Test de primalidad, el AKS

Cruz Enrique Borges Hernández

7 de julio de 2005

Agradecimientos.

Antes de comenzar el texto me gustaría agradecer a mis compañeros de clase, en especial a Rodrigo, por aguantar mis discursos y discusiones varias sobre los temas más absurdos imaginables y por sus sugerencias tanto a la hora de hacer el trabajo como en los quehaceres diarios del estudiante.

También deseo darle las gracias a los compañeros "virtuales" del foro [76] por sus locuras y paranoias varias, pero sobre todo por volcarse cuando pedí ayuda a la hora de recopilar los datos para construir la gráfica 1. Así como también darle las gracias a Javi, becario del departamento de Matemáticas, Estadística y Computación, que me permitió usar todos los ordenadores del aula del departamento y me puso todas las facilidades que pudo.

En este punto considero importante darle las **NO** gracias al servicio de informática de la Universidad de Cantabria, por las dificultades y trabas, al alumnado, para el uso de los todas las recursos y herramientas que la red nos pone a disposición. Especialmente en el Colegio Mayor Juan de la Cosa (aunque en las salas de informática suceda algo parecido) lo que ha motivado que la realización de este trabajo sea muchísimo más complicada de lo que debiera haber sido (aunque en el fondo deberíamos de estar agradecidos pues "al menos tenemos Internet"). Y desde aquí también me gustaría mostrar mi rechazo al cobro por la utilización de los recursos informáticos y bibliográficos que se está comenzando a sugerir o en el caso de la biblioteca, a cobrar.

Para finalizar darle las gracias a Luis por facilitarme muchísimo material para realizar el trabajo y por aguantar mis depresiones cuando el trabajo no terminaba de coger forma. Y también a toda la comunidad de desarrolladores de herramientas GNU [77], en especial a los desarrolladores de MiKTeX [78], TeXnicCenter [79], Firefox [80] y Dev-C++ [59] que han sido las herramientas usadas principalmente para la realización de este trabajo.

Introducción.

Mi objetivo principal en este trabajo ha sido realizar un recorrido histórico por las llamadas *clases centrales* de la teoría de complejidad algorítmica comparando sus distintas propiedades tanto a nivel teórico como práctico. Como excusa, o como pretexto, usaremos el bien conocido ejemplo (y realmente muy didáctico) de los *test de primalidad*.

Comenzaremos realizando un recorrido histórico, bastante completo, sobre los distintos algoritmos usados para comprobar la primalidad. Empezaremos por los algoritmos griegos y árabes en el apartado 1.1, lo cual nos conduce a introducir la clase **E**.

En el siguiente apartado trataremos los certificados de primalidad y composición clásicos, hablamos del *certificado de Pratt* y el *test indeterminista de compositud*. En este apartado nos introduciremos en el indeterminismo y la clase **NP**. Aunque no esté relacionado directamente con dicha clase, también trataremos en este apartado otros algoritmos importantes y didácticos. Por ejemplo, el *test de Willson* nos ilustra un algoritmo que a primera vista parece eficiente, pero del que actualmente no se sabe si lo es o no. Sin embargo, el *polinomio de Matijasevic* nos muestra algo completamente distinto, como caracterizar el conjunto de los números primos mediante una fórmula (con lo cual se demuestra que es un conjunto *diofántico*).

En el apartado 1.3 entraremos de lleno en el siglo XX y la algorítmica moderna. Con el *algoritmo de Solovay-Strassen* introduciremos el importantísimo concepto de los *algoritmos aleatorios* y la clase **Co-RP**. Hablaremos también del *algoritmo de Miller-Rabin*, hoy por hoy, el mejor y más rápido test de primalidad (tanto es así que es el test estándar en prácticamente todos los paquetes informáticos).

Posteriormente nos meteremos con los test en la clase **RP** basados en curvas elípticas. Hablamos del *ECPP* y del *APRCL*. Para finalizar este apartado introduciremos el reciente test *AKS* y con él volvemos al determinismo y entramos en la clase **P**.

A este algoritmo dedicaremos íntegramente la sección 2. Comentaremos pormenorizadamente la demostración de este algoritmo, su análisis de complejidad y las posibles mejoras que se han planteado.

Para terminar esta primera parte, comentaremos en la sección 3 las implicaciones que trae el algoritmo en sí mismo, sobre todo lo referente a la seguridad informática, criptografía y el criptosistema *RSA*. También hablaremos, someramente, sobre las

nuevas vías de investigación en *conjuntos cuestores* y *test de nulidad* que ha abierto el algoritmo.

En la segunda parte de trabajo comentaremos las diversas implementaciones del algoritmo que hemos desarrollado. Explicaremos con detalle los objetivos perseguidos y las dificultadas presentadas a lo largo de los diversos experimentos llevados a cabo. Para finalizar presentaremos un completo informe con las conclusiones obtenidas.

Índice

1. El camino hasta P

Tenemos la certeza que desde los tiempo de Pitágoras los números primos han sido objeto de estudio, pero con total seguridad, algunas de sus más elementales propiedades ya eran conocidas desde mucho más antiguo. Probablemente las primeras en descubrirlas fueron las mujeres del Neolítico, que tras recolectar alimentos para el grupo debieron enfrentarse al problema de repartirlo. Si la casualidad, o la mala suerte, aparecía y la recolecta constaba de un *número primo* de elementos, rápidamente se habrán dado cuenta de la imposibilitad de repartirlo de forma equitativa, o sea de hallar un divisor.

Otro ejemplo de la aparición de los números primos se presenta en el ciclo vital[1] de las cigarras *Magicicada septendecim* que es de 17 años. Un número muy alto comparado con los demás miembros de su familia, aparte de ser un número primo. ¿Por qué la evolución ha hecho que este tipo de cigarra tenga un ciclo vital tan largo y primo? La respuesta más plausible es la siguiente:

Supongamos que la cigarra tiene un parásito que tiene por ciclo vital n años. A la cigarra le interesará evitar coincidir, en su fase adulta, con este parásito, así que la evolución seleccionará aquellas cigarras cuyos ciclos vitales sean números primo. Ahora bien, el parásito tiene aún una oportunidad de sobrevivir, o adquiere un ciclo de un año o adquiere el mismo ciclo primo. Sin embargo para poder adaptarse a estos cambios el parásito habrá tenido que sobrevivir varias generaciones sin alimento, cosa altamente improbable, con lo cual se extinguirá. . . o cambiará de alimentación. Este hecho se avala en la certeza de no haberse encontrado nunca un parásito de esta especie de cigarra. Una explicación más detallada se puede encontrar en [7].

A lo largo del texto veremos los distintos métodos que han surgido a lo largo de la historia para certificar que un número es ciertamente primo. A este problema comúnmente se le denomina *test de primalidad* o simplemente **PRIMES** y es un ejemplo muy instructivo sobre como un mismo problema a ido cambiando su complejidad algorítmica a lo largo de los años.

El interés que han suscitado los números primos ha sido muy variado. Desde el interés puramente "espiritual" de los griegos (que los consideraban "bello") al que suscitaba durante el siglo XVIII-XIX como problema matemático duro. Podemos observar este hecho en las siguientes citas:

[1]Periodo entre el nacimiento y el apareamiento. Por norma general, tras el apareamiento y la puesta de huevos este tipo de insectos muere.

> Mathematicians have tried in vain to this day to discover some order in the sequence of prime numbers, and we have reason to believe that it is a mystery into which the human mind will never penetrate[2]. *Leonhard Euler.* (1707 - 1783)

> Problema, numeros primos a compositis dignoscendi, hosque in factores suos primos resolvendi, ad gravissima ac utilissima totius arithmeticae pertinere [...] tam notum est, ut de hac re copiose loqui superfluum foret. [...] Praeteraeque scientiae dignitas requirere videtur, ut omnia subsidia ad solutionem problematis tan elegantis ac celebris sedulo excolantur[3]. *Carl Friedrich Gauss.* (1777 - 1855)

> God may not play dice with the universe, but something strange is going on with the prime numbers[4]. *Paul Erdos.* (1913 - 1996)

Actualmente, el interés es debido a la criptografía. Los métodos criptográficos actuales usan números primos de muchas cifras decimales (llamados *primos industriales*) como parte fundamental del proceso de encriptación. El mayor problema es que la seguridad del método se diluye cuando elegimos un número que creemos es primo cuando sin embargo no lo es. Por lo que es fundamental tener algoritmos *rápidos y eficientes* que certifiquen la primalidad.

Vamos a hablar un poco sobre lo que entendemos aquí por *rapidez y eficiencia* de una forma muy simplificada e imprecisa[5]. Ambos términos están relacionados con la complejidad de un algoritmo, entendida la complejidad como el número de operaciones que realiza una máquina de Turing, "programada" con este algoritmo, hasta dar una respuesta. Esto es lo que entendemos por *rapidez* o complejidad en tiempo. O bien podemos considerar el tamaño máximo que toman las cintas de trabajo (memoria) que usa dicha máquina a lo largo de la ejecución del algoritmo. Es lo que denominamos complejidad en espacio o *eficiencia*.

[2]Los matemáticos han intentado, en vano hasta el momento, descubrir algún orden en la secuencia de los números primos y tenemos razones para creer que es un misterio en el cual la mente humana nunca podrá penetrar.

[3]El problema de distinguir un número primo de otro compuesto y descomponer éstos últimos en sus factores primos, es uno de los problemas mejor conocidos por su importancia y utilidad en aritmética, [...] sobra comentar la extensión de esta materia. [...] La dignidad de la ciencia requiere que todas las posibilidades sean exploradas para encontrar una solución elegante y celebrada al problema.

[4]Dios puede que no juegue a los dados con el universo, pero algo raro está haciendo con los números primos.

[5]Para una descripción detallada de las clases de complejidad mirar [2] o [5]

Es claro que la complejidad variará en función de la entrada que tenga el algoritmo. Intuitivamente se ve que a mayor entrada, es de esperar un mayor número de operaciones y/o memoria. En función de las *cotas* que tengamos sobre las funciones de complejidad podemos clasificar los algoritmos en distintas clases, como por ejemplo: **P**, la clase de los algoritmos que tienen como cota una función polinomial o **NP** la clase de los algoritmos indeterministas acotados por una función polinomial.

1.1. PRIMES está en E

Ahora bien, pero... ¿qué es un número primo? La definición clásica o escolar que todos conocemos es algo así:

Definición 1.1 (Número primo)**.**

Sea $n \in \mathbb{N}$. n es primo sí y sólo sí los únicos divisores que posee son 1 y n.

Esta definición se generaliza para los dominios de integridad, lo cual da lugar al concepto de *elemento primo*.

Definición 1.2 (Elemento primo)**.**

Sea D un dominio de integridad y sea $n \in D$. n es un elemento primo si y solo si:

a) $n \neq 0$

b) n no es una unidad.

c) Si p divide a ab con $a, b \in D$ entonces $p|a$ o bien $p|b$.

Nos enfrentamos ahora al problema en cuestión, ¿cómo certificar que un número es primo? Si se le pregunta a un escolar lo más probable es que te conteste que miremos la tabla de número primos que tiene en su libro de textos (si es muy vago... o muy listo). O bien podría contestar que probemos a dividir por todos los números menores que él (ahora bien, si es realmente listo dirá que solo hace falta hasta su raíz cuadrada).

Estas ideas simples fueron realmente los primeros test de primalidad. El primero es conocido como la *criba de Eratóstenes* y es debido a Eratóstenes (siglo II a.C.)[6].

[6]Cuesta creer que Euclides, que muchos años atrás había enunciado, y demostrado, el famoso

Algoritmo 1.3 (Criba de Eratóstenes)**.**

Input $= n \in \mathbb{Z}$

Paso*. 1 Escribir todos los números hasta* n*.*

Paso*. 2 Tachar el* 1 *pues es una unidad.*

Paso*. 3 Repetir hasta* n

- *Tachar los múltiplos del siguiente número sin tachar, excepto el mismo número.*

Output = Los números tachados son compuestos y los no tachados son primos.

Este algoritmo tiene la particularidad de que no sólo nos certifica si un número es primo o no, sino que nos da todos los primos menores que él. El algoritmo funciona pues estamos comprobando si es o no múltiplo de algún número menor que él, lo cual es equivalente a la definición de numero primo. Este método es óptimo cuando necesitamos construir la tabla con todos los números primos hasta n, pero es tremendamente ineficiente para nuestro propósito, pues se trata de un algoritmo exponencial tanto en tiempo como en espacio.

Sorprendentemente, no se tiene constancia del *método trivial* hasta el siglo XII cuando Leonardo de Pisa (más conocido como Fibonacci) introduce el método de las divisiones sucesivas, esto es, aplicar la definición tal cual. El algoritmo sería:

Algoritmo 1.4 (Divisiones Sucesivas)**.**

Input $= n \in \mathbb{Z}$

Paso*. 1 Mientras* $i < \sqrt{n}$

- *Si* $n = 0 \mod i$ *entonces Output = Compuesto*

Paso*. 2 Output = Primo*

teorema de la infinitud de los números primos, no hubiera construido una tabla como ésta usando el mismo método. Esto me hace suponer que es altamente probable que le estemos atribuyendo el mérito del algoritmo simplemente por ser la primera persona de la que tenemos constancia escrita de usarlo.

Observar que no es necesario comprobar hasta $n-1$, pues de tener un divisor mayor que $\sqrt{n}$ tendría otro más pequeño que ya habríamos comprobado. Esto también es aplicable a la Criba de Eratóstenes y es una modificación propuesta por ibn al-Banna, matemático marroquí contemporáneo de Leonardo de Pisa.

Este algoritmo es también exponencial en tiempo (no así en espacio). La importancia de estos algoritmos no radica en su *eficiencia* sino por ser ejemplos muy didácticos de lo que son los algoritmos de primalidad. Mención a parte, claro está, del interés histórico.

1.2. PRIMES está en NP

Hasta ahora los algoritmos que hemos expuesto son exponenciales, por lo cual el problema de decidir cuando un número es primo o compuesto pertenece a la clase **E**. Esto no cambió hasta bien entrado el siglo XVII después de que Pierre de Fermat enunciara el bien conocido *Teorema pequeño de Fermat* y tampoco es que se avanzara mucho en la creación de algoritmos, de ahí las citas de los grandes matemáticos de la época vistas en el capítulo anterior. No obstante, en el desarrollo teórico, no sucedió lo mismo pues muchos de los test actuales se apoyan, en gran medida, en teoremas demostrados durante esta época. Enunciemos este importantísimo teorema:

Teorema 1.5 (Pequeño de Fermat)**.**

$$\text{Si } p \text{ es un número primo, entonces } \forall\ a \in \mathbb{N},\ a^p = a \bmod p$$

Una demostración de este teorema se puede encontrar en cualquier libro de álgebra básica medianamente decente o en [6].

Desgraciadamente este teorema no es una equivalencia, existen números que cumplen el teorema y sin embargo no son primos, los llamados *números de Carmichael*[7]. Aunque la cantidad de números de Carmichael fuera finita, la propia existencia de estos números ya es de por si un problema, pero recientemente se ha demostrado [35] que existen infinitos, lo cual dificulta aún más la creación de algoritmos mediante el uso directo del teorema.

[7]Más adelante hablaremos sobre ellos, sin embargo se puede encontrar más información en [34] e incluso una lista de estos números en [33]. Dicha lista la usaremos en la implementación en el capítulo 4.

Sin embargo, añadiendo algunas hipótesis, el recíproco es cierto. Este es el llamado *Recíproco del Teorema pequeño de Fermat* o *Teorema de Lehmer*.

Teorema 1.6 (Recíproco del Teorema pequeño de Fermat)**.**

$n \in \mathbb{N}$ es primo si y solamente si $\mathbb{Z}_n^*$ es un grupo cíclico de orden $n-1$.

En particular, si $n \in \mathbb{N}$ es primo entonces $\exists\ x \in \{1, \ldots, n-1\}$ tal que:

a) $(x, n) = 1$

b) $x^{n-1} = 1\ mod\ n$

c) $\nexists\ e \in \mathbb{Z}$ *con* $e < n-1$ *tal que* $x^e = 1\ mod\ n$

A todo número verificando las propiedades anteriores se le denomina *testigo* de n.

Demostración. A partir de las hipótesis se deduce trivialmente que $o_n(x) = n-1$ donde $o_n(x)$ denota el orden de x módulo n. Luego el grupo cíclico generado por x tiene exactamente $n-1$ elementos distintos con lo cual deducimos que n es primo. □

A partir de este teorema podemos dar un algoritmo indeterminista que certifica la primalidad [22] y [23].

Algoritmo 1.7 (Certificado de Pratt)**.**

Input $= n \in \mathbb{Z}$

Paso*. 1 Guess*[8] *x testigo de n.*

Paso*. 2 Guess L =*{ *"Lista con los divisores de* $n-1$*"*}

Paso*. 3 Si* $\forall y \in L\ x^{\frac{n-1}{y}} \neq 1\ mod\ n$ *y* $x^n = 1\ mod\ n$ *entonces Output = Primo.*

En otro caso Output = Compuesto.

