convencional (de **clave simétrica**) en el que la clave correspondiente a ambas transformaciones es la misma. Cada pareja de interlocutores dispone de un par E_k, D_k que le son particulares, y nadie salvo ellos dos puede disponer de la información tratada por ellos.

2. El criptosistema de doble clave (de **clave pública o asimétrica**) caracterizado por el hecho de que conocer la transformación E_k no revela información alguna acerca de D_k, o viceversa.

La seguridad de un criptosistema no se basa en mantener secreto el criptosistema en sí (que generalmente es público). En los criptosistemas simétricos la seguridad se basa en mantener en

secreto el valor de la clave y en su dificultad de adivinarla. En los criptosistemas asimétricos la seguridad se basa en la complejidad de cálculo. Como señala Bruce SCHNEIER [Schn96] en la criptografía de clave pública encontramos unas funciones, que podemos llamar centrales en este arte, y que son las llamadas **funciones unidireccionales con trampa**: funciones fáciles de computar, y con existencia de su correspondiente función inversa, que requieren de un valor aleatorio llamado clave, que transforma el texto plano en texto cifrado. Funciones fáciles de invertir para quien tiene ese valor aleatorio; y prácticamente imposibles de "deshacer" sin la clave. Sorprende comprobar cómo una cantidad de información de cualquier volumen puede quedar perfectamente guardada mediante una clave que en muchos casos no supera unos pocos cientos de bits.

Por tanto todo ataque definitivo a un sistema de comunicación cifrado se basa en lograr conocer la clave con la que se está cifrando. Si un modelo criptográfico emplea un espacio de claves tal que cualquiera que disponga de medios no muy sofisticados puede adivinar la clave en un tiempo reducido, entonces ese modelo no es válido.

Un método para lograr obtener cualquier clave de un criptosistema es el conocido como **ataque por fuerza bruta**, que consiste en probar todas las claves posibles. Como ejemplo de este ataque podemos recordar los sufridos por el criptosistema DES (Data Encryption Standard) [Schn96], que era, hasta hace unos pocos años, uno de los sistemas de clave secreta más utilizados; maneja claves de 56 bits. Su espacio de claves es, por tanto

$$2^{56} = 72057\ \ 59403\ \ 79279\ \ 36 \approx 7{,}2 \times 10^{16}$$

Como recuerda Raúl DURÁN [Dura00], DES ha sufrido diferentes ataques por fuerza bruta: el más rápido y fulminante fue el ataque diseñado por EFF (Electronic Frontier Foundation), que construyó en mayo de 1998 una máquina, llamada DESCracker, que con un coste menor de 250.000 dólares fue capaz de probar todas las claves en 9 días. El tiempo medio para localizar una clave se redujo por lo tanto a tan solo 4,5 días. Actualmente se recomienda utilizar criptosistemas con espacios de claves mayores, como por ejemplo Triple DES o IDEA, de 112 y 128 bits de clave respectivamente (cfr. [Schn96] ó [Mene97]).

Una vez disponemos de un criptosistema de cifrado y descifrado suficientemente probado y conocido, con un espacio de claves que hagan al sistema invulnerable ante un ataque por fuerza bruta, surge un nuevo problema: ¿cómo lograr poner de acuerdo a dos interlocutores, que desean utilizar un canal público y poco seguro, sobre qué clave van a utilizar para realizar la comunicación? Desde luego no pueden emplear el canal inseguro para ponerse de acuerdo en este "nada pequeño detalle". Hay que tener además en cuenta la gran cantidad de claves que resultan necesarias para establecer un sistema de comunicación cifrado con criptografía simétrica: si contamos con n personas que desean comunicarse entre sí se requiere el uso de $n \times (n-1)/2$ claves: muchas son las claves que hay que lograr distribuir y renovar en un entramado

de comunicación cifrada.

DIFFIE y HELLMAN trabajaron en busca de un proceso matemático que permitiese a dos interlocutores ponerse de acuerdo en la clave simétrica que iban a emplear. En 1976 presentaron la criptografía de clave pública [Diff76], y un modelo que llevaba a la práctica este nuevo concepto de criptografía. El algoritmo que presentaron permitía el intercambio de una clave secreta en un canal inseguro o público. La matemática que sustentaba su algoritmo era la aritmética modular, y la operación básica la exponenciación.

Dos años más tarde del trabajo de DIFFIE y HELLMAN, los investigadores RIVEST, SHAMIR y ADLEMAN presentaron un algoritmo de clave pública que ha hecho fortuna: se trata del criptosistema que lleva como nombre las tres iniciales de estos tres autores: RSA [Rive78].

El algoritmo de clave pública o de criptografía asimétrica requiere que cada potencial receptor de información cifrada disponga de dos claves: una puesta a disposición de cualquiera que quiera cifrar un mensaje para remitirlo al receptor (la clave pública), y otra celosamente guardada por cada receptor (la clave secreta o privada), que le permite descifrar cualquier información cifrada con la clave pública a la que complementa. Las claves pública y privada de cada individuo que desee comunicarse de forma cifrada con este modelo son diferentes y complementarias.

El sistema de cifrado del criptosistema RSA está basado en la exponenciación modular. Sus dos claves, pública y privada, son pares $(\exp, \mathrm{mod})$, consistentes en un exponente $\exp$ y un módulo mod, donde mod es el producto de dos primos de cierta longitud. En ambas claves, el valor de mod es el mismo. Y la única forma conocida de calcular el exponente de la clave secreta (la llamaremos (d,n)) a partir del exponente de la clave pública (la llamaremos (e,n)) es conociendo los primos largos p y q que multiplicados forman el valor del módulo n.

Se sabe, por tanto, que factorizar el módulo RSA permite romper RSA. No está tan claro, sin embargo, si existe otro método capaz de romper RSA, sin necesidad de factorizar n.

1.1. EL RETO DE LA FACTORIZACIÓN DE ENTEROS

Como señala Carl POMERANCE [Pome96] cuando a mediados del siglo XX se potenció la investigación en problemas computacionales (propiciado en gran medida por el Conflicto bélico mundial), el problema de la factorización de enteros grandes fue ignorado: se consideraba una cuestión trivial. ¿Qué interés podría tener su estudio y discusión? ¿Qué sentido tenía investigar modos de hallar los primos que factorizan un compuesto si esa respuesta llegaría sola al aumentar el poder de computación de nuestras máquinas? Sólo unos pocos ignoraron la moda de su tiempo y continuaron el estudio y búsqueda de métodos y caminos más rápidos para encontrar los factores de un número.

En las últimas décadas hemos visto la llegada del poder de computación, que cada vez se ha hecho más accesible y más rápido. Y nos encontramos con que ese crecimiento en el poder de cómputo no ha resuelto el problema de la factorización; ha dejado en evidencia que el reto matemático de la factorización requiere estudio e investigación.

Y hemos visto también el crecimiento de los sistemas criptográficos, que basan su seguridad precisamente en la actual inhabilidad para factorizar con rapidez enteros de tamaños grandes. Actualmente son muchos los investigadores interesados en el problema de la factorización: no ya solamente porque este problema es punto de referencia para la seguridad de los sistemas criptográficos, sino porque es referencia y reto para la computación misma.

A lo largo del desarrollo de este trabajo presentaremos algunos de los algoritmos más conocidos y comúnmente empleados en los sistemas de cálculo simbólico que podemos encontrar en el mercado o en libre distribución por la red de Internet. Cada uno de ellos tiene su rango de valores donde su ejecución resulta perfectamente eficaz; pero llegamos rápidamente a un rango a partir del cual todos nuestros intentos son infructuosos. El entero más grande, producto de dos primos, del que sepamos que se ha logrado obtener su factorización, es el presentado en [Cava00]. El proyecto completo para obtener los factores de un compuesto de 155 dígitos decimales (512 bits) requirió de 7 meses de trabajo e intervinieron más de 300 PC, una Computadora SGI Origin 2000 y un Superordenador Cray C916. El algoritmo que se utilizó en la resolución del reto presentado por RSA Laboratories fue el de la criba de campo numérico (NFS), completamente desarrollado en [Lens93].

Este algoritmo, y el de la criba cuadrática (QS ó MPQS) presentado por Carl POMERANCE y largamente recogido en [Lens87] se basan en una idea inicial, compartida también por el algoritmo de las de factorización mediante la técnica de las fracciones continuas [Morr75]. Los tres algoritmos se basan en una idea presentada por el matemático FERMAT. Más adelante reseñaremos cada uno de estos algoritmos. La complejidad matemática de los algoritmos QS y NFS es alta, y su implementación ardua.

El problema de la factorización de los enteros es un problema abierto. Y como señala Carl POMERANCE en [Pome96], su solución llegará con el entendimiento y avance de las investigaciones en las dos ciencias implicadas: por una parte la teórica y rigurosa matemática; de otra la experimental ciencia de la computación e implementación de los algoritmos.

1.2. COMPLEJIDAD DE LOS ALGORITMOS DE FACTORIZACIÓN

El análisis del coste de computación de un algoritmo puede realizarse mediante el estudio de la cantidad de recursos (tiempo de ejecución y ocupación de memoria) necesarios para que el algoritmo se ejecute y termine dando el resultado buscado. En nuestro trabajo nos hemos centrado principalmente en el estudio de los tiempos de ejecución. No hemos hecho ningún

intento de reducir la ocupación en memoria, puesto que el rango de tamaños en el que hemos trabajado no ha requerido esa optimización; sí hemos procurado, como se verá, optimizar el acceso a los distintos niveles de memoria.

A la vista del progresivo incremento de la potencia de cálculo de los ordenadores, cabe preguntarse si vale la pena hacer un estudio de optimización de código destinado a mejorar la eficiencia de una determinada aplicación, o es más sencillo esperar la llegada de un ordenador mucho más rápido. Como veremos a continuación, un incremento en el orden de magnitud de la potencia del ordenador puede tener una repercusión insignificante en la reducción de recursos empleados por un determinado algoritmo. Y como se verá a lo largo del desarrollo de esta tesis, una modificación en la concepción del algoritmo o en su implementación puede ocasionar un incremento importante de velocidad.

Una vez tenemos especificado un algoritmo para una operación, la información que nos indica su eficiencia es el tiempo requerido para que el algoritmo se ejecute completamente. Un modo de medir este valor, de una forma que resulte independiente del ordenador utilizado y que sea expresión únicamente dependiente de la propia definición del algoritmo, es tomar la medida del número de operaciones a nivel de bit. Cuando calculamos el número de operaciones a nivel de bit necesarias para la ejecución de un algoritmo lo que obtenemos es la complejidad lineal de este algoritmo. El procedimiento teórico para el estudio de la complejidad lineal se realiza trabajando con órdenes de magnitud en ese número de operaciones a nivel de bit. Habitualmente medimos la complejidad de un algoritmo en lo que llamamos su comportamiento asintótico: el comportamiento que tiene cuando el tamaño de la entrada crece indefinidamente.

1.2.1. La notación "big – o".

Una notación matemática bastante extendida para representar en órdenes de magnitud esta complejidad lineal de un determinado algoritmo en su comportamiento asintótico es la llamada **O–notación**: supongamos que para todo $n \geq n_0$ las dos funciones, $f(n)$ y $g(n)$ están definidas, toman valores positivos y, para un valor constante c, se cumple que $f(n) < c \cdot g(n)$; entonces se dice que $f = \mathrm{O}(g)$ [Kobl98].

En la práctica, se toma una función $g(n)$ más sencilla que $f(n)$, pero que ofrezca una buena aproximación o idea del comportamiento de $f(n)$. Para ejemplificar esta notación podemos considerar el siguiente caso (tomado de [Rose93]): supongamos que para llevar a cabo una determinada operación sobre una entrada de datos de tamaño n se requieren como mucho $f(n) = n^3 + 8 \cdot n^2 \cdot \log n$ operaciones a nivel de bit. Dado que $8 \cdot n^2 \cdot \log n < 8 \cdot n^3$ para cualquier entero positivo, podemos decir que nuestro proceso requiere una máximo de $8 \cdot n^3$ operaciones a nivel de bit para cada tamaño de entrada de datos n. En este caso, por tanto, con la O–notación, diremos que esta operación requiere $O(n^3)$ operaciones a nivel de bit (tenemos que

$g(n) = n^3$.

Si $f(n) = \mathrm{O}(n)$, podemos suponer que doblando el valor de n se dobla el valor de $f(n)$.

Si $f(n) = \mathrm{O}(n^2)$, podemos suponer que si n se dobla, entonces $f(n)$ crece en un factor de 4.

Si $f(n) = \mathrm{O}(n^3)$, podemos suponer que si n se dobla, entonces $f(n)$ crece en un factor de 8.

Si $f(n) = \mathrm{O}(2^n)$, podemos suponer que si n se incrementa en 1, entonces $f(n)$ se dobla.

El comportamiento asintótico de un algoritmo y su valor recogido mediante la O–notación determina, en última instancia, si un algoritmo es factible o no factible, y de alguna manera indica si el problema que pretende resolver el algoritmo es tratable o no.

Cuando ocurre que $f(n)$ es mucho menor que $g(n)$ para valores de n grandes, se dice que $f = o(g)$, con la letra o minúscula (es la llamada notación **little–o**).

1.2.2. De los tiempos polinómicos a los tiempos exponenciales.

Neal KOBLITZ ofrece en [Kobl98] una definición pareja a la recogida antes, pero que nos permitirá trabajar mejor en nuestras clasificaciones de algoritmos: KOBLITZ define la función L_n, como

$$L_n(g;c) = O(\exp((c + O(1)) \cdot (\ln n)^g \cdot (\ln \ln n)^{1-g}))$$

para valores reales de g, c y n, y con $n \to \infty$.

Decimos que un algoritmo es **un algoritmo** $L(g)$ cuando aplicado a un entero n, tiene un tiempo estimado de ejecución de la forma $L_n(g;c)$, para algún c.

En particular, que

$$L_n(1;c) = O(\exp(c \cdot \ln n)) = O(n^c)$$

y que

$$L_n(0;c) = O(\exp(c \cdot \ln \ln n)) = O((\ln n)^c)$$

Y llamamos **algoritmo de tiempo polinomial** a un algoritmo $L(0)$, y llamamos **algoritmo de tiempo exponencial** a un algoritmo $L(1)$. Entendemos por **algoritmo de tiempo subexponencial** a un algoritmo $L(g)$, para algún $g < 1$. Como señala KOBLITZ, el valor del parámetro g mide el camino que existe entre el tiempo polinómico y el tiempo exponencial.

No se conoce ningún algoritmo de factorización que realice la operación en un tiempo polinomial. Algunos de ellos son incluso exponenciales, como por ejemplo el algoritmo de factorización por divisiones sucesivas (algoritmo $L(1)$).

Los algoritmos más modernos de factorización, como así los cataloga Henri COHEN en [Cohe93] al compararlos con los que él llama algoritmos de la "Edad Oscura", tienen unos valores de $g < 1$: a esos algoritmos los llamamos algoritmos de factorización subexponenciales.

Para el trabajo de nuestra tesis nos hemos centrado en uno de los algoritmos de factorización subexponenciales: el algoritmo de las fracciones continuas (CFRAC). Su comportamiento queda definido con los parámetros de la función $L_n(g;c)$ de valores $c=\sqrt{2}+O(1)$ y $g=1/2$. Es decir, es un algoritmo $L(1/2)$.

Los métodos QS ó MPQS y NFS aventajan en velocidad al algoritmo CFRAC. Como podemos ver en [Stin00], los tiempos asintóticos para los algoritmos de factorización posteriores al CFRAC son:

QS: Su comportamiento queda definido con los parámetros de la función $L_n(g;c)$ de valores $c=1+O(1)$ y $g=1/2$. Es decir, es un algoritmo $L(1/2)$.

NFS: Su comportamiento queda definido con los parámetros de la función $L_n(g;c)$ de valores $c=1.923+O(1)$ y $g=1/3$. Es decir, es un algoritmo $L(1/3)$.

De estos valores, se deduce que cada algoritmo presentado supera en velocidad a anterior. De todas formas, se debe tener en cuenta que estas expresiones reflejan un comportamiento asintótico y, como recuerdan los autores de [Denn93], hay tamaños de enteros (rango entre 100 y 150 dígitos) donde QS o CFRAC pueden resultar más rápido que NFS.

Esta notación consigue clasificar muy bien a los algoritmos que están cerca de ser polinómicos. Si se quiere hacer una equivalencia entre la O–notación y la función $L_n(g;c)$, diremos que una función de tiempo subexponencial es aquella cuyo comportamiento asintótico es de la forma $e^{f(n)}$, donde $f(n)=o(n)$.

1.3. MEJORAS EN EL RENDIMIENTO DEL CÓDIGO. OPTIMIZACIÓN.

Hemos hablado de las medidas teóricas de la complejidad de nuestros algoritmos de factorización. Disponemos también de herramientas que nos permiten medir el número exacto de instrucciones ejecutadas en un determinado proceso. También podemos obtener el número de ciclos que ha empleado un proceso hasta llegar a su término.

Entendemos por **productividad** la cantidad de trabajo hecho en un cierto tiempo. La productividad la medimos con el parámetro rendimiento. Principalmente, cuando hablamos del **rendimiento** de un procesador nos ocupamos del tiempo de respuesta o tiempo de ejecución. El tiempo es la medida del rendimiento del ordenador.

PATTERSON y HENNESSY, en [Patt00], traen una ecuación del rendimiento, que recoge la medida del tiempo de ejecución: Según señalan ambos autores, el tiempo de ejecución de una aplicación depende de tres factores: El número de instrucciones que se ejecutan (NI); el promedio de ciclos empleados en cada instrucción (CPI); el tiempo de ciclo (que depende de la frecuencia de reloj del ordenador: T_{ciclo}).

La expresión que relaciona los tres parámetros es:

$$T_{ejecución} = NI \times CPI \times T_{ciclo}$$

Optimizar un código consiste en modificarlo para lograr que realice la misma tarea en un tiempo menor. En el proceso de optimización debemos tener en cuenta la ley de AMDHAL: cuando realizamos una mejora en una función dentro de la aplicación, el peso de esa mejora será proporcional al peso que tenía esa función antes de ser optimizada: no compensa mejorar lo que apenas tiene peso en un proceso.

Las dos vías de mejora que hemos seguido a lo largo de nuestra investigación para lograr la optimización de nuestro código son: análisis del software del algoritmo en C (mejora por software); y análisis de la interacción entre el código compilado y la máquina (mejora por hardware).

El primero de los dos pasos consiste en realizar búsqueda de algoritmos mejores que los utilizados en los procesos que deseamos optimizar: podemos encontrar algoritmos que realicen el mismo proceso en un tiempo menor.

El segundo de los pasos consiste en mejorar el código teniendo en cuenta el hardware de la máquina. Para este proceso disponemos de una herramienta llamada RABBIT [Rabbit]. RABBIT es una herramienta que permite leer y manipular los contadores de eventos hardware de los procesadores Intel o AMD en los programas escritos en C bajo el sistema operativo Linux. Mucha es la información que hemos podido obtener de nuestro código gracias a esta herramienta, como podrá verse a lo largo de este trabajo.

Gracias a ambas técnicas (mejoras de software y mejoras de hardware) se han logrado avances a veces espectaculares en los tiempos de ejecución en las distintas funciones que intervienen en el proceso de factorización. Más adelante, en el Capítulo 7, se describe detalladamente todo el proceso seguido.

1.4. OBJETIVOS DE LA TESIS

Las páginas que preceden encuadran el ámbito en el que hemos desarrollado nuestro trabajo. Los objetivos específicos de esta tesis son los siguientes:

1. Determinar cuáles son, e implementar, todas las funciones necesarias para crear un entorno de trabajo apropiado en la aritmética modular de enteros grandes. Definir e implementar un tipo de dato para los enteros grandes: su dominio o rango de valores posibles y sus operadores básicos: aritméticos, relacionales y a nivel de bit. Seleccionar los algoritmos que mejor resuelvan una serie de funciones matemáticas básicas, como la exponenciación, el cálculo de la raíz cuadrada de un entero grande, tests de primalidad, etc. Analizar otras implementaciones y comparar las distintas formas de definir y codificar un entero largo.

2. Estudiar y elegir los algoritmos para la generación de secuencias aleatorias y pseudoaleatorias de valor criptográfico. Implementar esos algoritmos; especialmente hemos trabajado con el generador de secuencias pseudoaleatorias definido por L. BLUM, M. BLUM y M. SHUB en [Blum86] comúnmente llamado generador BBS. Se ha desarrollado una implementación de este generador que origine órbitas de periodo máximo, tal y como lo describen Luis HERNÁNDEZ et al. en [Hern98] y en [Alva98]. También hemos diseñado e implementado un generador de bits aleatorios por entrada de teclado, que ha sido probado mediante todos los tests estadísticos sugeridos en [Mene97] y el test universal de MAURER.

3. Implementar el algoritmo de factorización basado en el desarrollo por fracciones continuas [Morr75], incluyendo todas las sucesivas mejoras posteriormente introducidas.

4. Analizar los tiempos de ejecución de las diferentes funciones. Determinar sobre qué funciones se va a realizar el análisis en función de su peso relativo en el total de la aplicación diseñada, de acuerdo con la ley de AMDHAL. Determinar también sobre qué parámetros se va a realizar la optimización dirigida a reducir tiempos. Modificar de forma paulatina cada una de las funciones que intervienen en el proceso de factorización. Lograr reducir al máximo los tiempos de ejecución del algoritmo de factorización CFRAC. Como ha quedado antes explicado, esta optimización logrará reducir el factor c del comportamiento asintótico antes expresado por las función $L(g;c)$, que en el caso de nuestro algoritmo decíamos que era igual a $c=\sqrt{2}+O(1)$ y $g=1/2$. Como señala Carl POMERANCE en [Pome87] los distintos autores no alcanzaban un acuerdo sobre el valor de c para CFRAC. POMERANCE señala que según algunos está en $c=\sqrt{2}+O(1)$, según otros en $c=2+O(1)$ y según un tercer grupo en $c=\sqrt{3/2}+O(1)$. Nosotros hemos tomado $c=\sqrt{2}+O(1)$ que es el que toma como mejor este autor citado. Pero queda claro que, al margen del valor que tiene g, que resulta ser el mismo para CFRAC y para QS, el reto de optimización del algoritmo de las fracciones continuas está en reducir el valor de c.

1.5. BREVE RESUMEN DEL DESARROLLO DE ESTA TESIS.

En este primera Capítulo hemos presentado el contexto en el cual queda enmarcada la tesis, y hemos dejado definidos los objetivos principales de nuestro trabajo.

En el Capítulo 2 recogemos, en tres presentaciones, algunos conceptos básicos y fundamentales en el ámbito de la criptografía en el que ha quedado desarrollada la tesis. En primer lugar, tratamos sobre las condiciones para la existencia de inversos en la aritmética modular: desde un punto de vista intuitivo, o descriptivo y visual, y mediante un desarrollo más analítico del álgebra en la aritmética modular. Esa primera parte finaliza con una descripción del algoritmo RSA. Una segunda parte está dedicada a los números primos: a su localización y a su reconocimiento. En esta parte procuramos ofrecer una visión general básica de la matemática que se requiere para

desarrollar los tests de primalidad. Presentamos una justificación de por qué hemos elegido finalmente el test de MILLER–RABIN. Una tercera parte de este segundo Capítulo presenta una panorámica (estado del arte) de los algoritmos de factorización, y nos centramos especialmente en el desarrollo de los algoritmos subexponenciales: CFRAC, QS y NFS.

En el Capítulo 3 presentamos una breve descripción del proceso de optimización de código. Presentamos una definición de tiempo de ejecución y mostramos diferentes vías de mejora que se pueden aplicar al objetivo de reducir ese tiempo. Mostramos las principales causas de posibles retardos en la ejecución: riesgos estructurales, de control y de datos. Y mostramos una breve descripción del funcionamiento de la memoria y de las principales técnicas que hemos utilizado para reducir los retardos y optimizar su acceso.

En el Capítulo 4 presentamos un modelo de entero largo, que hemos definido e implementado para trabajar con valores enteros de gran magnitud. En primer lugar mostramos la descripción de nuestro modelo: fundamentos matemáticos y la implementación de nuestro tipo de dato nuevo creado; también mostramos otros modelos de enteros ya existentes y disponibles. En un segundo epígrafe mostramos los operadores que hemos necesitado implementar para los procesos matemáticos de generación de aleatorios, tests de primalidad y factorización: operadores auxiliares, relacionales y a nivel de bit; y operadores aritméticos: suma, resta, producto, cociente y módulo. También hemos necesitado implementar algunas funciones sencillas como son el algoritmo de EUCLIDES para el cálculo del máximo común divisor de dos enteros; el cálculo de la parte entera de la raíz cuadrada de un entero, necesario para todo el proceso de factorización mediante la técnica de las fracciones continuas; y la potencia. Finalmente presentamos algunas implementaciones de tests de primalidad.

El Capítulo 5 queda dedicado a la presentación y descripción de los generadores de bits aleatorios. En un primer epígrafe se describe el generador de bits aleatorios por entrada de teclado, y se muestran los resultados de los diferentes tests estadísticos a que hemos sometido al generador. En un segundo epígrafe presentamos nuestra implementación del generador de secuencias de bits pseudoaleatorias basado en el algoritmo BBS. La implementación de ambos generadores queda justificada por sí misma, por su interés criptográfico; y además hemos desarrollado estos generadores por la necesidad de disponer de abundantes bits aleatorios que nos permitieran generar grandes cantidades de enteros, compuestos de dos primos aleatoriamente buscados. Al final del Capítulo recogemos unas tablas con enteros primos característicos del generador BBS, de diferentes tamaños, y señalamos también el tiempo que hemos necesitado para encontrar cada uno de esos números.

El Capítulo 6 está enteramente dedicado al algoritmo de factorización basado en el método de las fracciones continuas. En un primer epígrafe hacemos una presentación general del algoritmo, desde sus principios más básicos, y vamos presentando las diferentes mejoras que fueron surgiendo a lo largo de los años 70, destinadas todas ellas a reducir los tiempos de ejecución. En

un segundo epígrafe recogemos los diferentes pasos que sigue nuestra implementación del algoritmo de factorización basado en la técnica de las fracciones continuas, y mostramos luego en una breve descripción, las funciones creadas para cada uno de esos pasos del proceso. También recogemos las estructuras de datos creadas y una relación de los parámetros que modifican el comportamiento del algoritmo. Estos parámetros pueden y deben ser estudiados en cada implementación que se realiza: en nuestro caso el proceso de factorización ha sido ejecutado repetidas veces, para distintos tamaños y diferentes valores de los parámetros; al final del capítulo traemos una expresión que muestra la relación entre el tamaño de los compuestos a factorizar y el valor óptimo de los parámetros.

El Capítulo 7 queda dedicado a la descripción del proceso de optimización. Como en ese Capítulo se describe, el código que hemos optimizado es el que se emplea en la factorización, con los algoritmos implementados y ya presentados en el Capítulo 5. La herramienta que hemos empleado en el desarrollo de nuestra trabajo es la llamada RABBIT (cfr. [Rabbit]). Primeramente recogemos un breve elenco de programas y aplicaciones, disponibles en el mercado o de libre distribución, con un también breve comentario sobre los algoritmos que emplean para la tarea de la factorización de enteros largos. A continuación, y después de una breve descripción del protocolo de actuación que hemos seguido, el Capítulo se desarrolla mostrando los avances más significativos obtenidos en cada uno de los pasos del protocolo: cambio de algoritmo; reducción de Instrucciones; técnicas para reducir instrucciones de salto, especialmente mediante el desenrollado de bucles; evitar las dependencias de datos; y optimizar los accesos a memoria. Recogemos, en un siguiente epígrafe, algunas gráficas que muestran las optimizaciones logradas en las principales funciones sobres las que hemos trabajado. Al final del Capítulo mostramos una gráfica comparativa de los tiempos empleados en la factorización de diferentes enteros compuestos, producto de dos primos de similar tamaño, realizada con nuestra aplicación antes y después de optimizar algunas de las funciones.

Por último, se recoge un breve Capítulo (el octavo) dedicado a las conclusiones de la tesis y a recoger diferentes trabajos que nos parecen de interés y en los pensamos que podríamos seguir investigando, tanto en la implementación como en la optimización de algunos algoritmos criptográficos, y en la definición del concepto de entero largo o de precisión múltiple.

1.6. GUÍA PARA LA LECTURA DE LA TESIS

Para terminar, hacemos un breve comentario al porqué de estos capítulos, y traemos una recomendación para su lectura. En el Capítulo 2 se leen los conceptos básicos de las matemáticas para la criptografía que hemos visto conveniente que quedasen recopilados en nuestra tesis. Recoge el esfuerzo que hemos realizado en el estudio y comprensión de esos conceptos. Son una redacción ordenada de conceptos que pueden hallarse en muchos de los

libros que vienen relacionados en la bibliografía. Pero nos parece que constituye una buena guía para quien no conoce esos conceptos. En su redacción hemos querido reflejar el itinerario que hemos seguido nosotros para lograr manejar con soltura esa matemática. Quien tenga conocimientos de la matemática de la aritmética modular y del criptosistema RSA puede evitar su lectura, sin perder continuidad el resto de la redacción. Para quien no los tenga, puede resultar una guía didáctica asequible: especialmente en sus dos primeros apartados.

El Capítulo 3 es otro breve resumen conceptual. También puede omitir su lectura aquel que haya trabajado en arquitectura de ordenadores. No pretende ser un resumen de conceptos de arquitectura, sino una recopilación de los posibles cuellos de botella a las que hemos hecho frente cuando hemos querido optimizar el código que hemos implementado. Este Capítulo puede ser un prólogo del Capítulo 7.

Los Capítulos 4 y 6 recogen nuestra implementación de entero largo y del algoritmo de factorización basado en la técnica de las fracciones continuas. Su lectura termina también enfocada al Capítulo 7, dedicado precisamente a optimizar todo ese código.

El Capítulo 5 es algo independiente, y surgió de una necesidad de nuestro trabajo. Era preciso poder dar valor a los muchos enteros que debíamos manejar para probar nuestras implementaciones de los operadores para nuestra definición de entero largo; más adelante fue necesario generar una gran cantidad de compuestos, producto de dos primos, de diferentes tamaños, para someter a pruebas y estudio nuestro algoritmo de factorización. Y poco a poco esta necesidad nos fue metiendo en el diseño del generador que allí presentamos y en la implementación del probado BBS.

Quizá quien contemple en índice de esta Tesis pueda no encontrar una única unidad temática. El hilo conductor de este trabajo se encuentra en su mismo desarrollo. Hemos tratado diferentes campos de la criptografía. Hemos hecho una optimización de unas determinadas funciones. Queda mucho campo de trabajo en cada uno de los asuntos tratados. Pensamos que esta Tesis nos ha dejado muchas puertas abiertas para posibles trabajos futuros.

2 ALGUNAS PROPIEDADES CRIPTOGRÁFICAS DE LOS NÚMEROS PRIMOS

El estudio y conocimiento de los fundamentos matemáticos de la criptografía ha adquirido un interés práctico que se le añade al intrínseco interés teórico del que gozan todas las matemáticas. La criptografía sobre la que hemos trabajado en esta tesis descansa en gran manera en la aritmética modular. En estos últimos años este campo de las matemáticas ha aumentado enormemente su interés.

Estas matemáticas alcanzaron un gran desarrollo teórico de la mano de matemáticos de los siglos XVII a XIX. Matemáticos que trabajaron sin los medios actuales de computación; y que alcanzaron un magnífico nivel de conocimientos: FERMAT, GAUSS, EULER, GALOIS, JACOBI, LEGENDRE,...

Gran parte de la criptografía descansa sobre estos matemáticos y su trabajo. Por citar algunos ejemplos podemos referirnos a la generación de valores pseudoaleatorios; la determinación de si un entero dado es primo compuesto (por ejemplo, con el algoritmo de MILLER–RABIN, que se fundamenta en el teorema de FERMAT, y con el método de SOLOVAY–STRASSEN que descansa en

el criterio de EULER y en la posibilidad del cálculo del símbolo de JACOBI); el criptosistema RSA (que define sus claves apoyado en expresiones de EULER y FERMAT: función de EULER, teorema de EULER, extensión del teorema pequeño de FERMAT, algoritmo extendido de EUCLIDES); los procesos de factorización de enteros largos, especialmente los subexponenciales...

Entre los muchos capítulos que abre la aritmética modular, nos centraremos especialmente, en esta presentación, en un secreto que esconden los enteros. Un secreto del que ya hemos hablado en el capítulo anterior, que sorprende por su sencillez, por su naturaleza cotidiana: la camuflada existencia de los números primos, su nada sencilla identificación, sus sorprendentes propiedades,... ¿Por qué los números esconden a sus primos? ¿Cómo se esconden esos primos? ¿Por qué no podemos disponer de una herramienta que nos permita identificarlos de forma sencilla y asequible mediante un proceso de cálculo no demasiado complejo ni gravoso? ¡ Son tan distintos de todos los demás!: Los números primos son las piezas elementales del "mecano" de los enteros. Kenneth ROSEN [Rose93] los considera como los bloques sobre los que se construye cualquier entero.

El concepto de **número primo** es universalmente conocido: dados dos enteros a y b, decimos que a divide a b (o que b es divisible por a) y se escribe $a|b$, si existe un entero c tal que $b = a \times c$. Se dice entonces que a es **divisor** de b o que a es **factor** de b. Un divisor de n se dice que es **divisor propio** si es distinto de n y distinto de 1. Un número será primo si es un entero mayor que 1 y no tiene divisores propios. Cuando el número tiene divisores propios se llama compuesto.

La identificación de los números primos constituye un problema paralelo al de la factorización de los enteros. Dos cuestiones podemos formular: la posibilidad de encontrar respuesta a cualquiera de ellas nos ofrecería la clave para descubrir la solución de la otra:

1. Dado un número: ¿cómo saber que si es primo?
2. Dado un compuesto: ¿cómo saber qué primos lo forman?

Tres características queremos destacar, sobre el conjunto de los primos y sobre el problema de la factorización:

1. El conjunto de los primos es infinito. No podemos definir el conjunto por extensión, mostrando su lista completa; necesitamos buscar una definición por sus propiedades. Y, efectivamente, como veremos, disponemos de esta descripción del conjunto de los primos y conocemos propiedades que los hacen identificables: pero la verificación práctica de esas propiedades resulta con frecuencia una tarea muy costosa en tiempo de computación.
2. No existe limitación en el número de veces que un primo puede intervenir en la factorización de un compuesto.
3. Tampoco existe limitación en el número de primos que pueden intervenir en la factorización de un compuesto.

Señalamos tres motivos que nos impulsan a presentar un esbozo teórico sobre los números primos,

y que justifican la redacción del presente capítulo.

1. La condición de primalidad ofrece la posibilidad de existencia de inversos para cualquier entero en la aritmética de módulo un primo: veremos, por ejemplo, que este problema de la existencia de inversos y de las propiedades de los primos en la aritmética modular es de gran importancia para el funcionamiento del criptosistema RSA.
2. Necesitamos conocer qué medios tenemos para detectar la presencia de un primo: el hecho de disponer de formas que nos permitan determinar la condición de primalidad de cualquier entero es fundamental en la criptografía, puesto que muchos procesos criptográficos deben su correcto funcionamiento precisamente a que trabajan con primos. De nuevo podemos considerar cómo el criptosistema RSA requiere, para su puesta en marcha, de la generación de dos primos que, además, verifiquen que el valor que resulta de restar (y también sumar) 1 a dicho primo y dividir por 2 ese resultado vuelva a ser un primo (estándar X.509). También se requiere la generación de enteros primos grandes en otros criptosistemas como el generador BBS [Blum96], que además exige fuertes condiciones al primo a utilizar (cfr. [Hern96] y [Alva98]).
3. Sería de enorme utilidad disponer de un modo sencillo de obtener los factores primos de un compuesto: la factorización, aparte de su indudable interés matemático, es la vía más directa para deshacer la eficacia de muchos criptosistemas, entre ellos en RSA.

Hemos centrado el estudio realizado y presentado en esta tesis en el conocimiento de algunos de los criptosistemas ya citados: RSA y BBS. Y como ya ha quedado dicho, ambos criptosistemas deben su seguridad a la dificultad en factorizar enteros producto de primos de longitud grande. Hemos centrado todo el trabajo de optimización de código en la implementación de un algoritmo de factorización.

En el presente capítulo presentamos tres epígrafes:

1. El primero dedicado a algunas propiedades de las que gozan los primos en la aritmética modular. En especial nos centramos en la posibilidad de disponer de valores inversos. A la vista de esas propiedades, presentamos un primer y breve esbozo del criptosistema RSA.
2. El segundo presenta una breve introducción sobre la distribución de los números primos en el conjunto de los enteros y, después de recoger algunas nociones matemáticas necesarias, como el concepto de residuo cuadrático o la definición de los símbolos de LEGENDRE y JACOBI, mostramos algunos algoritmos, basados en la aritmética modular, que se emplean para determinar si un determinado entero es primo o no.
3. El tercer epígrafe muestra una descripción breve de diferentes algoritmos (los más conocidos e implementados) de factorización. También aquí mostramos otras características exigidas al criptosistema RSA.

2.1. EXISTENCIA DE INVERSOS

2.1.1. Nociones Básicas

Vamos a presentar y a resolver una cuestión realmente útil en la aritmética modular; un cálculo repetidamente realizado en muchos diseños de herramientas criptográficas. Y, entre otras, en la puesta en marcha —como veremos— del algoritmo RSA.

2.1.1.1. Existencia de inversos y números primos.

17	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
0									0								
1		**1**	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
2		2	4	6	8	10	12	14	16	**1**	3	5	7	9	11	13	15
3		3	6	9	12	15	**1**	4	7	10	13	16	2	5	8	11	14
4		4	8	12	16	3	7	11	15	2	6	10	14	**1**	5	9	13
5		5	10	15	3	8	13	**1**	6	11	16	4	9	14	2	7	12
6		6	12	**1**	7	13	2	8	14	3	9	15	4	10	16	5	11
7		7	14	4	11	**1**	8	15	5	12	2	9	16	6	13	3	10
8		8	16	7	15	6	14	5	13	4	12	3	11	2	10	**1**	9
9	0	9	**1**	10	2	11	3	12	4	13	5	14	6	15	7	16	8
10		10	3	13	6	16	9	2	12	5	15	8	**1**	11	4	14	7
11		11	5	16	10	4	15	9	3	14	8	2	13	7	**1**	12	6
12		12	7	2	14	9	4	16	11	6	**1**	13	8	3	15	10	5
13		13	9	5	**1**	14	10	6	2	15	11	7	3	16	12	8	4
14		14	11	8	5	2	16	13	10	7	4	**1**	15	12	9	6	3
15		15	13	11	9	7	5	3	**1**	16	14	12	10	8	6	4	2
16		16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	**1**

Cuadro 1: Productos en aritmética modular, módulo 17. Quedan señalados en rojo aquellas posiciones cuyo producto es igual al elemento neutro.

¿Cuándo podemos decir que es posible encontrar un inverso en aritmética modular: un valor que multiplicado por otro, y reducido el producto al módulo o cardinal del conjunto, dé como resultado el valor 1? La respuesta a esta pregunta la presentamos en algunas formulaciones quizá poco académicas pero, creemos, muy intuitivas.

Llamamos $\mathbb{Z}_n$ al **conjunto finito de enteros menores que** n, sobre los que definimos las operaciones internas suma y producto. Por ejemplo, el conjunto $\mathbb{Z}_n$ para $n=17$ está formado por

los enteros $Z_{17} = \{0,1,2,3,...,15,16\}$. La operación suma no ofrece nunca problema alguno. La suma de cualquiera dos elementos de este conjunto es otro elemento del mismo conjunto. Se realiza mediante reducción modular. Pero... *¿y* el producto?

Analicemos la tabla del producto de este conjunto. En el Cuadro 1 vemos los resultados de multiplicar (producto con reducción modular) todos los elementos de Z_{17} entre sí. (Observación: en este cuadro 1, y en los sucesivos cuadros, hay resultados que resultan el mismo para todos los elementos de una fila o columna: en esos casos, hemos colocado ese valor abarcando toda la fila o columna).

15	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
0								0							
1		1	2	3	4	5	6	7	8	9	10	11	12	13	14
2		2	4	6	8	10	12	14	1	3	5	7	9	11	13
3		3	6	9	12	0	3	6	9	12	0	3	6	9	12
4		4	8	12	1	5	9	13	2	6	10	14	3	7	11
5		5	10	0	5	10	0	5	10	0	5	10	0	5	10
6		6	12	3	9	0	6	12	3	9	0	6	12	3	9
7		7	14	6	13	5	12	4	11	3	10	2	9	1	8
8	0	8	1	9	2	10	3	11	4	12	5	13	6	14	7
9		9	3	12	6	0	9	3	12	6	0	9	3	12	6
10		10	5	0	10	5	0	10	5	0	10	5	0	10	5
11		11	7	3	14	10	6	2	13	9	5	1	12	8	4
12		12	9	6	3	0	12	9	6	3	0	12	9	6	3
13		13	11	9	7	5	3	1	14	12	10	8	6	4	2
14		14	13	12	11	10	9	8	7	6	5	4	3	2	1

Cuadro 2: Productos en aritmética modular, módulo 15. Quedan señalados en rojo aquellas posiciones cuyo producto es igual al elemento neutro. En color azul aquellas posiciones cuyo producto es igual a cero.

Observamos que la disposición de los valores de los productos es simétrica con respecto a la bisectriz que discurre desde la esquina superior izquierda hasta la inferior derecha (constatación de que este producto es conmutativo). También observamos la existencia de un valor 1 en cada fila y en cada columna (y solo uno en cada fila y columna).

Veamos ahora el siguiente conjunto, similar al anterior, que presentamos en el Cuadro 2, y que recoge los productos para el conjunto definido con $n = 15$: Z_{15}. Ahora el comportamiento de los productos ha sufrido un cambio no pequeño. Vemos que hay filas y columnas donde tenemos un 1 y sólo uno. Pero en otras no tenemos ninguno. Además, observamos que en aquellas filas y

columnas donde no hay un 1 aparece (en ocasiones varias veces) un valor en principio sorprendente: el 0. Por ejemplo, $9 \cdot 5 = 0$.

A los valores a, b de $\mathbb{Z}_n$ que verifican que $a \neq 0$, $b \neq 0$, y $a \cdot b = 0$ los llamamos **divisores de cero**. Un divisor de cero es un entero no invertible, es decir, no existe ningún elemento de $\mathbb{Z}_n$ que multiplicado por el divisor de cero dé el elemento neutro o elemento unidad. Así lo podemos cotejar en el cuadro.

Los números 3, 5, 6, 9, 10 y 12 son divisores de cero. Todos ellos tienen una propiedad común: ninguno de ellos es coprimo con el valor de n: el máximo común divisor de cualquiera de ellos con el 15 es distinto de 1. Los demás números (1, 2, 4, 7, 8, 11, 13 y 14) no son divisores de cero. Todos ellos tienen elemento inverso: aquel que al multiplicarlo por el número ofrece como resultado el neutro 1. Todos ellos son coprimos con el valor de n. Decimos que $r \in \mathbb{Z}_n$ es un **elemento unitario** si $\exists s \in \mathbb{Z}_n$ tal que $r \cdot s = 1$, es decir, si tiene inverso. Está demostrado (y así lo podemos verificar en el ejemplo de $\mathbb{Z}_{15}$) que $r \in \mathbb{Z}_n$ es un elemento unitario o una unidad si y solo si $\text{mcd}(r, n) = 1$.

Hay otra propiedad que podemos observar de nuestro Cuadro 2: todos los valores de los productos que aparecen en una fila o columna de un divisor de cero son divisores de cero. Y los productos de los elemento unitarios toman todos los valores de $\mathbb{Z}_{15}$. ¿Qué ocurre entonces si eliminamos del cuadro de productos de $\mathbb{Z}_{15}$ las filas y las columnas de los divisores de cero... El resultado de esa operación puede verse en el Cuadro 3.

15	1	2	4	7	8	11	13	14
1	1	2	4	7	8	11	13	14
2	2	4	8	14	1	7	11	13
4	4	8	1	13	2	14	7	11
7	7	14	13	4	11	2	1	8
8	8	1	2	11	4	13	14	7
11	11	7	14	2	13	1	8	4
13	13	11	7	1	14	8	4	2
14	14	13	11	8	7	4	2	1

Cuadro 3: Productos del conjunto reducido de residuos módulo 15.

Como vemos queda un conjunto cuya operación producto resulta ser interna. Y además tenemos –de nuevo– un elemento neutro en cada fila y en cada columna. Este conjunto, que llamaremos $\mathbb{Z}_{15}^*$, tiene ahora la estructura algebraica de grupo abeliano. A este conjunto le llamamos **conjunto reducido de residuos**. Este nuevo conjunto $\mathbb{Z}_n^*$ tendrá estructura de grupo para el producto, pero evidentemente en él la operación suma ya no es interna.

Podemos enunciar ahora el **teorema general de la existencia de inversos**:

> Dado un entero $a \in \mathbf{Z}_n^*$ (es decir, $\text{mcd}(a,n)=1$), entonces existe un único valor $b \in \mathbf{Z}_n^*$ tal que $a \cdot b = 1$.

Y también podemos introducir un nuevo concepto: llamamos **orden de un grupo** al número de elementos del grupo. Evidentemente nos referimos al número de elemento unitarios (si consideramos el conjunto con divisores de cero no podemos hablar de un conjunto con estructura algebraica de grupo para la operación producto).

Ya sabemos la condición necesaria para que un entero, en un conjunto finito de enteros, y bajo la operación producto con reducción modular, pueda tener un inverso.

2.1.1.2. Función de EULER y otras nociones matemáticas.

Otras muchas preguntas podemos hacernos todavía. Por ejemplo, ya sabemos que un número $a \in \mathbf{Z}_n^*$ tiene inverso. Pero... ¿cómo calcularlo? Para responder a esta pregunta disponemos de la **función de EULER** Φ, que recoge el cardinal del conjunto reducido de residuos; es decir, el número de elementos de $\mathbf{Z}_n$ que no son divisores de cero; es decir, el número de elementos de $\mathbf{Z}_n$ que son coprimos con n; es decir, el orden del grupo $\mathbf{Z}_n^*$. Veamos cómo se define la función de EULER:

1. Si p es primo, entonces $\Phi(p) = p - 1$.
2. Si p es primo y $n = p^k$, entonces $\Phi(n) = p^k - p^{k-1}$.
3. Si m y n son coprimos entre sí, entonces $\Phi(m \cdot n) = \Phi(m) \cdot \Phi(n)$.
4. Si $n = \prod_{i=1}^{k} p_i^{e_i}$, entonces $\Phi(n) = \prod_{i=1}^{k} p_i^{e_i - 1} \cdot (p_i - 1)$.

El **teorema de EULER** demuestra que si $\text{mcd}(b,n)=1$, entonces $b^{\Phi(n)} \equiv 1 (\text{mod}\, n)$. Por tanto el teorema nos ofrece una vía de encontrar los inversos de un elemento unitario: $b^{-1} \equiv b^{\Phi(n)-1} (\text{mod}\, n)$.

Vemos ahora el Cuadro 4 de potencias (para el caso $\mathbf{Z}_{13}$, 13 es primo). De la observación de este cuadro señalamos lo siguiente:

1. Un elemento $g \in \mathbf{Z}_n^*$ primo se llama **Generador** si todo elemento de $\mathbf{Z}_n^*$ se puede expresar como potencias de g: por ejemplo, en $\mathbf{Z}_{13}$ los números 2, 6, 7 y 11.
2. Las potencias de cualquier $g \in \mathbf{Z}_n^*$ deben repetirse a partir de un determinado exponente. Para todo $g \in \mathbf{Z}_n^*$ existe un a mínimo tal que $g^a = 1$. A este valor de a le llamamos **orden del elemento** (que evidentemente es un concepto distinto al de orden de un grupo definido antes): por ejemplo, en $\mathbf{Z}_{13}$ el orden del número 5 es 4.
3. A un elemento lo llamamos **primitivo** si su orden es máximo, es decir si su orden es $p-1$. Se deduce de modo inmediato que solo tenemos primitivos si el módulo que define el conjunto finito es un entero primo y la definición coincide entonces con la de generador.
4. El **teorema de LAGRANGE** (cfr. [Fust00]) demuestra que todos los órdenes de los elementos de

un conjunto son divisores del orden del grupo. Vemos en el conjunto del Cuadro 4 que todos los órdenes son 2, 3, 4, 6 y 12. Y esos son, precisamente, los divisores de $p-1$. Gracias a esta propiedad que enuncia el teorema de LAGRANGE ocurre que para el valor del exponente $p-1$ todas las potencias alcanzan el valor 1. Como veremos, este es el enunciado del teorema de FERMAT.

13	0	1	2	3	4	5	6	7	8	9	10	11	12
0							0						
1		1	1	1	1	1	1	1	1	1	1	1	1
2		2	4	8	3	6	12	11	9	5	10	7	1
3		3	9	1	3	9	1	3	9	1	3	9	1
4		4	3	12	9	10	1	4	3	12	9	10	1
5		5	12	8	1	5	12	8	1	5	12	8	1
6	1	6	10	8	9	2	12	7	3	5	4	11	1
7		7	10	5	9	11	12	6	3	8	4	2	1
8		8	12	5	1	8	12	5	1	8	12	5	1
9		9	3	1	9	3	1	9	3	1	9	3	1
10		10	9	12	3	4	1	10	9	12	3	4	1
11		11	4	5	3	7	12	2	9	8	10	6	1
12		12	1	12	1	12	1	12	1	12	1	12	1

Cuadro 4: Potencias módulo 13. La primera columna recoge las bases. La primera fila los exponentes. Quedan señalados en verde las primeras potencias iguales a uno.

En el Cuadro 5 vemos que al ser n un valor compuesto no tenemos ningún primitivo ni ningún generador. Algunos valores x (3, 5, 6, 9, 10 y 12) nunca alcanzan a tener una potencia de valor 1 (no son coprimos con n y si tuviéramos un exponente $k<n$ que diera $x^k=1$ inmediatamente podríamos deducir que teníamos inversa para x en el resultado de x^{k-1}). Vemos además que todos los valores coprimos con n verifican el teorema de EULER: $b^{\Phi(n)} \equiv 1 (\mathrm{mod}\, n)$.

Una consideración nueva hemos de introducir si trabajamos con el conjunto, por ejemplo, de $n=16$. En ese caso, como vemos en el Cuadro 6, tenemos valores para los que, a partir de un determinado exponente, alcanza el valor cero y, por tanto, todas las sucesivas potencias siguen siendo cero. Estos valores los llamamos nilpotentes. Decimos que $a \in \mathbf{Z}_n$ es **nilpotente** si $\exists k \in N$, $k \neq 0$, tal que $a^k = 0$. Todo elemento nilpotente es divisor de cero, puesto que si $a^k = 0$, es inmediato que $a \cdot a^{k-1} = 0$. También es inmediato que el valor de la potencia a^{k-1} es un divisor de cero: podemos cotejarlo en todos los valores que aparecen en el Cuadro 6.

Otra observación, esta vez del Cuadro 4, (anteriormente mostrado) podemos hacer sobre una

propiedad que verifican los números primos. Esta propiedad, consecuencia directa del teorema de EULER ($b^{\Phi(n)} \equiv 1 (\mathrm{mod}\, n)$), es que la potencia $n-1$ de todos los elementos del conjunto es igual a 1: $b^{n-1} \equiv 1 (\mathrm{mod}\, n)$ (**teorema de FERMAT**). Si esto se verifica para todos los valores b menores que n, entonces podemos afirmar que n es un número primo. En el Cuadro 6 vemos que ningún valor del conjunto verifica que $b^{n-1} \equiv 1 (\mathrm{mod}\, n)$. En el Cuadro 5 vemos algunos valores que sí verifican esa propiedad, pero otros (la mayoría) no. Sólo puede cumplirse esta propiedad en aquellos valores que verifiquen que $\mathrm{mcd}(b, n) = 1$.

15	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
0								0							
1		1	1	1	1	1	1	1	1	1	1	1	1	1	1
2		2	4	8	1	2	4	8	1	2	4	8	1	2	4
3		3	9	12	6	3	9	12	6	3	9	12	6	3	9
4		4	1	4	1	4	1	4	1	4	1	4	1	4	1
5		5	10	5	10	5	10	5	10	5	10	5	10	5	10
6		6	6	6	6	6	6	6	6	6	6	6	6	6	6
7		7	4	13	1	7	4	13	1	7	4	13	1	7	4
8	1	8	4	2	1	8	4	2	1	8	4	2	1	8	4
9		9	6	9	6	9	6	9	6	9	6	9	6	9	6
10		10	10	10	10	10	10	10	10	10	10	10	10	10	10
11		11	1	11	1	11	1	11	1	11	1	11	1	11	1
12		12	9	3	6	12	9	3	6	12	9	3	6	12	9
13		13	4	7	1	13	4	7	1	13	4	7	1	13	4
14		14	1	14	1	14	1	14	1	14	1	14	1	14	1

Cuadro 5: Potencias módulo 15. La primera columna recoge las bases. La primera fila los exponentes. Quedan señalados en rojo las primeras potencias iguales a uno. En azul aquellas posiciones cuya potencia es igual a uno para el exponente igual al valor de la función de EULER.

Para concluir este breve repaso sobre algunas cuestiones matemáticas de la aritmética modular, señalamos que el conjunto finito $\mathbf{Z}_n$ tendrá la estructura algebraica de cuerpo si n es primo; si n es compuesto, su estructura algebraica es la de anillo abeliano.

2.1.2. Un desarrollo algebraico sobre la aritmética modular.

Ya hemos "visto" el comportamiento de los conjuntos finitos de enteros con las operaciones producto reducido en aritmética modular. Presentamos ahora brevemente un desarrollo más algebraico.

16	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0								0								
1		1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
2		2	4	8	0	0	0	0	0	0	0	0	0	0	0	0
3		3	9	11	1	3	9	11	1	3	9	11	1	3	9	11
4		4	0	0	0	0	0	0	0	0	0	0	0	0	0	0
5		5	9	13	1	5	9	13	1	5	9	13	1	5	9	13
6		6	4	8	0	0	0	0	0	0	0	0	0	0	0	0
7		7	1	7	1	7	1	7	1	7	1	7	1	7	1	7
8	1	8	0	0	0	0	0	0	0	0	0	0	0	0	0	0
9		9	1	9	1	9	1	9	1	9	1	9	1	9	1	9
10		10	4	8	0	0	0	0	0	0	0	0	0	0	0	0
11		11	9	3	1	11	9	3	1	11	9	3	1	11	9	3
12		12	0	0	0	0	0	0	0	0	0	0	0	0	0	0
13		13	9	5	1	13	9	5	1	13	9	5	1	13	9	5
14		14	4	8	0	0	0	0	0	0	0	0	0	0	0	0
15		15	1	15	1	15	1	15	1	15	1	15	1	15	1	15

Cuadro 6: Potencias módulo 16. La primera columna recoge las bases. La primera fila los exponentes. Quedan señalados en rojo las primeras potencias iguales a uno. En azul las potencias iguales a cero: corresponden a bases nilpotentes.

Definimos una **relación**, que llamamos **de congruencia**, y que es una relación de equivalencia en los enteros. Dado el entero n, decimos que a es congruente con b módulo n si y solo si $a-b=k\cdot n$. Se dice entonces que $a\equiv b(\mathrm{mod}\,n)$. Esta relación de equivalencia define n clases en Z: $[a]=\{m\in \mathrm{Z}/\, m\equiv a(\mathrm{mod}\,n)\}$. Al conjunto de las n clases de equivalencia lo llamamos $\mathrm{Z}/n\mathrm{Z}$, ó Z_n. Es un conjunto con estructura de anillo. Sus operaciones son módulo n y sus elementos pueden ser representados siempre como el conjunto de todos los enteros menores que n [Cohe93]

Llamamos **conjunto completo de residuos** $\mathrm{mod}\,n$ a un conjunto de n elementos $\{r_1, r_2, ..., r_n\}$ tal que, dado cualquier entero a, existe un único r_i del conjunto que cumple que $a\equiv r_i\,(\mathrm{mod}\,n)$. Este conjunto está formado por un representante de cada una de las clases que se definen con la relación de congruencia.

Las propiedades de este conjunto de relaciones o de residuos $\mathrm{Z}/n\mathrm{Z}$ han quedado "vistas" en la presentación previa mediante los diferentes cuadros que hemos ido mostrando. Podemos revisarlas ahora desde este ángulo menos intuitivo y más algebraico.

Definimos las operaciones básicas en una aritmética que se define sobre esta relación de equivalencia:

Si a, b, c, d y n son enteros tales que $n>0$, y se cumple que $a\equiv b(\mathrm{mod}\,n)$, y $c\equiv d(\mathrm{mod}\,n)$, entonces $a\pm c\equiv b\pm d(\mathrm{mod}\,n)$, $a\cdot c\equiv b\cdot d(\mathrm{mod}\,n)$. Además,

$$(a+b)\bmod n=(a\bmod n+b\bmod n)\bmod n$$

$$(a-b)\bmod n=(a\bmod n-b\bmod n)\bmod n$$

$$(a\cdot b)\bmod n=(a\bmod n\cdot b\bmod n)\bmod n$$

$$a\cdot(b+c)\bmod n=(a\cdot b\bmod n+a\cdot c\bmod n)\bmod n$$

La operación cociente tiene su peculiaridad que hay que tener en cuenta:

Si a, b, c y n son enteros tales que $n>0$, si $d=\mathrm{mcd}(c,n)$, si $a\cdot c\equiv b\cdot c(\mathrm{mod}\,n)$, entonces $a\equiv b(\mathrm{mod}\,n/d)$. Por tanto, si c es tal que $\mathrm{mcd}(c,n)=1$, entonces si $a\cdot c\equiv b\cdot c(\mathrm{mod}\,n)$ también es verdad que $a\equiv b(\mathrm{mod}\,n)$.

Con todas estas definiciones de las operaciones, y una vez conocidas las reglas de esta relación de equivalencia que hemos llamado de congruencia nos planteamos de nuevo la pregunta antes presentada y resuelta mediante los cuadros numéricos: ¿en qué ocasiones podemos hablar de existencia de elemento inverso para la operación producto modular?

Sean a, b y n enteros tales que $n>0$ y $d=\mathrm{mcd}(a,n)$.

1. Si d no divide a b ($d\nmid b$), entonces la ecuación $a\cdot x\equiv b(\mathrm{mod}\,n)$ no tiene solución: No hay inverso.
2. Si d divide a b ($d\mid b$), entonces la ecuación $a\cdot x\equiv b(\mathrm{mod}\,n)$ tiene exactamente d soluciones incongruentes entre sí.
3. Si $d=1$ entonces la expresión tiene solución y ésta es única. A la solución de la expresión $a\cdot x\equiv 1(\mathrm{mod}\,n)$ se la llama inversa de a.

Es inmediato descubrir la relación entre esta formulación algebraica y las ideas presentadas mediante los cuadros numéricos. En el Cuadro 1 veíamos la tabla de productos del conjunto de enteros menores que 17. Como 17 es primo entonces $\mathrm{mcd}(a,17)=1$ para cualquier $0\le a<17$. Por tanto, cualquier expresión de la forma $a\cdot x\equiv b(\mathrm{mod}17)$ (donde $b\in(0,16)$) tiene una solución única, y única es la solución en cada caso cuando $b=1$: en ese caso la solución x se trata del valor inverso de a. Son los casos señalados en rojo en el Cuadro 1. Además comprendemos que todos los valores del intervalo $[0,16]$ son incongruentes entre sí módulo 17 y este intervalo es, de hecho, un conjunto completo de residuos.

En el Cuadro 2 (en este caso el valor del módulo era el 15) encontramos los divisores de cero: son aquellos valores $a\in[0,14]$ que verifican que $\mathrm{mcd}(a,15)=d\neq 1$. Tenemos por tanto que cualquier

expresión de la forma $a \cdot x \equiv b(\mathrm{mod}\, n)$ sólo tendrá solución (no única) si $d \mid b$. Por eso podemos ver en el Cuadro 2 que las filas correspondientes a los números múltiplos de tres, únicamente tienen estos valores múltiplos de tres (que es el máximo común divisor de 15 con cualquiera de estos cuatro valores); y que las filas correspondientes a los números múltiplos de cinco, únicamente tienen estos valores múltiplos de cinco (que es el máximo común divisor de 15 con cualquiera de estos dos valores). Esos valores... ¡ y CEROS!

Esos números forman un subconjunto de elementos que definen sus propias congruencias:

1. sólo si a y b pertenecen al conjunto $\{0,3,6,9,12\}$, la expresión $a \cdot x \equiv b(\mathrm{mod}\, n)$ tiene solución (no única);
2. sólo si a y b pertenecen al conjunto $\{0,5,10\}$, la expresión $a \cdot x \equiv b(\mathrm{mod}\, n)$ tiene solución (no única); y
3. sólo si a y b NO pertenecen a ninguno de estos dos conjuntos (es decir, pertenecen al conjunto $\{1,2,4,7,8,11,13,14\}$) la expresión $a \cdot x \equiv b(\mathrm{mod}\, n)$ tiene solución (única en ese caso).

En este último caso además se da la posibilidad de hablar de inversos, pues tenemos al elemento neutro en el conjunto de valores posibles.

Por eso se puede trabajar con cada uno de estos subconjuntos y definir en ellos sus propias operaciones internas. Y así en el Cuadro 3 se ve el conjunto que hemos llamado conjunto reducido de residuos, formado por aquellos elementos a del conjunto completo de residuos que verifican que $\mathrm{mcd}(a,15)=1$. Este conjunto tiene la propiedad de que todos sus elementos tienen inversa.

De todo lo dicho hasta ahora podemos sacar una importante conclusión que es todo un enunciado:

> Dado el conjunto Z de los enteros (que tiene estructura algebraica de anillo euclideo), y dado $p \in \mathbf{Z}$ primo, entonces el conjunto $\mathbf{Z}/p\,\mathbf{Z}$ (que podemos definirlo por extensión tomando un representante de cada clase de equivalencia, y en concreto podemos tomar los valores comprendidos entre 0 y $p-1$) es un conjunto formado por p clases de equivalencia y que tiene estructura de cuerpo abeliano. Todo elemento $a \in [1, p-1]$ tiene inverso. El conjunto $\mathbf{Z}/p\,\mathbf{Z}$, con las operaciones suma y producto, con el 0 para el neutro de la suma y el 1 para el neutro del producto, verifica que cada elemento distinto de cero tiene un único inverso para el producto y además se cumplen las leyes conmutativa, asociativa y distributiva. Estos cuerpos son conocidos como **cuerpos de GALOIS** [Mora94].

Los cuerpos $\mathbf{Z}/p\,\mathbf{Z}$ (ó $\mathbf{Z}_p$) desempeñan un papel fundamental en la estructura de los cuerpos finitos. Está demostrado, (cfr. [Fust00]), que el número de elementos de cualquier cuerpo finito K

debe ser igual a la potencia de un número primo p; es decir, $\#K = p^m$. Al primo p se le llama **característica del cuerpo** K, que se puede representar como $K = GF(p^m)$ (GF: GALOIS Field). Y además también está demostrado que sólo existe un cuerpo finito con p^m elementos.

Todo este repaso a diferentes conceptos de aritmética modular no es más que un escueto esbozo. De todo lo dicho se deducen importantes propiedades matemáticas. Es fácil comprender el interés que presenta el número primo y sus posibilidades operatorias al disfrutar de las propiedades descritas. Se comprende el interés que levanta cualquier camino que facilite su reconocimiento dentro del conjunto de los enteros.

Como recapitulación de esta primer apartado señalamos que la condición de número primo es siempre señal de existencia de inversos: Si trabajamos en una aritmética modular en la que el módulo p es un primo, entonces todos los elementos de ese conjunto numérico tienen inverso y estamos trabajando en una conjunto con estructura de cuerpo abeliano. Decimos que son cuerpos de GALOIS. Si trabajamos en una aritmética con módulo n compuesto el conjunto tiene estructura de anillo abeliano y entonces al menos podemos asegurar que todo número coprimo con n tiene inverso. Cuanto menos factores primos tenga el compuesto y cuanto mayor sea el menor de los factores, menos números no coprimos con el módulo encontraremos, y por tanto más números del conjunto de los enteros menores que el módulo tendrán inverso en la aritmética modular módulo n.

2.1.3. Cálculo del inverso cuando desconocemos los factores del módulo.

Pero... ¿qué ocurre si el módulo n sobre el que trabajamos es un entero compuesto del que desconocemos sus factores primos? En ese caso no podemos acudir al teorema de FERMAT para calcular el inverso, pues el módulo no es primo. Pero tampoco podemos acudir al teorema de EULER, pues al desconocer la factorización del módulo no podemos obtener el valor de la función de EULER Φ.

Para llegar al cálculo del inverso de un entero en una aritmética de módulo un compuesto del que desconocemos sus factores, acudimos al algoritmo extendido de EUCLIDES. El **algoritmo de EUCLIDES** muestra el modo de obtener el máximo común divisor de dos o más enteros. Afirma que dados a y b, con $a \geq b$, $mcd(a,b) = mcd(b, a \bmod b)$ si $a \bmod b \neq 0$; y $\mathrm{mcd}(a,b) = b$ si $a \bmod b = 0$.

El **algoritmo extendido de EUCLIDES** es un método para calcular inversos. Sea a coprimo con n. (Si $\mathrm{mcd}(a,n) \neq 1$ entonces no cabe buscar un inverso que ya sabemos que no existe). Buscamos un entero b tal que $1 \equiv a \cdot b (\mathrm{mod}\, n)$. El proceso de búsqueda de inverso mediante el algoritmo de EUCLIDES consiste en buscar el valor de $\mathrm{mcd}(a,n)$, que evidentemente será 1. El proceso comienza con los valores $x_0 = n$, $x_1 = a$ y va generando sucesivamente valores

$$x_{i+1} \equiv x_{i-1} (\mathrm{mod}\, x_i) \quad \text{y} \quad q_i = x_{i-1} / x_i$$

hasta que lleguemos a un valor $x_{k+1} = 0$. El valor previo (x_k) será igual a 1, pues ese debe ser el máximo común divisor entre a y n.

En el proceso descrito vamos generando varias series finitas de la forma que presentamos a continuación:

valores iniciales: $s_0 = 1$, $s_1 = 0$, $t_0 = 1$, $t_1 = 0$

generación de las secuencias: $s_j = s_{j-2} - q_{j-1} \cdot s_{j-1}$, $t_j = t_{j-2} - q_{j-1} \cdot t_{j-1}$

relación entre ambas secuencias: $x_j = s_j \cdot n + t_j \cdot a$.

Vamos calculando las dos series t_k y s_k hasta llegar al valor $x_k = 1$. Entonces tendremos la expresión

$$1 = s_k \cdot n + t_k \cdot a$$

y aplicando a ambos lados de la expresión el operador módulo n tendremos que $1 \equiv 0 + t_k \cdot a (\mathrm{mod}\, n)$: es decir, t_k es el inverso de a.

Un ejemplo de este proceso viene recogido en el artículo antes citado de RIVEST, SHAMIR y ADLEMAN [Rive78], en su epígrafe VII.D. Lo recogemos a continuación, por su interés histórico y también para clarificar este método que es tan usado para la generación de claves del criptosistema RSA. El ejemplo calcula el inverso del valor `a = 157` módulo el valor `N = 2668`. Como se ve en él, se procede al cálculo de las dos series s_i y t_i y se llega al valor `t3 = 17`, que resulta ser el inverso módulo 2668 del valor `a = 157`.

```
N = 2668;          a = 157. Ambos valores son coprimos.
```

Las serie xi y qi son las siguientes:

```
x0 = 2668
x1 =  157          q1 =  16
x2 =  156          q2 =   1
x3 =    1          q3 = 156
x4 =    0               FIN
```

Las series si y ti son las siguientes:

s0 = 1 s1 = 0 t0 = 0 t1 = 1

s2 = s0 - q1 · s1, es decir, s2 = 1 –16 · 0 **= +1**

t2 = t0 – q1 · t1, es decir, t2 = 0 –16 · 1 **= -16**

s3 = s1 – q2 · s2, es decir, s3 = 0 – 1 · 1 **= -1**

t3 = t1 – q2 · t2, es decir, t3 = 1 – 1 ·(-16) **= +17**

Aplicamos los resultados:

x3 = s3 · N + t3 · a, es decir, 1 = (-1)·2668 + 17·157. Y aplicando módulo 2668 a ambos lados de la ecuación...

1 = 1 modulo 2668 = 17·157 modulo 2668, es decir:

$1 \equiv 17 \cdot 157 \ (\mathrm{mod}\ 2668) \Rightarrow$ **$17 \circ 157^{-1}$ (mod 2668)**

2.1.4. Algunas consideraciones sobre el criptosistema RSA.

Sobre estas propiedades descansa el algoritmo introducido por RIVEST, SHAMIR y ADLEMAN [Rive78] que se conoce como RSA. RSA necesita, para la generación de sus claves, del cálculo de inversos en aritmética modular. El criptosistema RSA se basa en la operación de exponenciación en aritmética modular.

Sea $m \in \{0, n-1\}$. Entonces la función de cifrado es $c = m^e \bmod n$, donde e y n constituyen los valores de la clave pública de cifrado. La operación de descifrado es $m = c^d \bmod n$, donde d (junto con los factores de n) son las claves privadas del criptosistema; d y n son las claves del descifrado.

Estas transformaciones se basan en el teorema de EULER ($b^{-1} \equiv b^{\Phi(n)-1} (\bmod n)$). Basta exigir que $e \cdot d \equiv 1 (\bmod \Phi(n))$ para que la función de descifrado sea la inversa de la de cifrado. Esta simetría otorga a estos métodos de cifrado la posibilidad de autentificar los mensajes enviados, mediante el procedimiento que ha venido en llamarse de firma electrónica o firma digital.

El **protocolo del RSA** es el siguiente:

1. Cada usuario U elige dos números primos, p y q, y calcula $n = p \cdot q$. El grupo a utilizar será $\mathbf{Z}_n^*$. El orden del grupo será $\Phi(n) = \Phi(p \cdot q) = (p-1) \cdot (q-1)$.

   Cualquiera que desconozca la factorización de n tendrá serias dificultades en calcular el valor de la función $\Phi(n)$. Como ya dijimos al comienzo de este capítulo, la seguridad de este criptosistema descansa precisamente en el desconocimiento de estos factores p y q.

2. U selecciona un entero positivo e de forma que $1 \le e \le \Phi(n)$ y de forma que sea coprimo con el orden del grupo: $mcd(\Phi(n), e) = 1$.

3. Se calcula el inverso de e en $\mathbf{Z}_n^*$, que será d. Tendremos que $e \cdot d \equiv 1 (\bmod \Phi(n))$, con $1 \le d \le \Phi(n)$ y d también coprimo con $\Phi(n)$. No se busca el valor d de manera que resulte el inverso de e en el conjunto $\mathbf{Z}_n$, sino en el conjunto $\mathbf{Z}_{\Phi(n)}$. Conocemos el valor de la función de EULER $\Phi(n)$ gracias a que conocemos los factores de n. Pero no conocemos el valor de $\Phi(\Phi(n))$, que es el necesario para poder calcular d mediante el teorema de EULER. Como ha quedado explicado, para calcular el inverso de e módulo $\Phi(n)$ utilizamos el algoritmo extendido de EUCLIDES.

4. La **clave pública** del usuario será (n, e). La **clave privada** será (n, d). Por supuesto, deben permanecer secretos los números p, q y, especialmente, $\Phi(n)$.

Las operaciones de cifrado y descifrado son

Cifrado: $c = E_e(m) = m^e \bmod n$

Descifrado: $m = D_d(c) = c^d \bmod n$

El mensaje m se obtiene asociando, a cada carácter del alfabeto en que está escrito el mensaje,

un valor numérico. Se tiene así un mensaje m a cifrar de una longitud indefinida y, en principio, grande. Este mensaje m se divide en bloques m_i numéricamente menores que n.

El mensaje cifrado, c, tendrá un tamaño similar en cada uno de sus bloques c_i. Cada bloque se cifra haciendo $c_i = m_i^e \bmod n$ y se descifra haciendo $m_i = c_i^d \bmod n$.

El motivo de que la exponenciación del mensaje cifrado con d sea la operación inversa a la exponenciación del mensaje plano con e queda manifiesto en el modo en que han sido definido los dos exponentes:

$$c_i^d = (m_i^e)^d = m_i^{e \cdot d} = m_i \cdot m_i^{e \cdot d - 1} = m_i \cdot m_i^{k \cdot \Phi(n)} \equiv m_i \pmod n = m_i$$

Para ello hemos tenido en cuenta que:

1. $e \cdot d \equiv 1 \pmod{\Phi(n)}$.
2. Teorema de EULER: $m_i^{\Phi(n)} \bmod n = 1$
3. Hemos supuesto que $mcd(m_i, n) = 1$. De otra manera el teorema de EULER no se cumpliría. La probabilidad de que no se cumpla esta condición es ínfima dado el modo en que ha quedado definido n: producto de dos primos de gran tamaño.

Llegar a tomar un valor de m_i tal que $mcd(m_i, n) \neq 1$ sería tanto como haber logrado al azar encontrar un factor propio de n. No se puede, por tanto, tomar esta suposición o exigencia como una limitación al criptosistema. Además, en el caso de que el azar nos trajera un valor m_i no coprimo con n ocurriría que ese bloque m_i quedaría cifrado con un valor c_i que, posteriormente, no podríamos quizá descifrar; pero puesto que un atacante no conocería el valor inicial m_i, y tampoco conocería que el destinatario no ha podido descifrar uno de los bloques, ese improbabilísimo problema no atenta a la seguridad de RSA.

RSA fundamenta su seguridad en la dificultad de factorizar números grandes. Sus dos claves (la pública y la privada) son función de un número largo producto de dos primos: $n = p \cdot q$. Cualquiera que pueda conocer el valor de $\Phi(n)$ podrá obtener, mediante el algoritmo extendido de EUCLIDES, el valor de la clave privada d a partir de la clave pública e.

Los procedimientos de la clave pública tienen muchas ventajas, como por ejemplo la posibilidad de generar firma digital. Tiene, sin embargo, dos principales inconvenientes si los comparamos con los sistemas simétricos: la ratio de velocidad del cifrado de clave pública respecto a la privada es, en el mejor de los casos, de $1/100$; además está el problema de la identificación: lograr la certeza de que la clave pública a la que libremente accedemos pertenece verdaderamente a la persona a quien queremos enviar una comunicación cifrada, y que no sufrimos un ataque por suplantación de personalidad. Para afrontar el primer problema se usa en la práctica el cifrado asimétrico sólo para cifrar pequeños mensajes: habitualmente claves que servirán posteriormente para una comunicación bajo cifrado simétrico. Para el problema de la identificación se han creado en la red los "centros de validación" o autoridades de certificación [Müll98].

Existen muchas exigencias sobre los primos que forman el módulo de RSA. Algunas de ellas las veremos al presentar, más adelante en este capítulo, las técnicas de factorización más empleadas. Son exigencias que surgen de la necesidad de defender el módulo del criptosistema RSA de determinados ataques por intento de factorización.

Señalamos aquí una condición, que se exige habitualmente a los primos que forman el módulo, para lograr que el exponente de la clave pública (el exponente e) tenga el valor $e=3$. Ya ha quedado dicho que cualquier valor del exponente que cumpla la condición $mcd(\Phi(n),e)=1$, será válido para nuestros propósitos. Pero es evidente que si el exponente es el más pequeño posible, el proceso de cifrado será el más rápido posible también. Por eso es una costumbre bastante extendida tomar ese valor para el exponente e. Y para lograr que $mcd(\Phi(n),3)=1$ basta con exigir que ambos primos verifiquen la condición $p \equiv 2(\mathrm{mod}\,3)$ y $q \equiv 2(\mathrm{mod}\,3)$ (cfr. [Denn93]).

2.2. LOCALIZACIÓN Y DISTRIBUCIÓN DE LOS NÚMEROS PRIMOS.

En el epígrafe anterior hemos pretendido explicar la importancia que tienen los números primos para la criptografía. Pero no es su interés sólo un fenómeno actual. Los números primos han tenido fascinados a los matemáticos desde los tiempos de EUCLIDES. Muchas preguntas han alimentado el afán de investigación de grandes matemáticos. ¿Cuántos primos hay? ¿Cómo saber si un entero escogido al azar es primo o no? ¿Cómo se distribuyen los números primos dentro del conjunto de los enteros, o dentro de los intervalos? ¿Existen funciones que definan a los números primos? ¿Podemos definir diferentes familias de números primos dentro del conjunto total de todos ellos? [Ribe91].

El avance de los computadores electrónicos desde la Segunda Guerra Mundial ha permitido encarar el problema de cómo averiguar de forma eficiente la condición de primalidad de un entero cualquiera. En años recientes, los números primos y los tests de primalidad han alcanzado gran importancia para la construcción de generadores de secuencias pseudoaleatorias seguros y la generación de claves en los criptosistemas de clave pública [Kran86].

En este epígrafe pretendemos dar respuesta a la siguiente cuestión: Dado un entero, ¿cómo saber si es o no es primo? En busca de una respuesta a esa pregunta, lo primero que tenemos es el siguiente teorema [Bres89]:

Si p es un primo impar, entonces $2^{p-1} \equiv 1(\mathrm{mod}\ p)$

Este teorema nos ofrece una primera orientación sobre el modo de detectar la condición de primalidad para un entero dado. De todas formas, la condición enunciada por el teorema es necesaria, pero no suficiente. Existen enteros impares n que verifican que $2^{n-1} \equiv 1(\mathrm{mod}\,n)$ y que, sin embargo, son compuestos. Por ejemplo, $2^{341-1} \equiv 1(\mathrm{mod}\,341)$ siendo que $341 = 11 \times 31$.

A estos compuestos los llamamos **pseudoprimos para la base *2***. La cantidad de pseudoprimos para la base 2 es muy inferior a la de primos. Por ejemplo, hay tres menores que mil: 341, 561 y 645. Y hay sólo 245 menores de 1 millón (en ese mismo rango de enteros contamos con 78.498 primos) [Bres89].

El teorema enunciado es, en realidad, un caso particular del ya enunciado teorema de FERMAT, que dice que si p es un entero impar primo, entonces para cualquier base b que verifique que $mcd(b,p)=1$ se cumple que $b^{p-1}\equiv 1(\mathrm{mod}\, p)$. Evidentemente, si p es primo, la condición $mcd(b,p)=1$ se cumple para todo valor $b<p$, y solamente será falsa si $b=k\cdot p$.

El teorema de FERMAT nos ofrece nuevas vías para determinar la condición de primalidad de un entero cualquiera dado. Por ejemplo, con la base 3, ($mcd(341,3)=1$) tenemos que $3^{341-1}\equiv 56(\mathrm{mod}\, 341)$, y así ahora deducimos que 341 es compuesto. Al cambiar la base hemos logrado descartar el 341 como candidato a primo.

Si *n* es un impar compuesto, coprimo con *b* y verifica que $b^{n-1}\equiv 1(\mathrm{mod}\, n)$, entonces decimos que *n* es un **pseudoprimo para la base *b***. Así, por ejemplo, hemos visto que el número 341 es pseudoprimo para la base 2, pero no para la base 3.

Desafortunadamente existen números n que son pseudoprimos para todas las bases b tales que $mcd(b,n)=1$. El primero de ellos es el número $561=3\times 11\times 17$. Estos números se llaman **números de CARMICHAEL**. Son extremadamente raros, y sólo hay 2.163 menores que 25×10^9 [Bres89]. Para todo número de CARMICHAEL n, si tomamos cualquier base b coprimo con n, ésta tendrá un orden multiplicativo que divide a $n-1$. Hay infinitos números de CARMICHAEL, y podemos definirlos como aquellos números n que verifican que $\Phi(n)$ divide a $n-1$ [Land99].

De lo dicho hasta ahora, podemos deducir que conocemos formas de identificar un entero primo con un alto porcentaje de acierto. De todas formas eso no es suficiente: Aunque son pocos los números de CARMICHAEL en relación al total de los enteros en un intervalo, la verdad es que no se ha conseguido con todo esto caracterizar de modo inequívoco a los primos.

Existen diferentes test deterministas de primalidad, todos ellos de enorme costo en computación. Son test deterministas, que reciben como entrada un entero candidato a primo y ofrecen como salida la respuesta cierta sobre si el número en cuestión es o no es primo. Información sobre estos tests puede encontrarse, entre otras referencias, en [Mene97]. También hemos acudido para su estudio a la publicación de Juan TENA y Félix LÓPEZ [Lope90]. En general cualquier obra sobre teoría de números recoge abundante documentación sobre estos tests.

Sí existen algunos métodos deterministas de búsqueda de primos especiales, como los llamados **primos de MERSENNE**: son primos obtenidos de entre los números de MERSENNE. Los **números de MERSENNE** son enteros de la forma $M(n)=2^n-1$. Los cuatro primeros números de MERSENNE son 3, 7, 15 y 31. Está demostrado que si n es compuesto, entonces $M(n)$ también lo es. Por desgracia existen números de MERSENNE $M(p)$ para p primo que son compuestos. De hecho, los

primos p para los que $M(p)$ es primo escasean casi desde el principio. En [Bres89] se recoge un método para verificar si un determinado número de MERSENNE es primo o compuesto.

Disponemos de algunos métodos llamados probabilisticos. Son métodos muy ágiles a la hora de descartar un candidato a primo que sea realmente compuesto, pero que en el caso de que el número sea primo nunca termina de certificarlo. Simplemente otorga una alta probabilidad de que el número sea realmente primo. Son algoritmos que se conocen como algoritmos de MONTECARLO. Estos tests son los que hemos implementado en nuestro trabajo, y es con los primos obtenidos a partir de ellos con los que hemos trabajado para la generación de compuestos producto de dos primos aleatorios grandes.

2.2.1. Símbolos de LEGENDRE y de JACOBI.

Sea p un primo impar. Sea a un entero coprimo con p. Vamos a detenernos en la pregunta: ¿Es a un cuadrado perfecto módulo p?, es decir, ¿existe un entero x tal que $x^2 \equiv a(\mathrm{mod}\, p)$?

Para el tratamiento de esta pregunta introducimos dos conceptos:

Si n es un entero mayor que 0, decimos que el entero a es un **residuo cuadrático** de n si:

1. $mcd(a,n)=1$
2. La congruencia $x^2 \equiv a(\mathrm{mod}\, n)$ tiene solución.

Si no se cumple la condición (2) decimos que a es un **no residuo cuadrático** de n.

Se puede demostrar que si p es un primo impar, entonces hay exactamente tantos residuos como no residuos cuadráticos de p sobre los enteros $1,2,...,(p-1)$. Esta idea viene expresada en el siguiente teorema:

> Si p es un primo impar, entonces hay exactamente $(p-1)/2$ residuos cuadráticos de p y $(p-1)/2$ no residuos cuadráticos de p entre los enteros $1,2,...,(p-1)$ [Rose93].

La introducción del concepto de residuo cuadrático requiere una explicación. ¿Qué interés tiene conocer si la expresión $x^2 \equiv a(\mathrm{mod}\, n)$ tiene solución? Y si esa respuesta que buscamos es importante, ¿cómo saber en cada caso si un determinado entero a tiene soluciones para un determinado entero n?

2.2.1.1. interés del residuo cuadrático en criptografía.

Ya ha quedado formulada la pregunta que sucintamente intentaremos resolver ahora. ¿Qué interés tiene, para nuestros trabajos de criptografía y de búsqueda de enteros primos, conocer si la expresión $x^2 \equiv a(\mathrm{mod}\, n)$ tiene solución? Vamos a señalar dos motivos que justifican la necesidad de lograr conocer en cada par (a,n) si a resulta ser residuo cuadrático o residuo no cuadrático de n.

Primer interés: El conocimiento de la condición de a como residuo cuadrático o no cuadrático de n nos ofrecerá un test probabilístico de primalidad para el entero n. Es el llamado test probabilístico de SOLOVAY–STRASSEN.

Segundo interés: Los algoritmos subexponenciales de factorización se fundamentan en una idea originaria de FERMAT de buscar dos cuadrados que verifiquen la relación $x^2 \equiv y^2 (\mathrm{mod}\, n)$. Y el método usado es siempre buscar muchas congruencias de la forma $x^2 \equiv t(\mathrm{mod}\, n)$, de forma que el valor de t sea B-suave: es decir, que t quede completamente factorizado por todos los primos menores que B. Evidentemente, un primo $p < B$ no podrá dividir a ningún valor de t si verifica que p es residuo no cuadrático de n, independientemente de cual sea el valor de x. Por tanto a la hora de determinar la propiedad de suavidad de cada valor de t es necesario determinar aquellos primos menores que B que realmente tienen alguna posibilidad de dividir a t: todo primo que sea residuo No cuadrático de n jamás podrá dividir a ningún valor de t que cumpla una relación $x^2 \equiv t(\mathrm{mod}\, n)$, sea cual sea el valor de x.

Ambas razones son más que suficientes para que busquemos respuesta a la segunda cuestión presentada en el epígrafe anterior: ¿cómo saber si un determinado entero a tiene soluciones para un determinado entero n en la expresión $x^2 \equiv a(\mathrm{mod}\, n)$?

2.2.1.2. Símbolos de JACOBI y LEGENDRE. Nociones y métodos de cálculo.

Sea p un primo impar. Sea a un entero no divisible por p. Definimos el **símbolo de LEGENDRE** como:

$(a/p) = 1$ si a es residuo cuadrático de p.

$(a/p) = -1$ si a es no residuo cuadrático de p.

Tenemos definido así el concepto del símbolo de LEGENDRE, pero necesitamos ahora un criterio para saber cuándo un entero es residuo cuadrático de un primo y cuando es no residuo cuadrático.

Este criterio nos lo presenta el teorema llamado **criterio de EULER**: Sea p un primo impar. Sea a un entero no divisible por p. Entonces $(a/p) = a^{(p-1)/2} (\mathrm{mod}\, p)$.

Además de este criterio, existen algunas propiedades que nos serán de gran utilidad para el cálculo del valor del símbolo. Estas propiedades se definen mediante una serie de teoremas que vienen recogidos y demostrados en [Rose93].

Sea p un primo impar. Sean a y b enteros no divisibles por p. Entonces:

1. Si $a \equiv b(\mathrm{mod}\, p)$, entonces $(a/p) = (b/p)$
2. $(a/p) \cdot (b/p) = (a \cdot b/p)$ De esta propiedad se deduce que el producto de dos residuos cuadráticos o de dos no residuos cuadráticos de un primo es un residuo cuadrático de ese primo, mientras que el producto de un residuo cuadrático y un no cuadrático resulta ser un no

residuo cuadrático.

3. $(a^2/p)=1$ (consecuencia inmediata de 2).

4. $(-1/p)=1$ si $p\equiv 1(\mathrm{mod}\,4)$; $(-1/p)=-1$ si $p\equiv -1(\mathrm{mod}\,4)$. Gracias a esta propiedad, podemos clasificar todos los primos según tengan o no a -1 como residuo cuadrático.

5. Lema de GAUSS: Sea p un primo impar. Sea a un entero tal que $mcd(a,p)=1$. Sea s el número de residuos (tomados todos ellos como el menor residuo positivo de la clase) de los enteros $a, 2\cdot a, ..., ((p-1)/2)\cdot a$ que sean mayores que $p/2$. Entonces se cumple que $(a/p)=(-1)^s$.

6. Si p es un primo impar, entonces
$(2/p)=(-1)^{\frac{p^2-1}{8}}$.

   Por lo tanto, 2 es un residuo cuadrático de todos los primos $p\equiv \pm 1(\mathrm{mod}\,8)$ y es un no residuo cuadrático de todos los primos $p\equiv \pm 3(\mathrm{mod}\,8)$.

7. Ley de la reciprocidad cuadrática. Supongamos que tenemos dos primos distintos, p y q. Supongamos además que sabemos si q es o no residuo cuadrático de p. ¿Sabemos entonces si p es un residuo cuadrático de q? LEGENDRE dio respuesta a esta cuestión enunciando la ley de la reciprocidad cuadrática [Rose93]. La conclusión final de la ley de reciprocidad cuadrática es que:

$$(p/q)=(q/p) \text{ si } p\equiv 1(\mathrm{mod}\,4) \text{ y/o } q\equiv 1(\mathrm{mod}\,4);$$

$$(p/q)=-(q/p) \text{ si } p\equiv q\equiv 3(\mathrm{mod}\,4).$$

Con todas estas propiedades podemos definir de modo definitivo un algoritmo que nos ofrezca en todos los casos el valor del símbolo de LEGENDRE de un entero cualquiera a con respecto a un primo cualquiera p.

El **símbolo de JACOBI** es una generalización del símbolo de LEGENDRE.

Sea n un entero positivo impar tal que $n=p_1^{t1}\times p_2^{t2}\times ...\times p_m^{tm}$. Sea a un entero relativamente primo con n. Definimos el símbolo de JACOBI (a/n) como:

$$(a/n)=(a/p_1)^{t1}\times(a/p_2)^{t2}\times ...\times(a/p_m)^{tm}$$

donde los símbolos de la derecha de la igualdad son los símbolos de LEGENDRE.

Cuando n es primo, los símbolos de JACOBI y LEGENDRE se identifican. (Por tanto nos vamos a centrar únicamente en la implementación del símbolo de JACOBI, pues calculado éste tenemos calculado el de LEGENDRE para los casos en que n sea un entero primo). Cuando n es compuesto, el valor del símbolo de JACOBI no nos dice si la congruencia $x^2\equiv a(\mathrm{mod}\,n)$ tiene soluciones. Sabemos que si las tiene, entonces $(a/n)=1$. Pero es posible tener este mismo valor del símbolo de JACOBI en caso de congruencias que no tengan solución.

Comencemos viendo algunas propiedades del símbolo de JACOBI, semejantes a las que vimos para el símbolo de LEGENDRE:

Sea n un entero positivo impar. Sean a y b enteros relativamente primos con n. Entonces:

1. Si $a \equiv b(\mathrm{mod}\, n)$, entonces $(a/n) = (b/n)$
2. $(a/n) \cdot (b/n) = (a \cdot b/n)$
3. $(-1/n) = (-1)^{\frac{n-1}{2}}$
4. $(2/n) = (-1)^{\frac{n^2-1}{8}}$
5. También se cumple la Ley de Reciprocidad para el símbolo de JACOBI al igual que se cumplía para el símbolo de LEGENDRE. Sean n y m enteros impares positivos relativamente primos. Entonces

$$(n/m) \times (m/n) = (-1)^{\frac{n-1}{2} \times \frac{m-1}{2}}$$

2.2.2. En busca de los números primos.

2.2.2.1. Distribución de los números primos dentro del conjunto de los enteros.

Está demostrado que la cantidad de números primos es infinita, y por tanto que podemos encontrar un primo mayor que cualquier entero dado [Bres89].

También está demostrado que en el conjunto de los enteros el salto entre un primo y el siguiente consecutivo puede ser arbitrariamente grande: es decir siempre es posible encontrar dos primos consecutivos cuya diferencia sea tan grande como se quiera [Ribe91]. Y excepto el caso de los números 2 y 3, todos los demás primos distan, el uno del otro, al menos en dos.

	Números primos en tramos de 100									
Desde 1 hasta 1000	25	21	16	16	17	14	16	14	15	14
Desde 10^6 hasta 10^6 + 1000	6	10	8	8	7	7	10	5	6	8
Desde 10^7 hasta 10^7 + 1000	2	6	6	6	5	4	7	10	9	6
Desde 10^12 hasta 10^12 + 1000	4	6	2	4	2	4	3	5	1	6

Cuadro 7: Cantidad de números primos en diferentes intervalos de enteros. (Datos tomados de [Ore48]).

Una última característica de la distribución de los primos hace referencia a la densidad de primos

en un determinado rango. Vemos en el Cuadro 7 la cantidad de primos que existen en diferentes intervalos de enteros.

Hay, como se puede ver, grandes irregularidades en la distribución de los primos. Sin embargo, cuando consideramos la distribución de primos a grandes escalas encontramos pequeñas regularidades que obedecen a leyes simples. Si llamamos $p(x)$ a la cantidad de enteros primos menores o iguales que x, una aproximación a este número viene dada por la expresión

$$\lim_{x\to\infty} \frac{p(x)}{x/\ln x} = 1,$$

es decir, $p(x)$ es asintóticamente equivalente a $x/\ln x$ [Kran86].

Existen otras expresiones que aproximan más el valor de $p(x)$. Por ejemplo ([Ribe91], [Ries87]):

$$\int_2^x \frac{dt}{\log t}.$$

Con toda la información que hemos presentado en las líneas que preceden podemos sacar las siguientes conclusiones:

1. Podemos encontrar un primo tan grande como queramos.
2. No se conocen pautas de regularidad en la distribución de los primos dentro del conjunto de los enteros. Dado un número primo no tenemos ninguna propiedad que nos ofrezca una pista para encontrar otro primo distinto.
3. La distribución de los primos en intervalos de los enteros tiene un comportamiento enormemente irregular. Esa es la razón por la que aparezca como imposible definir una simple fórmula que describa la distribución de los primos con cierto detalle. Podemos decir que los primos vienen completamente mimetizados dentro del conjunto de los enteros.

Resulta de gran importancia en la teoría de números saber determinar si un entero dado es primo o compuesto. A primera vista puede parecer que para dar respuesta a esa pregunta será necesario hacer la factorización del entero y concluir su condición de primo o compuesto a la vista del resultado de los factores obtenidos.

Afortunadamente existen tests de primalidad que no hacen necesaria la factorización. Eso es una gran suerte, pues los procesos de factorización son enormemente laboriosos. Los métodos para determinar la primalidad de un entero no ofrecen ninguna información sobre cuáles son los factores de ese entero en el caso de que el test determine que el entero es compuesto [Ries87].

2.2.2.2. Tests probabilísticos de primalidad.

Entendemos aquí por test probabilístico un **algoritmo del tipo de MONTECARLO**. Los algoritmos de MONTECARLO están especialmente indicados cuando no conocemos ningún algoritmo eficiente que en todos los casos posibles sea capaz de obtener una solución correcta. Son algoritmos que

pueden cometer un error en la respuesta pero que a cambio encuentran una buena aproximación a la solución correcta, con una alta probabilidad de éxito. Un algoritmo de MONTECARLO no debe en ningún caso presentar una alta probabilidad de obtener una solución errónea, pero tampoco garantiza que la solución aproximada que aporta sea la correcta.

Un **test probabilístico de primalidad** es un algoritmo de MONTECARLO que, aplicado a un número natural n permite afirmar la primalidad de n con una cierta probabilidad. Si n no pasa el test implica que n es compuesto. Si n pasa el test podemos asegurar con una cierta probabilidad que n es primo. Como el test puede repetirse tantas veces como se desee, si n pasa un número "suficientemente grande" de veces el test podremos deducir que n es "casi seguramente" primo.

