PRIMALIDAD:
de Eratóstenes a Agrawal-Kayal-Saxena

Claudio Gutiérrez
Depto. de Ciencias de la Computación,
Universidad de Chile,
Blanco Encalada 2120, Santiago, Chile.
`cgutierr@dcc.uchile.cl`

Noviembre 2002

Índice General

NOTA: Estos apuntes presentan una introducción al tema poniendo énfasis en el algoritmo de Agrawal-Kayal-Saxena (que sigue la presentación original y el artículo de F. Morain. No presentan detalles de implementación pues nos ha preocupado fundamentalmente la exposición pedagógica. (Los detalles de implementación pueden consultarse especialmente en el libro de Cohen citado en las referencias). Todos las demostraciones de resultados que no están demostrados en los apuntes están en las referencias.

1 Preliminares

1.1 Nociones básicas de Complejidad Computacional

Problemas y algoritmos. Un problema es una proposición dirigida a averiguar el modo de obtener un resultado cuando ciertos datos son conocidos. Los que aquí nos conciernen son problemas genéricos, aquellos que dependen de parámetros, y para los cuales queremos soluciones uniformes (un "algoritmo"), es decir, que sirvan para todos los parámetros de entrada. Un ejemplo es: "dado un sistema de ecuaciones lineales, determinar si tiene solución". Obsérvese que este enunciado es un conjunto infinito de problemas, uno por cada conjunto finito de números reales (los coeficientes de las ecuaciones). El *planteamiento* del problema (el cual generalmente identificaremos con el problema mismo) consta de:

1. *Entrada* (o instancia). El tipo de dato que necesitamos ingresar.

2. *Respuesta* (o salida). El tipo de dato que queremos obtener como respuesta.

Veamos algunos ejemplos, cuyo plantemiento nos servirá de modelo.

Problema 1 (PRIMALIDAD)
ENTRADA. *Un número natural n.*
RESPUESTA. *Sí o No, dependiendo si el número n tiene factores no triviales.*

Problema 2 (FACTORIZACIÓN)
ENTRADA. *Un número natural n.*
RESPUESTA. *La descomposición del número n en sus factores primos.*

La solución a un problema así planteado es un procedimiento, una prescripción exacta del orden determinado en que ha de ejecutarse un sistema de operaciones de tal forma que pudiéramos responder a cada una de las instancias del problema.[1] Es lo que se conoce como *algoritmo* que resuelve el problema en cuestión. El concepto de algoritmo es uno de los conceptos fundamentales de las matemáticas. Es un "procedimiento", una "prescripción" que "no requiere inventiva", que "puede hacerse mecánicamente". Uno de los logros de la lógica matemática del siglo XX es haber dado una definición formal de este concepto.[2]

Problemas insolubles. Hasta la década de los 30 había el consenso entre los matemáticos –aunque no lo decían de manera precisa– de que cualquier problema matemático que podía ser formulado de manera precisa podía ser resuelto. Esta creencia fue destruída por el matemático Kurt Gödel con su famoso teorema del año 1931. De acuerdo a éste, en todo sistema formal suficientemente expresivo existen sentencias imposibles de decidir. En este marco, un resultado importante de la teoría es que existen problemas que son imposibles

[1] Los problemas cuya respuesta es SI o NO (o "1" o "0", según la codificación) son llamados problemas de *decisión*. Un ejemplo de un problema de decisión es *Primalidad*.

[2] Hay muchas definiciones: máquinas de Turing, máquinas RAM, algoritmos de Markov, sistemas de Post, cálculo lambda, etc. Es posible demostrar que todos son equivalentes. Para nuestros efectos, el lector puede suponer que un algoritmo es un programa en un lenguaje de programación estándar como JAVA, FORTRAN, C, etc.

de resolver por algoritmos.[3] El caso más famoso –para matemáticos– de estos últimos es el problema décimo de Hilbert:

> "*Determinación de la solubilidad de una ecuación diofántica.* Dada una ecuación diofántica con cualquier número de variables desconocidas y con coeficientes racionales: encontrar un proceso acorde al cual pueda ser determinado en un número finito de operaciones si la ecuación es soluble en números enteros racionales."[4]

Y. Matiyasevich demostró en 1970 que este problema era insoluble. Pero este no es el único ejemplo. He aquí algunos otros problemas insolubles por algoritmos que tienen interés matemático (en corchetes quien demostró la insolubilidad).

