La Columna de Matemática Computacional

Sección a cargo de

Tomás Recio

El objetivo de esta columna es presentar de manera sucinta, en cada uno de los números de La Gaceta*, alguna cuestión matemática en la que los cálculos, en un sentido muy amplio, tengan un papel destacado. Para cumplir este objetivo el editor de la columna (sin otros méritos que su interés y sin otros recursos que su mejor voluntad) quisiera contar con la colaboración de los lectores, a los que anima a remitirle (a la dirección que se indica al pie de página*[1]*) los trabajos y sugerencias que consideren oportunos.*

En este número ...

... presentamos la segunda parte de un artículo del prof. Gómez Pardo, que versa ahora sobre la actualidad y el interés de buscar primos de gran tamaño, como el recientemente descubierto (el 14 de noviembre pasado) de más de cuatro millones de dígitos decimales, al que se hace referencia detallada en el texto. Como en el artículo anterior, el prof. Gómez Pardo conjuga, con un lenguaje asequible y con una prosa que "engancha" fácilmente al lector, las anécdotas y los fundamentos, las conjeturas y los logros, la descripción de las herramientas matemáticas y de los aspectos informáticos involucrados, las aplicaciones a la criptografía ...

[1]Tomás Recio. Departamento de Matemáticas. Facultad de Ciencias. Universidad de Cantabria. 39071 Santander. recio@matesco.unican.es

Aspectos computacionales de los números primos (II)

por

José Luis Gómez Pardo

Después de haber abordado, en la primera parte de estas notas [21], la cuestión del 'tamaño' del conjunto de los números primos, esta segunda parte estará dedicada a tratar de responder a la siguiente pregunta allí planteada: *¿Por qué buscar primos grandes?* Existen muchas razones para su búsqueda y, en lo que sigue, pasaré revista a algunas de ellas entre las que cabe mencionar:

- Para *probar el hardware* y, en particular, las nuevas CPUs.
- Para aprender más sobre la distribución de los números primos.
- Por *"amor al arte".*
- Por razones místicas.
- Por dinero (?!).
- Porque los primos grandes *son muy útiles en criptografía.*

Sobre la primera de estas razones mencionaré que, en tiempos recientes, las computaciones con primos grandes se han utilizado, a menudo, para probar nuevos microprocesadores, dado que éstos suelen hacer un uso muy intenso de la CPU. En ocasiones, estas pruebas fueron involuntarias como ilustra el episodio siguiente, ocurrido en 1994. Como mucha gente sabe, en dicho año se descubrió un "bug" en la FPU (unidad aritmética) del procesador Pentium que obligó a su fabricante, Intel, a reemplazar más de un millón de chips por un valor de unos 475 millones de dólares, además del impacto de la publicidad negativa, cuyo coste fue, sin duda, aun mayor. Mucha menos gente sabe que el descubridor de este fallo fue un matemático llamado Thomas Nicely, ya mencionado en [21]. Lo que realmente pocos conocen es lo que Nicely estaba haciendo cuando tropezó con el problema, que era, precisamente, calcular la constante de Brun. Este episodio puede servir también de ejemplo de otra motivación para trabajar con primos grandes: la de aprender más sobre la distribución de los primos.

La más antigua de las motivaciones para la búsqueda de primos grandes es, sin duda, la que considera dicha búsqueda como un honorable pasatiempo que entronca con el cálculo de números grandes que fue inaugurado por Arquímedes en su tratado *El arenario*, en el que estimó que bastaría un número de granos de arena comprendido entre 10^{51} y 10^{63} para llenar una esfera cuyo radio fuese igual a la distancia entre la tierra y el sol. En la actualidad, este impulso se puede enmarcar en la búsqueda de records de todo tipo con la intención de figurar en el *Libro Guinness*. Como dice Paulo Ribenboim en la

introducción de su libro [38]: *"Francamente, si yo leyera en el* Whig-Standard *que una reyerta en uno de nuestros pubs comenzó con una acalorada discusión sobre cual es el par de primos gemelos más grande conocido, yo encontraría esto altamente civilizado"*. Dicho sea de paso, estos primos gemelos son, en diciembre de 2001, $1807318575 \cdot 2^{98305} \pm 1$, cada uno de los cuales tiene 29603 dígitos decimales (Underbakke, Carmody).

La búsqueda de primos "por amor al arte" (o por afán de batir records, o quizá por coleccionismo) da lugar a records de la forma "el mayor primo conocido de tal tipo". Me voy a limitar aquí al record absoluto, es decir, al mayor primo conocido. En los últimos años, casi siempre este primo ha sido un "primo de Mersenne". Estos primos fueron popularizados por el monje francés Marin Mersenne en el siglo XVII y se definen como los números primos de la forma $M_p = 2^p - 1$, donde p es primo (condición necesaria, aunque no suficiente, para que M_p sea primo; por ejemplo, $M_{11} = 2^{11} - 1 = 2047 = 23 \cdot 89$ no es primo). En 1644, Mersenne escribió en el prefacio de su tratado *Cogitata Physica-Mathematica* que los números M_p son primos para $p = 2, 3, 5, 7, 13, 17, 19, 31, 67, 127$ y 257 y compuestos para los restantes primos $p \leq 257$. Se equivocaba, pero no demasiado teniendo en cuenta que, como el mismo admitió, no pudo comprobar todos estos números debido a su tamaño. A su lista le sobran dos números, M_{67} y M_{257}, que son compuestos, y le faltan otros tres, M_{61}, M_{89} y M_{107}, que son primos; la comprobación de todos ellos no se completó hasta el año 1947.

En realidad, los números de Mersenne ya aparecieron en la antigüedad a través del concepto de *números perfectos*, que son los enteros positivos n que tienen la propiedad ser igual a la suma de sus divisores positivos propios (distintos de n). Estos números poseen una de las propiedades místicas que tanto atraían a los griegos: *"El todo es igual a la suma de sus partes alícuotas"* y por eso han gozado de especial consideración no sólo en la antigüedad sino a lo largo de toda la Edad Media. En la antigüedad se conocían solo cuatro números perfectos: 6, 28, 496 y 8128, y la proposición final del libro IX de los *Elementos* de Euclides está dedicada a ellos. Se prueba en ella que si $2^n - 1$ es primo, entonces $2^{n-1}(2^n - 1)$ es perfecto, de modo que la búsqueda de primos de la forma $2^n - 1$, durante la Edad Media, se podía enmarcar en el contexto mucho más amplio de la "búsqueda mística de la perfección". Casi dos mil años después, Euler estableció el recíproco, demostrando que todos los números perfectos pares son de la forma $2^{p-1}M_p$ donde M_p (y, en consecuencia, también p) es primo. Así todos los números perfectos pares se obtienen a partir de los primos de Mersenne y lo que no se conoce es si existe algún número perfecto impar, ni tampoco si existen infinitos números perfectos. Por el momento sólo se conocen 39 números perfectos pares que corresponden a los 39 primos de Mersenne conocidos.

La razón de que los primos más grandes conocidos hayan sido casi siempre primos de Mersenne estriba en que para los números de Mersenne M_p hay un método muy eficiente para determinar si son primos. Un algoritmo que, dado un número n determina si es o no primo se llama habitualmente un

'test de primalidad'. Para los números de Mersenne existe el llamado "test de Lucas-Lehmer", que es varios órdenes de magnitud más rápido que los tests de primalidad que sirven para cualquier entero positivo.

Para describir el test de Lucas-Lehmer se define, por recurrencia, una sucesión de enteros positivos en la forma siguiente: $S(1) := 4$, $S(n+1) = S(n)^2 - 2$. Se tiene entonces, para $p > 2$:

Test de Lucas-Lehmer. *M_p es primo si y sólo si M_p divide a $S(p-1)$.*

El test de Lucas-Lehmer es muy fácil de programar y computacionalmente muy eficiente debido al hecho de que los ordenadores usan el sistema de numeración binario. En el cálculo de los términos de la sucesión se trabaja módulo $M_p = 2^p - 1$ para ahorrar tiempo y las divisiones por $2^p - 1$ en binario se reducen a dividir por 2^p (lo que es muy rápido al ser este número una potencia de 2) y a hacer sumas. En *Mathematica*, la siguiente función determina si M_p es o no primo:

```
LucasLehmer[2] = True;
LucasLehmer[p_Integer] := PrimeQ[p] &&
Nest[Mod[#^2 - 2, 2^p - 1]&, 4, p-2] == 0/; p > 1
```

Este test nos permite determinar fácilmente cuales de los M_p que consideró Mersenne, con $p \leq 257$, son primos e ir incluso bastante más allá. Usando la función `Eratostenes`, definida en [21] podemos hacer, por ejemplo:

```
Select[Eratostenes[4423], LucasLehmer[#]&]

{2, 3, 5, 7, 13, 17, 19, 31, 61, 89, 107, 127
521, 607, 1279, 2203, 2281, 3217, 4253, 4423}
```

lo que nos da los primos p correspondientes a los 20 primeros primos de Mersenne M_p, que eran todos los conocidos hasta el año 1963. Con *Mathematica* no se puede pretender descubrir primos record, pero si que se puede comprobar fácilmente si números de Mersenne de varios miles de dígitos decimales son o no primos. Por ejemplo, es fácil obtener, `LucasLehmer[216091] = True`. El primo M_{216091} tiene 65050 dígitos y era el mayor primo conocido entre 1985 y 1992, habiendo sido descubierto por David Slowinski con un ordenador Cray. El hecho de que una computación record establecida por un superordenador Cray en 1985 pueda ser ahora reproducida, utilizando un lenguaje de alto nivel y en menos tiempo, por un ordenador pequeño, es una corroboración empírica de la llamada *Ley de Moore* según la cual la potencia de los microprocesadores se dobla cada 18 meses (en realidad, lo que Moore predijo es que, durante los 10 años siguientes a 1965, la densidad de transistores en los circuitos integrados se duplicaría cada año y medio). Esto significa que el crecimiento de la potencia computacional sería exponencial, multiplicándose por un factor igual a $2^{t/18}$ cada t meses transcurridos. Está muy extendida la creencia de que la ley de Moore se ha cumplido con bastante exactitud durante los últimos años,

lo que de ser así supondría que la potencia computacional entre 1985 y 2000 se habría multiplicado, aproximadamente, por un factor de 1000 (teniendo en cuenta solamente los avances en la fabricación de chips y no las mejoras en los algoritmos y en el software).

Desde finales de los años 70 hasta mediados de los 90, los primos más grandes conocidos fueron primos de Mersenne, encontrados usualmente por Slowinski con ayuda de ordenadores Cray; paralelamente, estas búsquedas le sirvieron a Cray Research como banco de pruebas para su hardware. En 1996 se produjo un giro radical en este proceso, cuando George Woltman puso en marcha el proyecto llamado GIMPS *("Great Internet Mersenne Prime Search")* [19] cuyo objetivo es, como su nombre indica, la búsqueda de primos de Mersenne usando la potencia combinada de miles de ordenadores (usualmente PC's, para los que Woltman escribió una versión altamente optimizada del test de Lucas-Lehmer). Los resultados son enviados a un servidor central llamado "Primenet" [37], programado por Scott Kurowski, que controla y coordina todo el esfuerzo. A partir de 1996, se han descubierto cinco nuevos primos de Mersenne, todos ellos por GIMPS. El record actual se estableció el 14 de noviembre de 2001, cuando Michael Cameron, un estudiante canadiense, encontró, usando el programa de Woltman, el primo más grande que se conoce hasta la fecha.