Los algoritmos de MONTECARLO más conocidos y utilizados para los test probabilísticos de primalidad son el algoritmo de MILLER–RABIN, el algoritmo de SOLOVAY–STRASSEN y el algoritmo de BAILLIE–WAGSTAFF [Ribe91]. Podemos encontrar información de estos algoritmos en cualquier referencia bibliográfica sobre teoría de números. He tomado para la descripción de los algoritmos la presentación de Juan TENA [Lope90].

Algoritmo de SOLLOVAY–STRASSEN: Sea n entero. Elegimos k enteros a_i tales que $0 < a_i < n$ y que $mcd(a_i, n) = 1$. Para cada uno de los valores a_i aplicamos el criterio de EULER. Si el criterio de EULER no se verifica para alguno de los valores a_i entonces n es compuesto. Si el criterio de EULER se verifica para las k bases entonces la probabilidad de que n sea compuesto es menor o igual de $1/2^k$.

Criterio de EULER: Si n es primo, para todo a_i primo con n se verifica que

$$a_i^{\frac{n-1}{2}} \equiv (a_i/n) (\mathrm{mod}\, n)$$

donde (a_i/n) representa el símbolo de JACOBI.

Algoritmo de MILLER–RABIN: Sea n entero impar, $n-1 = 2^e \times t$, donde t es impar. Elegimos arbitrariamente k enteros a_i tales que $0 < a_i < n$. Para cada valor de a_i calculamos $a_i^t (\mathrm{mod}\, n)$. Si el resultado es ± 1 (es decir, 1 ó $n-1$), entonces n supera el test en esa base y pasamos al siguiente valor a_i. En caso contrario elevamos a_i^t al cuadrado en aritmética $\mathrm{mod}\, n$ y volvemos a elevarlo sucesivamente hasta obtener eventualmente el valor -1 (es decir $n-1$) en cuyo caso n pasa el test y pasamos al siguiente valor a_i.

Si llegamos a realizar e potencias de 2 y no obtenemos ningún -1

(es decir, $a_i^{2^{i+1} \times t} \equiv 1 (\mathrm{mod}\, n)$ pero $a_i^{2^i \times t} \equiv -1 (\mathrm{mod}\, n)$)

entonces n no pasa el test y es por tanto compuesto.

Si n pasa el test para k bases la probabilidad de que sea compuesto es, a lo sumo, $1/4^k$.

Existen otros tests de primalidad basados en algoritmos deterministas. Sin embargo, estos tests

tienen una gran dificultad de implementación y un altísimo coste de tiempo y de recursos de computación. A continuación veremos unos brevísimos apuntes sobre esos tests, pero finalmente, en nuestro trabajo, hemos implementado únicamente algoritmos de MONTECARLO. Son muchas las referencias en las que se recomienda el uso del algoritmo de MILLER–RABIN u otro probabilístico y no el de los tests deterministas (por ejemplo [Silv97]).

2.2.2.3. Tests deterministas.

Un test es determinista si ofrece certeza en la respuesta al algoritmo. Un test determinista de primalidad será aquel que dado un entero candidato el algoritmo determina con un acierto del 100 % que ese entero es primo o es compuesto. Ya ha quedado dicho que estos tests requieren un consumo de tiempo enorme que los hace inutilizables en la práctica.

Existen, sin embargo, algunos tests válidos para enteros especiales, como los números de MERSENNE o los números de FERMAT. Algunos de estos enteros son primos, y su condición de primalidad puede deducirse con certeza con algunos algoritmos.

Entendemos por **número de FERMAT** aquel de la forma $F_n = 2^{2^n} + 1$; a F_n se le llama n-ésimo número de FERMAT. Entendemos por número de MERSENNE aquel de la forma $M_m = 2^m - 1$; a M_m se le llama m–ésimo número de MERSENNE.

Los números de MERSENNE deben su nombre al matemático Marin MERSENNE, que fue su principal divulgador en el siglo XVII. Pero ya los manejaba cuatro siglos antes de Jesucristo el matemático EUCLIDES, que, en su búsqueda de números perfectos (m es perfecto si es igual a la suma de sus divisores propios: todos los divisores menos el mismo m). Ya EUCLIDES demostró que era suficiente, para que un entero m fuera perfecto, que se cumpliera que era

$$m = 2^{k-1} M_k$$

y que el número de MERSENNE M_k era primo.

Fue EULER quien, unos dos mil años más tarde, ya en el Siglo XVIII demostró que esa condición suficiente presentada por EUCLIDES era también una condición necesaria.

Pierre de FERMAT fue un matemático anterior a EULER en un siglo. FERMAT conjeturó que todos los números de FERMAT (que ya hemos presentado) eran primos; pero no pudo demostrarlo. EULER desmintió esa conjetura al presentar el valor del quinto número de FERMAT

$$2^{2^5} = 4.294.967.297$$

y comprobar que era divisible por el primo 641.

Pero al margen de estos apuntes históricos sobre los números de MERSENNE o de FERMAT [Dunh00], lo que sí podemos ver, por ejemplo en [Rose93], es que existen algoritmos que nos permiten saber si un determinado Entero de MERSENNE es primo o compuesto.

A partir del test de LUCAS–LHEMER (teorema 6.13 de [Rose93]) que ofrece un test de primalidad

para los números de MERSENNE, se ha trabajado mucho en la búsqueda de enteros de MERSENNE que sean primos (y que llamamos primos de MERSENNE). Gracias a este método hemos podido detectar los primos más grandes hasta la fecha conocidos. El último determinado como primo (el 14 de noviembre de 2001) es el número cuyo exponente es $n = 13.466.917$, que resulta un entero de 4.053.946 dígitos decimales. Para contemplar semejante cifra puede consultarse [Prim_1]. Más información sobre este número, o sobre los primos de gran tamaño puede encontrarse en [Cald02].

2.2.2.4. Algoritmo de MILLER–RABIN.

Nos detenemos aquí ahora en la presentación gráfica del test de MONTECARLO más universal y conocido: el test de MILLER–RABIN.

Ya hemos visto los enteros pseudoprimos para una determinada base y los números de CARMICHAEL. Hemos de introducir ahora el concepto de pseudoprimo fuerte:

Tomemos n entero que pase el test de pseudoprimalidad para una determinada base b, es decir, $b^{n-1} \equiv 1 (\mathrm{mod}\, n)$, lo que implica que n divide a $b^{n-1} - 1$.

Asumido que n es impar, podemos escribir que $n = 2 \cdot m + 1$, y entonces n divide a $(b^m + 1) \cdot (b^m - 1)$. Si n es realmente primo, entonces debe dividir al menos a uno de estos dos factores. Pero además no puede dividir a los dos a la vez, pues entonces debería dividir también a la diferencia que es 2 (que, como se sabe, es primo).

Por tanto, si n es primo verificará que ó $b^m \equiv +1 (\mathrm{mod}\, n)$ ó $b^m \equiv -1 (\mathrm{mod}\, n)$. Pero si n es compuesto, podría ser que algunos de sus factores dividieran a $(b^m + 1)$ y otros dividieran a $(b^m - 1)$: en ese caso, podría suceder que n pasara el test de pseudoprimalidad, pero no satisficiera ninguna de las dos expresiones de congruencia indicadas en este párrafo. Si miramos el Cuadro 8, que representa la tabla de potencias en $\mathbf{Z}_{13}$, podemos ver que para todas las bases la potencia $m = 6$ vale siempre ó 1 ó 12 (que es un miembro de la misma clase de equivalencia que el valor -1).

Generalizando un poco más esa idea, podemos suponer que n adopta la expresión

$$n = 2^e \cdot t + 1 \qquad (1)$$

donde t es impar y $e \geq 1$. Una vez más, si $b^{n-1} \equiv 1 (\mathrm{mod}\, n)$ implica que $n \mid b^{n-1} - 1$.

Ahora podemos expresar $b^{n-1} - 1$ como:

$$b^{n-1} - 1 = (b^t + 1) \cdot (b^t - 1) \cdot (b^{2 \cdot t} + 1) \cdot (b^{4 \cdot t} + 1) \cdot \ldots \cdot (b^{2^{e-1} \cdot t} + 1) \qquad (2)$$

Expresión que viene tomada de [Bres89] y de inmediata comprobación sin más que ir realizando los productos indicados, que se van convirtiendo siempre en diferencias de cuadrados.

Entonces decimos que el número impar $n = 2^e \cdot t + 1$ es pseudoprimo fuerte para la base b (con

$mcd(n,b)=1$), si $b^t \equiv 1 (\mathrm{mod}\, n)$, o bien si existe i, $0 \le i < e$ tal que $b^{2^i \cdot t} \equiv -1 (\mathrm{mod}\, n)$: es decir, si divide a alguno de los factores de la parte derecha de la igualdad, lo que querrá decir que $n \mid b^{n-1} - 1$.

Existen pseudoprimos fuertes, pero son especialmente extraños. El test de primalidad será más eficaz si le exigimos que se verifique para varias bases diferentes.

Se demuestra que si n es compuesto, el número de bases b, $1 < b < n$, $\mathrm{mcd}(n,b)=1$ para las que no es pseudoprimo fuerte es, al menos, $3 \cdot \Phi(n)/4$. Por tanto, la probabilidad de que, tomada la base b, n no sea pseudoprimo fuerte y pase el test de primalidad es, al menos, del 75 %.

Veámoslo con un ejemplo gráfico presentado en el Cuadro 8, de apariencia muy similar al ya mostrado Cuadro 4.

13		1	2	3	4	5	6	7	8	9	10	11	12
	0												
1	1	1	1	1	1	1	1	1	1	1	1	1	1
2		2	4	8	3	6	12	11	9	5	10	7	1
3		3	9	1	3	9	1	3	9	1	3	9	1
4		4	3	12	9	10	1	4	3	12	9	10	1
5		5	12	8	1	5	12	8	1	5	12	8	1
6		6	10	8	9	2	12	7	3	5	4	11	1
7		7	10	5	9	11	12	6	3	8	4	2	1
8		8	12	5	1	8	12	5	1	8	12	5	1
9		9	3	1	9	3	1	9	3	1	9	3	1
10		10	9	12	3	4	1	10	9	12	3	4	1
11		11	4	5	3	7	12	2	9	8	10	6	1
12		12	1	12	1	12	1	12	1	12	1	12	1

Cuadro 8: Comportamiento de las potencias módulo un entero primo, según el test de primalidad de MILLER–RABIN.

Lo primero que vemos es que para todas las bases se verifica que $b^{n-1} \equiv 1 (\mathrm{mod}\, 13)$. Eso ya nos indica que el número es primo, pues verifica el teorema de FERMAT para todas las bases $b < n$, y no cabe siquiera que sea un número de CARMICHAEL, pues en ese caso al menos encontraríamos algunas bases b tales que $mcd(b,n) \neq 1$, y entonces $b^{n-1} \not\equiv 1 (\mathrm{mod}\, 13)$. Pero vamos a ignorar esa evidencia y sigamos analizando los valores; al menos si n fuese un número suficientemente grande no podríamos hacer el cálculo $b^{n-1} \bmod n$ para todas las bases.

El entero 13 puede expresarse, según recoge (1), de la siguiente forma:

$$13 = 2^2 \cdot 3 + 1 \text{: es decir, } e = 2 \text{ y } t = 3$$

Primera exploración: buscar qué bases verifican que $b^t \equiv 1(\mathrm{mod}\, 13)$. Y encontramos las bases 3 y 9 (señalados en el Cuadro 8 con el color azul).

Segunda exploración: Tenemos que $t = 3$ y entonces, para valores de i en el intervalo $0 \le i < e = 2$ la expresión $2^i \cdot t$ toma los valores 3 ó 6. Probamos cada uno de estos dos valores buscando en qué casos se cumple que $b^{2^i \cdot t} \equiv -1(\mathrm{mod}\, n)$.

Con $i = 0$ ($2^i \cdot t = 3$) encontramos tres bases que verifican la expresión: la 4, la 10 y la 12 (en el Cuadro 8, de color verde).

Con $i = 1$ ($2^i \cdot t = 6$) encontramos seis bases (las restantes) que verifican la expresión: señalados en el Cuadro 8 con el color amarillo.

Como se puede comprobar contemplando el Cuadro 8, todas y cada una de las bases ha verificado alguna de las opciones señaladas que permiten pasar el test de MILLER–RABBIN.

Cuando trabajamos con candidatos de mayor tamaño (por ejemplo, cuando necesitamos encontrar enteros primos de, por ejemplo, 1000 bits), no es posible realizar las pruebas con cada uno de las bases en un tiempo razonable. Pero en el caso de que tomemos una base cualquiera se cumple que la probabilidad de que el comportamiento sea el requerido para el test es menor de $1/4$ en el caso de que el candidato sea compuesto.

Como señalan A. MENEZES, P. Van OORSCHOT y S. VANSTONE en [Mene97] el algoritmo de MILLER–RABIN es mejor por las siguientes razones:

1. El test de SOLOVAY–STRASSEN es computacionalmente más costoso.
2. El test de SOLOVAY–STRASSEN es más difícil de implementar y exige además el cálculo de los símbolo de JACOBI.
3. La probabilidad de error con el algoritmo de SOLOVAY–STRASSEN está acotada por $(1/2)^t$, mientras que con el algoritmo de MILLER–RABIN la cota es $(1/4)^t$.
4. Matemáticamente se comprueba que el test de MILLER–RABIN no se equivoca en ningún caso en que tampoco lo haga el test de SOLOVAY–STRASSEN.

La misma opinión defiende Henri COHEN [Cohe93] de forma contundente: "El test de SOLOVAY–STRASSEN ha sido superado por el test de MILLER–RABIN, que tiene dos ventajas: La primera, que no necesita la computación del símbolo de JACOBI. La segunda, que el número de veces que debemos pasar el test sobre en candidato a primo para alcanzar el mismo grado de certeza es la mitad en el test de MILLER–RABIN frente al número que exige el test de SOLOVAY–STRASSEN. Además –sigue afirmando Henri COHEN–, se puede probar que si un número satisface el test de MILLER–RABIN para una determinada base, entonces también satisface el test de SOLOVAY–STRASSEN para esa misma base, y por tanto el test de MILLER–RABIN supera completamente al test de SOLOVAY–STRASSEN".

2.3. OBTENCIÓN DE LOS PRIMOS QUE COMPONEN UN ENTERO: FACTORIZACIÓN.

En las últimas décadas hemos visto la llegada del poder de computación, que se ha hecho más accesible y más rápido. Y aunque al principio casi nadie prestó atención al problema de la factorización, por considerarse una cuestión trivial, hemos visto como la tarea de factorizar un entero, producto de dos primos largos, no resulta sencilla aunque dispongamos de mucha potencia de cálculo; y que los algoritmos actualmente existentes no logran dar solución a este problema: al menos entre los algoritmos conocidos por la comunidad científica.

Y así también han aparecido los sistemas criptográficos que basan su seguridad en esta supuesta inhabilidad para factorizar. Los tiempos necesarios para la factorización de enteros puestos en relación con el tamaño de esos números nos indican el grado de seguridad.

El estudio de las técnicas de factorización de números de gran tamaño, y todo posible avance para lograr reducir tiempos en esos procesos, resulta de interés para la criptografía. No es necesario señalar el interés matemático de la cuestión, que no puede darse por resuelta.

El teorema fundamental de la aritmética afirma que la factorización de un compuesto es única excepto en el factor 1 (que interviene tantas veces como queramos) y en el orden de los factores. Factorizar un entero consiste en encontrar esos primos cuyo producto es igual al entero dado. En este epígrafe presentamos sucintamente diferentes algoritmos de factorización, cada uno de ellos útil para un rango de tamaño de entero y para algunos enteros cuyos factores primos verifiquen alguna propiedad específica determinada que iremos viendo.

2.3.1. Intento de factorización por divisiones sucesivas.

El procedimiento teóricamente más sencillo para la búsqueda de los factores primos de un entero dado cualquiera n, consiste en tomar una tabla de los primeros valores primos y proceder a calcular el módulo de dividir n por los sucesivos primos, comenzando por el primero de ellos (el 2). Cada vez que se encuentra un primo p que divide al candidato n se inicia el proceso a partir de ese primo, con $n \leftarrow n/p$. Si, llegado a $\sqrt{n}$ no se ha encontrado ningún primo que divida a n o a lo que queda de él después de sucesivas divisiones, entonces podemos dar por terminado el proceso.

Este proceso puede ser útil para enteros pequeños (enteros que no superen el orden de 10^7), pero en cuanto se aumenta el tamaño del número a factorizar se hace impracticable pues requiere un tiempo de computación excesivo.

2.3.2. Algoritmo Rho de POLLARD

Un siguiente paso en los métodos de factorización será utilizar el método Rho de POLLARD. Este algoritmo está recogido en cualquier libro sobre factorización de los citados en la bibliografía de la tesis. Por ejemplo [Bres89]. Es un algoritmo útil para enteros n de rango entre 10^6 y 10^{12}.

Se toma una aplicación $f(x)$, fácilmente evaluable, de $\mathbf{Z}_n$ en $\mathbf{Z}_n$ (por ejemplo, un polinomio de coeficientes enteros: en la práctica suele tomarse la función $f(x)=x^2+1$). Llamamos d a uno de los divisores de n, que evidentemente desconocemos todavía. Se elige un valor particular inicial $x=x_0$ y se procede a calcular sucesivas iteraciones de la aplicación f:

$$x_1 = f(x_0) \bmod n, \qquad x_2 = f(x_1) \bmod n = f^2(x_0) \bmod n, \qquad x_i = f(x_{i-1}) \bmod n = f^i(x_0) \bmod n$$

Por otro lado supongamos la secuencia (que no podemos generar pues desconocemos el valor de d) $y_i = x_i \bmod d$.

Por el modo en que queda definida la secuencia x_i es inmediato ver que y_i es congruente módulo d con $f(y_{i-1})$. Existe una número finito de clases de congruencias y, por tanto, en algún momento se encontrará, en la serie y_i algún par de valores tales que $y_i = y_k$. Habremos encontrado entonces un comportamiento cíclico. Nuestra secuencia aparece entonces "como un círculo del que pende un tallo", es decir, adquiere la apariencia de la letra griega rho (r), que es lo que da nombre al método.

Si $y_i = y_k$ entonces se verifica que $x_j \equiv x_k \pmod d$ y, por tanto, d es múltiplo de $x_j - x_k$. Y entonces, si $mcd(x_j - x_k, n) = d \neq 1$ habremos encontrado un divisor propio de n.

El problema es que como desconocemos el valor de d no podemos calcular la secuencia y_i y no puedo determinar el par (j,k) que verifique que $y_j = y_k$. Sabemos que, de hecho, hay infinitos pares (j,k) para los que se verifica que $y_j = y_k$. Si la longitud del ciclo es, por ejemplo, c, entonces una vez estemos "fuera del tallo de la rho", podremos trabajar con cualquier par (j,k) que verifique que c divide a $j-k$. Necesitamos un modo sistemático de escoger pares (j,k) y computar cada vez el valor $\mathrm{mcd}(x_j - x_k, n)$ hasta llegar a un par tal que el máximo común divisor sea diferente de 1.

El método que se usa para escoger los distintos pares es el siguiente:

$x_{2^n-1} - x_j$, donde se toma j de tal manera que $2^{n+1} - 2^{n-1} \leq j \leq 2^{n+1} - 1$

Y procedemos a comparar los diferentes valores x_i, en busca de dos de ellos que tengan diferentes clases residuales módulo n, pero la misma clase residual módulo algún divisor de n. Una vez encontrados dos valores así, por ejemplo, x_j y x_k, tendremos que $\mathrm{mcd}(x_j - x_k, n) = d$, donde d será un divisor propio de n.

2.3.3. Algoritmo $(p-1)$ de POLLARD.

Este método es útil en el caso de que el número n que deseamos factorizar verifique que un factor suyo p sea tal que los primos que dividen a $(p-1)$ son todos ellos pequeños, es decir, menores que, por ejemplo, 10^4.

En ese caso, computamos $m = b^{10.000!} \bmod n$ (para cualquier base b coprima con n: un valor típico será $b = 2$).

Como $(p-1)$ divide a $10000!$ (pues ya hemos dicho que todos sus factores son menores que 10^4), se verifica que m es congruente módulo p con 1, es decir, que p divide a $m-1$. Por tanto, $g = \text{mcd}(m-1, n)$ será un divisor no trivial de n.

En la práctica el proceso se realiza evaluando $\text{mcd}(c^{k!}-1, n)$. Si el valor es 1 se continúa aumentando el valor de k. Si el valor es n se inicia el proceso probando con otro valor de c. Si el valor es distinto de 1 y de n es que se ha hallado un factor de n.

Una modificación a este proceso es determinar la lista de los primos menores de 10000; así, en lugar de ir calculando los factoriales, se puede realizar el proceso considerando únicamente los números primos. Un tercer procedimiento es calcular el mínimo común múltiplo de todos los enteros menores que el límite B de suavidad para $p-1$.

El algoritmo de factorización $p-1$ de POLLARD exige que los primos que su utilizan para RSA estén restringidos: Si $(p-1)$ ó $(q-1)$ sólo tienen factores primos pequeños, el ataque a la clave privada es inmediato.

Existe otro método, de características similares al ahora presentado, conocido como el método $(p+1)$ de WILLIAMS (cfr. por ejemplo [Bres89]) que es eficaz para aquellos compuestos en los que alguno de sus primos grandes p es tal que $(p+1)$ resulta divisible todo él por enteros primos pequeños.

2.3.4. Algunas exigencias para los primos que componen el módulo del criptosistema RSA, a la luz de estos algoritmos de factorización.

Supongamos que buscamos los primos p y q necesarios para la generación del módulo n, con $2 \cdot k$ bits, de un sistema de clave pública RSA.

Ambos primos deben pertenecer al conjunto de rango $\sqrt{2} \cdot 2^{k-1} < p, q < 2^k$.

El estándar ANSI X9.31 requiere que los primos p y q verifiquen la siguientes condiciones:

1. $p-1$ tiene un divisor primo p_1 con una longitud binaria de entre 100 y 120 bits.
2. $p+1$ tiene un divisor primo p_2 con una longitud binaria de entre 100 y 120 bits.

3. $q-1$ tiene un divisor primo q_1 con una longitud binaria de entre 100 y 120 bits.
4. $q+1$ tiene un divisor primo q_2 con una longitud binaria de entre 100 y 120 bits.
5. $|p-q| > 2^{k-100}$
6. Ambos valores, $p-1/2$ y $q-1/2$ deben ser primos con el exponente de la clave pública e.
7. Si el exponente de clave pública e es par, entonces se requiere que se verifique que $p \equiv 3(\mathrm{mod}8)$ y $q \equiv 7(\mathrm{mod}8)$

Las cuatro primeras exigencias se conocen con la expresión de que ambos primos p y q deben ser **primos fuertes**.

Al exigir que ambos primos sean fuertes se pretende defender al módulo n del ataque por factorización mediante los algoritmos de POLLARD o el de WILLIAMS (tomado de [Peyr01]).

Desde el mismo inicio de la existencia del criptosistema RSA ha estado abierto el debate sobre la necesidad de exigir que los primos fuesen fuertes. En los años 80 la respuesta era que sí. Los métodos de factorización de $(p+1)$ de WILLIAMS ó de $(p-1)$ de POLLARD así lo aconsejaban.

Los primos fuertes son sencillos de generar: GORDON ([Gord84], [Saou95]) ofrece un método sólo ligeramente más costoso en cómputo comparado con el generador de primos aleatorios.

Años más tarde, RIVEST y POMERANCE [Rive99] afirmaron que para resistir a los métodos de factorización de propósito general los factores primos del módulo de RSA necesitaban ser razonablemente grandes. Si los primos eran largos, y estaban generados de forma aleatoria, esos autores concluían que el resultado resistiría, en un alto grado de probabilidad, el ataque de factorización con los métodos de propósito especial; y por tanto, la exigencia de que los primos fuesen fuertes no añadía protección en la práctica. Únicamente daba una falsa sensación de seguridad.

El argumento quizá más convincente contra la necesidad de que los primos fuesen fuertes fue el método de las curvas elípticas (ECM). ECM, a diferencia del método $(p-1)$ de POLLARD y de otros previos de propósito especial, es igualmente efectivo sobre cualquiera entre todos los primos del mismo tamaño. No hace falta exigir a los primos más condición que la de la longitud: la debilidad de cada primo vendría dada únicamente por su longitud. El estándar era únicamente que los primos fuesen largos y aleatorios (cfr. [Kali99]).

Finalmente se ha llegado a un consenso general sobre la exigencia de que los primos sean fuertes. No para la defensa frente al ataque procedente de un atacante externo, sino también y sobre todo para la defensa contra las insidias de los propios usuarios cuando diseñan y construyen sus propias claves.

En circunstancias normales, la probabilidad de que, de forma fortuita o accidental, se tomen valores de claves débiles es extraordinariamente baja. Sin embargo, existe la posibilidad de que

una de las partes haga trampa y de forma deliberada intente generar una clave débil. Luego, esa persona podría repudiar mensajes firmados por ella, argumentando que su clave ha sido rota por un atacante y ha sido, por tanto, suplantado. Esto es lo que se llama ataque de primera parte ("first party attack"). Este ataque presentaba un serio problema a los diseñadores de los criptosistemas de clave pública.

El comité ANSI X9F1 sabía que un atacante de este forma podría saltarse las prescripciones de procedimiento sobre la generación de los primos, buscándolos de forma que fueran débiles. Por ese motivo, el comité exigió en el estándar X9.31 que los primos p y q se generasen a partir de seis semillas de valores, y que esas semillas debían almacenarse para poder probar de forma posterior que los primos no habían sido generados de forma fraudulenta. El protocolo X9.31 exige el uso de funciones de una vía para prevenir un ataque que provenga de trabajar hacia atrás, en busca de un grupo de valores para las semillas que conduzca a la generación de un particular par de primos [Maty99]. Estas nuevas exigencias suponen un coste asociado de almacenamiento (hay que guardar las seis semillas) y también introduce una nueva vía de inseguridad: si una de esas semillas llega a ser conocida por un atacante, éste podría llegar a determinar el valor de la clave secreta del sistema RSA [Peyr01].

2.3.5. Método de FERMAT.

FERMAT introdujo un nuevo método. Este se basa en la idea de que todo compuesto impar puede ser expresado como diferencia de cuadrados. Basta para demostrarlo con la identidad:

$$a \cdot b = \left(\frac{1}{2} \cdot (a+b)\right)^2 - \left(\frac{1}{2} \cdot (a-b)\right)^2$$

El intento de hallar un par de cuadrados para factorizar es, de hecho, uno de los métodos de factorización de FERMAT. El método de las divisiones sucesivas tiene casos muy eficaces, cuando los factores de n son primos pequeños. El método de la diferencia de cuadrados también tiene casos muy eficaces, cuando los factores primos son cercanos a $\sqrt{n}$. Pero, en general, este método de FERMAT es tan costoso como el método de las sucesiones sucesivas.

2.3.6. Algoritmos modernos (subexponenciales) de factorización.

En el año 1926, Maurice KRAITCHIK presentó una mejora interesante a la técnica de FERMAT. Esa mejora es la base de los algoritmos modernos de factorización: En lugar de intentar encontrar enteros u y v tales que $u^2 - v^2 = n$, KRAITCHIK propuso que era suficiente encontrar u y v tales que su diferencia de cuadrados fuesen un múltiplo de n, es decir, $u^2 \equiv v^2 (\mathrm{mod}\, n)$. Esta congruencias tiene soluciones no interesantes: aquellas que verifiquen que $u \equiv v (\mathrm{mod}\, n)$; y otra sí interesantes: aquellas donde $u \not\equiv v (\mathrm{mod}\, n)$. De hecho, si n es divisible por dos primos diferentes,

entonces al menos la mitad de las soluciones de $u^2 \equiv v^2 (\mathrm{mod}\, n)$ con el producto $u \cdot v$ coprimo con n serán las interesantes. Y cuando tenemos una solución válida, entonces $mcd(u-v, n)$ será un factor no trivial de n. Efectivamente, tendremos que n divide a $u^2 - v^2 = (u+v) \cdot (u-v)$, pero no divide a ningún otro factor. Y como ya conocemos, el cálculo del máximo común divisor de dos números es una tarea sencilla gracias al algoritmo de EUCLIDES.

La estrategia propuesta por KRAITCHIK para lograr encontrar una congruencia $u^2 \equiv v^2 (\mathrm{mod}\, n)$ consiste en intentar encontrar números x tales que los productos de $Q(x) = x^2 - n$ sean un cuadrado perfecto. Si

$$Q(x_1) \cdot Q(x_2) \cdot \ldots \cdot Q(x_k) = v^2 \quad \text{y} \quad x_1 \cdot x_2 \cdot \ldots \cdot x_k = u$$

entonces

$$u^2 = x_1^2 \cdot x_2^2 \cdot \ldots \cdot x_k^2 \equiv (x_1^2 - n) \cdot (x_2^2 - n) \cdot \ldots \cdot (x_k^2 - n) (\mathrm{mod}\, n) = Q(x_1) \cdot Q(x_2) \cdot \ldots \cdot Q(x_k) = v^2 (\mathrm{mod}\, n)$$

Y así habremos encontrado una solución para $u^2 \equiv v^2 (\mathrm{mod}\, n)$.

El método es teóricamente claro, pero el gran problema se presenta cuando debemos encontrar un conjunto de valores $x_1, x_2, \ldots, x_k$ que verifiquen que sus correspondientes productos $Q(x_1) \cdot Q(x_2) \cdot \ldots \cdot Q(x_k)$ formen un cuadrado.

2.3.7. Estrategia de MORRISON y BRILLHART: CFRAC.

John BRILLHART y Michael MORRISON [Morr75] definieron una estrategia y sistemática para encontrar una subsecuencia con su producto cuadrado. Y esta estrategia se apoya sobre el álgebra lineal. Consiste en tomar uno de los valores de la congruencia que sea cuadrado perfecto y el otro que sea "pequeño" en términos relativos: $u^2 \equiv v (\mathrm{mod}\, n)$.

A cada entero positivo v se le asocia un vector de exponentes, $e(v)$, que recibe sus valores en sus diferentes coordenadas en función de la factorización de v. BRILLHART y MORRISON sugirieron elegir un número B y buscar el subconjunto de valores de v's que verificase que su producto fuese un cuadrado perfecto, sólo entre aquellos números v de la secuencia que queden completamente factorizados con los B primeros primos (es decir, entre aquellos valores de v que sean B-suaves: un entero es B-suave si todos sus factores son menores que B). Por la misma definición, se exige que los primos p menores que B capaces de dividir a v deben verificar que $(n/p) = +1$ (símbolo de JACOBI). El vector $e(v)$ tendrá tantas coordenadas como primos haya menores que B que verifiquen la propiedad de que su símbolo de JACOBI respecto de n sea 1 (a ese conjunto de primos se le llama base de factores). Si un entero v queda descompuesto completamente con los primos de la base de factores, es decir,

$$v = \prod_{\substack{p_i < B \\ (n/p_i)=1}} p_i^{e_i} \quad (\text{donde } e_i \geq 0)$$

entonces la coordenada i-ésima del vector $e(v)$ valdrá el exponente del primo i-ésimo de la

base, reducido módulo 2. Si llamamos cfb al cardinal de la base de factores, entonces, cuando hayamos logrado $cfb+1$ congruencias $u^2 \equiv v(\mathrm{mod}\, n)$ con v factorizado con los primos de la base de factores, tendremos $cfb+1$ vectores en el espacio cfb dimensional F_2^{cfb}. Por álgebra sabemos que este conjunto de vectores debe ser linealmente dependientes: Dado que los únicos escalares del espacio vectorial son el 0 y el 1, una relación linealmente independiente es simplemente una suma de algunos de los vectores cuyo resultado sea el vector cero: y ese suma módulo 2 nos ofrece un camino para obtener la combinación de congruencias $u^2 \equiv v(\mathrm{mod}\, n)$ tales que el producto de sus respectivos v ofrece un valor cuadrado.

Muchas de las nuevas ideas para la tarea de factorización de enteros grandes se basan en el algoritmo presentado por BRILLHART y MORRISON que busca las congruencias $u^2 \equiv v(\mathrm{mod}\, n)$ del algoritmo de aproximación racional de $\sqrt{n}$ mediante la técnica de las fracciones continuas (CFRAC). Esta idea de buscar las relaciones con esta técnica se remonta al año 1931 y se debe a D. H. LEHMER y R. E. POWERS.

Este algoritmo quedará extensamente explicado, desarrollado e implementado en un capítulo de esta tesis dedicado a su completa presentación e implementación, y en otro capítulo dedicado a la optimización del código.

2.3.8. Criba lineal de SCHROEPPEL.

Richard SCHROEPPEL ofreció un método nuevo de búsqueda de congruencias $u^2 \equiv v(\mathrm{mod}\, n)$. SCHROEPPEL jamás publicó sus resultados: los conocemos gracias a sus cartas. A través de un estudio teórico sobre la complejidad, presentó un nuevo método que se conoció como método de la criba lineal. La aportación principal de SCHROEPPEL es una ingeniosa forma de reemplazar el tiempo invertido en las pruebas de división (para buscar valores v B-suaves) por otro en principio más breve: el tiempo necesario para realizar una criba como la de ERASTÓTHENES. Su idea era encontrar otros modos de producir residuos cercanos a $\sqrt{n}$ que tuviesen la ventaja de que pudieran factorizarse colectivamente mediante la criba.

El algoritmo de SCHROEPPEL es el siguiente. Sea $K = \lfloor \sqrt{n} \rfloor$ y consideremos la función:

$$S(A,B) = (K+A)\cdot(K+B) - n$$

donde A y B son enteros mucho menores, en valor absoluto, que K. Por ese motivo tendremos que $S(A,B)$ no será mucho mayor que K. La idea vuelve a ser intentar encontrar pares A_i, B_i tales que

$$\prod_{i=1}^{K} S(A_i, B_i)$$

sea un cuadrado, como con el algoritmo de CFRAC. Además, se debe lograr que cada valor distinto de $A_1 \ldots A_k, B_1 \ldots B_k$ sea asumido un número par de veces. Así entonces

$$\prod_{i=1}^{K}(K+A_i)\cdot(K+B_i)$$

será también un cuadrado.

Por tanto, si podemos encontrar tales colecciones fortuitas de pares A_i, B_i, entonces podemos encontrar enteros u y v tales que $u^2 \equiv v^2 \pmod{n}$.

El procedimiento para realizar la criba es tomar un valor fijo $A = A_0$ y variar B sobre enteros consecutivos. Estos números forman una progresión aritmética de tal manera que si $p \mid S(A_0, B_0)$, entonces $p \mid S(A_0, B_0 + p)$, $p \mid S(A_0, B_0 + 2 \cdot p)$,... En esta criba, en lugar de ir tachando los números múltiplos de cada primo que entra en juego en el proceso de la criba de ERASTHÓTENES, podemos ir dividiendo los compuestos por los sucesivos primos que los forman, llegando a transformar todos los compuestos al número 1 al final del proceso, cuando ya estén completamente factorizados. Si, además, en lugar de cribar por todos los primos hasta llegar a un valor de B igual a la raíz cuadrada del rango de la criba, lo hacemos sólo hasta el valor primo menor y más próximo al límite de suavidad tendremos a 1, al final del proceso, únicamente a aquellos valores que sean suaves.

Pero no todos los valores suaves quedarán reducidos al valor 1 al final del proceso: tenemos que considerar los casos en que un determinado valor $S(A_0, Bi)$ verifique que un mismo primo lo factorice más de una vez: tenemos el problema de las potencias de primos menores que el límite superior de suavidad. Debemos rectificar el método cribando también con estas potencias de primos menores que B.

Gracias a la criba, conocemos de antemano qué valores de B tendrán $S(A_0, B)$ divisibles por los primos p de la base de factores: los que después de la criba hayan quedado a 1 son valores de B para los que $S(A_0, B)$ es suave. Y entonces no debemos hacer un derroche de intentos de división en aquellos valores donde de antemano sabemos que no son suaves. Esto supondrá una ventaja importante sobre el algoritmo CFRAC, donde los valores a factorizar surgen de modo aparentemente aleatorio y no es posible realizar un proceso de criba.

La ventaja de la criba lineal de SCHROEPPEL sobre CFRAC es que un proceso de criba sustituye el proceso de intentos por división. La desventaja es que el método de SCHROEPPEL produce residuos que no necesariamente son cuadráticos.

2.3.9. Criba cuadrática (QS y MPQS).

Carl POMERANCE introdujo una variación a la criba lineal y definió lo que ha dado en llamarse la Criba Cuadrática (popularmente conocido como QS). QS disfruta de la ventaja de la criba lineal y solventa su inconveniente. De modo sumarísimo podríamos decir que QS es como el método de SCHROEPPEL donde se toma en todo momento $A = B$:

Dado $K = \lfloor \sqrt{n} \rfloor$, tomamos $S(A) = (K - A)^2 - n$, donde $S(A) = S(A, A)$.

El método seguido en QS es considerar el polinomio

$$Q(a) = (\lfloor \sqrt{n} \rfloor + a)^2 - n$$

o, lo que es lo mismo, $Q(a) \equiv x^2 \pmod{n}$, donde $x = \lfloor \sqrt{n} \rfloor + a$.

A primera vista, este es un método como otro cualquiera para buscar congruencias. No añade en principio valor alguno al definido con CFRAC. Pero el punto crucial, que marca la verdadera diferencia, es que con QS no es necesario factorizar todos los valores $Q(a)$ sobre la base de factores.

Aquí, $Q(a)$ es un polinomio con coeficientes enteros, que podemos usar para realizar una criba. Veamos cómo funciona: Supongamos cierto número m del que sabemos que $m \mid Q(a)$. Entonces, para cada entero k, se cumple automáticamente que $m \mid Q(a + k \cdot m)$. Para encontrar un valor de a (si existe) para el que se verifique que $m \mid Q(a)$ basta resolver la expresión $x^2 \equiv n \pmod{m}$ y tomando luego el valor $a = x - \lfloor \sqrt{n} \rfloor \bmod m$.

Al hacer la criba, sin peligro de perder generalidad, podemos tomar la restricción con las potencias de factores primos $m = p^k$. Si p es un primo Impar, entonces $x^2 \equiv n \pmod{p^k}$ tiene solución (en concreto dos) si y solo si $(n/p) = +1$. Por tanto, incluimos únicamente en la base de factores a aquellos primos menores que el límite B y que verifiquen esta expresión del símbolo de JACOBI. Y computamos explícitamente los dos valores posibles de $a \pmod{p^k}$ tales que $p^k \mid Q(a)$, y que llamamos a_{p^k} y b_{p^k}.

Si $p = 2$ y $k \geq 3$, entonces $x^2 \equiv n \pmod{2^k}$ tiene una solución (de hecho cuatro) si y solo si $n \equiv 1 \pmod{8}$ y podemos proceder entonces a su cálculo. Finalmente, si $p = 2$ y $k = 2$, tomamos $x = 1$ si $n \equiv 1 \pmod{4}$ (de otra forma, a no existe); y si $p = 2$ y $k = 1$, tomamos $x = 1$.

Tomando un intervalo suficientemente grande (para la criba), calculamos de modo aproximado $\ln|Q(a)|$. Almacenamos estos valores en un vector indexado con a. Ahora, para cada primo p de la base de factores y más genéricamente, para cada pequeña potencia de primo (cuando p es pequeño: es buena regla limitar a valores tales que p^k no superen un determinado límite) vamos restando el valor aproximado de $\ln p$ a cada elemento del vector que sea congruente con a_{p^k} o con b_{p^k} módulo p^k.

Cuando todos los primos de la base de factores han sido utilizados en la criba, queda claro que un valor $Q(a)$ será factorizado en nuestra base de factores si y solo si lo que queda en la posición de índice a de nuestro vector está próximo a cero (si los logaritmos fueran exactos, únicamente si es idénticamente igual a cero). De hecho, si $Q(a)$ no queda completamente factorizado, entonces el valor restante en la posición de índice a será al menos igual a $\ln B$; dado que este valor es bastante mayor que uno, queda justificado entonces que no sea necesario tomar valores muy exactos de los logaritmos.

El polinomio $Q(a)$ introducido antes es bastante útil para el proceso que deseamos realizar, pero desgraciadamente no es útil en la práctica dado que los valores de $Q(a)$ crecen rápidamente, más de lo que desearíamos. La idea (que debemos a Jim DAVIS y a Peter MONTGOMERY) que surge entonces es la de MPQS: criba cuadrática con polinomios múltiples. El nuevo método consiste en usar una serie de polinomios Q de forma que los valores de a vayan permaneciendo lo más pequeños posibles.

Tomamos polinomios cuadráticos de la forma $Q(x) = Ax^2 + 2Bx + C$, donde $A > 0$, $B^2 - AC > 0$ y tales que $n \mid B^2 - AC$. Estos polinomios ofrecen congruencias semejantes a las de antes:

$$AQ(x) = (Ax + b)^2 - (B^2 - AC) \equiv ((Ax + b)^2 (\mathrm{mod}\, n)$$

Además, queremos que los valores de $Q(x)$ sean tan pequeños como podamos en el intervalo de criba. Si queremos cribar en un intervalo de longitud $2M$, es natural que centremos el intervalo en el mínimo de la función Q; por ejemplo, la criba en el intervalo

$$I = [-B/A - M, -B/A + M].$$

Y entonces, para cada $x \in I$, tenemos que $Q(-B/A) \le A(x) \le Q(-B/A + M)$. Por lo tanto, para minimizar el valor absoluto de $Q(x)$ averiguamos si $Q(-B/A) \approx -Q(-B/A + M)$, lo que es equivalente a que $A^2M^2 \approx 2(B^2 - AC)$ y, por tanto

$$A \approx \frac{\sqrt{2(B^2 - AC)}}{M}$$

y entonces tenemos

$$\max_{x \in I} |Q(x)| \approx \frac{B^2 - AC}{A} \approx M\sqrt{(B^2 - AC)/2}$$

Como queremos que sea lo más pequeño posible, pero como también tenemos que $n \mid B^2 - AC$, escogemos entonces A, B y C tal que $B^2 - AC = n$; y el máximo valor para $|Q(x)|$ será aproximadamente igual a $M\sqrt{n/2}$.

Este es un orden de magnitud similar al caso QS (de hecho algo inferior), pero ahora ha quedado añadida la libertad de introducir un nuevo polinomio tan pronto como nuestros residuos alcancen valores demasiado altos.

En resumen: Primero escogemos una longitud de intervalo de criba apropiado, M. Luego tomamos un valor para A cercano a $\sqrt{2n}/M$ tal que A sea primo y $(n/A) = 1$. Luego buscamos un valor para B tal que $B^2 \equiv n (\mathrm{mod}\, A)$, y finalmente buscamos un $C = (B^2 - n)/A$.

Ahora, como en el caso del QS ordinario, debemos computar para cada potencia de primo p^k en nuestra base de factores los valores

$$a_{p^k}(Q) \text{ y } b_{p^k}(Q)$$

con los que iniciamos la criba. Son simples raíces módulo p^k de $Q(a) = 0$. Así, como el

discriminante de Q ha sido escogido igual que n, éstas raíces son iguales que $(-B + a_{p^k})/A$ y $(-B + b_{p^k})/A$, donde

$$a_{p^k} \text{ y } b_{p^k}$$

son las raíces de n módulo p^k que hemos de computar una vez para todo el proceso.

2.3.10. Criba de campo numérico (NFS).

El método más reciente y más poderoso conocido hasta la fecha para factorizar números es el de la criba de campo numérico (NFS). La idea básica es similar a la de QS: mediante un proceso de criba buscamos congruencias módulo n trabajando sobre una base de factores y obteniendo, mediante la técnica de eliminación gaussiana sobre $\mathbb{Z}/n\mathbb{Z}$ una congruencia de cuadrados hasta encontrar la deseada factorización de n.

En algunos aspectos, el algoritmo de factorización NFS es similar a los últimos presentados, pues su objetivo, una vez más, se centra en combinar congruencias hasta lograr una expresión de la forma presentada por FERMAT. Se emplea para enteros que puedan expresarse de la forma

$$n = r^e - s$$

donde r y $|s|$ son enteros positivos pequeños, $r > 1$ y e grande.

El algoritmo ha venido en llamarse de la criba de campo numérico porque su proceso depende de las propiedades aritméticas de un campo numérico seleccionado con adecuadas propiedades algebraicas, combinado con las técnicas tradicionales de criba, como las recogidas para el caso QS. Es, sin duda, el más complejo algoritmo de factorización conocido [Kobl94].

Los requerimientos básicos de este algoritmos pueden resumirse de la siguiente manera: Dado el entero n que se desea factorizar, se elige un grado d y se expresa n como el valor obtenido, a partir del polinomio irreductible de grado d, sobre un determinado entero m:

$$n = f(m) = m^d + a_{d-1} \cdot m^{d-1} + a_{d-2} \cdot m^{d-2} + ... + a_1 \cdot m + a_0$$

donde m y a_k son enteros de un orden $n^{1/d}$. Una vía para encontrar un polinomio así es tomar m como la parte entera de la raíz de grado d de n y obtener posteriormente la expansión de n en la base m. Para un tamaño de entero a factorizar de, por ejemplo, 125 dígitos decimales el valor para d recomendado es 5, de forma que los coeficientes y el mismo valor de m tienen en torno a los 25 dígitos.

El proceso de la criba del campo numérico busca entonces (mediante un proceso de criba similar al descrito para QS) tantos pares (a, b) como sea posible tales que tanto

$$a + b \cdot m$$

y también

$$b^d f(-a/b) = (-a)^d + a_{d-1} \cdot (-a)^{d-1} \cdot b + a_{d-2} \cdot (-a)^{d-2} \cdot b^2 + ... + -a_1 \cdot a \cdot b^{d-1} + a_0 \cdot b^d$$

sean suaves sobre la base de factores dada.

No presentamos ninguna descripción detallada de este último algoritmo; Lo recogido en estas notas ha sido tomado de [Kobl94]. Y como él mismo recomienda, la mejor documentación para el estudio de este algoritmo lo encontramos en el libro [Lens93]. Una buena explicación de los fundamentos básicos matemáticos necesarios para comprender este algoritmo se puede encontrar también en [Kobl98], en [Cohe93] ó en [Mene97].

En [Cava00] se presenta un cuadro con los retos de factorización alcanzados: el último de ellos publicado hasta la fecha un compuesto de 155 dígitos decimales: 512 bits. Con estos algoritmos subexponenciales citados aquí, se ha logrado factorizar una serie de números de tamaños crecientes. El primer logro data de 1970, cuando se factorizó un entero de 39 dígitos decimales mediante la técnica de CFRAC. Cuatro años más tarde se logró factorizar con el mismo método un entero de 50 dígitos. Entre los años 83 y 84 se factorizaron, mediante la técnica QS enteros de tamaños entre 55 y 71 dígitos decimales. Con la mejora de MONTGOMERY, el algoritmo MPQS llegó a resolver, en abril de 1994, el reto RSA–129. Los últimos tres logros de factorización se deben ya a NFS: RSA–130 en Abril de 1996, RSA–140 en Febrero de 1999, y finalmente, en Agosto de 1999 el número RSA–155 que tiene 512 bits.

2.3.11. Curvas elípticas.

Tratamiento matemático aparte merecen los curvas elípticas. Estas curvas han sido largamente estudiadas y existe abundante literatura sobre ellas. Recomendamos especialmente [Kobl94].

Neal KOBLITZ, en [Kobl94], hace una breve presentación de las curvas elípticas. Muestra diferentes criptosistemas basados en esas curvas, entre otros uno análogo al protocolo de intercambio de claves presentado en 1976 por DIFFIE y HELLMAN [Diff76], y otros. Presenta también algunos tests de primalidad fundamentados en esa matemática. Y finalmente muestra la clave del aumento del interés por el estudio de las curvas elípticas entre los investigadores de la criptografía: su uso, ideado por LENSTRA, para obtener un nuevo método de factorización que en algunos aspectos mejora a los ya conocidos. No logra una eficacia práctica mejor que los ya conocidos, pero el hecho de que se haya podido acometer el reto de la factorización desde una nueva perspectiva matemática cuestiona si algún día no dispondremos de caminos de factorización tratables desde el punto de vista de la computación.

Su estudio y su tratamiento matemático nos desvía de nuestro trabajo, centrado más en los algoritmos que basan su técnica en la idea original de FERMAT.

3 CONCEPTOS BÁSICOS DEL RENDIMIENTO DE UN PROCESADOR

Este breve Capítulo presenta un aspecto de nuestro trabajo que resulta esencial en la tesis: algunas consideraciones sobre el rendimiento del procesador, y posibles mejoras que podemos introducir en el código bajo consideraciones del hardware que estemos utilizando. Ya presentaremos en su momento el proceso que hemos seguido para optimizar la implementación del algoritmo de factorización basado en la técnica de las fracciones continuas.

3.1. MEDIDA DEL RENDIMIENTO.

Optimizar un código consiste en modificarlo para lograr que realice la misma tarea en un tiempo menor. Entenderemos que un programa o una función determinada han sido optimizados si se logra que la ejecución se realice en menos tiempo. Es evidente que si se traslada la ejecución del proceso a un ordenador de mejores prestaciones, la velocidad aumentará. Por tanto, cuando hablemos de optimización lo haremos siempre dentro de un mismo hardware.

Entendemos por **tiempo de ejecución** el tiempo de respuesta, es decir, el tiempo trascurrido desde el inicio hasta el final de una tarea Y entendemos por **rendimiento del procesador** el valor inverso al tiempo de ejecución. Una máquina tendrá mayor rendimiento cuanto menor sea su tiempo de respuesta.

El tiempo es la medida del rendimiento de un ordenador. Como señalan David A. PATERSSON y John L. HENNESSY [Patt00], el tiempo puede ser definido de maneras diferentes dependiendo de lo que se cuente. La definición más sencilla de tiempo se llama tiempo de reloj, tiempo de respuesta o tiempo trascurrido: esos tiempos se refieren al tiempo total que tarda una tarea en completarse.

Como a menudo los computadores son de tiempo compartido, y en un procesador pueden trabajar diferentes programas simultáneamente, habrá que distinguir entre el tiempo transcurrido y el tiempo de ejecución de CPU, que es el tiempo que la CPU dedica a ejecutar una tarea concreta. A la vez, el tiempo de CPU puede desdoblarse en el tiempo de la CPU del usuario y el tiempo de la CPU consumido por el sistema operativo (llamado también tiempo de la CPU del sistema). No es sencillo, en el desarrollo de una tarea, discriminar un tiempo de otro. Por tanto, en un trabajo de optimización de código es conveniente mantener siempre la máquina en la que se trabaja, y especialmente el sistema operativo en el que se desarrolla y ejecuta la aplicación a optimizar.

David A. PATERSSON y John L. HENNESSY [Patt00] señalan que todo estudio sobre el tiempo de ejecución ha de tener en cuenta tres factores:

1. Debido a que el compilador genera instrucciones para ejecutar, y la máquina ha de ejecutarlas para que el programa funcione, el tiempo de ejecución debe depender necesariamente del número de instrucciones del programa.
2. La arquitectura del repertorio de instrucciones, que resulta la interface entre la circuitería y los programas de bajo nivel (microinstrucciones). Eso introduce un nuevo concepto, que depende en parte de cada arquitectura, que es el de **ciclos de reloj por instrucción**, y que consiste en el número medio de ciclos de reloj que una instrucción necesita para ejecutarse. El término que recoge este concepto es el de ciclos de reloj por instrucción, y habitualmente se habla de él con el acrónimo **CPI**. Como instrucciones diferentes necesitan diferentes cantidades de ciclos dependiendo de lo que hacen, el CPI es una media de todas las instrucciones ejecutadas por el programa.
3. La frecuencia del reloj, que es el inverso de su periodo.

Con estos tres factores, la expresión presentada en [Patt00] que ofrece una forma de cálculo del tiempo de ejecución es

$$T_{ejecución} = NI \times CPI \times T_{ciclo} \quad (3)$$

Hay que tener en cuenta que la única medida completa y fiable del rendimiento de un

computador es el tiempo. Cambiar el código hasta llegar a otro programa con menor número de instrucciones no necesariamente implica reducir el tiempo de ejecución, pues se puede llegar a un conjunto menor de instrucciones pero con un valor de CPI mayor.

Con los ordenadores superescalares ha perdido interés el concepto y el valor del parámetro CPI. Actualmente se trabaja y se estudia con el valor inverso, llamado **IPC** (**instrucciones por ciclo**), que recoge el número de instrucciones que se ejecutan en cada ciclo de reloj. Cuando se logran valores de IPC mayores que 1 entonces tenemos que se están ejecutando más de una instrucción por ciclo de reloj y podemos decir que hemos logrado entrar en régimen de paralelismo en el microprocesador. IPC mayor que uno indica paralelismo.

3.2. LEY DE AMDHAL.

En todo proceso de optimización se debe tener en cuenta la **ley de AMDHAL**. Esta ley señala que cuando realizamos una mejora en una función de una determinada aplicación, el peso de la mejora es proporcional al peso que tenía esa función, antes de ser optimizada, en el total de la aplicación. Si optimizamos un código que supone el 1% del tiempo de ejecución del programa entero, entonces lo máximo que podemos lograr es reducir en un 1% el tiempo total del programa.

Aunque esta ley es muy evidente, es conveniente recordar su enunciado pues con facilidad se pierde esta perspectiva. No vale la pena mejorar lo que apenas tiene peso en el proceso.

Hay que tener en cuenta que al ir reduciendo los tiempos de diferentes partes o bloques del programa, ocurre a veces que otro bloque que antes no tenía peso significativo en el proceso adquiere ahora suficiente importancia por haber reducido el tiempo en los previamente más onerosos. Por eso el proceso de optimización se debe realizar de modo gradual e iterativo: el peso en cada momento debe ir señalando qué bloques se deben optimizar.

3.3. VÍAS DE MEJORA.

Las dos vías principales para mejorar (reducir) el tiempo de ejecución de un proceso son:

1. Análisis del software del algoritmo. A este proceso lo llamamos optimización software.
2. Estudio de cómo interactúa el código compilado y la máquina. A este proceso lo llamamos optimización hardware.

El primero de los dos pasos consiste en buscar algoritmos mejores que el utilizado. Podemos encontrar algoritmos que realicen nuestro proceso en un tiempo mucho menor. En ese caso no hay que optimizar el algoritmo usado sino cambiarlo por el algoritmo mejor. Cuando se procede a una optimización de código es conveniente no perder de vista este primer paso: ¿Existe un algoritmo más rápido que realice la misma tarea que el actual que deseo optimizar?

Una vez hecha la mejora software, el trabajo de optimización consiste ahora en mejorar el código teniendo en cuenta el hardware de la máquina. Para este proceso de optimización trabajaremos siempre de la siguiente manera:

1. Buscar instrucciones que puedan ser sencillamente eliminadas. Cualquier modificación del código que reduzca el número de instrucciones es buena. Reducir el número de instrucciones es siempre un paso conveniente. Bien entendido (ya lo hemos dicho antes) que esa reducción se logra mediante eliminación y no en un proceso de cambio de algoritmo, que logre menos instrucciones pero no necesariamente con igual o menor CPI. Si la reducción de instrucciones se logra por cambio de algoritmo, y además llegamos a un valor de CPI menor, entonces simplemente hemos cambiado el algoritmo por otro mejor y hemos realizado lo que antes hemos llamado análisis del software del algoritmo.

   Puede ocurrir que al reducir el número de instrucciones por eliminación quede aumentado el valor de CPI, puesto que el valor de CPI es un valor promedio. Pero es claro que si se han reducido las instrucciones y no se ha realizado otra modificación que la de eliminar, el tiempo final deberá ser siempre menor.

2. Buscar optimizaciones basadas en la iteración del hardware con el programa que estamos procurando optimizar. Para este proceso de optimización por hardware estudiaremos los siguientes factores: (a) **Riesgos**. Operaciones que llevan consigo una detención. Los riesgos pueden ser de tres tipos: estructurales, de control o de datos. (b) Accesos a la memoria.

3.4. RIESGOS ESTRUCTURALES Y DE CONTROL.

Una forma de mejorar el tiempo de ejecución de un programa es evitar o reducir sus riesgos.

Los **riesgos estructurales** son ocasionados por las limitaciones físicas del procesador y no pueden ser resueltos en un trabajo de optimización de software. De todas formas, como veremos, no todo se arregla aumentando circuitería. No entraremos a un análisis sobre este aspecto de las optimizaciones. Se podría hacer algún intento de modificación del código atendiendo a estos riesgos. Por ejemplo, cambiar algunos tipos de datos, utilizando siempre que se pudiera más variables tipo `float` de cara a utilizar las unidades funcionales en coma flotante que quizá estaban poco utilizadas en la implementación original.

Tenemos un riesgo de datos estructural cuando la circuitería no puede soportar la combinación de instrucciones que se quiere ejecutar en el mismo ciclo. Los problemas estructurales se pueden paliar haciendo "emisión fuera de orden". De esa tarea se encarga el procesador. Busca instrucciones fuera del orden establecido por la secuencia del programa. Al haber logrado previamente, mediante el desenrollado de los bucles, llenar la **ventana de instrucciones** (buffer donde se almacenan las instrucciones, previas a su ejecución), podemos ahora seleccionar entre todas las instrucciones pendientes aquellas que realmente pueden ser ejecutadas en algún cauce

de ejecución libre.

Los **riesgos de control, o riesgos de salto**, surgen de la necesidad de tomar una decisión basada en los resultados de una instrucción mientras otras se están ejecutando. Tienen que ver con los saltos: los bucles y los estructuras condicionales. Siempre que el procesador se enfrenta a una decisión a resolver, tenemos una detención. Y eso produce un decremento en el IPC.

Actualmente todos los procesadores superescalares presentan en su arquitectura un **predictor de saltos**. En el caso de que no lo tenga, cuando llega a una instrucción de salto, se pierden varios ciclos de reloj en calcular cuáles serán las siguientes instrucciones a ejecutar. En cambio, el predictor predice cuáles serán esas instrucciones. Se hace entonces lo que se llama "ejecución especulativa": ejecuta las instrucciones que cree deberán ser las seleccionadas en el salto. Si el predictor decide equivocadamente, el procesador deberá recuperar el estado de todas las variables tal como estaban antes de la sentencia de salto mal predicha. Si el predictor es bueno, entonces compensa su uso. Actualmente los predictores de saltos tienen entre un 93 y un 95 por ciento de aciertos.

Llamamos predictor Taken a un predictor que toma siempre la opción de salto como la verdadera. Para un programa repleto de sentencias **`if`** ese predictor será muy poco eficiente. Es, sin embargo, un buen predictor en un programa repleto de sentencias iterativas (**`for`**, **`while`** y **`do – while`**). Llamamos predictor No Taken al predictor que decide siempre la opción de salto como la falsa. Es evidente que resulta útil conocer el tipo de predictor con el que se trabaja a la hora de expresar las condiciones de permanencia en las estructuras de control.

Gran parte de los problemas de los saltos quedan salvados gracias a los predictores. De hecho esto posibilita que, cada vez que haya un acierto no haya detenciones.

El problema principal de los bucles, una vez eliminado el problema de la decisión de la condición mediante los predictores, es el de la gestión de las instrucciones. Lo primero que hace el procesador es emitir las instrucciones. Del código máquina se emiten hacia la ventana de instrucciones. Desde la ventana de instrucciones éstas pasan a ser ejecutadas. Si deseamos alcanzar un valor de IPC = 3 es necesario disponer de una emisión de instrucciones de al menos 3 instrucciones por ciclo. Supongamos que ese no es nuestro problema, y que incluso disponemos de mayor velocidad de emisión. Nuestro principal problema a la hora de llenar la ventana de instrucciones lo encontramos en las instrucciones de salto: instrucciones que impiden que la transferencia de datos sea todo lo rápida que podría atender el ejecutor de instrucciones. Especialmente se nota cuando el bucle es pequeño.

Supongamos un bucle de 4 instrucciones. En ese caso, el emisor de instrucciones envía a la ventana tres en el primer ciclo. El segundo ciclo sólo envía la cuarta. En el tercer ciclo sólo envía la instrucción condicional que decide el mantenimiento o no en el bucle. Así tenemos que la ventana no se llena y podemos llegar a tenerla vacía.

Es muy conveniente lograr llenar por completo la ventana de instrucciones de forma que este primer problema de transferencia de instrucciones desaparezca. Y una forma de lograrlo consiste en aplicar la técnica del "desenrollado" de bucles. Como el mismo nombre sugiere, el desenrollado de bucles se logra realizando múltiples copias del cuerpo del bucle y planificando juntas instrucciones que pertenecían a iteraciones diferentes. El desenrollado del bucle elimina saltos en el proceso. Cuando el bucle es grande es fácil lograr llenar la ventana de instrucciones.

3.5. RIESGOS DE DATOS.

El siguiente escollo con que podemos encontrarnos está ahora en que tengamos dificultades para que la ventana de instrucciones pase a ejecución al menor tres instrucciones en cada ciclo.

Y eso con frecuencia es un escollo real. La causa principal de esta dificultad son las dependencias de datos.

A veces no podemos ejecutar instrucciones porque podemos tener que las instrucciones posteriores dependen de resultados de instrucciones previas.

Existen tres tipos de dependencias de datos:

1. **Dependencia RAW** (read – after – write):

   Son instrucciones de la forma:

   ```
   r2 = a + b;          (I1)
   r3 = c * r2;         (I2)
   ```

   Leemos la variable `r2` después de haberla escrito. No podemos ejecutar I2 sin haber ejecutado previamente I1 y haber obtenido así el valor de `r2` que necesita I2.

2. **Dependencia WAR** (write – after – read):

   La inversa de RAW. Son instrucciones de la forma:

   ```
   a = r2 + b;          (I1)
   r2 = c * r3;         (I2)
   ```

   Leemos la variable `r2` y después variamos su valor. Esta dependencia nos impide ejecutar I2 fuera de orden.

3. **Dependencia WAW** (write – after – write): Escribimos y después volvemos a escribir:

   ```
   r2 = a + b;          (I1)
   (otras instrucciones)
   r2 = c * d;          (I2)
   ```

Al igual que en el caso WAR, si alteramos el orden de estas instrucciones al final tendremos que el valor de `r2` es el señalado en I1, cuando el que debe quedar recogido es el valor calculado en I2.

Siguiendo la secuencia de posibilidades, podríamos hablar de la **dependencia RAR** (read – after – read). Esta no tiene problema ninguno.

```
a = r2 + b;        (I1)
c = r2 * d;        (I2)
```

Ambas instrucciones requieren el valor de `r2`, pero éste no sufre variación en ningún momento. Por tanto esta dependencia no lo es en realidad.

Las dependencias WAR y WAW tienen una solución sencilla si declaramos más variables. Son falsas dependencias y no requieren más estudio que su eliminación mediante la introducción de nuevas variables.

Llamamos, por tanto, **dependencia de datos**, a la dependencia de tipo RAW. Las dos formas de resolver la dependencia RAW son:

1. Movimiento o reordenamiento de código. Si tenemos dos sentencias I1 e I2 con variables dependientes, puedo paliar su efecto de freno de transferencia de instrucciones mediante la inserción entre ambas sentencias de otras que no incurran en esa dependencia. Este procedimiento es útil, aunque ya lo intenta realizar el procesador. Pero se trata ahora de considerar que la ventana de instrucciones es limitada en su capacidad y podemos facilitar, mediante modificación del orden de las instrucciones, que siempre haya algunas libres de dependencia disponibles.
2. **Segmentación de software**. Esta técnica se usa cuando se tienen dependencias dentro de un bucle. Hay que tener en cuenta que esas dependencias pueden estar en iteraciones anteriores.

3.5.1. Accesos a memoria.

La memoria es el lugar físico donde se guardan los programas mientras se ejecutan. También contiene los datos requeridos para la ejecución. En los años 60 y 70 la restricción principal de los ordenadores y de su modo de trabajo era el tamaño de la memoria. El objetivo era minimizar el espacio en memoria para lograr hacer el programa más rápido. Esto ha cambiado y en la actualidad para ganar velocidad en los programas se hace conveniente conocer la naturaleza jerárquica de la memoria y la naturaleza paralela de los procesadores.

Habitualmente nos encontramos con dos tipos de memoria en un ordenador: la memoria principal DRAM o memoria dinámica de acceso aleatorio, y la memoria caché. La memoria principal tiene

la característica de que los accesos a ella toman el mismo tiempo independientemente de la posición de memoria a la que se acceda. La **memoria caché** es una memoria pequeña y rápida que actúa de buffer para la memoria DRAM.

Desde el principio de la era de la informática se ha deseado poder disponer de una memoria rápida e infinita en su capacidad. Y existe un modo de trabajar con ella que facilita esa visión de la memoria. Este modo de trabajar se basa en el llamado **principio de localidad**, que afirma que en un momento concreto los programas acceden a una parte relativamente pequeña de su espacio de direcciones.

Existen dos tipos de localidad:

1. **Localidad temporal**: si se consulta un dato, seguramente será consultado próximamente.
2. **Localidad espacial**: si se consulta un dato, seguramente se consultarán otros datos cercanos a él. Como normalmente se accede a las instrucciones secuencialmente, los programas muestran mucha localidad espacial: por ejemplo, los accesos a los elementos de un vector.

La localidad espacial ha sugerido la creación y diseño de la memoria de forma jerárquica. Una jerarquía de memoria consiste en múltiples niveles de memoria con diferentes velocidades y capacidad. Las memorias se construyen con jerarquías de niveles: más cercanas las más rápidas (que son también las más caras) y más lejanas las más lentas (que son también las más baratas). A las más cercanas son a las que llamamos memoria caché: caché fue el término escogido para representar el nivel de la jerarquía de memorias entre la CPU y la memoria principal.

Todos los datos se guardan en el nivel más bajo o lejano. Los niveles tienen un tiempo de acceso mayor cuanto más lejos del procesador estén. Entre niveles se transfieren bloques de información. Los datos se transfieren solo entre niveles adyacentes.

Diremos que hemos tenido un **acierto** si los datos que pide el procesador aparecen en algún bloque del nivel superior. Diremos que hemos tenido un **fallo** de acceso a memoria si los datos no se encuentran en el nivel superior. En este caso se debe acceder al nivel inferior para leer el bloque que contiene los datos deseados.

Por razón del rendimiento, la velocidad de los aciertos y de los fallos es importante. El tiempo de acierto es el tiempo necesario para acceder al nivel superior de la jerarquía. La penalización del fallo es el tiempo necesario para reemplazar un bloque del nivel superior por el correspondiente bloque del nivel inferior mas el tiempo de suministrar ese bloque al procesador.

3.5.1.1. Cómo aprovechar la localidad espacial.

Para la gestión de fallos, la propuesta básica es parar la CPU, congelando los valores de todos los registros. Un controlador separado se encarga del fallo de caché, buscando los datos en la memoria. Una vez que los datos están presentes, la ejecución se reanuda en el ciclo que ha causado el fallo de caché. Así, el procesamiento de un fallo de caché provoca un bloqueo del

procesador.

Existen modos de paliar el retardo que ocasiona un fallo de acceso a memoria. Uno de ellos es aprovechar la localidad espacial de los accesos a memoria. Para ello es bueno tener un bloque de caché mayor que una palabra. En caso de fallo, se buscarán múltiples palabras que sean adyacentes y conlleven una alta probabilidad de ser referenciadas en breve. El motivo del incremento del tamaño del bloque es sacar partido a la localidad espacial para incrementar el rendimiento. En general la tasa de fallo desciende cuando se incrementa el tamaño del bloque.

La tasa de fallos podría crecer si el tamaño del bloque es una parte significativa del tamaño de la caché porque el número de bloques que pueden estar en la caché sería entonces pequeño y, por tanto, habría una fuerte competencia entre esos bloques.

El mayor inconveniente de aumentar el tamaño del bloque es que la penalización por fallo crece.

3.5.1.2. Cómo medir y mejorar el rendimiento de la caché.

Presentamos dos técnicas para reducir la tasa de fallos:

1. Reducir la tasa de fallos reduciendo la posibilidad de que dos bloques diferentes de memoria vayan a la misma posición de la caché.
2. Reducir la penalización por fallo añadiendo un nivel adicional en la jerarquía: caché multinivel. Podemos hablar habitualmente de dos niveles de caché: la memoria L1 más cercana, y la memoria L2.

El tiempo de la CPU se puede dividir entre el número de ciclos en los que la CPU ejecuta el programa y el número de ciclos en los que la CPU espera la memoria. Normalmente se supone que el coste de los accesos a la caché que son aciertos son parte de los ciclos de ejecución normal de la CPU. Por tanto

$$\text{Tiempo de CPU} = \left(\text{Ciclos de ejecución de la CPU} + \text{Ciclos de bloqueo por memoria}\right) * \text{Tiempo de ciclo}$$

Los ciclos de bloqueo causados por memoria son principalmente debidos a fallos de caché, y así se puede asumir en adelante.

$$\text{Ciclos de bloqueo por memoria} = \text{Ciclos de bloqueo por lectura} + \text{Ciclos de bloqueo por escritura}$$

$$\text{Ciclos de bloqueo por lectura} = \frac{\text{Lecturas}}{\text{Programa}} * \text{Tasa de fallos por lectura} * \text{Penalización por fallos de lectura}$$

Despreciando las penalizaciones por escritura en buffer tenemos que los ciclos de bloqueo por escritura son

$$\text{Ciclos de bloqueo por escritura} = \frac{\text{Escrituras}}{\text{Programa}} * \text{Tasa de fallos por escritura} * \text{Penalización por fallos de escritura}$$

Y los ciclos de bloqueo por memoria nos quedan entonces

$$\text{Ciclos de bloqueo por memoria} = \frac{\text{Accesos a memoria}}{\text{Programa}} * \text{Tasa de fallos} * \text{Penalización por fallo}$$

1. Cuanto más bajo sea el CPI, más pronunciado será el efecto de los ciclos de bloqueo.
2. El sistema de memoria es improbable que mejore tan rápidamente como el tiempo de ciclo del procesador. Cuando se calcula el CPI, la penalización por fallo se mide en ciclos de CPU gastados por fallo. Por tanto, si las memorias principales de dos máquinas tienen el mismo tiempo absoluto de acceso, una CPU con un reloj más rápido tendrá una penalización por fallo más larga.

3.5.1.3. Algunas técnicas de optimización de los accesos a memoria.

⇒ **Prefetching**: Cuando un procesador necesita una variable lo que hace es buscarla en memoria. Antes (hasta el año 1994), la memoria (como hemos visto) era bloqueante. A partir de 1994 aparecieron los primeros procesadores con buffers de cargas y almacenamientos. Estas dos operaciones no bloquean al resto de operaciones, y el procesador puede seguir realizando instrucciones mientras se espera la llegada de algunos datos. Evidentemente, esto exige que las operaciones que ejecute mientras se espera no requieran la información que está llegando.

El caso es que no hay tanto buffer de memoria, y no es fácil tener muchas instrucciones en vuelo de carga–almacenamiento. Además, si disponemos de pocas instrucciones de otra naturaleza que no sea carga–almacenamiento, al final el proceso se queda bloqueado en la espera de datos. Y aquí es donde entra la tarea del prefetching.

Cada vez que el procesador acude a memoria L1 y no halla en ella la información que requiere pasa a buscar el L2, y si no, en la memoria principal. Cada vez que acude a la memoria principal, aprovecha el proceso y se trae una cierta cantidad de bytes de información siguientes.

Declarando bien las variables ya podemos ganar tiempo. Habitualmente el procesador almacena de modo consecutivo las variables que han sido declaradas de forme consecutiva. Y ya se sabe que los arrays se almacenan linealmente.

Lo que hace el prefetching es buscar las variables antes de su uso. Y se hace con algunas que se supone que estarán fuera de las líneas de caché. El prefetching no reduce los fallos de caché: pero enmascara el tiempo que invierte en acudir a esa memoria. El proceso será bueno si acertamos en los momentos en los que es conveniente hacer el prefetching.

⇒ Mezcla de vectores: Esta y las siguientes tareas lo que pretenden es explotar la localidad espacial de las variables en la memoria. Se trata de trabajar varios vectores en un mismo bucle. Se trata de unir variables.

Este método tiene interés en el caso en el que no se recorra el vector elemento a elemento. En ese caso, podríamos mezclar vectores para poder traer (cada vez que se vaya a memoria principal) información toda ella útil.

⇒ Intercambio de bucles: A veces podemos estar recorriendo los bucles de forma inversa a como está almacenado en memoria. Una vez más el objetivo es que cada vez que tenga un fallo de caché trabajar de forma que todo lo que me traiga a L1 sea información que utilizaré de forma inmediata.

⇒ Fusión de bucles: Fundir dos bucles en uno solo. Esto es útil cuando tengo dos bucles que siguen el trazo sobre el mismo vector.

⇒ **Blocking**: Trabajar por trozos o bloques. Para aprovechar la localidad espacial de la memoria. El objetivo es hacer trozos que quepan en la caché y que guarden entre sí todas las relaciones necesarias para las operaciones que deseo realizar.

⇒ Variables **`register`**: Quedan recogidas en el registro de la ALU.

⇒ Variables **`volative`**: En ningún caso se escriben en la L1.

4 UNA NUEVA IMPLEMENTACIÓN DE ENTERO LARGO PARA PROCESOS DE FACTORIZACIÓN.

En muchas de las implementaciones de herramientas criptográficas se hace necesario manejar enteros de gran tamaño: de tamaños mayores que los admitidos en los tipos de datos estándares de muchos de los lenguajes de programación. Y estos tamaños exigen una definición de entero largo: una estructura que almacene su valor en memoria, un dominio de valores delimitado, y unos operadores básicos para manejar esos enteros.

Los números que intervienen en las matemáticas y en los algoritmos diseñados para la criptografía de clave pública —por ejemplo, para las operaciones definidas para el criptosistema RSA— llegan a tener fácilmente más de 1000 dígitos decimales. Los procesos de factorización de un entero de varios cientos de bits también requieren del manejo de valores numéricos de gran cantidad de dígitos. Es, por tanto, un requisito previo a cualquier trabajo en criptografía, la implementación de las librerías necesarias para poder manejar esos valores numéricos.

El diseño e implementación de una librería de multiprecisión (para el manejo de enteros largos), no

es una tarea complicada, pero sí, desde luego, laboriosa. Indicaciones precisas sobre cómo debe realizarse esa implementación se encuentran en [Knut81] y en [Ries87]. Ambas referencias vienen recomendadas en [Cohe93] como las más apropiadas para la orientación en este trabajo.

Las matemáticas que sustentan esta tarea de diseño e implementación vienen presentadas en un primer epígrafe de este Capítulo. Allí presentamos unas nociones básicas sobre el clásico modelo de numeración y sobre las operaciones aritméticas básicas. Como señala Henri COHEN en su libro [Cohe93], a un sistema numérico diseñado para ser implementado en un computador le conviene una base de numeración distinta a la acostumbrada base 10. Habitualmente trabajaremos en aquellas bases que mejor permitan las arquitecturas de los ordenadores para los que hagamos la implementación: base binaria u otra base potencia de dos.

Las operaciones básicas que debe tener una implementación de un entero largo, según sigue diciendo Henri COHEN, son la suma y resta de dos enteros de longitud larga; el producto y la división de un entero largo con un entero corto (entendiendo aquí por entero corto aquel que queda completamente codificado en el espacio de memoria que reserva una variable estándar en el lenguaje en el que estemos trabajando); los desplazamientos de bits dentro de un entero largo, tanto hacia la izquierda como hacia la derecha; y, desde luego, las operaciones de entrada de datos por teclado y de salida de datos por pantalla.

La implementación de una librería para enteros de precisión múltiple exige un trabajo previo de estudio y análisis. Como recomienda Hans RIESEL en [Ries87], antes de comenzar la implementación de un paquete de multiprecisión, se deben tener en cuenta cuáles son los objetivos que se desean lograr al implementar toda la librería y el conjunto de sus funciones. En nuestro caso, hemos pretendido una variable nueva que tuviera validez general para cualquier operación que requiera la criptografía asimétrica. Pero hemos tenido especialmente en cuenta el objetivo de lograr una implementación de entero largo válida para el diseño del algoritmo de factorización basado en la técnica de las fracciones continuas.

Una vez mostrada toda la base matemática, tratamos, en un segundo epígrafe, sobre el modo en que puede codificarse un número para ser implementado en un computador. Mostramos algunos modelos ya existentes y que, pensamos, constituyen referencias obligadas en un trabajo de este tipo; desde luego, existen otros muchos modos de afrontar y resolver esta tarea de la que ahora tratamos: cualquier herramienta de precisión múltiple (en el Capítulo 7 recogemos una lista extensa) requiere su propia definición. Y presentamos a continuación nuestra propia definición de entero largo; también hacemos un breve estudio comparativo de cada uno de los modelos con el nuestro, y recogemos las razones que justifican esa forma de entero largo y el porqué no hemos tomado una implementación ya creada.

Finalmente mostramos cada una de las funciones implementadas en nuestra librería, y que hemos clasificado en dos grupos: las funciones auxiliares (entre las que se cuentan las que codifican los

operadores relacionales y los operadores a nivel de bit) y las funciones aritméticas. También recogemos en un cuarto epígrafe algunas implementaciones de funciones algo más complejas, como el algoritmo de EUCLIDES, el cálculo de la parte entera de la raíz cuadrada de un entero largo y el cálculo de potencias. Terminamos la presentación de las funciones básicas implementadas con las funciones necesarias para los tests de primalidad: la obtención de los valores de los símbolos de LEGENDRE y de JACOBI entre dos enteros largos y el test de primalidad de MILLER–RABIN.

En las sucesivas implementaciones que hemos ido realizando, se ha ido refinando la estructura de nuestro nuevo tipo de dato. También sus operadores han evolucionado a versiones cada vez más sencillas, simplificadas y eficientes. Cualquier mejora, por pequeña que resulte, supone un importante avance en el rendimiento total del proceso que pretendemos ejecutar, y que iniciamos con la implementación de esta librería. Del acierto en la definición de la noción de número, de su correcta codificación y del desarrollo de todos los operadores, depende, en gran manera, la mejora en la eficiencia de los distintos algoritmos criptográficos y de criptoanálisis implementados. El Capítulo 7 de la tesis muestra el trabajo realizado para lograr reducir el tiempo de ejecución de muchas de las funciones que presentaremos en este Capítulo.

4.1. FUNDAMENTOS MATEMÁTICOS PARA UN SISTEMA NUMERICO BÁSICO.

4.1.1. Nociones matemáticas para la correcta definición de un modelo de entero.

Todo número viene expresado en una **base** $B>1$. En cada base, todo número entero $a>0$ puede ser escrito de modo único en la forma

$$a = a_k \cdot B^k + a_{k-1} \cdot B^{k-1} + a_{k-2} \cdot B^{k-2} + ... + a_1 \cdot B^1 + a_0 \quad (1)$$

donde $k>0$ es un entero, y cada uno de los a_i son enteros que verifican

$$0 \le a_i \le B-1 \text{, para } j = 1,2,3,...,k \text{, y } a_k \neq 0 . \quad (2)$$

A los coeficientes a_i se les llama **dígitos del número** a. A la expresión (1) se la llama **expansión del número**. El número habitualmente se representa como

$$a = (a_k a_{k-1} a_{k-2} ... a_1 a_0)_B \quad (3)$$

Cualquier número viene representado en una base determinada, por una serie única de coeficientes a_i (ver 3). A cada serie de coeficientes, que codifica un número de modo único en

una determinada base, se le llama **cifra**. En una cifra importa tanto la posición relativa de cada dígito dentro de ella, como el valor de cada uno de esos dígitos. Cuanto más larga pueda ser la serie de dígitos, mayor será el rango de números que podrán ser representados. Como se sabe, y como se desprende de esta codificación, todo cero a la izquierda de estos dígitos supone un nuevo dígito que no aporta valor alguno a la cantidad codificada.

La expansión del número recoge el valor de cada dígito y su peso dentro de la cifra. El dígito a_0 del número a puede tomar cualquier valor comprendido entre 0 y $B-1$. Cuando se necesita codificar un número mayor o igual que el cardinal de la base (B) se requiere un segundo dígito a_1, que también puede tomar sucesivamente todos los valores comprendidos entre 0 y $B-1$. Cada vez que el dígito a_0 debiera superar el valor $B-1$ vuelve a tomar el valor inicial 0 y se incrementa en uno el dígito a_1. Cuando el dígito a_1 necesita superar el valor $B-1$ se hace necesario introducir un tercer dígito a_2 en la cifra, que también podrá tomar sucesivamente todos los valores comprendidos entre 0 y $B-1$, incrementándose en uno cada vez que el dígito a_1 debiera superar el valor $B-1$. El dígito a_1 "contabiliza" el número de veces que a_0 alcanza en sus incrementos el valor superior a $B-1$. El dígito a_2 "contabiliza" el número de veces que a_1 alcanza en sus incrementos el valor superior a $B-1$. Por tanto, el incremento en uno del dígito a_1 supone B incrementos del dígito a_0. El incremento en uno del dígito a_2 supone B incrementos del dígito a_1, lo que a su vez supone B^2 incrementos de a_0. Y así, sucesivamente, el incremento del dígito a_j exige B^j incrementos en a_0.

Podríamos decir que cada dígito está sujeto a las restricciones de la aritmética modular con módulo la base de codificación o representación de los números. Y que todos los dígitos posteriores a la posición j codifican el número de veces que el dígito j ha recorrido de forma completa todos los valores comprendidos entre 0 y $B-1$.

4.1.2. Nociones matemáticas para el desarrollo de algunas operaciones aritméticas con enteros.

Ya tenemos definido un modelo matemático para el manejo de cantidades enteras de longitud indefinida. Todo sistema numérico para la representación de enteros deberá poder codificar cantidades de longitud indefinida, porque así son de hecho los números que queremos codificar nosotros: tan grandes como queramos.

Queremos ahora analizar de forma breve las principales operaciones aritméticas. Ya hemos señalado que Henri COHEN (como se sabe, autor principal de la herramienta PARI, para cálculos de multiprecisión) señala esas funciones como las más básicas a la hora de implementar un tipo de dato para el manejo de enteros de longitud indefinida.

Los conceptos matemáticos que presentamos a continuación son tomados de la experiencia universal de cualquiera que haya realizado estas operaciones con papel y lápiz. COHEN dice en

su libro ya citado [Cohe93] que este modo de trabajo (el del papel y el del lápiz) es de sencilla implementación. Y señala también que, a la hora de programar para un ordenador todo este método operacional aprendido de niños, existe un condicionante que impone la propia arquitectura de los ordenadores, arquitectura que hace recomendable no trabajar en la clásica base 10. Pero eso será cuestión a tratar más adelante.

Presentamos únicamente una descripción teórica de los procesos de la suma y de la resta. El producto se reduce a una secuencia de sumas y de desplazamientos a izquierda; la división se reduce a una serie de restas y otros procesos auxiliares.

4.1.2.1. Aspectos teóricos de la operación para la suma.

Para realizar la suma de dos enteros ($z = x + y$) cuyas cifras vienen codificadas en la forma $x = (x_s, x_{s-1}, ..., x_1, x_0)_B$ e $y = (y_t, y_{t-1}, ..., y_1, y_0)_B$, hemos de ir calculando los valores de los sucesivos dígitos z_i de z.

Las dos cuestiones a resolver a la hora de realizar una suma son: determinar cuántos dígitos tendrá el resultado a partir del número de dígitos de cada uno de los dos sumandos; y determinar a partir de qué valores vamos obteniendo cada uno de los dígitos z_i de z.

A la vista de la expansión del número, se entiende que la suma de dos cifras se realiza mediante las sucesivas sumas de los dígitos del mismo peso. Y que esa suma dígito a dígito se realiza dentro del marco de la aritmética modular módulo B. Y que de alguna manera hay que tener en cuenta que si el valor de un dígito de la suma necesitase superar el valor máximo $B-1$ se requeriría tomar el valor $\bmod B$ e incrementar en 1 el valor del dígito siguiente: es lo que se conoce como el **acarreo**.

Si igualamos el número de dígitos de las dos cifras a sumar (añadiendo tantos ceros a la izquierda al menor de los dos sumandos como sean necesarios para igualar número de dígitos), tenemos:

$$\begin{aligned}&(x_k \cdot B^k + x_{k-1} \cdot B^{k-1} + ... + x_1 \cdot B^1 + x_0) + (y_k \cdot B^k + y_{k-1} \cdot B^{k-1} + ... + y_1 \cdot B^1 + y_0) = \\ &((x_k + y_k) \cdot B^k + (x_{k-1} + y_{k-1}) \cdot B^{k-1} + ... + (x_1 + y_1) \cdot B^1 + (x_0 + y_0))\end{aligned} \quad (4)$$

Tomemos el primer elemento de la expansión (4): $(x_0 + y_0)$. Si se verifica que $(x_0 + y_0) \geq B$, entonces podemos expresar ese término de la potencia B^0 como

$$(x_0 + y_0) \cdot B^0 = (z_0 + B) \cdot B^0 = z_0 \cdot B^0 + 1 \cdot B^1$$

Vemos por tanto que el dígito correspondiente al peso B^0 será z_0, y se deberá incrementar en uno el valor del dígito correspondiente al peso B^1. También se comprende de forma inmediata que el valor de $z_0 \equiv (x_0 + y_0) \bmod B$.

Para cualquier otro dígito z_j de la suma, tenemos una expresión semejante. En todos los demás casos, donde $j \neq 0$, tenemos que considerar la posibilidad de que de la suma de los dígitos previos haya quedado pendiente sumar un 1 (el del acarreo) en el actual peso. Llamamos C_{j-1} al

acarreo de la suma de los dígitos previos: valdrá 0 ó 1 dependiendo de que esa suma previa de los dígitos haya superado el valor $B-1$.

Si $(x_j + y_j + C_{j-1}) \geq B$ entonces

$$(x_j + y_j + C_{j-1}) \cdot B^j = (z_j + C_j \cdot B) \cdot B^j = z_j \cdot B^j + C_j \cdot B^{j+1}, \text{ con } C_j = 1$$

Por tanto, las dos cuestiones planteadas se resuelven de la siguiente manera: El resultado tendrá tantos dígitos como tenga el mayor de los dos sumandos (k), más uno si lo exige la existencia de un acarreo C_k en la suma de los dígitos de mayor peso de los sumandos. Y el valor de cada dígito del resultado será la suma (en aritmética modular modulo B) de los dígitos del mismo peso de cada uno de los sumandos más el acarreo de la suma de los dígitos previos:

$$z_j = (x_j + y_j + C_{j-1}) (\mathrm{mod}\, B)$$

$C_{j-1} = 1$ si se verifica que $x_{j-1} + y_{j-1} + C_{j-2} \geq B$

4.1.2.2. Aspectos teóricos de la operación para la resta.

En el desarrollo realizado sobre las operaciones de aritmética modular hemos considerado siempre que la diferencia se realiza de tal manera que el minuendo es mayor que el sustrayendo. De no ser así, bastará tener en consideración que $x - y = -(y - x)$: por tanto el proceso que aquí describimos para la resta es igualmente válido si tuviéramos que $y > x$: al final bastará hacer la consideración de que el resultado obtenido es el correcto cambiado de signo.

Al igual que hemos hecho con la operación suma, podemos desarrollar la expansión de la diferencia $z = x - y$:

$$\begin{aligned}&(x_k \cdot B^k + x_{k-1} \cdot B^{k-1} + ... + x_1 \cdot B^1 + x_0) - (y_k \cdot B^k + y_{k-1} \cdot B^{k-1} + ... + y_1 \cdot B^1 + y_0) = \\ &= ((x_k - y_k) \cdot B^k + (x_{k-1} - y_{k-1}) \cdot B^{k-1} + ... + (x_1 - y_1) \cdot B^1 + (x_0 - y_0))\end{aligned} \quad (5)$$

Veamos el primer elemento de la expansión (5): $(x_0 - y_0)$. Siempre se cumplirá que

$$\left|(x_0 - y_0)\right| \leq B \quad (6)$$

El caso en que se cumple que $x_0 > y_0$ es inmediato: el cálculo de z_0 no es más que la diferencia entre los dos valores: $z_0 = x_0 - y_0$.

Otro análisis exige la situación en que $x_0 < y_0$ y por tanto $z_0 = x_0 - y_0 < 0$.

En el sistema de codificación de enteros que hemos presentado hemos visto que todos los dígitos de una cifra toman valores entre el cero y el entero inmediatamente anterior a la base de numeración en la que trabajamos: no podemos expresar cifras con dígitos menores que cero. Para esos casos podemos hacer la siguiente operación

$$B^1 + (x_0 - y_0) \cdot B^0 = (B - d_0) \cdot B^0 = z_0 \cdot B^0 \text{ , donde } d_0 > 0 \text{ , } (x_0 - y_0) = -d_0 \text{ , y } z_0 = (B - d_0)$$

En el caso de que $x_0 < y_0$ podemos, por tanto, tomar $z_0 = (x_0 - y_0) \bmod B$, donde z_0 toma valores

del conjunto completo de residuos $\mathrm{mod}\, B$ formado por los residuos menores no negativos: el conjunto $\{0,1,2,...,B-1\}$ (ver [Rose93]).

Y vista la expresión (6), para el caso en que $x_0 > y_0$ también podemos tomar como válida la expresión

$$z_0 = (x_0 - y_0) \bmod B \tag{7}$$

y debemos tener en cuenta que, en el caso de que $(x_0 - y_0) < 0$, necesitamos "tomar prestada" una unidad al dígito de peso correspondiente a B^1. Llamamos a esta *deuda* el acarreo de la resta, o el **"debe" de la resta** (o *borrow*).

Para cualquier dígito posterior habrá que tener en cuenta el "debe" del dígito previo (b_{j-1}). De nuevo tenemos que siempre se cumple que

$$\left|(x_j - y_j - b_{j-1})\right| \le B \tag{8}$$

En el caso de que $(x_j - y_j - b_{j-1}) \ge 0$, tenemos que $z_j = (x_j - y_j - b_{j-1})$. Si, en cambio,

$$(x_j - y_j - b_{j-1}) = -d_j < 0,$$

deberemos tomar prestada otra unidad al dígito superior:

$$B^{j+1} + (x_j - y_j - b_{j-1}) \cdot B^j = (B - d_j) \cdot B^j = z_j \bmod B \cdot B^j .$$

donde $(B - d_j) = z_j$

Podemos concluir que, para la resta, el número de dígitos será como máximo el número de dígitos del minuendo; y el valor de cada dígito es siempre

$$z_j = (x_j - y_j - b_{j-1}) \bmod B,$$

trabajando en nuestro conjunto de residuos que antes hemos definido. El valor del "debe" a descontar del siguiente dígito será 1 ó 0 dependiendo del signo que adopte la diferencia $(x_j - y_j - b_{j-1})$.

4.2. MODELOS DE NÚMERO PARA UNA CODIFICACIÓN EN UN ORDENADOR.

En un ordenador, los números vienen codificados en base binaria ($B = 2$). Sus dígitos pueden ser, por tanto, el 0 y el 1. El límite del tamaño de los enteros que puede manejar el ordenador viene determinado por el número n de bits que utiliza el ordenador para almacenar un número. Ese parámetro se conoce como **longitud de palabra** (n).

El límite superior del número que se puede representar se llama **tamaño de palabra** (w). Se cumple que $w = 2^n - 1$. Además, con una palabra de longitud n podemos codificar todos los

valores comprendidos entre 0 y el límite superior marcado por el tamaño de la palabra w. Podríamos decir que las variables creadas y definidas como de un tipo de dato de 32 bits operan siempre en una aritmética modular de módulo 2^{32}. Una vez se ha codificado el valor 4.294.967.295, si incrementamos en uno esa cantidad volvemos al valor cero.

Para trabajar con enteros más grandes que $w = 2^n - 1$, podemos tomar cada elemento de n bits como un dígito en un sistema de numeración en base w. Así podemos definir los **enteros largos o de precisión múltiple** como un vector de elementos que codifican enteros de 32 bits, sin signo. En C, por ejemplo, un vector de elementos de tipo **`unsigned long int`**.

Tendremos un entero largo cada vez que tengamos un entero a mayor o igual que w ($a \geq w$): entero para el que será necesario dedicar más de una palabra para codificarlo en una cifra en el ordenador. Cada entero largo quedará codificado por una cifra como la presentada en la fórmula (3) donde la base $B = w + 1 = 2^n$. Por ejemplo, en C, con un vector de k elementos **`unsigned long int`** podríamos codificar $w^k = 2^{k \cdot n}$ valores numéricos: desde el 0 hasta el valor $2^{k \cdot n} - 1$. Ese número vendría codificado con k dígitos, donde cada dígito puede tomar valores entre 0 y w.

4.2.1. Algunos modelos de entero largo ya definidos.

```
typedef unsigned long int mpi_limb_t;
struct gcry_mpi
{
   int alloced;     /* array size (# of allocated limbs) */
   int nlimbs;      /* number of valid limbs */
   int nbits;       /* the real number of valid bits (info only) */
   int sign;        /* indicates a negative number */
   unsigned flags; /* bit 0: array must be allocated in secure
                      memory space */
                   /* bit 1: the mpi is encrypted */
                   /* bit 2: the limb is a pointer to some
                      m_alloced data */
   mpi_limb_t *d;  /* array with the limbs */
};

typedef struct gcry_mpi *MPI;
```

Cuadro 1: Modelo de entero de precisión múltiple empleado en la implementación del programa PGP de Philip ZIMMERMAN.

Se pueden encontrar muy diferentes definiciones del tipo de dato entero largo. De entre todas ellas hemos seleccionado dos que nos parecen especialmente interesantes por diferentes motivos:

1. Sus autores gozan de reconocido prestigio en el mundo de la criptografía y se han lanzado a

la tarea de implementar herramientas para el empleo de enteros de estas características aquí descritas.

2. Estas implementaciones han sido ampliamente probadas y son reconocidas como válidas en la comunidad científica. No recogemos aquí, pero se puede rastrear en muchos artículos de investigación, las referencias a herramientas que emplean uno de esos modelos.

El primer modelo que mostramos es el definido e implementado por el autor de PGP (Pretty Good Privacy): el conocido Philip ZIMMERMAN [PGP98]. Esta aplicación tiene una definición muy desarrollada del concepto informático de entero de precisión múltiple: presenta el número como un puntero a una estructura que contiene varios campos: aparte del puntero que contendrá el array del entero largo, trae otros como el tamaño del array, los elementos válidos, los bits válidos que posee el entero largo, el signo, y una variable de estado. Es una estructura amplia, con abundante información. La recogemos en el Cuadro 1. No es nuestro objetivo ahora hacer una crítica a las diferentes implementaciones que hemos encontrado. Tampoco queremos hacerla a la implementación de Philip ZIMMERMAN. Pero es claro que este modelo es complejo y a nuestro parecer se alejaba, al menos inicialmente, del que nosotros necesitábamos para nuestros trabajos. La implementación que buscamos queríamos que fuese más simple.

```
typedef long * _ntl_verylong;
```

Cuadro 2: Modelo de entero de precisión múltiple empleado en la implementación de programa FreeLIP, de Arjen LENSTRA

El segundo modelo es mucho más sencillo. Lo hemos encontrado implementado de una u otra forma en muchas herramientas. Es el modelo que presuponen como válido los autores de [Ries87] y de [Knut81]. En resumidas cuentas, asume que un entero largo no es más que un vector de elementos de 32 bits, aunque no necesariamente todos ellos se empleen para codificar cifras, ni necesariamente se trabaje en base 2^{32}. Para presentarlo mostramos en el Cuadro 2 la implementación realizada por Arjen LENSTRA en la librería FreeLIP (Large Integer Package), que desarrolló para acometer la tarea de romper el RSA–129, y que actualmente está mantenida por Leyland. El entero largo o de precisión múltiple queda definido simplemente como un puntero a **`long`**. Si x es una variable de este tipo (puntero a **`long`**), tenemos:

⇒ Si x es puntero nulo, entonces el número codificado es el cero.