El algoritmo es recursivo pues se basa en adivinar (*guessing*) los factores primos de $n-1$, para ello, se llama recursivamente sobre la lista de factores hallada en el paso 1. La correctitud está garantizada ya que para probar que $o_n(x) = n-1$ sólo es necesarios comprobarlo para los divisores no primos de $n-1$, que mediante

[8] Usaremos esta notación para la parte no determinista del algoritmo.

iterated squaring[9] se puede realizar en tiempo $O(\log(n))$. Conociendo que el número de factores primos es del orden de $O(\log(n))$ resulta ser un algoritmo polinomial indeterminista.

Veamos ahora un teorema muy básico (y su correspondiente algoritmo) que incluyen este problema en la clase **Co-NP**. Esto hace que **PRIMES** pertenezca a la clase **NP** $\bigcap$ **Co-NP**.

Teorema 1.8.

PRIMES $\in$ **Co-NP**.

Demostración. La demostración es bien simple. El algoritmo complementario a **PRIMES** es decidir si un número es *compuesto*. Esto es, si tiene algún divisor que no sea 1 o el mismo número. Luego, dado un divisor, comprobar que efectivamente lo es, se realiza en tiempo polinomial. Con lo cual tenemos un algoritmo indeterminista en tiempo polinomial para el algoritmo complementario. Esto es, **PRIMES** $\in$ **Co-NP**. □

El algoritmo que surge de este teorema es francamente trivial. Quedaría:

Algoritmo 1.9 (Test indeterminista de compositud)**.**

Input $= n \in \mathbb{Z}$

***Paso**. 1 Guess* d divisor de n.

***Paso**. 2* Si $n = 0 \bmod d$ entonces Output = Compuesto
en otro caso Output = Primo

Ahora hagamos un inciso para ver un teorema bastante curioso y que a primera vista parece la panacea y algo un poquito más "esotérico".

Teorema 1.10 (Wilson)**.**

$$\forall\ n \in \mathbb{Z},\ n > 1.\ n \text{ es primo si y solo si } (n-1)! = -1 \bmod n.$$

Demostración. "$\Leftarrow$"

El teorema es trivial para $n = 2, 3$ luego supongamos que $n > 3$.

[9]Este método también es conocido como *adition chain* y puede encontrarse una explicación completa en [4] o en [8].

Si n es un número compuesto $\exists\ a, b \in \{1, \ldots, n-1\}$ tales que $n = ab$ con lo cual n divide a $(n-1)!$ y se cumple que $(n-1)! + 1 = 1 \neq 0 \bmod n$.

"$\Rightarrow$"

Si por el contrario n es primo se tiene que $\mathbb{Z}_n$ es un cuerpo y por lo tanto $\forall\ x \in \mathbb{Z}_n \setminus \{0\}\ \exists |\ x^{-1}$ tal que $xx^{-1} = 1 \bmod n$. Además $x^{-1} \neq x \Leftrightarrow x = 1, p-1$ con lo cual podemos formar parejas "número-opuesto" quedando de la siguiente forma:

$$2 \cdot 3 \cdot \ldots \cdot (p-2) = (p-2)! = 1 \bmod n \Rightarrow (p-1)! = p - 1 = -1 \bmod n$$

□

El algoritmo que surge directamente de teorema es el siguiente:

Algoritmo 1.11 (Test de Wilson)**.**

Input $= n \in \mathbb{Z}$

Paso*. 1 Si* $(n-1)! + 1 = 0$ *mod* n *entonces Output = Primo*

en otro caso Output = Compuesto

Claramente se ve que es un algoritmo determinista que decide la primalidad de un elemento. Entonces... ¿dónde está la trampa? El problema estriba en el cálculo de $(n-1)!$ Actualmente nadie conoce la forma de realizarlo eficientemente. Ni siquiera el caso modular $(n-1)!$ mod n que evitaría el crecimiento de los resultados intermedios.

Cambiamos ahora un poco el contexto. Ahora en vez de buscar un algoritmo que nos certifique si un número es o no primo buscamos una función que nos genere números primos. Euler ya conocía que el polinomio $x^2 + x + 41$ devuelve números primos para $x = 0, 1, \ldots, 39$. La pregunta que se hicieron los matemáticos fue: ¿existe una función la cual al evaluarla en los números enteros devolviera los números primos? Para desgracia de muchos la respuesta vino en forma de teorema:

Teorema 1.12. *Sea* $p(x_1, \ldots, x_n) \in \mathbb{C}[x_1, \ldots, x_n]$ *un polinomio multivariado con coeficientes complejos.*

Si p *verifica que* $\forall\ z \in \mathbb{N}^n,\ p(z)$ *es un número primo, entonces,* p *es constante (i.e.* $p \in \mathbb{C}$*).*

La demostración de este teorema, así como información sobre la historia de su desarrollo se puede encontrar en [27] o en [28].

Tras este pequeño traspiés la investigación no decayó. Se relajó un poco las condiciones quedando ahora ¿existe un polinomio cuyos valores enteros positivos sea el conjunto de todos los números primos? La respuesta a esta pregunta la trajo Matijasevic [24].

Teorema 1.13 (Polinomio de Matijasevic)**.** *El siguiente polinomio* $p \in \mathbb{Z}[a, \ldots, z]$*:*

$$\begin{aligned}
&(k+2)\{1-[wz+h+j-q]^2-[(gk+2g+k+1)(h+j)+h-z]^2- \\
&-[2n+p+q+z-e]^2-[16(k+1)^3(k+2)(n+1)^2+1-f^2]^2- \\
&-[e^3(e+2)(a+1)^2+1-o^2]^2-[(a^2-1)y^2+1-x^2]^2- \\
&-[16r^2y^4(a^2-1)+1-u^2]^2- \\
&-[((a+u^2(u^2-a))^2-1)(n+4dy)^2+1-(x+cu)^2]^2- \\
&-[n+l+v-y]^2-[(a^2-1)l^2+1-m^2]^2- \\
&-[ai+k+1-l-i]^2- \\
&-[p+l(a-n-1)+b(2an+2a-n^2-2n-2)-m]^2- \\
&-[q+y(a-p-1)+s(2ap+2a-p^2-2p-2)-x]^2- \\
&-[z+pl(a-p)+t(2ap-p^2-1)-pm]^2\}
\end{aligned}$$

verifica:

$$\{Y = p(a, \ldots, z) \in \mathbb{Z} : (a, \ldots, z) \in \mathbb{Z}^{26}, Y \geq 0\} = \{n \in \mathbb{N} : n \text{ es primo}\}.$$

En otras palabras, los valores positivos de la imagen por p de los valores enteros son exactamente los números primos positivos.

Es un polinomio de 26 variables y grado 25 bastante poco manejable. Lo revolucionario de este teorema no es el hecho de que nos devuelva números primos, sino el hecho de poder expresar, mediante un polinomio, una secuencia de números recursivamente enumerable.

Anteriormente a los resultados de Matijasevic [29] este problema fue estudiado por Julia Bowman Robinson a lo largo de su brillante carrera [30] y condujo a la resolución del X problema de Hilbert [31].

1.3. PRIMES está en Co-RP

Dejamos ahora a un lado los algoritmos indeterministas y las curiosidades y veamos ahora el desarrollo de los algoritmos *modernos*[10]. Comenzaremos con una

[10]Es de destacar que hasta mitades del siglo XX no se produjo ningún avance, prácticamente.

idea, bien simple, que nos servirá de ejemplo.

Habíamos visto en el capítulo anterior que, por el teorema pequeño de Fermat, $\forall\ a \in \mathbb{Z},\ a^p = a \bmod p$. Basándonos en esto podríamos construir el siguiente algoritmo:

Algoritmo 1.14 (Test bruto de Fermat)**.**

Input $= n \in \mathbb{Z}$

Paso*. 1 Desde $a = 2$ hasta $n - 1$ haz*

- *Si $a^{n-1} \neq 1$ mod n entonces Output = Compuesto.*

Paso*. 2 Output = Probable Primo.*

El algoritmo es exponencial en tiempo, pues realizamos n pasos cada uno de $O(\log^3(n))$ lo cual nos da un algoritmo de orden $O(n \log^3(n))$. Además, sabemos que cuando nos encontramos con un número de Carmichael, el algoritmo falla estrepitosamente. La definición de este conjunto de números es:

Definición 1.15 (Números de Carmichael)**.**

$n \in \mathbb{Z}$ es un número de Carmichael si y sólo si:

a) n es compuesto.

b) $\forall x \in \mathbb{Z}_n^*\ o_n(x)$ es un divisor propio de $n - 1$.

Una posible idea para mejorarlo podría ser:

Algoritmo 1.16 (Test de Fermat)**.**

Input $= n, k \in \mathbb{Z}$ /* $k =$ número máximo de repeticiones */

Paso*. 1 Sea $aux = i = 1$*

Paso*. 2 Mientras $(i \leq k)$ y $(aux = 1)$*

- *Sea aux un número escogido aleatoriamente en $\{1, \ldots, n-1\}$*
- *$aux = aux^{n-1}$ mod n*
- *$i = i + 1$*

Paso. *3 Si* $(aux = 1)$ *entonces Output = Probable Primo.*

en otro caso Output = Compuesto.

Ahora tenemos que sólo hacemos a lo más k repeticiones con lo cual el algoritmo pasa a ser $O(\log^3(n))$, a costa de poder equivocarnos en la respuesta. La probabilidad de error descenderá dependiendo del número de veces que repitamos el proceso, pero en el caso de toparnos con un número de Carmichael seguirá fallando. Veamos a que clase pertenece:

- Para toda entrada es polinomial en tiempo.
- Si el input es primo entonces siempre devuelve primo.
- Si el input es compuesto entonces puede fallar.

A primera vista puede parecer que pertenece a la clase **Co-RP**, pero debido a la existencia de los números de Carmichael no pertenece a dicha clase (la probabilidad de error no esta acotada en dichos números, siempre falla).

Este primer ejemplo nos sirve de introducción a lo que será lo común a lo largo de este capítulo, los algoritmos probabilistas. Este tipo de algoritmos son muy rápidos a costa de poder fallar "a veces".

Veamos ahora otro teorema que nos ve a permitir construir un algoritmo probabilista:

Teorema 1.17. *Sea* $n, a \in \mathbb{Z}$. n *es primo* $\Leftrightarrow \forall\ a \in \{2, \ldots, n-1\}$ *se tiene que:*

$$\left(\frac{a}{p}\right) = a^{\frac{p-1}{2}} \bmod p.$$

Donde $\left(\frac{a}{p}\right)$ *es el* Símbolo de Legendre*, que se puede calcular de la siguiente forma:*

- *0 si* p *divide a* a.
- *1 si* $\exists\ x$ *tal que* $x^2 = a\ mod\ n$.
- *-1 si* $\nexists\ x$ *tal que* $x^2 = a\ mod\ n$.

El teorema nos da una caracterización de los números primos relativamente fácil de calcular pero que es exponencial si queremos concluir primalidad. Sin embargo, en este caso no tenemos el handicap de los números de Carmichael y realizando unas "pocas" pruebas podemos garantizar la primalidad con una probabilidad acotada. Veamos como quedaría el algoritmo [37]:

Algoritmo 1.18 (Test de Solovay-Strassen)**.**

Input $= n, k \in \mathbb{Z}$ /* $k =$ número máximo de repeticiones */

Paso*. 1 Desde* $i = 1$ *hasta* k *haz*

- *Sea* a *un número escogido aleatoriamente en* $\{2, \ldots, n-1\}$
- $aux = \left(\frac{a}{n}\right)$
- *Si* $aux = 0$ *o* $a^{\frac{p-1}{2}} \; - \; aux \neq 0 \; mod \; n$ *entonces Output = Compuesto.*

Paso*. 2 Output = Probable Primo.*

Observamos que el algoritmo realiza $O(kq)$ operaciones, donde q es el coste de calcular el símbolo de Legendre. H. Cohen demuestra en [3] que el símbolo de Legendre se puede calcular en no más de $O(\log^2(m))$ con $m = \text{máx}\{a, n\}$ con lo cual da lugar a un algoritmo de orden $O(k \log^2(n)) = O(\log^2(n))$. Además, si definimos $A = \{a \in \mathbb{Z}_n^*$ tales que no se cumple el teorema cuando n no es primo$\}$ (*conjunto de testigos*) se puede ver que $|A| > |\mathbb{Z}_n^*|/2$.

Veamos a que clase de complejidad pertenece:

- Para toda entrada es polinomial en tiempo.
- Si el input es primo entonces siempre devuelve primo.
- Si el input es compuesto la probabilidad de darlo por primo es menor que $1/2^k$.

Con lo cual concluimos que este algoritmo pertenece a la clase **Co-RP** pues puede dar por primos números que en realidad son compuestos[11].

Este es el primer ejemplo que se propuso de algoritmo probabilista y se puede considerar a Solovay y Strassen como los padres de dichos algoritmos.

[11]Notar que este es un algoritmo **RP** para el problema recíproco, es decir, para decidir si un número es compuesto o no.

El mayor problema de este algoritmo radica en la dificultad para implementar el cálculo del símbolo de Legendre. Sin embargo años más tarde aparecería otra caracterización de los números primos que posibilitaría un test sustancialmente mejor que éste. La idea del test se la debemos a Miller y la posterior mejora a Rabin (que transformó el algoritmo en probabilista usando las ideas de Solovay y Strassen).

La idea detrás del algoritmo es la siguiente. Sabemos que si n es primo, $\mathbb{Z}_n$ es un cuerpo y por lo tanto $x^2 - 1$ tiene exactamente dos raíces, 1 y -1. Sin embargo esto no es cierto cuando n es compuesto, es más, se puede comprobar que podrá tomar tantos valores distintos como dos elevado al número de factores primos que posea. Ahora bien, por el Teorema pequeño de Fermat sabemos que $a^{n-1} = 1 \bmod n$, con lo cual $a^{\frac{n-1}{2}}$ es una raíz cuadrada de la unidad y por lo tanto sólo puede ser 1 o -1. Repitiendo el proceso mientras tengamos factores pares, obtenemos una serie de valores que puede ser o bien $\{1, 1, \ldots, 1\}$ o bien $\{1, 1, \ldots, -1\}$. A los números con esta propiedad los podemos llamar *testigos* y se puede probar que más de la mitad de los enteros entre 1 y n cumplen dicha propiedad. Con lo cual probando con $a = 1, 2, \ldots, n/2$ averiguaríamos si efectivamente n es primo o compuesto.

A partir de este idea surge el siguiente algoritmo determinista que certifica primalidad [42].

Algoritmo 1.19 (Test determinista de Miller)**.**

Input $= n \in \mathbb{Z}$

Paso*. 1 Escribir $n - 1 = 2^s d$ dividiendo sucesivas veces $n - 1$ entre 2.*

Paso*. 2 Desde $a = 2$ hasta $\lceil 2\log^2(n) \rceil$ haz*

- *Si $a^d \neq 1$ mod n entonces Output = Compuesto.*
- *Desde $r = 0$ hasta $s - 1$ haz*
  - *Si $(a^{2^r d}) \neq -1$ mod n entonces Output = Compuesto.*

Paso*. 3 entonces Output = Primo.*

La correctitud de este algoritmo está garantizada por la caracterización dada anteriormente de los números primos, salvo por el pequeño detalle: *sólo* comprobamos los primeros $2\log^2(n)$ números. Esto se debe a que este algoritmo fue diseñado suponiendo cierta la siguiente conjetura (la cual nos permite acotar la distancia entre números primos y nos da cierta una idea de su distribución):

Conjetura 1.20 (Hipótesis de Riemann)**.**

$$\text{Definimos } \zeta(s) = \sum_{n=1}^{\infty} \frac{1}{n^s} \text{ con } s \in \mathbb{C} \setminus \{1\} \text{ función zeta de Riemann.}$$

$$\forall\ s \in \mathbb{C} \setminus \{1, -2, -4, \ldots, -2k\} \textit{ tales que } \zeta(s) = 0 \Rightarrow \mathcal{R}e(s) = 1/2$$

La conjetura sigue estando sin probar aunque es ampliamente aceptada[12]. Este es uno de los problemas abiertos más importantes de las matemáticas y el *Clay Mathematics Institute* ofrece una recompensa de un millón de dólares US$ a la persona que demuestre el teorema [11].

Volviendo al algoritmo de Miller, la complejidad viene determinada por el bucle anidado. En el peor caso realizamos en el bucle externo $2\log^2(n)$ operaciones y en cada una de ellas $O(\log^2(n))$ pues s es del orden $O(\log(n))$ y mediante *iterated squaring* se tarda del orden $O(\log n)$ en realizar cada potencia. Obtenemos por tanto un algoritmo de orden $O(\log^4(n))$.