Este número es $M_{13466917} = 2^{13466917} - 1$ y tiene exactamente 4053946 dígitos decimales, de modo que se trata de lo que a veces se llama un *megaprimo* (primo con más de un millón de dígitos decimales) de los que, por el momento, sólo se conocen dos. Una forma de expresar su gran tamaño es mediante su "longitud" que, si se escribe con fuente de 12 puntos, incluyendo puntos cada tres dígitos, es de más de veintidós kilómetros. Se puede hacer uno una idea de lo que esto significa viendo la expansión decimal del número, que se encuentra en [33], donde también figura su nombre en inglés, que es aun mucho más largo y si se imprimiese ocuparía muchos miles de páginas. Por eso, si se quiere darle un nombre permanente a este número, quizá fuese mejor seguir el camino de *Irineo Funes*, el personaje de Borges [8] insensatamente empeñado en construir un vocabulario infinito para los números naturales, a los que atribuía nombres tan sugestivos como *Máximo Pérez*, *El Ferrocarril*, *El Negro Timoteo* ...

La colaboración con GIMPS y Primenet está abierta a todo aquél que disponga de un ordenador conectado a Internet. Esto hace que más de 130000 personas de todo el mundo hayan colaborado con GIMPS y Primenet, y según datos de este último servidor, la máquina virtual formada por todos los ordenadores que colaboran tiene, en diciembre de 2001, un "sustained throughput" de 2415 gigaflops (1 gigaflop = mil millones de operaciones de coma flotante por segundo). Esto se traduce en 200,7 años de CPU de un Pentium 90 Mhz por día y, equivale a 43 ordenadores Cray del modelo más potente (T932), de modo que Primenet es, sin duda, uno de los "ordenadores" más potentes del mundo. Puede que en esto influya, además del "amor al arte", la existencia de premios en metálico para los descubridores de primos record. El descubridor del record anterior (el primo de Mersenne número 38, $M_{6972593}$), Nayan Hajratwala,

ganó 50000 dólares ofrecidos por la EFF (Electronic Frontier Foundation). En la actualidad, la EFF ofrece $100000 a la primera persona que encuentre un primo con más de 10 millones de dígitos decimales.

Analicemos a continuación la última y, probablemente, la más importante de las motivaciones para trabajar con primos grandes: *su utilidad en criptografía*. Esta historia se originó en 1978, cuando Rivest, Shamir y Adleman, propusieron un criptosistema de clave pública (o asimétrico), que se conoce como RSA por las iniciales de sus autores y, en el cual, los números primos juegan un papel decisivo. Dado que los detalles de RSA están bien explicados en la amplia bibliografía existente sobre el tema (véase, p. ej., [44], que también utiliza *Mathematica* como lenguaje para implementar los algoritmos criptográficos) me limitaré a indicar brevemente la razón de la importancia de los primos en este contexto. En RSA, la clave de cada usuario tiene dos partes. La primera de ellas es pública y consta de un "módulo", un número entero que es producto de dos primos grandes y otro número llamado "exponente de encriptación". La segunda es privada y consta del "exponente de desencriptación" que sólo es conocido por el usuario al cual pertenece. La seguridad de RSA se basa en la hipótesis (no demostrada, pero altamente plausible) de que no es computacionalmente factible el cálculo del exponente de desencriptación a partir de la clave pública. Para calcular su clave pública, cada usuario tiene que utilizar dos "primos grandes" –de, por ejemplo, 512 bits (dígitos binarios); más adelante indicaré la razón de mencionar precisamente este tamaño– y por tanto, es necesario que el problema de "reconocer primos" (dado un entero grande, determinar si es primo o, por el contrario, si es compuesto) sea "fácil". Por otra parte, el cálculo del exponente de desencriptación es también fácil si se conoce la factorización del módulo en producto de sus dos factores primos y, por tanto, la seguridad de RSA requiere que el problema de la factorización de enteros sea "difícil" en el sentido computacional del término, es decir, que no sea factible, por ejemplo, factorizar un entero de 1024 bits que sea producto de dos primos de 512 bits cada uno.

Para estudiar estas cuestiones es necesario precisar el significado de los términos "fácil", "difícil" y "no factible", utilizados anteriormente. Para ello se necesita un marco teórico que proporcione métodos para evaluar la dificultad de los problemas computacionales y dicho marco nos lo proporciona la teoría de la *Complejidad Computacional*. Esta teoría establece estimaciones para el *"tiempo de ejecución"* (o *"complejidad"*) de un algoritmo en función del "tamaño del input", tratando de que dichas estimaciones sean suficientemente precisas para que permitan, por ejemplo, calcular el tiempo requerido para ejecutarlo en una máquina concreta, con un tamaño de input determinado, suponiendo conocido el tiempo requerido por un input de tamaño menor.

El primer paso de todo este proceso es precisar que se entiende por *tamaño del input*. En el caso de los números enteros, la forma en que las máquinas (e incluso los humanos) los "ven" es como sucesiones de dígitos en un sistema de numeración que podemos suponer que es el sistema binario (no hay ningún

cambio esencial si se supone que se usa cualquier otra base en lugar de 2). Podemos definir entonces

$$\lg n = \begin{cases} 1 & \text{si } n = 0 \\ 1 + \lfloor \log_2 |n| \rfloor & \text{si } n \neq 0 \end{cases}$$

donde $\lfloor x \rfloor$ representa la "parte entera" de x, es decir, el mayor entero $\leq x$. Así, $\lg n$ es, simplemente, el número de bits en la representación binaria de n (sin contar el bit que habitualmente se dedica al signo). Obsérvese que, dado que para $n > 0$, $\lg n = 1 + \lfloor \frac{\ln n}{\ln 2} \rfloor$, usando la notación "$O$ mayúscula", la longitud de n puede ser también estimada asintóticamente como $O(\ln n)$.

La idea, ahora, es evaluar el tiempo de ejecución del algoritmo contando el "número de pasos" o "unidades básicas de computación" requeridas para ejecutar el algoritmo con un input determinado. El problema es determinar cuales deben ser estos "pasos" y esto puede depender del tipo de algoritmo con el que se trabaje. Cuando los algoritmos aceptan como input números enteros, la elección natural es tomar como paso lo que se llama una *operación bit*, que consiste en aplicar a una variable que toma los valores 0 o 1 una de las operaciones lógicas siguientes: conjunción ($\wedge$), disyunción ($\vee$) o negación ($\sim$), y que también se puede interpretar como una operación aritmética realizada con enteros de 1 bit. Esta elección es razonable, porque el tiempo requerido para ejecutar el algoritmo en una máquina dada será, aproximadamente, proporcional al número de operaciones bit requeridas. La constante de proporcionalidad depende de la máquina pero esto no es problema pues lo que se pretende es, simplemente, ver como crece el tiempo de ejecución al aumentar el tamaño del input. Por eso, la notación "O" es especialmente adecuada para las estimaciones de tiempo, ya que refleja bien su crecimiento asintótico.

Si se analizan las operaciones aritméticas básicas se ve que, por ejemplo, la suma de dos enteros $\leq n$ requiere $O(\lg n)$ operaciones bit y la multiplicación de dichos enteros por el método usual $O((\lg n)^2)$ (aunque, como mostraron Schönhage y Strassen en 1971, existe un algoritmo para multiplicar con tiempo $O(\lg n(\lg\lg n)(\lg\lg\lg n))$ [24] y todavía no se conoce cual es la "verdadera complejidad" de la multiplicación, es decir el algoritmo asintóticamente más rápido para realizarla). Los anteriores son ejemplos de algoritmos *de tiempo polinómico* porque su tiempo de ejecución se puede expresar en la forma $O(p(\lg n))$ donde p es un polinomio o, equivalentemente, en la forma $O((\lg n)^k)$ para un entero positivo k (con las modificaciones obvias, se pueden considerar algoritmos que admiten varias variables enteras como input, por ejemplo, en el caso de la multiplicación se obtiene que el tiempo de multiplicar dos enteros m y n es $O(\lg m \lg n)$). El significado de esta estimación es que, por ejemplo, si se duplica el tamaño de los números que se consideran –lo que corresponde a tomar números del orden del cuadrado de los números originales–, entonces el tiempo de ejecución se multiplica por 2^k (por 4 en el caso de la multiplicación). Este crecimiento es moderado y es ilustrativo compararlo con el requerido por el cálculo de $n!$ por el método usual de multiplicar todos los enteros positivos

$\leq n$ (para $n > 0$). El tiempo de ejecución de este algoritmo es $O(n^2(\lg n)^2)$ o, si se expresa solamente en términos de la longitud de n, $O((\lg n)^2 2^{2\lg n})$ [26]. El crecimiento asintótico de esta función es mucho más rápido pues, por ejemplo, si se duplica el tamaño de n, el tiempo de ejecución se multiplica por $2^{2(1+\lg n)}$. Los algoritmos que, como éste, tienen tiempo de ejecución igual a $O(2^{k\lg n})$, donde k es una constante, se dicen de *tiempo exponencial* pues la función que expresa su tiempo de ejecución es exponencial en el tamaño de n; dado que $2^{k\lg n} = e^{k\ln 2\lg n} = e^{c\lg n}$, con $c = k\ln 2$, se suele expresar esta función en la forma $O(e^{c\lg n})$, donde c es una constante. El tiempo de ejecución de estos algoritmos crece tan rápidamente que hace que el cálculo correspondiente no sea factible para valores grandes de n. Los algoritmos de tiempo polinómico y los de tiempo exponencial ocupan polos opuestos en el espectro de los problemas computacionales. Los problemas con algoritmos de tiempo polinómico se consideran "fáciles" y, al menos desde el punto de vista teórico son "factibles", mientras que aquellos con algoritmos exponenciales son "difíciles" y los cálculos correspondientes son "intratables" para n grande. En la práctica puede ocurrir que un algoritmo de tiempo exponencial sea más eficiente que uno de tiempo polinómico (correspondientes ambos a un cierto problema computacional) para un cierto rango de valores (normalmente pequeños) de n pero, asintóticamente, el polinómico siempre será más eficiente.

A partir de las estimaciones de tiempo de las operaciones aritméticas básicas se pueden ir obteniendo las de algoritmos cada vez más complicados. Entre los más importantes cabe mencionar el algoritmo de Euclides ("el abuelito de todos los algoritmos" según Knuth [24]). En su versión básica, el algoritmo de Euclides calcula el máximo común divisor de dos enteros $m, n > 0$ en tiempo $O(\lg m \lg n)$ y esta es también la complejidad del *algoritmo de Euclides extendido* que permite calcular enteros u y v tales que si $d = \mathrm{m}cd(m, n)$ es el máximo común divisor de m y n, entonces $d = um+vn$ Un algoritmo que juega un papel esencial en el proceso de encriptación y de desencriptación en RSA, y también en muchos otros algoritmos relacionados con los números primos, es la *exponenciación binaria* (cf. [5, 24]) que permite calcular $a^m \pmod n$, donde $n \geq 2$, $m \geq 0$ y $0 \leq a < n$. Consiste en ir calculando mediante sucesivas "elevaciones al cuadrado" los valores $a^{2^i} \pmod n$ para $i \leq \log_2 m$ y a continuación multiplicar un subconjunto de esas potencias $\pmod n$ para obtener $a^m \pmod n$. Su tiempo de ejecución es $O((\lg m)(\lg n)^2)$, o bien $O((\lg n)^3)$ si $m < n$ [5]. Esto significa que es un algoritmo de tiempo polinómico y, además, bastante rápido.