⇒ Si x no es nulo, entonces codifica un entero de signo igual al signo de `x[0]` y de tantos dígitos como indique el valor absoluto de `x[0]`. Si `n` es el valor absoluto de `x[0]` y `s` el signo de

`x[0]`, entonces el número codificado es:

`s * (x[1] + x[2]* BASE + ... + x[n] * BASE ^(n – 1))`

Donde `BASE` no necesariamente es 2^{32}; es más, su valor en este caso es 2^{30}, es decir, 1.073.741.824.

⇒ El cero puede codificarse de dos maneras: como ya ha quedado dicho, mediante el puntero nulo, o dando `x[0] = +1` y `x[1] = 0`.

Para reservar memoria y liberarla utiliza las funciones `malloc()` y `free()`.

Una implementación similar es la que presenta Henri COHEN en su herramienta PARI, aunque en ese caso define una multitud de posibles variables y tipos, tanto enteros, como reales, complejos, polinomios, vectores, etc. La forma que emplea para los enteros queda recogida en el Cuadro 3 y que resulta idénticamente la misma que le definida por LENSTRA.

```
typedef long * GEN;
```

Cuadro 3: Modelo de entero de precisión múltiple empleado en la implementación de programa PARI, de Henri COHEN

Otro ejemplo similar es el empleado en la implementación de los números que realiza RSA*Euro* Technical Reference [Barr96]: representa a sus números enteros como vectores de longitud arbitraria de elementos `NN_DIGIT`, donde el tipo de dato `NN_DIGIT` es el de los enteros sin signo de 32 bits (**`unsigned long int`**).

Estas implementaciones son efectivamente muy sencillas. Como ya hemos señalado, son también las más recomendadas en la bibliografía consultada. En un solo vector se dispone de la información sobre el signo del número, el tamaño y sus dígitos. Es una forma de implementar que nos parece muy adecuada para nuestros propósitos de crear una librería para factorizar enteros productos de dos primos largos. De todas formas nuestro modelo difiere en distintos detalles al modelo de LENSTRA o COHEN, y queda presentado en el siguiente epígrafe.

4.2.2. Presentación de nuestro modelo de entero largo.

Entendemos por **tipo de dato** la explicitación de un conjunto de valores, que llamaremos **dominio**, sobre el que se han definido unas determinadas operaciones. En esta sección del segundo epígrafe mostraremos el diseño que adopta nuestro entero largo implementado en C. En el

siguiente epígrafe describiremos cada una de las funciones que hemos definido e implementado para que nuestro tipo de dato tenga sus operadores propios, necesarios para su uso. El dominio del nuevo tipo de dato será el de los valores enteros de longitud tan grande como se quiera, sin más limitaciones previas que las impuestas por la capacidad de memoria del computador donde se implemente este tipo de dato.

El desarrollo de las operaciones necesarias en el proceso de factorización, que fuesen válidas para todo tipo de tamaño de entero largo, y la lucha constante por reducir los tiempos, nos ha conducido a una sucesiva redefinición en el concepto de entero largo. No se trataba únicamente de mantener una cadena de elementos de tipo entero; había otra información que resultaba habitualmente de interés y que necesitábamos almacenar actualizada, como la posición relativa del bit o del dígito más significativo dentro de cada cifra. Esa información resultaría muy útil para acelerar muchas de las funciones aritméticas básicas; y, desde luego, sería de inmediato uso en las operaciones relacionales. También permitiría verificar de un modo sencillo y rápido si un entero grande adquiría el valor uno (muy útil por ejemplo en test de primalidad) o el valor cero: bastaría verificar que el número de bits significativos era 1 ó 0, respectivamente.

```
typedef struct
{
   unsigned long int D;
   unsigned long int T;
   unsigned long int B;
   unsigned long int*N;
}NUMERO;
```

Cuadro 4: Nuestro modelo de entero largo.

El modelo por el que finalmente hemos optado para definir el entero largo, más allá de un sencillo puntero donde queden direccionados un número determinado de bytes, es el recogido en el Cuadro 4, donde en el puntero `N` se recoge la dirección de la cadena de **`unsigned long int`** que servirá para los distintos dígitos del número; en `D` se recoge la dimensión del número (dimensión del vector), es decir, la cantidad de elementos **`unsigned long int`** que se han asociado al número mediante la función `malloc()`; en `T` se recoge cuántos de esos `D` elementos han sido realmente empleados para almacenar dígitos del entero largo que estamos codificando; en `B` queda señalado lo mismo que en `T`, pero considerado el número expresado en binario: recoge el número de bits del entero codificado.

4.2.2.1. Comentarios comparativos entre los distintos modelos presentados.

Esencialmente el formato es el mismo que los presentados en los Cuadros 2 y 3. Señalamos cinco

diferencias entre los modelos de LENSTRA o COHEN y el nuestro:

1. Nuestro modelo carece de signo. Desde luego, quien desease que esa limitación no existiera bastaría con que añadiera un nuevo campo a la estructura definida o que, al igual que hace LENSTRA, definiese el valor del campo `T` de tipo **`signed long int`**, y recogiera en el signo de ese parámetro el signo del entero largo codificado.

   La realidad es que aunque al principio definimos la estructura con este elemento del signo, finalmente lo eliminamos, pues salvo en la implementación de una de las funciones, ese parámetro no nos ha sido necesario. E incluso es prescindible en general para cualquier implementación, puesto que trabajamos en el ámbito de la aritmética modular, donde el conjunto de elementos es finito y podemos tomar como representantes de la clase de equivalencia que se crea con la relación de congruencia todos los valores comprendidos entre el cero y el valor del módulo menos 1. Todos entero negativo en aritmética modular puede expresarse con su representante positivo en su conjunto completo de residuos.

2. En nuestro modelo no sólo se dispone del número de elementos del vector que están de hecho ocupados en la codificación del entero. También recogemos el número de bits empleados, lo que ofrece una ayuda importante para la definición del operador relacional que indica cuál es el mayor de dos enteros, y simplifica mucho el reconocimiento del valor cero (caso en que el campo `B` vale `0`) y del valor uno (caso en que el campo `B` vale `1`).

3. Una variación sustancial entre nuestro modelo y los presentados en el epígrafe anterior es que el vector que recoge los dígitos del entero grande es de tipo **`unsigned long`**, y no **`signed long`**, como puede comprobarse, en los Cuadros 2 y 3, que resultan ser en los modelos de COHEN y de LENSTRA. Como se puede apreciar en el Cuadro 1, el modelo de Philip ZIMMERMAN sí utiliza un vector de enteros de 32 bits sin signo para el vector que almacena los dígitos del entero largo.

4. La base en que trabaja nuestro modelo es 2^{32}. Eso no es posible cuando el tipo de dato que sustenta el vector de dígitos es un tipo de dato con signo. Ya hemos señalado, por ejemplo, que el modelo de LENSTRA trabaja en base 2^{30}.

5. En los modelos de COHEN y LENSTRA, el número de elementos reservados para el vector que recoge los dígitos del entero largo es un valor independiente del entero concreto para el que se hace la reserva de memoria; o al menos hay que decir que la dimensión de ese vector no está asociada al propio entero que se codifica. Nosotros hemos querido que cada entero tuviera determinado el valor máximo que podía codificar: cada vez que una función necesita un entero largo, esa función determina cuántos dígitos (en base 2^{32}) necesitará nuestra variable para codificar el valor del entero.

La relación de diferencias que acabamos de presentar no pretende ser una relación de mejoras. No es nuestro objetivo ahora demostrar que nuestra implementación es mejor o peor que otras.

Hemos querido justificar las razones de nuestra elección. A nosotros nos ha resultado más sencillo trabajar con estas modificaciones.

Tenemos también que presentar una lista de diferencias entre nuestro modelo y el realizado por Philip ZIMMERMAN y recogido en el Cuadro 1. La verdad es que, después de señalar las diferencias con los modelos de COHEN y LENSTRA, cabría pensar que nuestro modelo más se parece al del autor de PGP. Nosotros pensamos que las diferencias con este modelo son aún mayores, aunque hay dos semejanzas importantes:

1. El modelo de ZIMMERMAN también recoge el tamaño máximo de elementos para cada vector de dígitos (campo `alloced` en el modelo de ZIMMERMAN; campo `D` en nuestro modelo).
2. El modelo de ZIMMERMAN también recoge el número de dígitos empleados en la codificación del entero, tanto en base 2^{32} (campo `nlimbs` en el modelo de ZIMMERMAN; campo `T` en nuestro modelo) como en base 2 (campo `nbits` en el modelo de ZIMMERMAN; campo `B` en nuestro modelo).

También el modelo del autor del PGP tiene diferencias con respecto al nuestro:

1. El modelo de ZIMMERMAN también recoge el signo del entero, cosa que, como ya hemos dejado dicho, no se hace en nuestro modelo.
2. El modelo de ZIMMERMAN tiene un campo que recoge el estado del entero codificado.
3. El tipo de dato está finalmente creado como de tipo puntero.

Una vez hemos mostrado diferentes modelos de codificación de enteros y hemos relacionado sus semejanzas y sus diferencias en relación con nuestro modelo, queremos hacer una aclaración al proceso real que ha seguido nuestro trabajo. A pesar de lo que pueda parecer, por el modo en que hemos presentado nuestro modelo y las listas de diferencias con los otros modelos, la realidad es que el primer paso en la implementación de nuestra librería no fue buscar otras ya implementadas y tomar una definición de entero largo como punto de arranque: definición sobre la que posteriormente cabría hacer modificaciones. Lo que hicimos fue crear de forma intuitiva una primera definición de entero largo y, sobre ella, realizar el diseño e implementación de todas y cada una de las funciones que íbamos necesitando o que creíamos necesarias para una completa definición de todos los operadores de un tipo de dato entero.

Nuestro primer modelo de entero fue muy parecido al de los Cuadros 2 y 3. Era de la forma

```
typedef unsigned long int * NUMERO;
```

y por tanto se diferenciaba del modelo de COHEN y de LENSTRA, ya desde su inicio, en que los dígitos eran enteros sin signo, que la base de codificación era 2^{32} y que no reservábamos ningún elemento del vector que asociásemos a las variables tipo `NUMERO` para el signo o para el tamaño

del entero codificado.

A medida que avanzamos en la implementación de las distintas funciones fueron llegando las modificaciones sobre nuestro primer modelo. Al terminar todas las implementaciones de las funciones que presentaremos en este Capítulo nuestro modelo había ya quedado en la forma señalada en el Cuadro 4, salvo una diferencia: manteníamos un campo para el signo; campo del que, como ya hemos dicho, pudimos luego prescindir al comprobar su escaso o nulo uso que necesitábamos hacer de él.

El siguiente paso fue ya el de la búsqueda de otras implementaciones. Búsqueda y selección de aquellas que nos parecieran más válidas, según los criterios que ya antes hemos recogido. Se puede pensar que realizamos nuestra tarea en el orden inverso al que hubiera sido lógico en un trabajo como el nuestro. Quizá sea cierto. Pero gracias a este modo de proceder, pudimos comprobar que el proceso de implementación del entero nos había conducido a modelos muy semejantes a los que luego hemos encontrado ya hechos. Quizá se puedan implementar modelos de formato esencialmente diverso a los presentados en estas líneas. Pero quizá podamos pensar que la misma forma de ser de los enteros largos y con la actual arquitectura de los ordenadores de que disponemos, los modelos que se pueden definir apuntan a un formato como el de los presentados aquí.

Cabría pensar que el trabajo de implementar toda la librería es una tarea ya realizada por otras personas de reconocida capacidad. Pero como decíamos al principio de este epígrafe, un tipo de dato no es solo un dominio de valores, o un modelo de representación o codificación de esos valores; un tipo de dato también presupone un conjunto amplio de operadores que hay que definir. Y si queríamos realizar una implementación optimizada a la luz de la interacción del software compilado con el ordenador, era necesario conocer a la perfección cada una de las funciones. No sabemos cuál es, de entre todas las recogidas, la mejor implementación de entero largo. Pero sólo la nuestra ha podido ser sometida a la tarea de optimización. No nos parecía viable pretender modificar, al detalle al que hemos podido llegar en nuestras funciones, las funciones implementadas por otros programadores.

Más adelante, en el Capítulo 7, podremos comprobar cómo las funciones básicas que definen los operadores de este modelo, intervienen en el proceso de factorización una cantidad enorme de veces; y podremos comprobar cómo cualquier mejora en la implementación de cada una de esas funciones, supone una mejora el rendimiento en la ejecución y, por tanto, en la eficiencia de nuestro modelo. Y a la vista de los resultados recogidos en el Capítulo 7 podemos afirmar que nuestro proceso de optimización ofrece buenos resultados.

4.2.2.2. Otras consideraciones sobre nuestro modelo de entero largo.

Hay que determinar el orden de los dígitos (los diferentes elementos **`unsigned long int`**) en la memoria. Hemos tenido en cuenta para esta decisión que los ordenadores PC (que usan

microprocesadores Intel y AMD) son ***little–endian***: es decir, que una variable que use más de un byte para su codificación almacena el byte de menor peso en la dirección más baja de memoria y el byte de mayor peso en la dirección más alta.

En el Cuadro 5 recogemos el orden de los bits en un entero largo de 32 bits (**`unsigned long int`**) almacenado en memoria mediante el formato *little–endian*. Si nuestro número requiere más de un elemento de 4 bytes, la posición lógica a colocar los siguientes bits más allá del bit 31 es a partir del bit 32, que corresponde al bit 0 de un segundo elemento **`unsigned long int`**.

Hemos optado por este formato *little–endian* porque es el que utilizan los ordenadores sobre los que hemos realizado toda nuestra programación. Por tanto tenemos que el valor del campo `T` en nuestra estructura `NUMERO` indica la cantidad de elementos **`unsigned long int`** empleados para la codificación, pero también señala la posición del elemento más significativo. Y `B` señala la posición del bit más significativo. Como ya hemos dicho, este sistema de codificación deja muy accesible algunas operaciones: será mayor aquel número cuyo valor del campo `B` sea mayor. Un número será igual a cero si el campo `B`, o el campo `T`, es igual a cero. Un número es igual a uno si el campo `B` es igual a 1.

Evidentemente nada obliga a utilizar este orden y ambas posibilidades (*little–endian* vs *big–endian*) son igualmente válidas. Además, transformar un número de un formato a otro es tarea inmediata. Una vez tomada la opción, evidentemente, todas las operaciones se realizarán teniéndola en cuenta. La implementación de la librería FreeLIP también ha tomado el formato *little–endian*.

```
bits: [  7  6  5  4  3  2  1  0 ]  Byte 0  < Pointer
bits: [ 15 14 13 12 11 10  9  8 ]  Byte 1
bits: [ 23 22 21 20 19 18 17 16 ]  Byte 2
bits: [ 31 30 29 28 27 26 25 24 ]  Byte 3
```

Cuadro 5: Formato *little–endian*. Orden en que se almacenan los dígitos binarios. El puntero señala al byte 0, que es el que recoge los 8 bits de menor peso. A medida que nos alejamos de la posición señalada por el puntero aumenta el peso del dígito. Tal y como estamos habituados en nuestro sistema de numeración, los ceros que se añaden a la "izquierda" de una cifra no alteran su valor.

4.3. DESCRIPCIÓN DE NUESTROS OPERADORES EN NUESTRO MODELO DE NÚMERO.

Una vez hemos definido el modelo de entero largo y su esquema de codificación y, por tanto también, definido el dominio de valores que puede codificar nuestro tipo de dato, queda ahora mostrar los operadores. Los hemos clasificado en dos grandes grupos:

1. **Operadores auxiliares**: Así hemos llamado a los operadores de asignación de valores a los distintos campos de la estructura `NUMERO`: la asignación dinámica de memoria; el operador asignación necesario para inicializar a cero la variable y copiar el valor de una variable en la posición de otra variable; los operadores relacionales, que determinan cuál de entre dos valores es mayor, y si dos valores son iguales o no; los operadores a nivel de bit de cálculo del tamaño del número en bits, de desplazamiento a izquierda y a derecha, y de intercambio de valores entre dos variables.
2. **Operadores aritméticos**. En este grupo traemos definidas las operaciones clásicas aritméticas entre enteros: suma, resta, producto, cociente y módulo.

Todas estas operaciones resultan sencillas de implementar. Pero no todas las implementaciones válidas son igualmente rápidas. Cualquiera de ellas requerirá de un tiempo de ejecución mínimo, de escasos microsegundos; pero todas ellas se ejecutan una cantidad enorme de veces y, por tanto, que su implementación sea óptima es, en este caso, de gran importancia. Para ganar en velocidad a la hora de realizar un proceso más complejo, es necesario mejorar la velocidad de estas funciones más sencillas.

4.3.1. Operadores auxiliares.

Llamamos operadores auxiliares a los siguientes: reserva de espacio de memoria para el vector de dígitos de la variable, operador asignación, operadores relacionales y operadores a nivel de bit (desplazamientos a izquierda y a derecha e intercambio de valores entre dos variables de tipo `NUMERO`). Todos ellos forman el conjunto más elemental de operadores para el manejo de nuestro tipo de dato y constituyen la estructura básica necesaria para el desarrollo de los demás operadores y de diferentes funciones criptográficas.

Los operadores aritméticos, como la suma, resta, producto, cociente y módulo entre variables de tipo entero largo o entre una variable de este tipo y un entero simple; o los operadores matemáticos de exponenciación y potencia, cálculo de la raíz cuadrada y cálculo de logaritmo; o las funciones implementadas para la generación de números primos; ... requieren de estos operadores básicos que aquí presentamos.

En todo momento entendemos por tipo de dato `UINT4` el entero de 4 bytes sin signo (**`typedef unsigned long int`** `UINT4`).

4.3.1.1. Reserva de espacio en memoria.

Definimos la función

```
void CrearNumero(NUMERO*);
```

que recibe como parámetro una variable tipo `NUMERO` y le asigna al campo `N` un vector de tantos elementos `UINT4` como indique el campo `D`. Para esta asignación dinámica de memoria empleamos la función `malloc()`, definida en la biblioteca estándar de C `stdlib.h`. Hemos definido la función de tal manera que al ser invocada debe estar ya actualizado el valor del parámetro `D`. Antes de invocar a la función se debe indicar el valor de ese campo. Eso permite crear las variables de tipo `NUMERO` con un tamaño ajustado a los dígitos en base 2^{32} que se van a necesitar realmente. En todo momento, a lo largo de nuestra implementación, hemos procurado que cada variable de tipo `NUMERO` empleada tuviera el valor del campo `D` mínimo posible; es decir, hemos procurado que los tamaños de la memoria a recorrer fuesen mínimos. Así creados, los enteros largos gozan de gran versatilidad, y son capaces de tomar cualquier dimensión y variar así el dominio de valores que puede tomar la variable, acotándolo siempre al máximo valor necesario. Y así, además, todas las operaciones reducen su tiempo de ejecución gracias a que la dimensión del vector numérico (campo `N`) a recorrer reduce su tamaño.

La función deja la variable tipo `NUMERO` con todos los elementos `UINT4` a cero y los valores de los campos `T` y `B` a cero.

Todas las demás funciones que reciban como parámetro una variable de tipo `NUMERO` deben recibir esa variable con la memoria (vector `N`) ya reservada. Esta función es de obligado uso para cualquier variable tipo `NUMERO` que deseemos utilizar. Cuando una variable tipo `NUMERO` deja de ser usada en una función, es conveniente liberar la memoria del puntero `N` mediante la función `free()` de la biblioteca `stdlib.h`.

Una observación sobre el diseño de esta función. Hemos definido la estructura `NUMERO` de tal manera que la dimensión del vector de dígitos, recogido en el puntero `N` sea el valor del parámetro `D`. Podríamos haber permitido redimensionar el tamaño de una variable `NUMERO`, cambiando el valor del parámetro `D` e invocando a la función `realloc()`. Implementar una función así es, desde luego, una tarea sencilla; no lo hemos hecho porque no ha surgido la necesidad en todo el proceso de nuestro trabajo.

Al trabajar con variables de tipo `NUMERO`, se debe vigilar que no se pretenda acceder a más dígitos de los reservados en la memoria dinámica: si así ocurriese se produciría un error de violación de segmento, que o abortaría la ejecución del programa, o podría tener consecuencias impredecibles en los valores y resultados que se obtuviesen. Si se implementase esta función, deberíamos mantener la vigilancia sobre la posibilidad de que una variable tipo `NUMERO` quisiera alcanzar a más dígitos de los que es capaz de almacenar; y la función de realojo sería invocada en esas circunstancias.

4.3.1.2. Asignación: Inicializar a cero una variable NUMERO.

Hemos definido la función

```
void PonerACero(NUMERO*);
```

que deja a cero todos los elementos UINT4 direccionados por el puntero N y los valores de los campos T y B.

4.3.1.3. Asignación: Copia del valor de una variable en otra, ambas de tipo NUMERO.

Para esta operación hemos definido la función

```
void CopiarNumero(NUMERO*,NUMERO*);
```

que utiliza la función memcpy()definida en la biblioteca string.h.

Evidentemente, no se puede realizar esta operación cuando el valor del campo D de la variable "destino" (el segundo parámetro de la función) de la asignación sea menor que el valor del campo T de la variable original (el primer parámetro de la función). La misma función CopiarNumero() hace la verificación, y muestra un mensaje de error e interrumpe la ejecución del programa en caso de que alguna función haya pretendido realizar esta operación en esas malas condiciones y que supondría una violación de segmento.

En la implementación de esta y otras funciones, hemos dejado condicionadas muchas de estas verificaciones, utilizando directivas de preprocesador. Hemos utilizado la palabra DEBUG para condicionar todas estas verificaciones. Cuando se implementa una nueva aplicación, es conveniente introducir al principio del código la directiva #define DEBUG, que se puede quitar posteriormente cuando la aplicación ha sido suficientemente probada y se ha verificado que en todos los casos se invoca a la función correctamente y no se incurre en ningún error que provoque violación de segmento.

4.3.1.4. Operador para calcular tamaños (campos T y B) de una variable tipo NUMERO.

El valor de los parámetros T y B en la estructura NUMERO deben estar siempre actualizados. Todos los operadores ganarán en velocidad si logramos que así sea. La función que hemos definido tiene el siguiente prototipo:

```
void longitud(NUMERO*);
```

Aunque el proceso de esta función exige cierto tiempo en su ejecución, su trabajo compensa notablemente en la evaluación general de rendimiento de todas las demás funciones, que ganan facilidad de implementación y velocidad de ejecución gracias a la constante disponibilidad de estos dos valores siempre actualizados. Como veremos en el desarrollo del Capítulo que trata de la optimización de código, esta función ha quedado finalmente bastante reducida en el número de operaciones que debe realizar: de nuestra primera versión a la mejorada se ha reducido en 14 veces el

número de instrucciones a ejecutar; también ha quedado reducido a casi la mitad el número de veces que se llama a la función `longitud()` en el proceso de factorización de un entero largo mediante la técnica de CFRAC en nuestra implementación.

Podemos decir de esta función que su peso en el coste total de tiempo de ejecución de nuestra aplicación de factorización es muy reducido y se puede considerar despreciable, y el valor de la información que aporta facilita en muchas funciones una implementación más ágil y sencilla que si no dispusiéramos de los valores, siempre actualizados, de los parámetros `T` y `B`.

4.3.1.5. Operadores relacionales.

Una tarea fundamental es la definición de los operadores relacionales: determinar si el valor codificado en una variable tipo `NUMERO` es mayor, igual o menor que el valor codificado por otra variable tipo `NUMERO`.

Hemos definido la función

```
short orden(NUMERO*,NUMERO*);
```

que devuelve el valor `+1` si el valor numérico del primer parámetro es mayor que el segundo; devuelve `-1` si ocurre al revés; devuelve `0` si ambas variables tienen el mismo valor.

Un dato que ofrece una información significativa sobre el peso que una función de estas características puede tener en el proceso de factorización, es el número de llamadas que ésta recibe. En el Capítulo sobre la optimización de código veremos que en nuestra primera versión del programa de factorización, para obtener los dos primos que factorizan un entero de 100 bits, esta función alcanzaba más de 40 millones de llamadas. Al final del proceso de optimización, y después de un extenso proceso que en el Capítulo 7 se muestra, se ha logrado reducir a poco más de 700.000 las llamadas en el mismo proceso de factorización del mismo número. Aunque es evidente que la reducción es de altísimo porcentaje, también es cierto que, en términos absolutos, esta función ha seguido recibiendo una gran cantidad de llamadas.

4.3.1.6. Operadores a nivel de bit: Intercambio de valores entre dos variables tipo `NUMERO`.

Este intercambio de valores lo realizamos mediante el operador *or exclusivo*, repetido tres veces. El procedimiento es bien conocido; para intercambiar los valores de las variables enteras `a` y `b`, basta que ejecutemos en el orden aquí recogido estas tres instrucciones:

```
a ^= b; b ^= a; a ^= b;
```

Para esta operación ha quedado definida la función.

```
void inv_v(NUMERO*,NUMERO*);
```

Una vez más, y de la misma forma que ya quedó explicado con la función `CopiarNumero()`, el código de la función verifica que el valor del parámetro `D` de la primera variable tipo `NUMERO` no sea menor que el parámetro `T` de la segunda variable tipo `NUMERO`, y viceversa. En caso de que algunas de esas

condiciones se cumpliese, la función `inv_v()` imprime un mensaje de error y aborta la ejecución del programa. De lo contrario, si se continuase con el proceso,, se incurriría en un error de violación de segmento.

Una vez depurada la aplicación y comprobado que en todas las llamadas a esta función no se incurre en ese error, puede desactivarse el proceso de comprobación anulando la macro `DEBUG`, como ya ha quedado antes explicado.

4.3.1.7. Operadores a nivel de bit: Desplazamiento a izquierda.

Esta operación resulta muy necesaria para muchas de las operaciones aritméticas, tanto para doblar cantidades como para el proceso de multiplicación y cociente.

El prototipo de la función definida para esta operación es

```
void DESPL_izda(NUMERO*,short);
```

donde el parámetro tipo `short` recoge el número de bits que debe desplazarse hacia la izquierda la variable tipo `NUMERO`.

En nuestra primera versión de la aplicación para factorizar enteros mediante el algoritmo CFRAC esta función era llamada más de 10 millones de veces (para un compuesto de dos primos de 50 bits cada uno). Después de las optimizaciones, el número de llamadas se ha reducido en un factor mayor de 11, y recibe poco más de un millón de llamadas e invierte poco más de 100 millones de ciclos de reloj de un total de 2.400 millones que se invierte en todo el proceso de factorización. Y es que, como venimos diciendo, estas funciones que hemos llamado auxiliares, son básicas en toda la implementación posterior de las demás funciones, y el número de intervenciones es enorme en el total del proceso. Por eso, y una vez más, constatamos la importancia que tiene que el diseño de cada una de estas funciones se haga de forma aquilatada, sopesando cada modo de afrontar un cálculo. No es despreciable el hecho de que esta función consuma algo más del 4% del total de los ciclos de reloj del proceso de factorización en un entero de 100 bits.

Una vez más hacemos la observación sobre la necesidad de vigilar que no se intente, en una operación de desplazamiento a izquierda, trabajar con más dígitos que los reservados en memoria dinámica cuya cantidad viene recogida en el parámetro `D`.

4.3.1.8. Operadores a nivel de bit: Desplazamiento a derecha.

Semejante a la función anterior. Su prototipo es:

```
void DESPL_dcha(NUMERO*,short);
```

En el caso de esta función, como lo que ocurre en su ejecución es que se reduce el valor del entero codificado, no es necesario vigilar para que no se incurra en una violación de segmento. En el caso de que se pretenda una desplazamiento de un número de bits mayor que el valor del parámetro `B` el entero queda idénticamente igual a cero.

4.3.2. Operadores aritméticos.

Ya hemos presentado, en un epígrafe previo, algunos fundamentos matemáticos sencillos sobre el modo en que se realizan algunas de las operaciones aritméticas básicas. Vamos ahora a mostrar las funciones aritméticas implementadas.

4.3.2.1. Implementación de la operación para la suma.

Como hemos explicado antes, en nuestra implementación hemos trabajado con una codificación de enteros que toma la base $B = 2^{32}$: cada dígito viene codificado en un entero sin signo de 32 bits. Cada dígito es un elemento de memoria de cuatro bytes que codifica valores enteros comprendidos entre 0 y $2^{32} - 1$.

Para determinar si en cada suma de dígitos del mismo peso hemos tenido acarreo existen muchos procedimientos. El valor de cada dígito de la suma, obtenido a partir de los dígitos de cada sumando más el acarreo de la suma de los dígitos previos ($z_j = (x_j + y_j + C_{j-1}) (\mathrm{mod}\, B)$), se obtiene de forma inmediata sin necesidad de realizar la operación módulo $\mathrm{mod}\, B$: por la misma definición de dígito dentro de la memoria del ordenador (elemento de formato entero sin signo de 32 bits: dominio de valores entre 0 y $B = 2^{32} - 1$) se cumplirá que si el valor suma de los dígitos del mismo peso de cada uno de los sumandos es mayor que B (si $(x_j + y_j + C_{j-1}) \geq B$), tendremos una situación de overflow en la variable `UINT4` z_i: es decir, llegado al valor máximo $B = 2^{32} - 1$, el siguiente entero después del máximo es el cero; por lo tanto, la suma de dígitos se realiza automáticamente en su forma modular $\mathrm{mod}\, B$.

Para la operación de la suma hemos definido dos funciones:

```
void SUMA(NUMERO*,NUMERO*,NUMERO*);
```

```
void Suma(NUMERO*,UINT4,NUMERO*);
```

La primera de las dos funciones realiza la suma de los dos primeros parámetros, direcciones de variables de tipo `NUMERO`, y deja el resultado en la dirección de la variable que viene como tercer parámetro. La segunda función está definida para realizar la suma de una variable tipo `NUMERO` con un entero de 32 bits (tipo `UINT4`); el tercer parámetro de la función es la variable donde quedará almacenado el valor del resultado de la operación.

La primera función viene definida de manera que los tres parámetros pueden direccionar a una misma variable; o dos parámetros a una misma variable y el tercero a otra diferente; o los tres parámetros direccionar a variables distintas.

4.3.2.2. Implementación de la operación para la resta.

Para la implementación de la resta tenemos en cuenta cuanto hemos dicho antes, cuando hemos recogido algunos fundamentos matemáticos. El comportamiento de las variables en el

lenguaje de programación escogido facilita, como veremos ahora, la implementación de esta función.

Hemos definido la estructura `NUMERO`, encargada de codificar los enteros largos, de tal manera que cualquier entero codificado sea positivo. Y además cada dígito es un elemento **`unsigned long int`**. Cada dígito de nuestra estructura tipo `NUMERO` puede codificar un valor entero comprendido entre 0 y $2^{32}-1=4.294.967.295$. Su dominio queda restringido a valores enteros positivos.

Supongamos que $(x_j - y_j - b_{j-1}) > 0$. En tal caso, como ya ha quedado dicho, el residuo $z_j = (x_j - y_j - b_{j-1}) \bmod B$ será el dígito que debemos almacenar en la cifra de la resta. Además, el cálculo $\bmod B$ no será necesario en ningún caso puesto que $(x_j - y_j - b_{j-1}) < B$ siempre.

Supongamos ahora que ese valor $(x_j - y_j - b_{j-1}) = -d$, donde $0 < d < B$.

¿Cuál debe ser el dígito a guardar? : como ya quedamos, será el residuo $z_j \equiv -d \bmod B$. Como sabemos, $z_j \equiv -d \bmod B \Leftrightarrow z_j + d = k \cdot B$. Tomamos $k = +1$ y llegamos a que el dígito será un valor z_j tal que

$$z_j = B - d \tag{9}$$

Ese es el residuo dentro del conjunto de residuos en el que trabajamos. Y este es el valor que se obtiene en la ALU cuando se opera con dos variables de codificación entera sin signo. Al resultar un valor fuera del dominio del tipo de dato, se produce un overflow (el valor que se obtiene al restar 1 al valor 0 es el número $B-1=2^{32}-1$) y el valor obtenido como resultado es precisamente $2^{32}-d$.

Hemos probado una forma de implementar la resta, aprovechando el hecho de que los enteros con signo codifican el entero en su complemento a la base cuando el entero es negativo y en su forma binaria cuando el entero es positivo; y el signo –como se sabe– viene recogido en el bit más significativo: un cero si el número es positivo; un uno si es negativo.

En el compilador `gcc` de Linux disponemos de un tipo de dato estándar de formato entero de 64 bits: **`long long int`**, que puede ser declarado **`signed`** o **`unsigned`**. Definimos el tipo de dato SINT8: **`typedef signed long long int`** SINT8;

Supongamos por ejemplo la siguiente situación:

```
UINT4 a = 0x00000001;
UINT4 b = 0x00000002;
SINT8 c = (SINT8)a – (SINT8)b;
printf(“\n%llX\n”,c);
```

que ofrece como salida en pantalla el valor `0xFFFFFFFFFFFFFFFF`, que como sabemos y esperábamos resulta ser el complemento a la base del entero 1 (valor absoluto del resultado de la resta realizada), en una numeración de 63 dígitos binarios (el bit más significativo se reserva para el

signo y queda a 1 porque el resultado es negativo).

Los 32 bits menos significativos de la diferencia (la variable c) nos ofrecen (como veremos más adelante) el valor correspondiente al dígito de la resta de enteros largos que deberemos guardar. Los 32 bits más significativos nos ofrecen la información sobre el acarreo de la resta. Siempre que se produzca acarreo tendremos estos 32 bits al valor 1; si no hay acarreo los 32 bits más significativos estarán a 0. El caso extremo del anterior es...

```
UINT4 a = 0x00000000;
UINT4 b = 0xFFFFFFFF;
SINT8 c = (SINT8)a – (SINT8)b;
printf("\n%llX\n",c);
```

que ofrece como salida en pantalla el valor `0xFFFFFFFF00000001`.

y que, tomando los 32 bits menos significativos, codifica el complemento a la base del valor `0XFFFFFFFF`, que es el valor absoluto del resultado de la resta.

Por tanto, para averiguar si tenemos acarreo en la resta bastaría comprobar si cualquiera de los 32 bits más significativos de la variable c de tipo `SINT8` está a uno.

Veamos desde un punto de vista más teórico las posibilidades que ofrece la codificación con el complemento a la base de los enteros negativos.

El concepto de **complemento a la base** depende de una premisa: del número de dígitos utilizados para codificar el número. Supongamos que trabajamos en una base B y que disponemos de k dígitos para codificar nuestro número N. Evidentemente, si N no requiere de todos los k dígitos para su codificación en la base B, bastará dejar a cero todos los dígitos a la izquierda de la codificación. Definimos el complemento a la base de N (y lo denotamos como $C_B(N)$) al valor resultante de restar $B^k - N$. Y definimos el complemento a la base menos uno de N (y lo denotamos como $C_{B-1}(N)$) al valor $C_B(N) - 1$.

Como se sabe, esta definición es muy útil cuando trabajamos en base $B = 2$, porque entonces tenemos que el valor del **complemento a la base menos uno** de cualquier N se halla de forma inmediata: allí donde N tiene un dígito 1, $C_1(N)$ tiene un dígito 0; y viceversa.

Los ordenadores suelen almacenar los enteros negativos en formato complemento a la base. El bit más significativo queda reservado para el signo (1 si N es negativo; 0 si N es positivo); y el resto de los bits codifican el entero en su formato binario su $N \geq 0$, o codifican $C_2(|N|)$ si $N < 0$.

Es decir, si queremos codificar $-N$ (donde $N > 0$), tenemos que si

$$N = \sum_{i=0}^{31} N_i \cdot 2^i ,$$

donde cada N_i valdrá ó 0 ó 1, entonces

$$C_2(-N) = C_1(-N) + 1 = \sum_{i=0}^{31} \overline{N_i} \cdot 2^i + 1$$

donde cada $\overline{N_i}$ valdrá 0 si su correspondiente N_i vale 1 y valdrá 1 si su correspondiente N_i vale 0. Por lo tanto es inmediato deducir que

$$C_2(-N) + N = \sum_{i=0}^{31} \overline{N_i} \cdot 2^i + \sum_{i=0}^{31} N_i \cdot 2^i + 1 = \sum_{i=0}^{31} 2^i + 1 = (2^{32} - 1) + 1 = 2^{32} = B \Rightarrow$$
$$C_2(-N) = B - N \qquad (10)$$

De todo esto es inmediato deducir que si tenemos en la operación resta, para el dígito j–ésimo, que $z_i = (x_j - y_j - b_{j-1}) < 0$ entonces, si trabajamos con la forma aquí presentada, el valor z_i que obtendremos será un entero de 64 bits, en el que los 32 más significativos serán todos igual a 1 (y por tanto deduciremos que el acarreo para la resta de los siguientes dígitos será 1); y los 32 bits menos significativos serán el complemento a la base de $|z_i|$. Y como hemos visto en la expresión (10), $C_2(-|z_i|) = B - |z_i|$: expresión de formato idéntico a la recogida en (10). Es decir, los 32 bits menos significativos recogen exactamente el valor que estamos buscando.

Hemos probado esta forma de implementación de la resta. Pero como veremos más adelante, en el Capítulo de optimizaciones, el uso de las variables de 64 bits, aunque resulta muy cómodo a la hora de implementar un algoritmo como el que ahora estamos presentando, grava el tiempo de computación y, especialmente, el número de microinstrucciones a realizar. Por eso finalmente hemos abandonado este diseño para la resta y hemos optado por otro más sencillo de presentar, que requiere un poco más de código y que es más rápido sencillamente porque evita las variables de 64 bits. Para saber si se tiene acarreo ó "debe" para la resta de los siguientes dígitos el método que hemos seguido es considerar que tendremos acarreo si el dígito del sustrayendo más el acarreo de la operación de los dígitos inmediatamente anteriores resulta mayor que el del minuendo.

Para la operación de la resta hemos definido dos funciones:

```
void RESTA(NUMERO*,NUMERO*,NUMERO*);

void Resta(NUMERO*,UINT4,NUMERO*);
```

La primera de las dos funciones realiza la resta de los dos primeros parámetros, direcciones de variables de tipo `NUMERO`, y deja el resultado en la dirección de la variable que viene como tercer parámetro. La segunda función está definida para realizar la resta de una variable tipo `NUMERO` con un entero de 32 bits (tipo `UINT4`); el tercer parámetro de la función es la variable donde quedará almacenado el valor del resultado de la operación.

La primera función viene definida de manera que los tres parámetros pueden direccionar a una misma variable; o dos parámetros a una misma variable y el tercero a otra diferente; o los tres parámetros direccionar a variables distintas.

4.3.2.3. Implementación de la operación para el producto.

Para realizar el producto de dos enteros es útil observar la forma que adquiere el desarrollo de sus expansiones. Si deseamos calcular $z = a \cdot b$, donde

$$a = a_k \cdot B^k + a_{k-1} \cdot B^{k-1} + a_{k-2} \cdot B^{k-2} + ... + a_1 \cdot B^1 + a_0$$

$$b = b_k \cdot B^k + b_{k-1} \cdot B^{k-1} + b_{k-2} \cdot B^{k-2} + ... + b_1 \cdot B^1 + b_0$$

(no necesariamente $a_k \neq 0$ o $b_k \neq 0$).

Tenemos que

$$a \cdot b = a \cdot \sum_{j=0}^{k} b_j \cdot B^j = \sum_{i=0}^{k} a_i \cdot (\sum_{j=0}^{k} b_j \cdot B^j) \cdot B^i = \sum_{i=0}^{k} b_i \cdot (\sum_{j=0}^{k} a_j \cdot B^j) \cdot B^i = b \cdot \sum_{j=0}^{k} a_j \cdot B^j = b \cdot a$$

Es interesante analizar esta expresión considerando ahora que $B = 2$, es decir, que trabajamos en base binaria. En ese caso tendremos que los valores de a_i son 0 ó 1. Entonces tenemos que el producto se reduce a realizar un máximo de k sumas y k desplazamientos.

$$a \cdot b = \sum_{j=0}^{k} a \cdot B^j \cdot b_j = \sum_{j=0}^{k} b \cdot B^j \cdot a_j$$

(donde a_j y b_j valen 0 ó 1).

Gracias a la propiedad conmutativa del producto podemos elegir entre rastrear los ceros y unos sobre a ó hacerlo sobre b. La conveniencia de uno u otro camino dependerá

1. de la cantidad de dígitos distintos de cero que tenga cada uno de los dos números: unos en a (lo llamaremos U_a) y unos en b (lo llamaremos U_b)
2. De la longitud de cada uno de los dos números: dígitos en a (lo llamaremos B_a) y dígitos en b (lo llamaremos B_b)

Dos formas hemos implementado para el producto.

1. En la primera nos hemos planteado como objetivo tomar aquel orden de factores que logre minimizar las operaciones desplazamiento a izquierda y las operaciones suma. Y procurar además que los sumandos sean los menores posibles. En esta primera implementación definimos la operación producto de manera que se realizase según el menor de los dos valores: $U_a \times B_b$ y $U_b \times B_a$

   Si $U_a \times B_b < U_b \times B_a$, entonces

   $$a \cdot b = \sum_{j=0}^{k} b \cdot B^j \cdot a_j$$

   En caso contrario

$$a \cdot b = \sum_{j=0}^{k} a \cdot B^j \cdot b_j$$

La desventaja de esta implementación es que si bien el producto queda minimizado, se hace necesario determinar la cantidad de unos de cada uno de los dos factores, y luego se debe realizar el producto de cada uno de estos dos valores con el valor de los campos `B` del otro factor. El producto está minimizado, pero el trabajo para decidir cuál es la forma mínima de realizar este producto es lo suficientemente gravosa como para que al final este procedimiento no ofrezca una mejora significativa en los tiempos de ejecución.

2. En la segunda hemos evitado la desventaja señalada en la primera versión y hemos definido un procedimiento más sencillo: rastrear los unos del más corto de los dos enteros a multiplicar.

   Si $B_a < B_b$, entonces

$$a \cdot b = \sum_{j=0}^{k} b \cdot B^j \cdot a_j$$

   En caso contrario

$$a \cdot b = \sum_{j=0}^{k} a \cdot B^j \cdot b_j$$

Hemos probado ambas implementaciones y después de hacer el producto con bastantes factores de diferentes tamaños y valores hemos tomado como más rápida la versión señalada en el punto 2.

Para la implementación de esta operación, como queda visto en estas líneas previas, dejamos de trabajar en base $B = 2^{32}$ y pasamos a hacerlo en la base $B = 2$. Esta operación va a requerir las definiciones de la operación desplazamiento a la izquierda y la operación suma, ya presentadas.

Para la operación del producto hemos definido dos funciones:

```
void PROD_bit(NUMERO*,NUMERO*,NUMERO*);
```

```
void Prod_bit(NUMERO*,UINT4,NUMERO*);
```

La primera de las dos funciones realiza el producto de los dos primeros parámetros, direcciones de variables de tipo `NUMERO`, y deja el resultado en la dirección de la variable que viene como tercer parámetro. La segunda función está definida para realizar la multiplicación de una variable tipo `NUMERO` con un entero de 32 bits (tipo `UINT4`); el tercer parámetro de la función es la variable donde quedará almacenado el valor del resultado de la operación.

La primera función viene definida de manera que los tres parámetros pueden direccionar a una misma variable; o dos parámetros a una misma variable y el tercero a otra diferente; o los tres parámetros direccionar a variables distintas.

4.3.2.4. Implementación de la operación para la división.

Trabajaremos en esta sección dos operaciones muy semejantes: el cálculo del cociente de dos enteros y el cálculo del módulo o resto de una división.

La operación cociente de dos enteros D (dividendo) y d (divisor) calcula el mayor entero c que verifique que $D \geq d \cdot c$ y que $D < d \cdot (c+1)$. Al valor c se le llama **cociente de los enteros** D y d.

La operación **módulo de dos enteros** D y d busca el entero r, del conjunto de residuos formado por los residuos menores no negativos, congruente con $D \bmod d$. A este valor r se le conoce con el nombre de resto de dividir D con d.

Ambas operaciones están íntimamente relacionadas. A partir del entero dividendo D y del entero divisor d, los valores cociente c y resto r verifican que $D = d \cdot c + r$. Se cumple que $r < d$, de lo contrario ($r \geq d$) además de que el residuo estaría fuera del rango de nuestro conjunto de residuos, tendríamos que existiría una valor $c' = c+1$ tal que $D \geq d \cdot c' = d \cdot (c+1)$ que es contrario a la definición antes dada del cociente de dos enteros. Evidentemente, si $D < d$, entonces directamente asignamos $c = 0$ y $r = D$. Entendemos por operación división aquella que está destinada a obtener a la vez los valores del cociente y del resto.

Las operaciones cociente y módulo no son en absoluto inmediatas, y el proceso a seguir para el cálculo de los valores de c y r es el más lento de todos los definidos para las operaciones aritméticas. Como señalan Bruce SCHNEIER y Dong WHITING [Schn97] el producto y el cociente son operaciones que requieren múltiples clocks y pueden atascar el proceso de segmentación, retrasando la ejecución de las siguientes instrucciones.

La operación división se realiza mediante una serie de suboperaciones más simples. Suponiendo que d tiene k dígitos binarios y que D tiene $N = k + n$ dígitos binarios, tomamos los k dígitos más significativos de D y definimos el número D_n. Entonces calculamos

$$D_n = d \cdot C_n + R_n$$

obteniendo así el dígito más significativo del cociente c (C_n), que será 0 si $D_n < d$, ó 1 si $D_n \geq d$.

Proseguimos con una nueva división tomando como dividendo el valor D_{n-1} formado por la cifra R_n a la que se le añade a su derecha el dígito (el bit) de D más significativo después de los k primeros dígitos ya tomados. Realizamos entonces una segunda división que resultará

$$D_{n-1} = d \cdot C_{n-1} + R_{n-1}$$

obteniendo así el segundo dígito más significativo del cociente c (C_{n-1}).

Se siguen realizando divisiones como las definidas de forma que cada vez tenemos que

$$D_{n-j} = d \cdot C_{n-j} + R_{n-j},$$

obteniendo así el j–ésimo dígito más significativo del cociente c (C_{n-j}) y donde el valor D_{n-j} está formado por la cifra R_{n-j-1} a la que se ha añadido a su derecha el dígito de D más significativo

después de los $k+j-1$ primeros dígitos ya tomados.

En el último dígito (el menos significativo) de D realizamos el último cociente:

$$D_0 = d \cdot C_0 + R_0$$

que nos ofrece el último dígito (el menos significativo del cociente c) y terminamos el proceso. El valor R_0 resulta ser el valor del resto r. El número c queda $(C_n C_{n-1} ... C_1 C_0)_B$.

Por tanto el cociente de dos números exigirá $n+1$ divisiones sencillas. El cociente tendrá $n+1$ dígitos, donde el más significativo será cero si D_n resulta ser menor que d, es decir, si los k primeros dígitos de D codifican un número menor que d.

Este proceso exige en cada paso un desplazamiento a izquierda de la cifra R_i, una lectura de dígito en D y de asignación en el extremo derecha de R_i, y una operación resta entre R_i y $C_i \cdot d$.

La implementación ha quedado hecha en base $B=2$. Por lo tanto, cada dígito C_i será 1 si $R_i >= d$, y será 0 en caso contrario. Se realizarán por tanto n desplazamientos a izquierda de 1 bit, n lecturas de bit de D mediante una operación AND y n asignaciones de bit para la creación de los sucesivos R_i mediante una operación OR. Se realizarán tantas restas como dígitos 1 tenga el cociente hallado.

Para la operación del cociente hemos definido dos funciones:

```
void COCIENTE(NUMERO*,NUMERO*,NUMERO*);
```

```
void Cociente(NUMERO*,UINT4,NUMERO*);
```

La primera de las dos funciones realiza el cociente del primer parámetro, que será el dividendo, con el segundo parámetro, que será el divisor, y deja el resultado en el tercer parámetro. Los tres parámetros son direcciones de variables de tipo `NUMERO`. La segunda función está definida para realizar el cociente entre un dividendo, variable tipo `NUMERO` con un divisor entero de 32 bits (tipo `UINT4`); el tercer parámetro de la función es la variable donde quedará almacenado el valor del resultado de la operación.

Para la operación del módulo hemos definido dos funciones:

```
void MODULO(NUMERO*,NUMERO*,NUMERO*);
```

```
UINT4 Modulo(NUMERO*,UINT4);
```

La primera de las dos funciones realiza la división del primer parámetro, que será el dividendo, con el segundo parámetro, que será el divisor, y deja el valor del resto de la operación en el tercer parámetro. Los tres parámetros son direcciones de variables de tipo `NUMERO`. La segunda función está definida para calcular el módulo de la división entre un dividendo, variable tipo `NUMERO` con un divisor entero de 32 bits (tipo `UINT4`); el resultado será un valor menor que el divisor, y por tanto

cabrá en una variable de tipo `UINT4`, que es lo que devuelve la función mediante una sentencia **`return`**.

Las funciones `COCIENTE()` y `MODULO()` vienen definidas de manera que sus tres parámetros pueden direccionar a una misma variable; o dos parámetros a una misma variable y el tercero a otra diferente; o los tres parámetros direccionar a variables distintas.

Las operaciones `Cociente()` y `Modulo()` se realizan una cantidad enorme de veces en el proceso de factorización basado en CFRAC y que hemos implementado. Ambas funciones han sido posteriormente largamente analizadas y modificadas para lograr reducir el número de instrucciones a nivel de código máquina y los ciclos de reloj necesarios para su ejecución.

4.4. OTRAS FUNCIONES MATEMÁTICAS.

El cálculo del máximo común divisor de varios enteros puede ser considerado como una operación matemática básica. Especialmente interesa para encontrar los casos en que dos números tienen su máximo común divisor igual a 1, es decir, números que llamamos **coprimos**. También es un algoritmo necesario para calcular la clave secreta del criptosistema RSA a partir de la clave pública y del valor de la función de EULER del módulo.

El cálculo de potencias donde tanto la base como el exponente son enteros largos es la herramienta básica para el desarrollo del algoritmo del criptosistema RSA. También es una operación necesaria para los tests de primalidad de MILLER–RABIN o de SOLOVAY–STRASSEN que veremos más adelante.

Y la operación del cálculo de la parte entera de la raíz cuadrada de una entero largo es necesaria en el algoritmo de BHÁSCARA para el desarrollo de la aproximación por fracciones continuas de la raíz cuadrada de un entero; este desarrollo es la base de uno de los algoritmos principales de factorización: El algoritmo de las fracciones continuas (CFRAC).

Ninguna de las tres funciones se utiliza de forma masiva en los procesos de factorización. No hemos realizado por tanto un análisis a fondo de estos algoritmos que en ningún caso suponen más de un 0,2 % del tiempo total de proceso.

4.4.1. Algoritmo de EUCLIDES: cálculo del máximo común divisor.

Para el cálculo del máximo común divisor disponemos del algoritmo de EUCLIDES (cfr. [Rose93], [Bres89] y [Cohe93] entre otras muchas referencias para este algoritmo), basado en el teorema de la división de EUCLIDES. Este teorema demuestra que dados dos números enteros $a > b > 0$, se verifica que $mcd(a,b) = mcd(b,r)$ siendo r el resto de dividir a entre b.

El número de divisiones (n) que se necesita para encontrar el máximo común divisor de dos

enteros positivos usando el algoritmo de EUCLIDES (cfr. Teorema de LAME: [Rose93]) es tal que $n \leq 5 \cdot k$, donde k es el número de dígitos, en base decimal, del menor de los dos números. El cálculo del máximo común divisor de dos enteros positivos a y b, con $a > b$ se puede realizar usando $O((\log_2 a)^3)$ operaciones a nivel de bit.

La implementación del algoritmo de EUCLIDES, de aspecto muy reducido cuando trabajamos con variables de tipo básico en el lenguaje de programación que hemos utilizado, queda más largo y complejo al tener que manejar variables de tipo `NUMERO`.

Para esta operación ha quedado definida la función

```
void EUCLIDES(NUMERO*,NUMERO*,NUMERO*);
```

Donde los dos primeros parámetros recogen las direcciones de los dos valores sobre los que se realizará el cálculo de su máximo común divisor, que quedará almacenado en la dirección del tercer parámetro. La función viene definida de manera que los tres parámetros pueden direccionar a una misma variable; o dos parámetros a una misma variable y el tercero a otra diferente; o los tres parámetros direccionar a variables distintas.

4.4.2. Cálculo de la parte entera de la raíz cuadrada de un entero.

Para la operación de cálculo de la parte entera de la raíz cuadrada de un entero positivo hemos tomado un algoritmo de [Cohe93] que presenta una variante del método de NEWTON, y trabaja únicamente con números enteros en todo su desarrollo.

1	[Inicializar]	Asignar $x \leftarrow n$
2	[Paso de NEWTON]	asignar $y \leftarrow \lfloor (x + \lfloor n/x \rfloor)/2 \rfloor$ usando división de enteros y desplazamientos
3	[¿Finalizar?]	Si $y < x$, asignar $x \leftarrow y$ y volver al paso 2. Si no dar como salida el valor de x y finalizar el algoritmo.

Algoritmo 1: Algoritmo para el cálculo de la parte entera de la raíz cuadrada de un entero largo.

Dado un entero positivo n, el siguiente algoritmo computa la parte entera de la raíz cuadrada de n; es decir, el número m tal que $m^2 \leq n < (m+1)^2$. El algoritmo tiene la secuencia de instrucciones recogidas en Algoritmo 1.

Para esta operación ha quedado definida la función

```
void RaizCuadrada(NUMERO*,NUMERO*);
```

La función recoge la variable cuya dirección viene dada en el primer parámetro y deja el valor de la parte entera de la raíz cuadrada en la dirección que recoge el segundo parámetro. La función

viene definida de manera que ambos parámetros pueden direccionar a una misma variable.

4.4.3. Cálculo de la potencia con base y exponente enteros largos.

Nos referimos en este epígrafe al cálculo de la potencia dentro de la aritmética modular: es decir, que trabajamos en el campo numérico formado por los residuos del conjunto completo de los residuos menores positivos: $\{0,1,2,...,n-1\}$.

El proceso de potencia no puede realizarse a fuerza de multiplicar la base por sí misma tantas veces como indique el exponente: si el exponente tiene un valor enorme como suele ocurrir habitualmente con los enteros largos, el cálculo de la potencia podría eternizarse.

Para realizar el cálculo de una expresión como $a^x \bmod n$ se pueden utilizar algunas técnicas que reducen el número de productos a efectuar. Una técnica muy usada es tomar módulo en sucesivos pasos intermedios. Por ejemplo:

$$a^8 \bmod n = (a \cdot a \cdot a \cdot a \cdot a \cdot a \cdot a \cdot a) \bmod n$$

pero también podemos decir que

$$a^8 \bmod n = ((a^2 (\bmod n))^2 \bmod n)^2 \bmod n$$

Y si el exponente no es potencia de 2, hay que introducir simplemente una sencilla modificación. Por ejemplo, el número 25 queda representado en binario como 11001, es decir: $2^{25} = 2^{16} + 2^8 + 2^1$, y por tanto:

$$a^{25} \bmod n = (a \cdot a^8 \cdot a^{16}) \bmod n =$$
$$= ((((((a^2 \bmod n \cdot a) \bmod n)^2 \bmod n)^2 \bmod n)^2 \bmod n) \cdot a) \bmod n$$

Necesitamos únicamente seis multiplicaciones.

Información sobre el proceso de exponenciación rápida podemos encontrarla, por ejemplo, en [Bres89]. De esa referencia hemos tomado el algoritmo recogido en Algoritmo 2, para el cálculo de $d = a^b \bmod n$, para $b \geq 0$.

1	Asignar	$d \leftarrow 1$
2	MIENTRAS $b \neq 0$	SI b es IMPAR ENTONCES $d \leftarrow (d \times a)\%n$
		$b \leftarrow \lfloor b/2 \rfloor$ (mediante desplazamiento)
		$a \leftarrow (a \times a)\%n$
3	Devolver	d

Algoritmo 2: Algoritmo para el cálculo de potencias, donde tanto la base como el exponente pueden ser enteros grandes.

Otra referencia bibliográfica que presenta diferentes algoritmos e implementaciones para esta

operación es [Schn96].

Para esta operación ha quedado definida la función

```
void EXP_MOD(NUMERO*,NUMERO*,NUMERO*,NUMERO*);
```

La función viene definida de manera que cada uno de los parámetros puede direccionar a diferentes variables, o cualquiera dos, o tres, o todos ellos, pueden coincidir en la dirección que recogen.

4.5. FUNCIONES PARA LOS NÚMEROS PRIMOS.

4.5.1. Algoritmo para el cálculo de los símbolos de LEGENDRE y JACOBI.

Para la implementación del algoritmo de cálculo del símbolo de LEGENDRE y de JACOBI, de cuya utilidad y necesidad ya se ha hablado en el Capítulo 2, hemos seguido dos referencias bibliográficas: [Bres89], de donde hemos tomado la idea de la función `QuitarDoses()`, que presentamos en las siguientes líneas, y [Schn96]. Han quedado definidas dos funciones, con los siguientes prototipos y los algoritmos que las definen, recogidos en Algoritmo 3 (para la función `Símbolo()`) y Algoritmo 4 (para la función `QuitarDoses()`).

1	INICIO:	Leer n (entero) y p (primo). $legendre \leftarrow 1$ $n \leftarrow n \bmod p$
2	PROCESO:	SI $n = 0$ (es decir, si p divide a n) $legendre \leftarrow 0$ ir a FIN: SI ($p \bmod 4 = 3$) $legendre \leftarrow -1$
3	REPETIR:	Llamar a la función `QuitarDoses()` MIENTRAS ($n > 1$) HACER SI $(n-1)\times(p-1) \bmod 8 = 4$) ENTONCES $legendre \leftarrow -1 \times legendre$ $temp \leftarrow n$ $n \leftarrow p \bmod n$ $p \leftarrow temp$ Llamar a la función `QuitarDoses()`
4	FIN:	Devolver $legendre$

Algoritmo 3: Algoritmo para el cálculo del símbolo de LEGENDRE de un entero cualquiera con respecto a un primo (tomado de [Bres89]).

```
SINT2 Simbolo(NUMERO*,NUMERO*);

void QuitarDoses(NUMERO*,NUMERO*,SINT2*);
```

La función `QuitarDoses()` elimina todos los factores de 2 de la variable n. Si n es todavía mayor que 1, intercambiamos n y p, reduciendo el nuevo valor de n módulo p y quitándole los factores potencia de dos. Esto sigue iterado hasta que n valga 1. La variable $count$ guarda la paridad del exponente del 2 en n. La función devuelve los nuevos valores de n y de $legendre$.

La otra presentación que hemos tenido en cuenta para redactar nuestra implementación es el algoritmo presentado en [Schn96] (hemos mantenido la notación presentada en las obras citadas). Aquí $S(a,b)$ quiere significar el valor del símbolo de LEGENDRE (a/b). Este algoritmo queda recogido en Algoritmo 5.

1 INICIO: $count \leftarrow 0$
2 PROCESO: MIENTRAS (n PAR) HACER
$n \leftarrow n/2$
$count \leftarrow 1 - count$
SI ($count \times (p \times p - 1) \bmod 16 = 8$)
ENTONCES $legendre \leftarrow -1 \times legendre$

Algoritmo 4: Función auxiliar utilizada el Algoritmo 3, llamada `QuitarDoses()`.

Hacemos dos sencillas observaciones sobre la implementación de estas funciones:

Para la condición definida en el paso 4 del algoritmo:

Si $b^2 - 1/8$ es PAR, entonces $S(a,b) = S(a/2,b)$

Si $b^2 - 1/8$ es IMPAR, entonces $S(a,b) = -S(a/2,b)$

Lo que en realidad hace el código definido es calcular el dígito (trabajamos en base $B = 2^{32}$) menos significativo de $b^2 - 1$. Si este dígito tiene un uno en el cuarto bit menos significativo querrá decir que si dividiéramos por 8 el número entero el resultado quedaría impar; si tiene un cero en el cuarto bit menos significativo querrá decir que si dividiéramos por 8 el número entero el resultado quedaría par.

Para la operación de cálculo del símbolo, en los pasos 7, 8 y 9 del algoritmo tomado de [Schn96] viene la siguiente decisión:

Si $(a-1) \times (b-1)/4$ es PAR, entonces $S(a,b) = S(b,a)$

Si $(a-1) \times (b-1)/4$ es IMPAR, entonces $S(a,b) = -S(b,a)$

1 $a \leftarrow a \bmod b$
2 SI $a = 0$, ENTONCES $S(0,b) = 0$ y TERMINAR
3 SI $a = 1$, ENTONCES $S(1,b) = +1$ y TERMINAR
4 MIENTRAS que a sea PAR HACER
SI $b^2 - 1/8$ es PAR, ENTONCES $S(a,b) = S(a/2,b)$
SI $b^2 - 1/8$ es IMPAR, ENTONCES $S(a,b) = -S(a/2,b)$
5 $g = mcd(a,b)$
6 SI $g \neq 1$ ENTONCES
SI $g = a$ ENTONCES $S(a,b) = 0$ y TERMINAR
SINO $S(a,b) = S(a/g,b)$
7 $(a-1)\times(b-1)/4$
8 SI es PAR, ENTONCES $S(a,b) = S(b,a)$
9 SI es IMPAR, ENTONCES $S(a,b) = -S(b,a)$

Algoritmo 5: Algoritmo para el cálculo del símbolo de LEGENDRE de un entero cualquiera n con respecto a un primo p (tomado de [Schn96]). El paso 4 del Algoritmo 5 es el que queda definido con la función `QuitarDoses()`.

Y al igual que antes, lo que en realidad hace el código es calcular el dígito menos significativo de $(a-1)\times(b-1)$. Si este dígito tiene un uno en el tercer bit menos significativo querrá decir que si dividiéramos por 4 el número entero el resultado quedaría impar; si tiene un cero en el tercer bit menos significativo querrá decir que si dividiéramos por 4 el número entero el resultado quedaría par.

4.5.2. Test de las divisiones sucesivas. Criba de ERASTHÓTENES.

El test más sencillo de primalidad consiste en dividir el entero n por todos los primos menores que su raíz cuadrada

$$p_i \leq \sqrt{n}\,.$$

Este método es enormemente costoso en tiempo de computación, pero resulta ser el más útil cuando trabajamos con enteros pequeños (no mayores de 10^7).

Este método de intento por divisiones sucesivas puede resultar muy útil como primer estudio del candidato a primo. Como señala Hans RIESEL [Ries87], el 76 % de los enteros impares tienen un factor primo menor que 100. Haciendo un intento de división del entero n por los primos más pequeños obtenemos una primera aproximación a la condición de primalidad del entero n investigado. Si el entero n candidato a primo es dividido por algún primo menor que un límite dado entonces queda claro que ese número es compuesto. Si ninguno de esos primos divide a nuestro candidato a primo entonces podemos someterlo a otros tests de primalidad más trabajosos y que ofrecen mayor grado de

certeza sobre la condición de primalidad.

Se comienza generando una tabla con todos los números desde 1 hasta el límite superior introducido. El número 1 es un entero especial, que constituye el elemento neutro del producto y que no consideramos ni primo ni compuesto. Se toma el primer número después del 1, el 2, que ya desde ese momento se considerará primo, y se procede a borrar de la tabla todos los números posteriores a 2 y múltiplos de 2.

A continuación pasamos al siguiente número aún habilitado, que resulta ser el 3, por lo que queda considerado ya como primo, y procedemos a tachar todos los números posteriores a él y múltiplos suyos.

Ya hemos eliminado todos los múltiplos de 3. Ahora buscamos el siguiente número habilitado, que resulta ser el 5. El 5 queda considerado primo y ahora se procede a eliminar todos los múltiplos de 5 mayores que éste.

El proceso se va repitiendo hasta llegar al mayor entero menor que la raíz cuadrada del límite del campo a cribar. Todos los números que no hayan sido eliminados serán primos.

El modo de generar la tabla de los números primos que hemos implementado consiste en crear, mediante asignación dinámica de memoria o mediante un vector de dimensión predeterminada, una cadena de tantos elementos tipo **`char`** como requiera el usuario (límite superior del cuadro sobre el que determinar los primos). Al final del proceso de criba ese vector tipo **`char`** queda de tal manera que si el índice de la posición es primo el valor del carácter sea '`p`', y si el índice de la posición es compuesto el valor del carácter sea '`c`'.

Para todo este proceso hemos definido tres funciones:

La función **`void criba(char*);`**

que recibe como parámetro la dirección de la cadena de caracteres donde la función `criba()` deberá dejar el valor `'c'` en las posiciones de índice compuesto, y el valor `'p'` en las posiciones de índice primo.

La función `UINT4* GenerarMatrizPrimos(UINT4*)`

que recibe como parámetro una variable puntero donde la función asignará la dirección de un vector de enteros `UINT4`. La función invoca a la función `criba()` y cuenta cuántos índices de la cadena de caracteres cribada valen '`p`'. A continuación asigna al puntero recibido como parámetro un vector de enteros `UINT4` y copia en ese vector los valores de los índices de la cadena de caracteres con valor del carácter '`p`'. La función devuelve la dirección donde ha quedado posicionado el vector de enteros primos.

La función `UINT4 PrimosPequenyos(NUMERO*,UINT4*)`

recibe como parámetros la dirección de una variable tipo `NUMERO` de cuyo valor queremos determinar

si es divisible por alguno de los primos generados en la función anterior y que viene recogida en el segundo parámetro. Devuelve el primer valor primo que logra dividir al valor de la variable `NUMERO`; si ningún primo lo logra devuelve el valor 0.

4.5.3. Presentación del algoritmo de MILLER–RABIN.

El algoritmo del test de MILLER–RABIN ha quedado implementado con la función

```
UINT2 Rabin(NUMERO*,UINT4*);
```

El primer parámetro es la dirección de la variable tipo `NUMERO`, de cuyo valor queremos determinar si es o no es primo. El segundo parámetro recoge una dirección de un vector de los primeros primos. La función realiza el proceso de test repetidas veces. Cada vez que pase en test para una nueva base (base que tomará del vector de primos recibido como segundo parámetro) se multiplica la probabilidad de que el valor de la variable `NUMERO` sea primo.

1. Dado N, definir $N-1=2^s \cdot r$, donde r es impar
2. DESDE $i=1$ HASTA `MillerRabin` HACER
2.1. Tomar un a tal que $mcd(a,n)=1$,
2.2. Calcular $y=a^r \pmod N$.
2.3. SI($y \neq 1$ Y $y \neq N-1$) HACER
2.3.1. $j=1$
2.3.2. MIENTRAS($j \leq s-1$ Y $y \neq N-1$) HACER
2.3.2.1. $y \leftarrow y^2 \pmod N$
2.3.2.2. SI($y=1$) `return(1)`: COMPUESTO
2.3.2.3. $j \leftarrow j+1$
2.3.3. SI($y \neq N-1$) `return(1)`: COMPUESTO
2. `return(0)`: PRIMO

Algoritmo 6: Test de Miller–Rabin.

Bruce SCHNEIER [Schn96] dice que basta pasar el test 5 veces. En el libro de MENESSES [Mene97] se afirma que para números de menos de 1000 bits basta con hacer el test 3 veces. Teniendo en cuenta que la probabilidad de que un entero que pasa una vez el test sea compuesto es menor que $1/4$, para lograr que la probabilidad de que el número sea primo sea de al menos 99,99 % debemos pasar el test 7 veces: $(1-(1/4)^x)\times 100 > 99{,}99$ cuando llegamos al valor de $x=7$. Ese es el valor (número de veces que debe pasar el test) que hemos dado finalmente a nuestra función. Este valor viene recogido en una macro, o valor constante, que hemos llamado `MillerRabin`.

Nuestra función devuelve el valor 1 si el valor de la variable `NUMERO` no ha superado el test de primalidad y podemos por tanto asegurar que es compuesto; devuelve el valor 0 si supera el test de

primalidad en todas las bases probadas. El algoritmo de MILLER–RABIN queda recogido en Algoritmo 6.

4.5.4. Primos especiales: primos fuertes y primos doblemente seguros.

Como ya hemos señalado en el Capítulo 2, está recomendado el uso de primos fuertes en la implementación del criptosistema RSA. Disponemos de algoritmos que ofrecen caminos para encontrar primos de estas características. Algunos de ellos vienen recogidos en [Rive99]: el algoritmo de WILLIAMS–SCHMID ó el algoritmo de GORDON [Gord84].

Existe otro algoritmo que requiere del manejo de primos especiales: El generador de pseudoaleatorios BBS [Blum86], que veremos desarrollado en otro Capítulo de la tesis dedicado a la generación de secuencias aleatorias y pseudoaleatorias. En [Hern96] se indica que el generador BBS requiere, para lograr órbitas máximas, el uso de primos 2–seguro. Entendemos por **primo 2–seguro** aquel entero p que verifica las siguientes propiedades:

1. p es primo largo.
2. $p = 2 \times p' + 1$, donde p' es primo.
3. $p' = 2 \times p'' + 1$, donde p'' es primo.

La búsqueda de primos especiales como los dos referidos exige un trato especial en los tests de primalidad. Por ejemplo, para la búsqueda de primos 2–seguro debemos buscar un entero p'' que superase el test de primalidad; que al multiplicarlo por 2 y sumarle 1 obtengamos un p' que también supere el test de primalidad; y que al multiplicar p' por 2 y sumarle 1 obtengamos un p que también supere el test de primalidad. La probabilidad de que el candidato a primo 2–seguro supere los tres tests es baja, y la búsqueda de primos de estas características exige bastante tiempo de cómputo.

Para estos procesos de búsqueda de primos especiales, que deben verificar acumulativamente diferentes propiedades hemos definido una función, semejante a la anteriormente mostrada para el test de MILLER–RABIN, pero que realiza el test solo para la base 2. Si finalmente, el candidato a primo especial verifica todos los requisitos exigidos (superar el test reducido para p, p' y p''), entonces se repite la evaluación de las condiciones de primalidad, con el algoritmo completo del test de MILLER–RABIN. La función que hemos definido para este proceso la hemos llamado

```
UINT2 FastRabin(NUMERO*);
```

y no recibe como segundo parámetro un vector de primos porque realiza el test únicamente para la base 2. Al igual que la función anterior, devuelve el valor 1 si el valor de la variable `NUMERO` no ha superado el test de primalidad y podemos por tanto asegurar que es compuesto; devuelve el valor 0 si supera el test de primalidad en la base 2.

4.6. REFLEXIONES FINALES.

Pretendemos hacer un escueto balance del trabajo presentado en este Capítulo. No se trata de una recopilación. Se trata de comentar diferentes aspecto de nuestro trabajo: comprobaciones al código implementado; algún comentario sobre las futuras optimizaciones, que quedan largamente recogidas en el Capítulo 7. Nos gustaría poder explicar por qué hemos querido implementar nuestra propia librería y por qué hemos trabajado en C.

Todas las funciones presentadas en este Capítulo han sido probadas una gran cantidad de veces. Si se tiene en cuenta, por ejemplo, que la función `SUMA()` interviene un orden de 600.000 veces en el proceso de factorización de un entero producto de dos primos de 50 bits; o un orden de 7 millones de veces cuando el tamaño de los primos es de 60 bits; o un orden de 40 millones de veces cuando los primos son de 70 bits... Y si se tiene en cuenta que hemos probado nuestra aplicación de factorización para factorizar más de mil enteros para cada tamaño de los primos entre 32 y 80 bits... podemos dar por suficientemente probada la función referida.

	40	50	60	70
CrearNumero()	180.000	850.000	9.600.000	59.600.000
CopiarNumero()	340.000	1.600.000	18.700.000	117.000.000
PonerACero()	620.000	3.000.000	34.000.000	212.000.000
longitud()	270.000	1.300.000	13.100.000	78.000.000
orden()	18.000.000	48.000.000	255.000.000	55.000.000
DESPL_izda()	2.000.000	13.000.000	176.000.000	1.200.000.000
SUMA()	120.000	600.000	6.700.000	42.020.000
RESTA()	120.000	590.000	5.600.000	32.000.000
PROD_bit()	40.000	240.000	2.800.000	17.000.000
COCIENTE()	10.000	80.000	960.000	5.000.000
Cociente()	20.000	140.000	630.000	4.000.000
MODULO()	40.000	170.000	1.900.000	12.000.000
Modulo()	2.000.000	14.000.000	170.000.000	1.700.000.000

Tabla 1: Número de llamadas (en orden de magnitud) de las funciones presentadas en este Capítulo (las más invocadas) en la aplicación de factorización de enteros producto de dos primos según los tamaños de cada uno de los dos primos.

En la Tabla 1 mostramos el número de veces (en órdenes de magnitud) que es invocada cada una de las funciones que hemos ido relacionando en este Capítulo cuando ejecutamos nuestra aplicación de factorización mediante la técnica de CFRAC que presentamos más adelante, en el Capítulo 6. Recogemos esos números de llamadas a funciones para procesos de factorización de enteros producto de dos primos de 40, 50, 60 y 70 bits.

Estos valores son los correspondientes al número de llamadas de cada función en la primera versión de nuestra implementación. Como se verá más adelante, en el Capítulo 7 dedicado a la optimización del código, uno de los objetivos perseguidos ha sido reducir el número de llamadas, lográndose en algunos casos unos descensos notablemente significativos.

El objeto de traer ahora aquí esta tabla es mostrar que las funciones implementadas son ampliamente usadas. Y si, como ya hemos dicho, hemos ejecutado nuestra aplicación de factorización miles de veces para cada tamaño de entero producto de dos primos, podemos considerar que las funciones relacionadas han sido largamente probadas. Y en ninguna de las ejecuciones hemos hallado un problema o una interrupción, o un resultado erróneo.

Las funciones que no aparecen en la relación de la Tabla 1 intervienen menos de 5.000 veces en la factorización del entero más pequeño aquí recogido (el que es producto de dos primos de 50 bits). Algunas de estas funciones que no aparecen son empleadas pocas veces en todas nuestras implementaciones. Las más, son funciones que sí se ejecutan repetidas veces en otras aplicaciones: especialmente en la que también hemos diseñado para la generación de secuencias pseudoaleatorias y que presentamos en el Capítulo 5.

Otra cuestión, que aquí solo anunciamos y que quedará desarrollada en el Capítulo 7, hace referencia a la necesidad de optimizar el código de las funciones presentadas en este Capítulo. Visto el número de veces que interviene cada una de ellas, se comprende la importancia que tiene lograr cualquier reducción, por pequeña que sea, en el número de instrucciones código máquina y, especialmente, en el número de ciclos de reloj que emplea cada una de estas funciones, en ejecutarse. No recogemos ahora más comentarios sobre esta tarea, pero en los Anexos (cfr. Anexo VI), al final de la tesis, pueden verse las reducciones de ciclos de reloj y de instrucciones en cada una de estas funciones y en otras muchas que se explican más adelante.

Y precisamente esta tarea desarrollada de optimización es una de las principales causas por las que nos hemos decidido a realizar nuestra propia implementación de enteros largos aquí presentada. Un código como el presentado por COHEN, el PARI, que contiene 65.000 líneas de código en C, resulta muy complicado de analizar y de optimizar al nivel al que hemos podido llegar con nuestro propio código. Insistimos en que no hemos pretendido crear una mejor implementación. Ni peor ni mejor: distinta; pero nuestra desde la primera línea. Y gracias a ello hemos podido profundizar en cada función, en cada bucle, en cada instrucción condicional, del código que hemos pretendido optimizar.

Queda justificar el uso del lenguaje de programación: el C. A esta pregunta sobre ¿por qué C? la primera respuesta que nos viene a la cabeza podría ser ¿y por qué no? En C ha sido escrito LiDia, PARI, MAPLE, Matemática. En C escribió LENSTRA su librería (FreeLIP) a la que hemos hecho referencia antes en este Capítulo. En C está escrito PGP. C es el lenguaje que emplea Bruce SCHNEIER para mostrar los algoritmos criptográficos en su libro [Schn96].

Por otro lado, para optimizar una aplicación y reducir sus ciclos de reloj o el número de instrucciones de código máquina, resulta imprescindible disponer de herramientas apropiadas. La que hemos utilizado (RABBIT) sirve para analizar el comportamiento de programas escritos en C y compilados en un sistema Linux. Quizá sea más sencillo avanzar en la construcción de algoritmos criptográficos si trabajamos directamente con alguna herramienta ya creada, como Matemática (así lo hace David BRESSOUD en [Bres00]). Pero esas implementaciones no hubieran podido ser optimizadas como sí hemos podido hacer nosotros con nuestras funciones.

5 DISEÑO E IMPLEMENTACIÓN DE GENERADORES DE BITS ALEATORIOS Y PSEUDOALEATORIOS

Una característica asombrosa de la criptografía es que reduce el problema de la protección de enormes cantidades de datos a la protección de una pequeña cantidad de información llamada clave; de la seguridad de esas claves depende la seguridad de toda la información cifrada.

En el corazón de todo sistema criptográfico se encuentra la necesidad de disponer de números secretos, imposibles de adivinar, origen de las claves utilizadas. La generación de valores secretos, aleatorios e impredecibles es, por eso, una tarea fundamental en toda herramienta diseñada para conferir seguridad.

La seguridad de muchos sistemas criptográficos depende de su capacidad para generar esas cantidades o valores impredecibles que servirán luego de claves. Todos los sistemas necesitan que esas cantidades generadas sean lo suficientemente grandes y lo suficientemente aleatorias; que la probabilidad de generar cualquier valor concreto sea lo suficientemente baja como para evitar que un adversario logre adivinarla mediante el uso de estrategias de búsqueda fundamentadas

en técnicas de cálculo de probabilidades.

Sin embargo, la generación de estas pequeñas cantidades de esa materia prima necesaria para la formación de claves no es tarea trivial. Reunir ciertas cantidades de valores aleatorios (verdaderamente aleatorios) requiere a veces una carga de tiempo que puede llegar a resultar inadmisible para muchas aplicaciones. La generación de secuencias de bits aleatorios constituye por eso un punto crítico en la criptografía.

Un generador de valores válidos para ser usados como claves criptográficas es, en resumidas cuentas y en una primera instancia y aproximación, un generador de secuencias de bits aleatorios donde cada posible valor resulta equiprobable e independiente de todos los anteriores valores generados. Deben ser generadores a los que no se les pueda detectar ni correlación (decimos que hay correlación si la probabilidad de que el generador emita un 1 depende del bit previo emitido) ni sesgo (decimos que hay sesgo si la probabilidad de que el bit generado sea el 1 no es la misma que la de que el bit generado sea el 0). Podemos decir que la calidad de una secuencia de bits aleatoria depende de la entropía de sus bits [Rifa91].

Podemos clasificar los generadores de bits aleatorios en tres grupos (ver por ejemplo [Schi02]):

1. **Generadores verdaderamente aleatorios** (TRBG). Aquellos cuyo origen y génesis de la aleatoriedad descansan en las fluctuaciones que sufren diferentes procesos. Como regla general, podemos afirmar que los mejores TRBG son los generadores físicos: por ejemplo el tiempo de emisión por descomposición radiactiva, medida con un contador Geiger; las fluctuaciones de la atmósfera, como grados de humedad, temperaturas, presiones; variaciones de temperatura en diferentes circuitos eléctricos; ruido electromagnético que puede detectarse en el entorno físico donde se encuentra el generador; la distribución térmica de semiconductores o resistores, la inestabilidad de un oscilador, la potencia de sonido en micrófonos o la intensidad de video de una cámara... Muchos de los sistemas de generación de valores verdaderamente aleatorios son componentes hardware, diseñados expresamente para ese fin.

   Sin embargo, y por razones técnicas, resultan más usados otros métodos, basados en software, y de acceso universal a todos los usuarios, y en los que también se hace uso de eventos en cierta medida independientes de la intencionalidad humana: el sistema de reloj del ordenador; el tiempo entre cada activación del teclado o el ratón; el contenido de buffers de entrada o salida [East94]; uso de algunas estadísticas del sistema operativo que no puedan ser observadas por un potencial oponente [Schw02], etc.

   Si un generador nos ofrece una secuencia de bits de longitud n, si el generador es verdaderamente (*True*) TRBG, entonces el valor numérico que codifica la secuencia (que estará en el rango $(0,2^n-1)$) es impredecible, y la probabilidad de acierto no es mayor que $1/2^n$. Y en el proceso de generación de la secuencia, si hemos generado m bits, ningún

observador puede predecir el siguiente bit $m+1$ con un probabilidad mayor de $1/2$ [Kels99].

No es sencillo disponer de abundantes entradas físicas aleatorias. Habitualmente, la cantidad de información aleatoria secreta a generar es grande y el método de obtención mediante eventos aleatorios externos resulta gravoso. Una buena opción consiste en obtener entradas aleatorias de diferentes fuentes disociadas e independientes y mezclarlas luego con alguna función fuerte de mezcla [Math96]. Entendemos por función de mezcla aquella que combina dos o más entradas y produce una salida donde cada bit de salida es función compleja no lineal de todos los bits de entrada y, de promedio, al cambiar un bit de la entrada cambian el 50 % de los bits de la salida.

2. **Generadores de bits pseudoaleatorios** (PRBG). Son generadores deterministas (también llamados DRBG). Generan secuencias de forma determinista y a muy buena velocidad, a partir de un valor inicial llamado semilla, generada con un generador físico aleatorio o con un generador híbrido.

   La entropía de la secuencia generada con un PRBG, por larga que sea esa secuencia, no será nunca mayor que la entropía de la semilla. Es imposible para un PRBG aumentar la entropía total de una secuencia inicial que se ha tomado como semilla [AIS20]. El PRBG puede definirse como una función l, que podemos llamar **expansiva**, que recibe una entrada binaria de longitud n y ofrece como salida una secuencia de longitud $l(n)$, que habitualmente cumple que $l(n) >> n$ (y de esa propiedad surge el interés de estos generadores) pero que resulta computacionalmente indistinguible de una secuencia de la misma longitud $l(n)$ tomada de una fuente física aleatoria [Schw02]. Una secuencia de bits obtenida mediante un PBRG puede usarse como si fuese auténticamente aleatoria, suponiendo que el adversario tiene una cantidad limitada de recursos computacionales y que la semilla es suficientemente grande como para frustrar ataques por fuerza bruta. Se dice que el PRBG expande la semilla seleccionada aleatoriamente dando lugar a una secuencia de bits más larga pseudoaleatoria.

   Con estos generadores siempre hay que tener presente que la salida $l(n)$ no es realmente aleatoria, y sólo podemos considerar aleatoria una porción $2^n/2^{l(n)}$ de todas las posibles secuencias binarias de longitud $l(n)$. Por ejemplo, si estamos utilizando un sistema criptográfico con claves de 128 bits; y las claves se obtienen utilizando un generador de secuencias de bits aleatorios con una semilla de 8 bits; entonces, y aunque aparentemente hay 2^{128} claves posibles, un adversario sólo debe probar las $2^8 = 256$ combinaciones posibles para la semilla (128 en promedio realmente). Es decir, solo tenemos 8 bits de información en esos 128 de clave: casi toda esta secuencia es, por tanto, redundante.

3. **Generadores híbridos**. Son PRBG que refrescan la semilla de forma regular. Cada nueva semilla se genera con un TRNG a partir de valores aleatorios tomados de la iteración con el usuario o

de valores registrados en el propio ordenador: mediante alguna aplicación de transformación (algoritmos de compresión) se procesa esas distintas entradas y se genera una nueva semilla. Este proceso intenta paliar la limitación de los PRBG de que la entropía de la salida es idénticamente igual a la entropía de la semilla de inicio.

Existen abundantes librerías de diferentes compiladores de muchos lenguajes de programación, que ofrecen al usuario sus funciones de generación de secuencias aleatorias. Muchas de estas herramientas son del todo inapropiadas para propósitos de seguridad. Estas librerías están diseñadas para ser usadas en simulaciones, en juegos, y otras aplicaciones de este estilo; pero no son lo suficientemente aleatorias para su uso en funciones de seguridad como por ejemplo, generación de claves, o la generación de secuencias para cifrados de flujo.

Existen diferentes procedimientos para calibrar la calidad de la aleatoriedad de una secuencia de bits. Desde un punto de vista criptográfico es imperativo, para cualquier sistema que se use para generar bits aleatorios, someter sus secuencias a algún análisis estadístico de aleatoriedad. Una definición criptográficamente fuerte de aleatoriedad exige que la secuencia sea indistinguible de una verdaderamente aleatoria uniformemente distribuida. Para determinar esta propiedad existe una larga lista de tests estadísticos, que permiten detectar defectos en las fuentes aleatorias testeadas. El hecho de que un generador supere y pase una lista determinada de tests estadísticos no permite certificar su condición de aleatoriedad de forma definitiva: El National Institute of Standars and Technology presenta una larga lista de tests (hasta 16) para los generadores de secuencias pseudoaleatorias [NIST00]. Otra publicación del mismo organismo gubernamental [FIP140] recoge los tests recomendados para los generadores de secuencias de bits aleatorios. Los mismos tests recomienda también el European Telecommunications Standards Institute para estos generadores [ETSI03]. Y son los mismos tests que recoge el libro de MENEZES [Mene97].

Y ejemplos de tests de aleatoriedad disponibles en el mercado son, por ejemplo, la aplicación comercial Crypt–X, que presenta e implementa 6 tests: no añade nuevos tests a los que han presentado todas las demás referencias traídas en estas líneas. No es de libre distribución. MARSAGLIA ofrece un CDROM lleno de bits aleatorios: 4.8 millardos de bits aleatorios almacenados en 60 archivos de 10 Mbytes cada uno. El autor del CD garantiza que los bits que vende han superado todos los tests estadísticos por él implementados, y que se conocen como Batería de tests DIEHARD, recogidos en [Mars96], de donde se puede obtenerse el código fuente en C con la implementación de cada uno de ellos.

Para las pruebas sobre nuestras implementaciones de funciones y operadores para los enteros grandes, y en nuestros estudios sobre el algoritmo de factorización basado en las fracciones continuas, hemos necesitado de una cantidad verdaderamente grande de valores enteros y, por tanto, de bits generados de forma aleatoria. Hemos necesitado disponer de algunas herramientas que nos generasen bits para formar esos enteros. Y hemos implementado dos generadores. Uno,

de secuencias de bits aleatorias, para alimentar –como ya hemos explicado– el segundo, de secuencias de bits pseudoaleatorias. A lo largo de este Capítulo presentamos estos dos generadores:

1. Para el primero hemos empleado la aleatoriedad que ofrece la entrada de teclado del usuario: un generador que depende de la tecla pulsada y de la cadencia de las pulsaciones. El uso para el que hemos diseñado el generador es la alimentación de la semilla inicial de un PRBG. A lo largo del presente Capítulo presentamos el algoritmo que hemos diseñado e implementado para que sirva de generador de bits aleatorios. Mostramos además los resultados obtenidos en el análisis de su aleatoriedad mediante los tests estadísticos a los que lo hemos sometido. Como veremos en su presentación, hemos testeado el generador con todos los tests recomendados en [Mene97] que, como acabamos de decir, son los tests más recomendados en todas las referencias que hemos consultado.
2. El otro, generador de secuencias de bits pseudoaleatorias, es el presentado en 1986 por BLUM, BLUM y SHUB (generador BBS) [Blum86]. Mostramos una implementación del generador BBS que hemos realizado con todas las funciones definidas y presentadas en el Capítulo 4. Aunque hemos probado el comportamiento estadístico de nuestra implementación pasando el test universal de MAURER (que ha superado), realmente no es necesario hacer pruebas sobre este generador que es reconocido por todo el mundo como un generador de uso criptográfico seguro. Por ejemplo, en el ya citado [ETSI03] se recomienda el uso de generadores basados en la exponenciación modular iterada en un módulo compuesto. Y expresamente recomienda el uso del PRBG BBS.

5.1. DESCRIPCIÓN DE UN GENERADOR DE SECUENCIAS DE BITS ALEATORIAS POR ENTRADAS DE TECLADO.

5.1.1. Presentación del generador implementado.

El generador que hemos definido toma sus valores de las distintas entradas del teclado realizadas por el usuario y de los tiempos empleados entre las sucesivas pulsaciones de las teclas.

En nuestro generador, el valor aleatorio i–ésimo generado depende de:

1. El valor del carácter i–ésimo introducido por teclado.
2. La diferencia de tiempos transcurrido entre la introducción del carácter (i – 1)–ésimo y la introducción de carácter i–ésimo. Para el cálculo de esos tiempos hemos utilizado la función `times()` de la biblioteca `sys/times.h`. Esta función devuelve un valor proporcional al número de pulsos de reloj transcurridos desde un instante arbitrario del pasado. En Linux, si el parámetro de la función es el puntero `NULL`, este instante es el momento del último arranque del sistema.

El número de pulsos por segundo que recoge la función viene señalado con el parámetro `_SC_CLK_TCK`.

3. El valor aleatorio previo generado.

VARIABLES: `unsigned long t_1,t_2,rot;`
`unsigned short aleat;`
`unsigned char letra;`

VALORES INICIALES:
`t_1 = times(NULL);`

REPETIR MIENTRAS NO SE PULSE LA TECLA ESCAPE:

1. Esperar pulsación de una tecla por parte del usuario. Cuando se produzca:
   - 1.1. `letra` igual al carácter introducido por teclado.
   - 1.2. `t_2 = times(NULL);`
2. Ajustar intervalos de tiempo...
   - 2.1. `dift = t_2 – t_1;`
   - 2.2. `t_1 = t_2;`
3. Primera rotación (a izquierdas): sobre el valor `t_2`.
   La rotación depende del intervalo de tiempo transcurrido (`dift`) y del valor previo de la variable `aleat`.
   - 3.1. `dift *= dift;`
   - 3.2. SI `aleat` DISTINTO DE `0`, ENTONCES `dift *= aleat;`
   - 3.3. `rot = dift % 31;`
   - 3.4. `t_2 *= t_2;`
   - 3.5. `t_2 = (t_2 >> rot | t_2 << (32 - rot));` (`t_2` variable de 32 bits).
4. Primera operación XOR sobre el valor aleatorio a generar.
   - 4.1. `aleat ^= t_2;` Se emplean los 16 bits menos significativos de la variable `t_2`. La variable `aleat` tendrá el valor final del proceso en la iteración anterior o un valor inicial cualquiera si estamos en la primera iteración.
5. Segunda rotación (a derechas): sobre el valor `aleat`.
   La rotación depende del valor ASCII de la tecla pulsada y del actual valor `aleat`.
   - 5.1. SI `aleat` DISTINTO DE `0`, ENTONCES `letra *= aleat;`
   - 5.2. `rot = (UINT2)letra % 13;`
   - 5.3. `aleat = (aleat << rot | aleat >> (16 – rot))`
6. Guardar el valor introducido en un vector de valores aleatorios.
7. SI `letra` ES IGUAL A ESCAPE IR A FIN.

FIN

Algoritmo 1: Pasos del generador de aleatorios por entrada de teclado.

Para la función mezcla recomendada en [Math96] y antes mencionada, hemos utilizado el operador a nivel de bit XOR y la aplicación, también a nivel de bit, rotación, definida mediante desplazamientos a izquierda y derecha y el operador OR a nivel de bit.

Los pasos que sigue el generador definido quedan recogidos en el Algoritmo 1. Por cada pulsación de tecla quedan generados 16 bits de secuencia.

El Generador definido es de muy sencilla implementación, y los valores generados dependen de dos ocurrencias aleatorias como son el valor de las sucesivas teclas pulsadas por el usuario y el tiempo —medido mediante la función `times()` de la librería `sys/times.h`— transcurrido entre pulsación y pulsación. En cada pulsación de tecla obtenemos 32 bits de información procedente del valor devuelto por la función `times(NULL)` y 8 bits que dependen del valor del carácter introducido por teclado y que obtenemos mediante una función que presentamos más adelante y que hemos llamado `caracter()`.

El valor 31 que empleamos como divisor para obtener el módulo que nos indicará el número de desplazamientos de la rotación (paso 3.3.) está elegido como el primo mayor menor que el número de bits de un **`unsigned long`**. El valor 13 que empleamos como divisor en el paso 5.2. está elegido como el primo mayor menor que el número de bits de un **`unsigned short`**.

En el anexo I, al final de la Tesis, presentamos una posible implementación, en C, del algoritmo 1.

Mostraremos, como ejemplo de uso del generador, los valores que se obtienen al introducir por teclado una secuencia cualquiera de caracteres, por ejemplo, las primeras palabras del título de este mismo apartado: "*Ejemplos de valores obtenidos*". Estos valores obtenidos quedan recogidos en la Tabla 1.

Mostramos también otra tabla de valores, con la suposición de que en la ejecución siempre se introduce el mismo carácter: por ejemplo, una secuencia constante de letras '*A*'. En ese caso, los valores obtenidos son los recogidos en la Tabla 2.

Mostramos en tercer lugar otra tabla (Tabla 3), con la suposición esta vez de que en la ejecución siempre se introducen los diferentes caracteres con la misma cadencia de tiempos: por ejemplo un incremento de 6 en el valor que devolvería la función `times(NULL)` entre pulso de tecla y pulso de tecla.

Y mostramos en cuarto lugar los valores que se obtendrían si siempre se introdujese el mismo carácter (por ejemplo, una vez más, el carácter '*A*', y siempre con la misma cadencia (una vez más un incremento de seis por cada pulsación). En este caso la secuencia generada no será necesariamente siempre la misma, pues la salida depende también del valor inicial de la variable `aleat` y del valor de `t1` en el momento de empezar la ejecución del proceso. Esos valores quedan recogidos en la Tabla 4.

Más adelante presentaremos un estudio estadístico de la calidad de la propiedad de aleatoriedad para cada una de estas cuatro formas de obtener las secuencias.