Ahora bien, supongamos que no nos creemos la hipótesis de Riemann, Rabin [41] realizó una pequeña modificación (usando las ideas de Solovay y Strassen como indicamos anteriormente) en el algoritmo para convertirlo en probabilista. El algoritmo quedaría así:

Algoritmo 1.21 (Test Miller-Rabin)**.**

Input $= n, k \in \mathbb{Z}$ /* $k =$ número máximo de repeticiones */

Paso*. 1 Escribir $n-1 = 2^s d$ dividiendo sucesivas veces $n-1$ entre 2.*

Paso*. 2 Desde $i = 1$ hasta k haz*

- *Sea a un número escogido aleatoriamente en $\{1, \ldots, n-1\}$*
- *Si $a^d \neq 1$ mod n entonces Output = Compuesto.*
- *Desde $r = 0$ hasta $s-1$ haz*
  - *Si $a^{2^r d} \neq -1$ mod n entonces Output = Compuesto.*

Paso*. 3 Output = Probable Primo.*

[12]En el año 2004 Xavier Gourdon [17] verificó numéricamente la conjetura a lo largo de los primeros diez trillones de ceros no triviales de la función.

El algoritmo es de orden $O(k\log^2(n)) = O(\log^2(n))$ como vimos anteriormente. Ahora bien, si definimos $A = \{a \in \mathbb{Z}_n^*$ tales que no se cumple el teorema cuando n no es primo$\}$ (*conjunto de testigos*) se puede demostrar que $|A| > 3/4|\mathbb{Z}_n^*|$. Con lo cual tenemos:

- Para toda entrada es polinomial en tiempo.
- Si el input es primo entonces siempre devuelve primo.
- Si el input es compuesto la probabilidad de darlo por primo es menor que $1/4^k$.

Esto nos da un algoritmo en la clase **Co-RP**, como el caso de Solovay-Strassen, con la gran ventaja de no tener que calcular el símbolo de Legendre y de tener el doble de testigos (con lo cual una probabilidad de error mucho menor). Actualmente este es el algoritmo implementado en la inmensa mayoría de paquetes informáticos aunque en algún caso se combina con otros test para bajar extremadamente la probabilidad de error [39].

Antes de continuar me gustaría realizar un comentario hacerla de la probabilidad de error en estos algoritmos. Realizar 50 iteraciones del algoritmo de Miller-Rabin es algo plausible de llevar a cabo y tendríamos que la probabilidad de errar sería $1/4^{50} = 1/2^{100}$ una probabilidad muy inferior a la de que caiga un meteorito y destroce el laboratorio donde se está realizando el experimento. Pero es que incluso realizando "sólo" 20 iteraciones tenemos que la probabilidad de error es $1/2^{40}$ que es inferior a la probabilidad de que se produzca un error de *hardware* que produzca un error de cálculo y haga fallar al algoritmo. Teniendo en cuenta estas afirmaciones se ve claramente que hay que tener muy mala suerte para que uno de estos algoritmos falle en la práctica.

1.4. PRIMES está en RP

La pregunta ahora es, ¿no existen algoritmos que certifiquen primalidad? O equivalentemente ¿no existen algoritmos en **RP**? La respuesta es que son tremendamente complejos, sin embargo vamos a hablar un poco sobre ellos.

En 1980 Adleman, Pomerance y Rumely presentaron un algoritmo probabilístico que certificaba primalidad. Posteriormente fue mejorado por H. W. Lenstra y H. Cohen en 1981. El resultado fue el conocido como APRCL, un algoritmo de orden $O((\log(n))^{c\log(\log(\log(n)))})$ para cierta constante c. Teóricamente, el APRCL, es un

algoritmo exponencial aunque en la practica $\log(\log(\log(n)))$ se comporte como una constante para cualquier input que podamos considerar razonable[13]. Existe también una versión determinista mucho más impracticable y una implementación de ambos algoritmos, pero son ciertamente muy complejos de explicar (se basa en comprobar congruencias del estilo del Teorema pequeño de Fermat en cuerpos ciclotómicos). Una descripción del algoritmo se encuentra en [3], [43], [44] y [45].

Por la misma época surge otro algoritmo teórico (el ECPP) de la mano de Goldwasser y Kilian basado en curvas elípticas. Fue modificado posteriormente por Atkin sobre la cual se realiza una implementación (con Morain). El orden de este algoritmo es $O(\log^6(n))$, pero no en el caso peor, sino en el caso medio (esto es, para casi todos los inputs es polinomial, pero existen algunos inputs para los que es exponencial). Posteriormente Adleman y Huang obtuvieron un test probabilístico en tiempo polinomial (clase de complejidad **RP**) para el problema **PRIMES** aunque es totalmente impracticable. Una descripción de dichos test y de su implementación se puede encontrar en [3], [48] y en la pagina web de F.Morain [46].

Las características de este algoritmo son:

- Para toda entrada es polinomial en tiempo.
- Si el input es compuesto entonces siempre devuelve compuesto.
- Si el input es primo la probabilidad de darlo por compuesto es menor que $1/2^k$.

Actualmente las certificaciones de primalidad se realizan con implementaciones y mejoras de este algoritmo. La idea aquí es similar al algoritmo de Pratt, sustituyendo el grupo de $n-1$ elementos por el generado por una curva elíptica módulo n. Mientras tanto, el testigo pasa a ser un punto de la curva. La idea ahora es que el rango de valores del grupo es amplio y basta con ir probando con distintas curvas elípticas hasta encontrar una que genere un grupo cuyo orden sepamos factorizar (con esto evitamos, en parte, el indeterminismo de factorizar $n-1$ aunque generamos una búsqueda que puede ser muy compleja). Luego se aplica una idea similar al algoritmo de Pratt.

Como corolario obtenemos que $\mathbf{PRIMES} \in \mathbf{RP} \bigcap \mathbf{Co\text{-}RP} = \mathbf{ZPP}$. Veamos una idea que nos permitiría crear una implementación realista y seria que nos garantizaría la primalidad:

[13] ¡¡Y apara los no razonables también!! Para muestra un botón: $\log(\log(\log(10^{10^8}))) \approx 5$. Nótese que MAPLE produce un bonito mensaje de error (¡¡¡por *number overflow*!!!) si $n = 10^{10^9}\ldots$

Algoritmo 1.22 (Test ZPP)**.**

Input $= n \in \mathbb{Z}$

Paso. *1 Correr el test de Miller-Rabin*

Paso. *2 Si Output = Probable Primo entonces correr el test ECPP // en otro caso Output = Compuesto.*

Paso. *3 Output = Output del test ECPP.*

Con este algoritmo obtenemos un test determinista de primalidad polinomial en media. Es decir, siempre vamos a obtener una respuesta correcta aunque en algunos casos tardemos mucho tiempo. Esto es así pues si el número es *compuesto* y el algoritmo de Miller-Rabin devolviera *probable primo* el algoritmo ECPP sólo podría devolver *primo* pues de devolver *probable compuesto*, de hecho, lo sería. En los demás casos no había ninguna duda al respecto de la respuesta.

Como hemos visto el camino hasta la certificación de primalidad ha sido muy duro y en los últimos tiempos, enrevesado. Ha sido necesarios más de dos mil años y el trabajo de muchas personas brillantes para *solamente* conseguir algoritmos probabilistas.

Pero el panorama cambió el 8 de agosto de 2002. Manindra Agrawal, Nitin Saxena y Neeraj Kayal dan a conocer un borrador [50] (que aun se encuentra en ese estado, después de su tercera revisión) en el que presentan un test de primalidad determinista en tiempo polinomial extremadamente simple. El algoritmo se basa en una generalización del Teorema pequeño de Fermat que con unos pequeños arreglos permite construir un test en tiempo polinomial. Como la próxima sección está dedicada íntegramente a la explicación de este algoritmo obviaremos los detalles por el momento.

Aunque en sus primeras versiones usaba argumentos de teoría analítica de números y el orden de complejidad era bastante alto ($O(\log^{12}(n))$), gracias a las modificaciones de H. Lenstra y a A. Kalai, A. Sahai y M. Sudan incluidas en la segunda y tercera versiones del borrador, respectivamente, el algoritmo ofrece una prueba de la cota de complejidad $O(\log^{10,5}(n))$ totalmente elemental[14], mientras que con un poco más de esfuerzo y algo de teoría analítica de números, se prueba la cota $O(\log^{7,5}(n))$.

[14]Por elemental se entiende a nivel de un alumno con un primer curso de álgebra elemental cursado

Es de notar que bajo ciertas hipótesis como la anteriormente mencionada *hipótesis de Riemann*, *la conjetura de Artin* [14] o la *conjetura de la densidad de los primos de Sophie-Germain* [15] el algoritmo pasa a ser $O(\log^6 n)$ sin realizar absolutamente ninguna modificación[15].

A partir de la idea revolucionaria que presentaron Agrawal y compañía, primero Berrizbeitia [53], luego Qi Cheng [54] y por último Bernstein [55] presentaron[16] sendas mejoras realizando híbridos entre esta nueva idea y los test de curvas elípticas ya existentes. El resultado es un test probabilista que certifica primalidad de orden $O(\log^4(n))$ capaz de competir con el ECPP.

Hay que comentar que si la siguiente conjetura es cierta, una pequeña modificación en el algoritmo reduciría la complejidad del mismo a $O(\log^3(n))$ ¡Sin perder el determinismo!.

Conjetura 1.26. *Si r es un número primo que no divide a n y $(x-1)^n = x^n - 1$ mod $(x^r - 1, n)$ entonces o n es primo o $n^2 = 1$ mod r.*

Actualmente comienzan a aparecer implementaciones de este algoritmo [57] que confirman los resultados teóricos y es cuestión de tiempo que sea implementado por las principales librerías de funciones y paquetes de cálculo simbólico.

Veamos, a continuación, con más detalle el algoritmo y su demostración[17].

[15]Dichas conjeturas aseguran:

Conjetura 1.23 (Conjetura de Artin)**.** *Dado $n \in \mathbb{N}$ tal que $n \neq a^b \ \forall\ a, b \in \mathbb{N}$ el número de primos $q \leq n$ para los cuales $o_q(n) = q - 1$ se aproxima asintóticamente a $A(n)\frac{n}{\ln(n)}$ donde $A(n) > 0,35$ es la constante de Artin.*

Definición 1.24 (Primos de Sophie-Germain)**.** *Se denomina primos de Sophie-Germain a las parejas de números $(n, 2n+1)$ tales que ambos son números primos.*

Conjetura 1.25 (Conjetura de la densidad de los primos de Sophie-Germain)**.** *Dado $n \in \mathbb{N}$ el número de primos $q \leq n$ de Sophie-Germain se aproxima asintóticamente a $\frac{2C_2 n}{\ln^2(n)}$ donde $C_2 \approx 0,66$ es la constante de los primos gemelos [16].*

[16]Notar que todos las citas anteriormente mencionadas están en fase de borrador y no han sido publicadas.

[17]En los apéndices se puede encontrar dos implementaciones del algoritmo original, una en MAPLE y otra en C++ así como la modificación propuesta para aplicar la conjetura 1.26

2. PRIMES está en P, el algoritmo AKS

2.1. El algoritmo y su correctitud

Con anterioridad hemos estudiado muchos algoritmos que realizan un test de primalidad, desde la criba de Eratóstenes hasta el AKS. Veamos con más detalle este último pues es un resultado muy importante al ser el primer algoritmo determinista en **P** que certifican primalidad.

El algoritmo es bien simple y podría explicarse como "extra" al finalizar un primer curso de álgebra abstracta. Se basa fundamentalmente en el siguiente resultado bien conocido desde hace mucho tiempo[18]:

Teorema 2.1 (Generalización del Teorema pequeño de Fermat)**.**

Sea $a \in \mathbb{Z},\ n \in \mathbb{N},$ con $(a, n) = 1$ entonces n es primo si y solo si $(x+a)^n = x^n+a \mod n$

Demostración. Por la fórmula del binomio de Newton, sabemos que el coeficiente que acompaña a x^i en el polinomio $f(x) = (x + a)^n - (x + a)$ es $\binom{n}{i}a^{n-i}$

Si n es primo tenemos que $\binom{n}{i} = 0$ pues es un múltiplo de n.

Si n es compuesto, sea q un primo tal que divida a n y sea k la mayor potencia de q que divida a n.

Tenemos que:

- q^k no divide a $\binom{n}{q}$ pues si $n = q^k r\binom{n}{q} = \frac{n(n-1)...(n-q)}{q!} = \frac{q^{k-1}r(n-1)...(n-q)}{(q-1)!}$ y como en el numerador no hay ningún múltiplo de q obtenemos el resultado.
- $(q^k, a^{n-q}) = 1$ pues $(n, a) = 1$

Por lo que el coeficiente que acompaña a $x^q \neq 0$ y por lo tanto no se tiene que $(x + a)^n = x^n + a \mod n$ □

Con esto obtenemos una caracterización de los números primos que podríamos usar de forma algorítmica. Pero surge un problema, el número de operaciones es exponencial, con lo cual hemos retrocedido bastante. Sin embargo vemos como podemos reducir el número de operaciones.

Lo ideal sería que una afirmación de este tipo fuera cierta:

[18]El teorema generaliza el Teorema pequeño de Fermat

Conjetura 2.2. *Sea $a \in \mathbb{Z}$, $n \in \mathbb{N}$ y $(a,n) = 1$ entonces n es primo si y solo si $(x+a)^n = x^n + a \ mod \ (x^{r-1}, n)$ donde r es suficientemente pequeño.*

Pero por desgracia la anterior propiedad es falsa. Bueno, más bien, no se da en general, existen números compuesto que la cumplen. Ahora bien, no está todo perdido, veremos a continuación que si se da la propiedad para un determinado r y un conjunto de a podemos recuperar la caracterización. Básicamente esto es el algoritmo.

El factor novedoso de este algoritmo está exactamente ahí, en la elección de la r y las a. Es una idea novedosa y que previsiblemente servirá de inspiración para futuros algoritmos no necesariamente relacionados con los test de primalidad.

Presentamos a continuación el algoritmo original del AKS. Más adelante comentaremos diversas modificaciones y posibles mejoras:

Algoritmo 2.3 (AKS)**.**

Input $= n \in \mathbb{Z}$

***Paso**. 1 Si $n = a^b$ para algún $a \in \mathbb{N}$ y $b > 1$ entonces Output = Compuesto.*

***Paso**. 2 Encuentra el menor r tal que $o_r(n) > 4\log^2(n)$*

***Paso**. 3 Si $1 < (a,n) < n$ para algún $a \leq r$ entonces Output = Compuesto.*

***Paso**. 4 Si $n \leq r$ entonces Output = Primo.*

***Paso**. 5 Desde $a = 1$ hasta $\lfloor 2\sqrt{\varphi(r)}\log(n) \rfloor$ comprueba*

- *si $(x+a)^n \neq x^n + a \ mod \ (x^r - 1, n)$ entonces Output = Compuesto.*

***Paso**. 6 Output = Primo.*

Vamos a ver que efectivamente este algoritmo cumple su cometido y distingue números primos de compuestos.

Teorema 2.4. *Si n es primo entonces el algoritmo devuelve Primo.*

Demostración. Observamos que si n es primo tanto en el primer como en el tercer paso, el algoritmo no puede nunca devolver Compuesto. Así como, por el teorema 2.1, tenemos que el quinto paso tampoco devolverá Compuesto. Por lo tanto el algoritmo devolverá Primo o en el cuarto paso o en el sexto. □

Para realizar el recíproco de este teorema deberemos trabajas más. La idea es definir un conjunto de valores a partir de n y comprobar que generan un grupo cíclico que acotaremos. Ahora bien, si se cumple el paso quinto del algoritmo y suponemos que n es compuesto llegaremos a un absurdo, pues no se respetarán las cotas prevista. Sin más vamos a desarrollar la teoría.

Veamos primero si existe un r con las propiedades pedidas en el segundo paso. Para ello usaremos el siguiente lema:

Lema 2.5.

$$\text{lcm}\{1,\ldots,k\} \geq 2^k \text{ si } k > 1$$

Demostración. Sea $d_n = \text{lcm}\{1,\ldots,n\}$ y consideramos para $1 \leq m \leq n$ la integral:

$$I = I(m,n) = \int_0^1 x^{m-1}(1-x)^{n-m}dx = \sum_{r=0}^{n-m}(-1)^r \binom{n-m}{r}\frac{1}{m+r}$$

Que resolviéndola por partes obtenemos:

$$I = \frac{1}{m\binom{n}{m}}$$

Tenemos entonces que:

- $Id_n \in \mathbb{N}$
- $m\binom{n}{m}$ divide a d_n pues $Im\binom{n}{m} = 1 \Rightarrow d_n Im\binom{n}{m} = d_n$ y como $d_n I = q \in \mathbb{N}$ tenemos que $m\binom{n}{m}$ divide a d_n para todo $1 \leq m \leq n$.