Como ya se ha indicado, la implementación de RSA requiere que cada usuario (o su software) construya su clave pública multiplicando dos primos grandes de, por ejemplo, 512 dígitos decimales. Los primos elegidos no pueden tener una forma especial (no pueden, por ejemplo, ser primos de Mersenne) ya que ello facilitaría enormemente la labor del criptoanalista (el que trata de romper el criptosistema). En consecuencia, deben de ser elegidos en la forma más aleatoria posible y, para ello, es necesario disponer de un *test de primalidad*

que se pueda aplicar a enteros arbitrarios (a diferencia de, por ejemplo, el test de Lucas-Lehmer que sólo sirve para enteros de una forma especial). La forma natural de proceder entonces sería tomar (seudo)aleatoriamente enteros del tamaño adecuado y someterlos al test de primalidad hasta encontrar los primos buscados. Esto plantea dos problemas:

- *¿Existen tests de primalidad eficientes para determinar rápidamente si un entero de, por ejemplo, 512 bits, es primo?*
- Suponiendo que la respuesta a la cuestión anterior es afirmativa *¿Cuántas veces –por término medio– habrá que aplicar el test hasta encontrar un primo?*

A menudo se da por supuesto que basta resolver afirmativamente el primer problema para poder encontrar primos del tamaño requerido por RSA, pero no se trata de algo evidente y aunque efectivamente es así, ello se debe a que el segundo problema también tiene una solución satisfactoria, pues el número esperado de aplicaciones del test va a ser bastante bajo. Es aquí donde se pone de manifiesto la gran importancia práctica de la pregunta *¿Cuántos primos hay?* que se estudiaba en la primera parte de estas notas [21]. Imaginemos, por un momento, que nuestra tarea fuese encontrar un cuadrado perfecto de 512 bits y que esto hubiésemos de hacerlo sin multiplicar o, para ser más precisos, eligiendo al azar enteros de este tamaño y luego comprobando si son cuadrados mediante el cálculo de su raíz cuadrada. Dado que la probabilidad de que un entero grande n tomado al azar resulte ser un cuadrado perfecto se puede estimar como $\frac{1}{2\sqrt{n}}$, vemos que, en el caso que nos ocupa, esta probabilidad sería del orden de $\frac{1}{10^{78}}$, es decir, comparable a la probabilidad de elegir un átomo predeterminado entre todos los del Universo (se ha estimado que su número no debe exceder de 10^{80}). Aunque la comprobación de cada número fuese muy rápida, está claro que así no sería factible encontrar un cuadrado de este tamaño. Por suerte, la densidad de primos es mucho más alta que la de cuadrados, y ello nos va a permitir ver que sí es factible encontrar un primo.

Usando la aproximación de $\pi(x)$ dada por $\mathrm{li}(x)$ (y la función `li` definida en [21]) se puede ver que el número de primos de 512 bits es, aproximadamente, `li[2.`^{512}`]-li[2`^{511}`] = 1.8906370642693397*10`^{151}. Por otra parte, hay exactamente `(2`^{512} `- 2`^{511}`)/2` $\approx$ `3.3519519824856493*10`^{153} números impares de 512 bits. Así, la proporción de primos entre los números impares de 512 bits es de uno por cada `(2.`^{512} `- 2`^{511}`)/(2 (li[2`^{512}`] - li[2`^{511}`])) = 177.292`, es decir, aproximadamente uno entre cada 177 números. Por tanto, vemos que el teorema del número primo resuelve satisfactoriamente el segundo problema, al mostrar que el número de tests esperado es bastante bajo, así que ahora pasaremos a buscar una respuesta al primero.

Dicha respuesta es afirmativa, pero tampoco se trata de algo completamente evidente. Por ejemplo, se podría tratar de usar la criba de Eratóstenes como test de primalidad: dado n construimos la lista de todos los primos $\leq n$ y vemos si n es uno de ellos. En teoría esto funciona pero en la práctica no

tiene utilidad alguna, salvo para números muy pequeños. Pensemos en nuestro problema de obtener, para usarlo en una clave de RSA, un primo de 512 bits. Dado que

```
N[li[2^512]] = 3.788709544954673*10^151
```

vemos que, por el teorema del número primo, el tamaño de la lista de números que habría que construir sería del orden de 10^{151}, lo que resulta imposible al ser este número mucho mayor que el número estimado de átomos del Universo. Algo parecido se puede decir del algoritmo consistente en ir probando a dividir n por todos los primos en orden creciente. Si alguno de ellos divide a n (en el sentido de la división entera), ya sabemos que es compuesto, pero si en esta sucesión de intentos de división alcanzamos a un primo mayor o igual que la raiz cuadrada de n sin que ninguno de los primos anteriores divida a n, llegamos a la conclusión de que n es primo. Este algoritmo es muy poco eficiente, pues su tiempo de ejecución es $O(n^{1/2+\varepsilon})$, donde $\varepsilon > 0$ es arbitrariamente pequeño. Este tiempo es exponencial en función del tamaño de n y así, aunque en teoría este método se puede utilizar para comprobar la primalidad de cualquier número (e incluso, para factorizarlo en caso de ser compuesto), en la práctica no es útil. Si tomamos un primo de 512 bits y queremos probar que efectivamente lo es usando éste método, tendríamos que hacer unas `N[li[Sqrt[2^512]]] = 6.56269130133754*10^74` divisiones, lo cual no es factible. De hecho, se estima en [16, p. 4] que demostrar de esta manera la primalidad de un número de 50 dígitos requeriría reproducir todo el esfuerzo computacional realizado por todas las máquinas en toda la historia (hasta el comienzo del siglo XXI). El problema es pues buscar un test de primalidad *eficiente*. Uno muy sencillo lo proporciona el *Teorema de Wilson* que (junto con su recíproco) dice que un entero positivo n es primo si y sólo si $(n-1)! \equiv -1 \pmod{n}$. El inconveniente de este test es que, como hemos indicado, el cálculo del factorial de n requiere tiempo exponencial en la longitud de n, lo que hace que tampoco tenga utilidad práctica. Podemos hacer un experimento con *Mathematica* para ver si la estimación de tiempo se ajusta a lo que ocurre en la "vida real". Para ello hacemos una tabla de los tiempos de ejecución para una serie de valores y, teniendo en cuenta que el tiempo de ejecución estimado es $O((\lg n)^2 2^{2\lg n})$, aproximamos los puntos de la lista obtenida usando el método de los mínimos cuadrados:

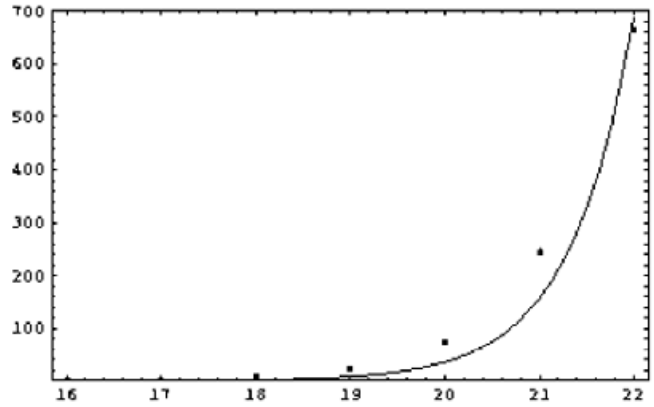
```
TiempoFactorial[n_] := Timing[(2^r)!][[1, 1]];
   ListaTiemposFactorial =
   Table[{r, TiempoFactorial[r]}, {r, 16, 22}];
      Fit[ListaTiemposFactorial, (r^2)*2^(2r), r]
```

resultando la función:

```
8.077390899285305*10^-14*2^(2 r)*r^2
```

que aproxima el tiempo de ejecución del algoritmo. Si comparamos su gráfica con los datos ya obtenidos para enteros de longitud r comprendida entre 16 y 22 obtenemos:

```
Plot[8.077390899285305*10^-14*2^(2 r)*r^2 , {r, 16, 22},
    PlotRange -> {0, 700}, Frame -> True, Epilog ->
    {PointSize[0.01], Map[Point, ListaTiemposFactorial]}];
```



Esto confirma que el tiempo requerido por *Mathematica* en este rango de valores crece de forma exponencial, aunque algo más lentamente que la función obtenida por aproximación. Si el crecimiento fuese realmente el dado por esta función, se tendría que, puesto que para $r = 512$, el valor de la función `8.077390899285305*10^-14*2^2r*r^2` es `3.806506560322153*10^300`, este sería el número de segundos requerido para comprobar la primalidad de un número de 512 bits en la misma máquina. Teniendo en cuenta que este tiempo es equivalente a más de 10^{293} años es fácil darse cuenta que, aunque el crecimiento real del tiempo sea algo más lento, es completamente imposible tratar por este método con números de este tamaño (para comparación, la estimación más actual sobre la edad del Universo, fija ésta en $1,2 \cdot 10^{10}$ años [30]).

En consecuencia, es necesario buscar un "test de primalidad" eficiente. Un posible candidato es el "teorema pequeño de Fermat", según el cual si p es primo y b es un entero arbitrario, entonces $b^p \equiv b \pmod{p}$. La comprobación de la congruencia de Fermat para un b y un p dados se puede hacer mediante el algoritmo de exponenciación binaria, que es muy eficiente y no plantea problema alguno aunque p tenga 512 bits (o incluso muchos más). Sin embargo, la congruencia de Fermat no se puede considerar como un test de primalidad, dado que el recíproco no se verifica, es decir, $b^n \equiv b \pmod{n}$ no implica que n es primo (se dice entonces que n es un seudoprimo para la base b) ni siquiera suponiendo que la fórmula se verifica para todos los enteros b. Por ejemplo, $n = 561 = 3 \cdot 11 \cdot 17$ satisface la congruencia de Fermat para todo b pero no es primo. Los números de este tipo se llaman *números de Carmichael*, quien

los estudió a principios del siglo XX y en 1992 Alford, Granville y Pomerance probaron que hay infinitos. A pesar de ello, lo que sí permite a menudo la congruencia de Fermat es demostrar con gran facilidad que un entero *no es primo*, pues si la congruencia $b^n \equiv b \pmod{n}$ no se cumple para alguna "base" b, entonces ya sabemos que n es compuesto.

El test de Fermat puede ser mejorado teniendo en cuenta que si n es un seudoprimo para la base b, donde $\text{mcd}(b, n) = 1$, se verifica que $b^{n-1} \equiv 1 \pmod{n}$ y así, si n es primo y se van extrayendo raíces cuadradas módulo n, la primera clase residual distinta de 1 que se obtiene tiene que ser necesariamente -1 puesto que $+1$ y -1 son las únicas raíces cuadradas de 1 módulo un primo. Se dice entonces que un entero impar $n > 3$, tal que $n - 1 = 2^s t$, con t impar, *pasa el test de Miller para la base* b (o que n pasa el test de seudoprimos fuertes para la base b), donde $1 \leq b < n$, cuando se verifica que o bien $b^t \equiv 1 \pmod{n}$ o bien existe un entero r tal que $0 \leq r < s$ y $b^{2^r t} \equiv -1 \pmod{n}$.