5.1.2. Primer estudio de las características del generador descrito.

No se dispone de ninguna prueba matemática que determine de forma categórica la propiedad de aleatoriedad de una secuencia de bits dada. El estudio y análisis de la aleatoriedad de una

secuencia cualquiera se lleva a cabo mediante diferentes tests estadísticos. Cada test determina si la secuencia posee, o no, alguna propiedad que permita considerar a la secuencia de bits como aparentemente aleatoria. Las conclusiones de los diferentes tests nunca son definitivas, pero sí probabilísticas. Si la secuencia estudiada supera todos los tests estadísticos a los que ha sido sometida, entonces diremos que la secuencia de bits es aceptada como aleatoria: aceptada porque no ha podido ser rechazada.

aleat inicial					F54C
t1 inicial					69.899.287
Pasos del algoritmo...					
	1.2.	2.1.	3.3.	5.2.	5.3.
	t2	dift	rot	rot	aleat
E	69.899.426	139	13	7	68D7
j	69.899.459	33	3	12	6A94
e	69.899.472	13	25	3	AA0C
m	69.899.484	12	5	7	7229
p	69.899.507	23	15	10	8BBB
l	69.899.518	11	9	12	D8CE
o	69.899.531	13	13	12	06BC
s	69.899.549	18	18	8	AB4D
	69.899.565	16	28	4	4B65
d	69.899.575	10	9	12	DC9E
e	69.899.591	16	30	10	808D
	69.899.601	10	2	11	2AC3
v	69.899.619	18	25	8	7A0E
a	69.899.641	22	24	2	2C91
l	69.899.653	12	20	1	A8FD
o	69.899.670	17	5	9	C4AE
r	69.899.685	15	17	4	8600
e	69.899.702	17	25	10	4CD0
s	69.899.720	18	16	1	E325
	69.899.732	12	15	11	2D0D
o	69.899.864	132	2	6	A778
b	69.899.905	41	24	0	58BA
t	69.899.937	32	22	3	3C7D
e	69.899.958	21	19	9	E767
n	69.899.979	21	17	5	5A0E
i	69.899.988	9	27	0	A800
d	69.900.010	22	23	4	05B6
o	69.900.021	11	16	1	19BD
s	69.900.033	12	30	4	642E

Tabla 1: Entrada por teclado aleatoria en cadencia de pulsaciones aleatoria.

aleat inicial					F54C
t1 inicial					69.914.111
Pasos del algoritmo...					
	1.2.	2.1.	3.3.	5.2.	5.3.
	t2	dift	rot	rot	aleat
A	69.914.218	107	24	9	FA15
A	69.914.246	28	23	6	4801
A	69.914.256	10	9	8	09BE
A	69.914.267	11	20	4	D279
A	69.914.288	21	21	0	2DBD
A	69.914.294	6	17	6	3B4E
A	69.914.309	15	29	11	9E25
A	69.914.327	18	17	0	00F0
A	69.914.408	81	26	8	08FF
A	69.914.427	19	25	12	0C44
A	69.914.443	16	10	11	988E
A	69.914.454	11	18	6	21E1
A	69.914.466	12	26	3	F08E
A	69.914.489	23	1	1	062D
A	69.914.498	9	0	1	F052
A	69.914.513	15	20	0	E4EF
A	69.914.529	16	12	12	3F77
A	69.914.547	18	11	6	2A76
A	69.914.565	18	1	6	2E8F
A	69.914.574	9	6	6	6A0D
A	69.914.597	23	17	10	C7A7
A	69.914.618	21	6	12	73EA
A	69.914.641	23	14	6	552A
A	69.914.660	19	25	6	4837
A	69.914.667	7	12	1	7C53
A	69.914.675	8	11	1	7B1C
A	69.914.688	13	1	2	CC71
A	69.914.701	13	2	6	6EE0
A	69.914.713	12	8	8	5EAA

Tabla 2: Entrada por teclado fija en cadencia de pulsaciones aleatoria.

Guiados por los tests que sugieren los autores de [Mene97], vamos a presentar los resultados obtenidos en el análisis de diferentes secuencias producidas por el generador que presentamos.

aleat inicial F54C
t1 inicial 69.927.566

	1.2. t2	2.1. dift	3.3. rot	5.2. rot	5.3. aleat
E	69.927.572	6	12	2	BA98
j	69.927.578	6	16	3	AAD0
e	69.927.584	6	28	12	E552
m	69.927.590	6	22	11	EBA8
p	69.927.596	6	10	0	9860
l	69.927.602	6	19	8	9A16
o	69.927.608	6	8	6	5C93
s	69.927.614	6	13	8	4F82
	69.927.620	6	28	12	22E9
d	69.927.626	6	9	11	5E98
e	69.927.632	6	25	6	A837
	69.927.638	6	20	6	8254
v	69.927.644	6	9	4	A427
a	69.927.650	6	28	7	33AE
l	69.927.656	6	27	11	4BDD
o	69.927.662	6	13	8	3914
r	69.927.668	6	24	12	AC68
e	69.927.674	6	22	12	8532
s	69.927.680	6	21	5	A450
	69.927.686	6	16	11	EC82
o	69.927.692	6	15	5	1404
b	69.927.698	6	14	2	D145
t	69.927.704	6	25	5	D405
e	69.927.710	6	11	10	CD5E
n	69.927.716	6	21	11	7A61
i	69.927.722	6	2	3	D6C7
d	69.927.728	6	7	0	8295
o	69.927.734	6	24	5	AA2F
s	69.927.740	6	29	8	D055

Tabla 3: Entrada por teclado aleatoria en cadencia de pulsaciones fija.

aleat inicial F54C
t1 inicial 69.935.067

	1.2. t2	2.1. dift	3.3. rot	5.2. rot	5.3. aleat
A	69.935.073	6	12	6	6A3B
A	69.935.079	6	9	12	AA3D
A	69.935.085	6	6	4	F38C
A	69.935.091	6	4	6	E187
A	69.935.097	6	3	5	302D
A	69.935.103	6	6	5	E5B2
A	69.935.109	6	6	11	92C5
A	69.935.115	6	5	5	98C6
A	69.935.121	6	2	1	111C
A	69.935.127	6	14	1	3FC6
A	69.935.133	6	7	12	0B54
A	69.935.139	6	23	0	F49B
A	69.935.145	6	26	3	86EA
A	69.935.151	6	20	10	5E49
A	69.935.157	6	2	8	F78C
A	69.935.163	6	9	2	1FA8
A	69.935.169	6	3	9	71AB
A	69.935.175	6	12	1	9E2C
A	69.935.181	6	30	0	C289
A	69.935.187	6	13	2	F4FD
A	69.935.193	6	20	12	2E07
A	69.935.199	6	12	8	F8E0
A	69.935.205	6	4	7	4EC7
A	69.935.211	6	23	7	239E
A	69.935.217	6	20	10	FCD2
A	69.935.223	6	1	10	E81D
A	69.935.229	6	1	3	B4CA
A	69.935.235	6	26	0	7688
A	69.935.241	6	6	10	B752

Tabla 4: Entrada por teclado fija en cadencia de pulsaciones fija.

5.1.2.1. Postulados de aleatoriedad de GOLOMB.

Como señalan los autores de [Mene97], estos postulados tienen más bien un interés histórico: fueron los primeros definidos y usados para establecer algunas condiciones necesarias para que

una secuencia de bits pudiera ser considerada aleatoria. Actualmente estas condiciones no se consideran suficientes para poder tomar como aleatoria la secuencia estudiada.

Los **postulados de aleatoriedad de GOLOMB** son los siguientes:

1. En la secuencia s_n, el número de unos difiere del número de ceros a lo máximo en una unidad.
2. En la secuencia s_n, al menos la mitad de las cadenas de dígitos iguales (precedidas y seguidas por el otro dígito) tiene longitud 1, al menos una cuarta parte de esas cadenas tienen longitud 2, al menos una octava parte tienen longitud 3, etc. Y además para cada una de las longitudes se dispone de tantas cadenas de unos como cadenas de ceros. (El postulado 2 implica el postulado 1).
3. La función de correlación $C(T)$ tiene dos valores. Esto es, para algún valor K

$$N \cdot C(t) = \sum_{i=0}^{N-1} (2s_i - 1) \cdot (2s_{i+t} - 1) = \begin{cases} N, \text{si } t = 0 \\ K, \text{si } 1 \le t \le N-1 \end{cases}$$

Las tres propiedades quedan recogidas y superadas en nuevos tests que presentan los autores de [Mene97] . No presentamos por tanto el análisis de la secuencia según estos tres postulados, que quedarán superados con los análisis que sí presentamos.

5.1.2.2. Estudio de la complejidad lineal del generador.

Para el estudio del perfil de la complejidad lineal hemos necesitado introducir algunas breves nociones básicas sobre los generadores LFSR.

Un **registro de desplazamiento realimentado linealmente** (LFSR) de longitud L consiste en L estados (o elementos de retraso) numerados $0,1,...,L-1$, cada uno de ellos capaz de almacenar un bit, y que tienen una entrada y una salida; y un reloj que controle los movimientos de datos.

Durante cada unidad de tiempo se desarrollan las siguientes operaciones:

1. El contenido del estado 0 es sacado fuera y forma parte de la secuencia de salida de nuestro LFSR.
2. El contenido del estado i es desplazado al estado $i-1$, para cada valor de i tal que $1 \le i \le L-1$.
3. El nuevo contenido del estado $L-1$ se obtiene por suma módulo 2 de los contenidos previos de un subconjunto fijo del conjunto de estados $0,1,...,L-1$.

Un LFSR se denota por el par $\langle L, C(D) \rangle$, donde

$$C(D) = 1 + c_1 \cdot D + c_2 \cdot D^2 + ... + c_L \cdot D^L \in \mathrm{Z}_2[D]$$

es el llamado **polinomio de conexión** o **polinomio característico**.

Si el contenido inicial del estado i es $s_i \in \{0,1\}$ para cada i tal que $0 \le i \le L-1$, entonces $[s_{L-1},...s_1, s_0]$ se llama estado inicial del LFSR. SI el **estado inicial del LFSR** es $[s_{L-1},...s_1, s_0]$, entonces la secuencia de salida $s = s_0, s_1, s_2,...$ viene únicamente determinado por la siguiente expresión recursiva:

$$s_j = (c_1 \cdot s_{j-1} + c_2 \cdot s_{j-2} + ... + c_L \cdot s_{j-L}) \bmod 2 \text{, para } j \ge L$$

Si todos los estados iniciales son iguales a cero, entonces la secuencia de salida es la secuencia cero.

Cualquier secuencia finita puede ser generada por un LFSR. Se denomina **complejidad lineal de una secuencia finita dada** a la longitud del mínimo LFSR que es capaz de generarla. Una secuencia binaria empleada en aplicaciones criptográficas debería tener una gran complejidad lineal para no ser vulnerable a un ataque mediante un sistema de ecuaciones lineales.

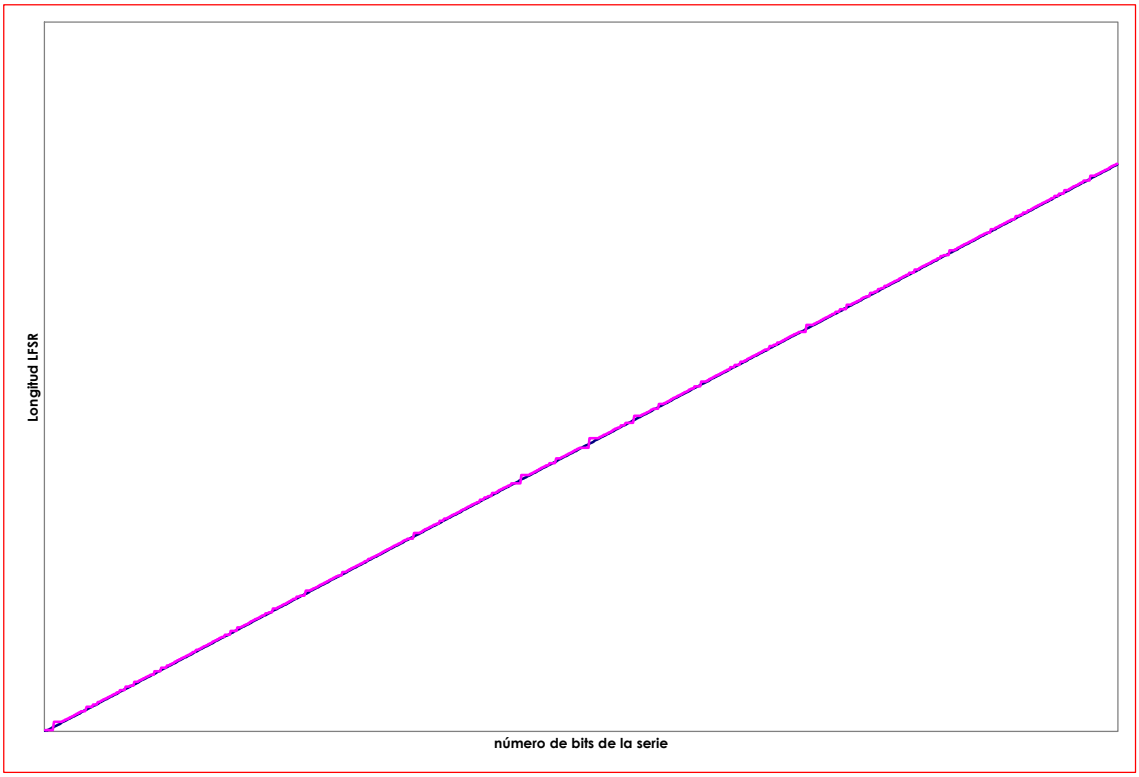
Si $s^N = s_0, s_1,..., s_{N-1}$ es una secuencia binaria finita, la secuencia $L_1, L_2,..., L_N$ (donde L_i es la longitud del mínimo LFSR capaz de generar la subsecuencia de s^N formada por los i primeros valores) se llama **perfil de la complejidad lineal** de s^N. Podemos representar gráficamente los diferentes valores de las sucesivas complejidades lineales, fijando los puntos (N, L_N) en el plano $N \times L$.

La gráfica del perfil de la complejidad lineal de una secuencia verdaderamente aleatoria tiene la forma de una escalera creciente que sigue próxima y de forma irregular a la recta $L = N/2$ (siendo N el número de bits de cada subsecuencia) [Mene97].

5.1.2.3. Algoritmo de BERLEKAMP–MASSEY para determinar el perfil de la complejidad lineal del generador.

Para determinar la complejidad lineal de una secuencia disponemos del algoritmo de BERLEKAMP–MASSEY. Este algoritmo encuentra uno de los LFSR más cortos capaz de generar la secuencia dada y, por tanto, determina así la complejidad lineal de la secuencia. En la práctica el algoritmo determina la longitud, el polinomio característico y el contenido inicial del LFSR que puede generar la secuencia considerada. Un LFSR decimos que genera la secuencia finita $s = s_0, s_1,..., s_{n-1}$ si existe algún estado inicial para el que la salida del generador LFSR es exactamente la secuencia s.

Dados n bits de una secuencia, el algoritmo de BERLEKAMP–MASSEY determina las características del mínimo LFSR capaz de generarlos. Al tomar el siguiente bit $n+1$, chequea si ese nuevo elemento puede ser generado por el registro caracterizado anteriormente. En caso afirmativo, se mantiene la longitud del registro y, por tanto, la complejidad de la secuencia analizada hasta el momento no aumenta. En caso de que el actual registro no pueda ser el generado por el actual LFSR habrá que buscar otro LFSR de longitud mayor que sea capaz de generar los $n+1$ bits; en ese caso se incrementa también la complejidad lineal de la secuencia analizada.



Gráfica 1: Perfil de complejidad lineal de nuestro generador.

Para la implementación de ese algoritmo hemos tomado como base el diagrama de flujo presentado en [Mass94] y que recogemos en el anexo II, al final de la Tesis. También viene recogida la descripción del algoritmo en [Mene97] (algoritmo 6.30). Nuestra implementación ha sido probada con secuencias de las que ya conocíamos el perfil de su complejidad lineal: tanto de algunas secuencias cuyo perfil era semejante al esperado para una secuencia aleatoria como para perfiles que delatan unas deficiencias en el generador, al discurrir la gráfica del perfil muy por debajo de la recta $L = N/2$.

Hemos hecho un estudio del perfil de la complejidad lineal de las secuencias creadas por el generador presentado en este Capítulo. Hacemos el análisis de cuatro secuencias generadas según los cuatro usos ya antes contemplados: el uso ordinario de nuestro generador en el que introducimos una secuencia de caracteres, como por ejemplo "*Algoritmo de BERLEKAMP–MASSEY para determinar el perfil de la complejidad lineal del generador*" (puede verse el perfil en la Gráfica 1: hemos dejado representada en rojo el perfil de la complejidad lineal de la secuencia estudiada); el segundo caso en que siempre se pulsa la misma tecla y por tanto se introduce siempre el mismo carácter, que volveremos a tomar 'A'; el tercer caso en que se introduce la misma secuencia que el caso 1 pero siempre en la misma cadencia, que volvemos a tomar igual a 6; y el cuarto caso en que se introduce siempre y con la misma cadencia el mismo carácter 'A'. En todos los casos la gráfica del perfil de la complejidad lineal presenta una forma muy semejante: La recta de tendencia presentada en la Gráfica 1 tiene la ecuación `y = 0,5 * x + c`, donde `c`

= 0,2478 para el caso primero. Las otras tres rectas de tendencia para los otros tres casos presentan la misma forma, variando en algún caso ligeramente el valor del coeficiente independiente: c = 0,2484 en el caso de que se pulse siempre la misma tecla ('A') con cadencia aleatoria; c = 0,2483 en el caso de que se pulse una frase aleatoria como la indicada arriba, pero con una cadencia constante de un pulso de tecla cada incremento en 6 del valor que devuelve la función times(NULL); c = 0,2478 en el último caso, en que se pulsa siempre la misma tecla ('A') con la misma cadencia de 6.

La recta de tendencia ha quedado representada en color azul. Como cabía esperar, si el generador tiene un comportamiento aleatorio, la recta tendencia tiene, como hemos dicho, la ecuación de la recta $L = N/2$.

Antes ha quedado dicho que la gráfica del perfil de la complejidad lineal de una secuencia verdaderamente aleatoria tiene la forma de una escalera creciente que sigue próxima y de forma irregular a la recta $L = N/2$ (siendo N el número de bits de cada subsecuencia). Esta afirmación recoge una condición necesaria para poder afirmar que una secuencia es realmente aleatoria. Pero no es suficiente.

Para un estudio y análisis más a fondo de la condición de aleatoriedad de las secuencias obtenidas con nuestro generador acudimos a los cinco tests que recogen y recomiendan los autores de [Mene97].

5.1.3. Tests estadísticos. Tests de contraste de hipótesis.

Ya ha quedado dicho antes que no se dispone de prueba matemática alguna que asegure que un generador es verdaderamente aleatorio. Los tests de contraste de hipótesis pretenden precisamente lo contrario: encontrar debilidades en nuestro generador que nos lleve a rechazarlo y a no considerarlo aleatorio.

Si en la secuencia estudiada falla alguno de los tests estadísticos, el generador deberá ser rechazado y considerado no aleatorio. Si la secuencia supera todos los tests, entonces decimos que no podemos rechazar su condición de aleatoriedad y podemos considerarlo válido: en ningún caso afirmaremos rotundamente la condición de aleatoriedad.

Todos los tests estadísticos que se recogen en [Mene97] calculan valores estadísticos que tienen la distribución Normal o la distribución Chi cuadrado.

Un **test de contraste de hipótesis** es un procedimiento basado sobre las observaciones de variables aleatorias que nos lleva a rechazar o a no rechazar la hipótesis inicial. A la hipótesis que consideramos cierta la llamamos **hipótesis nula** (H_0). Es la hipótesis que tratamos de contrastar, de forma que al final del proceso quede o no quede rechazada. En nuestro caso la afirmación de H_0 es: "la secuencia binaria producida por nuestro generador es aleatoria".

A la hipótesis complementaria la llamamos **hipótesis alternativa** (H_A). El rechazo de H_0 lleva consigo la aceptación de H_A. En nuestro caso H_A afirmaría que "la secuencia binaria producida por nuestro generador NO es aleatoria".

Una vez calculado el valor estadístico se ha de establecer el **nivel de significación** (probabilidad de que quede rechazada la hipótesis nula H_0 cuando en realidad resultaba ser cierta). Al nivel de significación lo denotamos con la letra α.

El nivel de significación representa el rango de error que nos podemos permitir. Si H_0 resulta rechazado cuando en realidad era la hipótesis cierta incurrimos en un error llamado **error de tipo I**. Cuanto menor es el valor de α, menos probabilidades se tiene de poder rechazar la hipótesis H_0. Si nuestra hipótesis H_0 no ha sido rechazada para un $\alpha = \alpha_0$, no lo será tampoco para ningún $\alpha < \alpha_0$. Cuanto mayor es el valor de α, mayor es el peligro de desechar la hipótesis H_0 cuando realmente era cierta. Pero también es mayor la probabilidad de que H_0 sea realmente cierta si no ha sido finalmente rechazada.

Puede ocurrir que no rechacemos H_0 cuando en realidad la hipótesis resulte falsa. A este error lo llamamos **error de tipo II** . Llamamos β al complementario a 1 del nivel de significación: $\beta = 1 - \alpha$. Cuanto menor sea el riesgo de un error tipo I ocasionado por la elección de un valor de α demasiado bajo, mayor es el riesgo de un error de tipo II ocasionado por una elección de un valor demasiado elevado para β. Para un determinado tamaño muestral no podemos reducir simultáneamente los dos errores, por lo que siempre deberemos sacrificar uno de los dos si queremos disminuir el otro. Si α está prefijado, el modo de reducir β es aumentar el tamaño de la muestra.

Para cada nivel de significación α debemos estudiar la región crítica asociada. La **región crítica** recoge aquellos valores del test estadístico que si ocurren nos llevan a rechazar nuestra hipótesis nula H_0. Dependiendo de la distribución de nuestra variable estadística y dependiendo también del valor de α, tendremos uno u otro valor de corte: ese valor de corte acota la región crítica y la **región de confianza** (la que no es región de confianza).

	H_0 VERDADERA	H_0 FALSA
H_0 NO rechazado	Decisión correcta	ERROR Tipo II
H_0 SÍ rechazado	Error Tipo I	Decisión correcta

Tabla 5: Opciones de validación o rechazo de las hipótesis.

Dado un α y un grado de libertad n, el valor x_α correspondiente señala el valor de la abcisa para el cual la gráfica de la distribución chi cuadrado correspondiente al grado de libertad n ha cubierto el $(1-\alpha)\times 100$ por ciento de la superficie. Cuanto menor es α más difícil será que

rechacemos la hipótesis H_0.

Si rechazamos H_0 se dirá: "Existe evidencia suficiente al nivel de significación α para indicar que se verifica H_A". Si NO rechazamos H_0 se dirá: "NO existe evidencia suficiente al nivel de significación α para indicar que se verifica H_A".

Cada investigador debe determinar el porcentaje o valor de α utilizado para discriminar. Cuanto mayor sea el valor de α más exigente es el proceso de validación de la hipótesis H_0. Es importante elegir adecuadamente el nivel de significación. En [Mene97] se recomienda, para los tests estadísticos de un generador de aleatorios, valores de α comprendidos entre $0{,}001$ y $0{,}05$.

Si el valor estadístico cae dentro de la región de confianza podemos decir que los resultados no son significativamente estadísticos como para desestimar H_0. Si el valor estadístico cae dentro del nivel de riesgo, entonces aceptar como válido H_0 supone un riesgo alto de equivocarse.

En nuestro estudio estadístico del generador de aleatorios, el valor esperado para todos los cinco estadísticos calculados es cero. Cuando no lo es, la cuestión es decidir si es debido a que la hipótesis H_0 es falsa, o se debe a desviaciones de la muestra tomada. El valor de α indica cuánto margen otorgamos al resultado de la muestra para decidir si la diferencia con respecto al valor esperado es debido a una u otra causa.

5.1.3.1. Descripción de los cinco tests estadísticos recomendados en [Mene97].

Sea $s = s_0, s_1, s_2, \ldots s_{n-1}$ una secuencia binaria de longitud n. Los cinco tests que se presentan a continuación son usados habitualmente para determinar si la secuencia binaria s posee características que permitan considerarla como verdaderamente aleatoria.

1. **Test de Frecuencia** (Test monobit). Determina si el número de ceros y el número de unos en la secuencia s son aproximadamente los mismos, tal y como cabe esperar de una secuencia generada por un generador aleatorio. Si llamamos n_0 y n_1 al número de ceros y de unos de la secuencia, el valor estadístico que estudiamos es:

$$X_1 = \frac{(n_0 - n_1)^2}{n}$$

   y evidentemente $n = n_0 + n_1$, y sigue aproximadamente una distribución Chi cuadrado con un grado de libertad ($n = 1$).

   Que el valor esperado para la variable X_1 sea cero es equivalente a la exigencia recogida en el primero postulado de GOLOMB.

2. **Test de Series** (Test de los dos bits). Busca y cuenta el número de ocurrencias de las cuatro subsecuencias: 00, 01, 10 y 11 y estudia si su distribución es tal y como esperaríamos de una secuencia generada por un generador verdaderamente aleatorio.

   Ya tenemos definidos los parámetros n_0 y n_1. Definimos también ahora los siguientes cuatro

parámetros: n_{00}, n_{01}, n_{10} y n_{11}, que recogen el número de ocurrencias de cada una de las cuatro subsecuencias.

El valor estadístico que estudiamos es:

$$X_2 = \frac{4}{n-1} \cdot (n_{00}^2 + n_{01}^2 + n_{10}^2 + n_{11}^2) - \frac{2}{n} \cdot (n_0^2 + n_1^2) + 1$$

que seguirá aproximadamente una distribución Chi cuadrado con dos grados de libertad $(n = 2)$.

Se verifica y es fácilmente comprobable que

$$n_{00} + n_{01} + n_{10} + n_{11} = n - 1.$$

3. **Test Poker**. Estudia todas las subsecuencias diferentes de un tamaño que determina previamente.

   Primero busca el mayor entero positivo *m* que verifique que

   $$\left\lfloor \frac{n}{m} \right\rfloor \geq 5 \cdot (2^m)$$

   Tomamos entonces el valor

   $$k = \left\lfloor \frac{n}{m} \right\rfloor$$

   Dividimos la secuencia s en k partes no solapadas, cada una de ellas de longitud *m*. Definimos 2^m parámetros n_i $(1 \leq i \leq 2^m)$ para almacenar el número de ocurrencias de cada una de las secuencias de la longitud indicada.

   El test Poker determina si cada una de las posibles secuencias de longitud *m* aparecen un número semejante de veces.

   El valor estadístico que estudiamos es:

   $$X_3 = \frac{2^m}{k} \cdot (\sum_{i=1}^{2^m} n_i^2) - k$$

   que seguirá aproximadamente una distribución Chi cuadrado con $2^m - 1$ grados de libertad $(n = 2^m - 1)$.

4. **Test de Rachas**. El test toma una secuencia binaria s y calcula el número de rachas (huecos y bloques) de diferentes longitudes.

   Dada una secuencia binaria *s*, llamamos **racha** a una subsecuencia de s formada por una determinada cantidad de ceros o de unos, y que no vienen ni precedidas ni seguidas por el mismo dígito. Si la racha está formada por ceros lo llamamos **hueco**. Si la racha está formada por unos lo llamamos **bloque**.

Para determinar la longitud de las diferentes rachas a estudiar, primero calculamos los sucesivos parámetros e_i definidos según la siguiente expresión:

$$e_i = \frac{n-i-3}{2^{i+2}} \text{ para } 1 \le i \le k$$

donde k es el mayor valor de i que verifica que $e_i \ge 5$.

Al número de huecos de longitud i lo almacenamos en el parámetro G_i. El número de bloques en el parámetro B_i.

El valor estadístico que estudiamos es:

$$X_4 = \sum_{i=1}^{k} \frac{(B_i - e_i)^2}{e_i} + \sum_{i=1}^{k} \frac{(G_i - e_i)^2}{e_i}$$

que seguirá aproximadamente una distribución Chi cuadrado con $2 \cdot k - 2$ grados de libertad ($n = 2 \cdot k - 2$).

Que los valores esperados para las variables X_2, X_3 y X_4 sean cero es una exigencia superior a la presentada en el postulado segundo de GOLOMB.

5. **Test de Autocorrelación**. El propósito de este test es verificar si tenemos correlación entre la secuencia s y la secuencia s' obtenida de desplazar sin rotación la secuencia s un número determinado de posiciones (d) a derecha o a izquierda.

   El número de bits de s que no coinciden con la correspondiente secuencia desplazada d posiciones $A(d)$ se calcula fácilmente

$$A(d) = \sum_{i=0}^{n-d-1} s_i \oplus s_{i+d} \qquad 1 \le d \le \lfloor n/2 \rfloor$$

   donde $\oplus$ representa la operación a nivel de bit or exclusivo (XOR).

   El valor estadístico que estudiamos es:

$$X_5 = \frac{2 \cdot (A(d) - \frac{n-d}{2})}{\sqrt{n-d}}$$

   que seguirá aproximadamente una distribución normal $N(0,1)$. Para valores pequeños de $A(d)$ está recomendado calcular su valor para diferentes desplazamientos.

   Y, una vez más, tenemos que el hecho de que el valor esperado para X_5 sea cero comprende la exigencia del postulado tercero de GOLOMB.

5.1.3.2. Resultados de la aplicación de los tests y conclusiones.

Estos cinco tests presentados han sido implementados para estudiar las secuencias obtenidas con nuestro generador. La implementación para cada uno de los cinco tests estadísticos ha sido probada con diferentes ejemplos de secuencias. Por ejemplo la facilitada en el libro de Menezes

[Mene97] en su ejemplo 5.31. También hemos probado con otras muchas secuencias de las que sabíamos de antemano los valores estadísticos que deberían resultar para cada uno de los tests implementados, o secuencias de las que sabíamos que uno o varios de los tests debería ofrecer un resultado muy alejado de la distribución esperada. En todos los casos hemos obtenido los resultados esperados. Aceptamos como correcto nuestro código.

Un primer análisis del generador, realizado con estos cinco tests, lo hemos hecho sobre secuencias generadas de forma idéntica a como hemos trabajado antes para la búsqueda del perfil de la complejidad lineal. Como se recordará, utilizamos el generador en cuatro formas diferentes, cada vez con más limitaciones en su capacidad de "aleatoriedad": pulsando teclas de forma aleatoria, en una cadencia también aleatoria; manteniendo siempre la pulsación sobre la misma tecla, con cadencia aleatoria; pulsando teclas de forma aleatoria, pero manteniendo constante la cadencia; y manteniendo constante la tecla pulsada y la cadencia de pulsación. La cadena de caracteres introducida ha sido la misma que la empleada para la determinación del perfil de la complejidad lineal con el algoritmo de BERLEKAMP–MASSEY.

	X1	X2	X3	X4	X5
mediciones 1	0,3021	1,5849	28,5000	8,6665	0,2272
mediciones 2	0,2042	0,7209	23,8333	9,3400	0,4220
mediciones 3	0,2042	1,0084	23,1667	8,3557	0,3571
mediciones 4	1,3500	2,3820	33,8333	12,0554	0,1623
x_a	3,8410	3,8410	44,9850	15,5070	1,2860

Tabla 6: Valores obtenidos mediante los cinco tests estadísticos propuestos por Menezes [Mene97].
Mediciones 1: Secuencia aleatoria en pulsos aleatorios.
Mediciones 2: Secuencia constante ('A') en pulsos aleatorios.
Mediciones 3: Secuencia aleatoria en pulsos constantes (6)
Mediciones 4: Secuencia constante ('A') en pulsos constantes (6).
x_a : valores de x_a para $a = 0{,}05$

Presentamos en la Tabla 6 los resultados obtenidos para los cinco tests, en los cuatro casos citados. Para las secuencias generadas mediante pulsaciones de teclado de cadencia de pulsación aleatoria, los valores que presentamos son las medianas de los obtenidos en 20 secuencias. Para los otros dos casos, en que la cadencia es constante, y la entrada de teclado o es una frase establecida o es una secuencia de caracteres siempre iguales hemos preferido mostrar un caso concreto y no calcular medianas de diferentes casos de teclas y de cadencias (hemos tomado la cadencia igual a 6 y para el cuarto caso la tecla pulsada 'A' mayúscula). En las cuatro mediciones, los grados de libertad para el valor `X3` es 8, y para el valor de `X4` es 31. Como se puede observar, el generador mantiene sus valores por debajo de los límites de los tests estadísticos aún en el caso más desfavorable en que la secuencia de entrada se realiza mediante

una entrada constante de teclado en una cadencia constante.

Hemos hecho otro análisis, con los mismos tests, centrando el estudio en un empleo ordinario del generador: secuencias obtenidas por pulsaciones aleatorias de teclado en una cadencia irregular. Hemos probado con diferentes secuencias de longitudes de bits. En la Tabla 7 recogemos los valores estadísticos obtenidos con cada uno de los cinco tests. Para el test de frecuencia y para el test de series, los grados de libertad obtenidos han sido siempre 1. Los tests de poker y de rachas han tenido diferentes grados de libertad, que han ido creciendo a medida que aumentaba la longitud de la secuencia analizada. En la Tabla 7 se recogen esos valores de los grados de libertad para estos dos tests. Los valores que se recogen para x_a en estos tests son los correspondientes a sus respectivos grados de libertad. Los valores en cada columna "Xi" son las medianas de los valores X_i sobre un muestreo de 40 secuencias en cada uno de los tamaños. Las columnas "l3" y "l4" recogen esos grados de libertad de "X3" y "X4" respectivamente. Hemos tomado $a = 0{,}05$. Todos los valores superan las exigencias de los tests estadísticos, incluso si hubiéramos tomado valores de x_a para un $a = 0{,}1$.

	X1	X2	X3	l3	X4	l4	X5
96 bits	0,521	1,577	1,750	3	3,875	2	0,3254
x_a	3,841	3,841	7,815		5,991		1,6449
160 bits	0,400	1,811	6,925	7	4,894	4	0,0000
x_a	3,841	3,841	14,067		9,488		1,6449
320 bits	0,256	1,518	11,400	15	5,025	4	0,3413
x_a	3,841	3,841	24,996		9,488		1,6449
480 bits	0,613	1,517	12,200	15	5,011	4	0,2276
x_a	3,841	3,841	24,996		9,488		1,6449
640 bits	0,531	1,409	10,400	15	4,230	4	0,2844
x_a	3,841	3,841	24,996		9,488		1,6449

Tabla 7: Valores de los estadísticos calculados para nuestro generador de bits. Los valores de x_a son los que corresponden para $a = 0{,}05$. Se recogen 5 referencias para diferentes longitudes de secuencias estudiadas.

5.1.4. Test estadístico universal de MAURER.

5.1.4.1. Presentación teórica del test de MAURER.

El **test estadístico universal de MAURER** ofrece la posibilidad de detectar una gran variedad de deficiencias de carácter estadístico en un generador de aleatorios. Entre los defectos que detecta el test de MAURER se cuentan también los que logran encontrar los cinco tests estadísticos presentados en [Mene97] y desarrollados en el epígrafe anterior del presente Capítulo. La desventaja principal del test de MAURER está en la cantidad enorme de bits (más de mil millones) que se requieren para que el test pueda realizarse sobre el generador estudiado de forma efectiva. Salvada esa dificultad, si nuestro generador es capaz de ofrecer una cantidad muy grande de bits (ya veremos cuántos) entonces resulta un test muy eficiente.

El test estadístico de MAURER tiene como objeto el análisis de las secuencias obtenidas mediante generadores pseudoaleatorios. Sin embargo hemos querido utilizarlo para testear nuestro generador de secuencias por entrada de teclado. Más adelante explicamos el modo en qué hacemos uso de nuestro generador para lograr que se comporte como pseudoaleatorio y poder así obtener una secuencia suficientemente larga.

Para realizar el test, hay que calcular el valor estadístico X_u para la secuencia de salida $s = s_0, s_1, s_2, \ldots s_{n-1}$. Se debe escoger un parámetro L dentro del intervalo $[6,16]$. La secuencia s se particiona entonces en bloques no superpuestos de L bits cada uno; se pueden descartar los bits residuales que sobran después de realizar toda la partición. El número total de bloques a realizar es $Q+K$, donde Q y K se definen más adelante.

Para cada bloque i de longitud L ($1 \le i \le Q+K$) definimos b_i como el valor del entero que queda codificado de forma binaria en ese bloque i–ésimo. Los bloques son leídos por orden. Se define una tabla T, donde cada valor de posición j ($T[j]$) es el valor i de la última ocurrencia de un bloque cuyo valor binario es igual a j, $0 \le j \le 2^L - 1$.

Los primeros Q bloques de s se emplean para inicializar la tabla T; Q acostumbra a tomarse de tal forma que sea al menos $10 \cdot 2^L$, de forma que tengamos una alta probabilidad de que se den, al menos una vez, cada una de las 2^L posibles ocurrencias de los bloques de L bits entre los Q primeros bloques. Los restantes K bloques se emplean para el cálculo del valor estadístico X_u, como sigue.

Para cada i, $Q+1 \le i \le Q+K$, sea $A_i = i - T[b_i]$; A_i es el número de posiciones desde la última ocurrencia del bloque b_i. Entonces

$$X_u = \frac{1}{K} \sum_{i=Q+1}^{Q+K} \log A_i \qquad (1)$$

K debe ser al menos de un valor $1000 \cdot 2^L$ (y por tanto, la muestra de s debe ser de una longitud de bits de al menos $1010 \cdot 2^L \cdot L$).

En la Tabla 8 se muestran las medias m y las varianzas s_1^2 de X_u para una secuencia verdaderamente aleatoria con los parámetros K y Q tendiendo a infinito. A partir de esos valores se puede calcular la desviación estándar de X_u que, según podemos ver en [Mene97] vale:

$$s^2 = c(L,K)^2 \cdot s_1^2 / K \tag{2}$$

O, lo que es lo mismo, (y así viene presentado en [Maur92])

$$s = c(L,K) \cdot \sqrt{s_1^2 / K} \tag{3}$$

donde (según [Mene97])

$$c(L,K) \approx 0.7 - \frac{0.8}{L} + \left(1.6 + \frac{12.8}{L}\right) \cdot \left(\frac{1}{K}\right)^{\frac{4}{L}}, \text{ para } K \geq 2^L \tag{4}$$

Y según [Maur92]

$$c(L,K) \approx 0.7 - \frac{0.8}{L} + \left(4.0 + \frac{32.0}{L}\right) \cdot \left(\frac{1}{K}\right)^{\frac{3}{L}} \cdot \frac{1}{15}, \text{ para } K \geq 2^L \tag{5}$$

L	m	s_1^2
1	0.7326495	0.690
2	1.5374383	1.338
3	2.4016068	1.901
4	3.3112247	2.358
5	4.2534266	2.705
6	5.2177052	2.954
7	6.1962507	3.125
8	7.1836656	3.238
9	8.1764248	3.311
10	9.1723243	3.356
11	10.170032	3.384
12	11.168765	3.401
13	12.168070	3.410
14	13.167693	3.416
15	14.167488	3.419
16	15.167379	3.421

Tabla 8: Valores de la media y la varianza, precalculados para una secuencia aleatoria

El test estadístico universal de MAURER emplea el valor calculado de X_u de la siguiente forma: Una vez tenemos X_u, y los valores de las medias m y de las varianzas s^2 (tomados de la Tabla 8)

y con la expresión de la función $c(L,K)$ entonces calculamos la desviación estándar (a partir de las expresiones (2) y (3)). Y a partir de estos datos obtenemos el valor estadístico Z_u :

$$\mathrm{Z}_u = (X_u - m)/s \tag{6}$$

que si la secuencia es aleatoria deberá seguir la distribución normal $N(0,1)$.

Supongamos que el valor estadístico Z_u para una secuencia aleatoria sigue la distribución normal $N(0,1)$. Supongamos que el valor estadístico puede ser el esperado entre dos valores, por arriba y por abajo, que serían los límites para las secuencias no aleatorias. Para el nivel de significación, a , el valor umbral x_a se toma (usando la Tabla 9), de tal manera que

$$P(\mathrm{Z} > x_a) = P(\mathrm{Z} < -x_a) = a/2 .$$

Si el valor del estadístico Z_u para la muestra de salida satisface $\mathrm{Z}_u > x_a$ ó $\mathrm{Z}_u < -x_a$, entonces la secuencia falla el test; en otro caso, pasa el test. Este test se llama por tanto test de dos caras. Por ejemplo, si $a = 0{,}05$, entonces $x_a = 1.96$ (que es el valor x_a correspondiente para $a = 0.05/2 = 0.025$), y se espera que el test sobre la secuencia aleatoria falle sólo en un 5 % de los casos.

Otro modo equivalente de hacer la validación del test es a partir del valor X_u : se calcula la desviación estándar s obtenida a partir de (2) ó (3); se toma la media esperada m tomada de la Tabla 8; y se busca el valor umbral para un nivel de significación a determinado. Obtenemos entonces los umbrales de aceptación o de rechazo k_1 y, k_2 como

$$k_1 = m - x_a \cdot s \qquad k_2 = m + x_a \cdot s \tag{7}$$

La prueba se supera cuando se verifica que

$$k_1 \le X_u \le k_2 \tag{8}$$

Como observamos en las expresiones (2) y (3), el valor de la varianza s decrece en la forma $1/\sqrt{K}$ cuando K aumenta. Así como ocurre también con cualquier otro test estadístico, al aumentar la longitud de la secuencia se reduce la desviación estándar y, por tanto, se permite detectar menores desviaciones de los valores estadísticos estudiados.

Con las definiciones tomadas de [Maur92], tenemos que el intervalo de valores entre los que debe quedar fijado X_u (valores de la expresión (7)) para no desechar a la pretendida secuencia aleatoria, es el expresado en (8). Si calculamos la diferencia entre los dos valores extremos, tenemos que

$$k_2 - k_1 = 2 \cdot x_a \cdot s$$

Los valores recomendados por MAURER [Maur92] para el test universal son:

$$6 \le L \le 16 \qquad Q \ge 10 \cdot 2^L \qquad K \ge 1000 \cdot 2^L$$

Y recomienda tomar los valores de los niveles de significación (ver Tabla 9) entre 0,001 y 0,01.

a	0.1	0.05	0.025	0.01	0.005	0.0025	0.001	0.0005
x_a	1.2816	1.6449	1.9600	2.3263	2.5758	2.8070	3.0902	3.2905

Tabla 9: Valores percentiles de una distribución normal estándar. Si Z es una variable aleatoria que tiene una distribución normal estándar, entonces $P(Z > x) = a$

Y para estos valores recomendamos, al tomar el último extremo $L = 16$, tenemos que $Q = 655.360$ y que $K = 65.536.000$. Y entonces el valor de la desviación estándar es $s = 0{,}000155$, si tomamos el valor de $c(L,K)$ definido por [Maur92] ó $s = 0{,}000152$, si tomamos el valor de $c(L,K)$ definido por [Mene97]. Y tomando un valor de a (por ejemplo $a = 0{,}005$), tenemos un $x_a = 2{,}8070$ (valor correspondiente a $a = 0{,}005/2 = 0{,}0025$). Y con todo eso llegamos a que

$$k_2 - k_1 = 2 \cdot x_a \cdot s = 0{,}000853$$

El valor estadístico calculado después de 65.536.000 sumas y logaritmos, debe caer en un intervalo de valores verdaderamente estrecho.

5.1.4.2. Algoritmo para el test estadístico universal de MAURER

El Algoritmo que desarrolla el test estadístico universal de MAURER es el recogido en el Algoritmo 2.

Entrada	Una secuencia binaria $s = s_0, s_1, s_2, \ldots s_{n-1}$ de longitud n, y los parámetros L, Q, K.
Salida	El valor estadístico X_u para la secuencia s

1. Poner a cero la tabla T. PARA j, $0 \le j \le 2^L - 1$ HACER $T[j] \leftarrow 0$
2. Inicializar la tabla T. PARA i, $0 \le i \le Q$ HACER $T[b_i] \leftarrow i$
3. $sum \leftarrow 0$
4. PARA i, $Q+1 \le i \le Q+K$ HACER
   - 4.1. $sum \leftarrow sum + \log(i - T[b_i])$
   - 4.2. $T[b_i] \leftarrow i$
5. $X_u \leftarrow sum/K$
6. Return(X_u)

Algoritmo 2: Test estadístico universal de MAURER.

5.1.4.3. Algunos valores obtenidos. Interpretación.

Si queremos realizar un estudio de nuestro generador con el test de MAURER, debemos tener en cuenta que, para los valores aconsejados para los parámetros,

$$6 \le L \le 16 \qquad Q \ge 10 \cdot 2^L \qquad K \ge 1000 \cdot 2^L,$$

cuando queramos estudiar el caso $L=16$, aunque tomemos los valores mínimos recomendados para Q y K, necesitaremos que el número de bits de cada secuencia que queramos analizar sea al menos $1010 \cdot 2^L \cdot L$, es decir, 1.059.061.760.

Para proceder a este estudio deberemos diseñar una forma de utilizar el generador que no requiera la intervención de ningún usuario: para generar esta cantidad de bits se requeriría que un usuario realizara 66.191.360 pulsaciones de teclado, lo que supondría un trabajo prolongado durante algo más de 76 días, realizado por una persona que pudiese mantener un ritmo constante de diez pulsaciones por segundo, sin interrupción. Como además es necesario someter a test un número amplio de series, el trabajo para someter nuestro generador al test de MAURER se convierte en imposible.

Hemos diseñado un procedimiento para poder testear el generador con el test universal de MAURER. Como se ha visto en la Tabla 5, ya hemos realizado un primer análisis (con los cinco tests recomendados en [Mene97]) de nuestro generador suponiendo que la entrada de teclado fuese siempre la misma y que la cadencia de pulsaciones fuese también constante.

El procedimiento consiste en emplear nuestro generador como si fuese un PRNG, de forma que las entradas por teclado estén prefijadas y los valores de tiempo transcurrido entre una pulsación y la siguiente también: entrada de teclado constante y cadencia de pulsación constante.

L	k1	X[L]	k2
6	5,206854	5,219121179	5,228556
7	6,188070	6,200019001	6,204432
8	7,177613	7,121962139	7,189719
9	8,172000	8,179261904	8,180850
10	9,169116	9,174770197	9,175533
11	10,167718	10,17270238	10,172346
12	11,167102	11,17094146	11,170428
13	12,166879	12,1708851	12,169261
14	13,166841	13,17018484	13,168545
15	14,166880	14,170022	14,168096
16	15,166945	15,15182785	15,167813

Tabla 10: Valores de X_u calculados para los valores de L desde 6 hasta 16. Y extremos (k_1 y k_2)de los intervalos donde deberían estar estos valores.

Como podemos ver en los sucesivos pasos de nuestro generador, hay dos valores iniciales que no dependen para nada de las entradas del teclado, ni de su cadencia: el valor inicial de la variable `aleat` (variable de tipo **`unsigned short`**, de 16 bits) y el primer valor que toma la variable `t_1` (variable de tipo **`unsigned long`**, de 32 bits). Los dos valores que hemos dicho que dejamos prefijados son el de la variable `dift` (variable tipo **`unsigned long`**, de 32 bits) y el de la variable

`letra` (variable tipo **`char`** de 8 bits). En total son 88 bits que podemos considerar como una semilla inicial de nuestro generador, convertido ahora en generador determinista. El generador así empleado tarda menos de 20 segundos en presentar al test de MAURER la secuencia de más de mil millones de bits.

Hemos realizado 50 veces el test de MAURER sobre 50 secuencias de 1.059.061.760 bits, generadas a partir de 50 "semillas" iniciales. Los valores de las medianas de cada X_u para cada uno de los 11 valores de L estudiados quedan recogidos en la Tabla 10. En esta tabla, además de los valores de las medianas, recogemos los extremos k_1 y k_2 que le corresponden, calculados con $\alpha = 0{,}005$, y el valor de x_a para $\alpha/2$ $x_a = 2{,}8070$, tal y como hemos explicado antes y ha quedado recogido en la expresión (7).

En cuatro de los once casos ($L = 6,7,9,10$) los valores de X_u han caído dentro de intervalo. En dos casos ($L = 8,16$) el valor está por debajo del mínimo en cinco centésimas y en una centésima, respectivamente. En cinco casos ($L = 11,12,13,14,15$) el valor está por encima del máximo en valores de entre una centésima y tres milésimas.

5.1.5. Valoraciones sobre nuestro generador.

El generador ha obtenido un perfil de complejidad lineal muy próximo a la recta $L = N/2$. Ha superado los cinco tests estadísticos de contraste de hipótesis. Consideramos, por tanto, que nuestro algoritmo es válido para generar secuencias que se pueden considerar aleatorias.

En un segundo estudio, hemos analizado nuestro generador en un comportamiento determinista, en el que no ha intervenido ningún factor externo (pulsación aleatoria de teclado, cadencia aleatoria de pulsación), y toda la secuencia generada ha dependido únicamente de un valor inicial bastante reducido: 88 bits. Hemos analizado las secuencias obtenidas mediante este procedimiento con el test universal de MAURER. Los resultados obtenidos mediante ese test son aparentemente buenos. Pero sólo aparentemente, puesto que la semilla inicial aporta muy poca entropía a la serie total (de cada 88 bits iniciales generamos más de mil millones de bits).

Ya hemos dicho que el test de MAURER está diseñado para los generadores pseudoaleatorios. Pero ya que disponíamos de un medio de obtener una secuencia suficientemente larga, hemos querido analizar el comportamiento estadístico de nuestro generador cuando no hay un agente externo que introduzca incertidumbre, aleatoriedad. Y el resultado, como se ha visto, ha sido que el algoritmo logra superar el test de MAURER en varios de los valores de L; y se queda a una distancia de milésimas en los otros valores. No diremos que nuestro generador ha superado las pruebas estadísticas del test de MAURER. Tampoco lo pretendíamos realmente, pues como ya hemos señalado el origen de cada secuencia analizada arranca de una semilla que se expande... ¡ 22.625.410 de veces! Pero hemos querido recoger el estudio por dos razones:

1. Los resultados ofrecidos por el test de MAURER nos han parecido realmente buenos: mejores de los que nosotros hubiéramos esperado inicialmente.

2. La velocidad de generación de una secuencia de 1.059 millones de bits nos ha parecido grande. Especialmente si la comparamos con el generador que presentamos a continuación, el BBS, que requiere unos tiempos grandes (varias horas) para esas longitudes. En una máquina con las características que señalamos en la Tabla 11 cada secuencia de esa cantidad de bits ha sido generada en poco menos de 20 segundos.

vendor_id	AuthenticAMD
model name	AMD-K6(tm) 3D+ Processor
cpu MHz	400.916
cache size	256 KB

Tabla 11: Características técnicas del ordenador utilizado para las pruebas del generador de secuencias aleatorias por entrada de teclado.

5.2. GENERADOR DE SECUENCIAS PSEUDOALEATORIAS. GENERADOR DE BLUM, BLUM Y SHUB.

Un generador pseudoaleatorio obtiene las secuencias de números o de bits a través de un código o programa. Es un generador determinista con apariencia de aleatoriedad. Un PRBG arranca siempre desde un valor inicial llamado semilla. La calidad de la semilla es de vital importancia para el correcto uso de un PRBG. Una mala semilla empobrece la implementación de los sistemas criptográficos. Así como la resistencia de una cadena se mide en el eslabón más endeble, de la misma forma, si tenemos un sistema criptográfico sólido cuyo generador de claves es perfectamente predecible, tenemos un sistema tan débil como débil es el proceso de generación de valores aleatorios.

Un mismo generador produce secuencias diferentes cuando las semillas de entrada son diferentes. Si el mismo generador se ejecuta dos veces con la misma semilla inicial, en las dos ocasiones obtendremos idéntica secuencia. Los valores obtenidos mediante un PRBG son perfectamente predecibles para quien conoce la semilla inicial, debido al carácter determinista del algoritmo que los genera, y resulta de gran importancia que la semilla inicial de donde arranca toda la secuencia permanezca oculta. Cada PRBG se define mediante un algoritmo y unos parámetros, llamados claves, que junto con la semilla constituyen las claves secretas del generador: cualquiera que disponga del algoritmo, de sus claves y de la semilla inicial podrá obtener la secuencia completa.

Entre los muchos algoritmos de PRBG documentados, los que hemos seleccionado para un estudio inicial han sido los siguientes:

1. Sucesiones cifrantes producidas por LFSR. Una buena referencia de estos generadores se encuentra en [Mene97] y también [Mass94].
2. Generadores de NAOR–REINGOLD. [Naor97].
3. Generadores de tipo $x^m \pmod n$, para valores de m mayores o iguales que 2. Entre ellos, el generador BBS que es un caso particular de estos generadores con $m = 2$ [Blum86], [Mene97].

De entre todos estos algoritmos hemos escogido finalmente el generador BBS, definido por BLUM, BLUM y SHUB. La seguridad criptográfica de este generador se basa en la dificultad de extraer raíces cuadradas en $\mathbf{Z}_n^*$ y en la dificultad en factorizar n cuando es producto de dos primos grandes, cada uno de ellos congruentes con $3 \bmod 4$: es un generador de secuencias aleatorias impredecibles que pasan todos los test probabilísticos de tiempo polinómico. Los autores de [Blum86] demuestran que la obtención de un bit localizado en cada estado, o la obtención del bit de paridad de cada estado ofrece un generador criptográficamente seguro; y conjeturan esa misma propiedad si tomamos, de cada estado, la paridad y la localización.

Algunas referencias bibliográficas que apuestan por este generador como el más recomendado son: Bruce SCHNEIER [Schn96] afirma que es el más simple y el más eficiente de los algoritmos generadores basados en la complejidad teórica; William STALLINGS [Stal98] dice que BBS es el generador que tiene la prueba pública más fuerte de su seguridad criptográfica. También viene recomendado, como ya hemos señalado al principio del presente Capítulo, en [ETSI03]. Su principal desventaja es el costo de computación, más elevado que otros generadores.

5.2.1. Descripción del generador BBS.

El **generador BBS** es un generador de secuencias pseudoaleatorias. Como concluyen sus creadores [Blum86] es un generador que puede usarse en criptografía de clave pública y también como generador de secuencias pseudoaleatorias criptográficamente seguras, útil para cifrados de flujo.

Sean p y q primos tales que $p \equiv q \equiv 3 (\bmod 4)$. Sea $N = p \times q$. Sea x_0 un entero, residuo cuadrático módulo N (es decir, $x_0 \equiv u^2 (\bmod N)$ para algún entero u) con $mcd(x_0, N) = 1$. El generador BBS se define como $b_i = paridad(x_i)$, donde $x_i \equiv x_{i-1}^2 (\bmod N)$ para $i = 1,2,...$ trabajando en el conjunto de residuos no negativos.

Al entero N producto de los dos primos con las condiciones restrictivas antes indicadas se le llama **entero de BLUM**. Estas condiciones a los primos se exigen porque, como demuestra [Blum86] en su Lema 1, si los primos p y q cumplen esa condición entonces cada residuo cuadrático módulo N tiene exactamente una raíz cuadrada que es residuo cuadrático.

Los autores del generador BBS [Blum86] demuestran que el generador es impredecible. Postulan:

1. El conocimiento de N es suficiente para generar las secuencias $x_0, x_1, x_2, \ldots$ a partir de cada valor inicial x_0 (semilla) dado. Es por tanto suficiente para obtener la secuencia de bits $b_0, b_1, b_2, \ldots$
2. Dado N, sus factores p y q son necesarios y suficientes para generar secuencias en dirección reversible. Es decir, a partir de un valor cualquiera de la secuencia x_i podemos obtener todos los anteriores hasta llegar a x_0. [Blum86] recoge, en el Teorema 3, un algoritmo de reconstrucción de la secuencia hacia atrás conocidos los factores de N.

Esta propiedad de la impredecibilidad es la que en mayor medida ofrece seguridad a nuestro generador cuando se usa para criptografía de clave pública. Además de esta propiedad hay otras más que deben ser exigidas al generador para poder otorgarle la validez para usos criptográficos, y que veremos en el siguiente epígrafe.

Los dos empleos criptográficos de este generador son:

1. Cifrar en flujo con clave secreta, mediante la suma módulo 2 del mensaje a cifrar con la secuencia de bits generada. La clave estaría constituida por el módulo n y la semilla inicial x_0.
2. Como sistema de clave pública. La clave privada estaría constituida por los primos p y q, y la clave pública sería el valor del modulo n.

Queda demostrado en [Blum86] que el generador que toma cada valor b_i no mediante la paridad de cada x_i sino por la extracción de un bit localizado en cada estado también es criptográficamente seguro. Y deja conjeturado que el generador que toma de cada x_i un bit de cada uno de las dos formas (por paridad y por extracción de bit localizado) es criptográficamente seguro.

5.2.2. Propiedades del generador BBS.

Como explican los autores de [BBS86], muchas de las propiedades del generador BBS, que le otorgan las propiedades de la impredecibilidad y de la seguridad criptográfica se basan en la asunción de la intratabilidad de ciertos problemas de la teoría de números. Especialmente señalan dos de esos problemas ya antes recogidos en estas páginas: la dificultad de hallar, conociendo el valor y el valor x que verifique que $y \equiv x^2 (\mathrm{mod}\, N)$; y la dificultad para obtener los factores de un entero largo.

5.2.2.1. Impredecibilidad.

En [Blum86] se prueba que la secuencia descrita goza de la propiedad de la impredecibilidad: dado x_0 y N, pero desconocidos los factores de N, un criptoanalista no puede hacer de modo eficiente una predicción sobre los valores de $x_{-1}, x_{-2}, x_{-3}, \ldots$ Esta impredecibilidad se fundamenta

en la imposibilidad de obtener, conociendo y, el valor x que verifica que $y \equiv x^2 (\mathrm{mod}\, N)$ (podemos saber si el valor y es cuadrado perfecto módulo N mediante el cálculo del símbolo de JACOBI; pero no es en absoluto inmediato determinar cuál es el valor x de la congruencia).

Como ha quedado dicho, la seguridad del sistema reside fundamentalmente en esta propiedad: cualquier usuario puede generar la secuencia cifrante a partir de x_0, puesto que conoce la clave pública N, pero únicamente el usuario que posee la clave privada (p,q) puede calcular de forma eficiente la secuencia cifrante a partir del valor x_{k+1} hacia atrás.

La propiedad de impredecibilidad del generador BBS descansa en la ocultación de los valores p y q que forman N. Y si la seguridad del criptosistema depende de esa propiedad, es evidente que la seguridad de BBS descansa en la dificultad de factorizar números grandes. (Cfr. por ejemplo [Bren98], [Cusi95]).

La impredecibilidad también está supeditada a la longitud de la secuencia, puesto que si el periodo no es suficientemente elevado y la secuencia generada es de gran longitud, la seguridad del sistema se verá seriamente comprometida.

5.2.2.2. Acceso aleatorio.

Otra propiedad demostrada del generador es su acceso aleatorio: tanto x_i como x_{-i} pueden ser computados de modo eficiente para cualquier entero positivo i [Blum86]. Para este cálculo se requiere conocer la factorización del módulo N.

5.2.2.3. Periodo largo.

Los PRBG a usar en criptografía deben tener un periodo suficientemente largo. Esto no siempre ocurre en un generador del tipo $x^2 \bmod N$.

En [Blum86] se ofrece un algoritmo eficiente para encontrar el periodo de las secuencias y analizan las condiciones para las que los valores de N y x_0 logran secuencias de máximo periodo. Como resultado de ese estudio define los conceptos de primo especial (o también primo 2–seguro o doblemente seguro, que así los llamaremos de ahora en adelante) y de número especial. Un primo p se llamará **primo doblemente seguro** o **primos 2–seguro** si

$$p = 2 \cdot p' + 1 \text{ y } p' = 2 \cdot p'' + 1 \qquad (9)$$

donde p' y p'' son primos impares.

Un **número** N se llamará **especial** si $N = p \times q$, p y q son primos doblemente seguros distintos y se verifica que $p \equiv q \equiv 3 (\mathrm{mod}\, 4)$. Para obtener el periodo máximo se debe exigir que el módulo N sea un número especial.

Las condiciones a exigir en la generación de una clave módulo N, si deseamos obtener una órbita máxima ([Hern96]) quedan: (1) que los dos valores primos p y q sean doblemente seguros

(cfr. expresión (10)); y (2) que uno de los dos primos (p) verifique que es congruente

$$p \equiv 7(\mathrm{mod}16) \tag{10}$$

y el otro (q) que lo sea

$$q \equiv 7(\mathrm{mod}8) \tag{11}$$

Y la dificultad principal para generar claves públicas de estas condiciones en un criptosistema BBS es la de encontrar primos de estas condiciones en el ámbito de tamaños en el que se debe trabajar para evitar ataque por factorización: hay que buscar primos de estas características de tamaños mínimos en torno a los 512 bits. En [Bren98] se señala que los primos doblemente seguros son significativamente más difíciles de identificar que los primos de BLUM largos, cuya única condición es que verifiquen ser congruentes $3 \bmod 4$. Plantea si tomar el valor de N con la fuerte restricción de que sea número especial no pudiera reducir el trabajo de los criptoanalistas a la hora de pretender factorizar N .

Como se señala en [Alva98], con claves de ese orden de magnitud, la longitud de las secuencias generadas es de unos 2^{1021} bits (2^{988} Gigabytes). Por tanto, si bien los primos p y q deben tener una longitud de 512 bits para evitar ataques por factorización del módulo, no es necesario buscarlos de manera que el periodo de las secuencias generadas sea máximo. Los autores de [Alva98] demuestran que la no satisfacción de la condición de que los primos sean doblemente seguros sólo reduce el periodo de la secuencia a la mitad del máximo, que sigue siendo, de sobra, un valor suficientemente elevado. Bastaría por tanto exigir a p y a q que fuesen primos 1–seguro (decimos que un **primo** p es **1–seguro** si $p = 2 \cdot p' + 1$, con p' primo). Respecto a la condición de que p o q sean congruentes con $3 \bmod 8$, y teniendo en cuenta que la mitad de los primos lo son, existe una probabilidad del 75 % de que se cumpla esa condición buscando dos primos 1–seguro sin más limitación. Si finalmente ninguno de los dos primos satisfacen esa condición, entonces el periodo de la secuencia vuelve a dividirse por dos, pero sigue siendo todavía entonces una secuencia de enorme longitud, más que suficiente para un trabajo ordinario con este generador.

Falta ahora determinar qué propiedades deben tener las semillas para que los periodos que se obtengan a partir de ellas no tengan un ciclo corto. Para que la semilla produzca una órbita de periodo máximo se exige [Hern96]:

$$x_0 \not\equiv 1, -1, 0(\mathrm{mod}\ p) \qquad x_0 \not\equiv 1, -1, 0(\mathrm{mod}\ q) \tag{12}$$

La probabilidad de encontrar una semilla de estas condiciones es alta (mayor o igual que $1 - 3 \times (p^{-1} + q^{-1})$).

5.2.2.4. Simetría.

En [Cusi95] se realiza un estudio de esta propiedad. Entendemos por simetría la propiedad de que

la secuencia tenga igual frecuencia de unos que de ceros en una ejecución larga. El artículo muestra que para muchos valores de N y x_0 se obtienen secuencias no simétricas. Esta desviación se produce en mayor grado precisamente cuando procuramos valores de N y x_0 que garanticen los máximos periodos. Pero termina concluyendo que aún en el caso más desfavorable las asimetrías no son tan grandes como cabría esperar en una secuencia aleatoria de longitud similar.

5.2.3. Algunas de las distintas funciones implementadas.

Han quedado recogidas anteriormente las características que debe tener el valor del módulo y el valor de la semilla si se quieren lograr periodos máximos. También ha quedado dicho que se puede suavizar el rigor de esas condiciones, evitando exigencias de cálculo que, como veremos, llevan consigo las condiciones de que los primos sean 2–seguro. Una disminución en las exigencias de los primos, reduce el periodo de la secuencia, pero ya hemos visto que su longitud sigue siendo más que suficiente para los usos del generador.

Nosotros hemos trabajado todo el tiempo con el generador BBS, con los primos 2–seguro. Eso nos ha permitido conocer la dificultad que supone que nuestros módulos del generador sean especiales. Reducir esta condición a que los primos sean 1–seguro no supone apenas trabajo de implementación y facilita enormemente la búsqueda de los primos para obtener el módulo N del generador.

Para las exigencias del periodo máximo se ve necesario definir una función que cree claves adecuadas que verifiquen las condiciones recogidas en (9) (10) y (11). Otra función deberá generar una semilla inicial que verifique las condiciones para las semillas señaladas en (12).

Funciones más sencillas son las que realizan la operación de generación de nuevos valores de x_i, y otras dos que calculen, a partir de cada valor x_i, el valor b_i, bien sea por cálculo de la paridad, bien sea por determinada ubicación de un bit de x_i.

Finalmente quedan definidas algunas otras funciones auxiliares.

5.2.3.1. Algoritmo de generación de enteros largos aleatorios.

La forma que hemos implementado para asignar un valor entero aleatorio de una longitud (en número de bits) determinada a una variable tipo `NUMERO`, es colocar un uno en la posición que determina la longitud y asignar valores cero o uno a los restantes dígitos binarios situados a la derecha de ese primer uno, mediante el algoritmo de generación de bits pseudoaleatorios BBS.

Hemos definido inicialmente dos algoritmos, sustancialmente iguales, para este proceso. La diferencia que existe entre uno y otro estriba en el modo en que se obtiene el valor binario de cada iteración: por paridad o tomando un bit de cada iteración x_i.

Los prototipos de las funciones diseñadas son:

```
void GeneradorBBS_p(NUMERO*,UINT4,NUMERO*,NUMERO*);
```

```
void GeneradorBBS_b(NUMERO*,UINT4,NUMERO*,NUMERO*);
```

La función recibe como parámetros la dirección de la variable tipo `NUMERO` donde quedará almacenado en forma de entero largo la nueva secuencia generada. Un segundo parámetro es la longitud (número de bits) que debe tener el número generado: es decir, el número de iteraciones que se deberán hacer. Los otros dos parámetros son los del módulo N y de la semilla x_0.

Estas funciones asignan bit a bit un valor de entero largo a la variable de tipo `NUMERO` recibida como primer parámetro. La función hace dos comprobaciones:

1. Que no se solicita asignar más bits de los que puede albergar en la dimensión en que ha sido creada la variable sobre la que se asignará un valor aleatorio.
2. Que la semilla que se ha tomado para los cálculos tiene una dimensión suficientemente grande para las operaciones de producto modular módulo N.

Ambas circunstancias exigirían abortar el proceso.

Los procesos de generación de enteros largos aleatorios definidos cumplen las exigencias presentadas en [Blum86] que lo hacen criptográficamente seguro: tomar un bit de paridad de cada nuevo valor x_i generado o tomar un bit de posición de cada nuevo valor x_i generado. La dificultad es la lentitud de este generador.

Un modo de incrementar la velocidad es tomando más valores binarios de cada nuevo x_i obtenido. Si hacemos esto, dejamos de tener un respaldo matemático que garantice la seguridad criptográfica del generador tal y como lo presentaron sus creadores. Pero indudablemente multiplicamos su velocidad. En [Vazi85] se demuestra que la seguridad del generador queda garantizada si de cada valor x_i tomamos $\log_2 \log_2 x_i$ bits de información para la secuencia que se está generando. Esta seguridad queda garantizada bajo el supuesto de la dificultad de factorizar el módulo del generador (que será del mismo rango que los valores x_i sucesivos).

Tomando en consideración esta afirmación demostrada, hemos definido otros nuevos procedimientos de generación de aleatorios, donde para cada elemento `UINT4` de x_i tomamos 4 bits para la secuencia. Esto supone un muy notable incremento en la generación de aleatorios –como veremos posteriormente en las gráficas comparativas de tiempos– pues, si por ejemplo, trabajamos con un módulo de 128 bits (que no es en absoluto grande, sino más bien demasiado pequeño) obtenemos 16 bits de información para la secuencia que generamos en cada x_i calculado, y no sólo un bit, como ocurría antes.

Al trabajar con enteros largos cuyos dígitos están formados por elementos `UINT4` (**`unsigned long`**

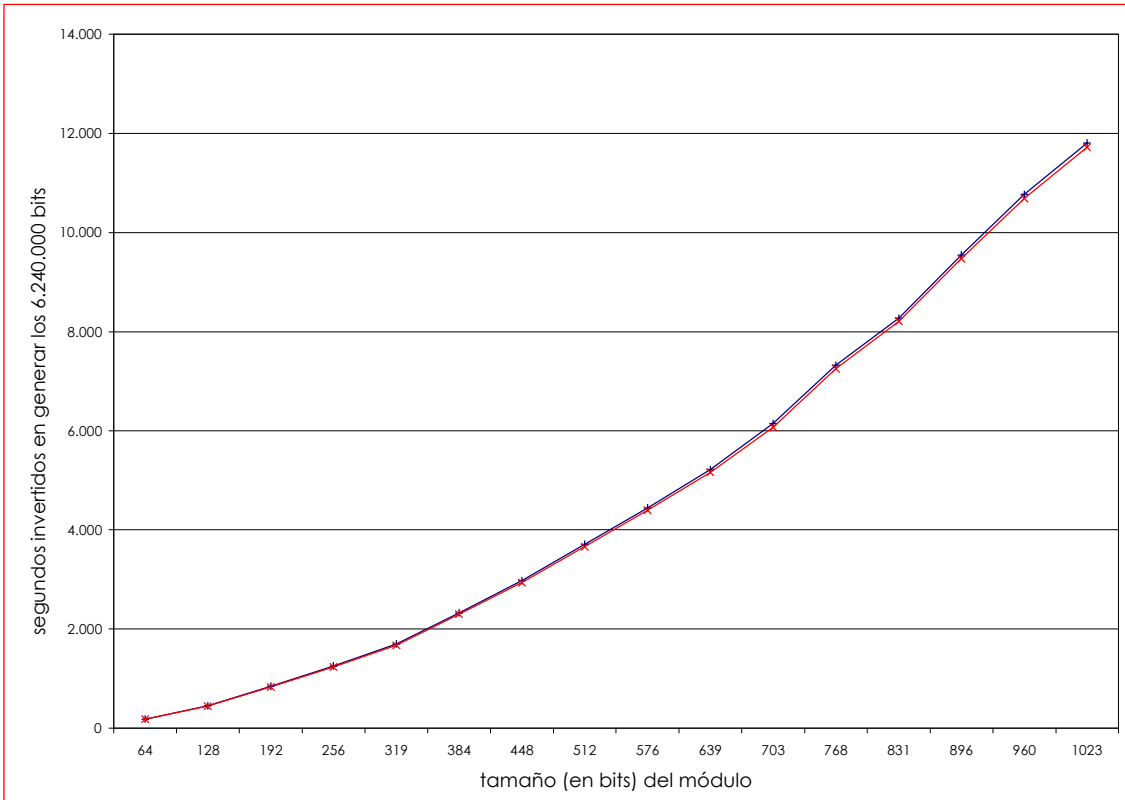


Gráfico 2: Tiempos que necesitan las funciones `GeneradorBBS_p()` (en azul) y `GeneradorBBS_b()` (en rojo) para generar 6.240.000 bits, necesarios para dar valor aleatorio a 500 enteros de 320 bits, 500 enteros de 352 bits,... hasta 500 enteros de 928 bits. Los tiempos (ordenadas) vienen recogidos en función del tamaño del módulo del generador BBS (abcisas).

`int`) la base de numeración es 2^{32}. Considerando uno de esos dígitos de x_i, y tomando el doble logaritmo en base 2 sobre el posible valor de ese dígito (que puede tomar cualquiera entre 0 y $2^{32}-1$ todos ellos equiprobables) resulta que de cada dígito de x_i podemos tomar hasta 5 bits de información:

$$\log_2 \log_2 2^{32} = \log_2 32 = \log_2 2^5 = 5$$

Por tanto, si tomamos 4 bits por cada dígito `UINT4`, estamos trabajando dentro del rango permitido y establecido en [Vazi85].

Los prototipos de las funciones diseñadas (tomando cuatro bits posicionados de cada elemento `UINT4` o dígito o tomando la paridad de cada byte del elemento `UINT4`) son:

```
void GeneradorBBS_P(NUMERO*,UINT4,NUMERO*,NUMERO*);
```

```
void GeneradorBBS_B(NUMERO*,UINT4,NUMERO*,NUMERO*);
```

Habrá que comprobar que el rendimiento de tiempos es realmente interesante. Para este estudio se han dado valor a 10.000 enteros largos (estructuras `NUMERO`) de tamaños comprendidos entre los 320 y los 928 bits (incrementando los tamaños de 32 en 32 bits): 500 para cada tamaño. En

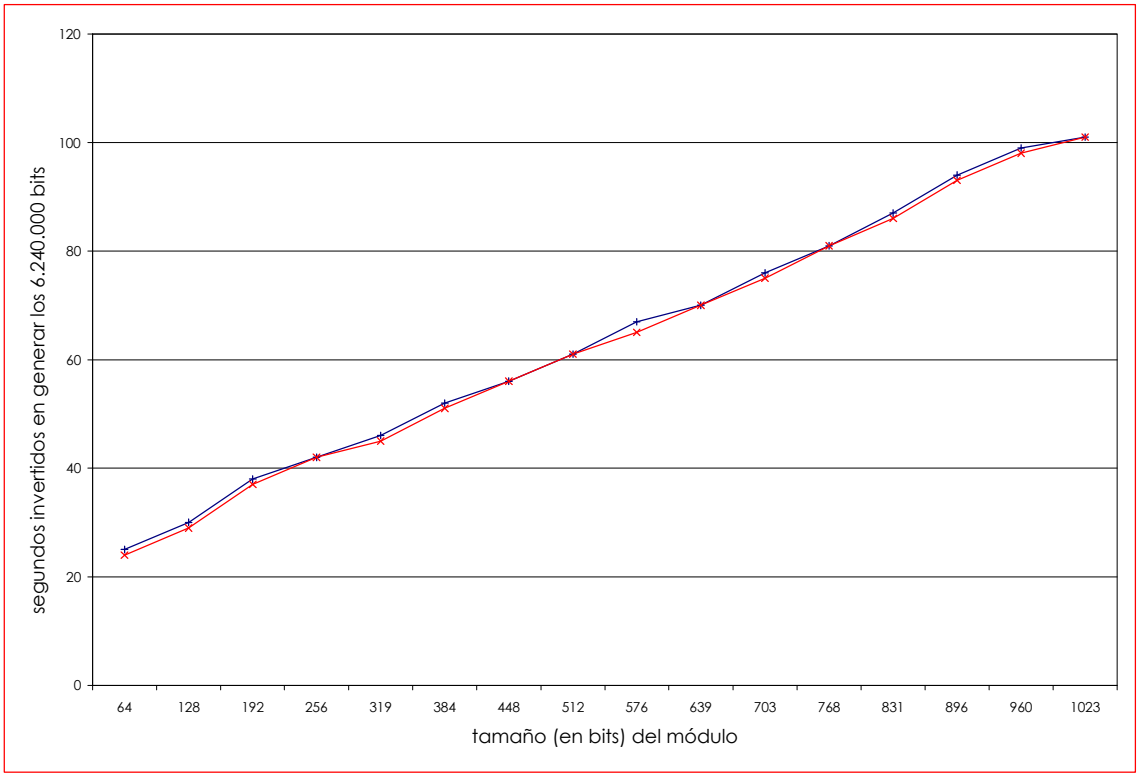


Gráfico 3: Tiempos que necesitan las funciones `GeneradorBBS_P()` (en azul) y `GeneradorBBS_B()` (en rojo) para generar 6.240.000 bits, necesarios para dar valor aleatorio a 500 enteros de 320 bits, 500 enteros de 352 bits,... hasta 500 enteros de 928 bits. Los tiempos (ordenadas) vienen recogidos en función del tamaño del módulo del generador BBS (abcisas).

total, por tanto se han generado 6.240.000 bits. Repetimos este proceso para diferentes tamaños del módulo del generador: desde un primer módulo de 64 bits, producto de dos primos 2–seguro de 32 bits hasta un módulo de 1023 bits, producto de dos primos 2–seguro de 512 bits. En total 16 módulos diferentes. Se han generado por tanto, un total de 99.840.000 bits.

El objeto de todo este trabajo es estudiar el tiempo que emplea el algoritmo BBS en generar un número determinado de bits para diferentes tamaños de enteros, en función del tamaño del módulo empleado. Las gráficas obtenidas representan el tiempo invertido en generar 6.240.000 (ordenadas) para cada uno de los tamaños del módulo empleado para el generador (abcisas).

Además hemos realizado todo el proceso de estudio descrito para cada uno de los cuatro generadores BBS implementados. En la Gráfica 2 se puede ver el comportamiento en el tiempo de los generadores primeros, que toman 1 bit por cada valor de x_i. En color azul se recoge el tiempo de generación de los números según los tamaños en bits del módulo del generador cuando se toma el bit de paridad de cada nuevo x_i generado. En color rojo se recoge el comportamiento cuando se toma un bit de posición por cada x_i generado. En ambas gráficas podemos observar que, a medida que aumenta el tamaño del módulo, el costo de tiempo para generar bits

aumenta muy considerablemente. El motivo es obvio: al aumentar el módulo aumenta el costo de tiempo en todos los cálculos que debe hacer el generador BBS pero no aumenta el número de bits que obtenemos: seguimos tomando uno por cada x_i generado. La diferencia de tiempos entre una función y otra es mínima y , de hecho, ambas gráficas están casi superpuestas.

Podemos ver ahora la Gráfica 3 de los nuevos generadores, en los que hemos considerado, amparados en lo que dice [Vazi85], que podemos tomar hasta 5 bits por cada 32 generados sin perder por ello seguridad. Son los resultados de ejecutar las funciones `GeneradorBBS_P()` (en azul) y `GeneradorBBS_B()` (en rojo). El incremento tiene ahora una forma más lineal.

Y, desde luego, los valores absolutos de los tiempos invertidos en la generación de los bits aleatorios son notablemente inferiores. Basta comparar las 21 horas, 26 minutos y 6 segundos necesarios para la generación de todos los números en todos los tamaños con el generador `GeneradorBBS_p()`, con los 14 minutos y 16 segundos necesarios para la misma generación con `GeneradorBBS_B()`. Todo este proceso de generación se ha realizado en una máquina cuyas características están recogidas en la Tabla 11 mostrada unas páginas más arriba.

Hemos obtenido las ecuaciones de las curvas recogidas en las Gráficas 2 y 3. Las dos primeras se comportan de forma potencial, como una gráfica de un polinomio de grado 2 de la forma $y = a \cdot x^2 + b \cdot x + c$. Las curvas de la Gráfica 3 tienen un comportamiento lineal de la forma $y = b \cdot x + c$. En la Tabla 12 recogemos los valores de los parámetros a , b y c para cada una de las cuatro gráficas.

	a	b	c
GeneradorBBS_p()	35,77	175,89	27,964
GeneradorBBS_b()	35,794	170,19	27,916
GeneradorBBS_P()	0	5,1044	20,675
GeneradorBBS_B()	0	5,1206	19,85

Tabla 12: Valores de los coeficientes que definen las ecuaciones de las funciones de las gráficas 2 y 3.

5.2.3.2. Test de MAURER para nuestras implementaciones más rápidas.

Ya hemos señalado antes que una de las principales pegas que tiene el generador BBS es su alto coste computacional. Luego hemos podido ver algunas medidas de los tiempos necesarios para generar bits para dar valor a enteros de diferentes tamaños. Hemos visto que haciendo caso a la sugerencia de VAZIRANI y VAZIRANI de tomar un bit de cada byte generado, aumenta en mucho la velocidad del generador.

Hemos querido someter a nuestra implementación, con la modificación sugerida en [Vazi85], al test universal de MAURER, y verificar si podemos trabajar con él como un generador cuyas

secuencias pseudoaleatorias tienen un comportamiento estadístico adecuado. Como la generación de las secuencias con las nuevas funciones que queremos testear (`GeneradorBBS_P()` y `GeneradorBBS_B()`) sigue siendo una tarea de alto coste, hemos realizado un test de MAURER con valores de las constantes Q y K un poco reducidos. Para el intervalo de valores de L definido ($6 \le L \le 16$) hemos escogido $Q \ge 8 \cdot 2^L$ y $K \ge 100 \cdot 2^L$, lo que supone tener que generar para el caso $L = 16$ un total de $108 \cdot 2^L \cdot L = 113.246.208$ bits. En la Tabla 13 recogemos los valores obtenidos para las diferentes X_L, acompañados de los valores k_1 y k_2 que recogen los extremos donde los valores X_L son válidos y suponemos por tanto que se logra superar el test.

L	k1	X[L]	k2
6	5,193388	5,228	5,242023
7	6,177841	6,162	6,214660
8	7,169999	7,181	7,197332
9	8,166408	8,190	8,186441
10	9,165047	9,198	9,179601
11	10,164777	10,169	10,175287
12	11,164985	11,165	11,172545
13	12,165360	12,168	12,170780
14	13,165754	13,165	13,169632
15	14,166103	14,166	14,168873
16	15,166391	15,168	15,168367

Tabla 13: Valores del estadístico X_u obtenidos al testear el generador BBS que toma 1 bit por cada byte generado: función `GeneradorBBS_B()`.

En 8 de los 11 valores de L el valor cae dentro del intervalo. En los otros tres casos ($L = 7,9,10$) en uno se queda corto por una centésima y en los otros dos se excede por una y dos centésimas. Damos por superado el test de MAURER para nuestro generador BBS.

5.2.3.3. Funciones para la generación de primos de BLUM y primos especiales.

Hemos llamado primo de BLUM a aquel entero primo p que verifique que $p \equiv 3 (\mathrm{mod}\, 4)$. Y hemos definido una función sencilla que genera un primo de esas características, cuyo prototipo es:

```
void GeneradorPrimoBBS(NUMERO*,UINT4*);
```

Esta función invoca al generador de aleatorios por entrada aleatoria del teclado. Al valor obtenido con esta llamada se le exige que sea impar (el bit menos significativo debe valer 1) y que sea además congruente $3 \bmod 4$ (el segundo bit menos significativo debe valer también un 1).

Después al candidato a primo de BLUM se le somete a prueba con dos tests de primalidad: el test de intento de división por los primeros primos más pequeños y el test de MILLER–RABIN. Si pasa esos dos tests ya hemos terminado el proceso. El motivo por el que exigimos al proceso que el candidato supere dos tests diferentes de primalidad es la optimización de tiempos: el primero es mucho más veloz que el segundo, y descarta un alto porcentaje de candidatos; un candidato a primo pasa el test de MILLER–RABIN únicamente si ha pasado ya el primer test de intento por división. Todos los candidatos descartados en el primero de los dos tests no han de superar el segundo test, mucho más costoso en tiempo de computación. Los autores de [Mene97] recomiendan hacer el test por divisiones sucesivas con todos los primos menores de 256.

El interés de este primer test por intento de división viene justificado por el alto número de enteros impares que tienen un divisor entre los primos más bajos. En [Ries87] se recoge una tabla (que reproducimos en la Tabla 14) de la proporción (g) de números impares sin un factor menor que G

G	g	G	g	G	g
10^2	0,2406	10^5	0,0975	10^8	0,0610
10^3	0,1619	10^6	0,0813	10^9	0,0542
10^4	0,1218	10^7	0,0697	10^{10}	0,0488

Tabla 14: Proporción (g) de enteros impares que no tienen un factor menor que G.

A los primos generados con esta función les exigimos un tamaño máximo de 64 bits. Imponemos esta condición porque esta función ha sido definida únicamente para iniciar la generación de primos especiales tal y como ha quedado explicado antes. El interés por generar estos primos en esta función está en poder arrancar un generador BBS de calidad suficiente para, con él, emprender la tarea de la búsqueda de primos especiales. Para esta segunda tarea ha quedado definida la función cuyo prototipo es

```
void GeneradorPrimoBBS_2_seguro(NUMERO*,UINT4*,NUMERO*,NUMERO*)
```

La función recibe como parámetros primero la dirección donde queremos almacenar el primo de BLUM doblemente seguro; segundo la dirección de un vector de primos (una tabla con los primeros primos de los enteros, generada mediante la criba de ERASTHOTENES); tercero, la dirección del módulo entero de BLUM que será un entero generado mediante dos primos de BLUM que a su vez habrán sido creados mediante la función anterior `GeneradorPrimosBBS()` antes presentada; el cuarto parámetro es la dirección de la semilla inicial, que se genera mediante la función `GeneradorSemillas()` que se presenta más adelante.

El algoritmo genera valores aleatorios mediante la función `GeneradorBBS_p()` con el valor del módulo de baja calidad criptográfica, `N,` obtenido con la función `GeneradorPrimosBBS()`, y con

la semilla, recibida como cuarto parámetro.

El algoritmo busca un primo doblemente seguro p que verifique que, si $p = 2 \cdot p' + 1$, el primo p' cumple que $p' \equiv 3 (\mathrm{mod} 8)$. Para esta condición basta exigir a p que verifique $p \equiv 7 (\mathrm{mod} 16)$ que se cumplirá si y solo si los tres bits menos significativos de p son `1` y el cuarto bit menos significativo es `0` (es decir, ponemos el bit hexadecimal menos significativo igual a `0x7`). Esa es la primera exigencia que le ponemos al candidato a `BBS`.

Una vez tenemos un entero con estos tres últimos bits asignados el candidato debe pasar tres tests de primalidad, tanto para p como para p' y p'' ($p'' = (p' - 1)/2$). Esto es lo que hace el resto del código de la función. Para cada uno de los tres números (p, p' y p'') pasamos tres tests: si los tres superan el test de división por los primos pequeños entonces exigimos a los tres que superen el test `FastRabin()` ya definido y explicado antes. Y si los tres pasan ese segundo test entonces exigimos a los tres que pasen el tercero y más severo de los tests: el test de MILLER–RABIN. Si alguno de los tres valores no supera alguno de los tests, entonces se genera otro candidato y el actual se desecha.

La razón por la cual hacemos pasar el candidato por tres tests es la de ahorrar tiempo de computación. El candidato p y los otros dos valores p' y p'' pasarán el test de MILLER–RABIN (más costoso que los otros dos en tiempo de computación) si estos tres números han logrado pasar los otros dos tests, de poca fiabilidad pero de bajo costo de computación.

El primo que se obtenga de esa función es un primo 2–seguro válido para, junto con otro, generar un entero especial de BLUM como se requiere para obtener generadores de órbita máxima.

Como señala [Alva98], no es demasiado aconsejable utilizar un método de prueba y error (escogiendo un número al azar y comprobando si es dos veces seguro), de forma recursiva hasta encontrar el primo buscado, pero no se presentan otras formas de hallar estos primos doblemente seguros.

En el anexo III presentamos unas tablas de primos BBS 2–seguro y de los tiempos invertidos en la generación de primos doblemente seguros en diferentes tamaños. La tabla presentada muestra los tiempos cuando el módulo BBS empleado en el generador tiene un tamaño de 64 bits y se ha empleado la función `GeneradorBBS_B()`. También quedan recogidos (en las tablas del Anexo III) estos valores primos generados. Como se puede ver, los tiempos tienen un comportamiento completamente aleatorio. Por ejemplo, con los primos doblemente seguros que son congruentes $7 (\mathrm{mod} 16)$ de tamaño 512 bits hemos necesitado tiempos desde un afortunado caso en el que han sido suficientes 46 segundos, como otros tiempos tan largos como 34.708 segundos (lo que supone una duración de 9 horas, 38 minutos y 28 segundos). A medida que crece el tamaño del primo especial que se busca crece la dificultad en su hallazgo.

5.2.3.4. Función para generar semillas que describan una órbita de periodo máximo.

También hemos visto cuáles son las condiciones que debe exigirse a la semilla generada (ver expresión (12)) para que arranque una órbita de periodo máximo.

La función

```
void GeneradorSemillas(NUMERO*, char*,UINT4);
```

exige conocer el valor de los primos p y q que factorizan al entero especial módulo de nuestro generador. Estos valores son la clave secreta y están almacenados en un archivo en formato binario. La función abre el archivo (el nombre viene recogido en el segundo parámetro) y lee los valores de los dos primos y luego genera, por entrada aleatoria de teclado, un valor candidato a semilla al que se le exigirá que verifique las seis condiciones presentadas antes.

La función recibe como parámetros la dirección donde quedará recogida la semilla, el nombre del archivo donde están almacenados los valores de los primos p y q, y la longitud que tienen los primos del archivo (es decir, número de elementos `UINT4` necesarios para que las variables que almacenen el valor de los primos den cabida a todos los dígitos).

La función deja en la variable semilla un valor que verifica todas las condiciones para el módulo que se va a utilizar posteriormente para el generador.

5.3. REFLEXIONES FINALES

Para nuestro trabajo de implementación de las funciones que definen el tipo de dato `NUMERO`, ya definido y presentado en el Capítulo 4, se hacía necesaria una herramienta que nos facilitase valores numéricos aleatorios para las variables que empleábamos para verificar el buen funcionamiento de esas funciones. Así surgió la primera implementación de un generador de aleatorios por entrada de teclado. Ese generador era suficiente para obtener varios cientos de bits aleatorios y proceder a las sumas, desplazamientos, cocientes, etc.

Más adelante, cuando empezamos a trabajar con el algoritmo de factorización, surgió una necesidad similar, pero de mucho mayor volumen: necesitábamos disponer de muchos valores enteros, producto de dos primos de longitudes prefijadas. De esta necesidad surgió el ánimo para implementar el generador BBS.

Más adelante quisimos centrar nuestro estudio sobre los generadores definidos e implementados. Ya ha quedado explicado a lo largo de la presentación de este tema que los generadores de secuencias aleatorias son herramientas clave para muchas tareas criptográficas, y que no todo algoritmo pretendidamente válido para obtener bits de forma aleatoria es adecuado para la criptografía. Por eso comenzamos a estudiar más a fondo nuestros generadores, que hasta entonces debían su implementación a una simple necesidad de disponer de secuencias de

valores.

El generador por entrada de teclado fue sufriendo no pocas modificaciones, en busca de cada vez mejores resultados en los tests estadísticos de contraste de hipótesis. Al final llegamos al generador que hemos presentado en el epígrafe anterior, que consideramos válido para uso criptográfico en la búsqueda de semillas para generadores pseudoaleatorios.

Y, aunque no le hemos dado validez para usos criptográficos, es desde luego veloz el generador que resulta de emplear nuestro algoritmo con entrada y cadencia prefijada por programa, sin necesidad de intervención de usuario. Hemos sometido ese comportamiento al test universal de MAURER y los resultados, aunque no los hemos dado por buenos, no han sido en absoluto malos.

La implementación del algoritmo BBS ha resultado más trabajoso. Sin embargo, su calidad estaba ya demostrada y no cabía en nuestro trabajo hacer valoración alguna sobre el generador. Sin embargo, sufríamos la lentitud del generador. Como se dice en [Ding97], el procedimiento del generador es lento y quizá por eso no sea recomendable para aplicaciones multimedia: resulta razonablemente seguro para comunicación diplomática o militar. Esa afirmación merecía que intentáramos agilizar el funcionamiento del generador. Amparados en la recomendación recogida en [Vazi85] hemos aumentado el número de bits en cada nuevo proceso de cálculo del generador. Hemos mostrado las nuevas velocidades que han resultado ser muy inferiores. Quedaba hacer un estudio de la calidad estadística del generador BBS en el uso recomendado por VAZIRANI y VAZIRANI, y por eso hemos sometido a las secuencias generadas por nuestro algoritmo implementado al test universal de MAURER. El resultado ha sido satisfactorio y podemos considerar que la implementación es válida desde un punto de vista estadístico. Y desde luego suficientemente rápida como para que pueda ser usada más allá de las necesidades de tipo diplomático o militar.

Hemos estudiado también los primos 2–seguro, que son aquellos que permiten disponer de las órbitas de periodo máximo. Hemos estudiado los tiempos que se requiere para la generación de uno de esos primos, cuando trabajamos en tamaños lo suficientemente grandes como para evitar ataques por factorización. Pensamos que, aunque estos primos son los mejores a la hora de garantizar órbitas máximas, es conveniente reducir las condiciones a los primos que producen el módulo del generador BBS para lograr tiempos mucho más reducidos para su búsqueda y obtener periodos, sí, más cortos, pero más que suficientemente largos para los propósitos habituales de este generador.

6 DISEÑO E IMPLEMENTACIÓN DEL ALGORITMO DE FACTORIZACIÓN POR FRACCIONES CONTINUAS

Las técnicas de factorización de velocidad subexponencial que se han utilizado en los últimos 20 años son la técnica de las fracciones continuas (CFRAC), introducida en 1975 por Michael A. MORRISON y John BRILLHART [Morr75], la técnica de la criba cuadrática (QS), introducida a principio de la década de los 80 por Carl POMERANCE [Pome82], y la técnica de la criba de campo numérico (NFS), introducida en los primeros años de la década de los 90 por A. K. LENSTRA, H. W. LENSTRA jr. M. S. MANASSE, J. M. POLLARD Y C. POMERANCE y recogida en [Lens93].

Todas ellas se basan en la técnica de factorización introducida por FERMAT que pretende la búsqueda de una relación de la forma

$$x^2 \equiv y^2 \pmod{N} \qquad (1)$$

siendo N el número que se desea factorizar. De la relación (1), se deduce que $x^2 - y^2 = k \cdot N$, es decir, que

$$(x+y)\cdot(x-y)=k\cdot N \tag{2}$$

y buscando mediante el algoritmo de EUCLIDES el máximo común divisor de $(x+y)$ con N y de $(x-y)$ con N, tenemos un 50 % de probabilidades de que el resultado sea un factor no trivial de N.

Los tres métodos señalados coinciden en dedicar el máximo de sus esfuerzos en lograr hallar una relación como la recogida en (1). Todos ellos procuran métodos de generación de relaciones de la forma

$$P^2 \equiv C(\mathrm{mod}\, N) \tag{3}$$

que verifiquen que C tiene todos sus divisores primos menores que un valor límite prefijado B. Una vez encontradas y almacenadas suficientes relaciones de esta forma (3) se buscan subconjuntos de estas relaciones halladas, que llamaremos subconjuntos S, que verifiquen que el producto de todos los C de cada uno de esos subconjuntos sea cuadrado perfecto. Para la búsqueda de los diferentes subconjuntos S se utiliza la técnica de eliminación gaussiana. Todo este proceso viene ampliamente documentado en muchos buenos libros (por ejemplo [Cohe93] y [Rose93]).

El producto de todos los elementos de uno subconjunto S nos llevará a una expresión de la forma

$$\prod_{Ci\in S} P_i^2 \equiv \prod_{Ci\in S} C_i\,(\mathrm{mod}\, N) \tag{4}$$

donde al haberse logrado que la parte derecha de la congruencia sea cuadrado perfecto, tendremos que la expresión (4) es análoga a la (1).

De los tres algoritmos subexponenciales, cada nuevo que apareció superaba en velocidad al anterior a partir de un tamaño del entero a factorizar. Entre ellos se diferencian fundamentalmente en el modo de buscar relaciones de la forma (3), y el factor diferenciador de cada uno de ellos reside en el modo de lograr cuánto antes una cantidad suficiente de esas relaciones donde sus respectivos valores C tengan todos sus divisores primos menores que un límite B prefijado.