Problema 3 (PROBLEMA DE LAS PALABRAS PARA GRUPOS [NOVIKOV, 1955])
ENTRADA. *$G, P, (w_1, w_2)$, donde G es el grupo definido por la presentación P (un conjunto finito de generadores X y un número finito de relaciones entre ellos), y w_1, w_2 son dos elementos de G.*
RESPUESTA. *Sí o No, dependiendo si $w_1 = w_2$ es válido en G.*

Problema 4 (HOMEOMORFISMO DE VARIEDADES 4-DIMENSIONALES [MARKOV, 1958])
ENTRADA. *Dos variedades 4-dimensionales.*
RESPUESTA. *Sí o No, dependiendo si son homeomorfas.*

Problema 5 (EXISTENCIA DE INTEGRAL INDEFINIDA ELEMENTAL [RICHARDSON, 1966])
ENTRADA. *Una función elemental f compuesta de números racionales, variable x, usando un número finito de veces la suma, resta, multiplicación, división, potencias, raíces y expresiones trigonométricas y sus inversas.*
RESPUESTA. *Sí o No, dependiendo si f tiene una integral indefinida que sea elemental.*

Algoritmos practicables e impracticables. La *solubilidad algorítmica* de un problema puede estar muy lejos de su solubilidad *práctica*. Un clásico ejemplo es el encontrar una estrategia ganadora en Ajedrez:

Problema 6 (AJEDREZ)
ENTRADA. *Una posición legal del juego de ajedrez.*
RESPUESTA. *SI o No, dependiendo si hay jugada ganadora para las blancas.*

---

[3] Para mostrar una solución a un problema dado se presenta un algoritmo. No es difícil en general darse cuenta si este procedimiento cumple con los requisitos pedidos. Lo que es bastante complicado es demostrar que para cierto problema *no existe ningún algoritmo* que lo resuelva. Y es aquí donde cobra importancia fundamental una definición formal de algoritmo.

[4] Una formulación más formal:
ENTRADA. Un polinomio de varias variables $P(x_1, \ldots, x_n)$ con coeficientes enteros.
RESPUESTA. Sí o No, dependiendo si la ecuación $P(x_1, \ldots, x_n) = 0$ tiene alguna raíz *entera*.
La versión original en alemán dice así: "Entscheidung der Lösbarkeit einer diophantische Gleichung. Eine diophantische Gleichung mit irgendwelchen Unbekannten und mit ganzen rationalen Zahlkoeffizienten sei vorgelegt: man soll ein Verfahren angeben, nach welchen sich mittels einer endlichen Anzahl von Operationen entscheiden lässt, ob die Gleichung in ganzen rationalen Zahlen lösbar ist."

Para este problema existe un algoritmo muy simple: la *estrategia minimax* [5], pero que es *impracticable* computacionalmente, es decir llegar a una solución tomaría demasiado tiempo, o nos ocuparía demasiado espacio (memoria) el desarrollarla. Decimos que es un problema difícil desde el punto de vista computacional.

La dificultad de un problema puede expresarse por la cantidad de recursos que son necesarios para resolverlo. La *teoría complejidad computacional* investiga cuantos recursos son necesarios para la solución de determinado problema. Los recursos tradicionalmente considerados más importantes son tiempo y espacio (memoria), que corresponden aproximadamente al número de operaciones elementales necesarios y a la memoria disponible en un computador. En estos apuntes nos preocuparemos sólo del tiempo, que para todos los efectos prácticos identificaremos con el "número de pasos elementales" que hace el algoritmo correspondiente.[6] Esta teoría permite comparar la dificultad de diferentes problemas y clasificarlos de acuerdo a ello. ¿Es más difícil multiplicar que dividir? ¿Es más simple encontrar las raíces de una ecuación algebraica que calcular el símbolo de Jacobi? ¿Cuanto "cuesta" invertir una matriz entera? La teoría de complejidad computacional da un marco teórico donde estas preguntas tienen perfecto sentido.