El test de Miller se puede programar en *Mathematica* de la forma siguiente (excluimos los casos triviales $b = 1$ y $b = n - 1$):

```
TestdeMiller[n_?OddQ,b_] :=
   Module[{s = IntegerExponent[n - 1, 2], x},
    x = PowerMod[b, (n - 1)/2^s, n];  x == 1 || x == n - 1 ||
     NestWhile[PowerMod[#, 2, n] &, x, # != n - 1 &, 1, s - 1] ==
     n - 1]/;1< b < n-1
SetAttributes[TestdeMiller, Listable]
```

Un número compuesto puede pasar el test de Miller: `TestdeMiller[2047,2] = True` significa que $2047 = 23 \cdot 89$ pasa el test de Miller para la base 2. Los números como este, que son compuestos pero pasan el test de Miller para una basa dada b, se llaman *seudoprimos fuertes para la base* b (o *b-seudoprimos fuertes*; obsérvese que un b-seudoprimo fuerte es también un b-seudoprimo, aunque el recíproco no se verifica). Pero si se consideran todas las posibles bases ya se obtiene una condición necesaria y suficiente para la primalidad de n: n es primo si y sólo si n pasa el test de Miller para todas las bases b con $1 < b < n$. Por tanto, no existe el análogo de los números de Carmichael para el test de Miller, es decir, no existen números que sean seudoprimos fuertes para todas las posibles bases. De hecho, se tiene un resultado más fuerte obtenido, independientemente, por L. Monier y M. Rabin:

• *Si $n > 3$ es un entero compuesto impar, entonces n es un seudoprimo fuerte para a lo sumo $1/4$ de las posibles bases b con $1 < b < n - 1$.*

Esto muestra que es suficiente someter n al test de Miller para más de $1/4$ de las posibles bases para obtener un test de primalidad. Sin embargo, cuando n es grande, esto no es práctico pues al ser el número de bases $O(n)$, este test sería de tiempo exponencial. No obstante, el resultado anterior sugiere el siguiente test:

Test de Miller-Rabin *Sea n un entero impar. Se eligen aleatoriamente k bases b tales que $1 < b < n-1$, y se somete n al test de Miller sucesivamente*

para cada una de estas bases. Si para alguna de ellas n no pasa el test de Miller el algoritmo se detiene y n es declarado compuesto. Si n pasa el test para las k bases, entonces n es declarado primo.

El test de Miller-Rabin para un entero n, con k bases elegidas aleatoriamente, se puede programar como sigue en *Mathematica*:

```
MillerRabin[n_?OddQ, k_] := Module[{i = 1, t = True},
     If[ n > 4,
       While[t == True && i <= k,
         t = TestdeMiller[n, Random[Integer, {2, n - 2}]]; i++],
       t = (n == 3)]; t]/;n > 0
```

Si se elige $k = O((\lg n)^d)$ para un entero positivo d, el test de Miller-Rabin funciona en tiempo polinómico $O(k(\lg n)^3) = O((\lg n)^{d+3})$ pero como entre las k bases elegidas pueden no estar todas las posibles, el algoritmo puede terminar declarando que n es primo (puesto que ha pasado los k tests de Miller) cuando en realidad es compuesto. Sin embargo, esto es muy poco probable pues, en virtud del teorema de Monier-Rabin, antes mencionado, se tienen las posibilidades siguientes para un entero impar n sometido al test de Miller-Rabin:

- *Si n es primo, entonces n pasará los k tests y será declarado primo.*
- *Si n es compuesto, la probabilidad de que el algoritmo termine con n declarado primo es* $\leq \frac{1}{4^k}$.

Esto significa que el test de Miller-Rabin puede ser considerado como una búsqueda probabilística de una demostración de que n es compuesto que, de ser así, terminará muy probablemente hallándola (encontrando una base para la cual n no pasa el test de Miller). Esto puede expresarse en términos de la teoría de la complejidad computacional. Un problema de decisión (con respuesta "sí" o "no") se dice que pertenece a la clase P (de *tiempo polinómico*) cuando existe un algoritmo de tiempo polinómico para resolverlo. Más generalmente, se dice que el problema pertenece a la clase RP (de *tiempo polinómico aleatorio*) cuando existe un algoritmo de tiempo polinómico con la propiedad de que si la respuesta es "no", entonces el algoritmo obtiene siempre la respuesta correcta ("no"), mientras que si la respuesta es "sí", el algoritmo responderá correctamente "sí" al menos el 50% de las veces. En ocasiones se dice también que una propiedad está en la clase RP (o en la clase P) cuando el problema de decisión asociado pertenece a RP (respectivamente, a P) y es claro que se verifica la inclusión $P \subseteq RP$. El test de Miller-Rabin demuestra que la propiedad de ser un entero *compuesto* está en RP o, lo que es lo mismo, que el problema de decisión *¿Es n compuesto?* pertenece a dicha clase. Obsérvese que esto no es lo mismo que demostrar que la propiedad de ser primo pertenece a RP (esto último es mucho más difícil pero también ha sido demostrado

en 1988 por Adleman y Huang [1], usando un test de primalidad basado en "variedades abelianas" que tiene únicamente importancia teórica). Por esta razón a veces se dice, más propiamente, que el test de Miller-Rabin es un *test de composición* probabilístico.

El test de Miller-Rabin deja abierto el problema de si la propiedad de ser compuesto pertenece a P. Esto es equivalente a que la propiedad de ser primo pertenezca a P pues es evidente que un test que pueda resolver el problema ¿Es n compuesto? en tiempo polinómico también responde en tiempo polinómico a la pregunta ¿Es n primo? (en otras palabras, la clase complementaria de la clase P, formada las propiedades cuya negación puede ser probada en tiempo polinómico, coincide con la propia P). Así nos preguntamos *¿La propiedad de ser primo está en P?*

La respuesta no se conoce, pero hay un resultado importante en esta línea, en el cual juega un papel la Hipótesis de Riemann extendida (HRE) [21]:

- *Si HRE se verifica, entonces la propiedad de ser primo está en P*

Esto es consecuencia de un resultado de Ankeny (1952) a partir del cual se puede deducir que si HRE se verifica y n es un número compuesto impar, entonces n no es seudoprimo fuerte para alguna base $b = O((\lg n)^2)$. Más tarde Bach obtuvo, en 1985, una cota explícita que permite afirmar que, de hecho, n no es un seudoprimo fuerte para alguna base $b \leq 2(\ln n)^2$. En consecuencia, si HRE es cierta, el llamado *"test de Miller determinista"*, que es simplemente el test de Miller tomando como bases todos los b tales que $2 \leq b \leq 2(\ln n)^2$, es un test de primalidad determinista de tiempo polinómico. De hecho, como el test de Miller tiene tiempo $O((\lg n)^3)$ y se requieren a lo sumo $O((\lg n)^2)$ iteraciones de dicho test, el tiempo de ejecución del test de Miller determinista es $O((\lg n)^5)$.

El test de Miller determinista se puede programar en *Mathematica* de la forma siguiente:

```
MillerDeterminista[n_?OddQ] :=
  Module[{i = 2, t = True, l = Floor[2*Log[n]^2]},
    If[ n > 13, While[t == True && i <= l,
       t = TestdeMiller[n, i]; i++],
    t = (n == 3 || n == 5 || n == 7 || n == 11 || n == 13)]; t]
/; n > 1
```

pero, a pesar de ser de tiempo polinómico, es poco eficiente y no sirve para primos como los de RSA.

Sin embargo, el test de Miller-Rabin con un número pequeño k de bases (se suele tomar un valor de k muy inferior al requerido por el test de Miller determinista, cuando n es un número del tamaño de los primos que se usan habitualmente en RSA) es muy eficiente y permite obtener rápidamente "primos probables", sobre los cuales no se tiene la seguridad de que sean realmente primos. Henri Cohen les llamó *"primos industriales"*, puesto que han pasado el test de Miller-Rabin con k bases y sabemos que a lo sumo 1 entre cada

4^k números compuestos pasa dicho test. La probabilidad de que un número compuesto pase este test se puede hacer menor que cualquier cota fijada de antemano eligiendo k suficientemente grande para que 4^{-k} sea inferior a dicha cota. Por ejemplo, si se elige $k = 84$, se tiene que esta probabilidad es inferior a 10^{-50}, una cifra sobre la que E. Borel dice en *Las probabilités et la vie* (citado en [25]): *"Un fenómeno cuya probabilidad es* 10^{-50} *no se producirá, pues, jamás o, al menos, no será jamás observado"*. Si uno quiere aun mayor seguridad puede optar por tomar $k = 135$ y en ese caso la probabilidad de que un número compuesto aleatorio n pase el test sería inferior a 10^{-80}, es decir, inferior a la probabilidad de elegir un átomo predeterminado entre todos los que forman el Universo. Estas probabilidades son muy inferiores a la probabilidad estimada de un "fallo de hardware" en cualquier máquina, que en caso de producirse podría hacer que incluso un test con probabilidad teórica de error igual a cero diese lugar a un resultado erróneo. En cualquier caso los "primos industriales" podrían ser usados en la industria y el comercio en las condiciones usuales, es decir, con garantía de sustitución en caso de resultar defectuosos.

Pero aun hay más, pues la efectividad del test de Miller-Rabin es, de hecho, muy superior a la predicha por la cota proporcionada por el teorema de Monier-Rabin. Para comprobar esto podemos hacer un sencillo experimento. Si la probabilidad de que un número compuesto pase el test de Miller para una base dada estuviese próxima a 1/4, entonces, teniendo en cuenta que:

```
Length[Select[10^15 + Range[1,1000, 2], ! PrimeQ[#] &]]
476
```

muestra que en el intervalo $[10^{15}, 10^{15} + 1000]$ hay, exactamente, 476 números compuestos impares (recuérdese que, para $n < 10^{16}$, `PrimeQ` da siempre la respuesta correcta), cabría esperar que el test de Miller-Rabin con una sola base declarase primos a más de cien de estos números. Pero en la práctica esto no ocurre, pues si se ejecuta:

```
Length[Select[Map[MillerRabin[#, 1] &,
  Select[10^15 + Range[1,1000, 2], ! PrimeQ[#]&]], # == False &]]
```

el resultado habitual es, de nuevo, `476` (aunque, por supuesto, dado que el test utiliza una fuente de números (seudo)aleatorios, existe la posibilidad de obtener un número ligeramente inferior). Este experimento se puede repetir con otros intervalos, y se puede observar que siempre se obtienen resultados parecidos, enormemente más favorables que los sugeridos por el teorema de Monier-Rabin. La razón de este fenómeno es que los seudoprimos fuertes son relativamente escasos y abundan mucho menos que los primos. Si hacemos:

```
SeudoprimoFuerte[n_, b_] := ! PrimeQ[n] && TestdeMiller[n, b];
SetAttributes[SeudoprimoFuerte, Listable];
```

tenemos un criterio para determinar si un entero n es seudoprimo fuerte para la base b. Por ejemplo, los 2-seudoprimos fuertes menores que 100000 son:

```
Select[Range[5, 100000, 2], SeudoprimoFuerte[#, 2] &]

{2047, 3277, 4033, 4681, 8321, 15841, 29341, 42799, 49141,
52633, 65281, 74665, 80581, 85489, 88357, 90751}
```

Podemos fácilmente definir una función que comprueba si un número es seudoprimo fuerte para alguna base prima $\leq$ que un entero dado r (o, con una modificación obvia, para alguna base $\leq r$ no necesariamente prima, pues tiene sentido considerar bases que son números compuestos, aunque lo más usual es considerar sólo primos).

```
SeudoprimoFuerteHasta[n_,r_] :=
  And @@ SeudoprimoFuerte[n, Eratostenes[r]];
Select[Range[3, 2000000, 2], SeudoprimoFuerteHasta[#, 3] &]

{1373653, 1530787, 1987021}
```

El cálculo anterior nos muestra que sólo hay tres números menores que 2000000 que sean a la vez 2- y 3-seudoprimo fuerte, y ninguno menor que un millón; siendo 52953 los enteros menores que 10^{16} con esta propiedad, según cálculos de D. Bleichenbacher [7]. Aunque este número puede parecer grande, es pequeño en comparación con el número de primos en ese mismo rango, que es $\pi(10^{16}) = 279238341033925$, de modo que si un entero aleatorio menor que 10^{16} pasa el test de Miller para las bases 2 y 3, es altamente probable que sea primo.