Una vez hallada una relación de esa forma se debe probar, como acabamos de decir, que C tiene todos sus divisores primos menores que un valor límite prefijado; si no es así, la relación es desechada y se pasa a probar una nueva relación. Por tanto, ganamos velocidad si

1. generamos nuevas relaciones de forma más rápida;
2. generamos relaciones que tengan mayor probabilidad de que C sea tenga todos sus divisores primos por debajo del límite marcado; y esa probabilidad aumenta a medida que se logran valores de C menores (principal ventaja del método CFRAC); o a medida que se logra determinar con cierta probabilidad que esa propiedad se cumple antes de generar el valor de C (principal ventaja del método QS);

3. como hemos señalado en el Capítulo 1 y veremos más desarrollado en el Capítulo 7, ganaremos velocidad si optimizamos el código implementado.

Este tercer camino —el de la optimización del código— es el que hemos querido recorrer en esta tesis: tomar una implementación de un algoritmo de factorización e intentar reducir los tiempos de ejecución. Para este proceso, debemos escoger uno de los algoritmos con los que vamos a trabajar, implementarlo, y posteriormente optimizarlo. ¿Cuál de los tres algoritmos escoger?

Nosotros hemos implementado y posteriormente optimizado el algoritmo basado en el desarrollo de las fracciones continuas. Las razones de esta elección (razones no determinantes, sino de conveniencia, o prudenciales) son las siguientes:

1. Históricamente es el primero, y es con el algoritmo CFRAC con el que se comienzan a afrontar, en la comunidad científica, los primeros retos de factorización de enteros grandes.
2. Los tres algoritmos comparten muchos de los procesos (la búsqueda de un gran número de congruencias módulo N y el posterior paso de eliminación gaussiana sobre el Cuerpo $\mathbf{Z}/2\mathbf{Z}$) y muchas de las funciones básicas (todas las operaciones aritméticas, matemáticas y auxiliares y algunos de los algoritmos sobre los test de primalidad presentados en los Capítulos 2 y 4). La optimización de todas estas funciones, realizada para mejorar el rendimiento de uno cualquiera de estos tres algoritmos de factorización, será igualmente válida para la mejora del rendimiento de cualquiera de los otros dos algoritmos. (Estas dos primeras razones son las mismas que recoge Henri COHEN en su libro [Cohe93] para recomendar el estudio de este algoritmo).
3. Es el más rápido de los tres para enteros de un orden menor de 35 dígitos decimales. Su tope superior queda colocado (cfr. [Bree00]) en enteros de 50 dígitos. Y ése es el ámbito en que nosotros vamos a trabajar. Además, el proceso de optimización exige la ejecución, repetida muchas veces, de la aplicación que se desea optimizar (varias decenas de miles de veces la hemos ejecutado para el trabajo presentado en el Capítulo 7). Por eso, es recomendable trabajar, en la optimización, con enteros no demasiado grandes, que enlentecerían mucho el proceso.
4. Es el más sencillo de implementar, el más sencillo en su presentación matemática. Ya hemos dicho en el Capítulo de Introducción que un camino para llegar a factorizar enteros cada vez más grandes en menor tiempo se recorre desde el análisis matemático, en la búsqueda de métodos de factorización de menor coste computacional. Este no es el camino que pretendemos recorrer en esta tesis. La otra vía es más experimental, y se recorre desde la ciencia de la computación y la implementación del algoritmo: ese es el camino que describimos en la tesis. Por eso, pensamos que, si bien es verdad que los algoritmos QS y NFS son mejores desde un punto de vista computacional, el algoritmo CFRAC, por su mayor sencillez, resulta más conveniente a la hora de presentar y de poner en práctica un protocolo

de optimización de código. Por otro lado, aunque digamos que es el más sencillo de los tres, su implementación no es en absoluto trivial, y el trabajo de optimización ha exigido mucho volumen de trabajo, de análisis de muy diversas funciones e instrucciones. Y la dificultad de comprensión matemática de un algoritmo como el NFS no hubiera ayudado a una mayor claridad en la presentación de nuestro protocolo de optimización.

6.1. DESCRIPCIÓN DEL ALGORITMO DE FACTORIZACIÓN POR LA TÉCNICA DE LAS FRACCIONES CONTINUAS.

Sea $N>1$, un entero compuesto impar. El procedimiento a seguir en el método de factorización de CFRAC es el que sigue (tomados, estos tres pasos, de [Morr75]; ver también [Ries87]):

1. Mediante el algoritmo de BHÁSCARA–BROUNCKER, basado en la técnica de las fracciones continuas, buscamos una expresión racional, aproximada del valor real $\sqrt{N}$ ó $\sqrt{k\cdot N}$, para algún $k\geq 1$ entero. En el proceso generamos cinco secuencias (A_i, B_i, C_i, P_i y Q_i), donde la fracción P_i/Q_i ofrece la aproximación i–ésima en forma racional del valor real $\sqrt{N}$.En cada nueva iteración, el valor P_i/Q_i queda más próximo al valor de $\sqrt{N}$. En el desarrollo del algoritmo de generación de estas cinco secuencias, se verifica la siguiente expresión:

$$P_{n-1}^2 - k\cdot N\cdot Q_{n-1}^2 = (-1)^n\cdot C_n,$$

y si aplicamos el operador $\bmod N$ a ambas partes de la igualdad, tenemos que

$$P_{n-1}^2 \equiv (-1)^n\cdot C_n \pmod N$$

Expresión que es de la misma forma que la recogida en /(3). Llamaremos, a cada par de enteros positivos (P_{n-1}, C_n) en esta congruencia, un **par $P-C$.**

2. El segundo paso consiste en encontrar, de entre el conjunto de pares $P-C$, ciertos subconjuntos, que llamaremos "**conjuntos *S***", cada uno de ellos con la propiedad de que el producto

$$\prod_i (-1)^i\cdot C_i$$

de todos los C_i's de cada conjunto S sea un valor cuadrado perfecto. Si no se encuentra ningún subconjunto de esas características, entonces se debe volver al paso 1 en busca de más pares $P-C$.

3. Cada conjunto S encontrado en el paso anterior verifica la congruencia

$$P^2 \equiv \prod_i P_{i-1}^2 \equiv \prod_i (-1)^i\cdot C_i = C^2 \pmod N,$$

donde $1\leq P<N$. Calculamos los valores de P y de C y $mcd(P-C,N)=D$, para los diferentes conjuntos S hallados. Si en algún conjunto S tenemos que $1<D<N$, entonces el método ha logrado encontrar un factor no trivial de N, que es precisamente D.

6.1.1. Búsqueda de pares $P-C$.

Para la búsqueda de los pares $P-C$ se ha tomado el algoritmo de BHÁSCARA–BROUNCKER, presentado en [Bres89] y recogido aquí en el Algoritmo 1. Tiene como entrada el número N, y genera las secuencias A_i, B_i, C_i, P_i, Q_i.

1. INICIALIZAR VARIABLES
   $M \leftarrow \lfloor \sqrt{k \cdot N} \rfloor$
   $A_1 \leftarrow M$
   $B_0 \leftarrow 0 \qquad B_1 \leftarrow M$
   $C_0 \leftarrow 1 \qquad C_1 \leftarrow N - (M \times M)$
   $P_0 \leftarrow 1 \qquad P_1 \leftarrow M$
   $Q_0 \leftarrow 0 \qquad Q_1 \leftarrow 1$
   $i \leftarrow 1$
2. MIENTRAS ($C_i \neq 1$) HACER
   2.1. $k \leftarrow i-1 \qquad j \leftarrow i \qquad i \leftarrow i+1$
   2.2. $A_i \leftarrow \lfloor (M + B_j)/C_j \rfloor$
   2.3. $B_i \leftarrow A_i \times C_j - B_j$
   2.4. $C_i \leftarrow C_k + A_i \times (B_j - B_i)$
   2.5. $P_i \leftarrow P_k + A_i \times P_j$
   2.6. $Q_i \leftarrow Q_k + A_i \times Q_j$

Algoritmo 1: Algoritmo de BHÁSCARA–BROUNCKER para la generación de los valores de las secuencias que aproximan un racional a un valor real (raíz cuadrada de un entero dado como entrada del algoritmo). De las cinco secuencias generadas, dos de ellas verifican una relación semejante a la expresión (3) y son válidas, por tanto, para intentar factorizar el valor de entrada.

Recogemos algunas observaciones sobre los valores de las variables y los pasos del Algoritmo 1:

1. Los enteros B_i y C_i toman sus valores en los límites $0 \le B_i < \sqrt{k \cdot N}$, $0 \le C_i < 2 \cdot \sqrt{k \cdot N}$, para $n \ge 0$. Esta afirmación, tomada de [Morr75], nos indica no sólo el límite superior de ambos valores, sino también que ambos valores son siempre positivos, hecho que nos ha simplificado en cierta medida la implementación: recuérdese que, como señalábamos en el Capítulo 4, la estructura que define nuestra variable `NUMERO` no dispone de un campo para el signo.
2. La expansión en fracciones continuas de $\sqrt{k \cdot N}$ es siempre periódica, dados los límites entre los que se mueven los valores B_i y C_i. Llegaremos a un ciclo cuando tengamos un valor $C_i = 1$. Una vez se ha alcanzado ese valor, si se siguiera el proceso volveríamos a obtener la misma secuencia de valores C_i. Si en la búsqueda de relaciones llegamos al caso de cerrar un ciclo, podremos seguir buscando más relaciones nuevas a partir de un nuevo valor de k

[Morr75]. Henri COHEN indica en [Cohe93] que el hecho de que toda secuencia de BHÁSCARA sea periódica resulta completamente irrelevante dado que, excepto para números muy especiales, nunca llegaremos a necesitar computar todos los valores del periodo.

3. Todas las congruencias realizadas en las búsquedas de las relaciones se hacen modulo N, y en ningún caso módulo $k \cdot N$ [Morr75].

6.1.2. Búsqueda de los elementos que forman el conjunto S

Encontrar relaciones o pares $P-C$ es tarea sencilla, como se ve. La dificultad del proceso empieza ahora que podemos tener almacenadas un montón de ellas: ¿Cómo entresacar, de todo el vasto conjunto de relaciones, estos subconjuntos S de pares $P-C$ que verifiquen que el producto de todos sus correspondientes C_i's resulte un cuadrado perfecto? ¿Cómo saber siquiera si existe uno solo de esos subconjuntos?

Por suerte, conocemos un método que da respuesta a estas dos cuestiones. Pero una respuesta a un alto precio.

Existe una forma de detectar subconjuntos S dentro de la enormidad de posibles pares $P-C$ candidatos. Para ello debemos exigir una fuerte restricción a los posibles candidatos a formar parte de esos subconjuntos: únicamente podrán formar parte de un subconjunto S aquellos pares $P-C$ para los que logremos conocer todos los factores de su correspondiente C_i. Y no solo eso, sino que además esos factores primos de los C_i deberán ser "suficientemente pequeños" (queda pendiente definir de forma precisa ese "suficientemente pequeños"). Por lo tanto, cualquier par $P-C$ que no logre cumplir esta condición será desechado.

Grave restricción, pero necesaria si queremos hallar combinaciones de pares $P-C$ que nos conduzcan al final de nuestro proceso.

Veamos ahora cómo se desarrolla la selección de pares.

6.1.2.1. Factorización de los distintos C_i's.

Como ya hemos dicho, al menos inicialmente desestimaremos todos los pares $P-C$ que verifiquen que su correspondiente C_i tiene un factor primo mayor que el límite prefijado.

La cuestión es saber cuándo un factor de C_i será "demasiado" grande para que no consideremos válido el par $P-C$ correspondiente. Para determinar esto lo que se hace es trabajar con un conjunto de primos todos ellos menores que un valor predeterminado. Al conjunto de estos primos le llamamos **base de factores**.

El procedimiento a seguir para determinar si un par $P-C$ puede ser considerado candidato a formar parte de un subconjunto S será intentar factorizar por completo el correspondiente valor

C_i con los primos de esa base de factores. Si se logra, entonces tenemos que ese par tiene todos los factores primos de C_i "suficientemente pequeños".

Los primos de la base de factores, deben cumplir dos propiedades:

1. todos ellos son menores que un límite superior dado B;
2. según queda demostrado en [Morr75], si un primo impar p divide a un valor C_i, $i \geq 1$, entonces el valor del símbolo de LEGENDRE $(k \cdot N/p)$ ó es 0 ó es 1.

Por tanto, la base de factores se escoge entre todos aquellos primos impares menores que el límite dado que verifiquen que $(k \cdot N/p)$ vale 0 ó 1; el primo 2 siempre forma parte de la base de factores. Cualquier primo que no verifique esta condición no logrará dividir a ninguna de las expresiones C_i calculadas en la expansión por fracciones continuas de $\sqrt{k \cdot N}$: introducirlo en la base de factores no haría más que alargar de forma improductiva el proceso de factorización, pues sabemos seguro que ese primo no podrá dividir a ningún C_i que sea congruente con un valor P_i^2, cuadrado perfecto.

Conviene introducir ahora un nuevo concepto: decimos que un número entero es ***B*–suave** si todos sus factores primos son menores que B.

En nuestro proceso, por tanto, vamos generando, mediante el algoritmo de BHÁSCARA–BROUNCKER antes presentado, pares de relaciones $P - C$, que serán almacenados o desechados según se cumpla que el valor C_i correspondiente quede completamente factorizado o no por los primos de la base de factores, es decir, según se cumpla que C_i sea o no B–suave.

Llamamos **conjunto *F*** al conjunto de estos pares $P - C$ almacenados. Al **cardinal del conjunto *F*** (número de relaciones $P - C$, con C_i suave, halladas y almacenadas) lo llamamos f.

Una vez halla logrado "suficientes" pares $P - C$ con C_i B–suave (y más adelante deberemos concretar ese nuevo concepto de "suficientes") habrá que localizar subconjuntos S.

Llamemos r al **cardinal de la base de factores**. Todos los pares $P - C$ almacenados cumplen que los valores C_i tienen todos sus factores entre estos r primos, es decir,

$$C_i = \prod_{j=1}^{r} p_j^{l_j}$$

Donde l_j será igual a 0 si el primo p_j no divide a C_i, y será igual o mayor que 1 si divide a C_i una o más veces.

El nervio principal del método que se emplea para localizar subconjuntos S se basa en que encontrar un grupo de valores C_i's que sean cuadrado perfecto es lo mismo que encontrar un grupo de valores C_i's que verifiquen que la factorización de su producto sea tal que todos los primos que intervienen lo hagan un número par de veces. El proceso se centra por tanto en encontrar valores C_i tales que la suma de sus potencias l_j para cada p_j sea par o sea cero.

Este proceso se realiza mediante la técnica de eliminación gaussiana. En la presentación de este procedimiento queda concretado cuándo podemos decir que tenemos "suficientes" pares $P-C$ para decidirnos a iniciar la búsqueda con certeza de encontrar al menos un subconjunto S.

6.1.2.2. Proceso de eliminación gaussiana.

Para realizar el proceso con eficiencia, introducimos un vector binario que llamaremos vector de exponentes. Además, para llegar a buen fin en el procedimiento de eliminación gaussiana necesitaremos asociar, a cada vector exponente otro que llamaremos vector histórico.

Tomamos la base de factores de forma ordenada, $(p_1, p_2, ..., p_r)$. A cada elemento i–ésimo de F le asociamos su correspondiente **vector de exponentes** $e_i = (l_0, l_1, ..., l_r)$, donde

$$l_0 = \begin{cases} 1, & \text{si } i \text{ es impar.} \\ 0, & \text{en cualquier otro caso.} \end{cases} \qquad \text{(corresponde al factor } (-1)^i\text{)}$$

y, para $1 \le j \le r$,

$$l_j = \begin{cases} 1, & \text{si } p_j \text{ divide a } C_i \text{ un número impar de veces.} \\ 0, & \text{en cualquier otro caso.} \end{cases}$$

nótese que el bit de signo l_0 corresponde al signo $(-1)^i$

Para cada e_i definimos su **vector histórico** $h_i = (h_0, h_1, ..., h_r)$, donde

$$h_m = \begin{cases} 1, & \text{si } m = i \\ 0, & \text{en cualquier otro caso.} \end{cases}$$

Una vez definidos estos dos vectores para cada una de los pares $P-C$ hallados, procedemos, mediante la técnica de eliminación gaussiana sobre $\mathbf{Z}/2\mathbf{Z}$, a buscar cuáles serán los diferentes conjuntos S, subconjuntos de F útiles para encontrar los factores de N.

Al final, una vez hallados f pares $P-C$, tendremos f vectores e_i de dimensión $r+1$. Por álgebra lineal sabemos que si tenemos $r+1 < f$, entonces necesariamente nuestra colección de f vectores es linealmente dependiente: existe al menos una suma (módulo 2) de algunos de esos vectores que resulta el vector nulo. Que la suma de varios de estos vectores (que recogen en cada coordenada si la potencia del primo correspondiente es par o impar) es igual a cero implica que el producto de los valores C_i de las relaciones a las que corresponden esos determinados vectores es tal que todos los primos de la base de factores o no intervienen o intervienen un número par de veces.

Por lo tanto podemos asegurar que, si tenemos halladas más relaciones que el cardinal de la base de factores más el factor –1 (es decir, más de $(r+1)$ relaciones), entonces al menos encontraremos un subconjunto S. La experiencia muestra que, de hecho, se encuentran bastantes más que uno de esos subconjuntos. Queda así perfectamente concretada la cantidad, antes vagamente señalada ("suficientes") de pares $P-C$ que debemos hallar con la condición de que C_i sea B–suave.

La búsqueda de las posibles combinaciones lineales se puede hacer mediante el **procedimiento de eliminación gaussiana**.

Creamos una matriz, que llamamos **matriz de vectores** o **matriz de relaciones**, formada por los f vectores e_i. Tendremos una matriz de f filas y $r+1$ columnas. Para poder seguir la pista a las combinaciones de los valores de C_i cuyos productos resultan ser un cuadrado perfecto, asociamos a la matriz de vectores otra matriz, que llamamos **matriz histórica**, que es una matriz cuadrada (dimensión $(r+1)\times(r+1)$) y que hemos definido inicialmente como una matriz identidad.

Comenzamos el proceso con la primera columna más a la izquierda de la matriz. Si encontramos alguna fila que contenga un 1 en esa primera columna, entonces sumamos (suma módulo 2) esa fila (que llamamos, por ejemplo fila j) con todas las demás filas que tengan también un 1 en esa misma columna. Eliminamos finalmente esa fila j donde hemos encontrado el 1 y que ha servido para eliminar los 1's de esa columna en todas las demás filas que lo tuvieran. Las restantes r filas tienen ahora un 0 en la primera columna.

El proceso de suma módulo 2 también se realiza con la matriz histórica, en las mismas filas que lo hemos realizado con la matriz de vectores. Ahora en la matriz histórica hemos eliminado una fila (la misma que en la matriz de vectores: la fila j) y tenemos un 1 en la columna j de todas las filas donde antes teníamos un 1 en la primera columna de la matriz de vectores.

Repetimos el proceso, en busca de la primera fila que tenga un 1, en la segunda columna más a la izquierda (por ejemplo, la fila k) y sumándola módulo 2 a todas las demás que hallemos un 1 en esa segunda columna, y eliminando finalmente esa fila que hemos sumado a todas las demás. También habremos realizado el proceso de sumas en las filas correspondientes de la matriz identidad, y tendremos un 1 en la columna k de todas las filas donde antes teníamos un 1 en la segunda columna más a la izquierda de la matriz de vectores. Eliminamos también la fila k de la matriz identidad.

Una vez hayamos realizado el proceso para todas las columnas, tendremos al menos $f-(r+1)$ ($r+1<f$) filas no eliminadas con todos los valores en todas sus columnas igual a cero. Y tendremos "grabado" en la matriz histórica cuáles han sido las relaciones responsables de llegar a una combinación cuya suma módulo 2 haya dejado a cero cada una de esas filas.

6.1.2.3. Factores primos grandes y otras optimizaciones.

La exigencia de que C_i sea B–suave es enormemente restrictiva. Cabría estudiar una vía para "suavizar" la exigencia de la suavidad.

Supongamos que

$$C_i = \prod_{p_j \in Base} p_j^{l_j} \times R_i$$

es decir, supongamos que después de buscar factores de C_i entre los primos de la base de

factores, quedase un resto R_i, factor de C_i, que no fuese divisible por ningún primo de la base. Llamamos a R_i **factor grande**.

Supongamos que p_q es el **primo mayor de la base de factores**. Si el valor R_i verifica que $R_i < p_q^2$, entonces es inmediato deducir que R_i (que no lo divide ningún primo menor que su raíz cuadrada) es primo también.

Como veremos más adelante, si C_i es tal que su factor grande $R_i < p_q^2$, bien podemos sacarle partido a ese par $P-C$, y resulta por tanto de gran utilidad almacenar esa relación, con su correspondiente vector de exponentes e_i y el factor restante R_i.

El valor p_q^2 es el **límite superior**, a partir del cual un par $P-C$ deberá ser descartado. Algunos autores toman valores límite, para la aceptación o el rechazo del factor R_i, diferentes. Por ejemplo, en [Pome83] se aconseja tomar los valores notablemente mayores que p_q^2 excepto cuando exista un coste no trivial en el almacenamiento de los pares $P-C$ y sus respectivos factores. Nosotros hemos tomado el valor p_q^2.

Consideraremos pues todos aquellos pares $P-C$ que verifiquen que el valor C_i queda completamente factorizado sobre la base de factores, o que su factor grande es menor que el límite superior. Evidentemente, si $R_i = 1$ tenemos que C_i es B–suave.

Si ocurre que C_i y C_k tienen el mismo factor grande ($R_i = R_k$), ambos menores que el límite superior (y por tanto, ambos factores son primos), entonces $C_i \times C_k$ será una relación que podrá ser considerada en nuestra matriz de relaciones, dado que el factor primo largo actúa como un cuadrado perfecto.

Este modo de trabajar ofrece una gran ventaja y es que podemos usar una base de factores de cardinal menor, obteniendo resultados muy parecidos.

El procedimiento a seguir con estos factores grandes viene bien determinado en [Pome83]:

Si C_i queda factorizado con la base de factores más el factor grande R_i menor que el límite superior marcado, entonces almacenamos, además del par $P-C$, el vector de exponentes e_i y el valor del factor grande R_i. El vector e_i recoge la factorización de C_i/R_i.

Para buscar las relaciones almacenadas que verifiquen que tienen el mismo factor grande procedemos de la siguiente manera: Las filas de las matrices de relaciones e histórica son primeramente ordenadas según el tamaño del factor grande R_i. Así se puede realizar una pasada a la lista eliminando todos los factores grandes: Si R_i no viene repetido, entonces la relación $P-C$ queda rechazada y eliminamos los correspondientes valores e_i, R_i. Si los pares

$$e_{ij}, R_{ij}; e_{ij+1}, R_{ij+1}; ...; e_{ij+k}, R_{ij+k}$$

tienen todos el mismo factor grande

$$R_{ij} = R_{ij+1} = ... = R_{ij+k}$$

entonces eliminamos el par

$$e_{ij}, R_{ij}$$

y el resto quedan reemplazados por los vectores

$$e_{ij} + e_{ij+1}; ...; e_{ij} + e_{ij+k}$$

(reducidos módulo 2, por supuesto). Así, con un pequeño coste de tiempo y de espacio, quedan eliminados todos los factores grandes, dejando a cambio relaciones válidas para el proceso de eliminación gaussiana.

El artículo [Pome83] presenta la estrategia llamada "Early Abort Strategy", que consiste en abandonar el proceso de factorización si, probados un número determinado de primos de la base, el tamaño del resto a factorizar (del valor de C_i) es superior a un límite previamente determinado. Esta estrategia puede establecerse repetidas veces en el proceso de cada factorización. Lo propone [Morr75]: dado que la factorización de cada valor de C_i ocupa más del 90 % del tiempo de cálculo para la mayoría de los números a factorizar, podemos obtener un significativo incremento de velocidad si descartamos todos aquellos C_i's que permanezcan mayores que un tamaño prefijado después de un cierto número de divisiones con los primos de la base de factores. Nosotros hemos hecho una implementación muy sencilla de esta estrategia, y hemos desestimado aquellos pares $P - C$ tales que su valor C_i fuese inicialmente mayor que un límite prefijado. A este valor lo hemos llamado **valor de aceptación**.

Arjen K. LENSTRA y Mark S. MANASSE [Lens90] mencionan la paradoja del cumpleaños para justificar que un moderado número de relaciones con factor grande mayor que 1 ofrece bastantes posibilidades de obtener relaciones válidas para el proceso de eliminación gaussiana. La **paradoja del cumpleaños**, para el caso que nos ocupa, afirma que si escogemos j enteros de forma aleatoria, tomados del conjunto de enteros menores que un límite prefijado L, y si $j > \sqrt{L}$, entonces tenemos una probabilidad mayor que $1/2$ de que dos de los enteros escogidos al azar sean el mismo. Si tenemos en cuenta que:

1. los valores de los factores grandes son, de hecho, cantidades nunca mayores que el cuadrado del último de los factores de la base, y que
2. entre los factores grandes sólo encontramos enteros primos,

a partir de una cantidad de relaciones con factor grande del orden del cardinal de la base de factores, es muy probable encontrar algunas de ellas coincidentes en el valor del factor grande. De hecho, así ocurre.

6.1.2.4. Consideraciones sobre el valor de k en los diferentes valores de $k \cdot N$ a estudiar.

En [Pome83] se recoge que a menudo es más conveniente trabajar con la expansión en fracciones continuas de $\sqrt{k \cdot N}$ para pequeños valores de k libres de cuadrados perfectos que con la expansión de $\sqrt{N}$. De hecho cabe pensar que el proceso irá más rápido si ocurre que

encontramos valores de k tales que su base de factores para su correspondiente $k \cdot N$ tenga, de promedio, primos menores.

6.1.3. Búsqueda de factores de N.

Cuando disponemos de uno o varios conjuntos S, tan solo nos queda la tarea de multiplicar los valores de los pares $P - C$, en búsqueda, como antes hemos dicho, de una expresión de la forma de la expresión (1). El producto de todos los valores C_i de cada conjunto S forma un valor que es cuadrado perfecto. N queda entonces expresado como diferencia de cuadrados, tal y como se ve en la expresión (2), y aplicando el algoritmo de EUCLIDES podemos llegar a encontrar los factores primos de N.

El autor de [Wund79] señala como dato experimental que habitualmente son necesarias cinco o seis relaciones halladas de la forma

$$P^2 \equiv \prod_i P_{i-1}^2 \equiv \prod_i (-1)^i \cdot C_i = C^2 \pmod N$$

para lograr la factorización del número. Así, efectivamente, lo hemos verificado nosotros.

6.2. IMPLEMENTACIÓN DEL ALGORITMO CFRAC.

Hemos desarrollado una implementación del algoritmo de factorización mediante la técnica de las fracciones continuas presentado en el epígrafe anterior. Los pasos más generales en la implementación del algoritmo, y las funciones implicadas en cada momento, son las siguientes:

1. Asignar valores a los parámetros del proceso: número de bits del entero a factorizar; cardinal de la base de factores; dirección de memoria donde se creará el vector para la base de factores (vector que contiene tantos primos como señala el cardinal de la base, comenzando por el 2 y sucesivamente todos los siguientes que verifiquen que el símbolo de LEGENDRE del número N a factorizar respecto a esos primos es igual a +1); número de relaciones completamente factorizadas necesarias (que tomamos como una más del cardinal de la base de factores); número de relaciones que tengan factor grande a buscar; máximo número de bits que podrá tener una relación C_i para que pueda ser tomada en consideración para su estudio de suavidad (si el valor C_i del par $P - C$ obtenido tiene un número de bits mayor que el señalado por ese parámetro, entonces esa relación es directamente desestimada) y que define el valor de aceptación; y límite superior para el factor grande: valor que será igual al cuadrado del mayor de los primos de la base de factores.

2. Iniciar las estructuras creadas para el almacenamiento de las relaciones. Una relación queda completamente definida cuando tenemos el valor de P_i, el valor de C_i, el valor de R_i y el valor del vector de exponentes e_i que recoge los primos que intervienen un número impar de

veces en el proceso de factorización de C_i con los primos de la base de factores. Esos son algunos de los campos que contiene la estructura que hemos definido para almacenar las relaciones que nos interesan para el posterior proceso de eliminación gaussiana. Además, la estructura tiene la dirección de la matriz histórica, un vector de validación de relaciones y un índice de ordenación de relaciones.

3. Iniciar otra estructura que hemos definido para la gestión y creación de nuevos valores de C_i y de P_i mediante el algoritmo de BHÁSCARA.

4. Inicializar contadores. El programa tiene creados cuatro contadores: uno para contar el número de relaciones con factor grande igual a 1 halladas hasta el momento (relaciones suaves); otro para contar el número de relaciones con factor grande mayor que 1 pero menor que el límite máximo, y por tanto válidas en primera instancia; un tercer contador para contabilizar el número de relaciones sobre las que se ha estudiado la suavidad y se han desechado por quedar un factor grande mayor que el límite superior prefijado; un cuarto contador sirve para contabilizar el número de relaciones que se han descartado directamente por ser su valor C_i mayor que el valor de aceptación.

5. Proceso de búsqueda de relaciones. Esta parte del algoritmo es la más larga, y se repite una y otra vez, en busca de relaciones válidas, hasta lograr tener tantas como indique el cardinal de la base de factores más uno. Los pasos son los siguientes:

   5.1. Buscar una nueva relación $P - C$.

   5.2. Estudio de la relación.

      5.2.1. Si el valor de C_i es idénticamente igual a 1, entonces hemos cerrado un ciclo y los valores van a repetirse nuevamente. No hay que seguir. Se abandona el proceso y se continúa buscando nuevas relaciones con un nuevo valor $k \cdot N$. Antes hay que volver al paso 1 para redefinir otra nueva base de factores.

      5.2.2. Verificar si el valor de N es un cuadrado perfecto. Eso se detecta comprobando si el valor de C_i se hace cero. Esta verificación sólo la hacemos para el primer valor C_1.

      5.2.3. Comprobar si el número de bits de C_i es menor que el límite superior señalado (es decir, si C_i es menor que el valor de aceptación). Si no lo es, aumentar en uno el contador de relaciones descartadas por ser C_i demasiado grande y volver a empezar en el punto 5.1.

      5.2.4. Intentar factorizar el valor C_i bajo los primos de la base de factores. Para este proceso hemos definido una función que devuelve el vector de exponentes e_i con el valor a 1 en aquellas posiciones que se refieren a los primos que han intervenido un número impar de veces en la factorización de C_i. Además devuelve el valor

que resta a C_i (R_i) después de haber sido dividido por todos los primos de la base de factores que lo factorizan.

5.2.5. Comparar el valor de R_i con el límite superior (que, como ya hemos señalado antes, lo hemos definido como el cuadrado del primo mayor de la base de factores).

5.2.5.1. Si es mayor, entonces la relación queda descartada y se incrementa en uno el contador de relaciones rechazadas por ese motivo.

5.2.5.2. Si es menor que ese límite y mayor que 1, entonces se almacena la relación y se incrementa en uno el contador de relaciones de factor grande almacenadas. La validez de esa relación se pone a valor 2.

5.2.5.3. Si R_i es igual a 1 entonces se almacena la relación y se incrementa en uno el contador de relaciones suaves. La validez de esa relación se pone a valor 1.

5.3. El objetivo de todo el proceso es buscar relaciones cuyo factor grande sea igual a 1 o, si es mayor que 1, que tengamos otras con el mismo valor de factor grande de forma que podamos, de varias de ese último tipo, obtener algunas válidas para el proceso final de eliminación gaussiana. Por eso, durante el proceso de búsqueda de relaciones, hacemos un alto de vez en cuando para determinar de cuántas de las relaciones de factor grande halladas podemos sacar relaciones válidas para el proceso posterior de eliminación gaussiana. Cada determinado número de relaciones con factor grande encontradas se realiza el siguiente proceso:

5.3.1. Ordenar las relaciones por orden de valor del factor grande de menor a mayor.

5.3.2. Realizar una búsqueda de factores grandes iguales y, entonces, anular una de las relaciones (poniendo el valor de la validez de la relación a 0) y sumar (suma módulo 2) los vectores de exponentes y el histórico de la relación anulada con todas las demás (que quedarán con su validez a 1) que tengan el mismo factor grande.

5.3.3. Una vez hecho el proceso volvemos a contabilizar el número de relaciones que tenemos completamente validadas (validez igual a 1) y aquellas que siguen siendo de validez igual a 2.

6. Una vez halladas todas las relaciones con validez igual a 1 necesarias (al menos una relación más que el cardinal de la base de factores) podemos liberar parte de la memoria reservada: aquella reservada para la creación de algunas estructuras que han sido necesarias mientras buscábamos los pares $P - C$.

7. Volvemos a ordenar las relaciones y a provocar una eliminación de relaciones como la señalada en los puntos 5.3.1., 5.3.2. y 5.3.3.

8. Realizamos en proceso de eliminación gaussiana.

9. Una vez hecha la eliminación gaussiana buscamos las relaciones que han quedado válidas para el proceso de factorización: aquellas en las que su vector de exponentes (que es una fila de la matriz de relaciones o matriz de vectores) e_i halla quedado con todos sus valores a cero. Hemos implementado el proceso de forma que todas las relaciones que no tengan su vector de exponentes nulo se les asigna la validez 0.

10. Contar el número de relaciones válidas disponibles.

11. Ordenar las relaciones válidas según el número de unos que tenga la correspondiente fila de la matriz histórica. Cuantos menos unos tenga la fila, menos valores de P_i y de C_i deberemos manejar: eso supone menos cálculos.

12. El proceso descrito en este punto se deberá repetir hasta lograr tomar una relación válida que ofrezca un resultado no trivial en la factorización de N.

    12.1. Recoger la siguiente relación no descartada y no estudiada todavía en este bloque.

    12.2. Calcular los valores que ofrece el producto de relaciones para

$$X = \prod P_i$$

$$Y = \sqrt{\prod C_i}$$

    12.3. Averiguar si los valores X, Y ofrecen, mediante el algoritmo de EUCLIDES una solución no trivial a la factorización de N. Si lo hacen, entonces el proceso ha terminado; en caso contrario, se vuelve a comenzar el proceso 12 anulando la relación actual y buscando de nuevo la relación indicada en el punto 12.1.

13. TERMINAR: mostrar resultados y liberar la memoria de todas las matrices y estructuras.

6.2.1. Parámetros del proceso.

Los parámetros de la factorización son los siguientes:

⇒ Número de bits del entero N a factorizar.

⇒ Cardinal de la base de factores. Es el número de primos que tendrá la base de factores: los primeros primos del conjunto de los enteros que verifican que el símbolo de JACOBI de N con respecto a cada uno de esos primos es igual a 1, e incluido el número 2 como elemento siempre presente en cualquier base. Los valores de C_i que serán almacenados son aquellos que quedan factorizados (o con un factor restante no mayor al cuadrado del primo más grande) en la base de factores. El vector de exponentes e_i señala (un 1 en dicha posición) qué primos intervienen de un modo impar en la factorización del valor C_i. Cada primo de la base de factores tiene una posición relativa predeterminada en el vector e_i, que es un bit dentro de una cadena de elementos `UINT2` (posteriormente, en el Capítulo 7, justificaremos el

cambio a cadena de elementos tipo `UINT4`).

Pensamos que vale la pena una breve justificación de por qué la base de factores debe estar formada únicamente por aquellos primos p_i tales que $(N/p_i)=+1$. El motivo de tal condición se encuentra primeramente en una expresión que relaciona los valores de las secuencias generadas con el Algoritmo 1, y que viene recogida en casi todos los manuales que presentan el algoritmo de las fracciones continuas: por ejemplo, expresión (10.7) de [Bres89], o la expresión (A5.31) de [Ries87]. Esta expresión, que copiamos aquí, es la siguiente:

$$P_i^2 - N \cdot Q_i^2 = (-1)^i \cdot C_i \qquad (5)$$

Si un primo p_i divide al valor C_i de la relación $P-C$ que en este momento estamos estudiando, tendremos que se cumple (aplicando a ambos miembros de la expresión 5 el operador $\bmod p_i$): $P_i^2 - N \cdot Q_i^2 \equiv 0 (\bmod p_i)$ y, por tanto, $N \equiv (P_i/Q_i)^2$ es un cuadrado $\bmod p_i$, o lo que es lo mismo, $(N/p_i)=+1$.

Es decir, podemos afirmar que si p_i divide a C_i, entonces también se verifica que $(N/p_i)=+1$. Y al revés, si no se verifica que $(N/p_i)=+1$, entonces nunca ocurrirá que el primo p_i divida a ningún valor C_i que verifique la expresión (5). Y todos los valores C_i que obtenemos la verifican.

Queda ahora aclarar por qué el primo 2 pertenece a la base de factores de forma incondicional. Para dar respuesta a esta cuestión basta la siguiente observación:

Las definiciones de símbolos de LEGENDRE y de JACOBI están hechas únicamente para primos o para enteros impares. No se considera el caso del símbolo de LEGENDRE para el primo 2. No se ha definido el valor de este símbolo para $(N/2)$. ¿Por qué? Sencillamente porque solo caben dos posibilidades analizando una expresión $x^2 \equiv a (\bmod 2)$:

1. Si x es par (y por tanto x^2 es par), entonces a es congruente con 0 módulo 2, y tenemos que $(a/2)=0$.
2. Si x es impar (y por tanto x^2 es impar), entonces a es congruente con 1 módulo 2, y tenemos que $(a/2)=(1/2)=+1$.

Es decir, que si no existen definiciones de los símbolos para el primo 2 es porque la expresión $x^2 \equiv a (\bmod 2)$ tiene solución siempre y, por tanto, ó $(a/2)=0$ ó $(a/2)=+1$.

⇒ Número de relaciones, completamente factorizadas, a buscar. Serán aquellas que son B–suaves o aquellas con factor grande para las que hemos encontrado otra relación con factor grande coincidente. Por defecto, toma el valor del cardinal de la base de factores más uno.

⇒ Número de relaciones, que tengan factor grande, a buscar.

La matriz que crearemos para almacenar las relaciones será del tamaño indicado por la suma de relaciones con el valor de C_i completamente factorizados ($R_i=1$) más la suma de las

relaciones de factor grande (para C_i) mayor que uno ($R_i > 1$) que queremos buscar.

⇒ Máximo número de bits que podrá tener una relación C_i para que pueda ser tomada en consideración para el estudio de su suavidad. Si la relación C_i obtenida tiene número de bits mayor que el señalado en ese parámetro, entonces la relación será desestimada. Este valor define el valor de aceptación.

⇒ La dirección de memoria de la base de factores.

⇒ Límite superior para el factor grande R_i. Como ya hemos señalado, este valor será igual al cuadrado del primo mayor de la base de factores.

Todos estos parámetros (y otros auxiliares necesarios para la implementación) han quedado agrupados en una estructura que hemos definido y que hemos llamado `PARAMETROS`, y que tiene la siguiente forma:

```
typedef struct
{
      UINT4 bits1;
      UINT4 bits2;
      UINT2 tamN;
      UINT4 cfb;
      UINT2 cfb_b;
      UINT4 nrn;
      UINT4 nrfg;
      UINT4 nrfg_c;
      UINT4 UBR;
      BASE *base;
      NUMERO UBF;
}PARAMETROS;
```

Además quedan definidas dos funciones para el manejo de los parámetros:

```
void Parametros(PARAMETROS*);
```

```
void Param(PARAMETROS*);
```

Ambas funciones han sido definidas para que el usuario pueda modificar algunos valores de los parámetros generales del proceso de factorización. Además estas funciones calculan todos los valores de parámetros relacionados con los aquí recogidos.

6.2.2. Generación de la base de factores.

La base de factores la forman aquellos primeros primos del conjunto de los enteros que verifican que el símbolo de JACOBI del número N a factorizar con respecto a cada uno de ellos es igual a 1. En principio bastará almacenar en un vector de `UINT4` aquellos valores primos que verifiquen esa propiedad del símbolo.

Como ya hemos visto, el proceso de factorización se desarrolla tomando como módulo para la búsqueda de relaciones de BHASCARA el valor $k \cdot N$. Esto permite que el algoritmo pudiera

ejecutarse en paralelo, agilizando el proceso de búsqueda de relaciones en un cluster de PC, por ejemplo, dando a cada PC un valor de k. La optimización del código de la implementación del algoritmo, hecha para el caso en que se trabaje en una sola máquina, sería válida para el proceso ejecutado en paralelo con varias máquinas de características semejantes a la empleada en el proceso de optimización.

La dificultad de este modo de proceder es que los primos que tienen su símbolo a 1 para el módulo $k \cdot N$ no son necesariamente los mismos que lo tienen para $k' \cdot N$ (donde $k \neq k'$). Y cuando se almacena la factorización de una relación en el correspondiente vector de exponentes e_i para luego centralizar todos esos vectores en una única matriz de relaciones y en un único ordenador y realizar el proceso de eliminación gaussiana... ¿cómo proceder para que las distintas relaciones, buscadas con valores diferentes de k en los diversos ordenadores, mantengan una coherencia en las posiciones de los vectores e_i hallados, cada uno con una valor distinto de k? Es decir, ¿Cómo hacer para que distintas bases de factores relacionen al final los mismos primos?

Para lograr esta coherencia hemos definido una estructura para la base de factores que es como sigue:

```
typedef struct
{
      UINT4 factor;
      SINT2 simbolo;
}BASE;
```

Definimos la base de factores como un vector de elementos tipo `BASE`. En el campo `factor` de cada elemento del vector vamos colocando los sucesivos primeros primos. En el campo `simbolo` colocamos el valor del símbolo para cada valor $k \cdot N$ en el que trabajamos con respecto al primo recogido en `factor`.

La generación de la base de factores quedará, por tanto, diversa según la realicemos para un proceso donde trabajan varios ordenadores en paralelo o realicemos el proceso con una sola máquina.

En el primer caso tomaríamos como base de factores los primeros primos indicados por el cardinal de la base de factores y en cada ordenador se calcularía y almacenaría el valor del símbolo en el miembro de la estructura creado para tal fin. La función de suavidad debería entonces tener en cuenta este valor del campo `símbolo` y realizar el intento de división únicamente con aquellos primos que tuvieran el símbolo a +1. Aunque no hemos paralelizado el proceso, hemos querido diseñar la estructura de base de factores de forma que el salto de uno a varios ordenadores se pudiera hacer sin modificación del código básico de la factorización. Código que es el que hemos optimizado, como presentaremos en el siguiente Capítulo.

En el segundo caso (cuando trabajamos con un solo ordenador: ese será finalmente el proceso que estudiamos en esta tesis) guardamos en el vector únicamente aquellos primos que tengan el símbolo a +1, y la función que determina la suavidad trabajará con todos los primos de la base. En

ese caso, como solo se trabaja en un ordenador, no hace falta tener en cuenta el valor del campo `símbolo`: todos los primos de la variable tipo `BASE` son aptos para el proceso de factorización.

Cuando tenemos determinada la base de factores ya podemos definir el valor del límite superior para aceptar finalmente una relación C_i según que el factor grande R_i que queda después de intentar dividir por todos los primos de la base de factores sea menor que ese límite. El límite superior para los valores R_i aceptados será, como ya hemos indicado, el cuadrado del último de los primos de la base de factores.

El vector creado para la base de factores contiene un elemento más que el indicado por el cardinal de la base de factores. En ese último elemento colocamos el valor cero. El vector de la base de factores se recorre desde el inicio hasta llegar a un elemento cuyo valor es igual a cero.

Hemos definido una función para todo el proceso de la generación de la base de factores. Su prototipo es

```
void BaseGeneral_Ser(NUMERO*,PARAMETROS*);
```

y la tarea que realiza es la de guardar en un vector de elementos tipo `BASE`, en el campo `factor`, aquellos primos que verifiquen que el símbolo de JACOBI de N respecto a ellos es igual a +1. Cuando se trabaja en un solo ordenador no es necesario dar valores al campo `símbolo`.

6.2.3. Inicializar las estructuras de almacenamiento de relaciones.

El tercer paso para el proceso de factorización es reservar la memoria necesaria para el almacenamiento de las relaciones.

Para el mejor manejo de los valores a calcular, procesar y, en su caso, almacenar, hemos definido las dos siguientes estructuras:

```
typedef struct
{
      UINT2** M_Rel;
      NUMERO* V_Pi;
      NUMERO* V_Ci;
      NUMERO* V_Fact;
      UINT2** M_Id;
      UINT4 *ind;
      UINT2 *validez;
}RELACIONES;
typedef struct
{
      UINT2* Rel;
      NUMERO Pi;
      NUMERO Ci;
      NUMERO Factor;
}REL;
```

La primera de las dos estructuras (`RELACIONES`) ha sido creada para el almacenamiento de todas

las relaciones que sean válidas: tanto las suaves como las de factor grande menor que el límite superior. Los campos que contiene son:

⇒ Un puntero doble de tipo `UINT2`, llamado `M_Rel`, para la matriz de vectores e_i que contienen la información sobre la factorización de cada uno de los valores C_i guardados. Los sucesivos vectores de exponentes e_i almacenan un 1 o un 0 en columnas de bits. Cada 16 primos de la base de factores forman una columna de elementos `UINT2` del vector. Reservaremos tantos elementos `UINT2` para cada vector como sean necesarios para "enfilar" tantos bits como indique el cardinal de la base de factores. La matriz tendrá tantas filas como indique la suma de relaciones normales más la suma de relaciones de factor grande a buscar. Como veremos en el Capítulo 7, finalmente esta matriz quedará formada con elementos de tipo `UINT4`, enteros de 32 bits.

⇒ Un puntero de tipo `NUMERO`, para el vector que deberá almacenar los sucesivos valores de P_i de cada nueva relación. Tendremos tantos elementos en el vector como filas tiene la matriz de relaciones.

⇒ Un puntero de tipo `NUMERO`, para el vector que deberá almacenar los sucesivos valores de C_i de cada nueva relación. Tendremos tantos elementos en el vector como filas tiene la matriz de relaciones.

⇒ Un puntero de tipo `NUMERO`, para el vector que deberá almacenar los sucesivos valores de los sucesivos factores grandes (R_i) después de haber dividido C_i por todos sus divisores contenidos en la base de factores. Tendremos tantos elementos en el vector como filas tiene la matriz de relaciones.

⇒ Un puntero doble de tipo `UINT2` para la matriz histórica que será una matriz con tantas columnas `UINT2` como sean necesarias para tener "enfilados" tantos bits como filas tenga la matriz de relaciones y tendrá tantas filas como filas tenga la matriz de relaciones. Es una matriz cuadrada y al principio debe ser inicializada como matriz identidad. Esta matriz histórica es la que, al final de todo el proceso descrito en la introducción de este Capítulo, determinará los distintos conjuntos S de que disponemos. Del mismo modo que ocurre con la matriz de relaciones, como veremos en el siguiente Capítulo, este puntero quedará finalmente del tipo `UINT4`.

⇒ Un puntero de tipo `UINT4` que direccionará un vector que será un índice para la ordenación de las relaciones en los procesos de búsqueda de factores grandes iguales y en la eliminación gaussiana. Tendremos tantos elementos en el vector como filas tiene la matriz de relaciones.

⇒ Un puntero de tipo `UINT2` para el vector que deberá indicar la validez de cada una de las relaciones almacenadas. Tendremos tantos elementos en el vector como filas tiene la matriz de relaciones. Para cada relación, el valor correspondiente a la posición del vector será cero si no almacena relación alguna, o si ha sido invalidada en el proceso de eliminación gaussiana,

o si corresponde a una relación de factor grande mayor que 1 y que ha sido utilizada para validar otras relaciones con el mismo factor grande; será uno si la relación que almacena corresponde a una con un valor C_i completamente factorizado en la base de factores (es decir: cuyo factor grande es igual a 1), o si la relación tiene un factor grande diferente de uno pero ha sido validada por suma módulo 2 con otra que tiene el mismo factor grande; y valdrá dos si la relación que almacena es de factor grande (R_i mayor que uno y menor que el límite superior) y no ha sido validada al haber hallado otra relación con el mismo factor grande.

La segunda estructura (`REL`) ha sido creada para el manejo de una sola relación. Es la variable que empleamos para asignar un nuevo valor de par $P-C$ y la que recoge los valores ceros y unos del vector de exponentes e_i y el factor grande. Sus campos son:

⇒ Un puntero de tipo `UINT2` que direcciona un vector con tantos elementos `UINT2` como sean necesarios para alinear tantos bits como sea el cardinal de la base de factores más uno. Será el encargado de almacenar el vector de exponentes e_i correspondiente al valor C_i que se esté en ese momento intentando factorizar. Después de los procesos de optimización presentados en el Capítulo 7, quedará del tipo puntero a `UINT4`.

⇒ Dos variables de tipo `NUMERO` para almacenar los valores de P_i y de C_i.

⇒ Una variable de tipo `NUMERO` para almacenar el valor del factor grande R_i.

Como puede verse, los cuatro campos de la estructura `REL` tienen el formato de una fila de la estructura `RELACIONES`: el vector `Rel` es una fila de la matriz `M_Rel`; las variables `Pi`, `Ci` y `Factor` son elementos del vector `V_Pi`, `V_Ci` y `V_Fact` respectivamente. Los otros tres campos de la estructura `RELACIONES` están definidos para el seguimiento de los procesos necesarios para la eliminación de factores grandes y para el proceso de eliminación gaussiana.

Cuando se crea una variable de tipo `RELACIONES` o de tipo `REL` queda pendiente el trabajo de reservar todos los espacios de memoria de todas las matrices y de todos los vectores que serán direccionados por los punteros. No es una tarea trivial, y depende en cada caso de los valores del cardinal de la base de factores y del número de relaciones a buscar. Han quedado definidas dos funciones que reciben como parámetros la variable de tipo `PARAMETRO` del proceso de factorización y la variable de tipo `RELACIONES` o `REL` que se quiere inicializar:

```
void InicializarRelaciones(RELACIONES*,PARAMETROS);

void InicializarRel(REL*,PARAMETROS);
```

Estas funciones crean los espacios de memoria necesarios para cada vector de enteros, de variables tipo `NUMERO` o para las matrices de relaciones e histórica. Y asignan a cada espacio reservado los valores iniciales cero.

La reserva de memoria para todos los campos de las estructuras se realiza mediante la función `malloc()`. La memoria que se debe reservar para cualquier proceso de factorización es grande,

aunque para el rango de enteros que nosotros trabajamos es perfectamente manejable. Por ejemplo, si queremos factorizar un entero de 100 bits (realmente pequeño y sencillo de factorizar) con una base de factores formada por 300 primos, entonces el número de relaciones de validez 1 deberá ser al menos 301; si tomamos como valor del parámetro que indica el número de relaciones de factor grande a buscar el valor 3699 (que no es un valor muy alto), el número total de relaciones a buscar será por tanto igual a 4000. Con estos valores, la creación de la matriz de relaciones ocuparía 149 KBytes y la matriz identidad 1,9 MBytes. Otros valores de ejemplo: para un entero de 150 bits, si empleamos una base de factores de cardinal 500, entonces el valor del número de relaciones normales a buscar es 501. Supongamos además que creamos un espacio para almacenar hasta un total de 7500 relaciones con factor grande que pueda ser mayor que 1. En ese caso, la matriz de relaciones o matriz de vectores ocupará 500 KBytes, y la matriz histórica un total de 7,65 MBytes.

Como ya hemos señalado, el algoritmo de factorización CFRAC está recomendado para enteros no mayores de 50 dígitos decimales. Y este es precisamente el rango en el que hemos trabajado para las optimizaciones. Las cantidades de memoria que emplea el algoritmo no son excesivamente grandes. De todas formas sí son lo suficientemente extensas como para que superen los tamaños de las memorias cachés de los ordenadores que hemos utilizado. Eso nos ha llevado, como se verá en el siguiente Capítulo, a procurar un acceso a los valores de estas matrices racionalizado, de forma que queden minimizados los fallos a L1 y a L2. Pero eso es tema para el siguiente Capítulo: tan solo hacemos ahora la observación de que las funciones encargadas de operar con estas matrices van a ser objeto de amplias modificaciones para optimizar los accesos a memoria.

6.2.4. Búsqueda de un par $P - C$.

Para el manejo de todas las variables necesarias para el desarrollo del algoritmo de BHÁSCARA de expansión de fracciones continuas, hemos definido una estructura que hemos llamado BHASCARA. Este algoritmo es el que nos va proporcionando sucesivos valores de pares $P - C$. La única razón para esta definición de estructura, que ha quedado como una recopilación de variables, es la de otorgar comodidad en el traspaso de valores. Los valores iniciales de cada campo de la estructura BHASCARA se asignan mediante una función cuyo prototipo es:

```
void InicializarBhascara(BHASCARA*,NUMERO*);
```

que realiza los cálculos necesarios para tener todas los variables en su valor inicial concretado para el algoritmo.

Para la búsqueda de una nueva relación $P - C$, dividimos el proceso en dos pasos, cada uno de ellos descrito en una función.

El primero es poner a cero todos los elementos de la estructura REL. Esta estructura es la

encargada de almacenar el valor de la nueva relación y será la que luego recibirá los valores del vector de exponentes correspondiente al valor de C_i. Si la relación resulta al final válida para nuestros propósitos será almacenada en la variable de tipo `RELACIONES` que ya ha sido creada y que, como hemos visto, dispone de varias matrices y vectores. La función que realiza esta sencilla tarea tiene el siguiente prototipo:

```
void PonerACeroRel(REL*,PARAMETROS);
```

El segundo paso será generar una nueva relación o par $P-C$, siguiendo los pasos del algoritmo de BHÁSCARA. El algoritmo ha quedado implementado en una función cuyo prototipo es el siguiente:

```
void RelacionBhascara(BHASCARA*,REL*,NUMERO*);
```

En el Capítulo 7 mostraremos la forma de esta estructura y las modificaciones realizadas sobre ella y que nos han permitido reducir en mucho los tiempos de ejecución.

6.2.5. Estudio de la relación.

Una vez hemos comprobado que no se ha cerrado un ciclo (el valor de C_i no es igual a 1) y que C_i no tiene más bits que el límite fijado como límite superior, lo siguiente a realizar es el estudio sobre la suavidad del nuevo valor de C_i generado. Para este estudio ha quedado definida una función, de máximo peso en el proceso general de factorización, cuyo prototipo es:

```
void Suave_S(REL*,PARAMETROS);
```

y que toma la base de factores formada sólo por aquellos primeros primos que verifica directamente la propiedad de que el símbolo de JACOBI de N con cada uno de esos primos es igual a +1.

Desde el primero de los factores de la base definida (recuérdese que el primero es el factor 2) hasta encontrar un factor que sea igual a cero (así definimos la base de factores: un vector de enteros a cuyo último elemento le asignábamos el valor cero), vamos calculando el módulo de dividir el valor de C_i por cada uno de los sucesivos primos. Cuando encontramos un primo de la base que ofrece módulo cero, entonces hemos encontrado un divisor: realizamos el cociente entre el valor de C_i y ese primo y cambiamos el valor del vector de exponentes: lo ponemos a cero si estaba a uno; lo ponemos a uno si estaba a cero.

Si el primo j–ésimo de la base de factores no divide a C_i repetimos el cálculo del módulo con el siguiente primo. Si, en cambio, ese primo j–ésimo divide perfectamente a C_i, entonces, después de haber hecho el cociente y de haber cambiado el valor de la correspondiente posición del vector de exponentes, volvemos a repetir el intento de división con el mismo primo. Vamos modificando el valor de C_i cada vez que encontramos un primo de la base de factores que lo divide: el nuevo valor de C_i a estudiar es el cociente entre C_i y el primo que lo divide.

El proceso termina cuando se ha llegado al factor de la base de factores igual a cero o cuando se ha llegado a factorizar completamente el valor de C_i. La variable de tipo `REL` contiene ahora toda la información necesaria para decidir qué hacer con el par $P-C$ actual.

Si el valor que ha quedado por factorizar es mayor que el valor del limite superior, entonces la relación se desecha, se incrementa el contador de relaciones desechadas por ese motivo y se busca una nueva relación. En caso contrario la relación es válida y se almacena como relación con factor grande. Se hacen entonces los siguientes pasos:

1. Almacenar el valor de C_i inicialmente obtenido en la estructura `RELACIONES` en su campo correspondiente.
2. Almacenar el valor de P_i en la estructura `RELACIONES` en su campo correspondiente.
3. Almacenar el valor de C_i restante después de todo el proceso de estudio de la suavidad en la estructura `RELACIONES` en su campo correspondiente. Ese valor es el factor grande: si es igual a 1 entonces el valor C_i estudiado es suave y la validez de la relación se pone igual a 1 e incrementamos el contador de relaciones suaves halladas. Si es mayor que 1 ponemos la validez igual a 2 e incrementamos el contador de relaciones de factor grande halladas.
4. Almacenar el vector de exponentes e_i. Si el valor de i es impar, entonces ponemos a 1 el bit que recoge la información sobre el factor -1, tal y como ha quedado explicado unas páginas antes, en este Capítulo.

6.2.6. Obtención de relaciones válidas a partir de varias relaciones con factor grande.

Muchos de los pares $P-C$ almacenados no son válidos para un proceso posterior de eliminación gaussiana, destinado a la búsqueda de subconjuntos S de pares que verifiquen que el producto de todos sus C_i's forma un cuadrado perfecto: todas aquellos que tengan un factor grande distinto de 1. Como ya se ha explicado, si se encuentran dos pares $P-C$, ambos con el factor grande diferente de 1 pero iguales entre sí, podemos obtener un nuevo par $P-C$ que sí sea válido en el proceso de eliminación gaussiana.

Supongamos

$$C_{i1} = \prod_{p_j \in Base} p_j^{a_j} \times F$$

$$C_{i2} = \prod_{p_j \in Base} p_j^{b_j} \times F$$

El producto de los dos valores $C_{i1} \times C_{i2}$ queda:

$$C_{i1} \times C_{i2} = \prod_{p_j \in Base} p_j^{a_j + b_j} \times F^2$$

Creamos una nueva relación $P-C$ formada por un valor de P_i^2 que es producto de los dos correspondientes a las dos relaciones iniciales, y el valor de C_i que es producto de los dos correspondientes a las dos relaciones iniciales. Este par verifica que su valor C_i queda completamente factorizado en la base de factores excepto un factor restante que resulta ser cuadrado perfecto. Este nuevo valor de C_i, sí entra en consideración en cualquier proceso de eliminación gaussiana y arrastrará, además de todos los factores primos de la base que lo factorizan, otro factor que no estropeará el objetivo de que el producto de C_i's sea cuadrado perfecto, puesto que ese factor es ya un cuadrado perfecto.

Si encontramos varios (dos ó más) pares $P-C$, todos ellos con el mismo factor grande, entonces el proceso a seguir es el mismo: una relación se perderá para el proceso de eliminación gaussiana y toda las demás entran en consideración.

Para realizar la implementación de toda esta operación es muy conveniente primero ordenar la matriz de relaciones según el orden de los factores grandes. Para eso ha quedado definida una función que realiza esa ordenación y cuyo prototipo es:

```
void OrdenarRelaciones(RELACIONES*,PARAMETROS);
```

En una primera implementación para esta función usamos el método de la burbuja. Para esta ordenación quedó definido en la estructura `RELACIONES` un vector que se emplea como índice. Al principio le asignamos a los elementos de ese vector el valor de su posición. Y vamos ordenándolo luego según el criterio de menor a mayor del vector de los factores grandes. En trabajos posteriores será necesario optimizar este proceso usando otros medios de ordenación más veloces. Actualmente esta función ocupa más del 10 % del tiempo general del proceso de factorización. Como veremos en el Capítulo siguiente, dedicado a optimización de código, mediante otras técnicas de ordenación se reduce el tiempo de esta función a casi un tiempo nulo. Para esta parte del proceso hemos definido una función cuyo prototipo es:

```
void EliminarFactorGrande(RELACIONES*,PARAMETROS);
```

Una vez tenemos una ordenación que nos coloca los valores de los factores grandes ordenados de menor a mayor, resulta fácil encontrar aquellas relaciones con igual factor grande y, por tanto, podemos realizar el proceso de eliminación de relaciones de factor grande.

Si el factor es igual a cero, estamos ante un espacio de memoria donde aún no hemos almacenado relación alguna; esa fila no tiene ninguna información de interés: podemos pasar a la siguiente relación. Si el factor es igual a 1, la relación ya es perfectamente válida porque corresponde a un valor de C_i suave. Si el factor es distinto de uno, entonces a partir de la presente relación todas las siguientes serán relaciones de validez 1 ó 2, con un valor de factor grande mayor que 1 y menor que p_l^2.

El proceso descrito en este epígrafe se puede realizar en cualquier momento. Es conveniente realizarlo de tanto en tanto, para lograr incrementar el número de relaciones válidas para el

proceso de eliminación gaussiana. Cada vez que se realiza esta purga de relaciones, se logran depurar algunas, que quedan ya como relaciones de validez 1. Cuando se obtienen más relaciones de validez 1 que el valor del cardinal de la base de factores se puede dar por finalizada la búsqueda, pues ya hemos logrado suficientes relaciones como para obtener algunas combinaciones lineales en la matriz de relaciones. En la estructura `PARAMETROS` ya antes definida existe un campo cuyo valor señala la frecuencia con que se debe realizar este proceso. Para actualizar el número de relaciones aptas para el proceso de eliminación gaussiana, una vez ejecutada la función hay que hacer recuento de cada una de estos dos tipos de relaciones.

Las relaciones de factor grande que han sido utilizadas para validar a todas las demás han quedado con su validez igual a 2. Podrán ser utilizadas en procesos siguientes de eliminación gaussiana, una vez halladas nuevas relaciones, para seguir validando aquellas nuevas relaciones que vuelvan a tener el mismo factor grande que estas ya utilizadas.

6.2.7. Proceso de eliminación gaussiana.

El proceso de eliminación gaussiana ha quedado suficientemente explicado en la primera parte del presente Capítulo. Lo hemos dejado implementado en una función cuyo prototipo es

```
void EliminacionGaussiana(RELACIONES*,PARAMETROS);
```

Antes de ejecutarla se debe volver a ordenar la tabla de relaciones y volver a realizar un proceso último de eliminación de factores grandes.

Una vez terminado el proceso de eliminación gaussiana, el siguiente paso a realizar es la validación de las relaciones. Se trata de poner a cero la validez de todas aquellas relaciones en las que haya quedado un 1 en la matriz de vectores o matriz de relaciones después de la eliminación gaussiana. Esta operación la realizamos con la función

```
UINT2 ValidarRelaciones(RELACIONES*,PARAMETROS);
```

que devuelve el número de relaciones que han quedado útiles para continuar.

El último paso que realiza el algoritmo, antes de trabajar ya cada subconjunto S para buscar finalmente los factores no triviales del número N a factorizar, es ordenar las relaciones (que ahora, como hemos dicho, muestran la lista de valores C_i que determinan cada uno de los subconjuntos S) según el número de pares $P-C$ implicados: es decir, según el número de unos que hallemos en cada fila de la matriz histórica. Tener este orden sirve para comenzar el estudio siempre por aquellos subconjuntos S de menor cardinal lo que trae consigo menor cantidad de cálculo. La función que realiza esta última ordenación tiene como prototipo:

```
void OrdenFinal(RELACIONES*,PARAMETROS);
```

6.2.8. Trabajando con los conjuntos S. Búsqueda de factores.

El objetivo de todo el arduo proceso de búsqueda de pares $P-C$ con su valor C_i suave, y de subconjuntos de estos pares para las que se cumple que el producto de sus respectivos C_i's es un cuadrado perfecto, ha quedado felizmente cubierto. Ahora hay que tomar cada uno de estos subconjuntos y procurar obtener, de alguno de ellos, el valor de los dos factores no triviales del entero N que pretendemos factorizar.

Para la búsqueda de los factores primos de N quedan dos tareas pendientes:

1. Calcular los valores de los números X e Y a partir de una fila de la matriz histórica, es decir, como producto modular de los distintos elementos P_i y C_i de los pares $P-C$ implicados en un determinado subconjunto S .
2. Buscar los factores de N a partir de los valores calculados de X y de Y mediante el algoritmo de EUCLIDES de búsqueda del máximo común divisor entre N y $X+Y$ ó entre N y $X-Y$.

Para el cálculo de los productos de los valores C_i, el proceso es algo más largo y complejo. El producto de todos los C_i que forman parte de un subconjunto S resulta ser, por definición, un cuadrado perfecto módulo N. Para que cumplieran esa condición han sido buscados y se han determinado los diferentes subconjuntos S. El problema con que nos encontramos aquí es que primero habrá que calcular el producto de todos los valores C_i implicados en el conjunto S, y sólo al final podremos calcular la raíz cuadrada del número enorme hallado y obtener así finalmente el valor de Y calculando el resto de dividir la raíz cuadrada hallada con N.

En [Morr75] y también en [Ries87] encontramos un método que facilita todo este proceso de cálculo de la raíz cuadrada del producto de todos los valores C_i. La idea consiste en ir multiplicando sucesivamente los diferentes valores del vector C_i, pero antes de cada nuevo producto, chequear, mediante el algoritmo de EUCLIDES, si el nuevo valor C_i que vamos a multiplicar por todo el producto ya realizado de los anteriores tiene algún factor común.

El proceso de cálculo descrito y que se sigue es el recogido en el Algoritmo 2.

1. `aux <- 1` `Y <- 1`.
2. MIENTRAS queden `Ci`'s para multiplicar, HACER
   2.1. `mcd <-` máximo común divisor entre `aux y Ci`.
   2.2. `aux <- (aux / mcd) * (Ci / mcd)`.
   2.3. `Y <- Y * mcd`
3. `Y <- Y * Raíz Cuadrada(aux)`.
4. Reducir el resultado módulo `N`: `Y <- Y mod N`.

Algoritmo 2: Proceso para el cálculo de los valores producto de los diferentes Ci que forman parte de un determinado conjunto S.

Con este sistema queda reducido bastante el cálculo y el tamaño de la variable necesaria para llegar a obtener el valor de

$$Y = \sqrt{(\prod_{pi \in S} C_i) \bmod N}\ .$$

La función definida para el cálculo de los valores de X y de Y, tiene como prototipo:

```
void BuscarCuadrados(RELACIONES*,PARAMETROS,CUADRADOS*,NUMERO*);
```

donde el tipo de variable `CUADRADOS` queda definido como una estructura

```
typedef struct
{
      UINT4 f;
      NUMERO x;
      NUMERO y;
}CUADRADOS;
```

y que ha sido creada para comodidad en el manejo y paso de parámetros. El campo `x` y el campo `y` recogerán los valores X e Y calculados en un conjunto S dado. El campo `f` recoge la fila válida actual en el que estamos trabajando dentro de la matriz histórica.

La segunda parte del proceso consiste en buscar, mediante el algoritmo de EUCLIDES, los factores de N. Esta última función únicamente debe calcular los valores de $X+Y$ y de $X-Y$. Y a partir de estos dos valores calcular el máximo común divisor de cada uno de ellos con N. El prototipo de esta función es:

```
UINT2 BuscarFactores(CUADRADOS*,NUMERO*,NUMERO*,NUMERO*);
```

Donde los dos primeros parámetros de tipo `NUMERO*` son las variables donde se almacenan los valores de los factores; el tercer parámetro `NUMERO*` es el que recoge el valor de N. En la estructura `CUADRADOS*` han quedado almacenados los valores de X y de Y. La función devuelve un uno si los factores encontrados no son los triviales; devuelve un cero si son los triviales: en ese caso deberemos repetir el proceso de las funciones `BuscarCuadrados()` y `BuscarFactores()` con una nueva fila.

6.3. EJECUCIÓN DE LA APLICACIÓN. PRIMEROS RESULTADOS.

En el anexo IV, recogido al final de la tesis, presentamos los ordinogramas de las funciones principales de la aplicación que hemos implementado para la factorización de enteros, producto de dos primos. También queda recogido en ese anexo el ordinograma general de toda la aplicación.

En total hemos implementado 23 funciones, todas ellas para el uso específico en la implementación de esta aplicación. Estas 23 funciones suponen unas 2000 líneas de código. Además, se hace uso de otras 50 funciones, necesarias para las operaciones con enteros grandes, tests de primalidad y generación de valores compuestos para ser factorizados.

El régimen ordinario que hemos diseñado para el estudio de la aplicación implementada ha sido el siguiente: primero generamos un archivo con enteros de diferentes tamaños, que varían desde los 64 bits hasta los 160 bits: varios de cada uno de estos tamaños. La aplicación abre ese archivo y va tomando uno a uno los enteros y los factoriza, con diferentes cardinales para la base de factores y con diferentes cantidades de relaciones con factor grande a buscar. De cada intento de factorización se recogen una serie de datos que pueden resultar de interés: número de bits del entero a factorizar; cardinal de la base de factores; número de relaciones suaves mínimo a encontrar; número de relaciones de factor grande a buscar; número de relaciones suaves encontradas al final de proceso; número de relaciones de factor grande que han quedado inutilizadas; número de relaciones que se han tenido que tomar en consideración hasta encontrar una que ofrezca los factores no triviales del número a factorizar; tiempo empleado en el proceso de factorización; factores encontrados.

Este proceso se ha repetido muchas veces. En los más de dos años que llevamos trabajando sobre esta implementación, hemos factorizado más de trescientos mil enteros. Y la mayoría de ellos lo ha sido hecho repetidas veces, con diferentes parámetros. También hemos hecho una réplica de todas las funciones (las 23 y las 50 antes citadas), reconvertidas en macro, de forma que se evitan todas las llamadas a función. De todo este análisis presentamos en estas líneas algunas breves consideraciones. La primera es que la implementación ha sufrido cero errores en todo el tiempo que ha estado trabajando.

Dejamos para el siguiente Capítulo el estudio de los tiempos. En este epígrafe mostraremos algunos resultados que son interesantes para un buen ajuste de parámetros y un breve comentario sobre la mejora de rendimiento en la ejecución de la aplicación cuando se trabaja con macros.

6.3.1. Número medio de conjuntos S a probar para hallar finalmente los primos que factorizan al entero.

En el epígrafe 6.1.3. hemos comentado que, según el autor de [Wund79], habitualmente son necesarios una media de cinco o seis conjuntos S para lograr encontrar los factores del compuesto. Presentamos en este epígrafe las conclusiones que hemos sacado nosotros con nuestra implementación.

1. Tomada la mediana sobre 25.000 factorizaciones de enteros, desde 63 hasta 150 bits, el número de filas de la matriz histórica (ya ha quedado explicado que cada fila de la matriz histórica con validez igual a 1 define un conjunto S) que hemos tenido que analizar hasta llegar a encontrar los factores del entero es 4.
2. Este valor es el calculado sobre una larga lista. Hemos observado que en algunos números se necesita llegar a probar bastantes más filas que las señaladas en la mediana. Y como se

puede ver en el Cuadro 1, cuando los enteros a factorizar son pequeños es necesario trabajar con muchas más filas de la matriz histórica (con más conjuntos S) para llegar a obtener los primos. El cálculo de la mediana sobre todos estos valores es, efectivamente 4. Pero a partir de un determinado tamaño (enteros de más de 128 bits)podemos decir que la mediana oscila entre 2 y 3. No pensamos que sea tan alta como señalaba la referencia antes citada. El resultado obtenido es además más acorde con lo esperado teóricamente, cuando se afirma que la probabilidad de que un conjunto S otorgue un resultado satisfactorio es del 50%.

bits	filas	bits	filas	bits	filas	bits	filas
63	20	85	10	107	5	129	2,5
64	14	86	7	108	4	130	2
65	14	87	7	109	4	131	3
66	17	88	8	110	4	132	2
67	15	89	7	111	4	133	2
68	12	90	7	112	4	134	2
69	12	91	8	113	4	135	3
70	12	92	7	114	3	136	2
71	12	93	7	115	3	137	2
72	12	94	6	116	3	138	3
73	14	95	7	117	3	139	2,5
74	11	96	4	118	3,5	140	2
75	12	97	6	119	3	141	2
76	10	98	6	120	3	142	2
77	13	99	6	121	4	143	3
78	11	100	5	122	3	144	2
79	9	101	5	123	4	145	2
80	10	102	6	124	3	146	2
81	8	103	6	125	3	147	2
82	9	104	5	126	2	148	2
83	10	105	4	127	3	149	2
84	9	106	5	128	2	150	2

Cuadro 1: Número de filas de la matriz histórica (número de conjunto S necesarios a probar) en relación con el tamaño de los enteros a factorizar. La columna `bits` recoge el número de bits de los enteros a factorizar. La columna `filas` recoge el número de filas (mediana) a aprobar para cada tamaño.

3. En el Cuadro 2 recogemos otra información complementaria a la que hemos presentado en el Cuadro 1: los porcentajes, sacados del total de las factorizaciones realizadas, para cada uno de los valores del número de filas de la matriz histórica que han hecho falta para encontrar los factores del compuesto. Los más altos, y por orden de valor, son los dos intentos, los tres intentos, un solo intento, y cuatro intentos. A partir del valor 15 los porcentajes de intentos son

menores de 1%. Los valores máximos que hemos alcanzado en esta muestra de 25.000 factorizaciones son el 36, el 37,el 42 y el 45: cada uno de ellos ha sido alcanzado una sola vez.

conjuntos S probados	% de ocurrencias
1	12,74
2	17,36
3	15,10
4	11,37
5	8,83
6	6,48
7	5,54
8	4,44
9	3,28
10	2,52
11	2,09
12	1,77
13	1,66
14	1,23
15	1,00
mayor que 15	4,58

Cuadro 2: porcentaje de ocurrencias, para cada uno de los valores del número de filas de la matriz histórica que han hecho falta para encontrar los factores del compuesto.

4. Una última observación sobre el número de filas válidas de la matriz histórica obtenidas en cada proceso de eliminación gaussiana. Hemos querido obtener un valor que ofrezca información sobre el número de conjuntos S que podemos obtener en un proceso de eliminación gaussiana, en relación con el cardinal de la base de factores. Para hacer este cálculo hemos tenido en cuenta que no siempre hemos comenzado este proceso cuando ya habíamos logrado tantas relaciones como el valor que tiene el cardinal de la base de factores. Y a veces se inicia el proceso de eliminación gaussiana cuando se han obtenido bastantes más relaciones suaves que las estrictamente necesarias para obtener con certeza, al menos, un conjunto S.

   Hemos realizado los siguientes cálculos:

   a. La relación entre el número de conjuntos S hallados y el número de pares $P-C$ válidos para el proceso de eliminación gaussiana. Este valor no tiene por sí mismo ningún significado de interés: siempre se cumple que cuantas más relaciones se hayan encontrado, y más si esa cantidad es muy superior al cardinal de la base de factores, más conjuntos S vamos a poder construir; pero es un valor intermedio necesario, como

veremos.

b. La relación entre el número de pares $P-C$ hallados, válidos para el proceso de eliminación gaussiana y el cardinal de la base de factores. Si este valor es menor que 1 (nos ocurre en un 30% de los casos) entonces en esos casos hemos emprendido la tarea de la eliminación gaussiana antes de haber terminado la búsqueda de relaciones recomendada: una más que el cardinal de la base. Este valor es otro cálculo intermedio para el que buscamos ahora.

c. La relación entre los dos valores presentados en los apartados a y b. Esta relación recoge la siguiente expresión:

$$\frac{\text{número de conjuntos } S \text{ hallados} \ * \ \text{Cardinal de la base de factores}}{(\text{relaciones halladas})^2}$$

y que recoge el número de conjuntos S que se pueden hallar (de media) en relación con el número de relaciones halladas, cuando esta cantidad de relaciones se ajusta al valor del cardinal de la base de factores.

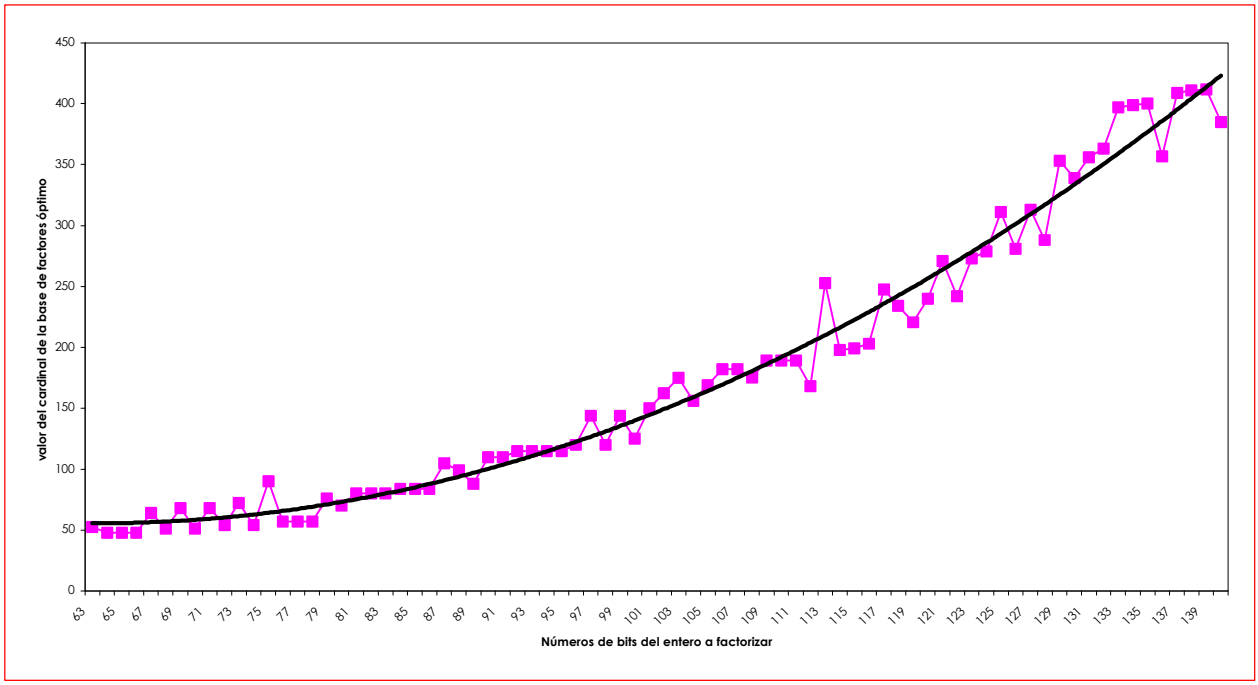
El valor de esta expresión es 0.14: se obtiene una cantidad de conjuntos S que es un 14% de la cantidad de pares $P-C$ hallados, cuando esa cantidad de pares es cercana al cardinal de la base de factores.

6.3.2. Valores óptimos del cardinal de la base de factores.

La cantidad de primos que contenga la base de factores influye en el tiempo que invierte la aplicación en factorizar un entero. Una base de factores especialmente pequeña ocasiona que casi todos los pares $P-C$ queden rechazados, pues se hace difícil lograr que el valor de C_i sea B–suave, y por otro lado el límite superior p_q^2 tiene un valor bajo y muchos pares $P-C$ que podrían ser almacenados por ser de factor grande quedan también rechazados. Una base de factores con un cardinal muy elevado exige que cada par a rechazar no logra ser descartado hasta después de haber pasado por el proceso de intentos por división con todos los primos de la base.

Por eso, como ya hemos señalado antes, hemos factorizado muchos números, de diferentes tamaños, todos ellos con diferentes valores (10 valores diferentes cada vez, y en cada caso con 5 valores diferentes en la cantidad de relaciones con factor grande a almacenar) del cardinal de la base de factores.

En un intervalo de tamaños que va desde 63 bits hasta los 145 bits, los mejores tiempos de factorización se han obtenido con los valores del cardinal de la base de factores que se recogen en la Gráfica 1. No presentamos los tiempos logrados para estas factorizaciones, que quedarán recogidos en el siguiente Capítulo de esta tesis. Mostramos, para cada tamaño, la mediana del



Gráfica 1: Valor de los cardinales de la base de factores que mejores resultados de tiempo ofrecen en el proceso de factorización de enteros, en función del tamaño (en bits) del número a factorizar.

mejor valor del cardinal de la base obtenida entre todas las factorizaciones de cada uno de los tamaños.

La gráfica puede aproximarse a una línea de tendencia. La ecuación que mejor se aproxima a nuestra gráfica es la de una parábola. En concreto nos ha resultado

$$y = 0.0624 \cdot x^2 - 0.1576 \cdot x + 55.694,$$

donde el valor de la variable $x = n - 63$, donde n es el número de bits del número a factorizar. La expresión es válida para ese intervalo de valores, y no puede tomarse por tanto como una expresión de tipo general.

Esta expresión —simplificada en la forma $y = 0.06 \cdot x^2 - 0.2 \cdot x + 50$— es la que hemos tomado para definir, en cada caso, el cardinal de la base de factores de los números, en este rango, que hemos querido factorizar.

6.3.3. Implementación de Macros.

Todas las funciones presentadas en este y en los Capítulos previos han quedado definidas también en forma de macro, mediante directivas de preprocesador `#define`. Y la aplicación de factorización también ha sido ejecutada paralelamente con las macros, sin usar ni una sola función y eliminando así todas las llamadas. El programa queda así mucho más largo en memoria,

pero como se verá reduce los tiempos de ejecución.

Para este estudio de tiempos hemos trabajado en una máquina con dos procesadores Intel Pentium III a 450 MHz. Ambos procesadores han trabajado paralelamente y hemos lanzado el programa de factorización a la vez: una versión con funciones y otra enteramente implementada con macros. No interesa aquí tanto el valor absoluto de los tiempos sino las diferencias entre una implementación y otra.

Para el estudio de tiempos hemos factorizado 3500 enteros tomando diferentes cardinales de la base de factores y diferentes cantidades de relaciones grandes a almacenar, y buscando los valores de parámetros que ofrecieran menor tiempo de ejecución. En el programa que hemos implementado cada compuesto es factorizado nueve veces: utilizando tres bases de factores distintas y, en cada base, tres valores de cantidad de relaciones de factor grande a buscar. En cada nuevo número a factorizar, los cardinales de las tres bases de factores son el triple del tamaño (número de bits) del número a factorizar, la primera; lo mismo incrementado en 30 la segunda; lo mismo incrementado en 60 la tercera. Y en cada base de factores, la cantidad máxima de relaciones de factor grande a buscar será 10 veces el cardinal de la base de factores; lo mismo incrementado 500; lo mismo incrementado 1000.

El proceso con las macros ha invertido 1.377.338 segundos en las tareas de factorización; el proceso con funciones ha consumido en cambio 1.604.290 segundos. El rendimiento es del 16,5 % superior en el proceso con macros: un incremento nada despreciable, y que podremos apreciar en las Tablas recogidas en el anexo V del final de la tesis.

7 OPTIMIZACIÓN DE CÓDIGO

Como señalan Bruce SCHNEIER y Doug WHITING en [Schn97] lo más importante, en una implementación de cualquier algoritmo criptográfico, es la seguridad. Pero, apostillan, es necesario realizar la implementación de forma que se obtengan al final herramientas eficientes.

Y es que es posible escribir programas en C, aparentemente eficientes, pero que posteriormente, al compilar, no logran utilizar de forma óptima los recursos que ofrecen las nuevas arquitecturas de los microprocesadores actuales. Y si bien es verdad que los compiladores han evolucionado y mejorado mucho en los últimos años, también es cierto que si tenemos en cuenta esas arquitecturas podemos llegar a códigos ejecutables mucho más eficientes.

El uso de la segmentación permite que en cada ciclo de reloj pueda finalizarse una instrucción, sin que ello implique que cada instrucción se ejecute en un sólo ciclo. Sin embargo, puede ocurrir que una instrucción más compleja (como por ejemplo, los productos o los cocientes) retarde la ejecución de las siguientes instrucciones.

También la ejecución superescalar supone una mejora importante de las nuevas arquitecturas.

Gracias a ella, el procesador puede ejecutar más de una instrucción por ciclo de reloj. Sin embargo, requiere una mayor vigilancia en la cadena de instrucciones, y eso exige un mayor coste de trabajo. Además, sólo algunas instrucciones pueden ejecutarse en paralelo, pues podemos tener dependencias entre instrucciones consecutivas del código en C que impidan emplear a pleno rendimiento la ejecución superescalar. Por tanto, como se afirma en [Schn97], aunque las arquitecturas superescalares ofrecen la posibilidad de incrementar de forma sustancial la velocidad, también incrementan de forma dramática las penalizaciones por una mala implementación de los algoritmos. Y conforme el grado de ejecución superescalar aumenta, las reglas de optimización son cada vez más complejas.

Otros dos factores de importancia decisiva para la eficiencia en la implementación de los algoritmos son la cantidad de accesos a memoria requeridos y el número de instrucciones de salto que deberá predecir y ejecutar el microprocesador. Una implementación será más eficiente cuando, al margen de otras consideraciones, se logre reducir al máximo el número de referencias a memoria de datos y, por tanto, la cantidad de fallos en el acceso a la memoria caché en sus diferentes niveles. Y de la misma manera, también ganaremos velocidad cuanto menor sea el número de instrucciones de salto ejecutadas y, especialmente, las instrucciones de salto equivocadamente predichas.

7.1. MEDICIÓN DE TIEMPOS DE EJECUCIÓN DE NUESTRA APLICACIÓN, ANTES DE SER OPTIMIZADA.

Las características técnicas más destacables de la máquina que hemos empleado para las mediciones finales y definitivas de los tiempos de ejecución, realizadas antes y después del proceso de optimización, quedan recogidas en la Tabla 1.

vendor_id	GenuineIntel
model name	Intel(R) Pentium(R) 4 CPU 2.00GHz
cpu MHz	2000.233
cache size	512 KB

Tabla 1: Características técnicas del ordenador utilizado para las mediciones últimas de los tiempos de factorización.

En el Capítulo anterior hemos mostrado la implementación de una aplicación para factorizar enteros, que sean producto de dos enteros primos, basada en la técnica de las fracciones continuas. Hemos explicado el modo en que esta aplicación ha sido estudiada y las mejoras que, a lo largo de sucesivos años, se han introducido en el algoritmo para lograr reducir tiempos de computación. Hemos mostrado los valores óptimos para el cardinal de la base de factores, el

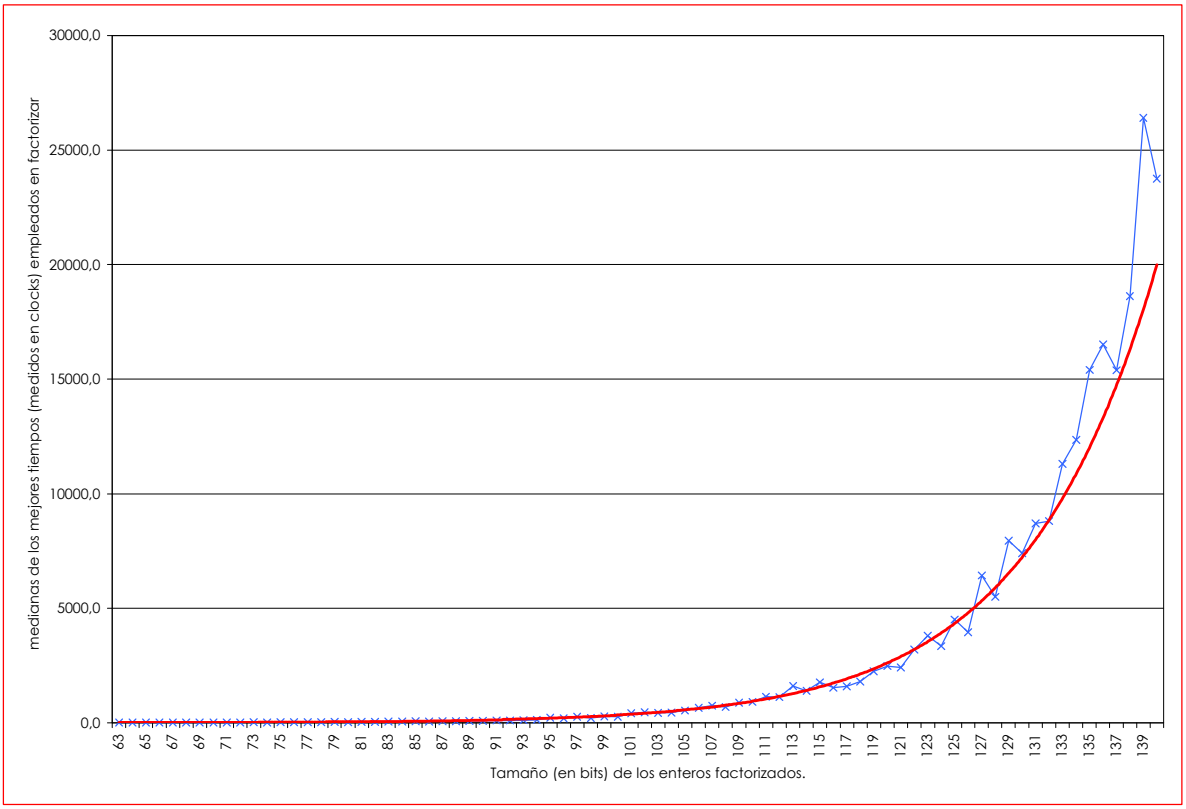
número de filas de la matriz histórica que, de media, es necesario emplear para lograr encontrar los factores no triviales del entero que queremos factorizar. Pero hemos dejado de lado cualquier estudio sobre los tiempos de ejecución. Esa es una tarea que hemos querido dejar reservada para este Capítulo. Presentamos ahora los tiempos de la aplicación no optimizada. Al final del Capítulo volveremos a realizar una exploración de tiempos, para poder obtener una gráfica comparativa, tomada para diferentes tamaños de los enteros a factorizar.

bits	clocks	bits	clocks	bits	clocks	bits	clocks
63	13,5	83	60,0	103	443,0	123	3805,5
64	13,0	84	59,0	104	450,0	124	3351,0
65	14,0	85	74,5	105	536,0	125	4506,0
66	14,0	86	61,5	106	661,5	126	3957,0
67	18,0	87	81,0	107	757,5	127	6428,5
68	15,0	88	81,5	108	691,0	128	5504,0
69	22,0	89	84,0	109	881,0	129	7958,0
70	18,0	90	91,0	110	917,0	130	7404,5
71	22,5	91	110,0	111	1144,5	131	8709,0
72	18,5	92	112,0	112	1124,5	132	8813,0
73	27,0	93	123,5	113	1608,0	133	11305,0
74	21,0	94	133,0	114	1402,0	134	12352,5
75	26,5	95	221,0	115	1769,5	135	15405,5
76	26,0	96	191,0	116	1538,0	136	16523,5
77	33,0	97	270,0	117	1592,0	137	15385,0
78	29,0	98	217,0	118	1807,0	138	18621,0
79	38,0	99	279,0	119	2257,5	139	26408,5
80	34,5	100	268,0	120	2477,0	140	23744,0
81	45,0	101	419,5	121	2412,5		
82	45,0	102	460,0	122	3199,5		

Tabla 2: Valores obtenidos con la función `times(NULL)`, según los tamaños de los enteros a factorizar, utilizando la aplicación basada en el algoritmo CFRAC, y sin optimización alguna de código. Para obtener el valor en segundos bastará con dividir los valores de la tabla por la constante `_SC_CLK_TCK`.

En la Gráfica 1 se recoge la representación de la evolución del coste de tiempo en la factorización según el tamaño (en bits) del entero a factorizar. Los valores representados son, para todos los tamaños, los mejores que hemos obtenido dentro de todos los procesos de factorización, y que, como ya hemos dicho en el Capítulo anterior, corresponden a los valores del cardinal de la base de factores que ha quedado representada en la Gráfica 1 de ese Capítulo 6.

Los datos recogidos en la Tabla 2 han quedado representados en una gráfica que viene representada en la Gráfica 1 y viene recogidos en color AZUL. La expresión matemática de la línea tendencia (recogida en color ROJO) que más se aproxima a nuestra gráfica es una



Gráfica 1: Tiempos empleados en la factorización de enteros de diferentes tamaños. Las abcisas recogen los tamaños en bits. Las ordenadas los tiempos medidos en ticks.

exponencial de la forma $y = 7{,}1556 \cdot \exp(0{,}1017 \cdot x)$.

La función que hemos utilizado para la medición de tiempos es la ya comentada `times()`, de la librería `sys/times.h`, que con el parámetro `NULL` devuelve un valor proporcional al tiempo transcurrido desde que el sistema se arrancó por última vez. El número de ticks o clocks por segundo es un parámetro, llamado `_SC_CLK_TCK`, que se puede obtener usando la función `sysconf()` y mostrando el valor que devuelve con el parámetro `sysconf(_SC_CLK_TCK);` y mostrando el valor mediante un pequeño programa. El valor que tiene en la máquina que hemos usado nosotros es 100: es decir, la precisión en la medición de los tiempos obtenidos en nuestro proceso es de centésimas de segundo.

7.2. PROTOCOLO GENERAL DE ACTUACIÓN.

Los autores de [Schn97] recogen una lista de tareas que consideran necesarias a la hora de procurar optimizar un algoritmo. También afirman que está empíricamente observado que, en muchos casos, un algoritmo optimizado para un procesador superescalar queda también optimizado para otro procesador más simple. La lista de tareas que recomiendan es la siguiente:

1. Evitar los saltos condicionales en los bucles más anidados. Cualquier cambio impredecible en

el control del flujo en el algoritmo produce normalmente un vaciado de la ventana de instrucciones y un costo extra de ciclos de reloj. En particular, cualquier instrucción **`if`** / **`else`**, o el operador condicional de C (`? :`), análogo a una estructura condicional **`if`** / **`else`**, con sentencias condicionadas simples, causarán en el lenguaje ensamblador instrucciones de salto que deberán ser ejecutadas.

2. Desenrollado de bucles: es crítico localizar cualquier dependencia entre las últimas instrucciones de una iteración y las primeras instrucciones de la siguiente iteración; estas dependencias pueden generar paradas en la segmentación.
3. Evitar instrucciones intrínsicamente complicadas y costosas en ciclos de reloj.
4. Limitar el número de variables. Si en un bucle anidado definimos un número demasiado alto de variables, no podrán ser almacenadas todas ellas en los registros y entonces la ejecución quedará gravada en tiempo por todos los accesos que deberemos hacer en memoria. Es decir, se trata de optimizar y minimizar el uso de los registros internos del microprocesador.
5. Limitar el tamaño de las tablas. Aunque desde un punto de vista criptográfico es mejor trabajar con grandes tablas, las tablas pequeñas son siempre mejores desde el punto de vista de la velocidad del software.
6. Permitir paralelismo. Permitir la ejecución del mayor número posible de instrucciones independientes. Este principio choca frecuentemente con las dependencias de datos, donde tendremos que una segunda operación no puede ejecutarse en paralelo junto con la primera, puesto que los operadores de la segunda dependen de los resultados de la primera.
7. Permitir adelantar la indexación de las tablas, de manera que cuando tengamos que acceder a un valor concreto de la tabla ya dispongamos del valor del índice.