Complejidad de un algoritmo. El número de pasos elementales que toma un algoritmo determinado depende del tamaño de la entrada: toma más tiempo sumar dos números grandes que dos pequeños. En general en teoría de números la medida del tamaño de la entrada es el número de bits de los números correspondientes.[7] Por ejemplo, para la entrada a su tamaño es $\lg\lfloor a\rfloor + 1$. En general nos importará el comportamiento asintótico (ver sección sobre notación asintótica más adelante) del número de pasos. Así, decimos que el algoritmo de la suma toma $\mathcal{O}(n)$ número de pasos elementales, indicando con ello que para números de tamaño n (es decir de n bits) sumarlos toma n pasos elementales. El algoritmo de multiplicación corriente toma entonces $\mathcal{O}(n^2)$ número de pasos.

A cada algoritmo A que resuelve un problema P le asociaremos una función $t_{A,P} : \mathbb{N} \longrightarrow \mathbb{N}$ definida como el máximo número de pasos que usa A en instancias de P de tamaño n. Como hay un número finito de instancias de un tamaño fijo, $t_{A,P}(n)$ está bien definido. También supondremos que $t_{A,P}(n) \geq n$, es decir al menos usa el tiempo que demora en leer el dato de entrada.

[5] La estrategia minimax consiste en desarrollar *todas* las posibles variantes hasta llegar a posiciones de mate o tablas, las que las marcamos B, N, T, de acuerdo a quien gana o si es tablas. Una vez hecho esto, recursivamente nos devolvemos hacia la raíz marcando los nodos (posiciones) así: si juega blanco en esa posición (i.e. el nodo tiene distancia par a la raíz) lo marcamos B si tiene un hijo B, marcamos T si tiene un hijo T pero no hijos B, y de lo contrario lo marcamos N. Para los nodos en que juega negro (distancia impar a la raíz), marcamos N si tiene un hijo N, de lo contrario marcamos B. Si al final de este proceso la raíz, i.e. la posición de la cual partimos tiene B, blanco tiene estrategia ganadora, etc.

[6] Es necesario normalizar qué es "número de pasos elemental". Para esto usualmente se introduce un modelo normalizado de computador, la máquina de Turing o la RAM, y un paso elemental es una operación básica en esas máquinas. Para los efectos de estos apuntes esta falta de normalización no nos afectará mayormente, y en general consideraremos un paso elemental una operación aritmética elemental entre dos números con 1 bit (o decimal) cada uno.

[7] Si consideramos el número de decimales los resultados no varían más que en una constante cuyo valor aproximado es 3.

Estamos en condiciones de definir el concepto central en complejidad computacional, el de *clase de complejidad*. Supongamos que $f : \mathbb{N} \longrightarrow \mathbb{N}$ sea una función "razonable".[8] La clase $\mathrm{TIME}(f(n))$ es el conjunto de problemas P para los cuales hay un algoritmo A tal que $t_{A,P}(n) \in \mathcal{O}(f(n))$. Es decir, el conjunto de problemas P cuyas instancias de tamaño n son resolubles en tiempo asintóticamente menor que $f(n)$.[9]

La clase P. Hay consenso en la comunidad computacional que *tiempo polinomial* (es decir, tiempo acotado por una polinomio en el tamaño del dato de entrada) es una propiedad que se acerca a la *resolubilidad práctica*.[10] De hecho, la clase de complejidad más importante en la práctica es la de *tiempo polinomial*, denotada P, y definida como sigue:

$$\mathrm{P} = \bigcup_{k>0} \mathrm{TIME}(n^k).$$

Los algoritmos polinomiales son a menudo muy interesantes matemáticamente, debido a que son construidos con una profunda visión de de la estructura matemática de los problemas, y a menudo usan potentes herramientas matemáticas.

Comportamiento asintótico: la notación $\mathcal{O}$, Ω y Θ. Lo que nos interesa en la práctica no es una estimación precisa del tiempo que toma determinado algoritmo, sino una estimación asintótica.

Para cada función $g : \mathbb{N} \longrightarrow \mathbb{R}$ definimos las clases de funciones (con dominio los números naturales) $\mathcal{O}(g)$, $\Omega(g)$, $\Theta(g)$. Denotemos por $f : \mathbb{N} \longrightarrow \mathbb{R}$ una función arbitraria.