Podemos observar esto desde un punto de vista ligeramente diferente si nos preguntamos, dado un entero compuesto n, cual es la menor base b para la cual n no es b-seudoprimo fuerte, es decir, la menor base b que revela que n es compuesto, por lo cual recibe el nombre de *"primer testigo fuerte"* (para n). Como hemos visto al discutir el test de Miller determinista, esta base es $\leq 2(\ln n)^2$ si HRE se verifica, pero parece que esta cota no es muy ajustada y que, en la práctica, la situación es mucho más favorable. De hecho, tanto los resultados obtenidos computacionalmente como ciertos argumentos heurísticos [5, p. 315] sugieren que el número de bases requerido por el test determinista debería ser más bien $O((\lg n)(\lg\lg n))$. El programa:

```
PrimerTestigoFuerte[n_?OddQ]:= Module[{b = 2},
If[PrimeQ[n], Print[StringForm["`` es primo", n]],
While[TestdeMiller[n, b], b++]; b]]/;n > 1
```

busca el "primer testigo fuerte" para n. Por ejemplo, se tiene que:

```
PrimerTestigoFuerte[341550071728321] = 23
PrimerTestigoFuerte[4498414682539051] = 19
```

siendo estos los dos únicos números menores que 10^{16} cuyo primer testigo fuerte es mayor que 17. De hecho, aunque el primer testigo fuerte de un número

compuesto no es necesariamente primo, se puede ver fácilmente a partir de la lista de 2- y 3-seudoprimos fuertes menores que 10^{16} compilada por Bleichenbacher [7] que, si se excluye a los dos números citados cuyo primer testigo fuerte es 23 y 19, respectivamente, todos los demás compuestos menores que 10^{16} no pasan el test de Miller para alguna base *prima* ≤ 17. En consecuencia, el test de Miller para las bases primas comprendidas entre 2 y 17 es un test de primalidad determinista para los números < 341550071728321. Más aun, para los números (impares) $\leq 10^{16}$, el siguiente programa proporciona un test de primalidad determinista muy eficiente:

```
P17 = {2, 3, 5, 7, 11, 13, 17};
T[n_?OddQ] := If[n <= 17, MemberQ[P17, n],
   n != 341550071728321 && n != 4498414682539051 &&
     And @@ TestdeMiller[n, P17]] /; 1 <= n <= 10^16
```

Se han construido (por métodos 'ad hoc') números que son seudoprimos fuertes para muchas bases, como el siguiente de 56 dígitos, debido a R. Pinch:

```
PrimerTestigoFuerte[6852866339504691224422360590273835\
67197510827843866810 71]
101
```

o un número de 337 dígitos construido por Arnault en su tesis [3] que, aun teniendo 14 como primer testigo fuerte, tiene la propiedad de ser seudoprimo fuerte para todas las bases primas menores que 200. Pero en todos los casos mencionados (y en otros que han sido estudiados), se observa que no sólo el primer testigo fuerte (y, más aun, la primera base prima b para la cual el número n no pasa el test de Miller) está por debajo de $2(\ln n)^2$, sino incluso por debajo de $\ln n$, lo cual tiende a confirmar la hipótesis de que este número es $O((\lg n)(\lg\lg n))$. Por otra parte, todos estos números sucumben fácilmente al test de Miller-Rabin y se puede asegurar que el que un número compuesto grande pase dicho test con 25 bases aleatorias se sitúa en la categoría de fenómenos inobservables de que hablaba Borel. Mejor aun, se puede ver que la probabilidad de que un número grande que ha pasado el test de Miller-Rabin para un número pequeño de bases sea, a pesar de todo, compuesto, también es extremadamente baja.

Para analizar esto, hay que tener en cuenta que la probabilidad de que un número declarado primo por el test sea compuesto (a diferencia de la probabilidad de que un número compuesto sea declarado primo) no se puede acotar usando sólo el teorema de Monier-Rabin. Sea $P(r,k)$ la probabilidad de que un número impar de r bits, elegido aleatoriamente, y que ha pasado el test de Miller-Rabin con k bases aleatorias sea compuesto (es decir, la probabilidad de error del test de Miller-Rabin). Por el teorema del número primo, la probabilidad de que un entero impar aleatorio de 512 bits sea primo es, aproximadamente, $\frac{1}{177}$. Por tanto, si la probabilidad de que un número compuesto pasase el test para una sola base fuese aproximadamente $1/4$, se tendría claramente que $P(512,1) > 1/2$, pues sería mucho más probable tener la "mala

suerte" de escoger un compuesto que pasa el test (con probabilidad cercana a $\frac{1}{4}$) que tener la "buena suerte" de escoger un primo (cuya probabilidad es sólo $\frac{1}{177}$). No obstante, esto no ocurre, porque la probabilidad de que un número compuesto pase el test para una sola base es usualmente mucho menor que $\frac{1}{4}$. De hecho, en [11] se demuestra que también $P(r,k) < 4^{-k}$ y, para números grandes, la situación es aun mucho más favorable pues $P(r,k)$ va decreciendo al aumentar r; por ejemplo, $P(500,1) < 4^{-28}$ y $P(500,10) < 4^{-60}$. Por tanto, un número de 512 bits elegido al azar y que pase el test de Miller-Rabin para una sola base aleatoria puede ser perfectamente considerado como un "primo industrial". Además, aunque no se menciona explícitamente en [11], se deduce de sus resultados que la probabilidad de que un número de 500 (o 512) bits que ha pasado el test de Miller-Rabin con 16 bases resulte ser compuesto, es inferior a 10^{-50}.

La función `PrimeQ` de *Mathematica* que hemos utilizado en muchos de los experimentos anteriores también usa el test de Miller y, como ya se ha indicado, no está claro que la respuesta que proporciona sea siempre correcta, por lo que es conveniente analizar un poco más su funcionamiento. `PrimeQ` comienza sometiendo al número al test de Miller para las bases 2 y 3. Si el número pasa el test para ambas bases, entonces es sometido a un "test de seudoprimos de Lucas" [10,16]. Si también pasa este test, entonces el número es declarado primo. La justificación de usar conjuntamente el test de Miller y un test de Lucas reside en la creencia en que ambos tipos de test son, en cierto sentido, "independientes" y, por tanto, es altamente improbable que un seudoprimo fuerte también pase un test de Lucas (aunque existen argumentos heurísticos que hacen pensar que esto debe de ser posible). De hecho, existe un premio en metálico, aunque más bien simbólico, para la primera persona que encuentre un número que es 2-seudoprimo fuerte y pasa un test de Lucas determinado. Este premio, que originalmente era de \$30, fue ofrecido por Pomerance, Selfridge y Wagstaff en su artículo [36]. A lo largo del tiempo, los autores han ido aumentando la cantidad para compensar la inflación y el premio ofrecido en la actualidad es de \$620, pero nadie lo ha reclamado todavía, a pesar de que los autores han rebajado las condiciones y el 2-seudoprimo ya no necesita ser "fuerte" (cf. [10, p.271] para los detalles sobre este problema, también llamado "desafío PSW", por las iniciales de sus autores). El desafío PSW tiene bastante interés práctico porque una combinación del test de Miller (o de Miller-Rabin) y un test de Lucas es el método adoptado usualmente por muchos sistemas de cálculo simbólico para comprobar la primalidad. Obsérvese que, en el caso de *Mathematica*, encontrar un ejemplo de un número compuesto que es declarado primo por `PrimeQ` es todavía más difícil que resolver el desafío PSW, porque en este caso no se usa el test de Fermat sino el de Miller y, además, para las bases 2 y 3. Por todo ello, el uso de `PrimeQ` es perfectamente válido para la obtención de datos estadísticos como los manejados en la primera parte de estas notas [21] y, en lo que se refiere a su uso en los ejemplos anteriores relacionados con el test de Miller-Rabin, sabemos que no hay problema pues,

según los cálculos de Bleichenbacher [7], ningún compuesto inferior a 10^{16} es declarado primo por `PrimeQ`.

La que es posiblemente la última versión (por ahora) del desafío PSW aparece en [16, Research Problem 3.41] y en [10, p. 273], donde recibe el nombre de "Desafío de seudoprimos de Fibonacci". Consiste en encontrar un entero n que sea $\equiv \pm 2 \pmod 5$ (es decir, terminado en 3 o en 7, si nos restringimos a números impares) y, a la vez, un 2-seudoprimo y un seudoprimo de Fibonacci (un tipo particular de seudoprimo de Lucas caracterizado, en el caso de ser $n \equiv \pm 2 \pmod 5$, por ser un número compuesto n que divide a F_{n+1}, siendo F_i el i-ésimo número de Fibonacci, definido recursivamente por $F_0 = 0$, $F_1 = 1$, $F_i = F_{i-1} + F_{i-2}$). En [16] se detalla que de los \$620 ofrecidos corresponden \$500 a Selfridge, \$100 a Wagstaff, y \$20 a Pomerance, pero que si alguien demuestra que un tal n no existe, entonces también recibirá \$620, con Pomerance y Selfridge intercambiando sus papeles. Para tratar de resolver el desafío PSW podríamos usar:

```
Fermat2[n_] := OddQ[n] && PowerMod[2, n - 1, n] == 1 /; n > 1
```

que devuelve `True` si n es primo o 2-seudoprimo, junto con:

```
TestdeFibonacci[n_] := (Mod[n, 5] == 2 || Mod[n, 5] == 3) &&
     Mod[Fibonacci[n + 1], n] == 0
```

que devuelve `True` cuando $n \equiv \pm 2 \pmod 5$ y, además, n es o bien primo o bien un seudoprimo de Fibonacci. Para asegurarnos de que n es compuesto no es bueno usar `PrimeQ` en este caso, pues lo que estamos buscando podría ser también (aunque no necesariamente) un número compuesto que fuese declarado primo por `PrimeQ`. Lo mejor sería partir de números n que ya se sabe que son compuestos por haberlos construido multiplicando otros números y aplicarles

```
Fermat2[n] && TestdeFibonacci[n]
```

con la esperanza de obtener como resultado, para alguno de ellos, `True`. En la página Web [22], Alford y Grantham dan una lista de 2030 primos de los cuales creen (basándose en un argumento heurístico de Pomerance) que algún subproducto pasará el desafío PSW.

La discusión anterior sugiere que el test de Miller-Rabin, por si sólo o en combinación con un test de Lucas, es perfectamente adecuado para proporcionar primos (industriales) como los requeridos por RSA. No obstante, es posible eliminar totalmente la ínfima probabilidad de que un número compuesto sea declarado primo, pues existen tests de primalidad deterministas eficientes que nos dicen con total certeza teórica si un entero arbitrario n es primo o si, por el contrario, n es compuesto. Uno de los más potentes es el test de Adleman-Pomerance-Rumely [2], mejorado por H. Cohen y H.W. Lenstra [13] e implementado por H. Cohen y A.K. Lenstra [14], por lo cual es conocido por las siglas APR-CL (también es llamado, en sus diferentes variantes, el

test de las sumas de Jacobi [16], el test de los anillos ciclotómicos [5] o CPP ("Cyclotomy Primality Proving") [32]). La alternativa a este test es el llamado test de las curvas elípticas (ECPP, iniciales de "Elliptic Curve Primality Proving") que apareció primero en forma teórica en [20] y fue desarrollado en forma práctica por Atkin y Morain [4].

El tiempo de ejecución de CPP es $O((\lg n)^{c \lg\lg\lg n})$ (donde c es una constante) [5]. Por tanto, CPP no es de tiempo polinómico, pero como la función anterior tiene un crecimiento muy lento (por ejemplo, $\lg\lg\lg n$ vale 5 cuando n tiene un millón de dígitos decimales), se puede considerar que, a efectos prácticos, se comporta como una constante [12]. Por esta razón, el test es sólo ligeramente más lento que un test de tiempo polinómico y se dice a veces que es de tiempo *superpolinómico* [45, p. 170]. En la práctica, este test puede comprobar números de 155 dígitos decimales como los que se usan en RSA en unos cuantos segundos y sirve también para probar la primalidad de números de miles de dígitos.