Es conveniente tener en cuenta estos siete puntos en el momento de implementar un algoritmo criptográfico. Como señalan Bruce SCHNEIER y Doug WHITING en este artículo largamente citado [Schn97], disponer de una guía general de optimización y realizar una exploración de la implementación del software puede ayudar en el futuro a mejorar la velocidad de ejecución de los algoritmos criptográficos. Y como afirma Craig S. K. CLAPP [Clap97], el incremento de la eficiencia de los procesadores, su segmentación y el aumento del paralelismo, pueden orientar hacia nuevos criterios a la hora de diseñar nuevos algoritmos de cifrado.

El procedimiento que nosotros hemos diseñado y seguido ha sido el siguiente:

En primer lugar, ejecutar repetidas veces el programa implementado para la factorización de un entero, compuesto de dos primos grandes, y que ya hemos mostrado en el Capítulo 6. Todo el proceso de optimización ha sido realizado con un número prefijado de 100 bits, producto de dos primos de 50 bits cada uno:

```
8 4218BCE3 04AFD040 00D35DB9 = 2C7E3 25E2BDE1 * 2F90B F8A9AAD9
```

El objetivo de esta reiterada ejecución es determinar cuáles son las funciones que consumen mayor cantidad de tiempo. Como sugiere la Ley de AMDHAL, convendrá centrar los procesos de optimización en esas funciones.

Para cada función a optimizar hemos seguido un protocolo de actuación. Este protocolo puede sintetizarse en los siguientes cinco pasos:

1. Búsqueda de un algoritmo, distinto al empleado, que reduzca el tiempo de ejecución. Es decir, búsqueda de algoritmos mejores a los utilizados en una primera implementación.
2. Procurar eliminar instrucciones. Una vez tenemos el Algoritmo seleccionado, lograr su ejecución con menos instrucciones supondrá siempre una mejora en la eficiencia de su ejecución. Se trata de reducir instrucciones sin variar el planteamiento del algoritmo seleccionado. De lo contrario estaríamos en un intento de mejora como el señalado en n.1; y hay que tener en cuenta que no toda reducción de instrucciones es conveniente: a veces puede lograrse al precio de aumentar los ciclos de reloj por instrucción (índice CPI), lo que puede llevar finalmente a un incremento del tiempo de ejecución. Hemos aplicado cinco métodos encaminados a esta reducción de instrucciones:
    2.1. Simple eliminación de instrucciones prescindibles, o reducción a llamadas a funciones. Nos referimos específicamente a la reducción de instrucciones por simple eliminación. Aunque este paso puede parecer trivial, la realidad es que a medida que el código a optimizar sufre diferentes modificaciones, orientadas a su mejora en la eficiencia, aparecen de hecho instrucciones residuales prescindibles.
    2.2. Cambio de tipo de datos. En la medida de lo posible, hemos procurado trabajar con variables de longitud igual a la longitud de palabra del ordenador en el que hemos trabajado: 32 bits. Las variables enteras de 64 bits exigen un alto coste de ciclos de reloj y de instrucciones a ejecutar. Estos cambios no son inmediatos, pues no siempre, en muchos algoritmos implementados, es fácil prescindir de la variable entera de 64 bits.
    2.3. Cambio en las estructuras de programación. A veces puede suponer una reducción de instrucciones cambiar una estructura iterada (**`while`**, ó **`do – while`**, en C) por una secuencia de instrucciones condicionadas (**`if – else`**) o viceversa.
    2.4. Eliminación de llamadas a funciones por inserción de su código en la función principal (in–line functions).
    2.5. Reuso de instrucciones: creación de variables que almacenan un valor repetidamente calculado.
3. Procurar reducir las instrucciones de salto: todo lo que reduzca estas instrucciones favorecerá el índice CPI. Para lograr este objetivo, una técnica es la de desenrollar los bucles, procurando un número suficiente de instrucciones dentro del bucle. Con esta tarea se pretende lograr que

la ventana de instrucciones se mantenga siempre llena de instrucciones a ejecutar. (Hemos obtenido que un valor adecuado puede estar en torno a las 15 instrucciones de alto nivel).

4. Evitar dependencias de datos: Para las dependencias tipo WAR o tipo WAW la solución es muy sencilla y se reduce a la definición de variables intermedias. Para las dependencias tipo RAW, como ya ha quedado dicho en el Capítulo 3, la solución se centra en separar las instrucciones que sufren este tipo de dependencia de datos: Insertar código entre dos sentencias con variables con dependencias RAW; y realizar segmentación software.
5. Optimizar los accesos a memoria. El número de accesos a memoria, y los fallos de acceso a memoria. Para optimizar estos accesos tenemos una colección de pequeñas técnicas: prefetching, la mezcla de arrays, el intercambio de bucles, la fusión de bucles, etc.

Cada nueva mejora introducida se estudia de forma aislada: cada mejora individual dirigida a un solo objetivo del protocolo de actuación genera una nueva versión de la función a optimizar, diferente de la versión previa únicamente en esa mejora individual. Si la reducción del número de instrucciones o de los ciclos de reloj invertidos es mayor que el 3 %, entonces la mejora queda como definitiva en el proceso, y se continúa el protocolo en el siguiente paso, buscando una nueva versión de la función a partir de la actual ya modificada y validada. El objetivo en todo momento es doble: reducir el número de instrucciones, y reducir el valor del índice CPI (o lo que es lo mismo, aumentar el valor del índice IPC).

7.3. LA HERRAMIENTA RABBIT

Para toda esta tarea que acabamos de describir, hemos podido hacer uso de una serie de contadores internos, disponibles en los nuevos microprocesadores, cuya lectura no es accesible mediante código en lenguaje C, pero a los que sí podemos llegar mediante algunas herramientas existentes en el mercado, tanto de libre distribución como propietarias (como por ejemplo, la Vtune [Vtune], implementada por Intel). Estas herramientas nos permiten estudiar y analizar el comportamiento de nuestro código y su interacción con la máquina sobre la que se realiza la ejecución del código.

La herramienta que hemos usado para acceder a los valores de los contadores de los registros internos del microprocesador ha sido RABBIT (cfr. [Rabbit]). Esta herramienta, de libre distribución, permite acceder y mostrar los contadores de eventos de los procesadores Intel y AMD ocurridos en la ejecución de programas implementados en C, bajo el sistema operativo Linux.

La puesta en marcha de esta herramienta exige algo de programación. Hemos definido un conjunto de macros, que nos permiten analizar los valores de los registros que deseemos, en cualquiera de las funciones o en fragmentos del programa general de factorización. Hemos modificado todas y cada una de las funciones que deseamos optimizar, introduciendo en ellas el código imprescindible para poder analizar su comportamiento en el proceso general.

Los eventos que hemos estudiado, de entre todos los que ofrece la herramienta RABBIT, han sido 10 (distribuidos en tres bloques), y que quedan recogidos en la Tabla 3. RABBIT ofrece una serie de eventos que dan buena información sobre las instrucciones de salto y sobre las predicciones: número de instrucciones de salto ejecutadas, número de instrucciones de salto equivocadamente predichas, número de saltos tomados ejecutados, y número de saltos tomados mal predichos y ejecutados. También ofrece información sobre los accesos a memoria. Podemos conocer el número de accesos a memoria que se tienen en la ejecución de un determinado código que estamos analizando para optimizar: ese evento de RABBIT ofrece el número de referencias a memoria, tanto a memoria caché como a la memoria principal o incluso a disco. También ofrece los eventos que recogen el número de fallos de acceso a la memoria L1 y el número de fallos de acceso a la memoria L2. En los tres eventos se incluyen las cargas especulativas para instrucciones ejecutadas y no realizadas finalmente; en ningún caso se incluyen los almacenamientos especulativos. Los valores de los registros del primer y del tercer bloque nos han sido de gran utilidad para buscar modificaciones que redujeran tiempos buscando la reducción de saltos o la reducción de accesos a memoria o de fallos a la caché. El objetivo en todo momento ha sido reducir el tiempo de ejecución de la aplicación, y para ello nos hemos centrado en las siguientes metas: reducir el número de instrucciones, y aumentar el valor del índice IPC (instrucciones por ciclo).

```
0x24  Fallos en L2
0x43  Referencias a memoria de datos
0x45  Fallos en L1

0x79  Ciclos de reloj
0xC0  Instrucciones ejecutadas
0xC2  Microinstrucciones ejecutadas

0xC4  Instrucciones de salto ejecutadas
0xC5  Instrucciones de salto equivocadas predichas
0xC9  Saltos tomados ejecutados
0xCA  Saltos tomados mal predichos ejecutados
```

Tabla 3: Listado de eventos estudiados con RABBIT.

El valor del número de instrucciones lo obtenemos directamente del registro `0xC0` (número de instrucciones ejecutadas) y del registro `0xC2` (número de microinstrucciones ejecutadas). Los ciclos de reloj los obtenemos directamente del registro `0x79` (número de ciclos de reloj transcurridos durante la ejecución de todas instrucciones del bloque que estamos analizando y procurando optimizar). Gracias al número de ciclos, y gracias al número de instrucciones ejecutadas, es inmediato obtener el valor del índice IPC como el cociente entre el valor del registro `0xc0` y el valor del registro `0x79`.

7.4. ANÁLISIS DEL PROCESO DE FACTORIZACIÓN.

Tal y como indicábamos en el epígrafe anterior, la primera tarea a realizar ha de ser la ejecución reiterada del programa de factorización para obtener, gracias a la lectura del contador `0x79`, el tiempo de ejecución de cada una de las 40 funciones que intervienen en todo el proceso. En la Tabla 5 recogemos la relación de las 20 funciones más costosas en tiempos de ejecución ordenadas según esos valores de tiempos obtenidos. Los valores presentados en esa tabla recogen los ciclos de reloj en los que cada función está ejecutándose en el proceso total de factorización. No todas las funciones son invocadas el mismo número de veces, y los valores que recoge la Tabla 5 son los acumulados en un solo proceso de factorización, después de que algunas funciones hayan recibido muchas llamadas. El tiempo de las otras 20 funciones podemos considerarlo despreciable, y supone menos del 1 % del total. Por la ley de AMDHAL, bien podemos prescindir de ellas para las tareas de optimizaciones.

vendor_id	GenuineIntel
nombre modelo	Pentium III (Coppermine)
cpu MHz	997.472345
tamaño cache L2	256 Kbytes
tamaño cache L1	16 Kbytes
tamaño bloque	32 bytes

Tabla 4: Características técnicas del ordenador utilizado para todos los trabajos de optimización mediante la herramienta RABBIT.

Todas las medidas obtenidas con la herramienta RABBIT y mostradas a lo largo de este capítulo han sido tomadas sobre un ordenador cuyas características quedan recogidas en la Tabla 4.

En estos tiempos mostrados en la Tabla 5, y como decíamos antes, hay que tener en cuenta que no todas las funciones son invocadas el mismo número de veces. También nos interesa saber el número de llamadas que recibe cada una de las 20 funciones. En la Tabla 6 mostramos las mismas 20 funciones de la Tabla anterior, ordenadas ahora por el número de veces que cada una es invocada en la aplicación. Esa información también es de gran utilidad, porque una pequeña optimización, realizada en una función invocada varios millones de veces, resulta siempre rentable, aunque su reducción de tiempo en una sola ejecución de la función sea pequeña.

En la Tabla 7 recogemos de nuevo la lista de las 20 funciones, esta vez ordenadas según el valor obtenido gracias al contador `0xC0` (instrucciones ejecutadas). Con estos valores, y los presentados en la Tabla 5, podemos conocer el valor del índice IPC de cada función, y detectar así el grado de paralelismo de cada una de ellas.

		ciclos de reloj	porcentaje del total
1	Suave_S	2.991.356.843	23,61
2	Modulo	2.302.815.183	18,17
3	RelacionesBhascara	1.325.755.488	10,46
4	EliminacionGaussiana	983.505.324	7,76
5	OrdenRelaciones	867.243.499	6,84
6	longitud	742.386.827	5,86
7	Cociente	617.352.622	4,87
8	MODULO	571.220.615	4,51
9	DESPL_izda	484.609.406	3,82
10	CrearNumero	336.206.555	2,65
11	PROD_bit	326.226.736	2,57
12	COCIENTE	304.322.614	2,40
13	PonerACero	271.170.391	2,14
14	SUMA	148.622.492	1,17
15	RESTA	132.761.172	1,05
16	CopiarNumero	95.507.792	0,75
17	BuscarCuadrados	77.532.679	0,61
18	Euclides	31.991.818	0,25
19	PonerACeroRel	24.662.592	0,19
20	orden	19.006.563	0,15

Tabla 5: Funciones más costosas en tiempo de ejecución (en ciclos de reloj).

El protocolo de optimización ha sido aplicado por riguroso orden de peso: tal y como nos indica el orden de la Tabla 5. En el siguiente apartado de este capítulo recogemos ejemplos de cada uno de los cinco pasos del protocolo descrito más arriba. En total se han originado más de 200 versiones diferentes de las distintas funciones optimizadas.

En el anexo VI del final de la tesis se recogen, en una serie de tablas, los valores de todos los registros para cada una de las 40 funciones que intervienen en la factorización, tomados antes de iniciar el proceso de mejora de código y después de terminarlo sobre todas las funciones analizadas.

Ya hemos señalado cuáles son las características técnicas del ordenador que hemos sutilizado para nuestro trabajo de optimización (cfr. Tabla 4). En la Tabla 8 recogemos las características del software de la máquina.

A lo largo de los próximos epígrafes recogemos ejemplos de cada uno de los pasos seguidos en el protocolo de optimización, acudiendo a una u otra función para presentar modificación de código y valores de los registros, de manera que quede mostrado cómo una modificación sencilla del código puede ofrecer mejoras sustanciales en los tiempos de ejecución.

Iremos presentando algunos ejemplos de cada uno de los pasos. En total hemos optimizado el código de 11 funciones: las que más recurso de tiempo empleaban. En la presentación que sigue recogemos ejemplos de cada una de las técnicas de optimización que hemos empleado.

Únicamente presentamos aquellos ejemplos más notables de cada uno de estas técnicas. En todos los ejemplos ofreceremos los valores de los parámetros más significativos antes y después de la mejora.

	número de llamadas
Modulo	14.641.005
DESPL_izda	12.921.596
PonerACero	3.000.411
CopiarNumero	1.659.896
longitud	1.345.085
orden	1.083.761
CrearNumero	849.711
SUMA	600.412
RESTA	598.654
PROD_bit	249.181
Cociente	145.303
MODULO	88.544
COCIENTE	83.123
PonerACeroRel	82.757
RelacionesBhascara	82.757
Suave_S	41.620
Euclides	950
BuscarCuadrados	9
OrdenRelaciones	4
EliminacionGaussiana	1

Tabla 6: Funciones más costosas, ordenadas según el número de llamadas que reciben.

	número de instrucciones ejecutadas
Suave_S	2.961.746.229
Modulo	2.057.140.312
RelacionesBhascara	1.713.630.879
OrdenRelaciones	1.212.667.593
longitud	944.378.153
MODULO	853.348.861
Cociente	820.917.600
DESPL_izda	780.978.112
COCIENTE	418.521.009
PROD_bit	350.243.126
CrearNumero	185.742.033
SUMA	140.773.445
RESTA	120.465.161
BuscarCuadrados	100.080.633
PonerACero	94.152.291
CopiarNumero	89.621.492
EliminacionGaussiana	55.235.941
Euclides	37.179.341
PonerACeroRel	14.897.570
orden	13.161.663

Tabla 7: Funciones más costosas, ordenadas según el número de instrucciones que ejecutan.

sistema operativo	Linux Red Hat
versión del sistema	6.2
versión del núcleo	2.2
compilador	`gcc`
opciones de compilación	`-03`
versión RABBIT	30 octubre, 2000

Tabla 8: Características software del ordenador utilizado para todos los trabajos de optimización.

7.5. PRIMERA MEDIDA: CAMBIO DE ALGORITMO.

No consideramos este paso como parte sustancial de la optimización de nuestros algoritmos ya que, de hecho, de lo que se trata en este caso no es de optimizar la implementación de un

determinado algoritmo, sino de sustituir ese algoritmo por otro esencialmente mejor. Pero pensamos que sí es conveniente recogerlo en esta memoria pues en el proceso de optimizar una aplicación lo primero es escoger el mejor entre los algoritmos posibles conocidos. Además, los valores que ofrece RABBIT en alguna de estas mejoras son enormemente significativos y no hemos querido omitir su presentación.

Mostramos dos ejemplos de este proceso de mejora. El primero consiste en una modificación del método de ordenación aplicado en la función `OrdenarRelaciones()`. Dejamos el método de ordenación de la burbuja y aplicamos el método desarrollado por C. A. R. HOARE e implementado mediante la función `qsort()`, de la biblioteca `stdlib.h` [Schi94]. El segundo es la corrección de parámetros en la función de la criba de ERASTHÓTENES, que debido a una serie de correcciones previas que presentaremos más adelante en otras funciones, puede redefinirse reduciendo sustancialmente los valores que debe calcular. No lo consideramos una mejora por reducción de instrucciones, sino que lo encuadramos en este apartado de modificación de algoritmo porque toda la ganancia (que en términos relativos es muy significativa) descansa en un ajuste en los valores de los parámetros que, colateralmente, implican una reducción de instrucciones.

7.5.1. Cambio de método de ordenación.

	v. 01 (método Burbuja)	v. 02 (función qsort)	factor de reducción
Ciclos de Reloj	496.202.121	10.003.455	49,60
Instrucciones Ejecutadas	591.134.570	8.558.199	69,07
IPC	1,191	0,856	
Fallos en L2	2.299	7.578	
Referencias a memoria de datos	375.134.409	7.196.684	52,13
Fallos en L1	14.303.852	28.665	499,00
Instr. de Salto ejecutadas	95.661.297	1.238.340	77,25
Inst. de Salto equivocadas	5.232.433	92.611	56,50
Saltos tomados ejecutados	74.811.679	933.860	80,11
Saltos tomados mal pred. ejec.	2.605.847	47.500	54,86

Tabla 9: Optimización en la función `OrdenarRelaciones()` por cambio de Algoritmo de ordenación.

La función `OrdenarRelaciones()` toma un vector de elementos `UINT4`, que llamamos `indice` y lo ordena según otro vector llamado `V_Fact`, de la misma dimensión, y también de tipo `UINT4`. El vector `indice` tiene asignados inicialmente sus valores ordenados desde el 0 hasta el valor de

la dimensión de los vectores menos uno. Al final del proceso tendremos los valores de `indice` ordenados de forma que el primer elemento indique la posición donde está el menor de los valores del vector `V_Fact`, el segundo elemento indique la posición donde está el siguiente de los valores de `V_Fact`, y así sucesivamente, hasta recoger en la última posición de `indice` la posición donde se encuentra el mayor de los valores de `V_Fact`.

Aparte de otras optimizaciones importantes que presentaremos más adelante, en otro apartado de este capítulo, la mejora más notable en este primer paso del proceso de optimización de código (cambio de algoritmo) la podemos observar en la sustitución del algoritmo de ordenación de la burbuja por el algoritmo antes señalado de la función `qsort()`. Desde luego este cambio era evidente desde el inicio, pero lo presentamos aquí porque presenta una mejora muy significativa en el proceso de ordenación, y para mostrar de forma clara la utilidad de la herramienta RABBIT. Mostramos en la Tabla 9 los valores comparados de todos los registros obtenidos con la herramienta RABBIT para la función con ambos algoritmos: en la versión v.01 está la implementación con el método de la burbuja; en la versión .02 está la implementación con la función `qsort()`.

La reducción de todos los parámetros (excepto el valor de fallos a L2) es de un factor mínimo de 50 en el peor de los casos. Hemos tenido que definir una función auxiliar que exige el diseño de la función `qsort()`. Pero todo el peso de su intervención queda reflejado en todos estos valores que recoge la Tabla 9.

7.5.2. Redefinición de parámetros en la función de la criba de ERASTHÓTENES.

Esta optimización apenas ha supuesto una modificación en el código, pero ha supuesto una mejora sustancial en los valores de los registros de una función. La mejora ha consistido simplemente en fijar los valores de los parámetros de la función de la criba de forma que reduzca al máximo las instrucciones que debe ejecutar. Los parámetros que hemos modificado son los encargados de definir la dimensión de la tabla de números para realizar sobre ella la criba de ERASTHÓTENES. La función modificada es la llamada `Erasthotenes()`: tiene un peso reducido en el conjunto del proceso de factorización; pero presentamos la optimización por su alto rendimiento relativo. Por la ley de AMDHAL, la mejora no será apreciable en el concurso total de la aplicación.

Más adelante presentaremos una serie de modificaciones realizadas sobre los valores que deben tener los primos de la base de factores. Debido a un cambio en el tipo de dato con el que trabajan algunas funciones, hemos exigido a la base de factores que sus primos sean de tipo de dato `UINT2`. Más adelante veremos también que, a pesar de esta reducción, la base de datos sigue siendo de una dimensión suficientemente grande para el rango de enteros que nosotros

podemos factorizar con nuestro algoritmo CFRAC. Lo importante ahora es, sencillamente, que pasamos de hacer la criba sobre una matriz de enteros que va desde el `0x1` hasta el `0x7A120` (500.000 en base decimal) a realizarla sobre un vector cuyo valor máximo sólo puede ser `0xFFFF`. Como es sabido, para el desarrollo de la criba bastará hacer el proceso de eliminación de números hasta llegar a la raíz cuadrada del valor máximo, que ahora es 65.535. Basta, por tanto, llegar a hacer la criba sólo con `0x100` primeros enteros.

Con estas dos modificaciones la cantidad de instrucciones del proceso queda reducido una enormidad. Hemos modificado los parámetros de forma que se llegue a una tabla igualmente válida, con menos datos y, sobre todo, con menos instrucciones. En la Tabla 10 podemos ver los valores de los registros de ciclos de reloj y de instrucciones ejecutadas. Los factores de reducción son elocuentes, y no merece la pena ningún comentario. Pero hay un valor que conviene señalar: la mejora muy sustancial que logra el índice IPC. Las causas de esta mejora podemos verlas en los demás parámetros obtenidos con RABBIT, y que vienen recogidos en la Tabla 11.

	v.00	v.01	factor de reducción
Ciclos de Reloj	178.593.924	623.709	286,34
Instrucciones Ejecutadas	8.059.039	651.422	12,37
IPC	0,045	1,044	

Tabla 10: Valores de la Optimización por ajuste de parámetros en la función `Erasthotenes()`.

Que disminuyan las instrucciones de salto era algo que suponíamos iba a ocurrir. La reducción de accesos a memoria también era previsible. Y pensamos que es el descenso notable de fallos a la caché (especialmente fallos de L2) lo que ha propiciado una mejora tan notable en el valor del IPC. Esta reducción se comprende al dejar de trabajar sobre un vector de 2 millones de bytes y pasar a trabajar sobre un vector de 256 Kbytes (ese es exactamente el tamaño de nuestra memoria L2: cfr. Tabla 4). Además ha quedado también reducido una enormidad el recorrido que debe hacerse sobre este vector, al limitarlo hasta el valor de la raíz cuadrada del entero mayor del rango numérico sobre la que realizamos la criba. Las mejoras de esta modificación de los parámetros quedan recogidas en las Tablas 10 y 11.

Como se puede comprobar, esta función no está recogida en la Tabla 5. No está por tanto entre las 20 que más peso tienen en el cómputo total de tiempos del proceso. No supone ni un 1 % de la aplicación. Pero era lógico hacer las modificaciones que hemos presentado, y que han venido dictadas por otros cambios diversos en otras funciones de más peso. La ley de AMDHAL no recomienda trabajar sobre una función como la que hemos mostrado. Pero hemos querido presentarla porque muestra, con unos valores muy elevados, que las mejoras por cambio de

algoritmo o por rediseño del algoritmo implementado pueden llegar a ser muy valiosas. Especialmente nos parece interesante porque se logra un aumento muy significativo del IPC en un factor de algo más de 23, que vemos muy relacionado con el descenso en los fallos en la caché.

	v.00	v.01	factor de reducción
Fallos en L2	507.044	4	126.761
Referencias a memoria de datos	2.036.263	341.791	5,96
Fallos en L1	746.322	47.922	15,57
Instr. de Salto ejecutadas	2.400.200	130.487	18,39
Inst. de Salto equivocadas	49.415	132	374,35
Saltos tomados ejecutados	2.314.529	130.324	17,76
Saltos tomados mal pred. ejec.	887	30	29,55

Tabla 11: Valores de los demás registros de la función `Erasthotenes()` obtenidos mediante la herramienta RABBIT.

7.6. SEGUNDA MEDIDA: REDUCCIÓN DE INSTRUCCIONES.

La reducción de instrucciones es siempre eficaz. Lograr el mismo resultado eliminando instrucciones que realmente no eran necesarias es, evidentemente, una forma de reducir tiempos.

En general, en el conjunto de toda la aplicación (y como se puede observar en el Anexo VI del final de la Tesis, donde se recogen las tablas de optimización todas las funciones) hemos procurado una reducción en el número de llamadas a muchas de las funciones, especialmente de las menores o más básicas como son la determinación de la longitud (número de bits del número), que pasa de 854.062 llamadas a 665.743; la función que copia el valor de un `NUMERO` en otra variable de ese mismo tipo, que pasa de 1.664.200 a 861.466 llamadas; la función que pone a cero todos los elementos de una variable de tipo `NUMERO`, que pasa de 3.009.224 a 1.610.065 llamadas; etc. Especial atención merece la reducción del número de llamadas a la función que define el operador relacional *mayor, menor o igual que* entre dos valores de tipo `NUMERO`: de casi 49 millones de llamadas a poco menos de ochocientas mil. Vamos a presentar en los siguientes apartados los distintos caminos que hemos seguido para lograr reducir las llamadas a las funciones.

Entendemos por instrucción inútil aquella que si se omite no altera el resultado final de la aplicación. Eso no quiere decir que la instrucción estuviera ya inicialmente de sobra. Un ejemplo de eso lo tenemos en la función que determina la suavidad o no suavidad de cada uno de los valores `Ci` que se intentan dividir con los primos de la base de factores (función `Suave_S()`). Cuando se intenta determinar la suavidad de un valor concreto `Ci`, lo que se hace (y eso una

cantidad enorme de veces: 14.641.005 de veces en el proceso completo de factorización de nuestro número de 100 bits) es averiguar si el cociente entre `Ci` y cada uno de los primos de la base de factores ofrece un resto igual a cero o no, sin pretender averiguar el valor concreto del resto en el caso de que finalmente éste sea diferente de cero. Podemos para ello invocar la función `Modulo()`, que devuelve el valor de ese resto. Pero también podemos quedarnos en averiguar si ese resto es cero o no. Saber si el resto es cero o no exige una operación de desplazamiento menos que saber su valor concreto. Y 14.641.005 desplazamientos menos es una buena reducción de instrucciones.

7.6.1. Eliminación de instrucciones prescindibles o reducción de llamadas a funciones. Ejemplo: función `RelacionBhascara()`.

Presentamos un caso de reducción de instrucciones que nos parece muy ilustrativo. Es el paso de la v. 00 a la v. 01 de la función encargada de buscar un nuevo par $P-C$, mediante el algoritmo de BHÁSCARA. La función la hemos llamado `RelacionBhascara()`, y trabaja con un tipo de dato, que es una estructura, que hemos ya mostrado en el Capítulo anterior, y que recoge todos los valores de las cinco secuencias que va generando el algoritmo en su búsqueda de una aproximación racional a un valor real irracional (raíz cuadrada de un entero que no es cuadrado perfecto).

La primera versión de la estructura `BHASCARA` es la que se recoge a continuación:

```
typedef struct
{
      NUMERO Ai;
      NUMERO Bi,Bj;
      NUMERO Ck,Ci,Cj;
      NUMERO Pk,Pi,Pj;
      NUMERO M;
      SINT4 i;
}BHASCARA;
```

En esta estructura se almacenan los valores de cuatro de las cinco secuencias del algoritmo de BHÁSCARA. Y para cada una de ellas guardamos el valor de la iteración actual (que identificamos con la letra `i`), el valor de la iteración previa, o `i-1`, (que identificamos con la letra `j`) y el valor de la iteración `j-1` ó `i-2` (que identificamos con la letra `k`). Los valores de las tres iteraciones son necesarios, pues cada valor `i`-ésimo depende de algún valor `j`-ésimo de algunas otras de las secuencias y de algún valor `k`-ésimo de algunas otras de las secuencias.

Por lo tanto, al final de cada proceso de ejecución de la función `RelacionBhascara()`, debíamos almacenar todos esos valores, lo que suponía una cadena de sentencias que eran llamadas a funciones:

```
      CopiarNumero(&Bh->Pi,&R->Pi);
      CopiarNumero(&Bh->Ci,&R->Ci);
```

```
CopiarNumero(&Bh->Aj,&Bh->Ak);
CopiarNumero(&Bh->Bj,&Bh->Bk);
CopiarNumero(&Bh->Cj,&Bh->Ck);
CopiarNumero(&Bh->Pj,&Bh->Pk);
CopiarNumero(&Bh->Ai,&Bh->Aj);
CopiarNumero(&Bh->Bi,&Bh->Bj);
CopiarNumero(&Bh->Ci,&Bh->Cj);
CopiarNumero(&Bh->Pi,&Bh->Pj);
PonerACero(&Bh->Ai);
PonerACero(&Bh->Bi);
PonerACero(&Bh->Ci);
PonerACero(&Bh->Pi);
Bh->i++;
```

Es decir, la versión v. 00 de la función lleva 10 llamadas a la función `CopiarNumero()` y cuatro llamadas a la función `PonerACero()`. Aunque estas funciones suponen un costo de computación no muy elevado, si se tiene en cuenta que en la ejecución de nuestro programa para el número de prueba la función `RelacionBhascara()` se ejecuta 82.711 veces, entonces se comprende el interés por eliminar todas estas llamadas a estas funciones, que supondría reducir sus llamadas en 827.110 llamadas menos en la función `CopiarNumero()` y en 330.844 llamadas menos en la función `PonerACero()`.

En un primer análisis del algoritmo observamos que no son necesarios todos los valores `j`-ésimos ni todos los valores `k`-ésimos. De las diez variables tipo `NUMERO` que tenía la primera estructura `BHASCARA` podemos quedarnos tan solo con 7 de ellos.

La segunda versión de la estructura `BHASCARA`, que surge al intentar modificar la función para eliminar un número grande de llamadas a funciones, es la que sigue:

```
typedef struct
{
      NUMERO BH[7];
      SINT4 i;
}BHASCARA;
```

Como se observa, hemos eliminado todos los elementos de tipo `NUMERO` que no son estrictamente necesarios, y los hemos agrupado en un vector.

La táctica que hemos seguido para eliminar llamadas a funciones es hacer un juego de índices y almacenar cada valor en una posición concreta dentro del vector `NUMERO BH[7]`: el último elemento (`BH[6]`) será siempre el valor que antes se llamaba `M`, y que recoge la parte entera de la raíz cuadrada del entero a factorizar. Los seis primeros elementos almacenan los valores `Bi` y `Bj` en las posiciones 4 y 5 del vector `BH` ; los valores de `Cj` y `Ck` (que al final lo identificamos con `Ci`) en las posiciones 0 y 1 del vector `BH`; los valores `Pj` y `Pi` en las posiciones 2 y 3 del vector `BH`. En cada iteración, en lugar de incrementar el valor de `BH->i` lo que hacemos es alternarlo de 0 a 1 y de 1 a 0: `BH->i = 1 – BH->i;` y finalmente definimos una variable `j` que en cada nueva iteración valdrá `j = 1 - BH->i;` y dos variables auxiliares tipo `NUMERO` (en la versión previa se empleaba una de esas variables auxiliares). Todo este encaje de variables y cambios en los valores de los índices nos permiten anular todas las llamadas a las funciones `CopiarNumero()`

y `PonerACero()`. En la Tabla 12 se recoge el valor de los índices que señalan la mejora de la función.

Se han reducido las instrucciones y se mantiene aproximadamente igual el valor del índice IPC. Como se comprueba en el código de la versión última y definitiva (v. 06) cualquier optimización que deseemos realizar sobre esta función `RelacionBhascara()` pasa ahora por mejorar cada una de las funciones implicadas en el proceso. Como se puede ver en las tablas del Anexo VI al final de la Tesis, los valores finales de los registros mostrados por RABBIT para esta función han quedado reducidos a menos de la mitad de cómo los vemos ahora en la Tabla 12. También puede verse más adelante, en el presente capítulo, en la Tabla 45 y en el Gráfico 3. Y eso gracias a que se han optimizado, posteriormente al análisis de esta función, todas las otras funciones implicadas en este proceso.

	v.00	v.01	% mejora
Ciclos de Reloj	1.329.099.790	1.144.199.782	13,91
Instrucciones Ejecutadas	1.713.631.075	1.461.734.869	14,70
IPC	1,289	1,278	

Tabla 12: Valores de la optimización en la función `RelacionBhascara()` al modificar la definición de la estructura `BHASCARA` para redefinir la función con menor cantidad de llamadas a otras funciones.

7.6.2. Cambio de los tipos de datos.

Presentamos cuatro ejemplos sobre este proceso de mejora basado en la reducción de instrucciones por cambio en el tipo de datos. Tres de ellos son el cambio del tipo de dato **`unsigned long long`** (y que llamaremos `UINT8`) al tipo **`unsigned long`** (que llamamos `UINT4`). El tercero es el cambio de tipo **`unsinged short`** (`UINT2`) al tipo de dato `UINT4`.

7.6.2.1. Reemplazo de variables de tipo **`unsigned long long`** por variables **`unsigned long`** en la función `Suave_S()`.

Para las funciones `Cociente()` o `Modulo()`, encargadas de calcular el cociente y el resto de dividir un entero largo definido con la estructura `NUMERO` (dividendo que consideramos que tiene `n` elementos `UINT4` para recoger todos sus dígitos) con una entero de tipo `UINT4`, empleamos una variable auxiliar `UINT8` (tipo **`usigned long long int`**). El proceso de división que hemos definido es el recogido en todas las referencias, y es el empleado habitualmente cuando realizamos una división: tomamos tantos dígitos a la izquierda del dividendo como dígitos tiene el divisor (llamamos a esta cifra, por ejemplo `D`: este valor puede almacenarse, como se verá

inmediatamente, en una variable `UINT8`) y realizamos esa división, obteniendo el primer dígito (el más significativo) del cociente (por ejemplo `*(c + n – 1)`: el cociente tendrá tantos dígitos como el dividendo, el primero de los cuales — el más significativo— puede ser cero: recuérdese que el divisor es de un solo dígito de base 2^{32}) igual al mayor entero que multiplicado por el divisor sea menor que `D`. Luego calculamos la resta entre `D` y ese producto `(*(c + n – 1)` multiplicado por el divisor), que llamaremos `dif`. El nuevo valor final de `D` queda establecido concatenando el siguiente dígito más significativo del dividendo a la derecha de `dif`. Se repite entonces el proceso hasta terminar con todos los dígitos del dividendo. Se llega así a un valor final de `D` menor que el divisor y que resulta ser el resto del cociente.

En la función que determina la condición de suavidad de un entero `Ci` (entero obtenido mediante el algoritmo de BHÁSCARA diseñado para calcular la aproximación racional, mediante la técnica de fracciones continuas, del valor de la raíz cuadrada de un entero; algoritmo que empleamos para la generación de relaciones necesarias para la factorización según la técnica de las fracciones continuas), necesitamos calcular muchas veces los restos de dividir un entero largo con cada uno de los primos de la base de factores. La función viene diseñada de tal manera que los divisores (los primos de la base de factores) puedan tener un tamaño de hasta 32 bits (tipo **`unsigned long int`**). Para ese caso, el código de la función que calcula el resto requiere una variable de tamaño 64 bits para almacenar el valor de la cifra que hemos llamado `D`: debe poder almacenar la diferencia `dif` que puede tener hasta 32 bits con la concatenación del siguiente dígito del dividendo que puede tener hasta 32 bits (como ya se señaló, trabajamos en base 2^{32}).

Si cambiamos esta exigencia y aceptamos que los primos de la base de factores no pueden sobrepasar los 16 bits entonces podemos definir la función que calcula el cociente y el resto de tal manera que la variable que almacena `D` sea `UINT4`.

Limitamos así el rango de la base de factores, pero no reducimos en nada las operaciones de la función, que simplemente sufre un cambio en el tipo de una de las variables. Ahora los primos de la base no pueden superar el valor 65.535 (es decir, $2^{16}-1$). Aunque esta limitación pueda parecer grave, la realidad es que en ninguno de los procesos de factorización que hemos ejecutado hemos alcanzado este rango. Existen 6.542 primos menores que ese límite superior. Si tenemos en cuenta que las bases de factores son tales que el símbolo del número `N` a factorizar con los primos que la forman ha de ser +1, y que eso corresponde a la mitad de los primos, tenemos que esa modificación hecha permite trabajar en procesos de factorización en los que la base de factores no supere los 3.200 primos.

Para analizar con más exactitud la limitación que supone este cambio de tipo de dato en el tamaño de los enteros que podríamos factorizar, tomamos la expresión calculada en el Capítulo 6 y que expresa los valores óptimos para los cardinales de las bases de factores en función de los tamaños de los enteros a factorizar. Esta expresión era $cfb = 0{,}06 \cdot (n-63)^2 - 0{,}2 \cdot (n-63) + 50$,

donde cfb es el valor óptimo para el cardinal de la base de factores y n es el tamaño (en bits) del número a factorizar. Si tomamos, como acabamos de señalar, como valor máximo para cfb el valor mitad de 6.542, es decir, 3.271, entonces el valor de n que obtenemos es 309, que es un tamaño que supera holgadamente el rango de enteros que pretendemos lograr factorizar con el algoritmo CFRAC y con nuestra implementación. Por lo tanto, podemos considerar nuestra restricción como válida y nada limitante, pues no cercena las posibilidades reales de nuestra implementación.

Para ver la repercusión de este cambio, presentamos los valores de los registros obtenidos gracias a la herramienta RABBIT antes y después de este cambio en la función que determina la suavidad de un entero grande (función `Suave_S()`). Los valores que mostramos en cada Tabla corresponden a los valores de los índices antes y después de ese cambio: muchas de las funciones optimizadas lo han sido en una sucesión de mejoras que han llegado a producir muchas versiones diferentes: por ejemplo en este caso (Tabla 13) esta modificación es la séptima de un total de 12 realizadas, y los valores de los parámetros ya están optimizados respecto a la versión .00 inicial.

	v. 06	v. 07	% mejora
Ciclos de Reloj	2.189.265.170	1.859.452.244	15,07
Instrucciones Ejecutadas	1.921.982.684	686.964.141	64,26
IPC	0,878	0,369	

Tabla 13: Optimización en la función `Suave_S()` por sustitución del tipo de dato **`unsinged long long int`** por el tipo de dato **`unsigned long`**.

	v. 06	v. 07	% mejora
Fallos en L2	1.061	920	13,29
Referencias a memoria de datos	1.457.211.380	412.023.444	71,73
Fallos en L1	16.088	7.836	51,29
Instr. de Salto ejecutadas	255.806.110	72.236.263	71,76
Inst. de Salto equivocadas	311.845	505.901	-62,23
Saltos tomados ejecutados	151.503.348	42.524.368	71,93
Saltos tomados mal pred. ejec.	85.382	126.479	-48,13

Tabla 14: Valores de los demás registros de la función `Suave_S()` obtenidos mediante la herramienta RABBIT en la modificación señalada en la Tabla 13.

Vemos que trabajar con variables de tipo `UINT8` supone un incremento muy grande de Instrucciones. Se ha reducido en un 15,07 % los ciclos de reloj, pero se ha reducido mucho más (un 64,26 %) las instrucciones a ejecutar. Hemos perdido por tanto bastante en el valor del índice IPC. Analizando los valores de los demás registros (ver tabla 14) observamos que mejoran

significativamente los accesos a la memoria caché (una reducción de fallos a L1 del 51,29 % y a L2 del 13,29 %), y las referencias a memoria (una reducción del 71,73 %). También es llamativo comprobar cómo disminuye en más de un 70 % las instrucciones de salto.

7.6.2.2. Reemplazo de variables de tipo **`unsigned long long`** por variables **`unsigned long`** en la función `OrdenarRelaciones()`.

Presentamos otro ejemplo de la mejora que supone cambiar el tipo de dato, de `UINT8` a `UINT4`. Nos trasladamos ahora a la función, ya tratada antes, llamada `OrdenarRelaciones()`. Ya hemos presentado la mejora de la v. 01 a la v. 02, cuando cambiábamos el método de ordenación. La mejora previa (de la v. 00 a la v. 01) es que veremos ahora.

Como ya hemos explicado, esta función toma un vector de elementos `UINT4` que llamamos `indice` y lo ordena según otro vector de la misma dimensión que almacena enteros de tipo (hasta el momento) `UINT8` y que llamamos `V_Fact`. Ya hemos explicado antes cómo se realiza la ordenación: quien determina el orden de los elementos del vector `indice` será el orden de los valores del vector `V_Fact`.

Cambiado el rango de los primos de la base de factores, que ya no pueden superar el valor `0xFFFF` (65.535), podemos cambiar también el tipo de dato para el valor `V_Fact`: este valor entero está definido, como puede verse en la lectura del Capítulo 6, para almacenar los factores grandes de cada factorización de cada valor de `Ci`. Puesto que el límite superior para un factor grande es el cuadrado del primo mayor de la base de factores, tenemos que este vector jamás necesitará más de 32 bits para almacenar factores grandes. No hay por tanto problema alguno ahora en cambiar el tipo de dato de este vector que, como hemos visto antes, al presentar la optimización por cambio de método de ordenación, queda finalmente como de tipo `UINT4`.

	v. 00	v. 01	% mejora
Ciclos de Reloj	867.767.008	496.202.121	42,82
Instrucciones Ejecutadas	1.212.564.416	591.134.570	51,25
IPC	1,397	1,191	

Tabla 15: Optimización en la función `OrdenarRelaciones()` por sustitución del tipo de dato **`unsinged long long int`** por el tipo **`unsigned long`**.

En la Tabla 15, hemos recogido la reducción de ciclos de reloj y de instrucciones que supone ese cambio en el tipo de dato del vector `V_Fact`, de forma que los operadores relacionales de ordenación se aplican sobre variables **`unsigned long int`** (`UINT4`) y no sobre variables **`unsigned long long int`** (`UINT8`). Los valores de los tantos por ciento de mejora muestran una vez más la gran carga de computación que supone trabajar con estas variables que no se ajustan al tamaño de palabra del computador.

7.6.2.3. Reemplazo de variables de tipo **`unsigned long long`** por variables **`unsigned long`** en la función `SUMA()`.

En la medida de lo posible hemos procurado eliminar en todo el código de nuestra aplicación la utilización de este tipo de dato. Solamente lo hemos mantenido en la función `SUMA()` definida para sumar dos enteros largos, y donde únicamente se emplea este tipo de dato para declarar una variable auxiliar necesaria para almacenar información en memoria y cuyo único operador que se le aplica es el de la suma con un entero de 32 bits; hemos eliminado cualquier otra operación aritmética o relacional con variables de este tipo de dato.

De todas formas, incluso en la misma operación `SUMA()` hemos modificado el código para eliminar en lo posible el uso de ese tipo de variables. El ejemplo del cambio realizado en esta operación es también significativo y vamos también a mostrarlo.

La función `SUMA()` emplea una variable `UINT8`. Es necesaria para poder realizar la suma de los dos dígitos del mismo peso de ambos sumandos y no realizar overflow: en el caso de que la suma de ambos dígitos `UINT4` de cada uno de los dos sumandos supere el valor `0xFFFFFFFF` tendremos en los 32 bits menos significativos de la variable `UINT8` el valor del dígito de la suma, y en el bit menor de los 32 bits superiores un 1. De esta forma controlamos el acarreo. Este proceso ha quedado extensamente presentado y explicado en el Capítulo 4.

Desde la versión v.03, el aspecto del bucle de la operación suma presenta una forma distinta, porque ha quedado ligeramente "desenrollado", como quedará explicado en un epígrafe posterior. El código presenta la siguiente forma:

```
for( ; i < l ; )
{     suma = suma >> Byte4;
      suma = suma + (UINT8)*(a->N + i) + (UINT8)*(b->N + i);
      *(c->N + i++) = (UINT4)suma;
      suma = suma >> Byte4;
      suma = suma + (UINT8)*(a->N + i) + (UINT8)*(b->N + i);
      *(c->N + i++) = (UINT4)suma; }
```

	v.03	v.04	% mejora
Ciclos de Reloj	94.956.822	88.111.967	7,21
Instrucciones Ejecutadas	96.059.192	84.208.787	12,34
IPC	1,012	0,956	
Fallos en L2	277	208	25,09
Referencias a memoria de datos	59.795.761	47.910.625	19,88
Fallos en L1	1.200	871	27,46

Tabla 16: Optimización de la función `SUMA()` por reducción del uso de una variable tipo `UINT8`.

La variable `suma` es de tipo `UINT8`. Todas las demás variables son de tipo `NUMERO`, y el campo de esa variable (`.N`) que aparece en este código es el del vector de elementos `UINT4` que almacenan los dígitos de los enteros a operar. `Byte4` es una macro cuyo valor es 32. El cambio que hacemos es muy simple, y los resultados que ofrece esa alteración se recogen en la Tabla 16. Simplemente introducimos una variable `UINT4`, que llamamos `C`, para que guarde el valor de `suma >> Byte4`. Así queda el bucle:

```
for( ; i < l ; )
{     C = suma >> Byte4;
      suma = C + (UINT8)*(a->N + i) + (UINT8)*(b->N + i);
      *(c->N + i++) = (UINT4)suma;
      C = suma >> Byte4;
      suma = C + (UINT8)*(a->N + i) + (UINT8)*(b->N + i);
      *(c->N + i++) = (UINT4)suma; }
```

La modificación es realmente simple, y salvo por los resultados que nos muestra la herramienta RABBIT, se podría pensar que además es inútil. En la Tabla 16 recogemos estos valores. Los ciclos de reloj se han reducido en un 7,21% y las instrucciones en un 12,34%. Comparando los demás registros de RABBIT antes y después del cambio vemos que los valores que han sufrido mayor modificación son los de acceso a memoria. Así quedan también recogidos en la Tabla 16: 20% menos de accesos a memoria; y más de un 25% de reducción en fallos a L1 y a L2.

7.6.2.4. Reemplazo de variables de tipo `unsigned short` por variables `unsigned long` en la función `EliminacionGaussiana()`.

La siguiente optimización que presentamos en este epígrafe de mejoras por cambio de tipo de dato es el cambio de **`unsigned short int`** (`UINT2`) a **`unsigned long int`** (`UINT4`) en la definición de las matrices necesarias para guardar los vectores de exponentes que forman la matriz de relaciones y las filas de la matriz histórica.

El motivo que inicialmente nos llevó a tomar esas matrices formadas por elementos de 16 bits (`UINT2`) fue procurar la optimización en el uso de la memoria: reservar la menor cantidad de espacio de memoria posible en la generación de las matrices, que llegan a ser de un tamaño bastante grande. El cambio a 32 bits (`UINT4`) supone un incremento en la velocidad lo suficientemente importante como para que merezca la pena ese posible "derroche" de memoria.

	v.02	v.03	% mejora
Ciclos de Reloj	1.188.999.698	694.463.499	41,59
Instrucciones Ejecutadas	55.102.886	29.532.113	46,41
IPC	0,046	0,043	

Tabla 17: Optimización en la función `EliminacionGaussiana()` por cambio de tipo de dato en el modo de definir las matrices de relaciones.

El cambio lo medimos en la función `EliminacionGaussiana()`, que es la que más veces recorre de columna en columna las dos matrices. Esta variación es el primer paso en nuestro proceso de optimización de esta función: paso de la v. 02 a la v.03 (de un total final de 9 versiones). Presentamos, en la Tabla 17, los datos de ambas versiones y comparamos valores.

La mejora no deja duda sobre la conveniencia del cambio. Y sin embargo los resultados que nos ofrece RABBIT nos exigen buscar nuevas versiones de la función: debemos localizar los motivos del valor tan bajo del IPC. A esta tarea van destinadas otras mejoras que presentaremos en los siguientes epígrafes y que logran al final los valores que mostramos en la Tabla 18, correspondiente a la versión .09 de la función `EliminacionGaussiana()`: un valor del IPC 8,46 veces mayor al de la versión v. 03 y una reducción de instrucciones del orden de 3,4 veces.

	v.09
Ciclos de Reloj	23.755.163
Instrucciones Ejecutadas	8.648.442
IPC	0,364

Tabla 18: Valores finales y cálculo del IPC de la función `EliminacionGaussiana()`.

La causa del valor del IPC en las versiones v. 02 y v.03 está en los accesos a memoria. Así podemos verlo en la Tabla 19, donde se recogen los valores de los índices de referencias a memoria y de fallos de acceso en las dos cachés. Como podemos observar, aunque el cambio presentado en este epígrafe ha supuesto una mejora muy considerable en el número de accesos a memoria (y esa es la causa fundamental de la reducción de instrucciones y de ciclos), seguimos manteniendo unos valores elevadísimos en estos tres registros.

	v.02	v.03	% mejora
Fallos en L2	3.886.537	2.403.867	38,15
Referencias a memoria de datos	45.991.579	25.599.156	44,34
Fallos en L1	9.700.184	4.676.690	51,79

Tabla 19: Valores de referencias a memoria y de fallos a la caché para la función `EliminacionGaussiana()` en la optimización por cambio de tipo de dato en el modo de definir las matrices de relaciones.

La cantidad de fallos obliga, en la ejecución de la función, a parar continuamente en espera de recibir los datos en el recorrido de matrices que ocupan varios Megabytes y que deben ser recorridas cientos de veces. Más adelante veremos cómo reducir estos valores.

Una observación final a las mejoras introducidas por cambio de tipo de dato. No nos parece evidente en un principio el enorme peso que suponía trabajar con variables de tipo `UINT8`, y la mejora que ha supuesto pasar a trabajar con variables de tipo `UINT4`. Menos evidente nos parecía la ventaja que íbamos a lograr cambiando de `UINT2` a `UINT4`. La herramienta RABBIT no sólo nos confirma que se ha mejorado, sino que además nos ofrece pistas para adivinar las causas de estas mejoras.

7.6.3. Cambios en las estructuras de programación. Ejemplo: función `longitud()`.

El ejemplo más claro de reducción de instrucciones por cambio en la estructura de programación lo encontramos en la función `longitud()`, que es la encargada de tener siempre actualizados los valores de los campos `B` y `T` de la estructura `NUMERO`: número de bits y número de dígitos en base 2^{32} empleados para la codificación del entero que recibe la función como parámetro.

La modificación que hemos realizado sobre la función alarga en mucho su código: para el cálculo del número de bits que ocupa la cifra codificada en la variable `NUMERO` se pasa de tres líneas de código a una cantidad muy superior. El cambio consiste en sustituir una estructura **`while`** por 32 **`if`** y 32 **`else`** concatenados y anidados hasta un nivel de 5. Con el nuevo procedimiento toda medida de longitud exige invariablemente 5 evaluaciones de sentencias condicionales; en la primera versión el número de evaluaciones de salto dependía de la longitud que tuviera la cifra, pero por término medio debería ser de 16 saltos. A pesar de aumentar notablemente el número de líneas del programa, queda sustancialmente reducido el número de instrucciones ejecutadas; a cambio, veremos, con los datos que nos aporta la herramienta RABBIT, cómo aumentan en mucho las instrucciones de salto y los errores de predicción.

La versión v. 00 de la función `longitud()` es la siguiente:

```
void longitud(NUMERO*n)
{     n->T = n->D;
      while(*(n->N + n->T - 1) == 0 && n->T != 0) (n->T)--;
      n->B = Byte4 * n->T;
      if(n->B)
      {     UINT4 Test = 0x80000000;
            while(!(*(n->N + n->T - 1) & Test))
            {     Test >>= 1;
                  n->B--; } } }
```

La versión v. 01 de la función `longitud()` es la siguiente:

```
void longitud(NUMERO*n)
{
      UINT4 N;
      n->T = n->D;
      while(*(n->N + n->T - 1) == 0 && n->T != 0) (n->T)--;
      if(!n->T)
      {     n->B = 0;
            return;}
```

```
N = *(n->N + n->T - 1);
if(0xFFFF0000 & N)
{     if(0xFF000000 & N)
      {     if(0xF0000000 & N)
            {     if(0xC0000000 & N) n->B = (0x80000000 & N) ? 32 : 31;
                  else n->B = (0x20000000 & N) ? 30 : 29;}
            else
            {     if(0x0C000000 & N) n->B = (0x08000000 & N) ? 28 : 27;
                  else n->B = (0x02000000 & N) ? 26 : 25;}}
      else
      {     if(0x00F00000 & N)
            {     if(0x00C00000 & N) n->B = (0x00800000 & N) ? 24 : 23;
                  else n->B = (0x00200000 & N) ? 22 : 21;}
            else
            {     if(0x000C0000 & N) n->B = (0x00080000 & N) ? 20 : 19;
                  else n->B = (0x00020000 & N) ? 18 : 17;}}}
else
{     if(0x0000FF00 & N)
      {     if(0x0000F000 & N)
            {     if(0x0000C000 & N) n->B = (0x00008000 & N) ? 16 : 15;
                  else n->B = (0x00002000 & N) ? 14 : 13;}
            else
            {     if(0x00000C00 & N) n->B = (0x00000800 & N) ? 12 : 11;
                  else n->B = (0x00000200 & N) ? 10 : 9;}}
      else
      {     if(0x000000F0 & N)
            {     if(0x000000C0 & N)n->B = (0x00000080 & N) ? 8 : 7;
                  else n->B = (0x00000020 & N) ? 6 : 5;}
            else
            {     if(0x0000000C & N) n->B = (0x00000008 & N) ? 4 : 3;
                  else n->B = (0x00000002 & N) ? 2 : 1;}}}}
```

Obtenemos una reducción notable de instrucciones, como se observa en los datos de la Tabla 20.

	v.00	v.01	factor de reducción
Ciclos de Reloj	177.617.505	61.331.944	2,90
Instrucciones Ejecutadas	161.769.933	51.930.170	3,12
IPC	0,911	0,847	

Tabla 20: Valores de la optimización en la función `longitud()` al sustituir una sentencia **`while`** por 32 sentencias **`if – else`**.

Podemos observar en la Tabla 21, donde se recoge el resto de los valores de los registros antes y después de la variación de código presentada, que la reducción de instrucciones supone una reducción enorme en los accesos a memoria y en las instrucciones de salto; se incrementa sin embargo notablemente los fallos de predicción de salto, lo que resulta lógico al haber sustituido el proceso de una sola condición recurrente a una sucesión de sentencias **`if`** todas ellas equiprobables. El aumento en los fallos a L1 y L2 es de unos pocos enteros en términos absolutos, y no merece mayor consideración, a pesar de que en términos relativos parezca que tenemos un

aumento importante.

	v.00	v.01	factor de reducción
Fallos en L2	55	152	
Referencias a memoria de datos	147.604.990	18.924.346	7,80
Fallos en L1	133	160	
Instr. de Salto ejecutadas	31.418.030	14.152.128	2,22
Inst. de Salto equivocadas	2.369.500	2.576.826	
Saltos tomados ejecutados	25.533.396	7.664.232	3,33
Saltos tomados mal pred. ejec.	813.476	1.576.163	

Tabla 21: Valores de los demás registros de la función `longitud()` obtenidos mediante la herramienta RABBIT.

7.6.4. Eliminación de llamadas a funciones (in – line functions).

7.6.4.1. Eliminación de llamadas en la función `Suave_S()`.

Un ejemplo muy claro de optimización por eliminación de llamadas a funciones está en la función `Suave_S()`. En el proceso de determinación de la suavidad de un valor `Ci`, si tenemos que un determinado primo de la base divide exactamente al valor actual `Ci` que estamos estudiando, entonces se ha de efectuar la operación `Cociente()` entre `Ci` y ese primo. La diferencia de código de la función `Modulo()` a la función `Cociente()` es muy reducida, y casi todos los pasos del algoritmo son idénticos. En nuestro proceso de factorización de nuestro número de prueba la función `Cociente()` es llamada 145.303 veces. Si logramos aprovechar los pasos de la función `Modulo()` para estas llamadas a la función `Cociente()`, entonces también reduciremos instrucciones. Estos son los razonamientos que nos han llevado a incluir el código de las dos funciones `Modulo()` y `Cociente()` dentro de la función `Suave_S()`. Y podemos comprobar (en los Anexos al final de la Tesis), por eso, que el número de llamadas a estas dos funciones ha quedado reducido a cero.

Estas mejoras presentadas aquí sobre la función `Suave_S()` son las recogidas en nuestras optimizaciones de código de esa función desde la versión v.00 hasta la v.035. Los datos de ambas versiones y los porcentajes de mejora han quedado recogidos en la Tabla 22.

Hay un análisis de los tiempos de esta función que creemos que es muy ilustrativo para mostrar el peso de las funciones `Modulo()` y `Cociente()` dentro de ella. Es interesante juntar los datos de los ciclos de reloj de las funciones `Suave_S()`, `Cociente()` y `Modulo()` antes de comenzar cualquier proceso de optimización. Quedan recogidos en la Tabla 23.

	v.00	v.035	% mejora
Ciclos de Reloj	2.990.925.687	2.554.343.340	14,60
Instrucciones Ejecutadas	3.005.315.496	2.446.082.379	18,61
IPC	1,005	0,958	

Tabla 22: Valores de la optimización en la función `Suave_S()` al insertar el código de las funciones `Modulo()` y `Cociente()`, y eliminar con ello las llamadas a esas funciones y algunas otras instrucciones.

Hay que tener en cuenta que después de la versión .02 de la función `Suave_S()`, donde quedan insertados los códigos de las funciones `Cociente()` y `Modulo()`, las llamadas a estas dos funciones queda reducidas a cero: todas las llamadas que recibían venían de la función `Suave_S()`. Y del análisis de la Tabla 23, que presentamos, queda patente que la tarea de la función `Suave_S()` consiste, esencialmente, en juntar las llamadas de las dos funciones eliminadas. La suma de ciclos de reloj que suponen las funciones `Cociente()` y `Modulo()` representa el 98,2% de los ciclos totales de la función `Suave_S()`. Tan solo la inserción del código de estas dos funciones (y por tanto la eliminación de todas sus llamadas: ver Tabla 22) representa como vemos una reducción del 14% de los ciclos de reloj de la función `Suave_S()`. Viendo las llamadas que recibe la función `Modulo()` en la factorización de nuestro entero de prueba, nos damos cuenta que evitar todas esas llamadas puede suponer un ahorro de trabajo para el ordenador.

		ciclos de reloj	llamadas
(1)	Suave_S()	3.069.441.708	41.620
(2)	Modulo()	2.395.473.808	14.641.005
(3)	Cociente()	618.329.800	145.303
(4)	(2) + (3)	3.013.803.608	14.786.308
(5)	(1) - (4)	55.638.100	

Tabla 23: Ciclos de reloj de la función `Suave_S()` y las dos funciones que intervienen en ella. También quedan recogidas las llamadas de cada función.

7.6.4.2. Eliminación de llamadas en las funciones `COCIENTE()` y `MODULO()`.

Otro ejemplo de cómo hemos reducido instrucciones lo tenemos en el paso de la versión v. 01 a la versión v. 03 en la función `COCIENTE()`. Esta función es invocada 83.123 veces en la factorización de nuestro número de prueba de 100 bits. Los datos de los registros que mostremos de esta función son, como en todos los casos que hemos mostrado, los acumulados de todas esas llamadas.

La reducción de instrucciones la encontramos, en esta función, en las versiones v.02 y v.03. En la versión v.02 eliminamos una llamada a la función que desplaza a la derecha todos los bits de un vector de la estructura `NUMERO`. En la v.03 eliminamos una llamada a la función `longitud()`. Estas modificaciones suponen no obtener toda la información que ofrecían las funciones que han dejado de ser llamadas; a cambio hemos tenido que introducir parte del código de estas funciones anuladas dentro del código de la función `COCIENTE()`. Ambas reducciones han supuesto una cierta pérdida de información: al no llamar a la función `DESPL_dcha()` una vez (modificación de la versión v. 02), perdemos el valor final del resto de la división. Por otro lado, al eliminar la llamada a la función `longitud()` perdemos la información sobre el valor del campo `T` de la estructura `NUMERO`, que obtenemos hasta llegar al final de toda la operación.

Las funciones que han sido eliminadas, y cuyo código ha sido insertado en la función `COCIENTE()` casi en su totalidad, realizaban unas pocas instrucciones más que las estrictamente necesarias para el cálculo que realiza la función `COCIENTE()`. En un lenguaje de programación estructurada es lógico invocar a las dos funciones, pues con ellas logramos de forma simplificada implementar la operación de la función. Pero cuando queremos eliminar instrucciones, si la función que optimizamos es invocada repetidas veces, como es el caso, puede resultar conveniente (y así lo hemos hecho) perder claridad en el código a cambio de eliminar unas pocas instrucciones de una y otra función. En la Tabla 24 podemos ver la reducción de instrucciones lograda con estas modificaciones.

	v.01	v. 03	% mejora
Ciclos de Reloj	177.427.139	150.387.966	15,24
Instrucciones Ejecutadas	199.842.944	169.171.979	15,35
IPC	1,126	1,125	

Tabla 24: Valores de la optimización en la función `COCIENTE()` al sustituir algunas llamadas a funciones.

Como se ve, la modificación se reduce a la eliminación de instrucciones de C, lo que ha llevado consigo una reducción de los ciclos de reloj (de más del 15%) completamente lineal con respecto a la reducción de instrucciones ejecutadas (también de más del 15%): El valor del IPC permanece constante después de estas modificaciones.

Pasos similares se han podido seguir en la optimización de la función `MODULO()` que, sustancialmente, es casi idéntica a la analizada. No mostramos ahora aquí los pasos dados en esta optimización, que son los mismos que los presentados para el caso de la función `COCIENTE()`.

7.6.5. Reuso de instrucciones.

Esta técnica consiste evitar que una operación sea ejecutada repetidas veces en una misma función, almacenando, por ejemplo, el valor de la operación en una variable creada a tal fin.

7.6.5.1. Reuso de instrucciones en la función `Suave_S()`.

Mostramos un ejemplo de optimización por reuso en la función `Suave_S()`. Aplicamos la técnica del reuso en el paso de la versión que ha sido llamada v. 035 a la versión v. 036. Hemos visto un poco más arriba el peso que tienen las dos funciones `Modulo()` y `Cociente()` dentro de la función `Suave_S()`. Y vemos que la función `Modulo()` es llamada un 99.02% del total de las llamadas recibidas entre las dos funciones. Únicamente se llama a la función `Cociente()` cuando la función `Modulo()` obtiene el valor de resto igual a cero.

	v. 035	v. 036	% mejora
Fallos en L2	1.576	1.794	
Referencias a memoria de datos	1.688.055.567	1.806.562.438	
Fallos en L1	130.864	217.920	
Ciclos de reloj	2.554.343.340	2.519.620.540	1,38
Instrucciones ejecutadas	2.446.082.379	2.309.274.577	5,92

Tabla 25: Reuso al emplear los valores intermedios obtenidos con el código correspondiente a la función `Modulo()` para los cálculos del código correspondiente a la función `Cociente()`.

	v. 036	v. 05	% mejora
Fallos en L2	1.794	1.453	23,47
Referencias a memoria de datos	1.806.562.438	1.546.880.594	16,79
Fallos en L1	217.920	116.535	87,00
Ciclos de reloj	2.519.620.540	2.325.219.870	8,36
Instrucciones ejecutadas	2.309.274.577	2.116.948.190	9,09

Tabla 26: Algunas modificaciones para corregir el aumento en los accesos a memoria que ha supuesto aplicar el reuso en la función `Suave_S()`.

Pero aunque el número de llamadas a la función `Cociente()` es proporcionalmente muy inferior al de la función `Modulo()`, en términos absolutos la función es invocada 145.303 veces. Y teniendo en cuenta que casi todos los pasos y valores intermedios que obtiene la función

`Modulo()` son los mismos que luego deberá obtener la función `Cociente()`, hemos estudiado la posibilidad de guardar todos esos valores intermedios producidos en la ejecución del código correspondiente a las instrucciones de lo que era antes la función `Modulo()` para así ahorrar trabajo en la parte del código de la función `Suave_S()` que contiene las instrucciones de lo que antes era la función `Cociente()`.

En la Tabla 25 se recogen los valores de los registros obtenidos con la herramienta RABBIT en la versión previa a esta mejora (v. 035) y en la mejora actual (v. 036). Las mejoras no son especialmente notables, pues aunque se reduce en un 6% el número de instrucciones sólo disminuye en poco más del 1% los ciclos de reloj. La causa de esta disminución del IPC está en que ha aumentado las referencias a la memoria y los fallos a L1 y a L2. En las versiones v. 04 y v. 05 se corrigen esos incrementos creando, para los valores intermedios que estamos almacenando para su reuso, un vector e introduciendo punteros para el recorrido de ese vector. En la Tabla 26 presentamos los datos comparativos de la versión v. 036 con los de la versión v. 05 (que es la que corresponde a las dos modificaciones señaladas).

Finalmente recogemos en la Tabla 27 los valores de los registros de las versiones previa y última después de refinar el reuso.

	v. 035	v. 05	% mejora
Fallos en L2	1.576	1.453	8,47
Referencias a memoria de datos	1.688.055.567	1.546.880.594	9,13
Fallos en L1	130.864	116.535	12,30
Ciclos de reloj	2.554.343.340	2.325.219.870	9,85
Instrucciones ejecutadas	2.446.082.379	2.116.948.190	15,55

Tabla 27: Resumen de las Tablas 25 y 26.

Podemos verificar que, en ese caso, el reuso ha resultado de gran valor, porque ha reducido los ciclos de reloj y se han reducido todos los accesos a memoria.

En este caso presentado se consigue una reducción de ciclos bastante significativa. En casi todas las funciones se recoge algún ejemplo de reuso, y se logran mejoras, no tan grandes, pero sí interesantes. Presentamos una ejemplo, tomado de la función `EliminacionGaussiana()`.

7.6.5.2. Reuso de instrucciones en la función `EliminacionGaussiana()`.

Es el paso de la versión v. 03 a la versión v. 04. en el código de la función se tiene (simplificado) la siguiente secuencia de instrucciones:

```
for(cbit = 0 ; cbit < Byte4 ; cbit++)
      for(f = 0 ; f < nr ; f++)
            if(C1 & (Test >> cbit)) I1;
```

donde `C1` es una expresión que se evaluará como verdadera o falsa, y las instrucciones `I1` e `I2` no importa ahora en qué consisten. El caso es que la variable `cbit` recorre los valores desde `0` hasta `31` (`Byte4` es igual a `32`) y la variable `f` varía desde `0` hasta `1910` (la variable `nr` vale `1911`). Por lo tanto, la operación `Test >> cbit` se evalúa, dentro del **`for`** regido por la variable `f` `1910` veces; y en todas ellas no varía ni `Test` ni `cbit`.

La versión .04 no hace otra cosa que introducir una nueva variable auxiliar, que almacena el valor de la operación de desplazamiento a derecha. Así, esa operación pasa de hacerse `32 * 1910` veces a hacerse tan solo `32` veces: tantas como valores diferentes adopta la variable `cbit`.

Con este sencillo cambio, los valores de los registros de la versión v. 03 y de la versión v. 04 son los recogidos en la Tabla 28.

	v. 03	v. 04	% mejora
Referencias a memoria de datos	25.599.156	24.294.912	5,09
Ciclos de reloj	694.463.499	684.339.509	1,46
Instrucciones ejecutadas	29.532.113	27.907.010	5,50

Tabla 28: Valores de optimización por reuso en la función `EliminacionGaussiana()`.

7.7. TERCERA MEDIDA: TÉCNICAS PARA REDUCIR INSTRUCCIONES DE SALTO.

7.7.1. Eliminación de un bucle **`while`** en la función `PROD_bit()`.

En la v. 00 de la función `PROD_bit()` podíamos encontrar un bucle parecido al que antes hemos mostrado en la función `longitud()`. La función `PROD_bit()` es la que realiza el producto de dos enteros grandes (de dos valores de estructura `NUMERO`). Mediante una variable llamada `Test` de 32 bits a la que se le asigna un valor inicial con un 1 en el bit menos significativo y un cero en los 31 bits restantes (`Test = 0x00000001;`) se recorre uno de los dos multiplicandos (el más corto de los dos) en busca de dígitos 1 en su codificación (el proceso de esta operación viene explicado en el Capítulo 4). Y cada vez que se encuentra un 1 se realiza un desplazamiento sobre la variable que va almacenando el valor del producto y una suma con el otro multiplicando. El valor de la variable `Test` varía cada vez mediante la operación desplazamiento a izquierda (`Test <<= 1;`). Cuando se llega a un valor `Test` igual a cero la variable vuelve a inicializarse con el valor `0x00000001` y se vuelve a comenzar sobre el siguiente elemento `UINT4` del multiplicando que estamos recorriendo en busca de unos.

El código de esta estructura de iteración **`while`** tenía la forma

```
while(Test)
{     if(digito & Test) I1;
      else I2;
      Test <<= 1; }
```

Así descrito, el proceso queda gobernado con una estructura **while**: (**while**(Test)). La modificación que introducimos con la versión v. 01 consiste en desenrollar ese **while** y eliminar la variable Test y sus desplazamientos, escribiendo el código mediante sentencias **if-else**, tomando cada uno de los 32 valores que adopta la variable Test (que, como se ve, es una variable de 32 bits, siempre con un bit a 1 y los otros 31 bits a 0). Es decir, por un lado eliminamos el operador desplazamiento a izquierda; por otro lado sustituimos una estructura **while**, la gobernada por la variable Test por una secuencia de sentencias condicionales. El código queda (de forma abreviada) de la siguiente manera:

```
if(digito & 0x00000001) I1;
else I2;
if(digito & 0x00000002) I1;
else I2;
```

(... seguimos con todos los valores a operar con dígito...)

```
if(digito & 0x40000000)I1;
else I2;
if(digito & 0x80000000) I1;
else I2;
```

En este caso hemos desenrollado completamente el bucle **while**.

Con esta modificación de código hemos llegado a los siguientes valores de los registros, recogidos en la Tabla 29 (la función ha sido llamada 249.043 veces en nuestro proceso de factorización; los valores que presentamos son los acumulados en el total de todas las llamadas).

	v.00	v.01	% mejora
Ciclos de Reloj	293.176.344	288.823.740	1,48
Instrucciones Ejecutadas	329.045.303	295.975.423	10,05
IPC	1,122	1,025	

Tabla 29: Valores de la optimización por eliminación de un bucle **while** en la función PROD_bit().

Esta modificación ha reducido notablemente el número de instrucciones. Lamentablemente, el número de ciclos ha bajado muy poco, y el valor del IPC ha bajado una décima. La causa de este decremento en el índice IPC podemos encontrarla en que en nuestra modificación de código ha ocasionado un notable incremento en los saltos tomados mal predichos y ejecutados: un incremento de casi un 30%. Estos datos pueden verse en la Tabla 30.

Como cabía esperar, al haber realizado un desenrrollamiento de bucle, han disminuido las

instrucciones de salto ejecutadas (en un 12,5%) y los saltos tomados ejecutados (en un 15,52%).

	v. 00	v. 01	% mejora
Fallos en L2	1.097	1.770	
Referencias a memoria de datos	185.496.483	177.135.689	4,51
Fallos en L1	2.000	1.807	9,65
Instrucciones de salto ejecutadas	63.738.955	55.739.022	12,55
Instr. de salto equivocadas predichas	3.767.025	3.786.235	
Saltos tomados ejecutados	49.934.735	42.184.803	15,52
Saltos tomados mal predichos ejecutados	1.831.097	2.349.885	

Tabla 30: Resto de registros, que complementan la información presentada en la Tabla 29.

7.7.2. Desenrollando de una estructura `for` en la función `SUMA()`.

Presentamos ahora otra optimización por reducción de saltos mediante la técnica de desenrollado de bucles. Esta vez lo veremos en la función definida para calcular la suma de dos valores enteros grandes (función `SUMA()`).

En la versión v.03 de la función `SUMA()` procedemos a desenrollar los bucles. La función `SUMA()` recorre ambos sumandos, realizando la suma de los dígitos de igual peso y arrastrando el acarreo de la suma de los dígitos previos. El proceso va sumando dígitos (en base 2^{32}) del mismo peso mediante una estructura **`for`**, comenzando por los dos dígitos más a la derecha (los posicionados en la dirección +0 en cada puntero donde se guardan los dígitos de cada sumando) y continuando sucesivamente por todos los demás dígitos cada vez más a la izquierda, teniendo en cuenta en todos esos casos el valor del acarreo de la suma de dígitos inmediatamente anterior. La estructura **`for`** es la encargada de decidir qué dígito se suma cada vez, y de decidir cuándo se dejan de realizar las sumas. Mediante un contador `i`, va en cada nueva iteración sumando los siguientes dígitos `+i` a partir de los campos `N` tipo puntero a entero de cada una de las dos estructuras `NUMERO` que estamos operando.

Una forma de desenrollar el bucle **`for`** consiste en que, dentro de cada iteración, realice las operaciones para los dígitos `i` y para los dígitos `i + 1` de cada uno de los dos sumandos. Para realizar estas modificaciones es necesario una tarea previa que consiste en verificar que el número de dígitos a sumar es par y, en caso de que no sea así, realizar una suma previa de los primeros dígitos y entrar ya luego en la estructura **`for`** con esta nueva implementación. A partir de ese momento, evidentemente, el incremento de la variable `i` en de la estructura **`for`** será `i += 2` y no `i++` como estaba hasta este momento.

En el caso de factorización de nuestro entero de prueba de 100 bits el tamaño del mayor de los sumandos es casi siempre menor o igual que cuatro, y en una abrumadora mayoría, los sumandos son de tamaño par. La función `SUMA()` ha sido invocada 600.084 veces. Los tamaños del mayor de los sumandos (que es quien decide la continuidad en la estructura **`for`**) en cada una de esas llamadas a la función vienen recogidos en la Tabla 31.

tamaño del entero	enteros de ese tamaño
1	969
2	373.795
3	10.696
4	200.232
5	3.577
6	3.654
7	568
8	133
9	140
10	133
más de 10	6.187

Tabla 31: Tamaños del mayor de los enteros en cada una de las sumas de nuestro proceso de factorización.

Los datos de los registros en nuestro proceso quedan recogidos en la Tabla 32 (que recoge los valores acumulados de todas las llamadas a la función `SUMA()` en nuestro proceso de factorización: 600.084 sumas).

	v.02	v.03	% mejora
Ciclos de Reloj	100.338.203	94.956.822	5,36
Instrucciones Ejecutadas	96.359.915	96.059.192	0,31
IPC	0,960	1,012	

Tabla 32: Valores de la optimización por "desenrrollamiento" de un bucle **`for`** en la función `SUMA()`.

Apenas hemos reducido instrucciones (un decremento del 0,31%), pero ha mejorado el IPC y los ciclos han sufrido una reducción del 5,36 %. Podemos ver los demás valores de los registros de esta mejora en la Tabla 33, que explican la mejora del IPC. En este caso el objetivo perseguido ha sido la reducción de instrucciones de salto. Esa reducción no ha sido muy notable (tampoco ha sido muy significativo el cambio que hemos realizado sobre la implementación de la función), pero sí suficiente como para aumentar el IPC. Además se han reducido en un porcentaje

significativo los accesos a memoria (aunque sólo sea porque se ha reducido aproximadamente a la mitad el número de incrementos sobre la variable `i`)

	v.02	v.03	% mejora
Fallos en L2	211	277	
Referencias a memoria de datos	64.125.759	59.795.761	6,75
Fallos en L1	2.438	1.200	50,78
Instr. de Salto ejecutadas	14.325.564	14.055.749	1,88
Inst. de Salto equivocadas	2.773.260	2.671.371	3,67
Saltos tomados ejecutados	8.981.133	8.664.534	3,53
Saltos tomados mal pred. ejec.	1.413.572	1.277.571	9,62

Tabla 33: Valores de los demás registros de la función `SUMA()` obtenidos mediante la herramienta RABBIT.

7.7.3. Eliminación de instrucciones de salto en una iteración de `EliminacionGaussiana()`.

Un ejemplo de reducción masiva de instrucciones de salto lo presenta la versión v.08 de la función `EliminacionGaussiana()`. Durante el proceso de optimización de esta función hemos ido logrando paulatinamente una reducción de las instrucciones ejecutadas; pero ha sido una reducción que no ha ido pareja con un descenso en el número de ciclos de reloj invertidos en el proceso. Y en cada nueva versión, cada vez ha sido menor el valor del IPC. Volveremos a ver este estancamiento en el ejemplo que ahora presentamos, donde el descenso de casi un 60% en las instrucciones a ejecutar lleva consigo tan solo un descenso de algo menos del 5% en los ciclos de reloj. La versión v.09 remedia el problema, dando a la función `EliminacionGaussiana()` una mejora muy sustancial. Esa última versión la presentamos en el epígrafe de optimización de accesos a memoria: el problema principal de la función `EliminacionGaussiana()` es que debe recorrer dos matrices, una de las cuales alcanza casi los 2 Mbytes en ocupación de memoria (para el caso de un entero a factorizar de 100 bits). Optimizar ese acceso será la clave para alcanzar valores del IPC más altos.

En el proceso de eliminación Gaussiana se debe recorrer una y otra vez, fila a fila, tanto la matriz de vectores o de relaciones como la matriz histórica. Cada vez que se accede a una nueva fila, y antes de hacer algo con ella, se debe verificar si esa fila está validada o ya ha sido anulada (para una explicación del proceso de la función `EliminacionGaussiana()` puede consultarse el Capítulo 6 de esta Tesis). La mejora que plantea la versión v. 08 consiste en crear un vector para recoger los índices de filas válidas, de forma que ya no se recorran las dos matrices de fila en fila, secuencialmente, sino que se recorran únicamente sobre aquellas filas que estén referenciadas en

ese nuevo vector. En la Tabla 34 recogemos los valores de los ciclos de reloj y del número de instrucciones ejecutadas. Y como señalábamos, logramos una notable disminución de instrucciones, pero también una caída en el valor del IPC (no llega a las dos centésimas de instrucción por ciclo).

	v.07	v.08	% mejora
Ciclos de Reloj	490.729.374	467.198.540	4,80
Instrucciones Ejecutadas	19.548.296	8.334.272	57,37
IPC	0,040	0,018	

Tabla 34: Optimización de la función `EliminacionGaussiana()` por reducción de accesos a las filas de las matrices de relaciones e histórica.

En la Tabla 35 mostramos los valores de los demás registros obtenidos con RABBIT antes y después de esta optimización. Podemos destacar dos cosas de esos valores: primero que aumentan en mucho los saltos tomados mal predichos ejecutados (casi en un 50%). Pero lo más destacable es el valor enorme que sigue teniendo los fallos a L1 y los fallos a L2. Como ya hemos señalado antes esas deberán ser las optimizaciones que persigamos finalmente con esta función. En esta nueva versión hemos logrado una reducción de accesos a memoria de más del 17% pero sólo una reducción de fallos a L1 del 2% y de fallos a L2 de algo menos del 4%. Como ya hemos dicho, la v. 09 remedia este desequilibrio: lo veremos al presentar la quinta media de optimización de accesos a memoria.

	v.07	v.08	% mejora
Fallos en L2	1.691.073	1.624.883	3,91
Referencias a memoria de datos	11.591.894	9.590.728	17,26
Fallos en L1	2.147.866	2.104.883	2,00
Instr. de Salto ejecutadas	5.485.774	1.108.013	79,80
Inst. de Salto equivocadas	131.272	59.024	55,04
Saltos tomados ejecutados	4.465.644	822.463	81,58
Saltos tomados mal pred. ejec.	13.489	19.713	

Tabla 35: Valores de los registros sobre las instrucciones de salto de la función `EliminacionGaussiana()` obtenidos mediante la herramienta RABBIT y correspondientes al proceso de optimización recogido en Tabla 34.

7.8. CUARTA MEDIDA: EVITAR DEPENDENCIA DE DATOS.

7.8.1. Dependencia de datos en la función SUMA().

Ya hemos visto en epígrafes anteriores algunas modificaciones realizadas sobre la operación SUMA(). Tanto por evitar en lo posible el uso de variables UINT8 como por desenrrollamiento de bucles. Otra ligera mejora la hemos logrado en el intento de evitar dependencias de datos. El código, como vimos quedaba de la siguiente forma:

```
for( ; i < l ; )
{     C = suma >> Byte4;
      suma = C + (UINT8)*(a->N + i) + (UINT8)*(b->N + i);
      *(c->N + i++) = (UINT4)suma;
      C = suma >> Byte4;
      suma = C + (UINT8)*(a->N + i) + (UINT8)*(b->N + i);
      *(c->N + i++) = (UINT4)suma; }
```

Existe una posible dependencia de datos en el uso de la variable suma, que aparece en todas las líneas del código. Gracias al desdoblamiento del bucle podemos ahora reducir esa dependencia, definiendo dos variables distintas, suma1 y suma2. El código quedaría así:

```
for( ; i < l ; )
{     C = suma1 >> Byte4;
      suma1 = C + (UINT8)*(a->N + i) + (UINT8)*(b->N + i);
      *(c->N + i++) = (UINT4)suma1;
      C = suma2 >> Byte4;
      suma2 = C + (UINT8)*(a->N + i) + (UINT8)*(b->N + i);
      *(c->N + i++) = (UINT4)suma2; }
```

El cambio de los valores de los registros es pequeño pero significativo, como se aprecia en la Tabla 36. Se ha mantenido constante el número de instrucciones ejecutadas y ha disminuido en un 2,45 % el número de ciclos de reloj. Ha mejorado el valor de índice IPC. Se mantiene en el mismo valor el registro de Instrucciones de salto ejecutadas, y aumenta ligeramente el número de fallos a L1 y a L2.

	v.03	v.04	% mejora
Ciclos de Reloj	88.111.967	85.954.467	2,45
Instrucciones Ejecutadas	84.208.787	84.208.766	0,00
IPC	0,956	0,980	

Tabla 36: Optimización de la función SUMA() por introducción de nuevas variables que reducen la dependencia de datos.

7.8.2. Dependencia de datos en la función `Suave_S()`.

Otra función que ha logrado una optimización mediante la técnica de eliminar dependencia de datos ha sido la encargada de determinar la suavidad de cada valor `Ci` generado en la función `RelacionBhascara()`. La versión v.08 de la función `Suave_S()` combina la técnica de desenrollado de bucle y de creación de nueva variable para evitar dependencias de datos.

La función `Suave_S()` toma un valor `Ci` y busca entre todos los primos de la base de factores aquellos que dividen a `Ci`. En la versión v. 07 se recorre la base de factores mediante un puntero y se va guardando los sucesivos restos del proceso de división (como ya quedó explicado en el epígrafe donde se presentan ejemplos de optimización por reducción de instrucciones); si el último resto es cero, entonces se procede a realizar la división, partiendo de los valores intermedios de los restos ya calculados. La versión v.08 repite todo el proceso, pero lo realiza, en cada ciclo de la estructura de control, dos veces: una con el factor primo de la base de factores apuntado por el puntero y otra por el factor primo siguiente al apuntado por el puntero; y se guarda la cadena de sucesivos restos también en dos vectores distintos (es un proceso de desdoblamiento de bucle). Al duplicar las variables evitamos posibles dependencias de datos, pues el proceso de cálculo de los sucesivos restos siempre es tal que necesitamos el valor `i` para lograr hallar posteriormente el valor `i + 1`. Al trabajar con dos procesos en paralelo evitamos en parte esas esperas. Además reducimos el número de saltos pues ahora, en cada iteración de la estructura de control que va probando uno a uno todos los primos de la base de factores, se prueban dos primos en lugar de solo uno.

Hemos tomado valores de los registros de RABBIT también para el caso de realizar el proceso de tres en tres, o de cuatro en cuatro, etc. Los mejores tiempos los hemos obtenido para el caso de que en cada iteración se analizasen sólo dos primos. Los valores de los registros obtenidos quedan recogidos en la Tabla 37.

	v. 07	v. 08	% mejora
Ciclos de Reloj	1.859.452.244	1.695.120.940	8,84
Instrucciones Ejecutadas	686.964.141	566.432.835	17,55
IPC	0,369	0,334	

Tabla 37: Optimización de la función `Suave_S()` por introducción de nuevas variables que reducen la dependencia de datos y "desenrrollamiento" de bucle.

7.9. QUINTA MEDIDA: OPTIMIZAR LOS ACCESOS A MEMORIA.

La meta de reducir los accesos a memoria ha sido la que mejor ha logrado liderar procesos de

optimización, y la que más resultados ha cosechado: si no en cantidad, desde luego sí en calidad.

Algunas de las ventajas de este proceso de optimización han quedado recogidas en los epígrafes anteriores. Es evidente que reducir los accesos a memoria es siempre rentable, y más aún reducir fallos de acceso a L1 o a L2.

Recogeremos la optimización en dos funciones: en la función `COCIENTE()` (de la misma manera ha quedado optimizada la función `MODULO()`), y en la función `EliminacionGaussiana()`.

7.9.1. Reducción de accesos a memoria en la función `COCIENTE()`.

Una descripción de la optimización de la función `COCIENTE()` viene presentada en [Patt00]. La idea que se recoge en ese libro, y que hemos utilizado para modificar el código de nuestra función `COCIENTE()` consiste en realizar todas las operaciones que conducen a la división en un sólo vector.

En la versión v. 00 tenemos una variable tipo `NUMERO` que almacena el valor que va tomando el cociente (variable que llamamos `coc`) y otra variable, también tipo `NUMERO`, que va guardando el valor que va tomando el resto (variable que llamamos `res`). Cada una de esas dos variables tienen su propio campo `.N`, puntero que recoge la dirección del vector donde se guardan las cifras (en base 2^{32}) de ambos números. En la versión v. 01 hemos definido dos variables `NUMERO` (`resto` y `res_coc`), pero en esta ocasión:

```
res_coc.D = 2 * Div->T;
resto.D = Div->T; y
resto.N = res_coc.N + Div->T;
```

es decir, ambas variables `NUMERO` comparten el mismo vector: el cociente irá en los `Div->T` elementos superiores del vector `res_coc.N` y el resto irá en los `Div->T` elementos inferiores de `res_coc.N`.

A la variable `res_coc` se le asigna espacio de memoria en el campo `N`: para esta asignación podemos usar la función ya presentada `CrearNumero()`. La dimensión de este vector será, como ya hemos señalado, `res_coc.D = 2 * Div->T`. A la variable `resto` se le asigna la dimensión `resto.D = Div->T`, es decir, la mitad de la dimensión asignada de `res_coc`. La diferencia ahora es que no asignamos memoria al campo `N` de la variable `resto` mediante la función `CrearNumero()` (es decir, no asignamos nueva memoria dinámica al puntero `N` de la variable `resto` con la función `malloc()`: tantos elementos `UINT4` como indique `resto.D`), sino que direccionamos este puntero a la mitad de la cadena de elementos `UINT4` que ha sido asignada al campo puntero `N` de la variable `res_coc`.

Para que esta modificación en la asignación de memoria pueda llevarse a cabo en nuestro algoritmo estamos obligados a realizar unas pequeñas modificaciones en el código de su implementación: por un lado pasamos de tener dos llamadas a la función `DESPL_izda()` en la

versión v. 00 a tener una sola en la versión v. 01; por otro lado la v. 01 incluye dos llamadas a la función `longitud()` (una de ellas condicionada mediante una estructura `if)` que no tenía la v. 00; la versión v.00, en lugar de esas llamadas a la función `longitud()`, tenía dos sentencias de asignación simple condicionadas con sentencias `if`. No podemos estudiar el comportamiento de la modificación de asignación de memoria de forma aislada, sin introducir esos cambios en las llamadas a diversas funciones: es exigencia necesaria para el correcto funcionamiento del proceso del cociente. Por ejemplo, era necesaria la introducción de las llamadas a la función `longitud()` porque cualquier modificación en el vector `res_coc.N` trae como consecuencia una variación en la longitud del vector `resto.N`; y cualquier variación en el vector direccionado por `resto.N` (que como ha quedado explicado corresponde a la mitad de los elementos del vector de `res_coc.N`) trae consigo una variación en la longitud de `res_coc`. Evitamos una llamada a la función `DESPL_izda()` porque ahora sólo tenemos que trabajar con un vector y no con dos como teníamos antes.

	v.00	v.01	factor de reducción
Ciclos de Reloj	425.853.638	177.427.139	2,40
Instrucciones Ejecutadas	510.600.606	199.842.944	2,55
IPC	1,199	1,126	

Tabla 38: Optimización de la función `COCIENTE()` por reducción de accesos a memoria, al unificar en un único vector lo que antes se almacenaba en dos vectores. El cambio ha propiciado también la reducción en el número de llamadas a alguna función auxiliar.

Los valores de los registros obtenidos mediante la herramienta RABBIT, antes y después de la optimización, están recogidos en las Tablas 38 y 39. En la Tabla 38 podemos comprobar el importante descenso en los ciclos de reloj y en el número de instrucciones ejecutadas. Los demás valores de los registros obtenidos gracias a la herramienta RABBIT quedan recogidos en la Tabla 39. La reducción en el número de accesos a memoria es muy significativa, y causa principal de la caída de valores en los registros mostrados en la Tabla 38. No sufren modificaciones importantes los fallos de acceso a L1 y L2. Los descensos en los registros que hacen referencia a las instrucciones de salto quizá estén motivados por la eliminación de dos sentencias condicionales `if` en la versión v. 01; de todas formas no podemos predecir de antemano estos valores, pues la introducción de las llamadas a la función `longitud()` implica muchas sentencias `if` en el código: como se ha visto antes, exactamente cinco sentencias condicionales en cada invocación a la función.

La misma mejora hemos logrado con la función `MODULO()`. En esta función los porcentajes de mejora son todavía mayores: se reduce en un factor 3,40 el número de accesos a memoria; se

reducen en un factor 3,99 las instrucciones ejecutadas; y en un factor 3,42 los ciclos de reloj. Las instrucciones de salto también ven una reducción en una factor de 3,00.

Las mejoras de ambas funciones suponen una optimización de gran peso en el proceso total de factorización.

	v.00	v.01	factor de reducción
Fallos en L2	1.283	667	1,92
Referencias a memoria de datos	308.928.062	117.584.844	2,63
Fallos en L1	4.901	4.523	1,08
Instr. de Salto ejecutadas	64.402.604	36.990.740	1,74
Inst. de Salto equivocadas	4.435.671	2.881.640	1,54
Saltos tomados ejecutados	35.310.895	23.060.057	1,53
Saltos tomados mal pred. ejec.	2.234.530	1.598.042	1,40

Tabla 39: Valor de los demás registros de la mejora recogida en la Tabla 38.

7.9.2. Reducción de accesos a memoria en la función `EliminacionGaussiana()`.

Pero el ejemplo más notable de mejora, causada por el intento de reducir los accesos a memoria y, sobre todo, los fallos en los accesos a las cachés, lo encontramos en la función `EliminacionGaussiana()`. Es esa una función que requería un número enorme de consultas a la memoria. Ya la hemos presentado en algunos epígrafes anteriores, donde llamábamos la atención sobre el llamativo valor del índice IPC que era extraordinariamente bajo. Ahora mostramos dos procesos de optimización de esta función originados por la reducción de accesos a memoria.

El primer paso de esta optimización viene presentado en la v. 02. Es un cambio bastante interesante, pues en él todos los índices empeoran excepto los dos que hacen referencia a los fallos de caché; y sin embargo el número de ciclos queda reducido en un 1,56 %.

Esta mejora consiste en lo siguiente. En las versiones previas, cada vez que se encontraba una fila que debía sufrir la operación XOR de todas sus columnas con todas las columnas de la fila de eliminación (en el Capítulo 6 vienen extensamente explicadas las operaciones que desarrolla esta función), se realizaba el proceso tanto para la matriz de relaciones como para la matriz histórica; en la versión .02 se duplica el proceso de decisión de cuáles son las filas sobre las que se debe realizar la operación XOR: primero se decide y se realiza la operación únicamente sobre la matriz histórica, y luego se vuelve a decidir sobre la matriz de relaciones.

Como veremos en el cuadro de valores de los registros (Tabla 40), se aumentan los accesos a

memoria y salen penalizados también todos los valores de salto; pero se reducen los fallos a caché (sobre todo a L1) y por esa causa disminuye el número de ciclos del proceso.

Aunque casi todo queda penalizado ganamos velocidad gracias a la reducción de fallos en L1 y L2. Evidentemente no supone esto una mejora sustancial (que sí lograremos en la versión v. 09), pero nos parece que es un ejemplo que muestra que lograr optimizar los accesos a memoria resulta muchas veces un camino acertado de optimización de código.

	v.00	v.02	% mejora
Fallos en L2	3.886.537	3.808.184	2,02
Referencias a memoria de datos	45.991.579	49.485.681	-7,60
Fallos en L1	9.700.184	8.970.631	7,52
Ciclos de Reloj	1.188.999.698	1.170.436.210	1,56
Instrucciones Ejecutadas	55.102.886	59.916.282	-8,74
IPC	0,046	0,051	
Instr. de Salto ejecutadas	7.751.129	9.088.784	-17,26
Inst. de Salto equivocadas	128.531	210.626	-63,87
Saltos tomados ejecutados	6.715.564	7.977.986	-18,80
Saltos tomados mal pred. ejec.	15.417	28.478	-84,72

Tabla 40: Optimización en la función `EliminacionGaussiana()` (versión v. 02) por reducción de fallos en el acceso a la memoria caché.

Veamos ahora la mejora presentada en la v. 09. Al igual que la anterior, todo el avance se centra en reducir fallos de acceso a la caché.

Contemplando los registros de la versión previa a la mejora que ahora presentamos (versión v. 09) observábamos cómo cada vez lográbamos reducir más todos los valores pero cada vez llegábamos a un índice IPC más bajo. En la Tabla 41, recogemos los valores comparados entre la versión v.02 que acabamos de comentar en las líneas previas y la versión inmediatamente anterior a la v. 09 que será la que ahora presentaremos y que logrará, felizmente, aumentar el valor del índice IPC hasta un rango más normalizado.

Sigue llamando la atención, en la última versión previa a la mejora (versión v. 08_bis) la cantidad enorme de fallos a L1 y a L2. Tenemos los accesos a memoria enormemente penalizados: de los 9,5 millones de accesos, en 1,6 millones sufrimos fallos en L1 y en 2,1 millones sufrimos fallos en el acceso a L2. Esta es claramente la causa de que tengamos un IPC tan bajo (0,018: lo que supone 56 ciclos por cada instrucción). Y a este objetivo de reducir drásticamente estos valores se ha dirigido la v. 09.