Decimos que $f(n) \in \mathcal{O}(g(n))$ si y sólo si existen $n_0 \in \mathbb{N}$ y $c > 0$ tal que para todo $n \geq n_0$:

$$0 \leq f(n) \leq cg(n),$$

es decir, asintóticamente $f(n)$ está acotada superiormente por $g(n)$ salvo una constante.

Similarmente, denotamos $f(n) \in \Omega(g(n))$ si y sólo si $g(n) \in \mathcal{O}(f(n))$ y $f(n) \in \Theta(g(n))$ si y sólo si $f(n) \in \mathcal{O}(g(n)) \cap \Omega(g(n))$, es decir los comportamientos asintóticos de $f(n)$ y $g(n)$ difieren sólo en una constante.

1.2 Complejidad de las operaciones fundamentales de la aritmética

Todas las operaciones aritméticas básicas son polinomiales: suma, resta, multiplicación y división de enteros con resto. (Recordemos que el largo de un entero n de entrada es

[8] Seguramente todas las funciones que imagine el lector intuitivamente son "razonables". Hacemos esta observación pues es posible definir funciones muy artificiosas que provocarían problemas con las definiciones que siguen.

[9] Usamos el inglés TIME pues está casi universalemente aceptado para señalar estas clases.

[10] Los argumentos son muchos. He aquí algunos: (1) Muchos algoritmos importantes en la práctica corren en tiempo polinomial. (2) Para exponentes k "razonables", esperar tiempo n^k es perfectamente posible incluso para entradas muy grandes; no ocurre lo mismo por ejemplo para tiempo 2^n, que a medida que crece el tamaño de la entrada n rápidamente se hace insostenible. (3) Las funciones polinomiales son una clase bastante robusta (por ejemplo, cerrada bajo suma, multiplicación y composición). (4) Todos los modelos de computación clásicos son simulables entre sí en tiempo polinomial.

el número de bits que se necesitan para representarle, es decir, $\log_2 n + \mathcal{O}(1)$). Aprendimos algoritmos polinomiales para todas estas operaciones en la escuela básica (algoritmos lineales en el caso de la suma y la resta, cuadráticos en el caso de la multiplicación y división). Consideraremos también la comparación de dos números como una operación aritmética básica y esto también puede hacerse en tiempo polinomial (lineal).

Exponenciación Claramente elevar a una potencia no es posible hacerlo en tiempo polinomial en la entrada pues nos tomaría tiempo exponencial en la entrada escribir el resultado. Afortunadamente lo que nos interesa en la mayoría de los casos es la exponenciación modular, que es un caso particular para el grupo $\mathbb{Z}/(m)$ del siguiente problema:

Problema 7 (EXPONENCIACIÓN)
ENTRADA. *Grupo* $(G, *)$*, elemento* $g \in G$ *y* $n \in \mathbb{Z}$.
RESPUESTA. g^n.

Casi en todos los algoritmos no triviales de teoría de números es necesario en algún momento calcular la potencia n de cierto elemento de un grupo, donde n puede ser muy grande (por ejemplo 10^{100}). Esto puede hacerse eficientemente (nótese que el método ingenuo requiere n multiplicaciones, esto es es exponencial en la entrada.)

La idea es la siguiente: Sea $(G, *)$ el grupo, y supongamos que queremos calcular g^n para algún $g \in G$ y $n > 0$ (la generalización a $n \in \mathbb{Z}$ es trivial). Si $n = \sum_i \epsilon_i 2^i$ es el desarrollo en base 2 de n, entonces

$$g^n = \prod_{e_i = 1} g^{2^i}.$$

Luego g^n puede calcularse por sucesivas elevaciones al cuadrado. ¿Cuantas? Aproximadamente $\log_2 n$.