En particular, $n\binom{2n}{n}$ divide a d_{2n} y como $(2n+1)\binom{2n}{n} = (n+1)\binom{2n+1}{n+1}$ tenemos entonces que $(n+1)\binom{2n}{n}$ y $(2n+1)\binom{2n}{n}$ dividen a d_{2n-1} Además, como $(n, 2n+1) = 1$ obtenemos que $n(2n+1)\binom{2n}{n}$ divide a d_{2n+1} y por lo tanto $d_{2n+1} \geq n(2n+1)\binom{2n}{n} \geq n\sum_{i=0}^{2n}\binom{2n}{i} = n(1+1)^{2n} = n2^{2n} \geq 2^{2n+1} \Rightarrow d_N \geq 2^N$ cuando $n > 1$ □

Teorema 2.6.

$$\exists\ r \leq \lceil 16\log^5(n)\rceil \text{ tal que } o_r(n) > 4\log^2(n)$$

Demostración. Supongamos que $A = \{r_1,\ldots,r_t\}$ son todos los números que cumplen que $o_{r_i}(n) \leq 4\log^2(n)$.

Cada uno de ellos dividirá a:

$$z = \prod_{i=1}^{\lfloor 4\log^2(n)\rfloor} (n^i - 1) < n^{16\log^4(n)} \leq 2^{16\log^5(n)}$$

Pues $n^{o_{r_i}} - 1 = qr_i \wedge o_{r_i} \leq 4\log^2(n)\ \forall\ i$

Además tenemos que $\forall\ x \in A, x < 16\log^5(n)$ pues

$$\prod_{x\in A} x \leq z < 2^{16\log^5(n)} \Rightarrow \sum_{x\in A} x < 16\log^5(n) \Rightarrow x < 16\log^5(n)\ \forall\ x \in A$$

Por otro lado, sea $B = \{1, \ldots, \lceil 16\log^5(n)\rceil\}$ Por el lema 2.5 se tiene que: $lcm\{x \in B\} > 2^{\lceil 16\log^5(n)\rceil}$ Además $A \subset B$, pero ¿se puede dar la igualdad? No, pues de darse tendíamos que:

$2^{\lceil 16\log^5(n)\rceil} > z > lcm\{x \in A\} = lcm\{x \in B\} > 2^{\lceil 16\log^5(n)\rceil}$ ♯

Por lo que tenemos que $A \neq B$ y por lo tanto $\exists\ r < \lceil 16\log^5(n)\rceil$ tal que $o_r(n) > 4\log^2(n)$ □

Veamos ahora la definición de una propiedad importante:

Definición 2.7 (Números introspectivos)**.** *Sea un polinomio $f(x)$ y un número $m \in \mathbb{N}$. Decimos que m es introspectivo para $f(x)$ si y solo si $[f(x)]^m = f(x^m)$ mod $(x^r - 1, p)$*

Lema 2.8. *a) Si m, n son introspectivos para $f(x)$ entonces mn también lo es.*

b) Si m es introspectivo para $f(x)$ y $g(x)$ entonces m también lo es para $f(x)g(x)$.

Demostración. a) Como m es introspectivo para$f(x)$ tenemos que $[f(x)]^m = f(x^m)$ mod $(x^r - 1, p) \Rightarrow \{[f(x)]^m\}^n = [f(x^m)]^n$ mod $(x^r - 1, p)$ y como n también es introspectivo tenemos que $[f(x)]^{mn} = [f(x^{mn})]$ mod $(x^r - 1, p)$

b) Trivial pues $[f(x)g(x)]^m = f(x)^m g(x)^m = f(x^m)g(x^m)$ mod $(x^r - 1, p)$ pues m es introspectivo tanto para $f(x)$ como para $g(x)$.

□

Vamos ahora a definir los grupos que nos servirán para encontrar una contradicción que nos asegurará la correctitud del algoritmo.

Definición 2.9.

Sea $I = \{n^i p^j \text{ con } i, j \geq 0\}$. Definimos $G = \{x \bmod r \text{ con } x \in I\} < \mathbb{Z}_r^*$ con $|G| = t$

Definición 2.10. *Sea $q_r(x)$ el r-ésimo polinomio ciclotómico sobre $\mathbb{Z}_p$ y $h(x)$ un polinomio de grado $o_r(p)$ divisor de $q_r(x)$. Denotamos ahora:*

$$P =< x+1, \ldots, x+l > \text{. y definimos } \mathcal{G} = \frac{P}{h(x)} < F = \frac{\mathbb{Z}_p[x]}{h(x)}$$

Corolario 2.11. $\forall\ m \in I,\ m$ *es introspectivo* $\forall\ f(x) \in P$

Demostración. La demostración se sigue de los dos lemas anteriores. □

Vamos a acotar ahora superior e inferiormente $\mathcal{G}$. Para ello necesitaremos el siguiente lema:

Lema 2.12. *El número de monomios distintos de grado d en n variables es* $\binom{d+n}{n}$

Demostración. La demostración de este lema se sigue por conteo directo. □

Lema 2.13.

$$|\mathcal{G}| \geq \binom{t+l-2}{t-1}$$

Demostración. Sean $f(x), g(x) \in P$ con $\delta(f(x)),\ \delta(g(x)) < t$. Supongamos que $f(x) = g(x)$ en $\mathcal{G}$ y que $m \in I$. Como m es introspectivo para $f(x)$ y $g(x)$ tenemos que:

$$[f(x)]^m = [g(x)]^m \Rightarrow f(x^m) = g(x^m) \Rightarrow q(x^m) = f(x^m) - g(x^m) = 0\ \forall\ m \in G$$

Sabemos que $|G| = t$ y por lo tanto $q(x)$ tiene al menos t raíces, pero esto es imposible, pues por construcción debe tener menos de t. Por lo tanto tenemos que $f(x) \neq g(x)$ en $\mathcal{G}$.

Como $l = \lfloor 2\sqrt{\varphi(r)}\log(n) \rfloor < 2\sqrt{r}\log(n) < r < p$ tenemos que existen al menos $l-1$ elementos de grado uno en $\mathcal{G}$ (pues alguno puede ser 0 si $\delta(h(x)) = 1$). Luego por el lema anterior obtenemos entonces $\binom{(t-1)+(l-1)}{l-1} = \binom{t+l-2}{t-1}$ □

Lema 2.14.

$$\text{Si } n \neq p^b \text{ entonces } |\mathcal{G}| < \frac{n^{2\sqrt{t}}}{2}$$

Demostración. Consideramos $J = \{n^i p^j \ /\ 0 \leq i, j \leq \sqrt{t}\} \subset I$. Tenemos entonces que $|J| = (\lfloor\sqrt{t}\rfloor + 1)^2 > t$ si $n \neq p^b$. Pero como $|G| = t$ tenemos que existen $m_1 > m_2 \in J$ tales que $m_1 = m_2 \bmod r$. Por lo tanto $x^{m_1} = x^{m_2} \bmod x^r - 1$.

Sea $f(x) \in P$. Entonces $[f(x)]^{m_1} = f(x^{m_1}) = f(x^{m_2}) = [f(x)]^{m_2} \bmod (x^r - 1, p)$. Tenemos entonces que $f(x) \in \mathcal{G}$ es una raíz de $q(y) = y^{m_1} - y^{m_2}$ en F. Luego $q(y)$ tiene al menos $|\mathcal{G}|$ raíces distintas pero $\delta(q(y)) = m_1 \leq (np)^{sqrtt} < \frac{n^{2\sqrt{t}}}{2}$. Con lo cual obtenemos que $|\mathcal{G}| < \frac{n^{2\sqrt{t}}}{2}$ □

Ahora estamos en condiciones de demostrar el converso al teorema 2.4.

Teorema 2.15. *Si el algoritmo devuelve Primo entonces n es primo.*

Demostración. Suponemos que el algoritmo ha devuelto Primo. De hacerlo en el cuarto paso estaríamos obviamente ante un primo, luego nos queda analizar cuando devuelve Primo en el sexto paso.

Por el lema 2.13 tenemos que si $|G| = t$ y $l = \lfloor 2\sqrt{\varphi(r)}\log(n) \rfloor$ entonces:

$$|\mathcal{G}| \geq \binom{t+l-2}{t-1} \geq \binom{l-1+\lfloor 2\sqrt{t}\log(n)\rfloor}{\lfloor 2\sqrt{t}\log(n)\rfloor} \geq \binom{2\lfloor 2\sqrt{t}\log(n)\rfloor - 1}{\lfloor 2\sqrt{t}\log(n)\rfloor} \geq$$
$$\geq 2^{\lfloor 2\sqrt{t}\log(n)\rfloor} \geq \frac{n^{2\sqrt{t}}}{2}$$

Ahora bien, por el teorema 2.14 tenemos que $|\mathcal{G}| < 1/2n^{2\sqrt{t}}$ si $n \neq p^b$ luego $\exists\, b > 0$ tal que $n = p^b$ pero si $b > 1$ el algoritmo hubiera determinado en el segundo paso que n es compuesto, con lo cual $b = 1$ y $n = p$ □

2.2. Análisis de complejidad

Vamos a comprobar ahora la complejidad del algoritmo. Nos basaremos en el siguiente resultado.

Teorema 2.16. *a) La adición, multiplicación y división de dos enteros de talla* $\log(n)$ *se puede realizar en* $\widetilde{O}(\log(n))$[19].

b) La adición, multiplicación y división de dos polinomios de grado d con coeficientes de talla a lo sumo $\log(n)$ *se puede realizar en* $\widetilde{O}(d\log(n))$.

Demostración. Los detalles y algoritmos se pueden encontrar en el libro [1]. □

Vayamos paso por paso.

[19] A partir de ahora denotaremos $\widetilde{O}(f(x)) = O(f(x)\log^{O(1)}(\log^{O(1)}(n)) = O(f(x)\log^{\varepsilon}(n))$

1. Si $n = a^b$ para algún $a \in \mathbb{N}$ y $b > 1$ entonces Output = Compuesto

Para realizar este paso realizamos la factorización del polinomio $x^b - n$ para $1 \leq b \leq \log(n+1)$. Si obtenemos un factor de grado 1 tenemos que: $x^b - n = (x-a)g(x) \Rightarrow a^b - n = 0$ y el algoritmo devuelve Compuesto.

La factorización de dicho polinomio sobre el $\mathbb{Z}_n$ se puede realizar en $\widetilde{O}(\log^2(n))$ usando el algoritmo de factorización descrito en [1]. Ahora bien, como ejecutamos dicho algoritmo un máximo de $\log(n)$ veces tenemos que este paso se puede realizar en un máximo de $\widetilde{O}(\log^3(n))$ pasos.

2. Encuentra el menor r tal que $o_r(n) > 4\log^2(n)$

Para este paso probamos de forma secuencial $n^k \neq 1 \bmod r$ con $1 \leq k \leq 4\log^2(n)$ hasta encontrar un r que cumpla dicha propiedad.

Por el lema 2.6 sabemos que no son necesarias más de $O(\log^5(n))$ pruebas y sabiendo que en cada prueba realizamos $O(\log^2(n))$ multiplicaciones obtenemos que en este paso usamos $O(\log^7(n))$.

3. Si $1 < (a,n) < n$ para algún $a \leq r$ entonces Output = Compuesto

En este paso realizamos r cálculos del *gcd*. Sabemos que cada cálculo conlleva $O(\log(n))$ operaciones (ver [1]) con lo cual tenemos que este paso realizamos $O(\log^6(n))$ operaciones.

4. Si $n \leq r$ entonces Output = Primo

Este paso es una simple comparación que se realiza en $O(1)$.

5. Desde $a = 1$ hasta $\lfloor 2\sqrt{\varphi(r)}\log(n) \rfloor$ comprueba

   a) si $(x+a)^n \neq x^n + a \pmod{x^r - 1, n}$ entonces Output = Compuesto

En este paso debemos realizar $\lfloor 2\sqrt{\varphi(r)}\log(n) \rfloor$ test de nulidad. Para cada uno de ellos debemos realizar $O(\log(n))$ multiplicaciones de polinomios de grado r con coeficientes no mayores que $O(\log(n))$. Por lo tanto realizamos $\widetilde{O}(r\log^2(n))$ operaciones. Con lo cual resulta que la complejidad de este paso es $\widetilde{O}(r\sqrt{\varphi(r)}\log^3(n)) = \widetilde{O}(r^{\frac{3}{2}}\log^3(n)) = \widetilde{O}(\log^{10,5}(n))$.

Como la complejidad de este ultimo domina a todas las anteriores tenemos que este algoritmo tiene una complejidad de $\widetilde{O}(\log^{10,5}(n))$.

2.3. Algunas mejoras

Vamos a comentar un poco las mejoras que ya anunciábamos al final del capítulo 1.4. A la luz del análisis de complejidad es claro que el mejor método para mejorar el orden del algoritmo es mejorar la cota para r. Eso es exactamente lo que realizamos cuando suponemos cierta las conjeturas de Artin, de Sophie-Germain o de Riemann. Al mejorar las cotas sobre r automáticamente se mejoran las cotas de los siguientes bucles.

Otra forma de rebajar la complejidad sería mejorar el conjunto de valores en los que comprobamos la igualdad del paso 5. Esa es la idea de las modificaciones de Berrizbeitia, Qi Cheng y Bernstein. Aunque dichas mejoras finalmente desembocaron en un algoritmo probabilista, la idea fue "mezclar" este algoritmo con los algoritmos ya conocidos de curvas algebraicas.

Por último destacar la mejora sustancial que se produciría de ser cierta la conjetura 1.26. Modificando ligeramente la búsqueda de r conseguiríamos reducir la complejidad a $O(\log^3(n))$. Esto haría a este algoritmo un test de primalidad extremadamente competitivo que podría sustituir no solo al ECPP sino a los test probabilistas.

3. Implicaciones del resultado

Comentemos ahora las implicaciones que va a tener este resultado a lo largo de los próximos años. La principal y mayor implicación va ha ser la de resolver uno de los problemas teóricos que traía de cabeza a los matemáticos desde hace dos mil años. El sueño de Gauss de conseguir un algoritmo capaz de distinguir primos de compuestos ha sido realizado.

El principal uso que se le está dando a los números primos actualmente es el de generador de claves criptográficas. Para ellos son necesarios computar números primos exageradamente grandes (típicamente de 256 dígitos o bits). Para esta labor usualmente se ha usado el test de Miller-Rabin con el posible riesgo que conlleva (ínfima, pero existente). Si la conjetura 1.26 se confirma, o la implementación de la modificación probabilista resulta ser viable deberían actualizarse todos los sistemas criptográficos a fin de aumentar la seguridad y / o eficiencia (pues ambos test asegurarían la primalidad de los resultados favorables).

Ahora bien, este algoritmo no va ha provocar una pérdida de seguridad pues las claves criptográficas actuales, más concretamente del RSA, se basan en el problema de factorizar un número (no en el problema de la primalidad). Esto es, encontrar $p_1, \ldots, p_i$ números primos y $n_1 \ldots, n_i$ tales que $n = p_1^{n_1} \ldots p_i^{n_i}$.

Actualmente, factorizar un número es un problema extremadamente duro de la clase **NP** y no parece que vaya a aparecer pronto un algoritmo que lo mejore. Los mejores algoritmo en este campo son algoritmos que se aprovechan de la teoría de curvas elípticas y de la criba de números siendo todos ellos bastante complejos. En el libro [3] se encuentra ampliamente desarrollado este tema.

Realmente debería de suceder más bien todo lo contrario. El AKS permitirá encontrar primos con más dígitos, y que no pertenezcan a ninguna familia conocida, con mayor facilidad. Lo cual dificultaría la búsqueda por fuerza bruta o por los métodos de criptoanálisis más comunes.

Para finalizar este capítulo habría que mencionar también otro de los grandes descubrimientos. El AKS representa el primer *test de nulidad* determinista, para un tipo concreto de polinomio, en tiempo polinomial. La idea intuitiva del resultado sería poder determinar si un polinomio es 0 evaluando en MUY pocos puntos (del orden de $O(\log^{O(1)}(d))$ donde d es el grado del polinomio). Hasta ahora sólo existían test probabilistas y el famoso teorema de interpolación polinómica que necesitaba evaluar en d puntos y por lo tanto exponencial.

Se espera que el resultado impulse esta rama de las matemáticas y abra la puerta a otros descubrimientos en el área que permitan mejorar la evaluación de polinomios univariados o multivariados.

4. Implementación del algoritmo AKS

Tras esta extensa demostración del algoritmo vamos ahora a comentar un poco las motivaciones que pueden llevar a realizar una implementación. Principalmente se pueden resumir en tres causas:

- Comprobar la velocidad del algoritmo.
- Comprobar la eficacia de los algoritmos probabilistas.
- Comprobar la conjetura propuesta por los autores del algoritmo.

Para cada uno de ellos dedicaremos más adelante un apartado, pero ahora vamos a centrarnos en lo que supuso el proceso de implementación.