En la v. 09 se redefinen la matriz de relaciones y la matriz histórica. La modificación se ha llevado a cabo aumentando las líneas de código de la función y, de hecho, aumentan las instrucciones

ejecutadas. Presentamos sucintamente los pasos que se han intercalado en el código para redefinir las nuevas matrices.

	v. 02	v.08_bis
Fallos en L2	3.808.184	1.578.092
Referencias a memoria de datos	49.485.681	9.723.173
Fallos en L1	8.970.631	2.160.619
Ciclos de Reloj	1.170.436.210	467.198.540
Instrucciones Ejecutadas	59.916.282	8.334.272
IPC	0,051	0,018
Instrucciones de salto ejecutadas	9.088.784	1.113.378
Instr. de salto equivocadas predichas	210.626	59.457
Saltos tomados ejecutados	7.977.986	826.639
Saltos tomados mal predichos ejecutados	28.478	19.819

Tabla 41: Comparativa de valores entre la versión presentada en la Tabla 40 y los valores de la versión inmediatamente anterior a la v. 09, para la función `EliminacionGaussiana()`.

Contamos el número de filas válidas. Se crea entonces una nueva matriz de relaciones que tenga ese número de filas, y no el total inicial. Se crea también una nueva matriz histórica, más reducida en su número de filas. Ambas matrices deben mantener el número de columnas: las columnas de la matriz de relaciones deben ser tantas como primos tenga la base de factores; las columnas de la matriz histórica deben ser tantas como relaciones totales se han intentado buscar. El cuarto paso es copiar en estas dos nuevas matrices los valores de las filas válidas de las antiguas matrices. Se libera entonces la memoria de las matrices antiguas, y finalmente asignamos a los punteros de las antiguas matrices las direcciones que tienen ahora las nuevas matrices.

A partir de estas modificaciones el código de la función queda inalterado. El proceso seguirá buscando entre las filas válidas alguna que tenga un 1 en la posición de la columna que se esté explorando y una vez encontrada buscará entre todas las demás aquellas, también válidas, que tengan un 1 en esa columna y debamos, por tanto, realizar la operación XOR entre todas las columnas de ambas filas.

Esta modificación exige además cambios en otras funciones: en la función `OrdenFinal()` y en la función `ValidarRelaciones()`. Mostramos en la Tabla 42 el cuadro comparativo de valores de los registros en la versión previa y en la versión modificada.

Podríamos decir que la mejora es espectacular: se ha reducido el número de fallos a L2 en un factor de casi 30; en un factor de 2,48 los fallos a L1. Y, sin embargo, ha aumentado en casi un 15% la cantidad de accesos a memoria. También han aumentado en algo más de un 5% las

instrucciones de salto ejecutadas. Especial atención merece el comprobar el ligero aumento en el número de instrucciones ejecutadas (aumento de casi un 4%). Todos los valores de los registros aumentan excepto los fallos a L1 y a L2. Y gracias a estas dos caídas, desciende también en picado (un factor de casi 20) los ciclos de reloj del proceso. Volvemos a tener un valor del IPC en un orden aceptable (que ha aumentado en una factor superior a 22): no podíamos admitir que cada instrucción exigiera el consumo de 50 ciclos de reloj.

	v.08_bis	v.09	factor de reducción
Fallos en L2	1.624.883	54.388	29,88
Referencias a memoria de datos	9.590.728	11.007.177	
Fallos en L1	2.104.883	849.317	2,48
Ciclos de Reloj	467.198.540	23.755.163	19,67
Instrucciones Ejecutadas	8.334.272	8.648.442	
IPC	0,018	0,364	
Instrucciones de salto ejecutadas	1.108.013	1.168.534	
Instr. de salto equivocadas predichas	59.024	59.558	
Saltos tomados ejecutados	822.463	881.566	
Saltos tomados mal predichos ejecutados	19.713	19.828	

Tabla 42: Optimización en la función `EliminacionGaussiana()` (versión v. 09) por reducción de fallos en el acceso a la memoria caché.

Y toda esta mejora ha venido de la mano de la búsqueda de una reducción de los fallos a caché. El hecho de poder conocer los valores de estos índices nos ayuda a detectar el proceso que debemos seguir para la optimización de nuestro código. Y hemos llegado a una versión que trae más instrucciones, más saltos, más accesos a memoria, y reduce un factor de aproximadamente 20 los ciclos de reloj.

7.10. RESUMEN DE LAS OPTIMIZACIONES DE LAS PRINCIPALES FUNCIONES.

Presentamos ahora en forma gráfica las optimizaciones realizadas en las siete funciones que consideramos más importantes. Iremos presentando función a función, y reseñando de modo muy esquemático los pasos que se han dado en cada mejora. Algunas "mejoras" no han sido tales, y cuando hemos medido los parámetros después de haber redefinido la función nos hemos encontrado que el paso era hacia atrás. En esos casos hemos mantenido la numeración de la versión aunque luego estas versiones fuesen invalidadas. Posteriormente hemos seguido construyendo las nuevas versiones a partir de la última tomada válida.

Existen otras funciones que han sido optimizadas: las funciones `SUMA()`, `PROD_bit()`, `Erasthotenes()` y `BaseFactores_Ser()`. Son funciones de poco peso en el coste total de la aplicación, cuyas optimizaciones han quedado ya recogidas en los epígrafes anteriores. No son funciones en las que, teniendo en cuenta la ley de AMDHAL, hayamos centrado nuestro trabajo, ni hayamos dedicado mucho tiempo. En el anexo VI recogemos, en tablas, todas las mejoras de cada una de las funciones en cada uno de los 10 registros estudiados.

En estas tablas del anexo VI también podemos observar que los valores finales de los registros obtenidos con RABBIT para el estudio de cada función no coinciden siempre con los valores que hemos presentado en los gráficos de este epígrafe. Un caso bastante claro de esta diferencia lo tenemos por ejemplo en la función `RelacionBhascara()`. En la Gráfica 3 que se presenta más adelante podemos ver ahora cómo los valores finales de esta función logran reducirse mucho, debido a que cada función invocada por `RelacionBhascara()` ha sido posteriormente optimizada. Es el punto de la gráfica que llamamos v. final: recoge la optimización que ha quedado realizada en cada una de las funciones que son invocadas desde esa función `RelacionBhascara()`.

En las gráficas que se recogen en este epígrafe representamos únicamente los valores de los registros que relacionamos en la Tabla 43, representados en las sucesivas gráficas con los colores que en esa tabla se indican. No representamos los valores de Fallos a L1 y a L2, porque sus valores son siempre inferiores a los de los demás registros obtenidos con RABBIT, y su representación gráfica queda siempre casi superpuesta al eje de abcisas.

`Referencias a memoria de datos`	AZUL
`Ciclos de reloj`	ROJO, línea gruesa
`Instrucciones ejecutadas`	VERDE, línea gruesa.
`Instrucciones de salto ejecutadas`	MARRÓN

Tabla 43: Relación de los registros representados en las gráficas de este epígrafe, y colores correspondientes.

En la Tabla 44 y Gráfica 2 quedan recogidos los datos de la función `Suave_S()`.

En la Tabla 45 y Gráfica 3 quedan recogidos los datos de la función `RelacionBhascara()`.

En la Tabla 46 y Gráfica 4 quedan recogidos los datos de la función `OrdenarRelaciones()`.

En la Tabla 47 y Gráfica 5 quedan recogidos los datos de la función `EliminacionGaussiana()`.

En la Tabla 48 y Gráfica 6 quedan recogidos los datos de la función `longitud()`.

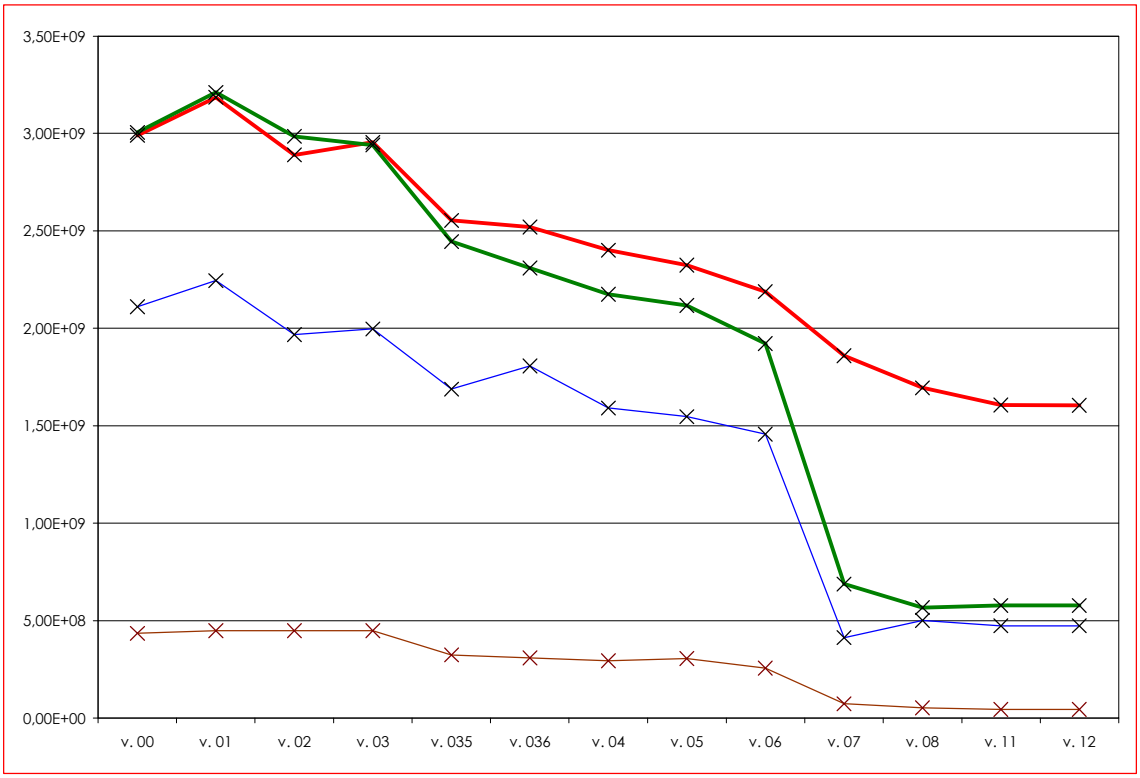
En la Tabla 49 y Gráfica 7 quedan recogidos los datos de la función `COCIENTE()`.

En la Tabla 50 y Gráfica 8 quedan recogidos los datos de la función `MODULO()`.

7.10.1. Datos de la función `Suave_S()`.

v. 00	Versión original de partida.
v. 01	Inserción del código de la función `Modulo()`.
v. 02	Inserción del código de la función `Cociente()`.
v. 03	Reducción de instrucciones.
v. 035	Reducción de instrucciones.
v. 036	Reducción de instrucciones: reuso.
v. 04	Recorrer los vectores con aritmética de punteros en lugar de índices.
v. 05	Convertir algunos vectores de memoria dinámica a memoria estática.
v. 06	Reducción de instrucciones.
v. 07	Conversión de tipos de datos, eliminando el uso del tipo `UINT8`.
v. 08	Evitar dependencias de datos: trabajar con dos vectores de resultados en lugar de con uno solo.
v. 11	Reducción de instrucciones.
v. 12	Reducción de instrucciones.

Tabla 44: Listado de modificaciones realizadas sobre la función `Suave_S()` en sus sucesivas versiones.

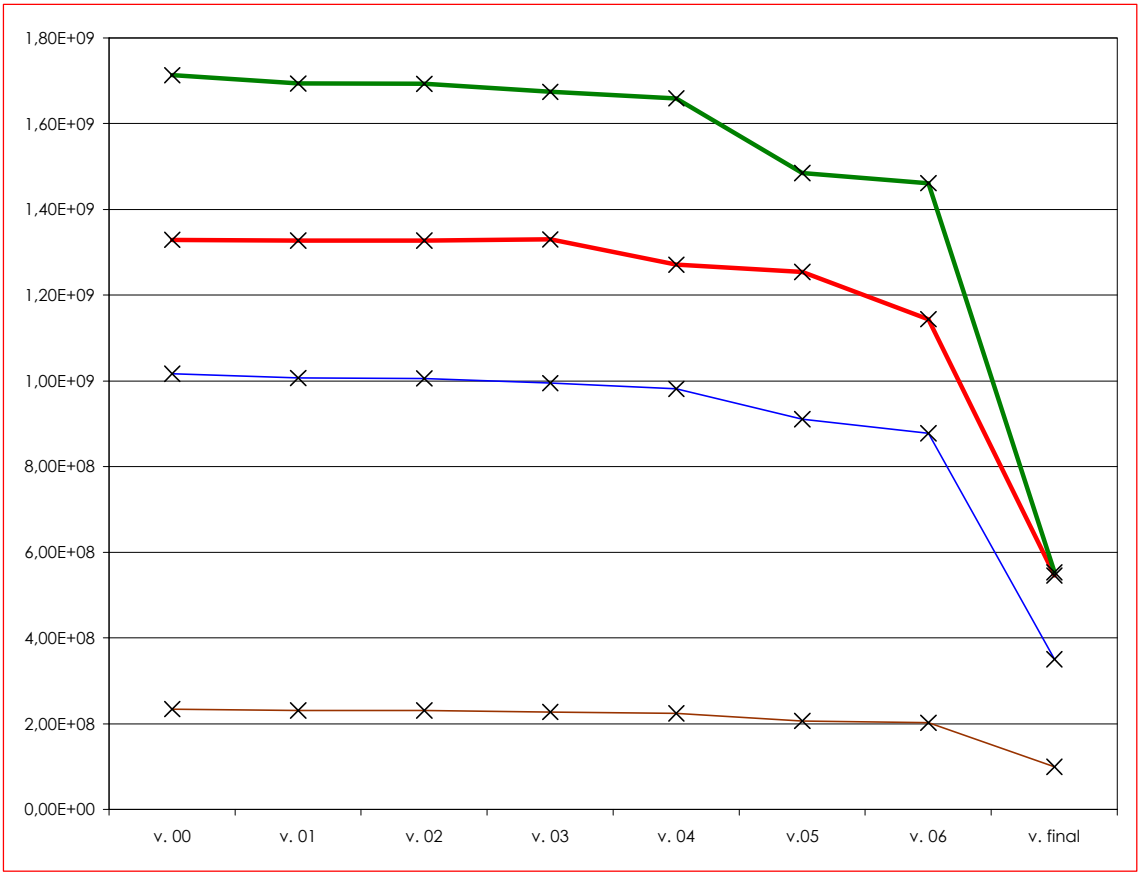


Gráfica 2: Representación de la evolución de los diferentes parámetros medidos con RABBIT en las sucesivas versiones de la función `Suave_S()`.

7.10.2. Datos de la función `RelacionBhascara()`.

v. 00	Versión original de partida.
v. 01	Reducción de instrucciones: llamadas a funciones.
v. 02	Reducción de instrucciones: llamadas a funciones.
v. 03	Reducción de instrucciones: llamadas a funciones.
v. 04	Reducción de instrucciones: reducir fallos a L1.
v. 05	Reducción de instrucciones: llamadas a funciones.
v. 06	Reducción de instrucciones. Redefinición de la estructura BHASCARA. Cambio del modo de acceso de los valores de la estructura. Descenso del número de accesos a memoria.

Tabla 45: Listado de modificaciones realizadas sobre la función `RelacionesBhascara()` en sus sucesivas versiones.

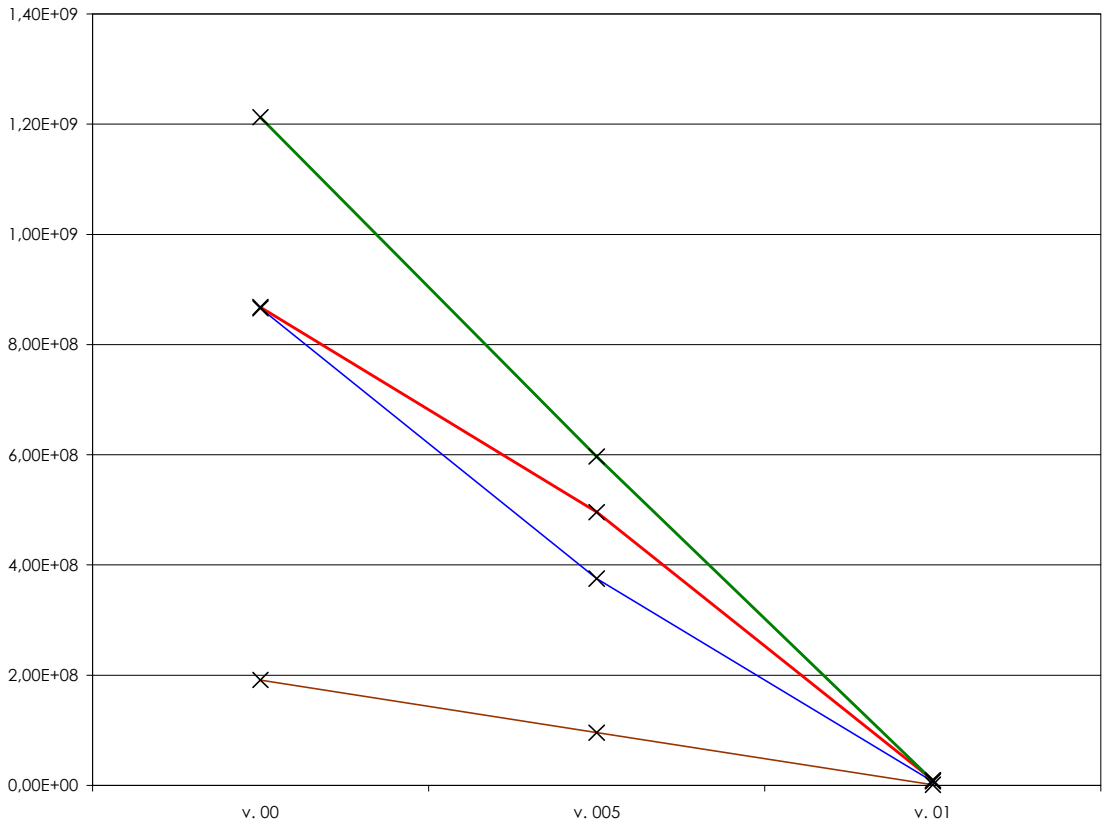


Gráfica 3: Representación de la evolución de los diferentes parámetros medidos con RABBIT en las sucesivas versiones de la función `RelacionBhascara()`.

7.10.3. Datos de la función `OrdenarRelaciones()`.

v. 00	Versión original de partida.
v. 005	Modificación de Algoritmo. Cambiar de tipo de dato `UINT8` a tipo `UINT4` para los valores del Factor Grande que queda después de haber intentado factorizar esos valores con la base de factores.
v. 01	Modificación de Algoritmo: Abandonar el algoritmo de ordenación por el método de la burbuja y trabajar con la función `qsort()` definida en `stdlib.h`.

Tabla 46: Listado de modificaciones realizadas sobre la función `OrdenarRelaciones()` en sus sucesivas versiones.

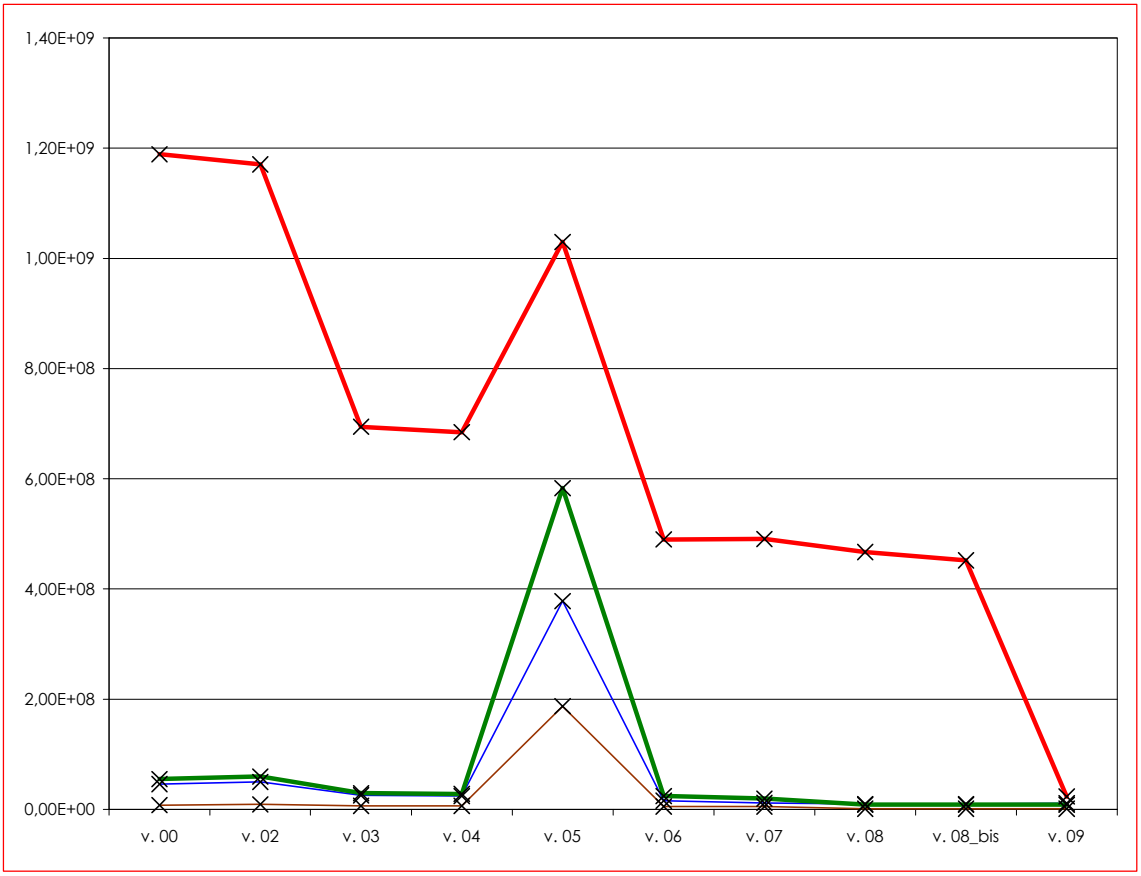


Gráfica 4: Representación de la evolución de los diferentes parámetros medidos con RABBIT en las sucesivas versiones de la función `OrdenarRelaciones()`.

7.10.4. Datos de la función `EliminacionGaussiana()`.

v. 00	Versión original de partida.
v. 005	Reducción de instrucciones.
v. 02	Trabajar las dos matrices (de relaciones y de identidad) por separado).
v. 03	Convertir las matrices de elementos `UINT2` en matrices de elementos `UINT4`.
v. 04	Reducción de instrucciones: reuso.
v. 05	Recorrer las matrices por columnas (y no por filas). Se crean nuevos punteros.
v. 06	Crear vector donde se guardan las direcciones de las filas en las que debemos hacer la operación XOR cuando recorramos las columnas fila a fila.
v. 07	Reducción de instrucciones: reuso.
v. 08	Creación de dos nuevos vectores que guardan el índice de las filas válidas.
v.08 bis	Modificación en la función `InicializarRelaciones()`. Todos los elementos de cada una de las dos matrices vienen definido en un solo vector.
v. 09	Redefinición de matrices. Eliminación de fallos a caché.

Tabla 47: Listado de modificaciones realizadas sobre la función `EliminacionGaussiana()` en sus sucesivas versiones.

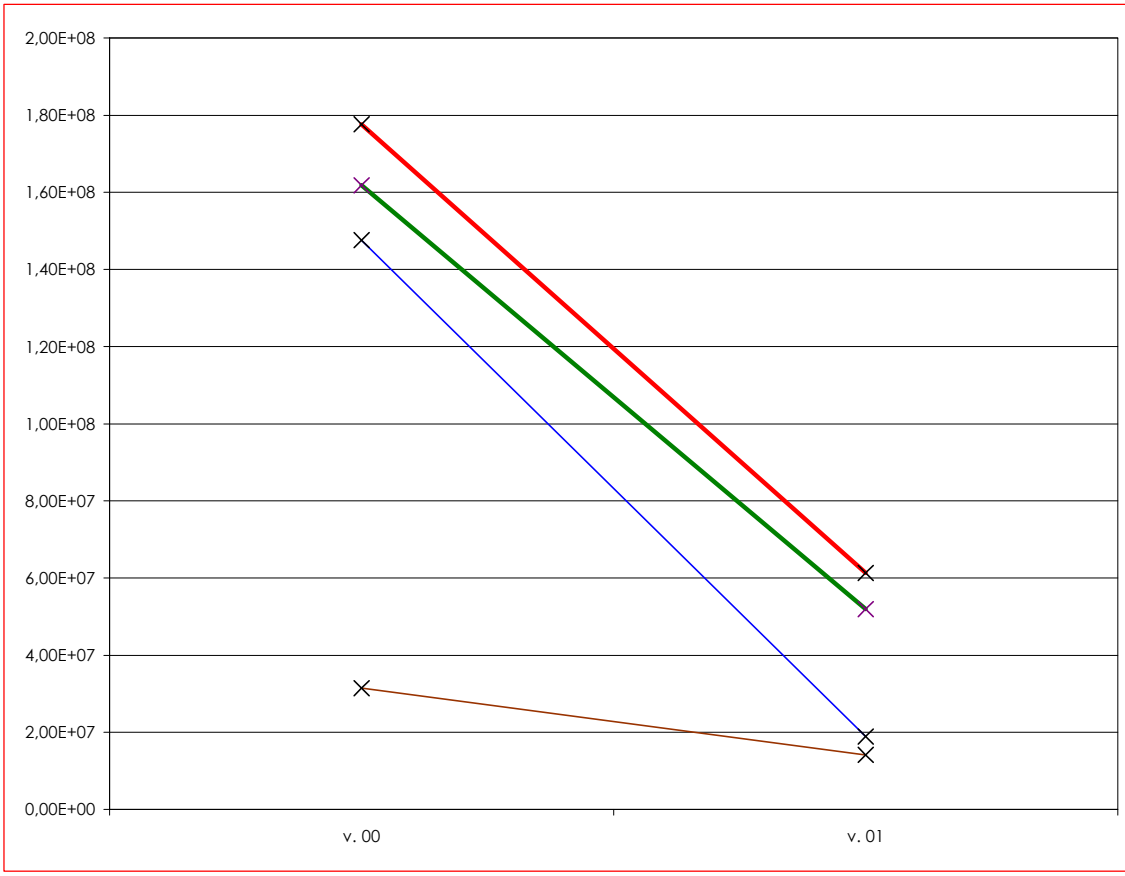


Gráfica 5: Representación de la evolución de los diferentes parámetros medidos con RABBIT en las sucesivas versiones de la función `EliminacionGaussiana()`.

7.10.5. Datos de la función `longitud()`.

v. 00	Versión original de partida.
v. 01	Reducción de instrucciones: Cambiar una estructura **`while`** por varias concatenadas de **`if`** – **`else`** eliminando instrucciones de salto.

Tabla 48: Listado de modificaciones realizadas sobre la función `longitud()` en sus sucesivas versiones.

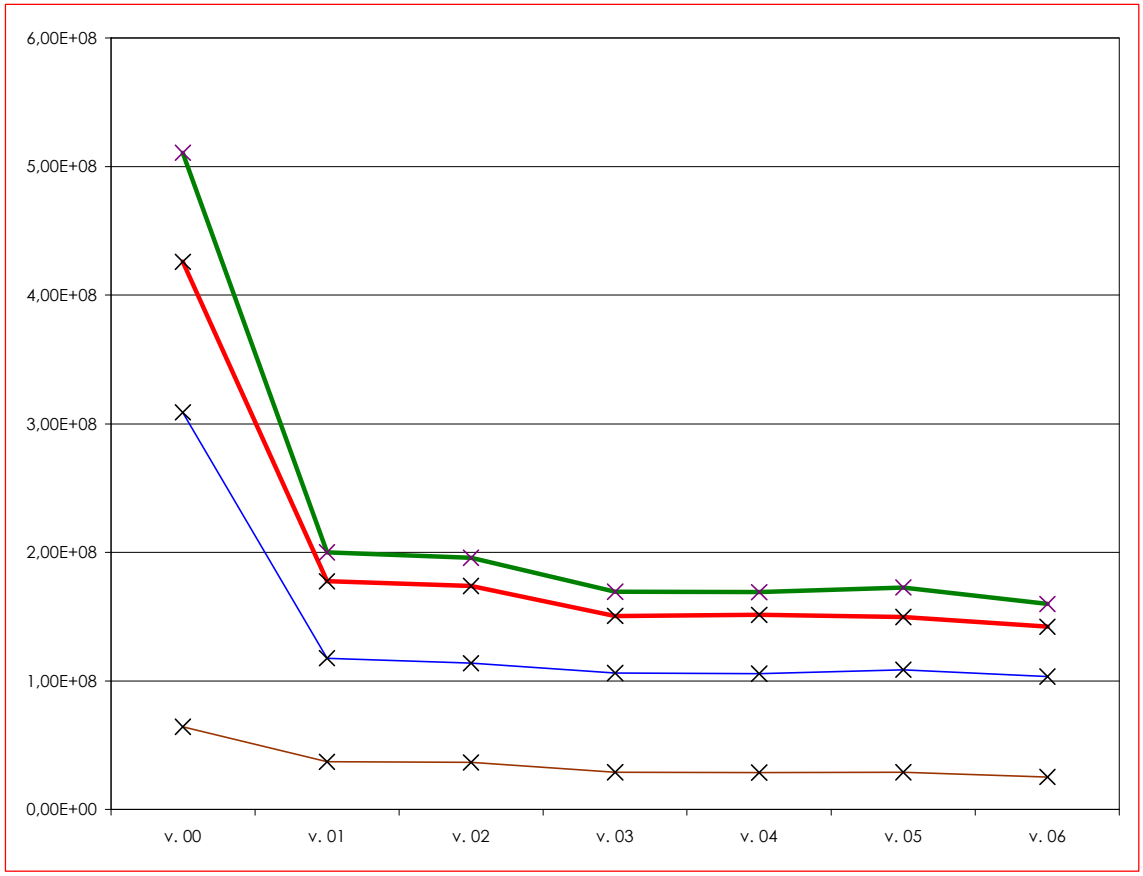


Gráfica 6: Representación de la evolución de los diferentes parámetros medidos con RABBIT en las sucesivas versiones de la función `longitud()`.

7.10.6. Datos de la función `COCIENTE()`.

v. 00	Versión original de partida.
v. 01	Reducir Accesos a memoria. Colocar varias variables de array en un mismo vector de memoria.
v. 02	Reducción de instrucciones.
v. 03	Reducción de instrucciones: llamadas a funciones.
v. 06	Reducción de instrucciones: llamadas a funciones.

Tabla 49: Listado de modificaciones realizadas sobre la función `COCIENTE()` en sus sucesivas versiones.

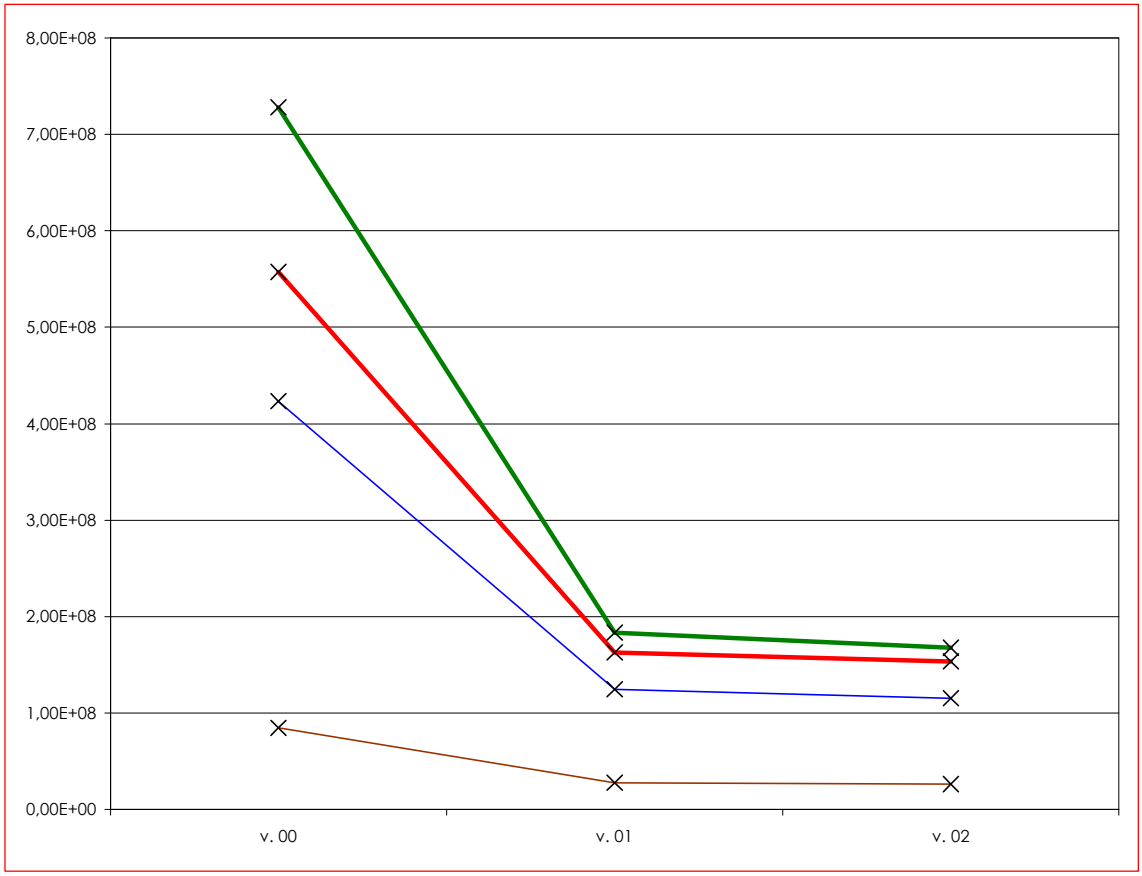


Gráfica 7: Representación de la evolución de los diferentes parámetros medidos con RABBIT en las sucesivas versiones de la función `COCIENTE()`.

7.10.7. Datos de la función `MODULO()`.

v. 00	Versión original de partida.
v. 01	Tomar todas las mejoras de la función `COCIENTE()` y trasladarlas a esta función.
v. 02	Reducción de instrucciones.

Tabla 50: Listado de modificaciones realizadas sobre la función `MODULO()` en sus sucesivas versiones.



Gráfica 8: Representación de la evolución de los diferentes parámetros medidos con RABBIT en las sucesivas versiones de la función `MODULO()`.

7.11.EVALUACIÓN Y ANÁLISIS DE RESULTADOS. CONCLUSIONES.

Una vez hemos mostrado diversos ejemplos de optimización en cada uno de los cinco pasos del protocolo definido, parece necesario poder analizar el comportamiento de la aplicación de factorización en su totalidad. En la Tabla 51 quedan recogidos los valores de los 10 registros que hemos estudiado y que nos han servido de guía para la optimización paso a paso, función a función, versión a versión.

	Versión V0	Versión V1	factor de reducción
Fallos en L2	5.254.965	25.781	203,83
Referencias a memoria de datos	4.441.428.694	808.793.008	5,49
Fallos en L1	80.424.542	93.366	861,39
Ciclos de reloj	7.672.819.100	1.702.847.174	4,51
Microinstrucciones ejecutadas	9.420.060.220	1.785.738.909	5,28
Instrucciones ejecutadas	7.038.923.209	1.181.864.423	5,96
Instrucciones de salto ejecutadas	1.029.056.327	186.338.505	5,52
Instr. de salto equivocadas predichas	31.804.944	15.702.519	2,03
Saltos tomados ejecutados	594.373.205	131.769.458	4,51
Saltos tomados mal predichos ejecutados	15.007.514	8.108.609	1,85

Tabla 51: Registros analizados con RABBIT, en la aplicación de factorización, antes y después de la optimización.

Hemos presentado un protocolo de actuación para optimizar implementaciones de algoritmos. El plan de trabajo puede resultar válido para cualquier tarea de optimización. Nosotros lo hemos definido y empleado para una aplicación de criptoanálisis, donde los algoritmos que se manejan habitúan a ser costosos en tiempos de computación y, por tanto, donde es conveniente procurar siempre métodos y procedimientos de trabajo que nos permitan aumentar la eficiencia de los algoritmos.

Hemos mostrado diferentes resultados de optimización, según haya estado centrada la atención en la reducción de instrucciones o específicamente en la reducción de instrucciones de salto; en la reducción de accesos a memoria; en el cambio de algoritmo por otro más eficiente; o en evitar dependencias de datos. Los datos que se recogen en esta Tabla 51 muestran una reducción importante en todos los valores de los contadores del microprocesador. De su análisis, lo primero que podemos señalar es que se ha logrado aumentar la velocidad de ejecución del proceso en un factor de 4.5. El número de instrucciones ejecutadas también ha bajado sensiblemente: casi en un orden de 6. El descenso de las referencias a la memoria de datos también es importante: del

orden de 5.5.

Pero los valores más llamativamente significativos respecto a los accesos a memoria los encontramos en el enorme descenso obtenido en los fallos a L1 (del orden de 861) y a L2 (del orden de casi 204). Este descenso ha sido alcanzado gracias a las redefiniciones de las matrices de relaciones que se generan en los procesos de factorización, y a las modificaciones en el modo de recorrer los valores de estas matrices, procurando aprovechar la localidad temporal y, en nuestro caso especialmente, la localidad espacial.

7.11.1. Breves comentarios a los resultados presentados.

Del estudio y análisis de las Tablas 44 a 50 y de las Gráficas 2 a 8 presentamos las siguientes observaciones:

1. Abandonar el tipo de dato `UINT8` (**`unsigned long long int`**) y hacer uso de las variables de tamaño la longitud de palabra (**`unsigned long int`**: `UINT4`) siempre ha resultado un cambio que ha provocado una importante reducción de instrucciones. Así puede verse en la Gráfica 2, versión v. 07 y en la Gráfica 4 versión v. 005. Así ocurría también en la optimización de la función `SUMA()`, que presentábamos en la Tabla 16. Por otro lado, y en todos los casos, esta sustitución ha supuesto una disminución del valor del índice IPC, como puede verse especialmente en la Gráfica 2, donde la caída de los ciclos de reloj es de una pendiente sensiblemente menor a la caída de las instrucciones ejecutadas.

2. Las Gráficas 3, 4, 7 y 8 muestran la línea de las instrucciones ejecutadas (línea verde gruesa) siempre por encima de la de los ciclos de reloj (línea roja gruesa). En estos casos, las funciones tienen un IPC mayor que 1 y podemos por tanto decir que en su ejecución el ordenador logra paralelismo al ejecutar, gracias a la segmentación y a que el procesador tiene varios cauces de ejecución de instrucciones (superescalar), más de una instrucción por cada ciclo de reloj.

3. Especialmente en la función `EliminacionGaussiana()` todo el trabajo se ha centrado en lograr aumentar el valor del IPC, y el principal objeto de estudio ha sido los valores de los fallos en la cachés. Para reducir estros fallos nos hemos centrado principalmente en atender al principio de localidad espacial de los accesos a memoria.

   Otros dos ejemplos muy claros de la ventaja de aprovechar el principio de localidad espacial los tenemos en las funciones `COCIENTE()` y `MODULO()`, representadas en las Gráficas 7 y 8. Si se rompe la distribución lógica de la información, que tiende a ser agrupada en vectores homogéneos, y se trabaja con otros criterios de almacenamiento de datos, basados en los principios de localidad espacial y temporal, entonces se llega a la implementación de códigos de más difícil interpretación, donde la ordenación lógica desaparece, pero se logra un código que optimiza los accesos de memoria. No se logran programas cómodos de interpretar, y no resulta sencillo construirlos. Para la implementación de una aplicación que tenga en cuenta

los principios de localidad hemos visto muy necesario el uso de herramientas como la presentada en este capítulo (RABBIT).

4. Las sentencias condicionales han ocasionado en todos los casos un aumento de los errores en los predictores de salto, y eso lleva consigo mayor número de instrucciones ejecutadas y, especialmente, decremento del valor del índice IPC. Sin embargo, como se puede comprobar en la Gráfica 6 y como hemos visto en la función `PROD_bit()` (Tablas 29 y 30), su buen uso, en sustitución de estructuras de control de iteración, puede reducir el número de instrucciones ejecutadas.

5. La Gráfica 3, con su punto más a la derecha llamado v. final, muestra que un camino cierto para lograr reducir tiempos e instrucciones es el trabajo metódico. Los valores presentados en la Tabla 51 son también una buena muestra de que se pueden lograr mejoras importantes con el procedimiento que hemos presentado, descrito y seguido en este Capítulo.

7.11.2. Datos comparados: antes y después de la optimización.

bits	clocks	bits	clocks	bits	clocks	bits	clocks
63	13,5	83	60,0	103	443,0	123	3805,5
64	13,0	84	59,0	104	450,0	124	3351,0
65	14,0	85	74,5	105	536,0	125	4506,0
66	14,0	86	61,5	106	661,5	126	3957,0
67	18,0	87	81,0	107	757,5	127	6428,5
68	15,0	88	81,5	108	691,0	128	5504,0
69	22,0	89	84,0	109	881,0	129	7958,0
70	18,0	90	91,0	110	917,0	130	7404,5
71	22,5	91	110,0	111	1144,5	131	8709,0
72	18,5	92	112,0	112	1124,5	132	8813,0
73	27,0	93	123,5	113	1608,0	133	11305,0
74	21,0	94	133,0	114	1402,0	134	12352,5
75	26,5	95	221,0	115	1769,5	135	15405,5
76	26,0	96	191,0	116	1538,0	136	16523,5
77	33,0	97	270,0	117	1592,0	137	15385,0
78	29,0	98	217,0	118	1807,0	138	18621,0
79	38,0	99	279,0	119	2257,5	139	26408,5
80	34,5	100	268,0	120	2477,0	140	23744,0
81	45,0	101	419,5	121	2412,5		
82	45,0	102	460,0	122	3199,5		

Tabla 52: Valores obtenidos con la función `times(NULL)`, según los tamaños de los enteros a factorizar, utilizando la aplicación basada en el algoritmo CFRAC, y después de haber optimizado algunas de las funciones del proceso.

En la Tabla 52 mostramos los resultados de medición de tiempos (tomados mediante la función `times()` de la biblioteca `sys/times.h`), en un proceso de factorización similar al presentado en la Tabla 2: recogemos ahora los valores de las medianas de los tiempos invertidos en la factorización de diferentes enteros, de tamaños entre 63 y 150 bits, producto de dos primos largos, realizado con la misma aplicación que antes empleamos en el cálculo de los datos de la Tabla 2; pero ahora con las funciones optimizadas.

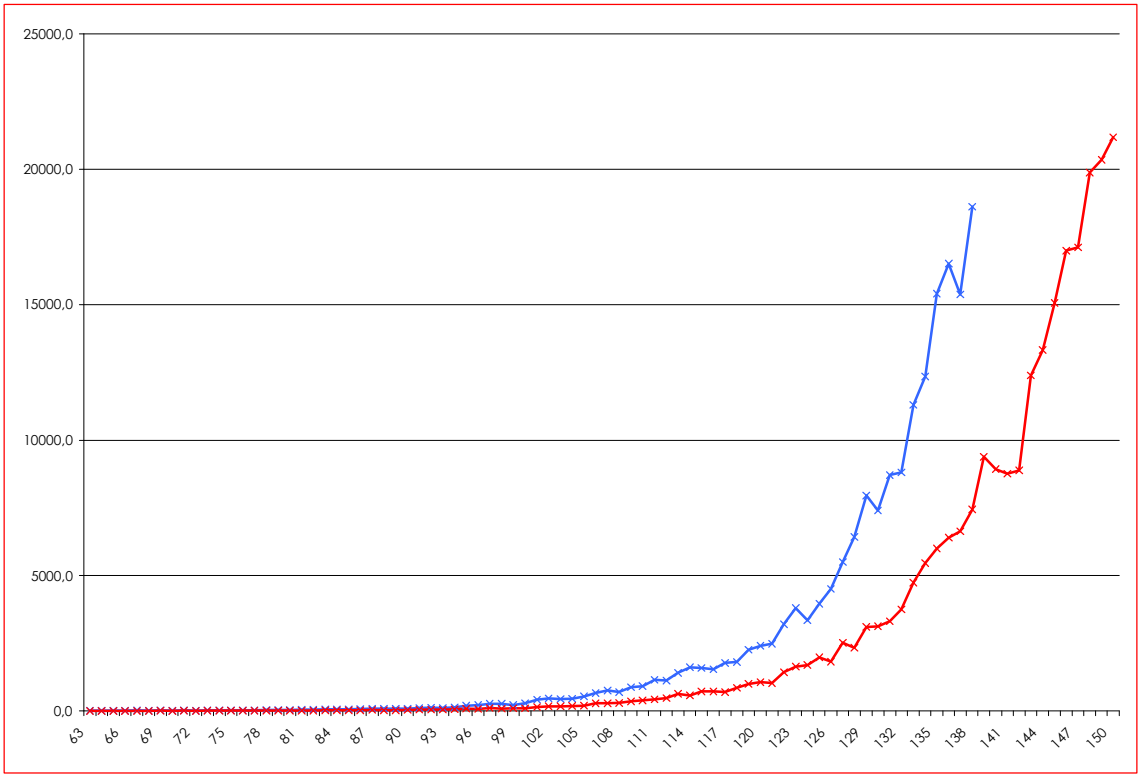
bits	factor de mejora	bits	factor de mejora	bits	factor de mejora	bits	factor de mejora
63	3,38	82	2,65	101	2,89	120	2,26
64	3,25	83	2,68	102	2,69	121	2,41
65	2,80	84	3,00	103	2,58	122	2,24
66	2,80	85	2,12	104	2,41	123	2,33
67	3,67	86	3,04	105	2,68	124	1,98
68	3,60	87	2,58	106	2,41	125	2,00
69	3,38	88	2,70	107	2,64	126	2,48
70	3,00	89	2,67	108	2,33	127	2,19
71	2,47	90	2,53	109	2,46	128	2,75
72	3,00	91	2,50	110	2,34	129	2,57
73	2,94	92	2,47	111	2,66	130	2,37
74	3,21	93	2,26	112	2,37	131	2,63
75	2,47	94	2,24	113	2,19	132	2,35
76	2,89	95	2,39	114	2,79	133	2,39
77	2,64	96	2,83	115	2,19	134	2,26
78	3,30	97	2,50	116	2,14	135	2,57
79	2,65	98	3,21	117	2,56	136	2,58
80	3,17	99	2,17	118	2,11	137	2,32
81	2,81	100	2,76	119	2,27	138	2,50

Tabla 53: Factor de mejora entre la versión sin optimizar y la versión optimizada, según el tamaño (en bits) del entero a factorizar. La mediana de todos estos valores es 2,57.

En la Tabla 53 mostramos los factores de mejora (relación entre el tiempo invertido en la versión sin optimizar y el tiempo invertido en la versión optimizada, para cada uno de los tamaños), que, como podemos observar, ha mantenido un valor bastante constante para todos los tamaños. La mediana de todos estos valores es 2,57.

Por último, presentamos una Gráfica que recoge la representación de las dos curvas, antes y después de la optimización (Gráfica 10): los valores de las Tablas 2 y 52.

Hemos realizado una búsqueda de aplicaciones de Cálculo Simbólico (también llamado Cálculo Formal o Álgebra Computacional) y de Cálculo Numérico. En todas las herramientas estudiadas hemos buscado los algoritmos de factorización implementados. Hemos querido conocer cuáles



Gráfica 9: Tiempos empleados en la factorización de enteros de diferentes tamaños antes (en azul) y después (en rojo) de realizar el proceso de optimización. Representación gráfica de los valores de la Tabla 52.

son los más empleados y qué secuencia de algoritmos emplean cada una de esas aplicaciones. En el anexo VII recogemos un resumen de ese estudio.

Podríamos clasificar esas aplicaciones en las siguientes clases:

Programas de Propósito General: programas abiertos sobre los que hay que diseñar las distintas aplicaciones: Lenguajes de Programación. Especialmente C y VB. Hojas de Cálculo. La más conocida actualmente es Excel. Asistentes Matemáticos: programas específicos para el trabajo en matemáticas.

Programas específicos: Programas de enseñanza asistida por ordenador; suelen ser Tutoriales. Programas de Estudio de Funciones. Programas de Geometría. Programas de Resolución de problemas.

Existen muchas y diferentes aplicaciones útiles para la computación en Teoría de Números. Muchas de estas aplicaciones han sido escritas primariamente para aplicaciones matemáticas, de ingeniería y física, y no siempre bien diseñadas para teoría de números.

8

CONCLUSIONES

En esta Tesis hemos trabajado en dos ámbitos: la criptografía y la arquitectura de los ordenadores. De la mano de los conocimientos de la criptografía, hemos desarrollado una implementación para trabajar con enteros grandes, capaces de codificar valores enteros tan grandes como se quiera, sin más limitación que la memoria total del ordenador. Hemos implementado una serie de herramientas criptográficas, y especialmente una de criptoanálisis, destinada a la factorización de enteros largos. Y de la mano de los conocimientos de la arquitectura, y gracias al análisis del comportamiento de nuestro código en su interacción con la máquina, hemos optimizado esa implementación y hemos logrado mejoras importantes en el rendimiento de las aplicaciones ejecutadas.

En la criptografía moderna, nacida de la mano del desarrollo de los ordenadores, se precisa con frecuencia ejecutar algoritmos de alto coste computacional. Como señala POMERANCE [Pome96] el avance de la criptografía en este campo ha de venir de la mano de la investigación matemática, que busque caminos de menor computación para obtener los mismos resultados; y

ha de venir de la mano de la programación, que busque modos de implementar que supongan una reducción del tiempo invertido en la ejecución de un algoritmo determinado. Y como señala [Clap97], el incremento de la eficiencia de los procesadores y el logro en el aumento de paralelismo (tanto desde un punto de vista de arquitectura como gracias a una mejora en la implementación que permita índices IPC más altos) puede ofrecer criterios que orienten a la hora de diseñar nuevos algoritmos criptográficos. Esta idea de la interrelación entre algoritmia y optimización de código constituye uno de los dos hilos conductores de este trabajo.

El otro hilo conductor se centra en el mismo reto de la factorización. Claude SHANNON fue el primer matemático en considerar el problema de la seguridad desde un punto de vista matemático; y concibió un modelo basado en el concepto de seguridad incondicional, que era el que tendría un criptosistema que fuese resistente al criptoanálisis incluso ante un criptoanalista dotado de recursos computacionalmente ilimitados: matemáticamente imposible, con independencia de los medios disponibles. Lo que subyace en la posterior propuesta de DIFFIE y HELLMAN es el concepto de seguridad computacional: computacionalmente no factible. El usuario de un criptosistema no debe esperar ya que el criptoanalista no tenga información suficiente para romperlo, sino que no tenga tiempo. De todo es sabido la enorme extensión de la que goza actualmente el criptosistema RSA, que descansa su seguridad en la dificultad de factorizar enteros de gran tamaño. Es, por tanto, muy importante, tener una idea de la dificultad del problema de la factorización para poder saber hasta que punto RSA es criptográficamente seguro.

8.1. TAREAS Y APORTACIONES

Entre las herramientas criptográficas que hemos implementado, además de todas las funciones necesarias para disponer de un tipo de dato que codifique enteros largos, están los diferentes algoritmos: especialmente destacamos los tests de primalidad y el generador de secuencias de bits aleatorios. También presentamos en la Tesis una implementación del generador de secuencias pseudoaleatorias BBS.

La herramienta de criptoanálisis desarrollada es una aplicación para factorizar enteros, que sean producto de dos primos largos. Esos enteros son requeridos en algunos algoritmos especialmente en uso actualmente como el criptosistema de clave pública RSA o el mismo generador de secuencias antes citado BBS. El algoritmo que hemos implementado para la factorización de enteros es el basado en la técnica de las fracciones continuas, y que fue introducido y desarrollado por Michael MORRISON y John BRILLHART en el año 1975. En la década de los 80 fue el principal algoritmo en los logros en retos de factorización.

Y de la mano de una herramienta (llamada RABBIT) que nos ha permitido conocer el comportamiento de nuestros programas en su interacción con el ordenador hemos optimizado el

rendimiento de algunas de nuestras implementaciones. Para este proceso de optimización de código hemos desarrollado un protocolo de actuación, todo él orientado, en último término, a la reducción de instrucciones a ejecutar y, sobre todo, a la reducción de los ciclos de reloj en la ejecución de esas instrucciones.

Las principales aportaciones de nuestro trabajo podemos sintetizarlas en los siguientes puntos:

1. La primera aportación que podemos recoger aquí es el trabajo de implementación desarrollado. Es cierto que ya existen disponibles bastantes librerías a disposición, y herramientas de libre distribución (ver en el anexo VII para más detalles). Hemos desarrollado nuestras propias funciones porque era el mejor camino para poder optimizarlas. En el desarrollo de la Tesis presentamos un estudio comparativo de nuestra concepción de entero largo con otras que consideramos especialmente interesantes, como la librería FreeLIP, desarrollada por LENSTRA o la definida por ZIMMERMAN y empleada en su famosa aplicación criptográfica PGP.
2. Hemos definido e implementado un generador de secuencias aleatorias, que ha superado los tests de estadísticos de aleatoriedad sugeridos en las principales referencias. Especialmente nos hemos ceñido a las sugeridas por MENEZES, et al. [Mene97] y al test estadístico universal de MAURER. Este generador es el que hemos empleado en todos nuestros trabajos, en la asignación aleatoria de valores iniciales de una semilla para el generador de pseudoaleatorios BBS.
3. Hemos implementado la aplicación de factorización basada en la técnica de las fracciones continuas. Hemos determinado, para nuestra implementación, los valores óptimos de los parámetros que ajustan el comportamiento del algoritmo.
4. Y hemos desarrollado un protocolo para la optimización de algoritmos. Este protocolo consta de cinco pasos fundamentales, y basa la reducción de instrucciones y de tiempos de ejecución en los siguientes logros: aprovechar el principio de localidad temporal y espacial de la memoria, procurando así reducir los fallos de acceso a los diferentes niveles de memoria caché; modificar el código de manera que disminuyan las instrucciones de salto, y especialmente que se reduzca el número de saltos tomados mal predichos y ejecutados y las instrucciones de salto equivocadamente predichas. Al final del proceso de optimización, que hemos llevado a cabo sobre nuestra implementación de la aplicación de factorización basada en la técnica de las fracciones continuas y que describimos paso a paso, hemos reducido los tiempos de ejecución en un orden de 2,5 veces.

8.2. TRABAJO FUTURO

Entre los muchos flecos que hemos tenido que dejar a lo largo del desarrollo de nuestro trabajo, y sobre los que nos interesaría volver para resolver y avanzar en nuestra investigación, destacamos los siguientes:

1. Optimizar, mediante la técnica que hemos descrito en nuestra Tesis, la implementación del generador de secuencias pseudoaleatorias BBS. Es un generador de muy buenas características que lo hacen criptográficamente seguro, pero que tiene como principal dificultad su lentitud en la generación de las sencuencias. Aunque, de la mano de una sugerencia presentada en [Vazi85], hemos desarrollado una implementación alternativa a la indicada por los autores del generador BBS que mejora notablemente la velocidad, sigue siendo necesario ganar velocidad: como señala [Ding97] el procedimiento del generador es lento y quizá por eso todavía no recomendable para aplicaciones multimedia. Lograr una implementación de este generador con una velocidad mayor es una primera tarea pendiente.

2. Otra tarea pendiente en la puesta en marcha del generador BBS es lograr un camino para generar primos doblemente seguros. Como mostramos en el Anexo III, estos primos son de difícil localización y, para tamaños criptográficamente seguros, no existe todavía un algoritmo suficientemente eficiente.

3. Posteriormente al algoritmo de factorización basado en la técnica de las fracciones continuas (CFRAC) han aparecido otros computacionalmente mejores, que exigen, a partir de un determinado tamaño del entero a factorizar, menos tiempo en la ejecución para alcanzar los factores primos. Son los algoritmos conocidos como QS y NFS. Muchas de las optimizaciones realizadas en nuestra implementación del algoritmo CFRAC son válidas para estos dos nuevos algoritmos, que tienen un mismo fundamento matemático y siguen, en parte, una misma trayectoria de ejecución. Un trabajo futuro, que seguirá los mismos pasos que el presentado en esta Tesis, es implementar estos dos algoritmos y someter la implementación al protocolo de optimización.

4. Por último, un trabajo que queda pendiente y permitirá lograr factorizar enteros de mucha mayor longitud será el uso de varios ordenadores, operando en paralelo, por ejemplo en un cluster de PC's. Hemos implementado todas las funciones teniendo en cuenta esta posibilidad de trabajo futuro. En la presentación de la implementación del algoritmo CFRAC hacemos referencia a ello. Hemos realizado algún ensayo en este sentido con la interfaz MPI (Message Passing Interface). Actualmente todos los retos de factorización acometidos y alcanzados (el último publicado el de la factorización de un entero de 155 dígitos decimales: 512 bits) se han logrado con diferentes ordenadores en paralelo.

A

ANEXOS

a ANEXO I: IMPLEMENTACIÓN DEL GENERADOR DE SECUENCIAS DE BITS ALEATORIOS

Recogemos en este anexo una posible implementación del algoritmo 1 presentado en el Capítulo 5 de la Tesis, y también la implementación de la función `caracter()` que se emplea para almacenar, en el sistema operativo Linux, cada uno de los caracteres introducidos por teclado.

```
clock_t t1,t2;
UINT4 dift;
UINT2 aleat,rot;
char letra;

t1 = times(NULL);
do
{
      letra = caracter();                                          Paso 1
      t2 = times(NULL);
      dift = t2 - t1;                                              Paso 2
      t1 = t2;
      dift *= dift;                                                Paso 3
      if(aleat) dift *= aleat;
      rot = dift % 31;
      t2 *= t2;
      t2 = ((t2 >> rot) | (t2 << (Byte4 - rot)));
      aleat ^= t2;                                                 Paso 4
      if(aleat) letra *= aleat;                                    Paso 5
      rot = (UINT2)letra % 13;
      aleat = (aleat << rot) | (aleat >> (Byte2 - rot));
      (insertar el código necesario para guardar el valor de aleat).  Paso 6
}while(letra != 27);                                               Paso 7



char caracter()
{
      char a;
      initscr();
      nonl();
      leaveok(curscr,0);
      raw();

      fcntl(0,F_SETFL,O_RDONLY);
      while(!read(0,(char*)&a,1));

      noraw();
      nocrmode();
      nl();
      endwin();

      fcntl(0,F_SETFL,O_NDELAY |  O_RDONLY);

      return(a);
}
```

a ANEXO II: ALGORITMO DE BERLEKAMP–MASSEY

Mostramos el ordinograma del algoritmo de BERLEKAMP–MASSEY, diseñado para obtener el perfil de la complejidad lineal de una secuencia de bits.

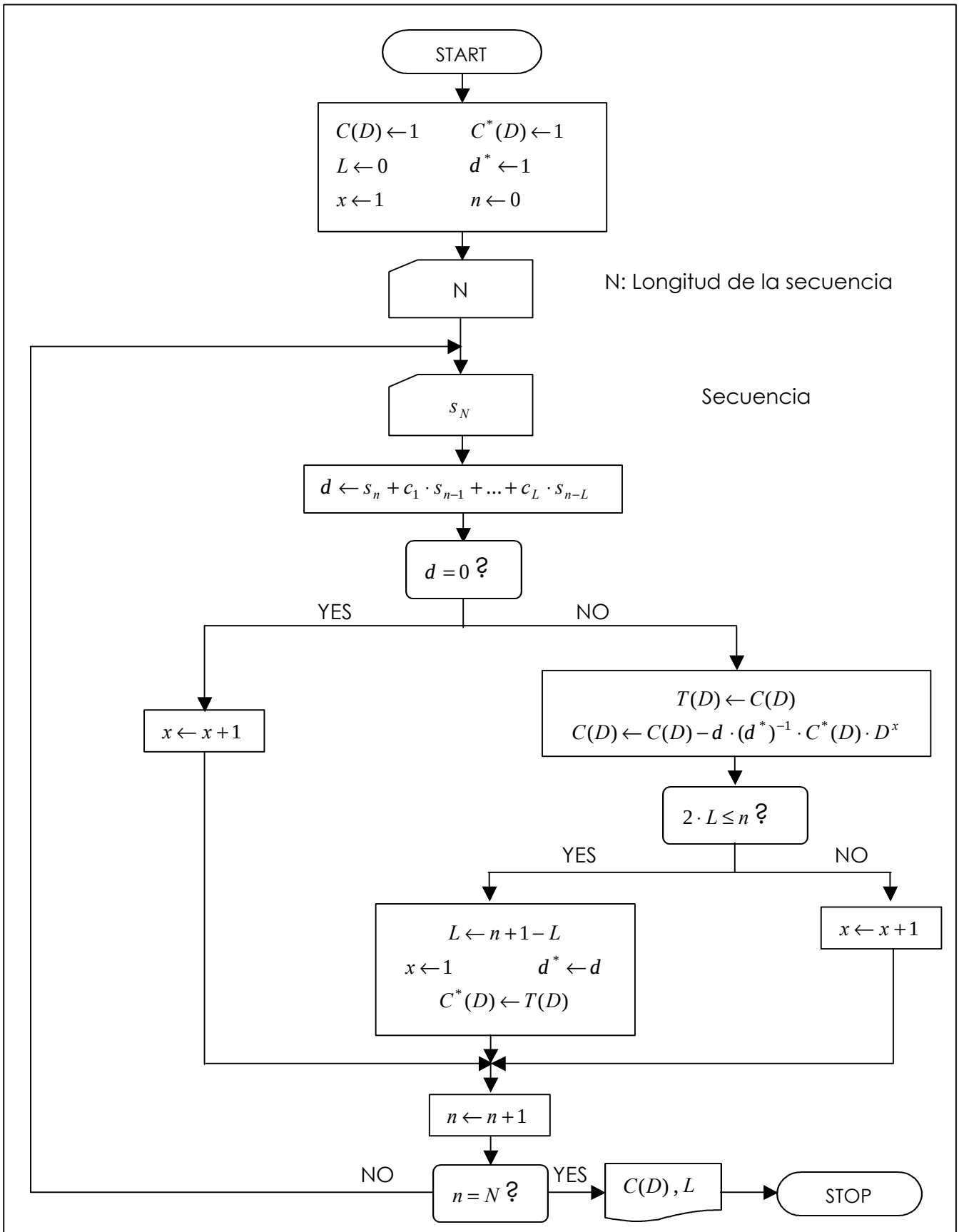


Figura 1: Diagrama de Flujo del Algoritmo BERLAKAMP–MASSEY.

a ANEXO III: PRIMOS DOBLEMENTE SEGURO PARA EL GENERADOR BBS

Mostramos en este anexo unas tablas de enteros primos 2–seguro, de tamaños que van desde los 32 bits hasta los 512 bits, en incrementos de 32 en 32. Recogemos 20 primos de cada tamaño. También queda recogido en las tablas el tiempo (en segundos) que se ha invertido en la búsqueda de cada uno de esos valores.

Primos Doblemente Seguros de 32 bits

sec.	Número en formato Hexadecimal...
0	9D6165C7
0	85F1E9F7
0	DB3A7537
0	A729AD17
0	D94429E7
0	D11F9DF7
0	90374D17
0	C1107227
0	B0612A77
0	A7333C27
0	9F2527F7
0	D2CFC1F7
0	84B3C4D7
1	A420E7F7
0	8A76DEC7
0	BB9C8F17
0	B1F4C937
0	CBD8AD07
0	EB704107
0	FBF46A97

Primos Doblemente Seguros de 64 bits

sec.	Número en formato Hexadecimal...	
0	D6D5A30F	86CD8127
1	AB319C56	32328E87
0	D7D84460	F8C6B9B7
1	A47860A8	70C365E7
1	D74364BC	5D7C0307
0	D8CD2A16	F60E7447
1	A68B8838	5FF871E7
1	A27CE9E8	344E4197
0	B95E54D1	F088CAC7
2	9D8429BC	98EE1EE7
1	A0585FCC	B6E4BC27
1	93B22659	D31413D7
0	F0196C8A	198DB457
1	899F5F75	D2617497
0	D9E4980D	4C529A07
0	DCAFAA8B	D24E52D7
2	986911A9	6A2D0C47
0	F11D0081	92082557
1	C0B31196	0DD23537
0	9D29002B	5B8C8647

Primos Doblemente Seguros de 96 bits

sec.	Número en formato Hexadecimal...		
3	94269E85	8CF10D6F	0753AAD7
7	FCD1E3EB	C14D81A8	AF8F99B7
1	A2382693	8256C51B	B3F68727
0	C4DD2FA5	22B514B6	E7BEA6C7
2	81E3AF7B	96F7197C	08AF7FF7
3	E53F2F83	2D0EE547	F7467FC7
1	E3C004E7	F38FC8D6	65CC7FB7
2	B74A53CB	69E1556D	3D8717B7
1	E17968A6	565EC095	FD979237
2	BB2646D0	D8F6FB2D	3A072FE7
1	DEDF8E30	120ED5F6	3AC75597
4	C8E8F13A	9B2DB0C9	FFD3F5D7
0	E912E5A8	0E5A4173	35F747C7
5	D73C7BB5	51F720E0	AD18F4B7
4	8C438314	5BC9E031	5DFB4297
2	867EB378	4516B299	820A3F87
1	83473CA8	CAFB3337	7AAAC407
2	D1610FF2	DDF42E84	9FD98167
2	AB27940F	5960206C	26F424E7
2	DD040BEC	FE657121	56CA3517

Primos Doblemente Seguros de 128 bits

sec.	Número en formato Hexadecimal...			
3	9951FB94	4812A020	E95AFAB6	B69920A7
4	86D7B243	8B338BF8	C41ED893	51B6FCF7
10	F69EB3DB	8F73F4E5	944D7BEA	27A39517
8	8BECB41E	6F1D6748	0A98C8FB	4E7DD3F7
3	9D20D6F3	AFA10822	B529EDDC	B85BC3C7
10	BBF216AB	9F252392	305AF483	DA63BF07
5	E6214C36	5A75A5BB	BBDBCA16	BB930837
9	F095C021	55FF4BA2	F7669E25	15AB5767
9	93373415	008519DF	C830BD96	B3789047
10	97AD3F70	8B433172	F6348E80	9ED71C07
2	EA0F3737	572DA71F	A2A5C086	E46AD1C7
9	F2566B7D	CA1A563B	9416CF55	011A1B17
6	BCB58559	233C8899	E9DEB040	82001727
7	839221F5	D7B4CE22	6B01280A	30E02057
2	90BB750A	1F61C2E2	1473DB31	84C81AD7
1	E3E92DDC	94D4ED3D	4992C92C	7FF41CE7
14	AC141EDB	F2237401	0AF88649	B8123897
7	EE82F950	193F4CE2	FC5A1650	C9B34A57
10	8136709F	96975DB5	556B146F	3E49D287
4	A7166EBF	8A4DE9B1	CB40A960	B6E0BD37

Primos Doblemente Seguros de 160 bits

sec.	Número en formato Hexadecimal...				
17	917C0253	1B5D6EDA	53F5D132	E12E7500	1254D8F7
35	8790D11F	514CCC5B	D4BF7954	46F2FF53	20B4E847
9	E66E9116	9AA74040	1BA49CC1	C475750D	D2011887
1	C09263F4	C96E2557	B5D54E7D	44AA3CAF	98E73E47
17	B43CBB4D	F6F8161B	E2079F73	500F6718	FAFD7A67
59	ADF4C18F	706E0445	7F97A4C8	E9E4595D	8B46C337
22	8C4DA595	1F4A2FA8	2641A863	8F70CA5E	4C2F4FD7
1	D918C622	DF32E26F	6D02CB4A	939A4AB4	B4FB94F7
5	83159293	AF3B2CFD	C461FE29	33C1987B	F96B6D97
61	A9AF1542	26E9314E	0660BD68	4B07FD21	C3A364C7
5	9BDD29D0	A36E0823	7EBF9955	AABBF94D	E8A25577
55	D96C54C7	4EBE93D8	3255C225	BD3003CD	1AB3AF77
44	BD78AC65	AA4F4D6D	B0E3BE23	7B2184E1	FAC05A57
17	E66FFB6D	51FD40FC	569AC8CA	4C8F6A7D	820608F7
50	8A93320F	4E3BEFEC	1DD7C360	C3184A9C	2129CC67
32	95C39D92	A8D1F2B0	76F9E0F6	4BC7BB72	BCA8F467
13	E4E0B850	7B79213E	D70B5CC7	66333622	B38756E7
32	DCCF99B8	21DFB0BF	319F753A	68C6BF53	C3E983F7
5	D34FC566	B8104CAC	8FFCD080	C151FDEB	40096B07
16	A9E42441	07185CC7	1A532ADB	08654538	45FAD907

Primos Doblemente Seguros de 192 bits

sec.	Número en formato Hexadecimal...					
21	FE5B2C2C	1D9701B2	DFB434C0	C1B030EE	4DAB61C5	94C10E17
58	91007C57	804D472A	E1B3950B	25947ECF	2304EA0E	BEA2E587
5	88236A0C	DB1CCDB3	A640C219	F69208A0	04E30ED9	E914BCD7
8	904890E9	CBAF410B	E07ED716	01620AB1	97A3D153	F8683D87
3	C919CB3E	DD0F4038	1EDCC5D5	FD4298F9	6914A59E	21A978A7
176	EF20F3D0	44BB9597	6421CF52	671FA5EC	60EFD901	5C9F5AE7
36	8DFA85C5	67E74E39	715FB3F0	A3D4B85A	2DCA5B4B	323B32E7
20	90E33E5D	DA5DDA5A	423640F3	835B148D	35832957	D8E75127
4	FB0B0CA0	2D331E76	AEB473B0	6C541503	1AA26455	5C126D67
22	C6A85E9F	A5B6C5A9	92A4A429	65C70EA4	1BE7505E	82EA1D77
62	AB0B719C	C8AF5C6B	5A91C2FF	5D6CB9B3	EDDD5205	629D0EC7
25	C059D4AE	D5569C63	525C5EC6	EF090C8D	80F9B176	E5911287
15	AC0B66C0	115F9E85	9A40F786	B0B29402	6FFB63BF	06A7ADB7
11	F1DC081A	503969FB	65125FD2	BDE85D61	6DD3A7D6	36EEAA47
9	8EDDE43F	14A264F5	0CAA2E08	3940841C	C5044E10	70D9B027
10	90EE73A9	0162BE10	84C96378	FCD9D738	7749D3B2	31D20C07
18	D8A3D1C2	8798EF7E	6E25EC96	426AA3A3	1FDD587F	FBDD0BC7
13	EC6BDA4E	FCD625A4	0920536A	14B9B5FF	E05D1CCF	8C705637
117	E9155024	61131F3E	B603EF04	50DF1577	DC3E2581	3B432AB7
30	9C99D49B	AC698295	D9821362	C25CCB65	4616C92D	B6EDF727

Primos Doblemente Seguros de 224 bits

sec.	Número en formato Hexadecimal...						
290	CD620472	BB8A140F	CD4C4F57	9D9984F1	9F601ED6	26027C23	476047D7
95	D83DD6C6	072554B8	389BD802	78E7E969	0046DED9	5997D1E6	864539D7
148	AA44C1A6	DDCF122F	F1504B8B	34B7848F	5CC04A23	4B5AD6C7	EB745467
86	DEFFEB35	A01F9A76	BAA27D08	0D409B15	AA252D45	4769C4BC	9E374257
135	C2C8899D	CCE6136F	4CA2CE87	BB53F9E3	B5F9EA20	6F0F5328	A86CAEA7
36	D9625B48	54E63573	AAB9ADD4	D093DB22	BFABE8A2	7096293B	C3BF9A17
46	8679A06E	D6138CC4	8C1BA621	27E76A1C	C4BD2E09	E598FCFC	86130827
10	E2AB4A02	EF46D52F	71378EC0	AF466F52	97AE8144	7736114B	9773EB77
98	B125FEA1	7234F3E3	3AD23106	F2AB00BC	D7736B08	DB6186E6	166D1477
135	CBCA6353	30363AA2	714B9A15	20990ED4	2F1C540D	4A1C565C	9CC18F57
64	B1ED99A1	C393D5DA	B30360B3	2BC908A3	AB83C792	DF1FE3D9	49FBABC7
36	C7D1F98C	F880DCEC	47722D0E	600E9DBA	92B883CF	98FFD2A7	B2042D77
38	A94B40A1	21706F07	7FFE4325	180B33BD	54B67375	8F8CC9EC	432FD717
62	9EDEFEEF	BB93E698	B6A8BCCE	6D90D09E	C949261A	23D43CEF	0AE8ED07
74	AE9DDB48	B84C3593	6934A134	7C7DF200	EDD7E09D	71ED3C71	6B7F4677
62	F7C7A165	5EC55BAE	8C2DDE98	E5E024E1	F008FDEC	3E3E6FA6	E0286437
283	9CEE7255	318A021B	DFEC979D	ED038CDD	08A3997F	CA2B1436	CD48AF17
14	F070E37A	FDF1ECA2	A4208B6B	4432FC00	3F12739A	31EF40F9	CBC8B6B7
44	D8280A77	0D2910DA	4F040AE9	48F06839	8A936666	C21BC435	B6B416F7
202	993C7B3E	BF90404A	9A44943A	D5CBED13	5FB043DB	C99AC259	2E752097

Primos Doblemente Seguros de 256 bits

sec.	Número en formato Hexadecimal...							
136	E325BE91	1C393B27	181C160F	6CEE82AE	A5F3E8B1	DA0AAC8B	1AF88F9D	A6DDC697
167	D040E956	62C939E7	1247243F	719726A2	05987ABD	37803475	211EC9B0	E0B57127
818	A0DDDE80	2E048803	8DBFB40F	27DD1ECF	AA2742DF	1782FC88	0B8F2F0D	95ADFF17
87	B1B976B2	F5A49DF8	8548FA9A	B997A447	6A41BCF8	330D8B4E	AFCFB49B	C8CFBCF7
21	AEFD6C36	0EEE0A06	3CB8067D	CDDD5857	50CE2C58	E1034CF9	6C57D0DD	B921B557
653	F2583BD5	81997A9F	34E24DD1	B57F8E99	DE7B23FB	352D929C	B9B4E6D4	0B4F6317
140	F3002D20	4900DE20	88E6B669	99780CBC	2F1C676C	DCC785A7	922098B1	24BF84B7
371	A81E4F68	1C5F7E79	FE9788F4	90ACDD4A	79E86F4C	D6564311	C45BC93A	F1D162F7
58	8302502D	45015DCB	84C46F67	68BED0A1	491591B9	558C1D2C	CA225FAA	68EA2DA7
64	B75BA93A	84812187	76A40319	5344F95C	8133634B	4FBFACAA	6C9B8B36	EEC0EDB7
326	8A702D94	53CC01CE	371D80C6	91E7F3A6	74934A23	48B0B361	64033BB7	F9400777
50	D2281D62	5F887EC8	0F432F44	53BF3D08	FB409B3D	34AD04F0	79195757	FF48EDC7
24	942F4C7A	E7477484	A9D7AD48	FC9FCF4B	3B5CB76F	A4076A3A	57787357	60C64DA7
372	D4E48ABA	0747E6B0	9A6C3E54	134B34C8	7EE53483	048F5054	8414246C	6330C407
78	A8BF84B2	B98CC680	D411A2BA	38797801	C69D59DE	94BBF42A	9EF05D72	5A64C717
203	EA4A4FCE	6DF6D66D	6C5AD785	69ACCAE0	9B991B7A	43F4E8A7	380743C7	66207447
243	B2FFF980	C62CD83A	2E1FEF4E	A39191E5	8EC11569	2154A7E9	5B628960	4D5EA2C7
702	FA6C05ED	7BB87725	B913B6F5	4B631D18	000E2C3A	B1910ACE	AF80DC1F	36B0E757
146	E7FB3F1E	1ED189D0	D1CD666D	0C5EDF70	E450C4A5	2D8B079E	839283F5	24155487
88	B0CB3F44	FE0FA796	10742DB2	CC645E74	9D371BFE	E6A4EB50	15F68BA1	539EEDD7

Primos Doblemente Seguros de 288 bits

sec.	Número en formato Hexadecimal...								
119	C22DF0DF	419E3591	EF48BD58	1D1FA14E	600C418E	3F31154F	DB7EF7FD	B59DB0C0	990B8257
663	BD95DEDA	0B10701F	7D08A623	7BA7A2A0	CF738445	77A3B9AF	6CA98EBD	2750F0DF	2D04A7C7
147	843F3D22	97C9A019	58E65BD2	51E7FE59	618A358A	49AE5104	F4F20EB3	0E6C5CD8	F94C9FA7
607	A1CB9E2D	B1D07A25	5C1163BF	7F600B9E	6AC409FC	7786E5B1	6FAF1BB7	A17E07DA	1267CE37
310	DB9226C4	1BAE76F4	7EAF7701	3679917E	69621035	F4A07256	C34C1ED3	DB7597BA	2287BB17
254	A7A52853	18F2C546	20CE4DDB	32077B4C	083FC88D	E8F53C0A	445B3028	6F5BD217	54CB0727
1933	8C9ABE1B	2329A6A2	5CBAF531	BC2E50B7	406DF8C6	FD733924	0F9F3676	F7FA7575	D8834307
464	DB4B2412	5D6E4B86	205CAF37	BAF7951D	C166BD0A	E2D05D40	24C19291	B9EA0865	071CBC67
224	8ECB471F	5D67B72E	5CE0EC66	20485136	CE87CB63	B4D09FAE	9134025A	ECB8B07C	81FF4297
497	C9C72D88	B4E15DA8	E0A88471	D2B4D9B4	98CFCF2D	6C564BF5	0A50365A	C86B9D52	552C2AF7
556	A2009C9E	560CB277	2545856D	4D5B1969	0284DB12	B18936A2	0C969C92	DD1BA661	F0EBE827
50	A095AD10	0D7AAC41	1AFDFCFB	126E9DF1	D183D463	CD2038AD	D9673A21	066EC22D	06D1BB27
127	D75E7730	9485CB67	C84D6062	082F1DA7	6C3A5FC6	970E9486	04675E93	063BE7BC	BE9E5227
562	D15D4E2C	8377D359	99765F2B	AF337DFA	769243D1	1EFA87CB	1119A335	6646D4B7	ADC62AA7
160	B1827139	FB1A4500	7AC229F9	E4C202A2	3F91E8EB	4C18D52F	98A16CAD	966067D6	8C55F217
46	CE703376	B66B6A27	71066ABB	BFE0497F	637521DA	F24C3F2A	38F48039	10F1BBF5	03268B37
535	F6B6055B	A66E1EA9	FCEAFCB1	FE3E9C7B	E521FF83	6CEA762C	9676492A	914B18F5	17839157
538	DB8B2733	CFC2B8A1	6934BC3F	2FAB1940	EDD4CD1E	43E4532E	5EE6718A	91C65BFF	D93081F7
19	81D228D9	6133D9CA	CBD844C6	1C6D82E5	D024BFE4	BEBFEF27	A1E6117C	32507AD7	76E71A77
565	95538CDE	DBF9A3C9	5AE6A403	8BB79288	9076ED44	EEF09149	0655991E	EA8278CF	D4ED26D7

Primos Doblemente Seguros de 320 bits

sec.	Número en formato Hexadecimal...									
279	9857EB34	C43A4116	D8B76688	5B30BF0C	91F55DC3	EABFB486	FD0BD232	0037C5A0	83283748	5C3355A7
1429	CCCD7AF7	01C41E19	24203191	48117192	12744529	0653CD3D	ADA90C33	C1695300	EFF16F0D	0F7E2A57
295	D6DC58D2	576898BF	FF60EB7E	79367D78	D35604C7	EAEF6A11	4FFF91D8	C47F48F3	28588EFD	70BB1A57
291	D6E70636	20C0D016	1BF02E45	D6C8810E	64711CDB	A36814D5	ACD5188E	10D27749	77554BC3	769A7897
568	90BBCAE4	997EEBCE	661242A6	E1B6EE4D	0F8F8648	B7AC5E1F	C2EECB8C	C3C1F90C	2E2A58DB	2F5361A7
263	B9CD41FD	E5D6CFB3	577B6704	1420576C	40C4A6A4	11F69EBF	00866D10	2E04E339	4CBC82B5	6F866857
296	991F2E62	C70A5D39	BB5712A0	DB1449A4	6AB0BCAE	14BAE842	6900F0BA	1194816C	765FF8AE	4739CF67
16	B06F5F5D	80046A14	CF81743D	3BA5B90B	F44FDAE0	59206D61	3725C8A8	10C0D120	2ABDD97F	5913DFC7
394	AC5D2A11	8142604B	216B2EE5	EB19AF64	D603E248	CA3E9CBD	6140429D	7EB05445	8900C77C	70B73567
596	82F51265	6357DF1D	FDD9C6C1	B08AE1CB	D2BBF559	9EC0DB5E	3C17A9CC	1B883F73	3E4C792F	D3B79D77
178	A39FBBAF	CBB5DB6E	C2103942	3E4DDF45	ED4B7109	344ED9E2	F4D89170	A0E18436	4B0F5C52	B9F17E37
287	821CC52C	59306426	A4A364FA	21C6CFB2	1A274A2F	C1049544	865DA5D3	FE326D2F	DD46C28C	F720B917
707	9C73B64C	DEFB637E	31408CF3	A34807A2	F68FB77D	9917CFDE	456FDBEC	619B6ED8	A24A0398	AFF64F97
72	D8B5DAE2	A0018D57	53D296B3	0602ED2B	480F861F	C561DCA7	86F5D1F4	BD333B67	1645EE40	14F6C907
1120	BAB2D590	09AF1353	1364DD31	EEFCA9F0	345E878E	78FF0E70	71D2A45E	4EC19F6B	1D1C5FF8	C1328E07
123	A9A54C3A	15DFE93D	2D08EA83	D14223E6	4634E19E	1822E55D	F2ECE8CA	CADE0018	615D8F3D	44439D27
798	9755092B	969F8201	C31E0271	9C386847	485698C0	3DCED71C	1D491436	9C64EB8D	768703ED	1C4161C7
110	B0524A5F	E55BC94F	74B552CC	E1B6181F	A4A913FB	4D1CB24A	B5B6BF52	6346BDC9	8D3629DF	FC2D63D7
235	956B5B28	8AA9F37C	52634CD9	DD290C1F	1DF99F93	29C0A776	9D376AA0	FD8D1C7C	DF444823	41CB4337
721	AAF7A564	9333DAC2	79C809ED	92D1E715	A60F99DB	219BBAEC	9DBC4B64	E15ABF0A	2EA0507B	2E252CB7

Primos Doblemente Seguros de 352 bits

sec.	Número en formato Hexadecimal...										
1582	E47C2BED	B9EB1C83	BFA1A5D3	15671BC7	95B30BB7	26C10B73	043DCDFB	EC59B27F	D9DEC4AB	0C8E71A0	61C497E7
10	83DB0A85	E5A7AD53	11CE0B25	29471921	54D8BFC4	CF60F3D7	B3B61D30	C972C042	658C01AD	E73498D4	5DC78007
2533	925D57DD	E1EB33C1	71546BF6	153C9805	4C9C8A6E	C65A09AB	8567F93F	BECB5EDC	7DAA8DBC	BA700CC0	E8F70A37
228	D92AE57D	CD7307E2	04E953E4	876B658A	2AB05761	F3A19518	F508591D	602714ED	3E5A2292	D0372F6F	E68F6627
2280	E0D89691	281D8AE6	420C5EFA	AF23B841	5BA36FB4	7106071D	2C397871	70213A98	84ED6DD5	7D185970	A673A8D7
798	DF062BEE	4704E646	20870164	AFAB3A43	B22566CA	7AA2A20B	C6E3449B	64AD6693	49D2332C	49AEA885	BCC10217
3483	90567E01	2E8F08C4	F70F1BD3	339A70D0	C53C34EE	618D0A13	35DE3497	71483F76	C9A18D9F	2152FBCF	E4DADEB7
3179	B70E6542	78F62550	84A180B1	66505304	43C0AAF8	C06051B3	83F6CDF0	D353E7EC	C913EB70	EFBA62BD	2E734B37
1002	8737D997	84E2E824	FAA4683D	D2C0C259	FD5B01F8	F33F8B9A	83C3CC7A	A63A0F27	C18CAEEB	BE9C966E	E3062BB7
736	A6178A0A	6CAE105C	BA1CC3FD	D9F20DE3	929A3813	B75D766D	C88B4690	D0B0CE41	A52BD2E8	AEE848EE	6AF66137
269	D57DAB24	9249ADDB	5697D5C2	23EF26A6	0C2CE439	45619D47	C9D3E63F	BD5E13D9	D8A91AA5	4727EFC6	1DA371F7
310	B5C67B6D	2B66A2A4	B0634794	1FA572A2	6C66F8B3	D92949E7	2306CF0A	3C3117CA	6101D152	6E828875	73F846D7
333	E19151A2	FEF8E4F0	D5F9D0D0	4F8D5537	10FF3C25	0AD1A3FE	FDB05597	B88905AD	C62BFA40	2777DFE1	25B8D977
708	BCEC07D6	8E75BCDB	FCE55E23	0DF8C6BC	09B2E9FD	3A40E05B	AC477216	0E657101	30D1CD55	862ACAA8	7C553927
1093	B1DE4A8D	A6BE81BE	1C3FB1CF	41D926E6	95A60281	3A822AFE	4020E270	6D83496F	613BADF1	90894343	814B4DC7
435	ECBD72F6	D2E45AA8	FD45CA81	8E29BE4B	E9757E9F	E0D5F9E8	424C9845	E7E312FB	DCE145B7	A0ED852F	C724F837
540	CF76329D	45593D0D	570C8C1F	78EE172A	511DFA44	E4621244	EC147806	10423C53	C3F843B1	97837488	A942BF47
1162	D8765AB7	4F601364	D06CE2E6	01CD9CC6	14AF0EA8	2F2F904C	D9818675	9ABDE1F6	B9ABCD47	931D4D23	1805D237
64	8C89DACA	715A1A58	23DB3D8E	9AB6C5AF	226B7C9B	D80F55D0	6BA3541D	DA8756CA	27280BF3	CC931FF2	85DC65E7
2257	FCDE577A	C6227C59	F30116B3	E61B5296	AED35E1B	07E9D7C4	58AA2F3F	0E23FB86	B062CBDA	B80CF5CF	99B2B4F7

Primos Doblemente Seguros de 384 bits

sec.	Número en formato Hexadecimal...											
178	B3DFC41B	970BCA22	E2F4D9AC	65BC5399	1E9D2E0F	79ED0C53	CD9187CD	5DA8A703	CD459839	754D252C	D7E270B6	62C29027
188	DE097602	BAD4B425	7566B7FA	9E8CBB3F	DFC17530	D987DD9B	30647A22	43550409	0BA31C6F	8F319514	00F79BBB	3B2F9647
1879	8C21AA2A	C56C66CC	B9E320EA	2B87AAF6	82BEFFB0	D0079016	1CFE88BE	BB0E4E7D	B5D83A28	2D479876	18FDE574	41211477
153	A4365BFD	EF29E149	C2352163	F4A24B95	DCD8F88D	C0F61B7F	9B28EA3C	03E54413	11828380	DA8F64BD	BA75A5F3	10EABD07
4119	91DD8EA5	6E65FA95	1FA69F58	04FFB964	D99D882E	4C540813	658D9A90	8476CD6D	038501EC	75188F28	707A843F	243F80B7
2582	A8A5CF61	FED7F5A2	498B09F5	E6AEF8AF	0C1FA91B	FC7036F2	A8ABD4A8	013B45E0	E428DB3E	C934F567	573A71AA	577C8D57
891	D15A840E	7B9C30EE	A9CEB927	E9E09C40	E1A2FECA	F8D5E83E	7ACC4DEC	F5C1256E	32A85FCE	B3F33485	2D97A059	E662C387
1035	9EE9CC34	1DBBA349	C957DA24	CBCE6D2F	5DB3D5E8	C4463F9E	9940BD8A	640C86EB	1A216F04	0EB44D58	713FADE4	2A41E3B7
966	DEFE8819	A6AF56FD	91F431B5	AA332167	AE68364F	D14DFBB1	40D10B7E	9B06A779	1AD5B886	D3533B9E	F01C9361	18C3FF67
1071	F1896E9B	4E80586A	0203A30D	5FE53210	224965D6	EB27C34D	FD222CE5	0E42FD03	1593909B	CE579D3B	D71569E7	39DCDB57
979	B568CC31	2FB7DF45	43E9F93E	B8346F49	38C0AE55	CD922D13	B1F6941F	88666726	ED8E580C	2ABC9672	E7727D87	48A3C567
713	B6BE7CE6	01A17665	79817BA3	D76F2C5A	06FA6C47	90BD8AD4	2D05D7B2	7B38AC46	6B2D8055	236FC9F8	4BE66FB1	31192F87
2642	F79EDE76	6CCADD1A	83FF837A	2F75115C	A5FC257D	E859C689	7CD3938F	7567C498	A3455B1D	11FC5573	68DC101D	41F7E5C7
1727	FBA4841E	72986762	D853180E	262D5B77	1DBEC82A	0B933F68	6F10AEDF	35A12031	356C5524	692D6D1B	7C8B3B9B	7B90ECA7
1051	8198B08A	14A73EE3	67B14359	6F9E358F	31A6FF1B	801B1A3E	B492FB96	888C860C	C67EDAD5	C3C34D61	8F8841B8	5295A2E7
1364	882132B4	A8BC3C19	8F0AADE9	B82E569C	9F8784A7	626CDCD5	F5D45CDD	76ED8A74	8273D408	294C5E35	750DF3F6	8312CE27
2255	E94483C2	4AA5515F	9A71F7D5	634CC91E	1C3923FC	9429A2A4	90E77DA0	CFDB791D	A3ABA626	FDFDE9B3	FF3612B3	BC826CC7
1585	B7CB2BDA	C6222834	78904170	95E6770E	BB97ECE5	B6651518	737D9046	388D6F0D	3AC426DC	05C44CDC	CD42D933	B4753E07
1068	BA57043C	779960FE	C12F995C	9EC80707	D86BBEB5	83E3DF55	FB6C295A	4CFDB79E	31C1C1E3	34CCB42D	E94570B1	E437D1F7
1827	C175A61C	9F5F2A4B	6242CCC6	E9AF92E9	18119DB5	5AE775C5	C3E2C8D9	AD95EC88	6C5F3442	E4AF6511	CEA66DF6	AC56C457

Primos Doblemente Seguros de 416 bits

sec.	Número en formato Hexadecimal...												
583	AFB880C5	DC44D45A	E2D67B84	598F5524	E8520533	6CDF544B	68EF047A	F91E4D48	76E35633	FCD045A7	AFFFF116	0005A1A3	92CBFFC7
1802	F08106EF	3275847B	D80A2ECE	473F950F	04EBC226	90F72CA2	658AD48B	13654632	3FEDA2AA	3BC507C8	9D9667C8	584A2E1C	BF5F39D7
318	CE627159	C996CA17	FF55C7D0	071C0164	3CEA3078	F578609D	12934196	2B31FD47	C19C8448	94BD9430	1A109EB9	E8799E00	B7359EF7
923	BE23A6FA	2A18D090	F9ACC50A	188E4470	DE2D6569	CC60A4F1	E58B0419	800BA637	7B54B51D	BB6A1B33	B6A4DB9C	0CE70BAA	D5352D87
866	F23DB3D3	D9A82191	2FE654D9	7754F1C5	AB7D130F	456791EE	FA0214D3	62C0A763	83DA0319	F3EA1A20	D934F82A	406AEA57	024DA377
3654	8EC9B6E6	51334539	C06444C0	09F6AB11	9A04A42C	0266138B	3B987490	65DD6B3F	E24CD0A8	6D07AEB3	3A5E547F	C37FA6E0	37464D27
2520	A980F252	2195D376	DE7299C9	001DC64D	C2920BE1	35CA1C89	DAD9390E	063D10E6	49C74C69	802690F2	E70FBA67	2BDD9BCC	EE6F6887
3012	DA8E008E	EECD079A	C98B2766	08C8EC03	EFE3EDBE	DD750985	7BE1E022	4EC0505D	4D2B6E06	CA5A9695	8E4455F9	0BAD0518	131F66D7
3893	96F5A144	4C2F26DA	5A2D0ADE	A0EB6003	11188A8D	6FA76ABE	987184F1	18FC03A7	716A1F5D	27832DD9	B1BF663E	1E537269	BF974C37
776	EE4652D2	BF953BDF	EE910F11	72ECC34A	278E5F7A	A4D730DF	A015D9C6	3E960CCF	1CF4DF33	FAAAFDBD	07D9F90F	5E2AA884	9C82A717
276	E857DB43	36F1FD3A	6953B0B5	BD206538	1593D13B	1CA81522	B75EE0F5	671C734C	7E79B3AB	3125943D	1A9AE3B4	07F2D243	1E148497
2532	C9DC4673	F9F5A846	C1254C44	5CB052F5	211D1FA1	A11317B8	7F125C26	2AD3E9C5	D0FFDF44	3C8F58AE	63AFEF92	9A01FF99	0BD581F7
379	E79B52B7	FA92026A	80D7B2EB	C2A6B742	7138B3EC	B98A1147	E296934C	2E7E89AD	74A8E7B0	7155123B	E6EB239F	5CE6079F	EB062E17
1488	97D94739	48FC9F95	A14F5F6B	4CD612E3	864DB452	46E5AEFF	C6A513D3	6183171C	1526295C	F9189464	73844042	92E3CB6B	CBD40797
2294	BA62CF8D	FE3A35E0	DF702513	4E809907	081A01F1	F56BE533	8E90EC4C	84A22005	27C55B13	C0446C6B	028BC06A	BFD51D23	BF3F5737
4032	B7D03394	A3C82CB5	11E5F81C	99FCFB36	BE092CF4	F5D7097B	B4648CC0	B506B61E	719359E9	8C2380DF	ED9E4FF5	F184B211	78B04B17
3355	F10E1811	2593EEB3	05FD7FF3	B414AF63	260F8CC4	B5352220	E7907C2E	6EEF369A	DE4FC995	69190C7D	26A40873	C9BF1A0B	CF020427
100	B28C22D8	3CC1F98C	0C4D76EC	D317C2E7	52B99099	87A35D25	4781A25F	4C523D0A	68B5C3C0	41BF8FA3	466F4281	592B1ABE	EAFD1BA7
1138	9915B3E4	BFA28420	7E71FC0E	4CC0DCE5	EE86FD3C	A0A379E6	1C8685E3	F37B73D5	05B2E23D	5E9A3805	C0FB3C71	40A115FB	D6961A37
2871	AA4AE0BA	95DDB4DD	0DA1B4DB	F689AC51	DDFE400F	67F279D3	BD765039	F1D90BF1	A90D7207	FCB87351	7BB2757C	FC9877E7	C7AD6867