El tiempo de ejecución de ECPP ha sido estimado heurísticamente en $O((\lg n)^{6+\varepsilon})$, con $\varepsilon > 0$, aunque aplicando técnicas de multiplicación rápidas este tiempo podría ser reducido a $O((\lg n)^{5+\varepsilon})$ [27]. Hay que hacer notar, sin embargo, que este test es "probabilístico" en el sentido de que depende de una fuente de números aleatorios y en algunos casos podría incluso no terminar. En consecuencia, se dice que funciona en *tiempo polinómico esperado* pues el tiempo indicado es sólo "por término medio" y para algunos inputs el tiempo de ejecución podría ser mucho mayor [12].

En la práctica, cabe esperar que ECPP sea más rápido asintóticamente que CPP, pero para valores relativamente pequeños de n este último es más rápido. De hecho, en [32] se indica que la complejidad *de facto* de CPP para números de hasta un millón de dígitos decimales es $O((\lg n)^{4,5})$ por lo que sería más rápido que ECPP en este rango. De todos modos, existe una versión de ECPP, programada por M. Martin [31] bajo el nombre de *Primo* que funciona en PCs y certificó, usando 13 semanas de proceso en un AMD Athlon de 1,3 Ghz durante Junio/Septiembre de 2001, que el número $10^{5019}+43157099231631693$, de 5020 dígitos decimales, es primo (este es el record actual de ECPP). También ha sido *Primo* (y su antecesor *Titanix*) el programa usado para certificar los primos titánicos que aparecen en [21].

Hay que destacar que estos tests de primalidad pueden demostrar que un entero es compuesto pero en ese caso no proporcionan información alguna sobre sus factores primos, es decir, no tienen utilidad para factorizar un número n salvo en el caso particular en que n ya sea primo. Sin embargo, ECPP tiene la ventaja de proporcionar un certificado de primalidad que, una vez probado que n es primo, permite comprobar que efectivamente es así muy rápidamente, sin necesidad de repetir todo el cálculo.

Veamos ya la forma concreta de obtener primos para RSA usando *Mathematica*. Un posible enfoque, recomendado, por ejemplo, en [39, p. 234], para buscar un primo aleatorio del tamaño requerido es tomar primero un entero aleatorio de dicho tamaño y a continuación buscar el primo siguiente (que

podría tener un bit más). Por ejemplo, podríamos hacer, usando la función `PrimoSiguiente` definida en [21]:

```
PrimodeLongitud[bits_] := Module[{t=1},
n = Random[Integer,{2^(bits-1), 2^bits-1}];
PrimoSiguiente[n]]/; bits > 1
```

Este es también el enfoque que utiliza la función `RandomPrime` que viene con [10], pero veamos lo que ocurre si seleccionamos 512000 primos de 10 bits y contamos el número de veces que entre ellos aparecen el 883 y el 907:

```
A = Table[PrimodeLongitud[10], {512000}];
   {Count[A, 883], Count[A, 907]}

{2019, 19928}
```

El resultado es muy desigual y la explicación es muy sencilla: debido a la irregular distribución de los primos, estos no son elegidos con distribución de probabilidad uniforme. Como el 883 está precedido por un hueco de longitud 2, es previsible que aparezca unas 2000 veces, por término medio, y como el 907 está precedido por un hueco de longitud 20, deberá aparecer con una frecuencia diez veces superior, es decir, unas 20000 veces por término medio (suponiendo que la función `Random` de *Mathematica* tenga distribución de probabilidad uniforme, lo que estos experimentos tienden a confirmar). Por supuesto que, al ir creciendo el tamaño del primo buscado, las oscilaciones en el tamaño de los huecos se van haciendo mayores y, en caso de ser cierta la conjetura de Shanks sobre la distribución de los huecos mencionada en [21], existirían primos de 512 bits cuya probabilidad de ser elegidos por este método sería casi 25000 veces superior a la de otros primos del mismo tamaño. A pesar de todo, si se utiliza este método para seleccionar un primo de 512 bits para RSA, un criptoanalista no podría obtener ninguna ventaja de estas oscilaciones, debido al enorme número de primos que hay de este tamaño, que excluye la posibilidad de factorizar el módulo conjeturando los factores más probables. Un caso distinto –aunque similar en el fondo– que sí tuvo consecuencias prácticas ocurrió en 1999, cuando se produjo la OPV de acciones de Terra Networks y, ante la enorme demanda, las acciones se asignaron por sorteo entre todos los solicitantes. El método elegido consistió en ordenar a éstos por orden alfabético del nombre de pila, elegir por sorteo una letra del alfabeto y asignar las acciones a los peticionarios cuyo nombre comenzase por dicha letra y por las letras siguientes (ordenadas alfabéticamente en forma cíclica) hasta agotar las acciones disponibles. Teniendo una idea de la distribución aproximada de las iniciales de los nombres en España es fácil darse cuenta que algunas iniciales (por ejemplo, la A) resultaban muy favorecidas frente a otras. Al final, muchos de los solicitantes cuyo nombre empezaba por A (aunque no todos) consiguieron las acciones, a pesar de que la letra que había salido en el sorteo estaba situada bastante más atrás.

Si queremos obtener un "primo aleatorio" de tamaño dado, con distribución de probabilidad uniforme (aproximadamente), podemos modificar el programa anterior. Se puede aplicar una criba (de Eratóstenes) a un intervalo centrado en un entero aleatorio del tamaño requerido y, a continuación, aplicar de nuevo `Random` de forma conveniente para elegir un entero entre los que sobrevivieron a la criba, repitiendo este proceso hasta encontrar uno que sea declarado primo por `PrimeQ`. Podría ocurrir que el intervalo elegido no contuviese ningún primo, por lo que habría que descartarlo al cabo de un cierto número de intentos fallidos y probar con un nuevo intervalo obtenido a partir de otro "entero aleatorio". Una alternativa, que resulta un poco más sencilla en este caso, es prescindir de la criba. Dado que el proceso podría ser un poco lento si se busca un primo muy grande y se utilizan sólo `Random` y `PrimeQ`, podemos utilizar la prueba de división para eliminar los enteros elegidos por `Random` que sean divisibles por primos pequeños, sin necesidad de llegar a aplicarles `PrimeQ`. Para ello, podemos utilizar la función:

```
PruebaDivision[n_Integer, r_Integer] := Module[{i = 1},
     While[i <= r && Mod[n, Prime[i]] != 0, i++];
     If[i > r, True, False]]/;r>0 && n > 1.7*r*Log[r]
```

que devuelve `True` si el entero n no es divisible por ninguno de los r primeros primos (y `False` en caso contrario). A continuación se puede usar:

```
PrimoAleatorio[bits_] := Module[{t = 1, s = bits},
   While[! PrimeQ[t], t = Random[Integer,{2^(bits-1), 2^bits-1}];
   While[PruebaDivision[t, s] = False,
     t = Random[Integer,{2^(bits-1), 2^bits-1}]]]; t]/;bits > 1
```

Por ejemplo, se puede obtener:

```
p = PrimoAleatorio[512]

11861541811872445857172290426088273901422620325695907\
17302339862620349714184158948146139325009497407158006\
10249629464651100984665158992857412721272239609333
```

Si queremos confirmar que este número es primo, podemos usar la función de *Mathematica* `ProvablePrimeQ` que usa el método de las curvas elípticas (Atkin-Morain) para *demostrar* que el número es, efectivamente, primo:

```
Needs["NumberTheory`PrimeQ`"]
ProvablePrimeQ[p]

True
```

Además, `ProvablePrimeQ` puede proporcionar un certificado que permite comprobar rápidamente la primalidad del número, sin tener que repetir todos los

cálculos del test ECPP. Si se quiere que *Mathematica* imprima este certificado, basta poner `ProvablePrimeQ[p, Certificate -> True]` pero no mostraré aquí el resultado de ejecutar esto sobre nuestro primo p de 512 bits, pues el certificado impreso ocupa unas cinco páginas. Sin embargo, si llamamos `cert` a dicho certificado y, a continuación, hacemos `PrimeQCertificateCheck[cert, p]` *Mathematica* nos responde en muy pocos segundos: `True`. Si el primo fuese más grande (por ejemplo, de 2048 bits o, incluso, como ya hemos visto, un primo titánico), entonces es conveniente usar el programa *Primo* al que me referí anteriormente, para certificar que es primo.

Si el primo buscado fuese a ser utilizado en RSA, se podrían tomar una serie de precauciones adicionales relacionadas con la seguridad de RSA, con objeto de hacer más difícil la factorización del módulo (por ejemplo, haciendo que $p-1$ y también $p+1$ tengan algún factor primo grande) pero no discutiremos esta cuestión en detalle por dos razones. La primera de ellas es que es una cuestión demasiado técnica y la segunda que no está claro que sea necesario usar en RSA "primos fuertes", es decir, primos que no sólo se han elegido aleatoriamente sino que se ha comprobado que cumplen condiciones adicionales como las antes mencionadas. De hecho, Rivest y Silverman argumentan en [40] que el uso de tales "primos fuertes" es innecesario y que los "primos aleatorios" ofrecen prácticamente el mismo nivel de seguridad.

Las observaciones anteriores confirman que el reconocimiento de primos es "fácil" y ello facilita la implementación de RSA, pero queda por analizar el problema de su seguridad. Como hemos observado, se cree que romper el criptosistema RSA tiene el mismo nivel de dificultad que factorizar el módulo n y así la pregunta ¿Es RSA realmente seguro? es probablemente equivalente a la siguiente: ¿Es el problema de la factorización de enteros difícil? La experiencia de que se dispone parece confirmar que es así aunque este hecho no está demostrado en el sentido matemático del término. De hecho, la única evidencia existente en el sentido de que la factorización es un problema computacionalmente difícil es de tipo histórico. En palabras de los hermanos Lenstra [27, p. 4] *"generaciones de matemáticos, un pequeño ejército de informáticos, y legiones de criptólogos dedicaron una considerable cantidad de energía a este problema y lo mejor que consiguieron son unos algoritmos relativamente pobres"*. No se puede excluir, sin embargo, la posibilidad de que alguien descubra algún día un algoritmo eficiente (e incluso, de tiempo polinómico) para factorizar enteros, en cuyo momento RSA dejaría de ser útil.

Desde el punto de vista teórico es muy fácil, dado un entero compuesto n, demostrar que n efectivamente lo es. Basta tomar un factor propio d de n y comprobar que d divide a n (o bien multiplicar dos factores cuyo producto es n). Sin embargo, hallar estos factores puede ser muy difícil. Hay una anécdota bien conocida que ilustra perfectamente este hecho. Sucedió en 1903, año en el que no se conocía la factorización del número de Mersenne $M_{67} = 2^{67} - 1$, el cual tiene 21 dígitos decimales. En una reunión de la American Mathematical Society, Frank N. Cole pronunció una "conferencia silenciosa" en la que, sin decir palabra, calculó por el método de exponenciación binaria la expresión

decimal de dicho número, escribiendo en la pizarra:

$$147573952589676412927$$

A continuación multiplicó

$$193707721 \times 761838257287$$

comprobando en pocos minutos que los resultados de ambos cálculos coincidían. De forma inusual, la sala estalló en aplausos hasta que Cole se sentó en silencio. La cosa quedó así hasta que en 1911 Eric T. Bell le preguntó a Cole como había hallado los factores. La respuesta de Cole fue: *"Me costó tres años de domingos"*.