4.1. Implementación en un paquete de cálculo simbólico

La primera consideración al plantearnos la implementación del algoritmo fue realizarlo en un paquete comercial de cálculo simbólico. La razón es simplemente disponer de una las herramientas necesarias para realizar el algoritmos, sin necesidad de programar estructuras de datos para los enteros de precisión arbitraria[20] ni para los polinomios ni de tener que programar las funciones básicas para operar con dichas estructuras. Se consideraron los siguientes paquetes:

- MAPLE [70].
- MuPAD [66].
- Magma [68].

Todos ellos de pago. Se decidió usar MAPLE por disponer de más experiencia con él y ser un software de amplio reconocimiento.

El programación presentaba tres dificultades claras: la factorización del primer paso, el cálculo del orden de r del segundo paso y el test de nulidad del penúltimo paso. MAPLE contaba con funciones propias que realizaban dichas tareas y

[20] Recordar que los ordenadores utilizan un sistema numérico finito. Por lo general el entero más grande que puede considerar un ordenador es de 2^32 que para nuestros propósitos es un número ridículo.

tras comprobar en la documentación los pormenores se realiza la implementación expuesta en el apéndice A sin mayores dificultades[21].

Las dificultades aparecieron inmediatamente al ejecutarlo para números relativamente pequeños. Para $n = 20000$ el algoritmo tardaba del orden de un minuto en determinar si un número es primo o no, lo cual es inaceptable cuando se compara con el test que posee MAPLE, basado en el test de Miller-Rabin, cuya ejecución es instantánea.

Al no poder acceder al núcleo de MAPLE y no saber a ciencia cierta que estaba sucediendo para obtener tan malos resultados, conjeturamos la posibilidad de que MAPLE estuviera desarrollando todos los coeficientes del polinomio en el quinto paso, antes de hacer la reducción. Lo cual daría como resultado un algoritmo exponencial. Se programó una versión alternativa al algoritmo que posee MAPLE basándose en *iterated squaring* con reducción módulo $x^r - 1, n$ de los coeficientes para controlar el crecimiento, pero el algoritmo resultante resultó ser aún más ineficiente, con lo que concluimos que MAPLE estaba realizando correctamente la operación y achacamos el pobre rendimiento al propio programa. La implementación se puede consultar en el apéndice A.1.

4.2. Implementación con una librería de C++

Tras este pequeño traspiés se consideró usar una librería de C++. Aquí el proceso de selección fue muchísimo más costoso al tener que leer la documentación de cada librería para comprobar que poseían las estructuras de datos requeridas y las funciones necesarias. Se consideraron las siguientes librerías:

- PariGP [67].
- LiDIA [69].
- NTL [71].

Finalmente se escogió la librería NTL de Victor Shoup por ser la que poseía la documentación más clara y todas las herramientas necesarias para la implementación del algoritmo... excepto una. Debido a la estructura de la implementación de los polinomios, el grado máximo que se puede alcanzar es de 2^{64} insuficiente para

[21] Aunque cumpliendo el dicho informático que afirma: "El 20 % del tiempo dedicado a la realización de un programa se invierte en la programación... mientras que el 80 % restante en su *debug*".

una prueba en condiciones con los test probabilísticos en su propio terreno, los *primos industriales*[22]. La alternativa era realizar la modificación de la estructura por nosotros mismos pero dicha tarea podría ser perfectamente, en sí misma, uno o dos trabajos dirigidos, por lo que se optó por continuar adelante teniendo en cuenta dicha limitación.

La programación fue algo más compleja debido al entorno menos amigable de una librería en C++ sobre un software comercial. Se usa un programa de desarrollo integrado, Dev-C++ v4.9.9.2 [59] para ser exactos, que trae como compilador MinGW [60], un "port" para windows del compilador estándar GCC 3.4.2.

De las tres dificultades nombradas anteriormente, la primera y la tercera se podían solventar con funciones implementadas en el propio paquete, mientras que para el cálculo del orden hubo que programarse una función. Tras este pequeño inconveniente, de fácil solución, se termina la programación y comienzan los primeros test. La implementación se puede consultar en el apéndice B.

4.3. Experiencias realizadas

4.3.1. Benchmark

Una vez obtenida la implementación del algoritmo, el primer paso fue compilarlo[23] y realizar una prueba muy simple: buscar todos los primos entre nueve mil y nueve mil cien devolviéndonos una tabla con el tiempo empleado y si es primo o no. Usamos dicho programa como benchmark en diferentes plataformas obteniendo los siguientes resultados:

En la gráfica 1 representamos el tiempo que tarda la implementación del algoritmo (sin la modificación de la conjetura) a la hora de calcular los números primos en el anterior rango (para los demás valores el algoritmo tarda un tiempo prácticamente despreciable). A la vista de los resultados se observa como la compilación del algoritmo sin optimizar (línea roja) es mucho más lenta que su contrapartida optimizada (en todas las plataformas). La gráfica concuerda completamente con los resultados teóricos esperados a priori (véase, por ejemplo, [61]) sobre potencia de cálculo de los distintos procesadores (con la sorpresa del buen rendimiento de los procesadores Pentium M, línea magenta). Dicha gráfica ha sido generada mediante el

[22] Primos con 1024 dígitos o bits.

[23] Se realizó una compilación completamente optimizada para ordenadores Pentium 4 y Athlon XP, esto es, para procesadores con las instrucciones SSE2, así como otra sin estas optimizaciones.

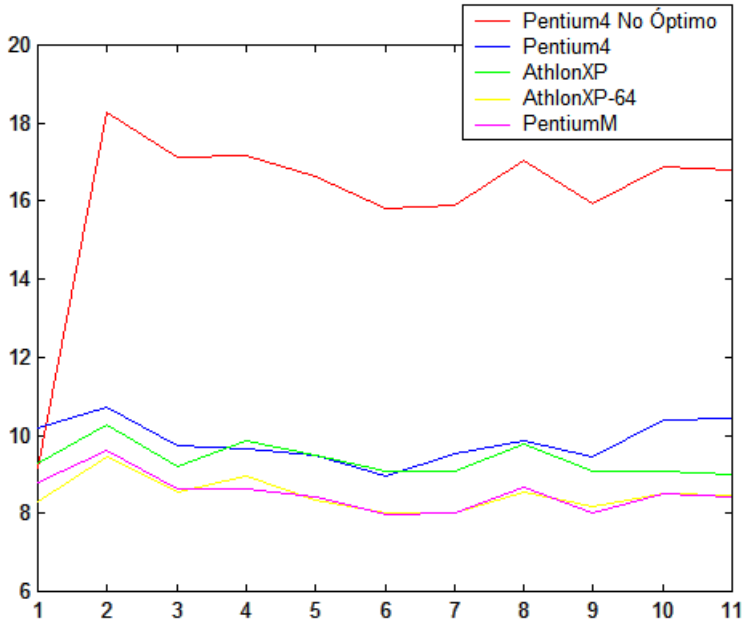

Figura 1: Tiempo de cálculo de cada procesador.

software MatLab [72] usando los datos generados por la implementación aportados por diversos colaboradores. El script se puede encontrar en el apéndice C.

Hubiera deseado incluir otras arquitecturas, como una estación de trabajo SUN, Macintosh o incluso Linux, pero la imposibilidad de disponer de dichos medios materiales y disponer de compiladores específicos, no me ha permitido completar la comparativa.

4.3.2. Computación GRID

Tras este primer experimento con el programa decidimos realizar un test de cómputo intensivo. Los objetivos eran los siguientes:

- Comparar la velocidad del algoritmo AKS con el de Miller-Rabin.
- Intentar hallar la curva asintótica de velocidad del algoritmo AKS.
- Buscar errores en el algoritmo de Miller-Rabin.
- Estimar la función $\Pi(x)$, esto es, comprobar empíricamente el teorema de densidad de los números primos [10].

Para realizar dicha experiencia, temiendo una necesidad de cálculo excesivo, se buscan alternativas a la ejecución en una buena máquina durante periodos extremadamente largos de tiempo. Usando las idea de paralelización de algoritmos o técnicas de cómputo GRID (para mayor información consultar el mayor experimento de cálculo GRID actualmente en ejecución, el proyecto *SETI@home* [62] o el proyecto europeo *DataGrid* [64] del cual el *IFCA* es un miembro participante) recurrimos a separar el cálculo en los ocho ordenadores del aula de informática que posee el departamento de Matemáticas, Estadística y Computación.

Las particularidades de este algoritmo posibilitan un paralelismo muy acusado. Se ve claramente que el cuello de botella se forma en el bucle del penúltimo paso. Ahora bien, como cada iteración del bucle es completamente independiente y no es necesario, aunque si aconsejable, realizarlas en un orden determinado, se podría *paralelizar* el algoritmo ejecutando una iteración en cada microprocesador disponible. Esto posibilitaría una reducción en un factor k la complejidad del algoritmo, donde k es el número de procesadores disponible. Como no se disponía de ningún entorno multiprocesador donde ejecutar el algoritmo y desconozco los detalles de la implementación que con lleva dicha modificación, finalmente no se realizó la experiencia[24].

Otra idea de paralelización consistía en repartir los números, sobre los que se iba a realizar el test, entre los distintos ordenadores. De esta forma cada ordenador corría el programa original sobre un conjunto de números de forma independiente. Lo ideal hubiera sido seguir un modelo *Cliente-Servidor* (ver figura 2) que es el usado en los programas de cómputo GRID como los mencionados anteriormente. El modelo es algo así como tener un ordenador corriendo un software especial (servidor) que se encargaría de repartir el trabajo de forma dinámica sobre los demás ordenadores, que correrían las aplicaciones clientes. Esto es, un ordenador cliente, contacta con el servidor y pide trabajo, que en este caso sería un rango de números. El servidor se lo envía y lo marca como *en proceso*. El servidor debe asegurar que nunca se envía el mismo rango a dos ordenadores a la vez y que nunca se computa dos veces el mismo rango de números. También debe encargarse de enviar rangos de números de forma que la media de tiempo que se tarde sea la misma para cada rango. Esto se consigue suponiendo que el crecimiento sigue una curva polinomial monótona creciente, con lo cual el tamaño de los rangos decrece al aumentar el número de cifras. Si pasado un tiempo no ha recibido la respuesta desmarca ese rango y lo vuelve a asignar a

[24]La implementación se realiza a nivel de lenguaje de programación usando técnicas de *multithread* [58], literalmente, multihilos.

otro ordenador. En caso de que obtenga respuesta marca el rango como *procesado* y se encarga de ensamblar los resultados en el archivo de salida.

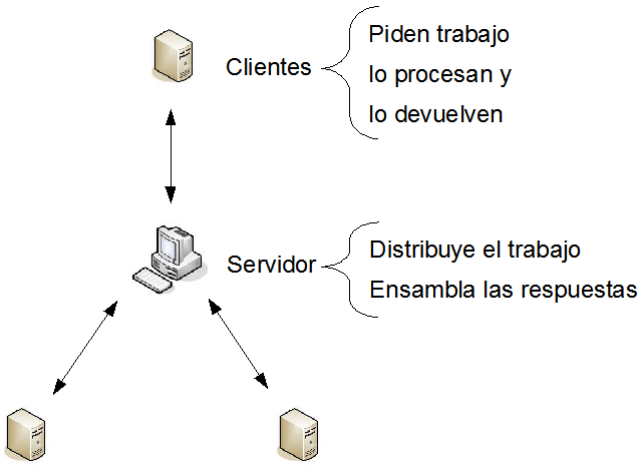

Figura 2: Diagrama de programación cliente-servidor.

Esta idea se descartó, parcialmente, al conllevar una programación mucho más delicada y de alto nivel, la cual no es el objetivo de este trabajo. Sin embargo se realizó una paralelización *estática* adoptando yo el papel de servidor, repartiendo los rangos *a mano* y, en principio, ensamblándolos también.

El resultado fue un fracaso absoluto. El tiempo previsto de cálculo no era ni una décima parte del necesitado en realidad, aparte del inconveniente de ser un aula en uso, con lo cual el programa fue cerrado antes de completar sus tareas en prácticamente todos los ordenadores.

4.3.3. Prueba de estrés al algoritmo de Miller-Rabin

Después de este traspiés la siguiente prueba fue buscar, por fuerza bruta, un error en el algoritmo de Miller-Rabin. Haciendo $k = 1$ (recordar que k era el parámetro que contaba el número de iteraciones del algoritmo y era el factor relevante para la estimación de la probabilidad de error) y ejecutando un millón de veces sobre un número de Carmichael que escogimos de entre la lista obtenida de [33]. El fichero de salida ocupa del orden de 11 MB de información y de forma asombrosa muestra que el algoritmo NO falló en ninguna ocasión, en contra de la probabilidad estimada a priori. Dichos datos nos hace sospechar, aunque en la documentación de la librería no lo deja muy claro, que el algoritmo implementado NO es el Miller-Rabin clásico, sino una combinación de éste con otros test como, por ejemplo, con el test de pseudoprimos de Lucas [40], combinación usada, por ejemplo en el paquete de cálculo

simbólico Mathematica [39]. Por falta de tiempo no se ha procedido a un estudio sistemático del código fuente de librería para averiguar dicho dato. Otra posibilidad sería que el número escogido no fuera realmente un número de Carmichael, aunque dicha posibilidad es mucho menos plausible a priori.

La modificación propuesta para la realización de este experimento se puede encontrar en el apéndice B.1.

4.3.4. Verificación de la conjetura 1.26

Por último, se realizó un test similar al segundo propuesto, pero esta vez ejecutado con la modificación que hace al algoritmo de orden $O(\log^3(n))$. Para ello realizamos en paralelo una llamada a nuestro algoritmo modificado y al algoritmo de Miller-Rabin. Cuando los resultados difieran llamamos al algoritmo estándar AKS. Los resultados empíricos muestran que la conjetura NO es válida, pues falla sobre algunos números.

La prueba se ha realizado en dos tandas, la primera vez sobre los primeros diez mil números primos y en la segunda sobre los veinte mil primeros. Ambas lista han sido sacadas de [18].

Se observa, además, que el algoritmo es sólo un poco más lento que Miller-Rabin, aun siendo una implementación susceptible de mejoras (la implementación del paso 1 se puede mejorar notablemente y otras partes del código son probablemente también mejorables).

Sorprendentemente, esta vez sí han aparecido números en los que el algoritmo de Miller-Rabin falla, lo cual nos ha llevado a repetir en dichos números, el test anterior. Para nuestra sorpresa ahora efectivamente produce salidas con errores de forma aleatoria, pero NO en los números de Carmichael.

Los resultados de dos ejecuciones de dicha prueba se pueden encontrar en el apéndice E. Se observa un fenómeno curioso, por muchas veces que se ejecute el test, siempre se obtiene el mismo número de errores y en los mismos números. La pregunta es ¿cómo es posible, siendo el algoritmo aleatorio? La respuesta es bien simple, desconocemos como se ha implementado la aleatoriedad en el método. Lo más probable es que NO se cambie la semilla de aleatoriedad, por defecto, en cada ejecución y por lo tanto, bajo la misma compilación del ejecutable, se obtengan exactamente los mismo resultados. Para comprobar esto habría que estudiar en profundidad, de nuevo, la implementación del test en la librería.

4.3.5. Prueba de estrés al algoritmo de Miller-Rabin, otra vez

A raíz de los resultados del anterior experimento realizamos la modificación propuesta en el apéndice B.3. Pretendemos evaluar que impacto tiene el número de iteraciones del algoritmo de Miller-Rabin en la cantidad de errores obtenidos. De esta experiencia obtenemos como resultado una tabla con los fallos de realizar cien mil test sobre uno de los números problemáticos. Presentamos los datos en la siguiente gráfica, obtenida a partir de los datos mostrados en el apéndice E.

Se observa como la probabilidad de error cae de forma brutal al aumentar el número de iteraciones. Con cinco iteraciones se puede considerar la probabilidad de error como 0. Es importante tener en cuenta que un aumento en el número de iteraciones no tiene un impacto muy grande en el tiempo de cálculo (en nuestro caso es 0) y que "por defecto"se realizan diez iteraciones, un número más que suficiente para asegurar la primalidad a la vista de nuestros datos.