Primos Doblemente Seguros de 448 bits

sec.	Número en formato Hexadecimal...													
938	823A9A0C	0137F7DB	E1B97441	C8723D75	A42A62F4	7D8DE8BD	F63E96E8	86F21AC2	03F2257B	4BC32EDC	7DC309F5	3F7034E0	BAF59A6F	3B71D2D7
2635	A1F71B15	2CFE6A28	47C78009	F25C34FA	C9D409E9	A1993F34	55143D34	642508C9	A5C4FCFE	F75FCE25	984BC6C7	AD442A96	99FE72F6	71D5C8C7
4410	E925AE31	84B0649C	1BD91B4C	16254D78	C36ABABE	8AE46514	7350FD62	3F1A7906	FB0BD4C4	E3C51963	2CE2B670	8400BE15	546AA1D9	ADED0FB7
6750	A34BAB9C	22416110	F851C7E8	848CE4C6	FC32C3C4	16304B81	B14ECC47	D5CB9B6E	1CBA49AA	DECDBDDC	0AA824E3	F5B7F42F	3BE25B2B	8775A237
1861	CD2A74CE	8888E550	65462B82	072A43ED	6DA8AE53	06733E18	EDB96869	585584E6	F99C7EF0	286E46FB	0DBBFF4F	C314C1BD	E02915E3	9C16B3F7
2455	83F333A4	1E919DC9	08E6E035	8276AD15	A0849B74	DBDF695D	2EB5449C	5CCD39D8	29ED05BE	E8C0C23C	B744C610	20225739	C0F96E01	2D24BFE7
3598	CD9DDCCF	A342166E	4772B414	7A728972	D871230D	B370598B	9083A3EA	1EF1FDFF	16EC5190	19A9CC9E	BF9D1A23	38500007	F63BDFFB	A2087247
1260	CE63C7C8	921AA6A0	EAF6A13B	957D085B	7AA83CE7	56131625	3CB8EB85	95410398	9AB71D68	544E3E8C	3CD7DF6F	081EE0C6	1CE2CF3B	A9B9C2D7
5370	8259B528	6B2428EA	F40EA78E	EBF320B6	A9FAC9DE	C084598A	055DC7FD	7BD63EBC	44C6A3FA	9704EE82	45B29CA6	F8E43C74	ED1CE88A	77B58807
1998	B1AE9509	CD53D93B	A08E8C8C	7B9CA873	7881B2A7	51DEAF12	B33752E4	A2AA3BA0	306B61BC	6BF9318E	099F637C	CDF9DC75	318B8AD1	52918D67
16397	F24F6250	A2666E89	8E354C1E	B76ACCA0	CF6F4423	1AFD00E0	0D335082	4E97940E	9013B915	A49947F3	F0DA3D71	4B2A1361	70F35853	B149C477
5329	F74BA6B1	F19DBEF1	928F2F55	CBF6A4D3	2C520839	4B6ADA39	94B23F6C	D6706422	BE718017	BB5A838E	17505D2A	39C84F31	81791731	F1868557
2287	A0C8B3C8	5B6BE9A7	2ABBB004	5C4662EF	71AF9DBF	91BB4AF1	91B9B3EB	69E69F6C	4CD9437B	AF2BBE09	174EFCF3	53DC6061	E5EDADC6	FB410857
2418	E3514105	24071746	2E26D211	1A55F1D3	13484EA6	BD02AB49	2C23A821	1237DD87	EBD20145	6B9E9B28	84EA1913	FD6AD796	DCADFF17	6DFF4347
969	92F825F0	EB63F6D6	1FBB4B65	946DA059	D639DF19	322B07DE	99A36156	2CAE98E0	56F27DA9	7A8F141D	39CD3AC2	5E26A2A0	9E11E1CD	E71D8297
4015	97797BC3	1A5BF70C	8DA40EFC	63EA9E87	62C5F574	4639E991	058CA739	31C9C789	D3AF684E	C4E4DE44	709052DA	FC55546D	42B9682B	7AFDDD17
10881	8D91459D	E97210D1	3F211853	2194ECC4	9FF3FF0D	D062745B	542DD4F4	80891D8D	9CBFB114	17FAA298	0EBF0B28	066DDC25	6003873F	44AF3467
4024	9279F01D	B1190B0A	BDA82FFD	814AE926	6C28E0BD	805C7914	19564F1A	80D8EF0C	3F345767	66A313E3	6156AD8B	AFC49DA9	FFD083CB	DC3F79F7
2170	EB31FF94	E604F5A9	A0CC436E	EFDB54EA	79F7355C	17A2A8FF	2973F377	728F1799	E8E17D23	883E49D4	88E09D52	FA6AA66B	D6C0E4C3	94DE46B7
6674	81523922	95C2C534	3E87706E	8061785F	87288982	22D3418D	48B4BF7F	9784E7A6	6FF25A1B	D4A85E6D	713EF1CC	6CFCE717	B1151441	B3FF52F7

Primos Doblemente Seguros de 480 bits

sec.	Número en formato Hexadecimal...															
3398	EA92DD40	D2C5295C	1C9CD60D	C9455ACB	F4AAEBC1	D5BAB680	E82FE9D9	1B8007AA	B8E6133A	52571B21	2FB1895F	D0BCB6C3	E0F5C5AB	32670913	6384B7A7	
1083	C378AD0F	FCC872AB	102232C4	9E2087F7	2926597D	F4CD60DB	79FA0101	43BA6839	CFD2FB22	84E9EE10	B484CBBD	A697A376	9049E18C	F3C5E914	91E03447	
1266	A3661A62	05026596	10D45E5C	BA30CA3D	F500272F	2DD90912	F411B7C3	C2E77330	A307BAA7	FC1743A4	9E248006	DDD89DC4	EA9785AB	B0ABE9FF	44AE1267	
7697	CE9C940D	9AB40939	C1A1294F	1C700DE3	D53A952E	D0CB171A	82668637	75BA639B	F439575E	E7B9DE42	AA31A172	657B86F4	C397558C	704898C5	ABCE7957	
2558	B0DA9854	77C8AD45	998AAA84	2A665AA3	69AD5E3A	D4E591B2	BFCD511C	726E62B7	8ECE656C	F00E01E7	3FBB55DD	06CCFDF4	A4EC26E5	379213B1	F8D071B7	
554	A42D4080	C8080E0E	42D0C1A7	6826DF49	E621B975	93420463	3EE0449F	2E25297D	F77A719E	1374E69B	7C0CF330	1246A122	947A4024	92C33B7D	5E695977	
6677	B9F981B7	183440D2	36C00AA2	A6AFB567	01333EF3	CE75CCE9	F4C51F6E	0C270A32	3583101B	7C33F4C1	2C194BFC	F603C533	3CDFD073	9C071218	BF39B187	
3310	CBC26B89	DA61A3CF	92D2055B	73DF495B	B249487D	A7BABF23	2037B146	36531C3D	C045698A	CA46F6D4	09816A25	6828AFBB	8E1B4640	5EA01E48	EABE0867	
3065	C2A3600D	C02E84B2	7D8434C7	6EC1ED83	95C4AD03	975ADB2E	F0E484EB	982A26EE	25755A0D	32C11EEF	1AD15518	F5912BB6	0CB67DA8	EBAA2F6D	B9C68B27	
9011	C8A82B07	6FB6E764	FDEF323D	14594390	AE2D168A	F9FC68A5	2AD709A8	1EF10340	3885DB3E	979558F8	76C06C05	420DDBF9	CA03C923	B77BAE0F	CFD9E6F7	
8093	A70CF71A	EA1D8DE0	F09FDDD3	E39130D9	0171BEF4	1B737CEF	7B21E84E	64851605	A9F62AD2	2089DC8F	1ABF3089	FFCB8822	9D8D1D56	F39A1A43	14226B77	
4106	FC1B3499	9CDDDFDE	3C618125	0B9A631A	3D0ABEFB	AF64AB80	B908A611	6EC451D3	B75E823F	E7C134EC	DDA2806C	4CBA58BF	180F4B40	2245E38B	32E0DFF7	
4102	A548B97A	AA790242	855BADCF	BF9196A7	21C5197E	CF7483C0	3E8C85D9	22565FD4	310DF158	A574E011	94A6466C	6C2AD5C9	32C311BC	D74C191F	33516327	
6274	E47101BB	CC7BE3B7	38BC1275	85F8C671	D5FE6C72	5F06020B	C65945C2	9323CDE0	E8749D3B	FE41E9C0	08BCE39B	30010F84	DC880DC2	C7F93DE3	7A886C27	
2021	895049AE	C661A843	B0AAC51C	CE76E61E	A647F0E2	97FBDF2C	BA38074F	E4B866FC	C31DDAF0	419E9770	F2CECC81	7DA6F220	E759156C	52112980	46BBA9F7	
1540	829D8432	07715C9E	299F30B2	1BD141E9	73E57266	B5569329	5F200531	72EDFEBD	8A9702BA	3D34BB5B	2DB66358	3BC8BFA5	85B167AB	F8E8F663	32F8C007	
13323	93707DC7	4FCC47E8	77CEA6F3	C1E15123	D2168C8A	3DB63B38	FFB00B00	0C262326	B4E56862	C3FD2857	A746D53D	5BA27CFD	3D451964	15003ECC	4B467E57	
175	A23FDEAF	F1559D8B	BD17D716	77F7E900	A157F00B	22636B47	0A32D879	4B3979E5	1DA4DD8D	596EA2F8	D14E3D9D	95C6E40D	3F53FAB9	EBBF2697	CE166347	
5246	C5ECA06D	4D2A84B7	B5BB4EEE	C59787B7	CCC07820	666EF662	67B94473	DF2913CC	F8DDA408	E279D07F	EADBEC3A	B41D8478	031A1781	0FA8297B	4EB324C7	
5571	EA26269C	7ADE4A4F	8070D71C	6899524A	7C62CC18	451B4571	5CAFF906	432CCDDC	D2D6CD0F	5673145A	E3879885	3ED491E4	5952D1B3	F4034543	229A41D7	

Primos Doblemente Seguros de 512 bits

sec.	Número en formato Hexadecimal...															
9753	CB731CB7	A4FD0762	B4B84188	5C568132	97614AC2	42DE91AB	BA8E77AA	73AE0198	0B843D14	BCE12EDA	02184287	11D1BFF5	F5755717	72193864	1FFBAFA7	FA63A587
5679	D9425351	C86B5378	DECECFEE	9AADDE4C	5EEF6806	AC70DA6F	CEE954F4	B3A34C1B	4FBE01C0	468FCEA3	0E169A30	14934BFB	28AAFC96	2DC9CC9F	4795ED7C	C07E9267
46	B028460B	F83AE823	E5409DE1	2CBAD445	E99EFC44	5EC68192	E9836F18	07DB6EBB	316D44DE	7F739F22	69060428	4FEE0FDB	D6F74320	C444D25B	352938C3	3829BD17
34708	95172CA8	2407C6DD	DBB0ED01	2320F6B8	5B824409	7DED0012	3E4700E0	97D1EF99	4A56B03B	627E0FE7	0AB849A8	AF0BC30F	57144E33	AC6CB811	BF7EC5BD	C5127E57
4228	ECFF1BAD	3B5E3E83	7359E322	C978D62B	F0887D02	D619BE59	2983BD42	C83551E4	FB3D5663	79913F85	00CFD614	551C08CA	1FE2F3B6	EE5CEA9B	5B2EECCF	4E6560B7
6645	DB5E31D8	DF221FCF	5470E9B7	316A2B0D	4404C568	B7AA4135	EB8AB1AC	5AC7911B	7B6BC119	808D0765	59CC03DE	86B0DC3C	DE569A0C	2E683127	D49266E2	7B197AD7
2050	861D4C9E	47F3683A	95B2B06C	9C8CEF2B	5D4601DE	7DD77541	44489B90	894DD2D6	C98914A6	A32F522E	A7A7BF07	69B33FEA	19AEE379	362D7A7C	C060142E	AC2EE527
2115	CD61603A	4D68E8DD	9FF03114	FBB4509F	B6BCB386	A2382399	A349D379	9E82C1DC	3FD8819B	1DB773FF	3E5CF004	C5B64433	C0D0E572	D16C7F66	7B51FB61	9E35BD47
3776	C4C1A02D	C9CBA48D	533F44D5	EF01A18F	0D39172C	856C3C73	F67F3ED4	D765750B	F5AE7468	DAD7A12A	75CFD16F	A03D777C	8B681D93	692CC01D	2F4F8716	EF203DC7
9655	BF5B5BF8	D79189D3	54DD66DD	47A4E9C7	FBCD7DB8	66056D68	7CDDA86D	466CD642	5BF66065	9BF49264	702CB1A4	D6595DDB	2703216D	3CA5DA98	2D1D2F67	DCDB7587
12461	88B4888B	CA88B42B	02885508	B1859CB6	ABF33A6A	BF216FA2	F0929E69	A81DCA85	B73001B3	022DE659	9DBF7525	2DD53A5C	8D6167E1	004EC938	E9AE09C5	A8001307
18013	F5D7B40F	AF7F7A39	91384CCE	11FE59EE	A6687081	B668F781	59BAA79F	5271CBF4	A7E10D59	C3BA070D	3C5E489C	2A87A55A	2734AF90	613D0E1F	BDDED667	EB162C67
20275	D2E82056	675C359C	C9654CE8	5F6402EF	519198E7	84A54FCC	29C24C98	398AFB96	B2FA4C43	F3B8696C	38C50C2A	FDC4A37F	83D544BB	B4A3F158	3B9A0BE2	A8763237
19500	A47475AC	328DFBD6	367C4A86	9A71BBE8	3423D88F	1F212D68	4627C62F	A72ADD32	1CE4A700	85E3E611	F43ED4B3	969561DD	13811355	C337F525	F59C459F	23BEAA37
12274	A04B2A6F	4FBC0373	9A98A8E0	2FC21A23	67A63820	C4E255B7	F2D2252C	79319124	AA4EEC02	D80D0FC3	C514A008	496AC0D9	4FFE65D1	B3866FCF	E5DA40B5	DE98BE57
5752	C726178B	43329A71	B3BF630B	1870CF24	4235107C	2E1E5D8E	11302361	7F8BE631	C2933821	8E200359	27A8614A	BF1A43CC	C4577375	6C890004	6275C346	EBDE4C27
2470	E6A2D240	6AEE24F5	864A0972	724D2442	E513D8EF	C12CA2E5	11FB0B25	D07B97AC	7C1E3D95	29341161	38720434	A6B68BB6	C84619D0	955D5C89	610752C0	2BAE35B7
5618	9FA56392	2660246E	0E36D9E4	CB6B12CD	6CA5A398	D44B68BD	C604DDA6	D3A78281	400F7599	AA9D6271	29DB8EA7	D0EB4B22	B24C1EBD	CC330DF7	B18618C6	CFFBE3C7
23478	B99EFE20	7C2B8F41	18BA7EAE	2CC81F9B	6443E85A	A813C899	D6FBC59D	F43F6D52	CAB63913	B0631C56	D7E2C7D8	275EF0A6	0EACDD36	301D492E	58C602B7	E78DBA77
27077	F72C3B6D	3F3E5FCA	20720705	5AF5EDCF	A65A99C8	C925D70B	5845B89B	FBCD6101	20685DC0	FA61FF48	BF3C4F81	B0CA2FD7	22F5E24B	005D49B3	4A9AF934	481AED37

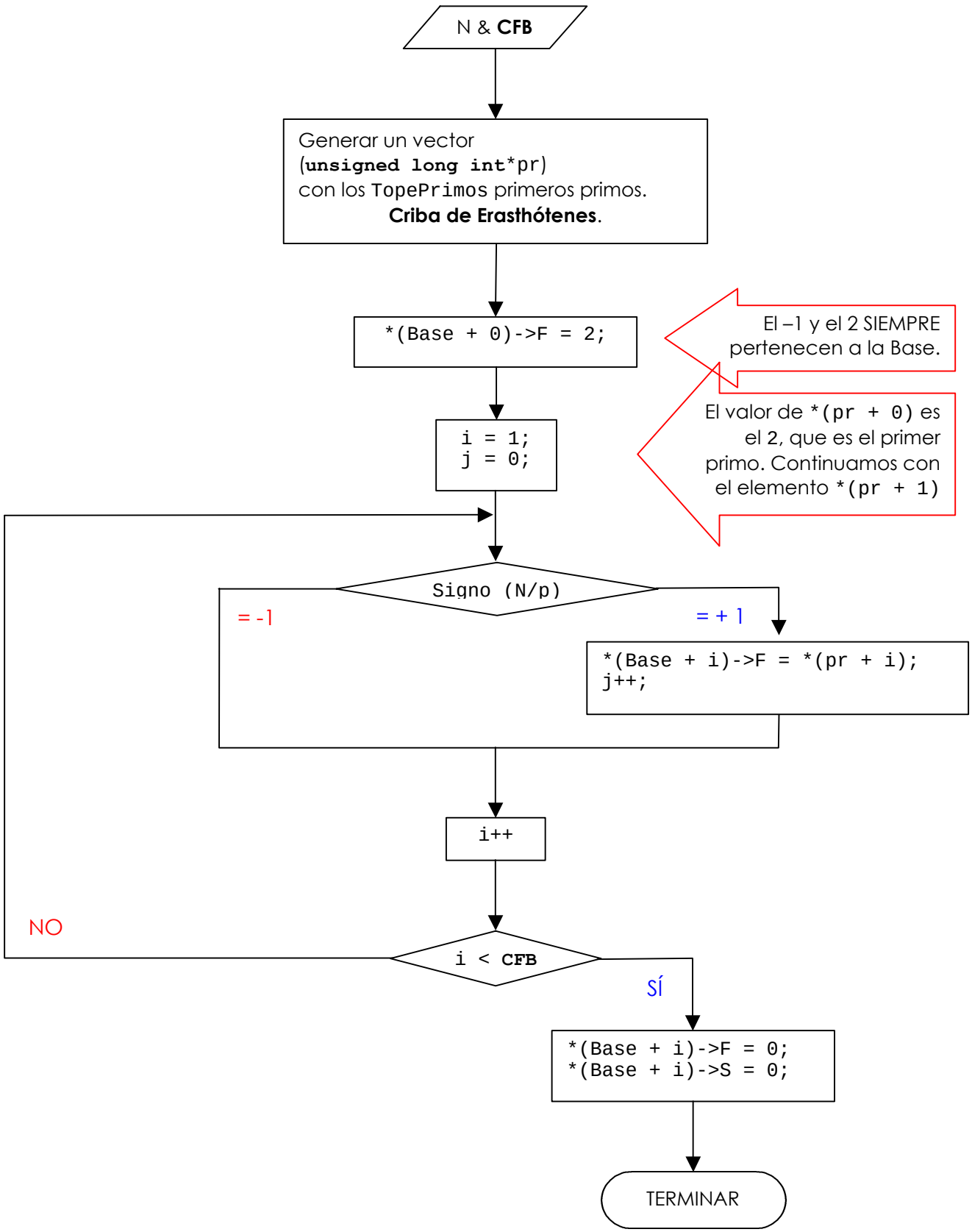
a

ANEXO IV: ALGUNOS ORDINOGRAMAS DEL PROCESO CFRAC

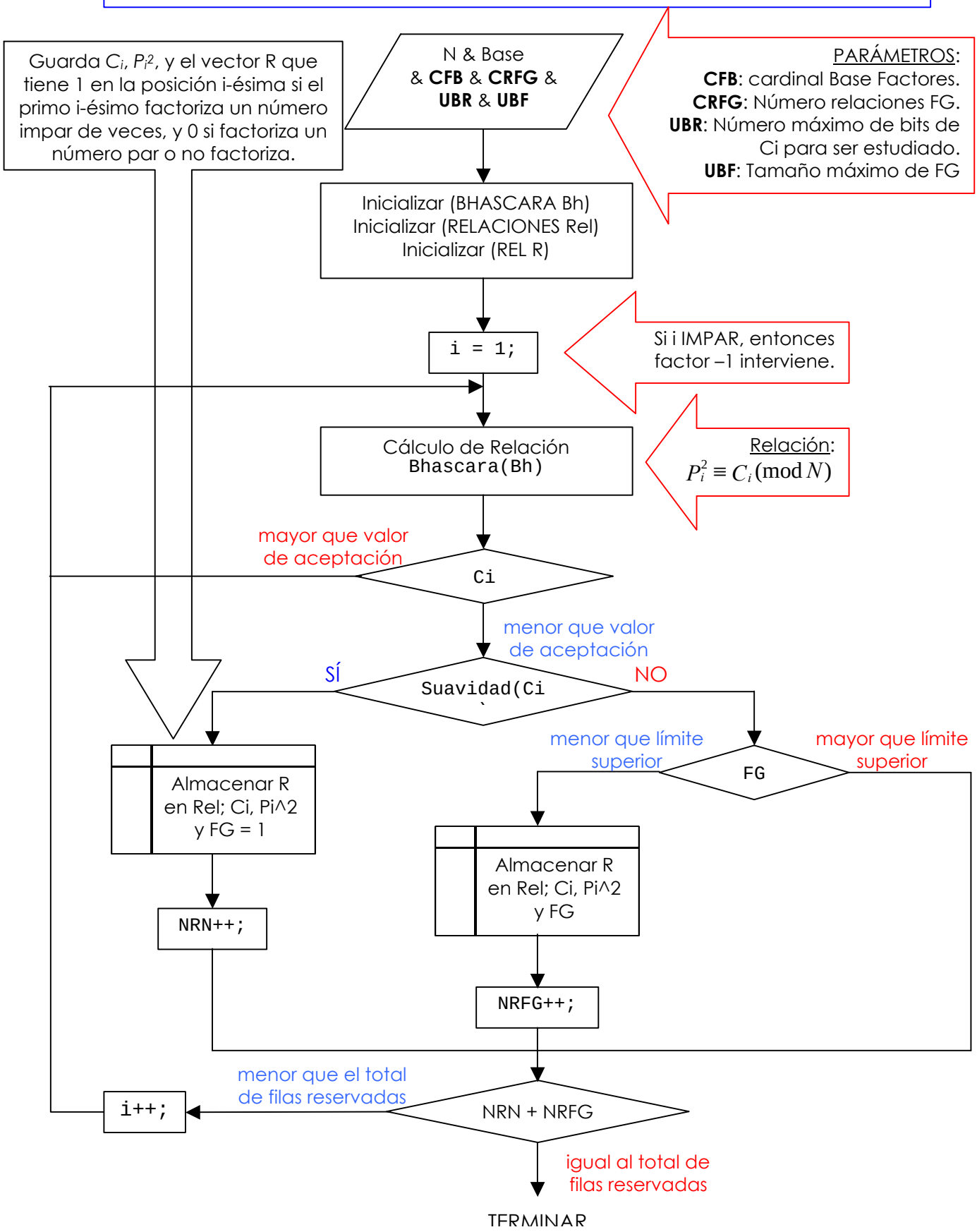
Recogemos en este anexo los diagramas de flujo de los principales procesos que componen la aplicación para factorizar enteros y que utiliza el algoritmo de las fracciones continuas. En concreto, presentamos cuatro diagramas:

1. Generación de la base de factores.
2. Generación de una nueva relación, o par $P - C$.
3. Determinación de la suavidad del valor C_i , del i–ésimo par $P - C$.
4. Un diagrama general del proceso completo de factorización.

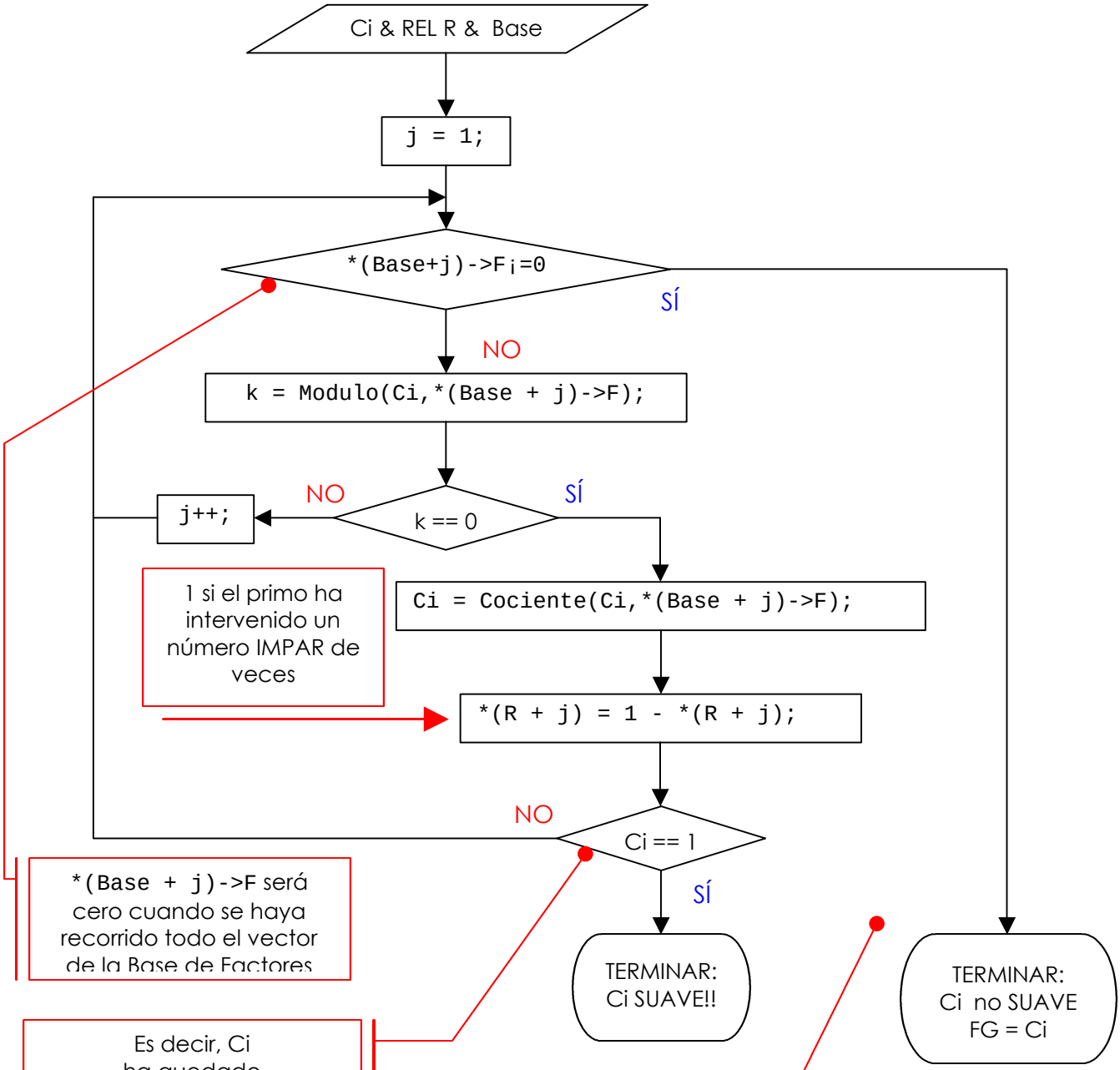
GENERACIÓN BASE DE FACTORES



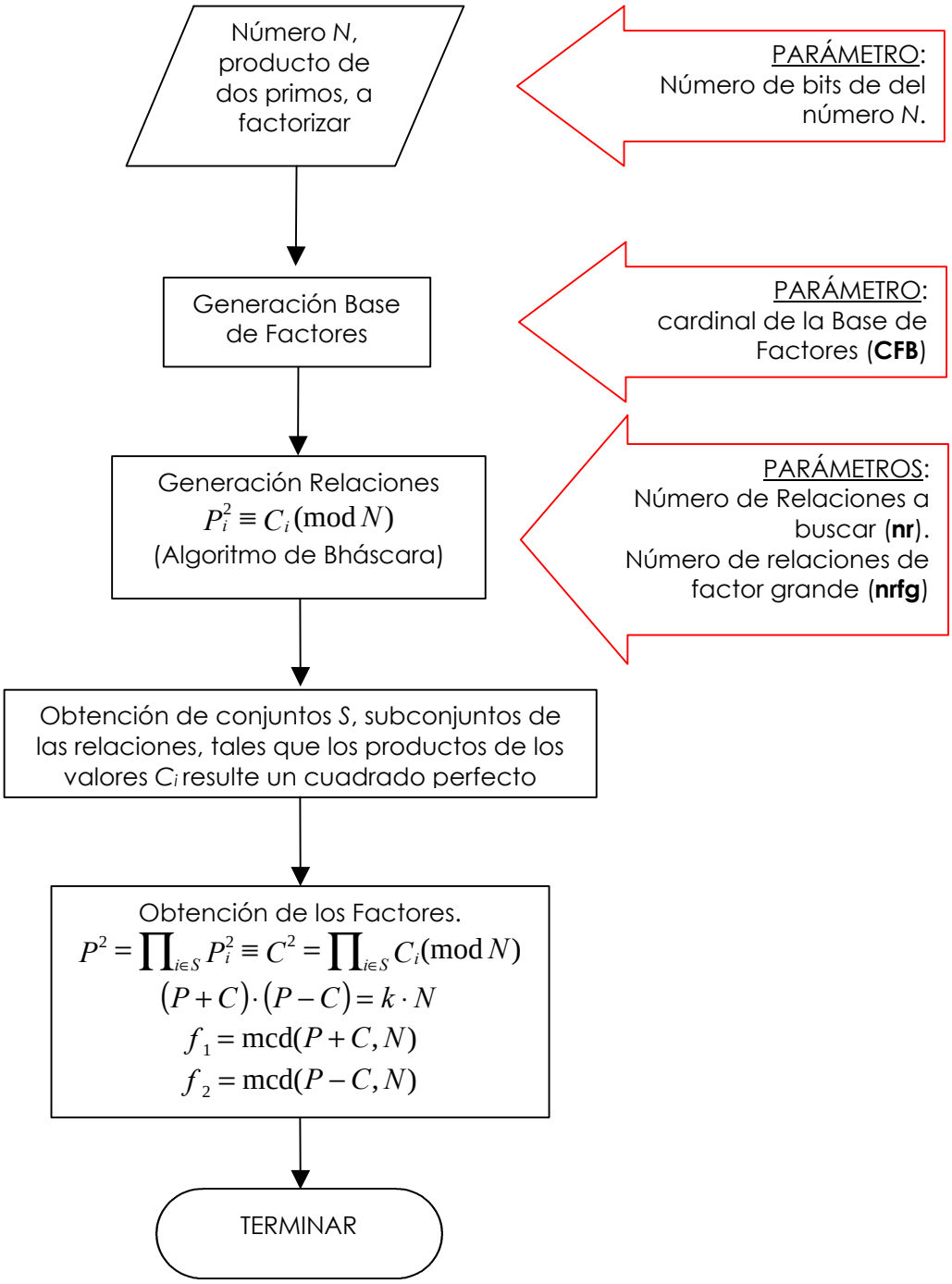
GENERACIÓN DE RELACIONES



SUAVIDAD


Ci & REL R & Base
j = 1;
*(Base+j)->F¡=0
SÍ
NO
k = Modulo(Ci,*(Base + j)->F);
NO
SÍ
j++;
k == 0
1 si el primo ha intervenido un número IMPAR de veces
Ci = Cociente(Ci,*(Base + j)->F);
*(R + j) = 1 - *(R + j);
NO
Ci == 1
SÍ
*(Base + j)->F será cero cuando se haya recorrido todo el vector de la Base de Factores
TERMINAR: Ci SUAVE!!
TERMINAR: Ci no SUAVE FG = Ci
Es decir, Ci ha quedado **COMPLETAMENTE FACTORIZADO**
Ci NO ES SUAVE. Queda determinar si el actor grande es suficientemente pequeño como para almacenar la relación.

ALGORITMO CFRAC: ESQUEMA GENERAL



a

ANEXO V: FUNCIONES Y MACROS

Presentamos en este anexo las tablas de tiempos y sus respectivas gráficas, en un primer estudio sobre los tiempos de factorización de nuestra aplicación; implementada mediante llamada a funciones, o con el uso de macros.

El resultado del proceso mediante funciones se muestra en la Tabla 1, que recoge las factorizaciones hasta el tamaño 132 bits. En la Tabla aparecen cuatro columnas: en la primera queda recogido el tamaño del número a factorizar. En la segunda queda recogido el tiempo más corto que se ha requerido para la factorización de uno de los 100 números del tamaño al que se refiere la fila correspondiente. En la tercera queda recogido la mediana de los 100 tiempos calculados para cada tamaño. En la cuarta queda recogido el tiempo máximo requerido en una factorización de entero del tamaño referido.

El proceso mediante macros queda recogido en la Tabla 2. Esta tabla recoge la misma información que la indicada en la Tabla 1.

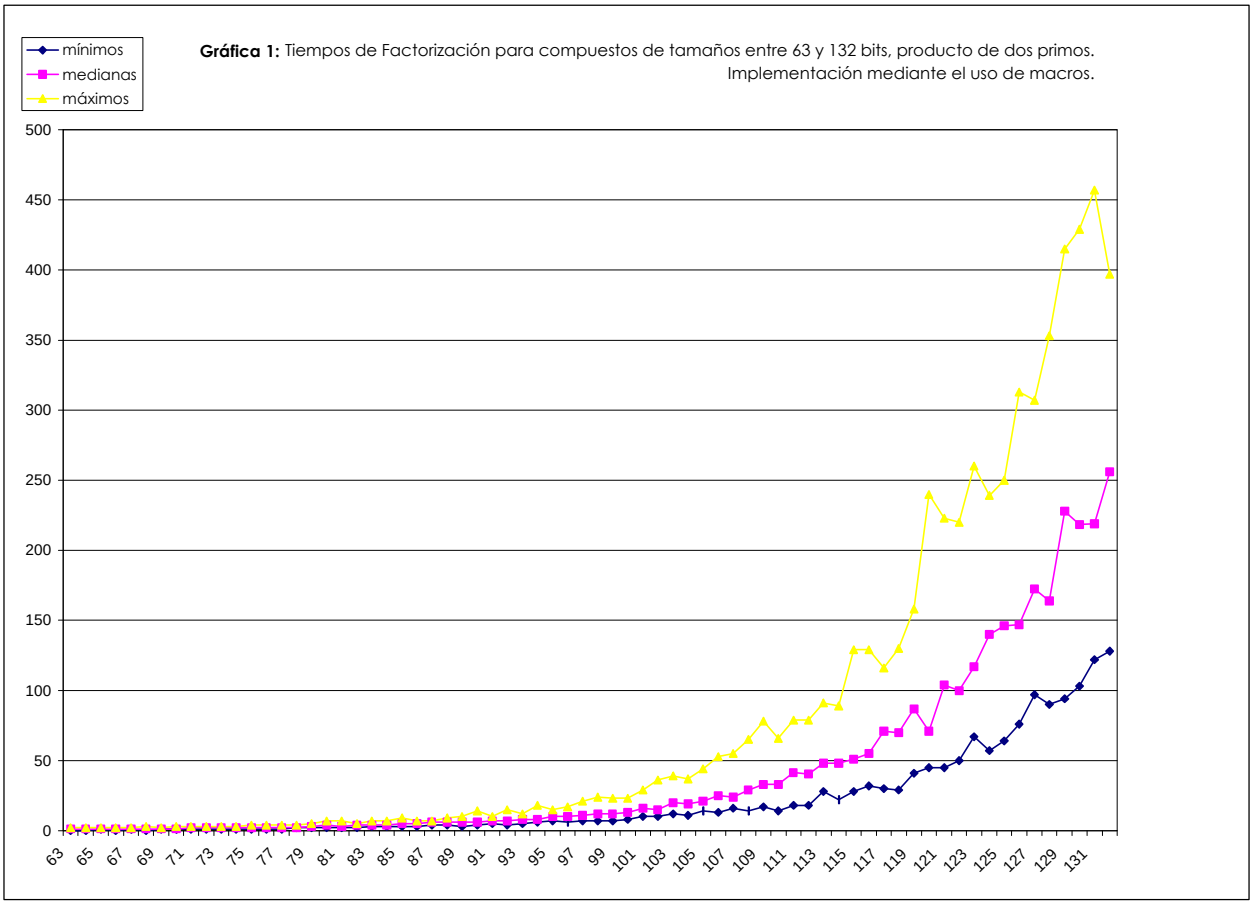
Las Gráficas 1 y 2 son una representación gráfica de los valores correspondientes de las Tablas.

bits	segundos min	med	max	bits	segundos min	med	max
63	0	1	2	98	8	14	28
64	0	1	2	99	9	15	28
65	1	1	2	100	8	16	28
66	0	1	3	101	12	19	35
67	1	1	2	102	11	18	42
68	0	2	3	103	14	23	46
69	1	2	3	104	12	22	41
70	1	2	3	105	16	26	51
71	1	2	4	106	15	30	61
72	1	2	4	107	19	28	66
73	1	2	4	108	16	34	78
74	1	2	4	109	19	39	93
75	2	3	4	110	16	38	77
76	1	3	5	111	21	48	96
77	2	3	5	112	20	48	92
78	2	3	5	113	34	56	110
79	2	4	6	114	25	55	103
80	2	4	9	115	33	61	150
81	3	4	10	116	37	64	151
82	2	5	6	117	36	84	137
83	3	5	7	118	34	81	148
84	3	5	8	119	50	103	185
85	3	6	10	120	52	80	269
86	3	6	8	121	60	122	264
87	5	7	9	122	56	118	258
88	4	7	10	123	80	139	304
89	4	7	12	124	66	164	274
90	5	7	16	125	76	171	297
91	6	8	12	126	88	168	359
92	5	9	18	127	112	204	361
93	6	10	14	128	105	188	410
94	7	9	22	129	110	270	481
95	8	12	18	130	119	252	487
96	8	11	21	131	142	262	532
97	9	13	25	132	148	292	446

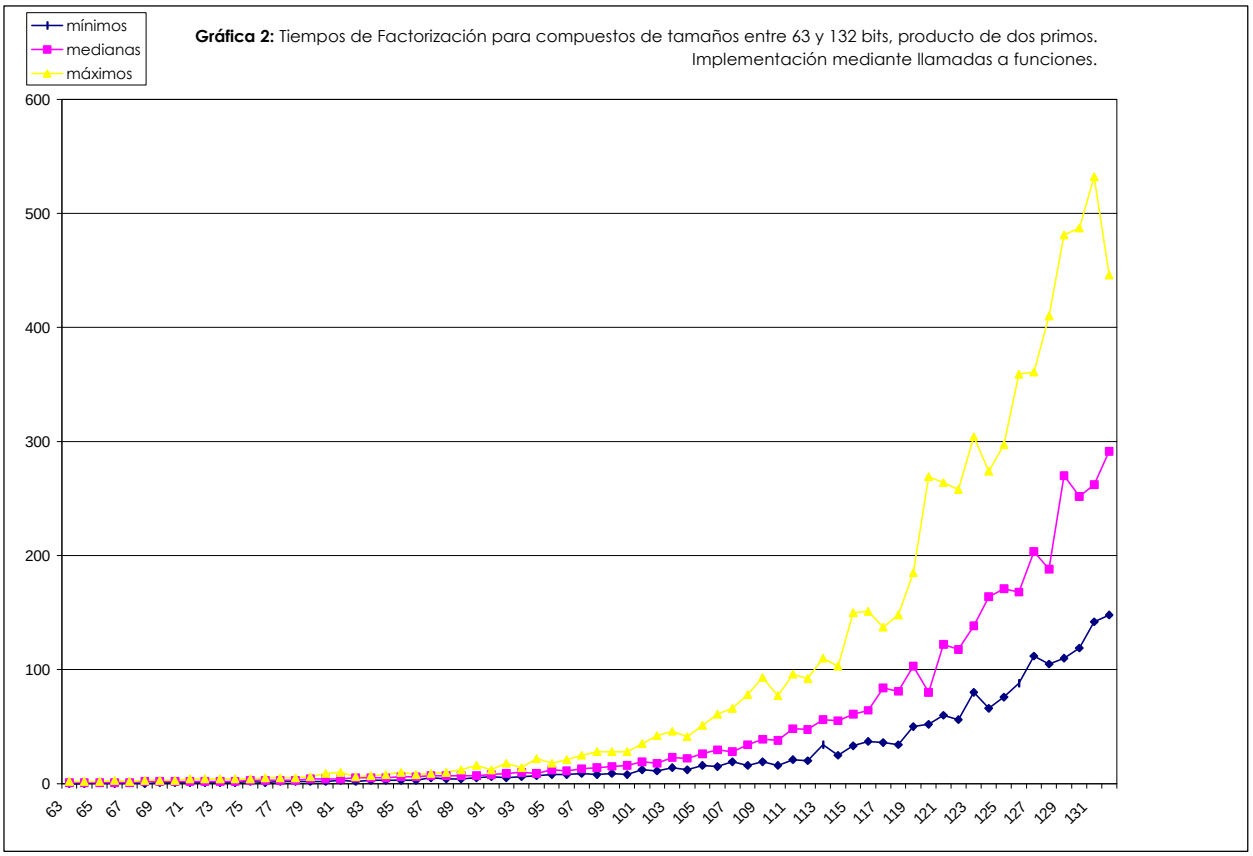
Tabla 1: Tiempos de factorización para enteros producto de dos primos. Implementación mediante llamadas a funciones.

bits	segundos min	med	max	bits	segundos min	med	max
63	0	1	2	98	7	12	24
64	0	1	2	99	7	12	23
65	1	1	2	100	8	13	23
66	0	1	2	101	10	16	29
67	1	1	2	102	10	15	36
68	0	1	3	103	12	20	39
69	1	1	2	104	11	19	37
70	1	1	3	105	14	21	44
71	1	2	3	106	13	25	53
72	1	2	3	107	16	24	55
73	1	2	3	108	14	29	65
74	1	2	3	109	17	33	78
75	1	2	4	110	14	33	66
76	1	2	4	111	18	42	79
77	1	2	4	112	18	41	79
78	2	2	4	113	28	48	91
79	2	3	5	114	22	48	89
80	2	4	7	115	28	51	129
81	2	3	7	116	32	55	129
82	2	4	5	117	30	71	116
83	3	4	7	118	29	70	130
84	3	4	7	119	41	87	158
85	3	5	9	120	45	71	240
86	3	5	7	121	45	104	223
87	4	6	7	122	50	100	220
88	4	6	9	123	67	117	260
89	3	6	10	124	57	140	239
90	4	6	14	125	64	146	250
91	5	7	10	126	76	147	313
92	4	7	15	127	97	173	307
93	5	8	12	128	90	164	353
94	6	8	18	129	94	228	415
95	7	10	15	130	103	219	429
96	6	10	17	131	122	219	457
97	7	11	21	132	128	256	397

Tabla 2: Tiempos de factorización para enteros producto de dos primos. Implementación mediante el uso de macros.



Gráfica 1: Tiempos de Factorización para compuestos de tamaños entre 63 y 132 bits, producto de dos primos. Implementación mediante el uso de macros.



Gráfica 2: Tiempos de Factorización para compuestos de tamaños entre 63 y 132 bits, producto de dos primos. Implementación mediante llamadas a funciones.

a ANEXO VI: PARÁMETROS DE LAS FUNCIONES ANTES Y DESPUÉS DE OPTIMIZAR

Mostramos en este anexo unas tablas que recogen los valores de los 10 registros estudiados mediante la herramienta RABBIT, para todas las funciones que intervienen a lo largo de toda la aplicación.

CUADRO COMPARATIVO DE LAS FUNCIONES DE auxiliar.c ANTES Y DESPUÉS DEL PROCESO DE OPTIMIZACIÓN

	0x24	0x43	0x45	0x79	0xc0	0xc2	0xc4	0xc5	0xc9	0xcA	llamadas
CrearNumero											
v. 1º	646.785	120.632.093	670.539	376.130.663	186.505.563	280.320.584	36.979.565	1.848.643	29.234.110	1.359.441	854.062
v. 2º	9.752	77.254.137	10.843	117.958.162	122.041.876	188.097.771	24.872.152	1.460.987	18.858.998	1.005.111	665.743
% mejora	98,49	35,96	98,38	68,64	34,56	32,90	32,74	20,97	35,49	26,06	22,05
v1 / v2	66,33	1,56	61,84	3,19	1,53	1,49	1,49	1,27	1,55	1,35	1,28
CopiarNumero											
v. 1º	16.365	61.451.958	22.453	92.935.841	89.776.402	122.387.001	14.127.583	1.716.796	7.365.693	785.583	1.664.200
v. 2º	13.033	34.114.815	14.326	53.230.166	48.999.654	66.644.017	7.421.036	824.532	4.114.155	417.782	861.466
% mejora	20,36	44,49	36,20	42,72	45,42	45,55	47,47	51,97	44,14	46,82	48,24
v1 / v2	1,26	1,80	1,57	1,75	1,83	1,84	1,90	2,08	1,79	1,88	1,93
PonerACero											
v. 1º	631.926	56.021.642	634.539	305.546.521	94.257.596	120.446.726	14.458.665	2.463.222	9.634.378	1.152.485	3.009.224
v. 2º	239	17.380.216	520	38.826.455	31.625.349	39.505.318	4.729.236	1.159.629	2.672.989	634.789	1.610.065
% mejora	99,96	68,98	99,92	87,29	66,45	67,20	67,29	52,92	72,26	44,92	46,50
v1 / v2	2.644,04	3,22	1.221,44	7,87	2,98	3,05	3,06	2,12	3,60	1,82	1,87
longitud											
v. 1º	396	580.446.554	43.321	742.835.227	944.379.180	1.136.321.898	204.781.464	2.212.065	113.688.783	614.033	1.345.085
v. 2º	127	19.151.747	73	53.082.999	51.293.466	58.112.961	13.530.678	1.913.618	7.038.904	1.123.207	824.665
% mejora	68,02	96,70	99,83	92,85	94,57	94,89	93,39	13,49	93,81	-82,92	38,69
v1 / v2	3,13	30,31	597,53	13,99	18,41	19,55	15,13	1,16	16,15	0,55	1,63
inv_v											
v. 1º	5	47.162	1	50.586	61.308	92.051	4.716	0	2.358	0	2.358
v. 2º	4	49.580	1	54.992	61.386	92.216	4.722	0	2.361	0	2.361
% mejora	20,00	-5,13	0,00	-8,71	-0,13	-0,18	-0,13		-0,13		-0,13
v1 / v2	1,25	0,95	1,00	0,92	1,00	1,00	1,00		1,00		1,00
orden											
v. 1º	177.327	257.110.919	64.818.553	791.340.682	684.684.377	688.815.956	157.525.170	7.619.376	105.323.984	3.183.967	48.909.278
v. 2º	62	4.381.968	168	14.232.810	9.944.802	9.973.006	2.420.630	643.658	1.333.841	268.386	795.989
% mejora	99,97	98,30	100,00	98,20	98,55	98,55	98,46	91,55	98,73	91,57	98,37
v1 / v2	2.883,37	58,67	386.976,44	55,60	68,85	69,07	65,08	11,84	78,96	11,86	61,44
DESPL_izda											
v. 1º	316	367.787.359	355	483.980.523	780.986.954	1.041.937.076	94.793.925	1.005.842	31.771.845	664.128	12.921.596
v. 2º	182	83.166.419	235	100.905.812	143.739.857	197.086.466	16.058.498	1.558.908	9.964.419	532.573	1.164.315
% mejora	42,41	77,39	33,80	79,15	81,60	81,08	83,06	-54,99	68,64	19,81	90,99
v1 / v2	1,74	4,42	1,51	4,80	5,43	5,29	5,90	0,65	3,19	1,25	11,10
DESPL_dcha											
v. 1º	17	146.655	8	160.574	258.838	315.345	25.076	88	9.326	83	3.158
v. 2º	22	147.256	4	162.878	259.171	316.241	25.108	105	9.338	82	3.162
% mejora	-26,47	-0,41	50,00	-1,43	-0,13	-0,28	-0,13	-18,75	-0,13	1,20	-0,13
v1 / v2	0,79	1,00	2,00	0,99	1,00	1,00	1,00	0,84	1,00	1,01	1,00

CUADRO DE LAS FUNCIONES DE suma.c, resta.c, productos.c, cocientes.c ANTES Y DESPUÉS DEL PROCESO DE OPTIMIZACIÓN

	0x24	0x43	0x45	0x79	0xc0	0xc2	0xc4	0xc5	0xc9	0xcA	llamadas
SUMA											
v. 1º	136	122.332.909	156	146.289.412	140.773.438	202.509.559	21.657.102	2.578.525	17.010.415	955.591	600.412
v. 2º	191	46.857.576	289	83.898.530	84.208.761	100.990.732	14.055.734	2.577.797	8.664.533	1.310.339	600.084
% mejora	-40,96	61,70	-85,26	42,65	40,18	50,13	35,10	0,03	49,06	-37,12	0,05
v1 / v2	0,71	2,61	0,54	1,74	1,67	2,01	1,54	1,00	1,96	0,73	1,00
RESTA											
v. 1º	80	114.951.653	79	134.843.626	120.465.166	185.743.631	22.229.389	1.949.248	17.623.098	756.389	598.654
v. 2º	96	75.120.121	76	106.305.277	104.360.244	147.984.494	21.746.686	2.245.314	13.943.831	1.247.164	583.426
% mejora	-19,38	34,65	3,80	21,16	13,37	20,33	2,17	-15,19	20,88	-64,88	2,54
v1 / v2	0,84	1,53	1,04	1,27	1,15	1,26	1,02	0,87	1,26	0,61	1,03
PROD_bit											
v. 1º	1.130	252.850.077	3.625	335.500.702	350.241.816	531.197.995	68.326.559	4.189.271	54.561.815	1.939.306	249.181
v. 2º	2.392	177.209.197	2.354	285.869.572	295.976.631	433.466.588	55.739.201	4.188.083	42.185.034	2.743.099	249.043
% mejora	-111,68	29,92	35,05	14,79	15,49	18,40	18,42	0,03	22,68	-41,45	0,06
v1 / v2	0,47	1,43	1,54	1,17	1,18	1,23	1,23	1,00	1,29	0,71	1,00
prod_bit											
v. 1º	31	1.560	2	8.552	1.891	2.851	349	45	244	20	1
v. 2º	0	0	0	0	0	0	0	0	0	0	0
% mejora	-	-	-	-	-	-	-	-	-	-	-
v1 / v2	-	-	-	-	-	-	-	-	-	-	-
COCIENTE											
v. 1º	853	225.383.732	10.174	308.031.482	418.521.025	549.270.254	53.996.484	4.387.287	25.002.623	2.092.848	83.123
v. 2º	744	103.139.161	4.448	141.457.250	159.688.453	228.607.367	25.187.966	2.389.992	16.835.966	1.161.097	83.077
% mejora	12,78	54,24	56,29	54,08	61,84	58,38	53,35	45,52	32,66	44,52	0,06
v1 / v2	1,15	2,19	2,29	2,18	2,62	2,40	2,14	1,84	1,49	1,80	1,00
Cociente											
v. 1º	398	460.809.056	7.874	618.329.800	820.917.608	940.273.186	178.174.046	450.066	90.704.591	261.199	145.303
v. 2º	0	0	0	0	0	0	0	0	0	0	0
% mejora	-	-	-	-	-	-	-	-	-	-	-
v1 / v2	-	-	-	-	-	-	-	-	-	-	-
MODULO											
v. 1º	601	449.243.006	620	570.501.906	855.467.743	1.068.793.261	93.671.584	6.412.311	45.521.050	2.991.399	177.931
v. 2º	779	115.261.674	1.135	158.835.644	167.774.038	239.252.893	26.257.876	2.874.649	18.338.283	1.580.145	95.140
% mejora	-29,62	74,34	-82,98	72,16	80,39	77,61	71,97	55,17	59,71	47,18	46,53
v1 / v2	0,77	3,90	0,55	3,59	5,10	4,47	3,57	2,23	2,48	1,89	1,87
Modulo											
v. 1º	1.836	1.584.565.651	15.441	2.395.473.808	2.138.810.861	3.311.738.067	259.263.656	251.589	148.064.243	198.929	14.641.005
v. 2º	0	0	0	0	0	0	0	0	0	0	0
% mejora	-	-	-	-	-	-	-	-	-	-	-
v1 / v2	-	-	-	-	-	-	-	-	-	-	-

CUADRO COMPARATIVO DE LAS FUNCIONES DE matematicas.c Y simbolo.c ANTES Y DESPUÉS DEL PROCESO DE OPTIMIZACIÓN

	0x24	0x43	0x45	0x79	0xc0	0xc2	0xc4	0xc5	0xc9	0xcA	llamadas
Euclides											
v. 1º	599	24.809.746	762	31.988.310	37.179.340	52.055.806	5.555.867	431.796	3.293.804	178.897	950
v. 2º	499	23.922.081	437	28.355.573	35.753.073	50.376.546	5.346.831	315.715	3.686.976	135.539	852
% mejora	16,69	3,58	42,68	11,36	3,84	3,23	3,76	26,88	-11,94	24,24	10,32
v1 / v2	1,20	1,04	1,74	1,13	1,04	1,03	1,04	1,37	0,89	1,32	1,12
Raiz_Cuadrada											
v. 1º	113	834.016	24	1.031.091	1.337.316	1.834.559	188.295	9.998	103.587	3.837	10
v. 2º	102	528.343	31	662.350	825.290	1.160.903	126.118	7.556	75.956	2.963	10
% mejora	9,78	36,65	-27,08	35,76	38,29	36,72	33,02	24,43	26,67	22,77	0,00
v1 / v2	1,11	1,58	0,79	1,56	1,62	1,58	1,49	1,32	1,36	1,29	1,00
QuitarDoses											
v. 1º	8	133.184	1	233.776	279.704	322.761	40.558	4.838	20.654	2.304	3.072
v. 2º	9	133.249	1	239.552	280.060	323.234	40.610	4.868	20.682	2.233	3.076
% mejora	-12,50	-0,05	0,00	-2,47	-0,13	-0,15	-0,13	-0,62	-0,14	3,10	-0,13
v1 / v2	0,89	1,00	1,00	0,98	1,00	1,00	1,00	0,99	1,00	1,03	1,00
Simbolo											
v. 1º	331	15.713.624	133	20.406.590	21.916.523	32.280.490	3.717.828	287.463	2.567.527	125.572	714
v. 2º	358	15.175.471	140	19.503.300	23.292.595	33.229.746	3.760.145	255.617	2.321.871	142.125	715
% mejora	-8,17	3,42	-5,66	4,43	-6,28	-2,94	-1,14	11,08	9,57	-13,18	-0,14
v1 / v2	0,92	1,04	0,95	1,05	0,94	0,97	0,99	1,12	1,11	0,88	1,00

CUADRO COMPARATIVO DE LAS FUNCIONES DE base.c Y relaciones.c ANTES Y DESPUÉS DEL PROCESO DE OPTIMIZACIÓN

	0x24	0x43	0x45	0x79	0xc0	0xc2	0xc4	0xc5	0xc9	0xcA	llamadas
Erasthotenes											
v. 1º	507.044	2.036.263	746.322	178.593.924	8.059.039	9.920.013	2.400.200	49.415	2.314.529	887	1
v. 2º	4	341.791	47.922	623.709	651.422	781.647	130.487	132	130.324	30	1
% mejora	100,00	83,21	93,58	99,65	91,92	92,12	94,56	99,73	94,37	96,62	0,00
v1 / v2	126.761,00	5,96	15,57	286,34	12,37	12,69	18,39	374,35	17,76	29,55	1,00
BaseFactores_Ser											
v. 1º	523.849	18.338.804	762.252	203.805.138	32.117.344	44.962.422	6.655.782	329.820	5.410.781	123.748	1
v. 2º	2.351	15.562.967	50.383	20.135.375	24.368.851	34.546.992	3.982.675	257.201	2.535.307	141.602	1
% mejora	99,55	15,14	93,39	90,12	24,13	23,16	40,16	22,02	53,14	-14,43	0,00
v1 / v2	222,87	1,18	15,13	10,12	1,32	1,30	1,67	1,28	2,13	0,87	1,00
InicializarRelaciones											
v. 1º	729.473	26.015.407	743.581	265.917.551	47.050.688	64.179.414	9.383.784	18.815	9.217.467	13.459	1
v. 2º	90.771	1.634.852	91.509	32.775.108	8.443.508	10.112.140	2.168.332	842	2.077.883	338	1
% mejora	87,56	93,72	87,69	87,67	82,05	84,24	76,89	95,53	77,46	97,49	0,00
v1 / v2	8,04	15,91	8,13	8,11	5,57	6,35	4,33	22,36	4,44	39,82	1,00
InicializarRel											
v. 1º	312	10.124	320	111.619	15.206	21.693	2.595	21	2.529	12	1
v. 2º	89	2.600	83	31.634	3.905	5.482	671	12	649	9	1
% mejora	71,47	74,32	74,18	71,66	74,32	74,73	74,14	41,46	74,34	29,17	0,00
v1 / v2	3,51	3,89	3,87	3,53	3,89	3,96	3,87	1,71	3,90	1,41	1,00
PonerACeroRel											
v. 1º	59	7.034.562	212	23.677.198	14.897.572	18.648.934	2.710.484	230.001	2.130.698	32.382	82.757
v. 2º	34	1.274.938	43	4.637.057	4.962.661	6.014.067	1.075.262	116.470	909.823	33.640	82.711
% mejora	43,22	81,88	79,91	80,42	66,69	67,75	60,33	49,36	57,30	-3,88	0,06
v1 / v2	1,76	5,52	4,98	5,11	3,00	3,10	2,52	1,97	2,34	0,96	1,00
OrdenarRelaciones											
v. 1º	13.224	1.022.903.491	68.974.379	1.396.555.518	1.798.001.843	2.110.910.704	249.566.931	11.176.486	121.303.419	5.556.186	4
v. 2º	7.472	7.192.976	28.894	10.009.804	8.558.197	15.901.550	1.238.337	93.611	933.860	47.010	4
% mejora	43,50	99,30	99,96	99,28	99,52	99,25	99,50	99,16	99,23	99,15	0,00
v1 / v2	1,77	142,21	2.387,15	139,52	210,09	132,75	201,53	119,39	129,89	118,19	1,00
OrdenFinal											
v. 1º	12.541	60.527.644	31.417	85.076.094	137.001.691	156.828.914	37.580.097	300.742	37.290.521	19.795	1
v. 2º	76	690.642	2.308	2.328.333	3.379.182	3.628.173	838.473	30.146	801.248	6.727	1
% mejora	99,40	98,86	92,65	97,26	97,53	97,69	97,77	89,98	97,85	66,02	0,00
v1 / v2	166,11	87,64	13,61	36,54	40,54	43,23	44,82	9,98	46,54	2,94	1,00
LiberarRS											
v. 1º	16.095	1.272.005	16.476	4.920.037	2.117.874	3.236.122	494.588	163	386.497	98	1
v. 2º	11.521	813.185	11.654	3.686.166	1.384.474	2.107.800	322.671	98	258.056	59	1
% mejora	28,42	36,07	29,27	25,08	34,63	34,87	34,76	39,88	33,23	40,00	0,00
v1 / v2	1,40	1,56	1,41	1,33	1,53	1,54	1,53	1,66	1,50	1,67	1,00
LiberarR											
v. 1º	11	365	5	2.671	520	913	116	8	96	6	1
v. 2º	9	189	2	2.543	267	553	58	4	45	3	1
% mejora	14,29	48,22	60,00	4,79	48,65	39,40	50,00	50,00	53,13	50,00	0,00
v1 / v2	1,17	1,93	2,50	1,05	1,95	1,65	2,00	2,00	2,13	2,00	1,00

CUADRO DE LAS FUNCIONES DE bhascara.c, suavidad.c Y factores.c ANTES Y DESPUÉS DEL PROCESO DE OPTIMIZACIÓN

	0x24	0x43	0x45	0x79	0xc0	0xc2	0xc4	0xc5	0xc9	0xcA	llamadas
InicializarBhascara											
v. 1º	154	494.252	19	620.648	808.193	1.093.499	109.090	5.309	59.335	1.984	1
v. 2º	191	329.471	22	425.043	506.109	709.233	75.050	4.362	45.583	1.313	1
% mejora	-24,10	33,34	-13,16	31,52	37,38	35,14	31,20	17,84	23,18	33,83	0,00
v1 / v2	0,81	1,50	0,88	1,46	1,60	1,54	1,45	1,22	1,30	1,51	1,00
RelacionBhascara											
v. 1º	2.502	1.019.717.456	15.442	1.336.780.752	1.713.631.293	2.302.761.820	234.189.507	17.416.300	137.972.617	7.725.222	82.757
v. 2º	2.447	350.737.097	6.349	546.142.123	553.218.646	804.268.009	99.365.390	9.734.681	70.494.530	5.277.911	82.711
% mejora	2,20	65,60	58,89	59,14	67,72	65,07	57,57	44,11	48,91	31,68	0,06
v1 / v2	1,02	2,91	2,43	2,45	3,10	2,86	2,36	1,79	1,96	1,46	1,00
LiberarBhascara											
v. 1º	15	1.241	9	4.772	1.742	2.866	366	14	292	7	1
v. 2º	10	650	6	2.781	990	1.556	207	6	170	3	1
% mejora	36,67	47,60	38,89	41,72	43,17	45,70	43,44	57,14	41,78	57,14	0,00
v1 / v2	1,58	1,91	1,64	1,72	1,76	1,84	1,77	2,33	1,72	2,33	1,00
Suave_s											
v. 1º	1.627	2.172.885.339	32.481	3.069.441.708	3.130.931.288	4.497.861.577	466.666.084	801.411	264.518.034	465.108	41.620
v. 2º	739	412.856.029	3.163	1.663.122.211	552.847.726	886.302.124	44.666.529	562.004	29.261.141	261.980	41.598
% mejora	54,58	81,00	90,26	45,82	82,34	80,30	90,43	29,87	88,94	43,67	0,05
v1 / v2	2,20	5,26	10,27	1,85	5,66	5,07	10,45	1,43	9,04	1,78	1,00
EliminarFactorGrande											
v. 1º	92.236	997.964	134.866	22.514.106	1.360.226	1.507.035	134.083	800	94.445	175	4
v. 2º	42.785	502.970	60.817	10.655.603	584.814	712.997	58.674	544	50.648	70	4
% mejora	53,61	49,60	54,91	52,67	57,01	52,69	56,24	31,96	46,37	60,17	0,00
v1 / v2	2,16	1,98	2,22	2,11	2,33	2,11	2,29	1,47	1,86	2,51	1,00
EliminacionGaussiana											
v. 1º	3.849.389	46.135.073	9.693.012	1.177.375.486	55.235.961	67.151.773	7.764.534	129.021	6.729.034	15.620	1
v. 2º	54.761	11.006.062	849.269	23.773.589	8.648.443	15.876.768	1.168.534	59.553	881.565	19.827	1
% mejora	98,58	76,14	91,24	97,98	84,34	76,36	84,95	53,84	86,90	-26,93	0,00
v1 / v2	70,29	4,19	11,41	49,52	6,39	4,23	6,64	2,17	7,63	0,79	1,00
ValidarRelaciones											
v. 1º	5.704	184.065	6.211	2.189.549	535.112	619.377	175.457	4.358	91.547	3.798	1
v. 2º	6	2.881	155	11.002	7.652	9.479	2.672	125	1.888	95	1
% mejora	99,89	98,44	97,50	99,50	98,57	98,47	98,48	97,14	97,94	97,50	0,00
v1 / v2	950,67	63,90	40,07	199,02	69,93	65,34	65,67	35,00	48,49	39,97	1,00
BuscarCuadrados											
v. 1º	2.951	58.866.608	6.505	77.333.536	100.080.680	130.964.951	13.132.477	943.855	8.858.925	336.304	9
v. 2º	1.663	95.194.707	3.300	109.862.643	147.395.692	204.324.144	20.397.885	1.108.325	14.512.661	443.637	9
% mejora	43,65	-61,71	49,28	-42,06	-47,28	-56,01	-55,32	-17,43	-63,82	-31,92	0,00
v1 / v2	1,77	0,62	1,97	0,70	0,68	0,64	0,64	0,85	0,61	0,76	1,00
BuscarFactores											
v. 1º	34	948.657	45	1.251.556	1.733.179	2.250.582	217.959	16.022	99.099	6.951	9
v. 2º	63	360.196	22	500.843	535.145	776.484	92.655	8.530	57.494	4.313	9
% mejora	-88,06	62,03	50,56	59,98	69,12	65,50	57,49	46,76	41,98	37,95	0,00
v1 / v2	0,53	2,63	2,02	2,50	3,24	2,90	2,35	1,88	1,72	1,61	1,00

RESUMEN MEJORAS FUNCIÓN `main()`

		versión 00	versión 01	versión 02
0x24	Fallos en L2	5.254.965	220.432	25.781
0x43	Referencias a memoria de datos	4.441.428.694	905.897.523	808.793.008
0x45	Fallos en L1	80.424.542	1.120.907	93.366
0x79	Ciclos de reloj	7.672.819.100	2.426.513.988	1.702.847.174
0xC0	Instrucciones ejecutadas	7.038.923.209	1.322.514.207	1.181.864.423
0xC2	Microinstrucciones ejecutadas	9.420.060.220	2.002.471.029	1.785.738.909
0xC4	Instrucciones de salto ejecutadas	1.029.056.327	177.488.624	186.338.505
0xC5	Instr. de salto equivocadas predichas	31.804.944	11.542.367	15.702.519
0xC9	Saltos tomados ejecutados	594.373.205	125.203.033	131.769.458
0xCA	Saltos tomados mal predichos ejecutados	15.007.514	5.796.703	8.108.609
	numero de llamadas	1	1	1

La versión .02 recoge el mismo codigo que la v. 01, pero se ejecuta con los parámetros `nrn` y `nrfg` ajustados a sus valores óptimos. Esos valores han quedado así: `nrn = 100`; `nrfg = 1200`.

a

ANEXO VII: ALGUNAS APLICACIONES DE CÁLCULO SIMBÓLICO

Recogemos en este anexo una breve descripción de diferentes aplicaciones de Cálculo Simbólico. Hemos realizado una larga búsqueda de estas aplicaciones. Aquí recogemos aquellas que hemos estudiado más extensamente.

A. VI.1. LiDIA.

Programa de libre distribución para el sistema operativo Linux. Está escrito en C++, para teoría de números. Provee una colección de implementaciones altamente optimizadas de varios tipos de datos multiprecisión. Desarrollado por el Grupo LiDIA de la Universidad Tecnológica de Darmstadt. Se puede encontrar en [Lidia]. Tiene implementados los siguientes algoritmos de factorización: Intento por divisiones sucesivas (TD), algoritmo Rho de POLLARD, algoritmo (p – 1) de POLLARD, algoritmo (p + 1) de WILLIAMS, algoritmo de las curvas elípticas (ECM), y el algoritmo de la Criba Cuadrática (MPQS).

A. VI.2. PARI.

Programa de libre distribución para plataformas UNIX. Escrito en C. Es un sistema de cálculo numérico capaz de manejar complejos tipos algebraicos y de teoría de números con completa recursividad. Su principal característica es la rapidez. Inicialmente desarrollado por Henri COHEN; actualmente PARI-GP tiene su soporte gracias a Karim BELADAS, de la Universidad de Paris–Sur Orsay con la ayuda de contribuciones voluntarias. Se puede encontrar en [Pari]. Tiene implementados los siguiente algoritmos de factorización: Método Rho de POLLARD, método de las Curvas Elípticas de LENSTRA y el algoritmo MPQS (este último adaptado del código de LiDIA).

A. VI.3. GAP (Groups, Algorithms and Programming).

Programa de libre distribución, que puede instalarse en sistemas UNIX, en Windows 9x y en sistemas Macintosh. Es un CAS (Computer Algebra System) para realizar cálculos con estructuras algebraicas como grupos, cuerpos finitos, espacios vectoriales, etc., desarrollado por Lehrstuhl D für Mathematik (LDFM) en Aachen (Alemania) y actualmente mantenido y ampliado por la Escuela de Matemáticas y Ciencias de la Computación de la Universidad St. Andrews (Escocia). Se puede encontrar información y bajar el programa en [GAP].

Los algoritmos que están implementados en esta aplicación son: Método (p – 1) de Pollard; método (p + 1) de WILLIAMS, Método de las Fracciones Continuas de BRILLHART y MORRISON (CFRAC); método de las curvas elípticas de LENSTRA (ECM); y método de la Criba Cuadrática de POMERANCE (MPQS).

Esta aplicación y sus bibliotecas han servido además de base para muchas otras aplicaciones desarrolladas a partir de GAP: GUAWA, MeatAxe, Sisyphos, CHEVIE, GRAPE.

A. VI.4. MuPAD (MultiProcessing Álgebra Data tool).

Sistema de propósito general, bastante difundido. Realiza cálculos numéricos y simbólicos. Quizá su principal defecto sea la lentitud en los cálculos. Dispone de versiones de libre distribución para Windows 9x y para Linux. Está desarrollado por el grupo de investigación MuPAD, de la Universidad de Paderbom. Se puede encontrar información y bajar la aplicación en la dirección [MuPAD].

Los algoritmos que utiliza para la factorización son el de las Fracciones Continuas (CFRAC) y el de las Curvas Elípticas (ECM).

A. VI.5. MAPLE.

No existe versión de libre distribución. Aplicación desarrollada por Waterloo MAPLE. Es un sistema de cálculo científico, es decir, simbólico, numérico y gráfico, que se viene desarrollando desde 1980 en la Universidad de Waterloo, en Canadá. El lenguaje de programación utilizado es el C.

Se puede encontrar diferente información en la dirección [Maple]. Y existe la posibilidad de bajarse una versión de prueba para 30 días.

Los algoritmos de factorización que hemos encontrado documentados y sobre los que trabaja esta aplicación son el método (p - 1) de POLLARD y el método de las Curvas Elípticas (ECM).

No hemos podido hacer comparaciones con nuestra implementación de forma directa. Hemos podido realizar factorizaciones de enteros largos, productos de dos primos grandes. El tiempo de respuesta de esta aplicación es mayor que el que invierte nuestra implementación optimizada. De todas formas no podemos sacar conclusiones, puesto que las medidas se han hecho sobre máquinas diferentes y los datos comparativos los hemos realizado extrapolando las medidas según la frecuencia de reloj de una y otra máquina.

A. VI.6. MAGMA.

Aplicación comercial que puede ser instalada bajo Windows o en sistemas Linux. Se puede encontrar información sobre esta aplicación en la dirección [Magma]. Es un CAS diseñado para solucionar problemas computacionalmente complicados de álgebra, teoría de números, geometría y combinatoria. Su kernel contiene clases de las cinco ramas principales del álgebra: Teoría de Grupos, Teoría de Anillos, Teoría de Cuerpos, Teoría de Módulos y Teoría de Álgebras.

Tiene implementados muchos algoritmos de factorización. Entre otros el de Curvas Elípticas (ECM), el de la Criba Cuadrática (MPQS) y el de la Criba de Campo Numérico (NFS), que es actualmente el algoritmo más rápido conocido.

A. VI.7. Mathemática.

Escrito por Stephen WOLFRAM, aparece en el mercado en 1988. Está implementado en C. Es una poderosa herramienta matemática que incluye gráficos, cálculo simbólico y cálculo numérico. Mathemática fue un punto y aparte en el desarrollo del cálculo simbólico, debido posiblemente a dos características singulares: sus amplias posibilidades gráficas y el uso pionero de Internet en su difusión.

Al menos hasta la versión 4.1. los algoritmos utilizados para factorizar son el método Rho de POLLARD, el método (p - 1) de POLLARD, el método (p + 1) de WILLIAMS y el método de las Fracciones Continuas (CFRAC).

A. VI.8. Derive.

Programa de cálculo matemático, propiedad de SOFT WAREHOUSE, INC (Honolulu, Hawai, USA). Escrito en LISP. Su propósito es la resolución de cálculos matemáticos de carácter general, lo que implica que puede hacer bastantes cosas aceptablemente, pero sin la potencia de otros programas matemáticos específicos. Es un programa especialmente interesante: además de por su enorme facilidad de manejo y simplicidad informática (ocupa poco espacio y funciona en cualquier PC compatible), goza de una gran versatilidad que posibilita la integración entre los aspectos numéricos y formales con los gráficos y geométricos.

Se puede encontrar información sobre esta aplicación en la dirección [Deri_eur]. También hemos encontrado abundante información sobre esta aplicación en [Deri_esp]. No hemos logrado encontrar qué algoritmos emplea para la factorización de enteros.

En [Hern01] se recogen hasta 99 paquetes matemáticos de cálculo simbólico (y cálculo numérico). Son muchos más los existentes en el mercado. Henri COHEN [Cohe93] dedica al final del libro un apéndice sobre los paquetes para Teoría de Números. No recogemos comentarios sobre otras muchas aplicaciones conocidas, pero de las que no hemos encontrado información sobre los algoritmos de factorización que tienen implementados o de las que no hemos encontrado referencia a una implementación bajo el sistema operativo Linux.

b

BIBLIOGRAFÍA

[Abal97] "Introducción a la encriptación en las comunicaciones". Francisco ABAL y Josu ARAMBERRI. Informática y Automática, Vol. 30–1. 1997

[AIS 20] "Functionality Classes and Evaluation Methodology for Deterministic Random Number Generators". Versión 1. (02.12.1999). Mandatory if a German IT security certificate is applied for; English Trastation).
http://www.bsi.bund.de/zertifiz/zert/interpr/ais20e.pdf

[AIS 31] "Functionality Classes and Evaluation Methodology for Physical Random Number Generators". Versión 1. (25.09.2001). Mandatory if a German IT security certificate is applied for; English Trastation).
http://www.bsi.bund.de/zertifiz/zert/interpr/ais31e.pdf

[Alva98] "Generación de Claves del Criptosistema de Clave Pública de Blum, Blum y Shub". G. ÁLVAREZ MARAÑÓN, F. MONTOYA VITINI, A. PEINADO DOMÍNGUEZ. V Reunión Española sobre Criptología, Torremolinos, pp. 55–65, sept. 1998.

[Anto97] "Programación estructurada en C". James L. ANTONAKOS y Kenneth C. MANSFIELD Jr. Prentice Hall, 1997.

[Barr96] "RSAEuro. Technical Reference". Nick BARRON. RSAEuro. Nov. 1996.

[Blum86] "A simple unpredictable Pseudo-Random Number Generator". L. BLUM, M. BLUM and M. SHUB. SIAM Journal of Computing, Vol. 15, Nº 2. pp 364–381. 1986.

[Bren98] "Anayisis of Iterated Modular Exponentiation: The orbits of $x^a \bmod N$ ". J.J. BRENNAB and Bruce GEIST. Designs, Codes and Cryptography, 13. pp 229–245. 1998.

[Bres00] "Computational Number Theory". David M. BRESSOUD and Stan WAGON. Key College Publishing. 2000.

[Bres89] "Factorization and Primality Testing". David M. BRESSOUD. Springer Verlag. 1989.

[Buch93] "An implementation of the general number field sieve". J. BUCHMANN, J. LOBO and J. ZAYER. LNCS 773. Advances in Cryptology - CRYPTO'93.1993.

[Caba00] "Criptología y Seguridad de la Información". Pino CABALLERO y Candelaria HERNÁNDEZ. RA–MA. 2000.

[Cava00] "Factorization of a 512–Bit RSA Modulus". Stefania Cavallar, Bruce DODSON, Arjen K. LENSTRA, Walter LIOEN, Peter L. MONTGOMERY, Brian MURPHY, Herman TE RIELE, Karen AARDAL, Jeff GILCHRIST, Gérard GUILLERM, Paul LEYLAND, Joël MARCHAND, François MORAIN, Alec MIFFERT, Chris and Craig PUTNAN, Paul ZIMMERMANN. LNCS 1807. Eurocrypt'00. 2000.

[Cava99] "Factorization of RSA-140 Using the Number Field Sieve". Stefania CAVALLAR, Bruce DODSON, Arjen LENSTRA, Paul LEYLAND, Walter LIOEN, Peter L. MONTGOMERY, Brian MURPHY, Herman te RIELE and Paul ZIMMERMANN. LNCS 1716. Advances in Criptology–ASIACRYPT'99. 1999.

[Clap97] "Optimizing a Fast Stream Copher for VLIW, SIMD, and Superescalar Processors". Craig S. K. CLAPP. LNCS 1267. Fast Software Encryption. 1997.

[Cohe93] "A Course in Computational Algebraic Number Theory". Henri COHEN. Springer Verlag. 1993.

[Cont99] "The Factorization of RSA – 140". Scott CONTINI. RSA Laboratories. Bulletin n. 10. Marth. 1999

[Coro98] "An Accurate Evaluation of Maurer's Universal Test". Jean-Sébastien CORON and David NACCACHE. LNCS 1556. Selected Areas in Cryptrography. SAC'98. 1998

[Coro99] "On the Security of Random Sources". Jean-Sébastien CORON. LNCS 1560. PKC'99. March 1999.
http://www.gemplus.com/smart/r_d/publi_crypto/pdf/Cor99rng.pdf.

[Cusi95] "Properties of the Pseudorandom Number Generator". Thomas W. CUSICK. IEEE Transactions on information theory, vol. 41, nº 4. pp 1155–1159. July 1995.

[Davi83] "Factorization Using the Quadratic Sieve Algorithm". James A. DAVIS and Diane B. HOLDRIDGE. UC32 SAND83–1346. Unlimited Distribution. Printed Dec. 1983.

[Deit99] "C++: cómo programar". H. M. DEITEL y P. J. Deitel. Prentice Hall, 1999.

[Denn93] "On the factorization of RSA–120". T.DENNY, B. DODSON, A. K. LENSTRA, M. S. MANASSE. LNCS 773. Advances in Cryptology–CRYPTO'93. 1993

[Diff76] "New Directions in Cryptography". Whitfield DIFFIE and Martin E. HELLMAN. IEEE. Transactions on Information Theory. Vol IT–22, nº 6. Nov. 1976.

[Ding97] "Blum-Blum-Shub generator". C. DING. Electronic Letters Online Nº: 19970440. Feb. 1997.

[Dunh00] "Euler. El maestro de los matemáticos". William DUNHAM. Ed. Nivola. 2000.

[Dura00] "Ataques a DES y módulos factorizados de RSA". Raúl DURÁN DÍAZ, Luis HERNÁNDEZ ENCINAS, Jaime MUÑOZ MASQUÉ. Ágora SIC. Divulgación. Vol 20 jun. 2000.

[East94] "Randomness Recommendations for Security". D. Eastlake, S. Crocker, J Schiller. Network Working Group. RFC 1750. Dec. 94.
http://rfc-1750.rfcindex.com/rfc-1750.htm.

[ETSI03] "Electronic Signatures and Infrastructures (ESI); Algorithms and Parameters for Secure Electronic Signatures". European Telecomunications Standars Institute 2003. ETSI SR 002 176 v1. 1. 1. (2003–03) Special Report.
http://webapp.etsi.org/action%5CPU/20030401/sr_002176v010101p.pdf

[FIP140] "Annex C: Approved Random Number Generators for FIPS PUB 140–2, Security Requirements for Cryptographic Modules". Jean CAMPBELL, Randall J. EASTER, Annabelle LEE and Ray SNOUFFER. NIST. FIPS PUB 140 - 2. Annex C, Draft. March 17, 2003.
http://csrc.nist.gov/publications/fips/fips140-2/fips1402annexc.pdf.
http://csrc.nist.gov/cryptval/140-2.htm.

[Fust00] "Técnicas Criptográficas de Protección de Datos". Amparo FUSTER y otros. RA–MA 2000.

[Gold97] "New Directions in Cryptography: Twenty Some Years Later (or Cryptography and Complexity: A Match Made in Heaven)". Shafi GOLDWASSER. Invited paper to the Proceedings of the 38th Annual IEEE Symposium on FOCS, Miami Beach, Florida. pp 314–324. Oct. 1997.
http://theory.lcs.mit.edu/~cis/pubs/shafi/1997-focs.pdf

[Gold99] "Fragments of a chapter on Encryption Schemes (Extracts from Foundations of Cryptography in preparation)" Chapter 3: Pseudorandom Generators. Oded GOLDREICH. Department of Computer Science and Applied Mathematics. Weizmann Institute of Science, Rehovot, Israel. Dec. 25, 1999

[Gome02] "Criptografía y curvas elípticas". José Luis GÓMEZ PARDO. La Gaceta de la real sociedad matemática española, vol. 5, n. 3. Septiembre–Diciembre, 2002.

[Gonz02] "VII Reunión Española sobre Criptología y Seguridad de la Información". Santos GONZÁLEZ, Consuelo MARTÍNEZ , editores. Tomos I y II. 2002.

[Gord84] "Strong Primes are Easy to Find". John GORDON. LNCS 209. Eurocrypt'84. 1984. http://dsns.csie.nctu.edu.tw/research/crypto/HTML/PDF/E84/216.PDF.

[Hern01] "Comparación y Evolución de los Sistemas de Cálculo Simbólico". Proyecto Fin de Carrera para optar al Título de Licenciado en Informática. Raúl HERNÁNDEZ VISIER. Universitat Politécnica de Valencia. Facultat d'Informàtica.

[Hern96] "Algoritmo de Cifrado con Clave Pública mediante una Función Cuadrática en el Grupo de los Enteros Módulo n". L. HERNÁNDEZ ENCINAS, J MUÑOZ MASQUÉ, F. MONTOYA VITINI, G. ÁLVAREZ MARAÑÓN, A. PEINADO DOMÍNGUEZ. Actas de la IV Reunión Española de Criptología (1996), pp. 101–108.

[Hern98] "Maximal Period of Orbits of the BBS Generator". L. HERNÁNDEZ ENCINAS, F. MONTOYA VITINI, J MUÑOZ MASQUÉ, A. PEINADO DOMÍNGUEZ. Proc. of CISC'98. Seoul (Korea) Dec. 1998. pp. 71–80.

[Hong96] "New Modular Multiplication Algorithms for Fast Modular Exponentiation". Seong–Min HONG, Sang–Yeop OH and Hyunsoo YOON. LNCS 1070. Eurocrypt'96. 1996.

[Isaa94] "Algebra. A Graduate Course". I. Martin ISAACS Brooks/Cole Publishing Company. 1994.

[Kali99] "Emerging Standards for Public–Key Cryptography". Burton S. KALISKI Jr. LNCS 1561. Lectures on Data Security. 1999.

[Kels99] "Yarrow–160: Notes on the Design and Analysis of the YarrowCryptographic Pseudorandom Number Generator". John KELSEY, Bruce SCHNEIER and Niels FERGUSON. LNCS 1758. Selected Areas in Cruptography. SAC'99. 1999.

[Kern91] "El lenguaje de programación C". Brian W. KERNIGHAN y Dennis M. RITCHIE. Prentice Hall. 1991.

[Khos98] "A Law of the Iterated Logarithm for Stable Processes in Random Scenery". Davar KHOSHNEVISAN and Thomas M. LEWIS. Stochastic Processes and their Applications, 74, pp. 89–121. 1998.

[Kill01] "A Proposal for: Functionality Classes and Evaluation Methodology for True (Physical) Random Number Generators. Versión 3.1. (25.09.2001)". Wolfgang KILLMANN and Werner SCHINDLER. mathematical - technical reference of [AIS 31]. (English translation). http://www.bsi.bund.de/zertifiz/zert/interpr/trngk31e.pdf

[Knut81] "The art of Computer Programing. Vol 2. Seminumerical Algorithms". D. E. KNUTH. Addison–Wesley, 1981.

[Knut87] "El arte de programar ordenadores. Vol 3. Clasificación y búsqueda". D. E. KNUTH. editorial Reverté, s.a. 1987.

[Kobl94] "A Course in Number Theory and Cryptography". Neal KOBLITZ. Springer Verlag. 1994.

[Kobl98] "Algebraic Aspects of Cryptography". Neal KOBLITZ. Springer Verlag. 1998.

[Kran86] "Primality and Cryptography". Evangelos KRANAKIS. Wiley–Teubner Series in Computer Science. 1986.

[Land99] "Primality Tests and use of Primes in Public Key Systems". Peter LANDROCK. LNCS 1561. Lectures on Data Security. 1999.

[Lens00] "Integer Factoring". Arjen K. LENSTRA. Designs, Codes and Cryptography, 19, pp. 101–128. 2000.

[Lens01] "Selecting Cryptographic Key Sizes". Arjen K. LENSTRA and Eric R. VERHEUL. Journal of Cryptology. 14: pp. 255–293. 2001.

[Lens87] "Computational Methods in Number Theory (part I y II)". edited by H. W. LENSTRA jr y R. TIJDEMAN. Mathematical Centre Tracts 154. 1987.

[Lens90-1] "Factoring with two large primes". Arjen K. LENSTRA, Mark S. MANASSE. LNCS 473. Eurocrypt'89. 1989.

[Lens90-2] "Primality Testing". Arjen K. LENSTRA. Proceedings of Symposia in Applied Mathematics. Vol 42. 1990

[Lens93] "The development of the number field sieve". A. K. LENSTRA and H. W. LENSTRA jr. Editors. Lecture Notes in Mathematics, 1554. Springer Verlag. 1993.

[Lens99] "Selecting Cryptographic Key Sizes in Commercial Applications". Arjen K. LENSTRA and Eric R. VERHEUL. CEE Quarterley Journal, 3: pp. 3–10. 1999.

[Lope90] "Introducción a la Teoría de Números Primos. Aspectos Algebraicos y Analíticos". Félix LÓPEZ FERNÁNDEZ – ASENJO y Juan TENA AYUSO. Instituto de Ciencias de la Educación. Universidad de Valladolid. 1990.

[Luce99] "Criptografía y Seguridad en Computadores". Manuel José LUCENA LÓPEZ. http//www.kriptopolis.com. (edición 1999).

[Mars96] "A battery of Tests of Randomness". George MARSAGLIA. http://stat.fsu.edu/~geo/diehard.html.

[Mass94] "Cryptography: Fundamentals and Applications (Copies of Transparencies)". James L. MASSEY. Apuntes de curso impartido en Advanced Technology Seminars (Zürich. Suiza). 1994.

[Math96] "Suggestions for Random Number Generation in Software". Tim MATHEWS. Bulletin n 1. News and advice from RSA laboratories. 22, jan, 1996.

[Maty99] "An Effective Defense Against First Party Attacks in Public-Key Algorithms". Stephen M. MATYAS Jr. and Allen ROGINSKY. ACSAC'99. Dec. 1999. http://www.acsac.org/1999/papers/thu-a-0830-roginsky.pdf.

[Maur92] "A Universal Statistical Test for Random Bits Generators". Ueli M. MAURER. Journal of Cryptography, vol 5, nº 2. 1992. ftp://ftp.inf.ethz.ch/pub/crypto/publications/Maurer92a.pdf.

[Mene97] "Hand Book of applied cryptography". A. MENEZES, P. van OORSCHOT, and S. VANSTONE. CRC Press. 1997.

[Mora94] "Seguridad y Protección de la Información". J. L. MORANT, A. RIBAGORDA, J. SANCHO. Centro de Estudios Ramón Areces S. A. 1994.

[Morr75] "A Method of Factoring and the Factorization of F7". Michael A. MORRISON and John BRILLHART. Math. of Comp, Vol 29. Nº 129. pp. 183–205. Jan 1975.

[Müll98] "The Security of Public Key Cryptosystems Based on Integer Factorization". Siguna MÜLLER and Winfied B. MÜLLER. LNCS 1438. Information Security and Privacy. ACISP'98. 1998.

[Naor97] "Number–Theoretic Constructions of Efficient Pseudo–Random Functions (Preliminary Verion)". Moni NAOR and Omer REINGOLD. Proc. 38th IEEE Symp. On Foundations of Comp. Sci. (FOCS'97) Miami Beach. IEEE. 1997.

[Ning01] "Efficienty Software Implementation for Finite Field Multiplication in Normal Basis". Peng NING and Yiquin Lisa YIN. ICICS 2001. LNCS 2229. pp. 177–188. 2001

[NIST00] "A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications". NIST Special Publication 800–22. 2000. http://csrc.nist.gov/rng/SP800-22b.pdf.

[Odly95] "The future of integer factorisation". Andrew M. ODLYZKO. CryptoBytes, 1(2). pp. 5–12. 1995.

[Ore48] "Number Theory and Its History". Oystein ORE. DOVER Publications, Inc. 1976 (1º ed. 1948).

[Patt00] "Estructura y Diseño de Computadores". David A. PATTERSON y John L. HENNESSY. Editorial Reverté, S.A. 2000.

[Pera98] "Técnicas de Diseño de Algoritmos". Francisco PERALES LÓPEZ y Miquel MASCARÓ PORTELLS. UIB. Materials didactics–54. 1998.

[Pera98-b] "Diseño y verificación de algoritmos". Francisco PERALES LÓPEZ. UIB. Materials didactics–56. 1998.

[Peyr01] "Alternative Method for Unique RSA Primes Generation". Mohammad PEYRAVIAN, Stephen M. MATYAS, Allen ROGINSKY and Nevenko ZUNIC. Computers $ Security, Vol 2, number 20. 1991.

[PGP98] "PGP for Personal Privacy (version 5.5)". Copyright (C) Free Software Foundation, Inc. 1998.
http://www.geocities.com/SiliconValley/Pines/2332/index.html

[Pome82] "Analysis and comparison of some integer factoring algorithms". Carl POMERANCE. Computational Methods in Number Theory. (H. Lenstra and R. Tijdeman eds.). pp. 89–141. 1982.

[Pome83] "Implementation of the continued fraction integer factoring algorithm". Carl POMERANCE, Samuel S. WAGSTAFF Jr. Congressus Numerantium, vol 37. pp. 99–118. 1983.

[Pome85] "The Quadratic Sieve Factoring Algorithm". Carl POMERANCE. Advances in Cryptography. Lecture Notes in Comput. Science, 209. pp. 169–182. 1985

[Pome87] "Analisys and Comparison of some integer factoring Algorithms". Carl POMERANCE. Computational Methods in number Theory. Part I, edited by H. W. Lenstra jr, and R. Tijdeman. pp. 89 – 139. 1987

[Pome90] "Factoring". Carl POMERANCE. Proceedings of Symposia in Applied Mathematics. Vol 42. 1990.

[Pome96] "A Tale of Two Sieves". Carl POMERANCE. Noticies of the AMS. Dec. 1996.

[Rabbit] http://www.scl.ameslab.gov/Projects/Rabbit/index.html.

[Ribe91] "The Little Book of Big Primes". Paulo RIBENBOIM. Springer Verlag. 1991.

[Ries87] "Prime Nubers and Computer Methods for Factorization". Hans RIESEL. Birkhäuser, (2d. Ed.). 1987.

[Rifa91] "Comunicación Digital". Josep RIFÀ i COMA y Llorenç HUGUET ROTGER. Masson, S.A. 1991.

[Rive78] "A Method for Obtaining Digital Signatures and Public – Key Cryptosystems". R. L. RIVEST, A. SHAMIR and L. ADLEMAN. Comunication of the ACM. V 21, n. 2. pp. 120–130. Feb. 1978.

[Rive99] "Are 'Strong' Primes Needed for RSA?" . Ronald L. Rivest and Robert D. SILVERMAN. 1999. http://theory.lcs.mit.edu/~rivest/RivestSilverman-AreStrongPrimesNeededForRSA.pdf

[Rose93] "Elementary Number Theory and its applications". Kenneth H. ROSEN. Addison–Wesley. 1993.

[Saou95] "A new method for the generation of strong prime numbers". Yannick SAOUTER. Institut de Recherche en Informatique et systémes aléaroires. Publication interne n. 931. 1985.

[Schi02] "Evaluation Criteria for True (Physical) Random Number Generators Used un Cryptographic Applications". Werner SCHINDLER and Wolfgang KILLMANN. LNCS 2523. CHES 2002. 2002.

[Schi94] "C. Guía de Autoenseñanza". Helbert SCHILDT. Osborne McGraw–Hill. 1994.

[Schi99] "Functionality Classes and Evaluation Methodology for Deterministic Random Number Generators. Version 3.1. (02.12.1999)". Werner SCHINDLER. mathematical–technical reference of [AIS 20] (English translation). Dec. 1999.

[Schi99] Evaluation Methodology for Random Number Generators Werner SCHINDLER AND Wolfgang KILLMANN. http://www.bsi.bund.de/zertifiz/zert/interpr/trngk20e.pdf.

[Schn96] "Applied Cryptography. Protocols, Algorithms and Source Code in C". Bruce SCHNEIER. John Wiley & Sons, Inc. 2d. Ed. 1996.

[Schn97] "Fast Software Encryption: Designing Encryption Algorithms for Optimal Software Speed on the Intel Pentium Processor". Bruce SCHNEIER and Doug WHITING. LNCS 1267. Fast Software Encryption. 1997.

[Schw02] "Principles of Network Security". Christian SCHWAIGER. 2002. http://www.ict.tuwien.ac.at/skripten/Penzhorn/N/N14_PRNG.pdf

[Sebe89] "Cryptography: An Introduction to Computer Security". Jennifer SEBERRY and Josef PIEPRZYK. Prentice Hall. 1989.

[Silv00] "A Cost–Based Security Analysis of Symmetric and Asymmetric Key Lengths". Robert D. SILVERMAN. RSA Laboratories. Bulletin n. 13. Apr. 2000.

[Silv87] "The Multiple Polynomial Quadratic Sieve". Robert D. SILVERMAN. Mathematics of Computation. Vol 48, n. 177, pp. 329–339. Jan. 1987.

[Silv97] "Fast Generation of Random, Strong RSA Primes". Robert D. SILVERMAN. CryptoBytes, Vol 3, nº 1. RSA Laboratories. Spring 1997.

[Stal00] "Organización y Arquitectura de Computadores". William STALLINGS. Prentice Hall. 2000.

[Stal98] "Cryptography and Network Security. Principles and Practice". William STALLINGS. Prentice Hall. 2nd ed. 1998.

[Stin00] "Cryptography: Theory and Practice". Douglas R. STINSON. CRC Press, Inc. 2000.

[Tane00] "Organización de Computadoras. Un enfoque estructurado". Andrew S. TANENBAUM. Prentice Hall. 2000.

[Vazi85] "Efficient and Secure Pseudorandom number generator". U. V. VAZIRANI and V. V. VAZIRANI. Wiley–Teubner Series in Computer Science. 1986.

[Vtune] http://www.intel.com/software/products/vtune.

[Wang99] "Linear Complexity versus Pseudorandomness: On Beth and Dai's Result". Yongge WANG. LNCS 1716. Advances in Criptology–ASIACRYPT'99. 1999.

[Wund79] "A Running Time Analysis of Brillart's Continued Fraction Factoring Method". Marvin C. WUNDERLICH. Lecture Notes in Mathematics, 751. Springer Verlag. 1979.

[Wund85] "Implementing the Continued Fraction Factoring Algorithm on Parallel Machines". Marvin C. WUNDERLICH. Mathematics of Computation. Vol 44, n. 169, pp. 251–260. Jan. 1985.