Lema 1 *Sean a, b y m tres números naturales. Entonces $a^b (\bmod m)$ puede ser computado en tiempo polinomial, o más exactamente con $\mathcal{O}(\log m)$ operaciones aritméticas, sobre números naturales con $\mathcal{O}(\log m + \log a)$ dígitos.*

DEMOSTRACIÓN. Escribamos b en notación binaria:

$$b = 2^{r_1} + \cdots + 2^{r_k},$$

donde $0 \leq r_1 < \cdots < r_k$. Es obvio que $r_k < \log b$ y de aquí se sigue que $k \leq \log b$. Ahora bien, los números $a \bmod n$ para $0 \leq t \leq \log b$ son fácilmente calculables elevando al cuadrado repetidas veces, y luego multiplicamos aquellos k que juntos construyen a^b. Por supuesto realizamos todas las operaciones módulo m, es decir, luego de cada multiplicación además realizaremos una división con resto por m. $\odot$

Nota 1 *No se sabe si es posible calcular $a! \bmod m$ o $\binom{a}{b} \bmod m$ en tiempo polinomial.*

Detección de Potencias

Problema 8 (DETECCIÓN DE POTENCIAS)
ENTRADA. *Número natural n.*
RESPUESTA. *SI o No, dependiendo si hay números naturales a, k tal que $n = a^k$.*

Mostraremos aquí que es posible decidir este problema en tiempo polinomial. La idea es sencilla. Basta recorrer los números $p < \log n$, y en cada caso calcular el signo de $n - p^k$ para todos los enteros $k \leq \log n$ mediante aproximaciones. Este procedimiento, aunque toma tiempo polinomial (¿cuanto?: ejercicio) es ineficiente. De hecho, existen algoritmos para la verificación de potencias mucho más eficientes, aunque más complicados.

Algoritmo de Euclides Un algoritmo aritmético polinomial menos trivial es algoritmo de Euclides, que resuelve el siguiente problema

Problema 9 (MÁXIMO COMÚN DIVISOR)
ENTRADA. *Dos números enteros a y b.*
RESPUESTA. *El máximo común divisor de a y b, denotado $\text{mcd}(a, b)$.*

El algoritmo de Euclides funciona como sigue. Podemos suponer que a es más pequeño que b. Si $a = 0$ entonces el máximo común divisor de a y b es $\text{mcd}(a, b) = b$. Si $a > 0$ entonces, dividimos b por a con resto, que denotamos por r. Entonces $\text{mcd}(a, b) = \text{mcd}(a, r)$ y es suficiente entonces calcular el máximo común divisor entre a y r. Como $r < a$ la recurrencia terminará en un número finito de iteraciones y tenemos el máximo común divisor entre a y b.

Lema 2 *El algoritmo de Euclides toma tiempo polinomial. Más exactamente realiza $\mathcal{O}(\log a + \log b)$ operaciones aritméticas para la entrada (a, b).*

DEMOSTRACIÓN. Dado que $0 \leq r < a \leq b$, el algoritmo de Euclides terminará tarde o temprano. Veamos que lo hace en tiempo polinomial. Notemos que $b \geq a + r > 2r$ y además $r < b/2$. Luego $ar < ab/2$. Por lo tanto después de $\lceil \log ab \rceil$ iteraciones el producto de los dos números será menor que 1, es decir, uno de ellos es cero, lo que dice que el algoritmo termina. Cada iteración consiste en operaciones aritméticas básicas y se hace por lo tanto en tiempo polinomial. $\odot$

Otra importante cualidad del algoritmo de Euclides es que no sólo entrega el valor del máximo común divisor, sino que además es posible obtener dos enteros p, q tales que $\text{mcd}(a, b) = pa + qb$. Para esto simplemente tenemos que llevar el siguiente cálculo durante cada iteración. Si $a' = p_1 a + q_1 b$ y $b' = p_2 a + q_2 b$ y si dividimos b' por a' con resto r', es decir $b' = ha' + r'$, entonces

$$r' = (p_2 - hp_1)a + (q_2 - hp_2)b,$$

y así obtenemos la representación del nuevo número r' de la forma $p'a + q'b$.

Aritmética modular Las operaciones de suma, resta y multiplicación pueden también ser calculadas en tiempo polinomial en el escenario de la *aritmética modular* (anillo de las clases de restos módulo m). Representamos una clase de restos por el menor resto no-negativo que contiene. Realizamos la operación entre ellos como enteros y posteriormente realizamos división con resto por m.