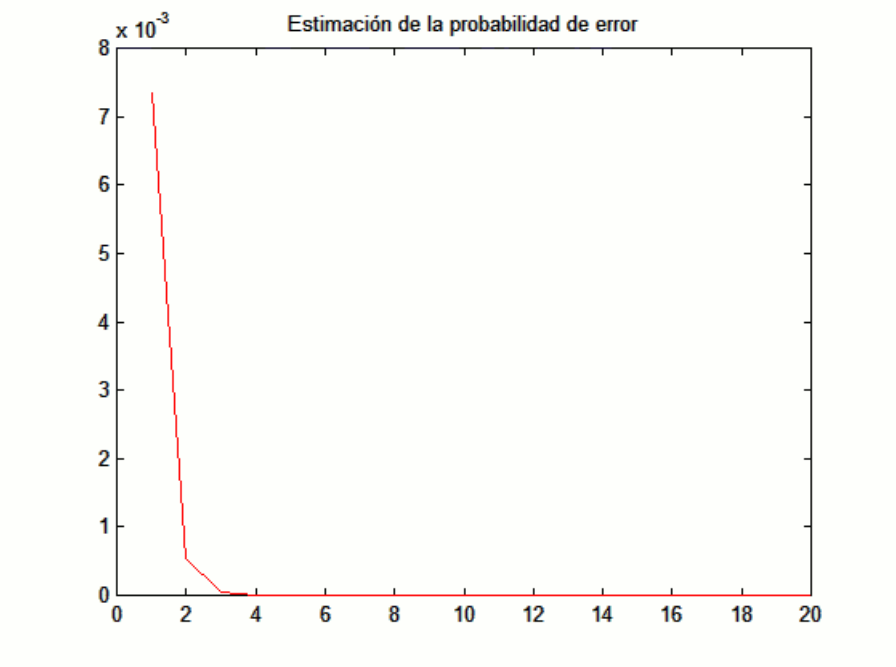

Figura 3: Estimación de la probabilidad de error.

4.4. Conclusiones derivadas de la implementación

A la vista de los resultados se observa que el algoritmo estándar no puede competir con los algoritmo probabilistas tipo Miller-Rabin, ni en tiempo, ni en efectividad.

Esto viene a reafirmar los argumentos que defienden la computación probabilista, pues es un claro ejemplo de efectividad de dichos métodos. Sin embargo, el algoritmo modificado se comporta de manera extraordinaria y es capaz de enfrentarse en igualdad de condiciones a los algoritmos probabilistas. Es necesario un mayor estudio de los posibles errores que se puede cometer al utilizar dicho algoritmo, o dar una demostración de la conjetura a fin de comprobar la efectividad del algoritmo.

A. Implementación del algoritmo en MAPLE

Presentamos aquí una pequeña implementación del algoritmo en el software comercial MAPLE.

```
AKS:=proc(n::posint)
#Rutina que comprueba si un numero es primo o es compuesto mediante
#el algoritmo conocido por AKS. Acepta como input un numero entero
#positivo y devuelve 0 si es compuesto o 1 si es primo.

#Variables locales
        local i::posint, aux::list, r::posint;

#i       contadores
#aux     lista donde poner la factorizacion
#j       polinomio (expresion)
#r       modulo sobre el que dividimos
#orden   guardo el orden

with(numtheory,order);

#Paso 1
i:=2:
while (ceil(log(n))> i) do
   aux:=factors(x^i-n) mod n: #factorizo.
   for j in aux[2] do
      if degree(j[1])=1 then RETURN (0); fi:
   end do: #compruebo si n=a^i, si hay factores de grado 1.
   i:=i+1:
end do: #si sale de aqui \nexists a,i tales que a^i=n.

#Paso 2
r:=2:
orden:=order(n,r); #Calculo el orden
if evalb(orden=FAIL) then orden:=0; fi;
#compruebo si hay divisores de 0
while (orden < round(4*(log(n))^2+1)) do
   r:=r+1;
   orden:=order(n,r);
   if evalb(orden=FAIL) then orden:=0; fi;
end do; #obtengo r tal que o_r(n) > 4log^2(n)

#Paso 3
for i from 2 to r do
   if evalb(gcd(i,n)<>1) and i<n then RETURN(0); fi:
```

```
end do:

#Paso 4
if (n < r) then RETURN(1); fi:

#Paso 5
i:=1:
while (i < trunc(2*sqrt(r)*log(n))) do
    if evalb(modpol((x+i)^n-x^n-i , x^r-1,x,n)<>0) then RETURN(0); fi:
    i:=i+1:
end do: #compruebo (x+i)^n - x^n - i = 0 mod x^r - 1, n

#Paso 6
RETURN(1);

end proc;
```

A.1. Implementación alternativa del algoritmo en MAPLE

Modificación de la implementación del algoritmo en el software comercial MAPLE para intentar corregir los malos resultados prácticos obtenidos.

```
AKS:=proc(n::posint)
#Rutina que comprueba si un numero es primo o es compuesto mediante
#el algoritmo conocido por AKS. Acepta como input un numero entero
#positivo y devuelve 0 si es compuesto o 1 si es primo.

#Variables locales
        local i,k,r,aux_2::posint; aux::list;

#i      contadores
#aux    lista donde poner la factorizacion
#j      polinomio (expresion)
#r      modulo sobre el que dividimos
#orden  guardo el orden.

with(numtheory,order);

#Paso 1
i:=2:
while (ceil(log(n))> i) do
   aux:=factors(x^i-n) mod n: #factorizo.
   for j in aux[2] do
      if degree(j[1])=1 then RETURN (0); fi:
   end do: #compruebo si n=a^i, si hay factores de grado 1.
   i:=i+1:
end do: #si sale de aqui /∃a,i tales que a^i=n.

#Paso 2
r:=2:
orden:=order(n,r); #calculo el orden
if orden=FAIL then orden:=0; fi; #compruebo si hay divisores de 0
while (orden < round(4*(log(n))^2+1)) do
   r:=r+1;
   orden:=order(n,r);
   if orden=FAIL then orden:=0; fi;
end do; #obtengo r tal que o_r(n) > 4log^2(n)

#Paso 3
for i from 2 to r do
   if gcd(i,n)<>1 and i<n then RETURN(0); fi:
end do:
```

```
#Paso 4
if n < r then RETURN(1); fi:

#Paso 5
i:=1:
while (i < trunc(2*sqrt(r)*log(n))) do
    k:=1:
    pot[1]:=x+i;
    #compruebo si me paso en la siguiente iteracion
    while (2^k <= n) do
        k:=k+1:
        #elevo al cuadrado mod x^r-1,n
        pot[k]:=modpol(pot[k-1]*pot[k-1],x^r-1,x,n);
    end do; #en k va la posicion de la mayor potencia de 2 < n
            #k es la primera potencia mayor que 2
    aux_p:=pot[k];
    dif:=n-2^(k-1);

    while dif <> 0 do
        #hayo potencia a multiplicar
        aux_2:=trunc(Re(evalf(log[2](dif))));
        #completo el exponente
        aux_p:=modpol(aux_p*pot[aux_2+1],x^r-1,x,n);
        dif:=dif-2^aux_2;
    end do; # aux_p = (x+i)^n mod x^r-1,n
    if evalb(modpol(aux_p-x^n-i, x^r-1,x,n)<>0) then RETURN(0); fi:
    i:=i+1:
end do; #compruebo (x+i)^n - x^n - i = 0 mod x^r-1,n

#Paso 6
RETURN(1);

end proc;
```

B. Implementacion del algoritmo en C++ (bajo la libreria NTL)

Presentamos aqui una pequeña implementacion del algoritmo en C++ bajo la libreria NTL de Victor Shoup [74]. Dicha libreria se puede descargar de [71]. Existe codigo fuente para plataformas Unix y Windows. Se distribuye bajo la licencia GNU.

```
**********Fichero main.cpp**********
#include <cstdlib>
#include <iostream>
#include <NTL/ZZ.h> //libreria con numeros ZZ
#include "imp_C_O(12).h" //libreria con el AKS
#include <fstream> //escritura de archivos
NTL_CLIENT

int main(int argc, char *argv[])
//Programa que realiza un test de esfuerzo bruto. Preparado para
//realizar computacion GRID "manual". Testea la primalidad (mediante
//el algoritmo AKS y una implementacion de Miller-Rabin) de los
//numeros dados desde rango.txt e imprime los resultados en
//salida.txt.
//Con la variable opcion, tercer parametro leido de rango.txt,
//controlamos que version del algoritmo AKS deseamos ejecutar.
{
    ZZ a; //inicio
    ZZ i; //contador
    ZZ b; //final
    int opcion; //0 = algoritmo estandar, !=0 algoritmo modificado
    unsigned long j; //contador de primos
    short aux; //auxiliar

    ifstream entrada("rango.txt", ios::in); //abro fichero con rango
    entrada >> a >> b >> opcion; //leo limites
    entrada.close(); //cierro fichero

    ofstream salida;  // objeto de la clase ofstream
    salida.open("salida.txt",ios::out); //abro archivo

    clock_t total; //tiempo total
    clock_t t; //variable que controla el tiempo

    salida << "Tabla de tiempos AKS" << endl;
```

```
    j=0; //inicializo contador de primos
    total=clock(); //cronometro total

for (i=a;i <= b; i++)  //criba AKS
{
    t=clock(); //cronometro
    aux=AKS(to_ZZ(i),opcion); //auxiliar para contar la
                              // suma de los primos
    salida << i << " " << aux << " "//
       << (clock()-t)/(double)CLOCKS_PER_SEC << endl;
    j+=aux; //cuento el numero de primos
}
    salida << endl << "Tiempo total empleado = "//
       << (clock()-total)/(double)CLOCKS_PER_SEC << endl;
    salida << endl << "La cantidad de numeros primos en el intervalo ["//
       <<a<<","<<b<<"] es " << j << endl;

    salida << endl << "Tabla de tiempos Miller-Rabin test" <<endl;

    j=0; //inicializo contador de primos
    total=clock(); //cronometro total

for (i=1; i <=b; i++)  //criba probabilistica
{
    t=clock(); //cronometro
    aux=ProbPrime(to_ZZ(i), 1);
    salida << i << " " << aux << " "//
       <<(clock()-t)/(double)CLOCKS_PER_SEC << endl;
    j+=aux;
}
    salida << endl << "Tiempo total empleado = "//
       << (clock()-total)/(double)CLOCKS_PER_SEC << endl << endl;
    salida << endl << "La cantidad de numeros primos en el intervalo ["//
       <<a<<","<<b<<"] es " << j << endl;

    salida.close(); //cierro el fichero
} // main
************************************************************
******************Fichero imp_C_O(12).cpp*******************
// Cabeceras de la libreria con el algoritmo de primalidad AKS

#include <NTL/ZZ_pXFactoring.h> //libreria para factorizar en Z_p[x]
#include <NTL/ZZ_pEX.h> //libreria para computar en Z_p[x]/(x^r-1)
NTL_CLIENT
```

```
ZZ orden(ZZ n, ZZ r); //calcula el orden de n modulo r, o_r(n)
int AKS(ZZ n, int opcion); //test de primalidad AKS
*******************************************************
*******************Fichero imp_C_O(12).c*******************
//Libreria con la implementacion del AKS (test de primalidad)

#include <NTL/ZZXFactoring.h> //libreria para factorizar en Z_p[x]
#include <NTL/ZZ_pEX.h> //libreria para computar en Z_p[x]/(x^r-1)
NTL_CLIENT

long orden_G(ZZ n, long r)
//funcion que calcula el orden de n modulo r, o_r(n)
//!!!!!!! NO CONTROLA LA EXISTENCIA !!!!!!!!
//if (GCD(n,to_ZZ(r))!=1) ord=0; //compruebo si tiene orden
{
//declaracion de variables

   ZZ_p::init(to_ZZ(r));  //inicializo el modulo ZZ_p
         ZZ_p aux = to_ZZ_p(n); //auxiliar para ir calculando
                                //potencias inicializado a n
   long ord; //auxiliar para calcular el orden

   ord=1; //inicializo
   while (aux != 1)
   {
      mul(aux,aux,to_ZZ_p(n)); //aux = aux*n  mod r = n^ord
      ord++;
   }
   return (ord);
}

int AKS(ZZ n, int opcion)
//funcion que implementa un test deterministico de primalidad AKS.
//si opcion es igual a 0 se realiza el test estandar, para cualquier
//otro valor usamos la conjetura propuesta por AKS que hace el
//algoritmo O(log^3(n)).
//Devuelve 0 en caso de ser compuesto, 1 si es primo.
//Actualmente solo podemos trabajar hasta entero de tamano long
//por problemas con el exponente :(
{
//declaracion de variables

   unsigned long i,j; //contadores, pueden llegar a ser muy grande
```

```
unsigned long r; //modulo sobre el que dividimos
unsigned long ord; //auxiliar para guardar los ordenes
ZZ aux; //auxiliar para calculos en la parte de la conjetura
ZZX f; //polinomio a factorizar en paso 1.
ZZ_pX modulo; //modulo sobre el que trabajaremos en el paso 5
ZZ_pX pot1,pot2; //auxiliar para calcular las potencias del paso 5
ZZ_pXModulus Modulo; //variable en la que guardaremos precalculos
vec_pair_ZZX_long factores; //factorizacion, me la devuelve asi

//Paso 1
i=2;
while (i < trunc(log(n))+1)
{
   f=ZZX(i,1) - n; //polinomio para hallar la factorizacion, x^i-n
   ZZ c; //variable que contendra el mcd de los coeficiente
   factor(c,factores,f,0,0); //factorizo
   for (j = 1; j < factores.length()+1; j++)
   {
      if ((deg(factores(j).a) == 1) & (factores(j).b != i)) return 0;
   } //compruebo grado de los factores
   i++;
} //si sale de aqui /∃a,i tales que a^i=n.

//Paso 2
if (opcion==0)
{
   r=2;
   if (GCD(n,to_ZZ(r))!=1) ord=0; //compruebo si tiene orden
   else ord=orden_G(n,r);  //calculo
   while (ord < trunc(4*log(n)*log(n))+1)
   {
      r++;
      if (GCD(n,to_ZZ(r))!=1) ord=0;
      else ord=orden_G(n,r);
   } //obtengo r tal que o_r(n) > 4log^2(n)
} //algoritmo clasico.
else
{
   r=2;
   sqr(aux,n); //aux = n^2
   sub(aux,n,1); //aux = n^2 - 1
   while (divide(aux,r)) r++; //r = n^2 - 1?
}//usando conjetura para O(log^3(n))
```

```
    //Paso 3
    for (i=2; i < r; i++) if ((GCD(to_ZZ(i),n)!=1) & (i < n))   return 0;

    //Paso 4
    if (n < r) return 1;

    //Paso 5
    ZZ_p::init(n); //inicializo el modulo Z_p
    i=1;
    modulo=ZZ_pX(r, to_ZZ_p(1)); //x^r
    sub(modulo,modulo,1); //x^r - 1
    build(Modulo,modulo); //construyo precalculos para x^r - 1

    while (i < 2*SqrRoot(r)*log(n))
    {
        PowerXPlusAMod(pot1,to_ZZ_p(i),n,Modulo); // (x+i)^n mod x^r - 1,n
        PowerXMod(pot2,n,Modulo); //x^n mod x^r - 1,n
        if (!(IsZero((pot1 - pot2 - i) %Modulo))) return(0);
        i++; //compruebo (x+i)^n - x^n - i = 0 mod x^r - 1,n
    }
    //Paso 6
    return(1);
}
*********************************************
```

B.1. Búsqueda de errores en el algoritmo de Miller-Rabin

Modificación del archivo *main.c* de la implementación en C++ para realizar el test de fuerza bruta para la búsqueda de errores en el algoritmo de Miller-Rabin. Destacar que es absolutamente necesario haber modificado el archivo *rango.txt* con un número de Carmichael.

```
**********Fichero main.cpp**********
#include <cstdlib>
#include <iostream>
#include <NTL/ZZ.h> //libreria con numeros ZZ
#include "imp_C_O(12).h" //libreria con el AKS
#include <fstream> //escritura de archivos
NTL_CLIENT

int main()
//Programa que realiza un test de esfuerzo bruto. Preparado para
//realizar computacion GRID "manual". Testea la primalidad (mediante
//el algoritmo AKS y una implementacion de Miller-Rabin) de los
//numeros dados desde rango.txt e imprime los resultados en
//salida.txt.
{
    ZZ a; //inicio
    ZZ i; //contador
    ZZ b; //final
    unsigned long j; //contador de primos
    short aux; //auxiliar

    ifstream entrada("rango.txt", ios::in); //abro fichero con rango
    entrada >> a >> b; //leo limites
    entrada.close(); //cierro fichero

    ofstream salida;  // objeto de la clase ofstream
    salida.open("salida.txt",ios::out); //abro archivo

    clock_t total; //tiempo total
    clock_t t; //variable que controla el tiempo

for (i=1; i <=1000000; i++)  //criba probabilistica
{
    t=clock(); //cronometro
    aux=ProbPrime(to_ZZ(a), 1);
    salida << i << " " << aux << " "//
       <<(clock()-t)/(double)CLOCKS_PER_SEC << endl;
```

```
    j+=aux;
}
    salida << endl << "Tiempo total empleado = " //
      << (clock()-total)/(double)CLOCKS_PER_SEC << endl << endl;
    salida << endl << "La cantidad de numeros primos en el intervalo ["//
      <<a<<","<<b<<"] es " << j << endl;
    salida.close(); //cierro el fichero
} // main
*************************************************************
```