Las propiedades que, como la de ser n compuesto, pueden ser comprobadas en tiempo polinómico con la ayuda de un "certificado" (que en el caso de un compuesto n es un factor propio de n) se dice que pertenecen a la clase NP (de *tiempo polinómico no determinista*). Es decir, son aquellas propiedades que pueden ser demostradas fácilmente (en tiempo polinómico) aunque no se conoce un algoritmo fácil (de tiempo polinómico) para encontrar una demostración. En el caso de la propiedad de ser n compuesto, su pertenencia a NP se puede deducir también como consecuencia de su pertenencia a la clase RP puesto que, obviamente, $RP \subseteq NP$. En este caso, si b es una base para la cual n no pasa el test de Miller, b es el certificado que permite comprobar que n es compuesto en tiempo polinómico. De hecho, como ya hemos observado, si la HRE es cierta, entonces la propiedad de ser n compuesto pertenece a la clase P. Sin embargo, cuando Cole pronunció su "conferencia silenciosa" hizo algo más que demostrar que $2^{67} - 1$ era compuesto, lo que por entonces ya se conocía a través del test de Lucas. Lo que Cole resolvió para $n = 2^{67} - 1$ fue el *problema de la factorización* que se puede plantear como un *problema de decisión* con respuesta sí/no en la forma siguiente: Dados dos enteros positivos n y k, ¿tiene n algún factor d tal que $2 \leq d \leq k$? Esta versión del problema parece más débil que lo que Cole hizo en realidad, que fue hallar los factores de n, pero ambos problemas se puede ver que son equivalentes en el sentido de que a partir de un algoritmo para uno de ellos se puede obtener (en tiempo polinómico) un algoritmo para el otro. En este caso, si se dispone de un algoritmo que resuelve el problema de decisión correspondiente a la factorización, se obtiene un algoritmo para obtener los factores del n por medio de una "búsqueda binaria" [26] y el proceso de reducción que permite pasar de un algoritmo al otro requiere sólo tiempo polinómico. El problema de decisión de la factorización de enteros se puede resolver en tiempo polinómico con ayuda de un certificado (un factor) y por tanto, se puede decir que pertenece a la clase NP. Fue este hecho lo que permitió a Cole dar su conferencia en unos pocos minutos, pero es muy probable que este problema no pertenezca a la clase P y por eso Cole tuvo que emplear "tres años de domingos" en la preparación de dicha conferencia.

La idea de la clase NP es que existe un algoritmo no determinista que en primer lugar conjetura un "certificado" y a continuación trata de verificar en tiempo polinómico que dicha conjetura es correcta. Es evidente que la clase P está contenida en NP y la cuestión de determinar si dichas clases son –como se piensa– distintas o si, por el contrario, se verifica que $P = NP$ es el problema abierto más importante de la Teoría de la Computación y es uno de los siete "Problemas del Milenio" para los que el Clay Mathematics Institute ha establecido premios de un millón de dólares [15]. Tal como demostró S. A. Cook en 1971 existen en la clase NP un tipo de problemas llamados *NP-completos* que tienen la propiedad de que la solución de cualquier otro problema de NP se reduce a la de cualquiera de ellos y el proceso de reducción requiere tiempo polinómico. Esto significa que los problemas NP-completos son al menos tan difíciles como cualquier otro problema de la clase NP y, por tanto, si se consiguiera resolver un problema NP-completo en tiempo polinómico, se habría demostrado que $P = NP$. Pero no se cree que esto vaya a ocurrir y si, como se piensa, la clase P está contenida propiamente en la clase NP, entonces se puede concluir que los problemas NP-completos son realmente "difíciles". El hecho de que actualmente se conocen muchos problemas NP-completos, y el que estos problemas son aparentemente intratables, es una de las principales razones en las que se basa la creencia de que $P \neq NP$. Entre los problemas NP-completos figuran, por ejemplo, el *problema del viajante*, consistente en: Dada una colección de lugares, una tabla de las distancias entre ellos, y un número k, ¿existe un recorrido por todos esos lugares cuya distancia total sea $\leq k$? y muchos otros, pero no el problema de la factorización, que no está demostrado que sea NP-completo. Por tanto, aun pensando que los problemas NP-completos son realmente difíciles, para admitir la dificultad de la factorización tenemos que contentarnos, de momento, con la evidencia histórica. Si, por el contrario, se llegase a demostrar un día que $P = NP$, entonces ese mismo día el criptosistema RSA habría pasado a ser historia [15].

El problema de la factorización ha llegado a ser calificado incluso de "feroz". En [23, p. 267] se cuenta como el *New York Times* publicó en su primera página, en su edición del día 12 de octubre de 1988, un artículo de Malcolm W. Browne bajo el título:

"A Most Ferocious Math Problem Tamed"

en el que se describe la factorización de un número de 100 dígitos decimales (con factores primos de 41 y 60 dígitos) por el método de la "criba cuadrática". Uno de los subtítulos de dicho artículo es todavía más espectacular si cabe: *"World's Fiercest Math Problem is Tamed by Hundreds of Computers"*. Por supuesto que estos titulares tienen una buena dosis de sensacionalismo, pero un breve repaso a la historia del problema puede ayudar a comprender los motivos que llevaron al periódico a utilizar tanta estridencia. Al ser un problema muy exigente desde el punto de vista computacional, no se desarrolló realmente hasta tiempos muy recientes. Durante las últimas décadas la potencia de los ordenadores aumentó espectacularmente, haciéndose al mismo tiempo

barata y accessible y este fenómeno trajo consigo otros dos que hicieron que el problema de la factorización pasase de ser sólo apreciado por algunos "numerólogos" extravagantes, a ser un problema matemático de gran interés al que se dedican enormes recursos. El primero de estos fenómenos es la aparición de la teoría de la complejidad computacional, que proporcionó a los problemas computacionales un aire de respetabilidad que anteriormente no tenían. El segundo es, precisamente, la aparición de la criptografía de clave pública y, más concretamente, el criptosistema RSA al que me he estado refiriendo. Como consecuencia de todo esto, se puede decir –en palabras de Crandall y Pomerance en [16, p. 2]– que *el teorema fundamental de la aritmética da lugar a lo que se puede llamar "el problema fundamental de la aritmética": dado un entero $n > 1$ hallar su factorización prima*. Naturalmente, este problema incluye también el reconocimiento de primos, pero ahora me referiré a la factorización propiamente dicha, o *escisión*, es decir, al problema de, dado un entero compuesto encontrar un factor no trivial del mismo. Los avances en esta cuestión han sido tan grandes que, en las últimas décadas, la corona de "método de factorización más rápido" ha cambiado de manos en varias ocasiones. Dado que, por razones de espacio, resulta imposible tratar aquí este tema con detalle, me limitaré a hacer un breve comentario sobre el estado actual de la cuestión; en la página Web [18] se pueden encontrar muchos artículos interesantes y también software sobre el tema.

Desde hace pocos años, el método de factorización más eficiente que se conoce para factorizar "números difíciles", es decir, números que, como los módulos de RSA, tienen sólo dos factores primos de, aproximadamente, el mismo tamaño, es el conocido como *Criba del cuerpo de números* (Number Field Sieve, NFS) [16, 28, 35]. Este método fue propuesto originalmente en 1988 por John Pollard y, en su versión original solo servía para factorizar números compuestos que estaban próximos a potencias de enteros. Después, a través del trabajo de Buhler, Lenstra y Pomerance, fue adaptado para factorizar números generales y su consagración definitiva como el método de factorización más potente llegó en 1996, cuando un numeroso grupo consiguió factorizar RSA130, un número de 130 dígitos del "Desafío de factorización RSA" (RSA Factoring Challenge). Este concurso de factorización, con premios en metálico, fue establecido por la compañía norteamericana RSA Security, que tenía la patente de RSA en EE UU (caducada en el año 2000) con el objetivo de monitorizar los avances en factorización que se produzcan y tener así una idea precisa de la seguridad de los productos criptográficos que comercializan [41].

El "estado actual de la cuestión" en cuanto a seguridad de RSA es el siguiente. El 29 de agosto de 1999, un equipo formado por personas de seis países diferentes y dirigido por Herman te Riele del CWI (Centro de Matemáticas e Informática) de Amsterdam, consiguió factorizar RSA155, un número de 155 dígitos del "Desafío RSA".

RSA-155 =
109417386415705274218097073220403576120037329454492059909138421314763499842889
347847179972578912673324976257528997818337970765372440271467435315933543338 97
=
102639592829741105772054196573991675900716567808038066803341933521790711307779
×
106603488380168454820927220360012878679207958575989291522270608237193062808643

Esta factorización es importante porque RSA155 es un número de 512 bits y éste era, en ese momento, el tamaño del módulo usado en casi todas las implementaciones comerciales de RSA, estimándose que era también el tamaño del 95% de los módulos de RSA utilizados en el comercio electrónico (aunque es de suponer que esto cambiará pronto). En esta factorización se emplearon cientos de ordenadores y se desarrolló a lo largo de 5,2 meses, pero si se tiene en cuenta la preparación previa necesaria el tiempo total utilizado fue de 7,4 meses [18]. Más recientemente, se produjo el desenlace del desafío llamado "The Cipher Challenge", contenido en [43] y dotado con un premio de 10000 libras esterlinas para la primera persona que consiguiese descifrar diez mesajes encriptados con diferentes criptosistemas de importancia histórica. Para descifrar el último (el más difícil de todos), había que factorizar un módulo de RSA de 512 bits, lo que consiguió el 7 de octubre de 2000 un equipo sueco formado por F. Almgren, G. Andersson, T. Granlund, L. Ivansson y S. Ulfberg, quienes ese día completaron la factorización usando también NFS [18].

La consecuencia que se extrae de estas factorizaciones es que RSA con un módulo de 512 bits, que es el más utilizado actualmente, ya no es seguro. El problema es pues determinar que tamaño de módulo proporciona un nivel de seguridad razonable en la actualidad y durante cuanto tiempo se podrá mantener esa seguridad para un módulo dado. Dado que el tiempo de ejecución de NFS se estima heurísticamente que es, con una ligera simplificación, del orden de $O(e^{(64/9)^{1/3}(\ln n)^{1/3}(\ln\ln n)^{2/3}})$ [16, Theorem 6.2.2], dividiendo el valor que esta función toma para $n = 2^{1024}$ por el correspondiente a $n = 2^{512}$, se obtiene:

```
E^((64./9)^(1/3)*Log[2^1024]^(1/3)*Log[Log[2^1024]]^(2/3))/
   E^((64/9)^(1/3)*Log[2^512]^(1/3)*Log[Log[2^512]]^(2/3))
```

```
7.491251106629684*10^6
```

lo que significa que, en el paso de 512 a 1024 bits, el tiempo de ejecución se multiplicaría aproximadamente por $7,49 \cdot 10^6$. Tomando entonces como base que la factorización de RSA155 requirió 5,2 meses (sin contar el tiempo de preparación) se llega a la conclusión de que factorizar un módulo de 1024 bits con el mismo software y el mismo hardware podría requerir mas de 3 millones de años. Si en lugar de 1024 pasamos a un módulo de 2048 bits, entonces el tiempo de ejecución, siempre tomando como referencia el software y el hardware usado en el caso de RSA155, sería del orden de $3,78 \cdot 10^{15}$ años. Teniendo en

cuenta que la edad del Universo actualmente estimada es de unos 12000 millones de años, se puede observar que el tiempo requerido para esta factorización sería unas 315155 veces dicha edad lo que puede hacer pensar a primera vista que dicha factorización es inalcanzable en el futuro previsible. Pero hay que ser especialmente cauto a la hora de extrapolar para predecir factorizaciones futuras pues la historia demuestra que, en general, las predicciones que se han hecho han sido ampliamente superadas en la práctica y ello a pesar de que sigue sin haber algoritmos de factorización realmente eficientes. Por ejemplo, W. S. Jevons conjeturó en 1874 que nunca nadie llegaría a conocer los factores del entero 8616460799, que él mismo había construido multiplicando dos primos [34]. Hoy en día, si usamos *Mathematica* podemos obtener algo así:

```
FactorInteger[8616460799] // Timing

{0.06 Second, {{89681, 1}, {96079, 1}}}
```

lo que muestra que los dos factores han sido obtenidos en un tiempo de CPU de 6 centésimas de segundo. Pero no hubo que esperar a la llegada de los ordenadores pues este número ya fue factorizado en 1925 por B. Brown. Mucho más recientemente, R. Guy manifestó, en 1976, que *"me sorprendería si alguien consigue factorizar regularmente números del tamaño de* 10^{80} *durante el presente siglo"*, algo que ocurrió ya a mediados de los años 80 [34].