B.2. Búsqueda de errores en conjetura 1.26

Modificación del archivo *main.c* de la implementación en C++ para realizar el text de búsqueda de errores en la conjetura 1.26.

```
*******************Fichero main.cpp*******************
#include <cstdlib>
#include <iostream>
#include <NTL/ZZ.h> //libreria con numeros ZZ
#include "imp_C_O(12).h" //libreria con el AKS
#include <fstream> //escritura de archivos
NTL_CLIENT

int main(int argc, char *argv[])
//Programa que realiza un test de esfuerzo bruto. Preparado para
//realizar computacion GRID "manual". Testea la primalidad (mediante
//el algoritmo AKS y una implementacion de Miller-Rabin) de los
//numeros dados desde rango.txt e imprime los resultados en
//salida.txt.
{
    ZZ a; //inicio
    ZZ i; //contador
    ZZ b; //final
    unsigned long jA; //contador de primos AKS
    unsigned long jM; //contador de primos Miller-Rabin
    short auxA; //auxiliar AKS
    short auxM; //auxiliar Miller-Rabin

    ifstream entrada("rango.txt", ios::in); //abro fichero con rango
    entrada >> a >> b; //leo limites
    entrada.close(); //cierro fichero

    ofstream salida;  // objeto de la clase ofstream
    salida.open("salida.txt",ios::out); //abro archivo

    salida << "Tabla de tiempos AKS" << endl;

    jA=0; //inicializo contador de primos
    jM=0; //inicializo contador de primos

for (i=a;i <= b; i++)  //criba AKS
{
    auxA=AKS(to_ZZ(i),1); //auxiliar para no contar la suma de los primos
    auxM=ProbPrime(to_ZZ(i), 1);
    if (auxA != auxM)
```

```
    {
        salida << i << " " << auxA << " " << auxM << " " << //
        AKS(to_ZZ(i),0) << endl; //muestro quien falla
    }
    jA+=auxA; //cuento el numero de primos
    jM+=auxM; //cuento el numero de primos
}
    salida << endl << "La cantidad de numeros primos en el intervalo ["//
        <<a<<","<<b<<"] es " << jA << "," << jM << endl;
    salida.close(); //cierro el fichero
} // main
*************************************************************
```

B.3. Búsqueda de errores en el algoritmo de Miller-Rabin, revisión

Modificación del archivo *main.c* de la implementación en C++ para realizar el test de fuerza bruta con el obejtivo de la búsqueda de errores en el algoritmo de Miller-Rabin y como afecta el número de iteraciones del método en la probabilidad de error. Destacar que es absolutamente necesario haber modificado el archivo *rango.txt* con un número problemático y el número máximo de iteraciones.

```
******************Fichero main.cpp******************
#include <cstdlib>
#include <iostream>
#include <NTL/ZZ.h> //libreria con numeros ZZ
#include "imp_C_O(12).h" //libreria con el AKS
#include <fstream> //escritura de archivos
NTL_CLIENT

int main()
//Programa que realiza una prueba sobre la mejora del
//la probabilidad de error del algoritmo de Miller-Rabin
//al aumentar el numero de iteraciones.
{
    ZZ a; //inicio
    ZZ i; //contador
    ZZ b; //final
    unsigned long j; //contador de primos
    unsigned long k; //contador
    short aux; //auxiliar

    ifstream entrada("rango.txt", ios::in); //abro fichero con rango
    entrada >> a >> b; //leo limites
    entrada.close();   //cierro fichero

    ofstream salida;   // objeto de la clase ofstream
    salida.open("salida.txt",ios::out); //abro archivo

for (k=1; k <= b; k++)
{
   j=0;
   for (i=1; i <=100000; i++)  //criba probabilistica
   {
      aux=ProbPrime(to_ZZ(a), k);
      j+=aux;
   }
```

```
    salida << k << " " << j << endl;
}
    salida.close(); //cierro el fichero
} // main
*************************************************************
```

C. Script en MatLab para el dibujado de gráficas

```
V=[9000     0   0.015000   0.031000   0.015000   0.000000   0.000000
  9001     1   9.172000  10.203000   9.281000   8.281000   8.792000
  9002     0   0.016000   0.000000   0.000000   0.016000   0.020000
  9003     0   0.000000   0.016000   0.016000   0.000000   0.010000
  9004     0   0.015000   0.016000   0.000000   0.015000   0.010000
  9005     0   0.016000   0.015000   0.016000   0.000000   0.010000
  9006     0   0.016000   0.000000   0.015000   0.016000   0.010000
  9007     1  18.296000  10.735000  10.266000   9.438000   9.624000
  9008     0   0.032000   0.015000   0.016000   0.000000   0.010000
  9009     0   0.015000   0.000000   0.000000   0.015000   0.010000
  9010     0   0.063000   0.032000   0.015000   0.031000   0.010000
  9011     1  17.109000   9.718000   9.219000   8.547000   8.612000
  9012     0   0.016000   0.016000   0.000000   0.000000   0.000000
  9013     1  17.172000   9.672000   9.875000   8.953000   8.613000
  9014     0   0.031000   0.015000   0.016000   0.000000   0.010000
  9015     0   0.016000   0.016000   0.015000   0.016000   0.010000
  9016     0   0.015000   0.000000   0.000000   0.000000   0.010000
  9017     0   0.031000   0.016000   0.031000   0.016000   0.010000
  9018     0   0.032000   0.015000   0.000000   0.000000   0.010000
  9019     0   0.015000   0.016000   0.016000   0.015000   0.010000
  9020     0   0.032000   0.016000   0.016000   0.000000   0.010000
  9021     0   0.015000   0.000000   0.015000   0.016000   0.020000
  9022     0   0.031000   0.015000   0.016000   0.000000   0.010000
  9023     0   0.016000   0.016000   0.000000   0.031000   0.010000
  9024     0   0.031000   0.015000   0.016000   0.000000   0.010000
  9025     0   0.016000   0.000000   0.000000   0.000000   0.000000
  9026     0   0.031000   0.016000   0.000000   0.016000   0.010000
  9027     0   0.016000   0.000000   0.015000   0.000000   0.010000
  9028     0   0.031000   0.016000   0.000000   0.000000   0.010000
  9029     1  16.641000   9.500000   9.485000   8.328000   8.432000
  9030     0   0.015000   0.015000   0.000000   0.000000   0.010000
  9031     0   0.032000   0.016000   0.015000   0.016000   0.010000
  9032     0   0.031000   0.000000   0.016000   0.000000   0.010000
  9033     0   0.015000   0.016000   0.015000   0.015000   0.010000
  9034     0   0.032000   0.015000   0.000000   0.016000   0.010000
  9035     0   0.031000   0.016000   0.016000   0.015000   0.020000
  9036     0   0.016000   0.015000   0.016000   0.000000   0.010000
  9037     0   0.031000   0.016000   0.000000   0.016000   0.010000
  9038     0   0.015000   0.016000   0.015000   0.000000   0.010000
  9039     0   0.032000   0.000000   0.000000   0.016000   0.010000
  9040     0   0.015000   0.015000   0.016000   0.000000   0.010000
  9041     1  15.813000   8.938000   9.078000   8.000000   7.962000
```

9042	0	0.015000	0.015000	0.000000	0.015000	0.010000
9043	1	15.907000	9.547000	9.063000	8.000000	8.001000
9044	0	0.015000	0.016000	0.000000	0.016000	0.010000
9045	0	0.032000	0.000000	0.015000	0.000000	0.020000
9046	0	0.015000	0.016000	0.000000	0.016000	0.010000
9047	0	0.031000	0.015000	0.016000	0.015000	0.010000
9048	0	0.016000	0.000000	0.015000	0.000000	0.010000
9049	1	17.047000	9.844000	9.766000	8.563000	8.653000
9050	0	0.016000	0.000000	0.000000	0.015000	0.010000
9051	0	0.015000	0.031000	0.016000	0.000000	0.010000
9052	0	0.031000	0.000000	0.015000	0.016000	0.010000
9053	0	0.016000	0.031000	0.000000	0.000000	0.010000
9054	0	0.016000	0.000000	0.016000	0.016000	0.010000
9055	0	0.031000	0.016000	0.016000	0.000000	0.010000
9056	0	0.016000	0.016000	0.000000	0.015000	0.010000
9057	0	0.031000	0.000000	0.015000	0.016000	0.010000
9058	0	0.015000	0.015000	0.000000	0.000000	0.010000
9059	1	15.938000	9.438000	9.078000	8.156000	8.022000
9060	0	0.016000	0.000000	0.000000	0.000000	0.010000
9061	0	0.046000	0.015000	0.016000	0.016000	0.010000
9062	0	0.016000	0.016000	0.000000	0.015000	0.010000
9063	0	0.016000	0.016000	0.016000	0.000000	0.010000
9064	0	0.015000	0.000000	0.015000	0.016000	0.010000
9065	0	0.032000	0.015000	0.016000	0.000000	0.010000
9066	0	0.015000	0.016000	0.000000	0.016000	0.010000
9067	1	16.860000	10.375000	9.062000	8.515000	8.492000
9068	0	0.031000	0.016000	0.000000	0.016000	0.010000
9069	0	0.031000	0.015000	0.016000	0.000000	0.010000
9070	0	0.016000	0.000000	0.016000	0.015000	0.010000
9071	0	0.031000	0.016000	0.000000	0.000000	0.010000
9072	0	0.016000	0.015000	0.015000	0.000000	0.010000
9073	0	0.015000	0.016000	0.016000	0.016000	0.010000
9074	0	0.031000	0.016000	0.000000	0.016000	0.010000
9075	0	0.016000	0.000000	0.015000	0.000000	0.010000
9076	0	0.031000	0.015000	0.000000	0.015000	0.010000
9077	0	0.032000	0.016000	0.016000	0.000000	0.020000
9078	0	0.015000	0.000000	0.016000	0.016000	0.010000
9079	0	0.016000	0.016000	0.000000	0.016000	0.010000
9080	0	0.015000	0.015000	0.015000	0.000000	0.010000
9081	0	0.032000	0.016000	0.000000	0.000000	0.010000
9082	0	0.015000	0.015000	0.016000	0.015000	0.010000
9083	0	0.032000	0.000000	0.016000	0.000000	0.010000
9084	0	0.015000	0.016000	0.000000	0.016000	0.010000
9085	0	0.016000	0.016000	0.015000	0.000000	0.010000

```
 9086    0   0.031000   0.015000   0.000000   0.015000   0.010000
 9087    0   0.016000   0.000000   0.016000   0.000000   0.010000
 9088    0   0.031000   0.016000   0.000000   0.016000   0.010000
 9089    0   0.016000   0.016000   0.015000   0.016000   0.020000
 9090    0   0.015000   0.015000   0.016000   0.000000   0.000000
 9091    1  16.813000  10.438000   8.984000   8.468000   8.412000
 9092    0   0.031000   0.015000   0.016000   0.000000   0.010000
 9093    0   0.016000   0.000000   0.000000   0.000000   0.010000
 9094    0   0.015000   0.016000   0.016000   0.016000   0.010000
 9095    0   0.031000   0.016000   0.015000   0.000000   0.020000
 9096    0   0.016000   0.015000   0.000000   0.016000   0.010000
 9097    0   0.031000   0.016000   0.016000   0.015000   0.010000
 9098    0   0.032000   0.015000   0.016000   0.016000   0.010000
 9099    0   0.015000   0.016000   0.000000   0.000000   0.010000];


j=0;
for i=1:100
    if V(i,2)==1;
        j=j+1;
        resul(1,j)=V(i,3);  %P4 No OP
        resul(2,j)=V(i,4);  %P4 OP
        resul(3,j)=V(i,5);  %K7 OP
        resul(4,j)=V(i,6);  %K8 OP
        resul(5,j)=V(i,7);  %PM OP
    end
end

x=[1:1:length(resul)];

plot(x,resul(1,:),'r-',x,resul(2,:),'b-',x,resul(3,:), %
    'g-',x,resul(4,:),'y-',x,resul(5,:),'m-');
legend('Pentium4 No Optimo', 'Pentium4', 'AthlonXP', %
    'AthlonXP-64', 'PentiumM')
```

D. Resultados de la verificación de la conjetura 1.26

Mostramos a continuación una tabla con los resultados de la prueba de la conjetura 1.26. La primera para una ejecución sobre los primeros diez mil números primos y la segunda sobre lo primeros veinte mil.

Número	AKS c	M-R	AKS	Número	AKS c	M-R	AKS
247	0	1	0	26599	0	1	0
1073	0	1	0	29341	1	0	0
1729	1	0	0	29891	0	1	0
2059	0	1	0	32551	0	1	0
2321	0	1	0	37787	0	1	0
2465	1	0	0	38503	0	1	0
2701	0	1	0	42127	0	1	0
2821	1	0	0	43739	0	1	0
3053	0	1	0	46657	1	0	0
5921	0	1	0	52633	1	0	0
6161	0	1	0	54097	0	1	0
8911	1	0	0	54511	0	1	0
9211	0	1	0	63973	1	0	0
13213	0	1	0	68101	0	1	0
13333	0	1	0	69721	0	1	0
14209	0	1	0	70673	0	1	0
14491	0	1	0	75409	0	1	0
14689	0	1	0	77963	0	1	0
15409	0	1	0	79501	0	1	0
15841	1	0	0	94657	0	1	0
22177	0	1	0				

Número	AKS c	M-R	AKS	Número	AKS c	M-R	AKS
247	0	1	0	43739	0	1	0
1073	0	1	0	46657	1	0	0
1729	1	0	0	52633	1	0	0
2059	0	1	0	54097	0	1	0
2321	0	1	0	54511	0	1	0
2465	1	0	0	63973	1	0	0
2701	0	1	0	68101	0	1	0
2821	1	0	0	69721	0	1	0
3053	0	1	0	70673	0	1	0
5921	0	1	0	75409	0	1	0
6161	0	1	0	77963	0	1	0
8911	1	0	0	79501	0	1	0
9211	0	1	0	94657	0	1	0
13213	0	1	0	105121	0	1	0
13333	0	1	0	111361	0	1	0
14209	0	1	0	115921	1	0	0
14491	0	1	0	117211	0	1	0
14689	0	1	0	120961	0	1	0
15409	0	1	0	126217	1	0	0
15841	1	0	0	128143	0	1	0
22177	0	1	0	147187	0	1	0
26599	0	1	0	153401	0	1	0
29341	1	0	0	162401	1	0	0
29891	0	1	0	167257	0	1	0
32551	0	1	0	172081	1	0	0
37787	0	1	0	182527	0	1	0
38503	0	1	0	188461	1	0	0
42127	0	1	0	190513	0	1	0

E. Estimación de la probabilidad de error en en algoritmo de Miller-Rabin

Mostramos a continuación una tabla con los resultados de la última experiencia en la que calculamos el número errores por iteracion que comete el algoritmo de Miller-Rabin

Número de iteraciones	Errores cometidos
1	7351
2	546
3	57
4	3
5	0
6	0
7	0
8	0
9	0
10	0
11	0
12	0
13	0
14	0
15	0
16	0
17	0
18	0
19	0
20	0

F. Ampliación del paquete *listings* perteneciente a LaTeX 2ε

Definición del lenguaje de programación MAPLE para el paquete *listings* de Carsten Heinz [75] perteneciente a LaTeX 2ε. Desarrollado para este trabajo pues dicho lenguaje no estaba soportado en la actual versión y era necesario para la correcta presentación de los anexos anteriores.

```
\lstdefinelanguage{MAPLE}%
   %keywords
   {morekeywords=[1]{and, assuming, break, by, catch,%
      description, do, done, elif, else, end, error,%
      export, fi, finally, for, from, global, if, implies,%
      in, intersect, local, minus, mod, module, next, not,%
      od, option, options, or, proc, quit, read, return,%
      save, stop, subset, then, to, try, union, use, while, xor},%
   %booleans, conditionals, operators, repeat-logic, etc
   morekeywords=[2]{Order, fail, options, read, save, break, local,%
      point, remember, stop, done, mod, proc, restart, with, end, mods,%
      quit, return, error, next},%
   %funtions
   morekeywords=[3]{about, abs, add, addcoords,%
      additionally, addproperty, addressof, AFactor,%
      AFactors, AIrreduc, AiryAi, AiryAiZeros,%
      AiryBi, AiryBiZeros, algebraic, algsubs,%
      alias, allvalues, anames, AngerJ,%
      antihermitian, antisymm, apply, applyop,%
      applyrule, arccos, arccosh, arccot,%
      arccoth, arccsc, arccsch, arcsec,%
      arcsech, arcsin, arcsinh, arctan,%
      arctanh, argument, Array, array,%
      ArrayDims, ArrayElems, ArrayIndFns, ArrayOptions,%
      assign, assigned, asspar, assume,%
      assuming, asympt, attributes, band,%
      Berlekamp, bernoulli, bernstein, BesselI,%
      BesselJ, BesselJZeros, BesselK, BesselY,%
      BesselYZeros, Beta, branches, C,%
```

```
cat, ceil, changecoords, charfcn,%
ChebyshevT, ChebyshevU, CheckArgs, Chi,%
chrem, Ci, close, coeff,%
coeffs, coeftayl, collect, combine,%
comparray, compiletable, compoly, CompSeq,%
conjugate, constant_indfcn, Content, content,%
convergs, convert, coords, copy,%
CopySign, cos, cosh, cot,%
coth, coulditbe, csc, csch,%
csgn, currentdir, curry, CylinderD,%
CylinderU, CylinderV, D, dawson,%
Default0, DefaultOverflow, DefaultUnderflow, define,%
define_external, degree, denom, depends,%
DESol, Det, diagonal, Diff,%
diff, diffop, Digits, dilog,%
dinterp, Dirac, disassemble, discont,%
discrim, dismantle, DistDeg, Divide,%
divide, dsolve, efficiency, Ei,%
Eigenvals, eliminate, ellipsoid, EllipticCE,%
EllipticCK, EllipticCPi, EllipticE, EllipticF,%
EllipticK, EllipticModulus, EllipticNome, EllipticPi,%
elliptic_int, entries, erf, erfc,%
erfi, euler, eulermac, Eval,%
eval, evala, evalapply, evalb,%
evalc, evalf, evalfint, evalhf,%
evalm, evaln, evalr, evalrC,%
events, Excel, exists, exp,%
Expand, Expand, expand, expandoff,%
expandon, exports, extract, extrema,%
Factor, factor, Factors, factors,%
fclose, fdiscont, feof, fflush,%
FFT, filepos, fixdiv, float,%
floor, fnormal, fold, fopen,%
forall, forget, fprintf, frac,%
freeze, frem, fremove, FresnelC,%
Fresnelf, Fresnelg, FresnelS, FromInert,%
frontend, fscanf, fsolve, galois,%
```

```
GAMMA, GaussAGM, Gausselim, Gaussjord,%
gc, Gcd, gcd, Gcdex,%
gcdex, GegenbauerC, genpoly, getenv,%
GetResultDataType, GetResultShape, GF, Greek,%
HankelH1, HankelH2, harmonic, has,%
hasfun, hasoption, hastype, heap,%
Heaviside, Hermite, HermiteH, hermitian,%
Hessenberg, hfarray, history, hypergeom,%
icontent, identity, IEEEdiffs, ifactor,%
ifactors, iFFT, igcd, igcdex,%
ilcm, ilog10, ilog2, ilog[b],%
Im, implicitdiff, ImportMatrix, ImportVector,%
indets, index, indexed, indices,%
inifcn, ininame, initialcondition, initialize,%
insert, int, intat, interface,%
Interp, interp, Inverse, invfunc,%
invztrans, iostatus, iperfpow, iquo,%
iratrecon, irem, iroot, Irreduc,%
irreduc, is, iscont, isdifferentiable,%
IsMatrixShape, isolate, isolve, ispoly,%
isprime, isqrfree, isqrt, issqr,%
ithprime, JacobiAM, JacobiCD, JacobiCN,%
JacobiCS, JacobiDC, JacobiDN, JacobiDS,%
JacobiNC, JacobiND, JacobiNS, JacobiP,%
JacobiSC, JacobiSD, JacobiSN, JacobiTheta1,%
JacobiTheta2, JacobiTheta3, JacobiTheta4, JacobiZeta,%
KelvinBei, KelvinBer, KelvinHei, KelvinHer,%
KelvinKei, KelvinKer, KummerM, KummerU,%
LaguerreL, LambertW, last_name_eval, latex,%
latex_filter, lattice, lcm, Lcm,%
lcoeff, leadterm, LegendreP, LegendreQ,%
length, LerchPhi, lexorder, lhs,%
Li, Limit, limit, Linsolve,%
ln, lnGAMMA, log, log10,%
LommelS1, LommelS2, lprint, map,%
map2, Maple_floats, match, MatlabMatrix,%
Matrix, matrix, MatrixOptions, max,%
```

```
maximize, maxnorm, maxorder, MeijerG,%
member, min, minimize, mkdir,%
ModifiedMeijerG, modp, modp1, modp2,%
modpol, mods, module, MOLS,%
msolve, mtaylor, mul, NextAfter,%
nextprime, nops, norm, Normal(inert),%
Normal, normal, nprintf, Nullspace,%
numboccur, numer, NumericClass, NumericEvent,%
NumericEventHandler, NumericException, numerics, NumericStatus,%
numeric_type, odetest, op, open,%
order, OrderedNE, parse, patmatch,%
pclose, PDEplot_options, pdesolve, pdetest,%
pdsolve, piecewise, plot, plot3d,%
plotsetup, pochhammer, pointto, poisson,%
polar, polylog, polynom, Power,%
Powmod, powmod, Prem, prem,%
Preprocessor, prevprime, Primitive, Primpart,%
primpart, print, printf, ProbSplit,%
procbody, ProcessOptions, procmake, Product,%
product, proot, property, protect,%
Psi, psqrt, queue, Quo,%
quo, radfield, radnormal, radsimp,%
rand, randomize, Randpoly, randpoly,%
Randprime, range, ratinterp, rationalize,%
Ratrecon, ratrecon, Re, readbytes,%
readdata, readlib, readline, readstat,%
realroot, Record, Reduce, references,%
Regular_Expressions, release, Rem, rem,%
remove, repository, requires, residue,%
RESol, Resultant, resultant, rhs,%
rmdir, root, rootbound, RootOf,%
Roots, roots, round, Rounding,%
rsolve, rtable, rtable_algebra, rtable_dims,%
rtable_elems, rtable_indfns, rtable_options, rtable_printf,%
rtable_scanf, SampleRTable, savelib, scalar,%
Scale10, Scale2, scan, scanf,%
Searchtext, searchtext, sec, sech,%
```

```
   select, selectfun, selectremove, seq,%
   series, setattribute, SFloatExponent, SFloatMantissa,%
   shake, Shi, showprofile, showtime,%
   Si, sign, signum, Simplify,%
   simplify, sin, singular, sinh,%
   sinterp, smartplot3d, Smith, solve,%
   solvefor, sort, sparse, spec_eval_rules,%
   spline, spreadsheet, SPrem, sprem,%
   sprintf, Sqrfree, sqrfree, sqrt,%
   sscanf, Ssi, ssystem, storage,%
   string, StruveH, StruveL, sturm,%
   sturmseq, subs, subsindets, subsop,%
   substring, subtype, Sum, sum,%
   surd, Svd, symmdiff, symmetric,%
   syntax, system, table, tan,%
   tanh, taylor, testeq, testfloat,%
   TEXT, thaw, thiele, time,%
   timelimit, ToInert, TopologicalSort, traperror,%
   triangular, trigsubs, trunc, type,%
   typematch, unames, unapply, unassign,%
   undefined, unit, Unordered, unprotect,%
   update, UseHardwareFloats, userinfo, value,%
   Vector, vector, verify, WeberE,%
   WeierstrassP, WeierstrassPPrime, WeierstrassSigma,%
   WeierstrassZeta, whattype, WhittakerM, WhittakerW, with,%
   worksheet, writebytes, writedata, writeline,%
   writestat, writeto, zero, Zeta,%
   zip, ztrans},
%constants
morekeywords=[3]{Catalan,I,gamma,infinity,FAIL,Pi},
%surface types
morekeywords=[4]{algebraic, anything, array, boolean,%
   equation, even, float, fraction, function, indexed,%
   integer, laurent, linear, list, listlist, logical,%
   mathfunc, matrix, `module`, moduledefinition, monomial,%
   name, negative, nonneg, numeric, odd, point, positive,%
   procedure, radical, range, rational, relation, RootOf,%
```

```
   rtable, scalar, series, set, sqrt, square, string,%
   table, taylor, trig, type, uneval, vector, zppoly},
%nested types
morekeywords=[5]{algfun, algnum, constant, cubic, expanded,%
   polynom, linear, quadratic, quartic, radnum, radfun, ratpoly},
%other types
morekeywords=[6]{abstract_rootof, algebraic, algext,%
   algfun, algnum, algnumext, And,%
   anyfunc, anyindex, anything, arctrig,%
   Array, array, atomic, attributed,%
   boolean, BooleanOpt, builtin, complex,%
   complexcons, constant, cubic, cx_infinity,%
   dependent, dimension, disjcyc, embedded_axis,%
   embedded_imaginary, embedded_real, equation, even,%
   evenfunc, expanded, exprseq, extended_numeric,%
   extended_rational, facint, filedesc, finite,%
   float, float[], fraction, freeof,%
   function, global, hfarray, identical,%
   imaginary, indexable, indexed, indexedfun,%
   infinity, integer, intersect, last_name_eval,%
   laurent, linear, list, listlist,%
   literal, local, logical, mathfunc,%
   Matrix, matrix, minus, module,%
   moduledefinition, monomial, MVIndex, name,%
   negative, negint, negzero, neg_infinity,%
   NONNEGATIVE, nonnegative, nonnegint, nonposint,%
   nonpositive, nonreal, Not, nothing,%
   numeric, odd, oddfunc, operator,%
   Or, patfunc, patindex, patlist,%
   Point, point, polynom, posint,%
   positive, poszero, pos_infinity, prime,%
   procedure, protected, quadratic, quartic,%
   Queue, radalgfun, radalgnum, radext,%
   radfun, radfunext, radical, radnum,%
   radnumext, Range, range, rational,%
   ratpoly, ratseq, realcons, real_infinity,%
   relation, RootOf, rtable, satisfies,%
```

```
   scalar, sequential, series, set,%
   sfloat, SimpleStack, specfunc, specified_rootof,%
   specindex, sqrt, stack, string,%
   subset, suffixed, symbol, symmfunc,%
   table, tabular, taylor, TEXT,%
   trig, truefalse, type, typefunc,%
   typeindex, undefined, uneval, union,%
   unit, unit_name, Vector, vector,%
   verification, verify, with_unit},
morecomment=[l]\#,%
morestring=[b]"
}[keywords,comments,strings]
```

Referencias

[**Algoritmos básicos, teoremas clásicos y clases de complejidad.**]

[1] Joachim von zur Gathen and Jürgen Gerhard *"Modern Computer Algebra"* . Cambridge University Press (1999).

[2] Christos H. Papadimitriou. *"Computacional Complexity"* . Addison-Wesley (1994).

[3] H.Cohen. *"A Course in Computational Algebraic Number Theory"*. Springer-Verlag (1993).

[4] P. Bürgisser, M. Clausen y M. Amin Shokrollahi. *"Algebraic Complexity Theory"*. Springer (1997).

[5] `http://en.wikipedia.org/wiki/Complexity_class`

[6] `http://en.wikipedia.org/wiki/Proofs_of_Fermat%27s_little_theorem`

[7] S. Singh. *"El enigma de Fermat"*. Planeta (2003).

[8] `http://en.wikipedia.org/wiki/Exponentiating_by_squaring`

[9] M. Nai *"On Chebyshev-type inequalities for primes"* . Amer. Math. Monthly. 89:126-129 (1982).

[10] `http://es.wikipedia.org/wiki/Teorema_de_los_n%C3%BAmeros_primos`

[**Hipótesis y conjeturas.**]

[11] `http://www.claymath.org/millennium/Riemann_Hypothesis/`

[12] `http://www.claymath.org/millennium/P_vs_NP/`

[13] `http://es.wikipedia.org/wiki/Hip%C3%B3tesis_de_Riemann`

[14] `http://en.wikipedia.org/wiki/Artin_conjecture`

[15] `http://en.wikipedia.org/wiki/Sophie_Germain_prime`

[16] `http://es.wikipedia.org/wiki/http://es.wikipedia.org/wiki/Conjetura_de_los_n%C3%BAmeros_primos_gemelos`

[17] `http://numbers.computation.free.fr/Constants/constants.html`

[**Test de primalidad, información básica.**]

[18] http://wikisource.org/wiki/Prime_numbers_%282_-_20000%29

[19] http://es.wikipedia.org/wiki/Test_de_primalidad

[20] http://www.utm.edu/research/primes/index.html

[21] http://cr.yp.to/primetests.html

[22] V. Pratt *"Every Prime Has a Succinct Certificate"*. SIAM J. Comput. 4:214-220 (1975).

[23] http://mathworld.wolfram.com/PrattCertificate.html

[**Polinomios de Matijasevic.**]

[24] Y. Matijasevic. *"A Diophantine representation of the set of prime numbers"* Dokl. Akad Nauk SSSR 196:770-773 (1971)

Traducido del ruso por R. N. Goss en Soviet Math. Dokl. 12:354-358 (1971)

[25] http://primes.utm.edu/glossary/page.php?sort=MatijasevicPoly

[26] http://primes.utm.edu/glossary/page.php?sort=FormulasForPrimes

[27] P. Ribrnboim. *"The Little Book of Big Primes"* Springer-Verlag (1991).

[28] G. H. Hardy y E.M. Wright. *"An Introduction to the Theory of Number"*. Oxford (1975).

[29] http://logic.pdmi.ras.ru/~yumat/Journal/jcontord.htm

[30] http://logic.pdmi.ras.ru/~yumat/JRobinson/Jpublications.html

[31] http://en.wikipedia.org/wiki/Hilbert%27s_problems#Status_of_the_problems

[**Test de Fermat.**]

[32] http://en.wikipedia.org/wiki/Fermat_primality_test

[33] http://arxiv.org/pdf/math.NT/9803082

[34] http://en.wikipedia.org/wiki/Carmichael_number

[35] W. R. Alford, A. Granville and C. Pomerance. *"There are Infinitely Many Carmichael Numbers"*. Ann. Math. 139:703-722 (1994)

[**Test de Solovay-Strassen.**]

[36] `http://en.wikipedia.org/wiki/Solovay-Strassen_primality_test`

[37] R. Solovay y V. Strassen. *"A fast Monte-Carlo test for primality"*. SIAM J.Comp. 6:84-86 (1977).

[**Test de Miller-Rabin.**]

[38] `http://en.wikipedia.org/wiki/Miller-Rabin_primality_test`

[39] `http://mathworld.wolfram.com/Rabin-MillerStrongPseudoprimeTest.html`

[40] `http://mathworld.wolfram.com/LucasPseudoprime.html`

[41] M. O. Rabin. *"Probabilistic algorithm for testing primality"*. J. Number Theory. 12:128-138 (1980).

[42] G.L. Miller. *"Riemann's hypothesis and test for primality"*. J. CSS. 13:300-317 (1976).

[**Test APRCL.**]

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

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

[45] W. Bosma y M.-P. van der Hulst. *"Primality proving with cyclotomy"*. Tesis, Universidad de Amsterdam, 1990.

[**Curvas elípticas.**]

[46] `http://www.lix.polytechnique.fr/~morain/Prgms/ecpp.english.html`

[47] L. Adleman y M. Huang. *"Recognizing primes in Random polinomial Time"*. 19*th* ACM Symp. on Theory of Computing 462-470 (1987).

[48] A. O. L. Atkin and F. Morain, *"Elliptic curves and primality proving"*. Math. Comp. 61:203 29-68 (1993)

[**Test AKS.**]

[49] http://en.wikipedia.org/wiki/AKS_primality_test

[50] http://www.cse.iitk.ac.in/news/primality_v3.pdf

[51] A. Granville. *"It is easy to determine whether a given integer is Prime"* . A.M.S. 42:3-38 (2004).

[52] F. Bornemann. *"PRIMES Is in P: A Breakthrough for "Everyman"¨*. Notices of the AMS. 50:545-552 (2003)

Traducido por Harold P. Boas.

[53] http://arxiv.org/abs/math/0211334

[54] http://www.cs.ou.edu/%7Eqcheng/paper/aksimp.pdf

[55] http://cr.yp.to/primetests/quartic-20041203.pdf

[**Implementaciones.**]

[56] http://www.ellipsa.net/

[57] http://fatphil.org/maths/AKS/#Implementations

[58] http://en.wikipedia.org/wiki/Simultaneous_multithreading

[59] http://www.bloodshed.net/index.html

[60] http://www.mingw.org/download.shtml

[61] http://www23.tomshardware.com/index.html

[62] http://setiathome.ssl.berkeley.edu/

[63] http://grid.ifae.es/datagrid/

[64] http://eu-datagrid.web.cern.ch/eu-datagrid/

[**Paquetes de cálculo simbólico y librerías.**]

[65] http://swox.com/gmp/index.orig.html

[66] http://research.mupad.de/home.html

[67] http://www.math.u-psud.fr/~belabas/pari/

[68] http://magma.maths.usyd.edu.au/magma/

[69] http://www.informatik.tu-darmstadt.de/TI/LiDIA/

[70] http://www.maplesoft.com/

[71] http://www.shoup.net/ntl/download.html

[72] http://www.mathworks.com/

[73] http://www.wolfram.com/

[**Colaboradores y agradecimientos.**]

[74] mailto:victor@shoup.net

[75] mailto:cheinz@gmx.de

[76] http://www.astroteamrg.net/foro/

[77] http://www.gnu.org

[78] http://www.miktex.org/

[79] http://www.texniccenter.org

[80] http://www.mozilla.org/