Una de las razones que han contribuido de manera decisiva (y sigue haciéndolo) a estos espectaculares avances, es la ley de Moore y otro factor importante es la cada vez mayor interconexión entre los ordenadores existentes (a través de Internet) que, junto con el desarrollo de métodos de factorización que, como NFS, son susceptibles de implementación en paralelo, permite realizar factorizaciones como las ya indicadas mediante la colaboración de ordenadores de todo el mundo.

Por otra parte, hay que tener en cuenta la posibilidad de que alguien pudiera descubrir un método de factorización eficiente sin hacerlo público, lo que le permitiría descifrar fácilmente toda la información encriptada con RSA. Aunque ello parece poco probable en la actualidad, no hay que olvidar que hay organizaciones poderosas para las cuales esto sería sin duda un objetivo muy deseable. El ejemplo mejor es la norteamericana NSA (National Security Agency) que se encarga de todo lo relacionado con las comunicaciones y la criptología y dedica enormes recursos a desencriptar todo tipo de información cifrada. La existencia de la NSA fue mantenida oficialmente en secreto durante muchos años, hasta el punto de que, en los círculos que estaban al tanto de su existencia se la denominaba, haciendo un juego de palabras con sus iniciales, "No Such Agency" ("No hay tal agencia"). Actualmente, la NSA es la organización de todo el mundo que emplea a un mayor número de matemáticos, superando a cualquier Universidad o Centro de Investigación y seguramente Lenstra y Verheul estaban pensando en ella cuando escribieron en [29] refiriéndose a RSA155: *" ... una clave de RSA de* 512 *bits fue factorizada en*

agosto de 1999. Aunque este resultado es la primera factorización publicada de un módulo RSA de 512 *bits, sería ingenuo creer que esta es la primera vez que una tal factorización ha sido obtenida."*

A pesar de todo se cree que, durante algún tiempo, un módulo de RSA de 1024 bits será seguro (esta es la razón de haber tomado primos de 512 bits en nuestros ejemplos: el producto de dos de ellos produce un módulo de 1024 bits). Si se requiere una seguridad muy alta y que pueda ser sostenible durante bastante tiempo, se recomienda usar módulos de, al menos, 2048 bits [9, 29, 34]. Por eso, en la versión actual del "concurso de factorización RSA" ("The New RSA Factoring Challenge", [41]), se establecen premios que van desde $10000 por la factorización de un módulo RSA de 576 bits (que ya superaría el record actual y parece inminente), hasta los $200000 ofrecidos por la factorización de un módulo de 2048 bits, que aun podría tardar bastantes años.

Pero la mayor amenaza a RSA no proviene del aumento de la potencia del hardware, sino de la posibilidad de que se descubra un algoritmo de factorización eficiente. De hecho, un tal algoritmo ya existe aunque es de un tipo especial que no se puede ejecutar en un ordenador digital clásico sino que requiere un *ordenador cuántico*. Tanto en [9] como en [29] se advierte explícitamente que las estimaciones allí contenidas sólo son válidas en la hipótesis de que en el tiempo a que se refieren no llegue a construirse un ordenador cuántico.

Los ordenadores cuánticos son máquinas que usan los principios de la mecánica cuántica para sus operaciones básicas. Esta frase aparentemente inocua esconde en realidad una idea auténticamente revolucionaria: que las computaciones pueden ser consideradas como procesos físicos, idea que fue propuesta a principios de los ochenta por Benioff, Feynman y Deutsch, entre otros (cf. [45, p. 230]). Sin entrar en detalles, mencionaré simplemente que el uso de la "superposición cuántica" permite, utilizando un "registro cuántico" formado por n *qubits* o *bits cuánticos*, ejecutar en un único paso computacional la misma operación matemática en 2^n números. Para poder ejecutar la misma computación, un ordenador clásico tendría que repetirla 2^n veces (o usar 2^n procesadores en paralelo), de modo que el ordenador cuántico ofrece una ventaja exponencial en los casos en que aplicar un "algoritmo cuántico" sea posible [16, 17, 45]. Uno de estos casos es precisamente el problema de la factorización de enteros según demostró Peter Shor en [42]: *un ordenador cuántico puede realizar la factorización en tiempo polinómico.*

Como consecuencia del algoritmo de Shor, es claro que si un ordenador cuántico adecuado pudiese ser construido, ello significaría el fin del criptosistema RSA. Sin embargo, no existe unanimidad sobre si un día los ordenadores cuánticos llegarán a ser una realidad. Aunque parece que no hay nada en las leyes de la física que impida su construcción, y de hecho ya se hacen experimentos con ordenadores cuánticos de unos pocos qubits, habrá que resolver problemas muy importantes antes de poder construir un ordenador cuántico capaz de ejecutar el algoritmo de Shor sobre un entero de los usados habitualmente en RSA. Por otra parte, la física cuántica no sólo podría ayudar a los criptoanalistas proporcionando el método para romper RSA, sino también a

los criptográfos. En efecto, utilizando el hecho de que *no es posible escuchar pasivamente un canal cuántico*, se han descrito varios protocolos cuánticos para el intercambio de claves (el primero de ellos, por Bennet y Brassard en [6]) que proporcionarían seguridad absoluta, pues la distribución cuántica de claves sería segura aun en caso de ser $P = NP$ o de que el criptoanalista dispusiera de un ordenador cuántico. Tampoco está del todo claro que este tipo de "criptografía cuántica" llegará un día a ser una realidad práctica pero, en principio, parece más fácil de conseguir que un ordenador cuántico.

Bibliografía

[1] Adleman, L. M., Huang, M. D.: *Primality Testing and Abelian Varieties over Finite Fields*, Lecture Notes in Math. 1512, Springer-Verlag, Berlin (1992).

[2] Adleman, L. M., Pomerance, C., Rumely, R. S.: *On distinguishing prime numbers from composite numbers,* Annals of Math. 117 (1983), 173-206.

[3] Arnault, F.: *Sur quelques tests probabilistes de primalité,* Université de Poitiers (1993).

[4] Atkin, A. O. L., Morain, F.: *Elliptic curves and primality proving,* Math. Comp. 61 (1993), 29-68.

[5] Bach, E., Shallit, J.: *Algorithmic Number Theory, Vol. 1*, The MIT Press, Cambridge (1996).

[6] Bennet, C. H., Brassard, G.: *Quantum cryptography: Public Key Distribution and Coin Tossing,* Proceedings of IEEE International Conference on Computer Systems and Signal Processing, Bangalore (1984), 175-179.

[7] Bleichenbacher, D.: *Efficiency and Security of Cryptosystems based on Number Theory,* Dissertation ETH No. 11404, Zürich (1996).

[8] Borges, J. L.: *Funes el memorioso,* en *Ficciones*, Alianza Emecé, Madrid (1971).

[9] Brent, R. P.: *Some parallel algorithms for integer factorisation,* preprint (1999).

[10] Bressoud, D., Wagon, S.: *A course in computational number theory,* Key College Publishing, Emeryville (2000).

[11] Burthe, R. J.: *Further investigations with the strong probable prime test,* Math. Comp. 65 (1996), 373-381.

[12] Cohen, H.: *A Course in Computational Algebraic Number Theory,* Springer-Verlag, Berlin (1993).

[13] Cohen, H., Lenstra, H. W.: *Primality testing and Jacobi sums,* Math. Comp. 42 (1984), 297-330.

[14] Cohen, H., Lenstra, A. K.: *Implementation of a new primality test,* Math. Comp. 48 (1987), 103-121.

[15] Cook, S.: *The P versus NP problem,* `http://www.claymath.org`.

[16] Crandall, R., Pomerance, C.: *Prime numbers. A computational perspective,* Springer-Verlag, New York (2001).

[17] Deutsch, D., Ekert, A.: *Quantum Computation,* Physics World 11 (3) (1998), 47-52.

[18] Factor World: `http://www.crypto-world.com/FactorWorld.html`.

[19] Gimps: `http://www.mersenne.org`.

[20] Goldwasser, S., Kilian, J.: *Almost all primes can be quickly certified,* Proc. 18th STOC (Berkeley, 1986), 316-329.

[21] Gómez Pardo, J. L.: *Aspectos computacionales de los números primos (I),* La Gaceta de la RSME, 4 (3) (2001), 649-673.

[22] Grantham, J.: `http://www.pseudoprime.com/pseudo.html`.

[23] Hankerson, D. G. et al.: *Coding Theory and Cryptography. The Essentials,* Marcel Dekker, New York (2000).

[24] Knuth, D. E.: *The Art of Computer Programming, Volume 2: Seminumerical Algorithms,* 2nd Edition, Addison-Wesley (1981).

[25] Koblitz, N.: *A Course in Number Theory and Cryptography,* 2nd. Ed, Springer-Verlag, New York (1994).

[26] Koblitz, N.: *Algebraic aspects of cryptography,* Springer-Verlag, New York (1998).

[27] Lenstra, A. K., Lenstra, H. W.: *Algorithms in Number Theory*, en *Handbook of theoretical computer science, Vol. A,* 673–715, Elsevier, Amsterdam (1990).

[28] Lenstra, A. K., Lenstra, H. W. (Eds.) : *The development of the number field sieve*, Lect. Notes in Math. 1554, Springer-Verlag, Berlin (1993).

[29] Lenstra, A. K., Verheul, E.: *Selecting cryptographic keysizes*, J. Cryptology 14 (2001), 255–293.

[30] Maoz, E. et al.: *A distance to the galaxy NGC4258 from observations of Cepheid variable stars*, Nature 401 (1999), 351 - 354.

[31] Martin, M.: `http://www.znz.freesurf.fr`.

[32] Mihăilescu, P.: *Recent developments in primality proving,* preprint.

[33] Noll, L. C.: `http://www.isthe.com/chongo/tech/math/prime/mersenne.html`.

[34] Odlyzko, A. M.: *The future of integer factorization,* Cryptobytes 1 (1995), 5-12.

[35] Pomerance, C.: *A tale of two sieves,* Notices of the Amer. Math. Soc. 43 (1996), 1473-1485.

[36] Pomerance, C., Selfridge, S., Wagstaff, Jr. S.: *The pseudoprimes to* $25 \cdot 10^9$*,* Math. Comp. 35 (1980), 1003-1026.

[37] Primenet: `http://www.mersenne.org/Primenet`.

[38] Ribenboim, P.: *The New Book of Prime Number Records,* Springer-Verlag, New York (1996).

[39] Riesel, H.: *Prime numbers and computer methods for factorization, 2nd Ed.,* Birkhäuser, Boston (1994).

[40] Rivest, R. L., Silverman, R. D.: *Are 'Strong' primes needed for RSA?*, preprint (1998).

[41] Rsa labs: `http://www.rsasecurity.com /rsalabs/challenges/factoring/index.html`.

[42] Shor, P. W.: *Polynomial time algorithms for prime factorization and discrete logarithms on a quantum computer,* SIAM J. Computing 26 (1997), 1484-1509.

[43] Singh, S.: *The Code Book,* Fourth Estate, London (1999).

[44] van tilborg, H. C. A.: *Fundamentals of Cryptology,* Kluwer, Dordrecht (2000).

[45] Yan, S. Y.: *Number theory for computing,* Springer-Verlag, Berlin (2000).

José Luis Gómez Pardo
Departamento de Álxebra, Universidade de Santiago
15782 Santiago de Compostela
correo electrónico: `pardo@usc.es`