Si m es número primo, entonces también podemos realizar la *división modular* en tiempo polinomial. ¡Esto no es lo mismo que división con resto! Dados números a, b y m tales que $0 \leq a, b \leq m-1$ y $b \neq 0$, podemos calcular un entero x con $0 \leq x < m$ tal que

$$bx \equiv a \bmod m.$$

(x es denotado a veces como $x \equiv a/b \bmod m$).

La solución es aplicar el algoritmo de Euclides para calcular el máximo común divisor entre b y m. Obviamente ya sabemos que el resultado es 1, pero lo que nos interesa son los números p, q tales que $bp + mq = 1$. En otras palabras $bp \equiv 1 \bmod m$, y por lo tanto $b(ap) \equiv a \bmod m$. Así, el cuociente x que buscamos se obtiene del resto ap después de dividirlo por m.

Teorema chino de los restos Mencionamos antes otra aplicación del algoritmo de Euclides. Supongamos que tenemos un número x desconocido, pero que conocemos sus restos $x_1, \ldots, x_k$ respecto de los módulos $m_1, \ldots, m_k$ que son primos relativos entre ellos. El *teorema chino del resto* dice que los restos determinan de manera única a x módulo el producto $m = m_1 \cdots m_k$. Pero ¿cómo podríamos computar ese resto?

Es suficiente considerar el caso $k = 2$, dado que para un k general el algoritmo se obtiene de éste por inducción. Estamos buscando un entero x tal que $x \leq x_1 \bmod m_1$ y $x \equiv x_2 \bmod m_2$ (además queremos que $0 \leq x \leq m_1 m_2 - 1$, pero esto puede ser logrado dividiendo por el resto al final).

En otras palabras , buscamos x, q_1, q_2 tales que $x = x_1 + q_1 m_1$ y $x = x_2 + q_2 m_2$. Restando, tenemos $x_2 - x_1 = q_1 m_1 - q_2 m_2$. Esta ecuación no determina de manera única a q_1 y q_2, pero eso no es importante. Usando el algoritmo de Euclides podemos encontrar q_1 y q_2 tales que

$$x_2 - x_1 = q_1 m_1 - q_2 m_2,$$

y se computa $x = x_1 + q_1 m_1 = x_2 + q_2 m_2$. Entonces $x \leq x_1 \bmod m_1$ y $x \equiv x_2 \bmod m_2$, como deseábamos.

Símbolo de Legendre-Jacobi-Kronocker El símbolo de Legendre $(\frac{a}{p})$ indica si a es resto cuadrático módulo el primo impar p, esto es, si la ecuación $x^2 \equiv a \pmod{p}$ tiene soluciones o no. Si tiene 2 soluciones, definimos $(\frac{a}{p}) = 1$; si tiene 1 solución, $(\frac{a}{p}) = 0$; si no tiene soluciones $(\frac{a}{p}) = -1$.

El símbolo de Jacobi $(\frac{a}{b})$ es una generalización del símbolo de Legendre para todo b entero. Definimos $(\frac{a}{0}) = 1$ si $|a| = 1$ y 0 en otros casos. Para $b = \Pi_i p_i \neq 0$, p_i primos (o -1 si b es negativo), definimos:

$$\left(\frac{a}{b}\right) = \prod_i \left(\frac{a}{p_i}\right),$$

donde $(\frac{a}{p_i})$ es el símbolo de Legendre cuando $p_i > 2$. Para $p_i = 2$ o $p_i = -1$ definimos:

$$\left(\frac{a}{2}\right) = \begin{cases} 0 & \text{si } a \text{ es par,} \\ (-1)^{(a^2-1)/8} & \text{si } a \text{ is impar.} \end{cases}$$

$$\left(\frac{a}{-1}\right) = \begin{cases} 1 & \text{si } a \geq 0, \\ -1 & \text{si } a < 0. \end{cases}$$

Problema 10 (Símbolo de Jacobi)
Entrada. *Enteros a, b.*
Respuesta. $(\frac{a}{b})$.

No es difícil delinear un algoritmo polinomial para calcular el símbolo de Jacobi a partir del siguiente lema. Dejamos la demostración del lema y el diseño del algoritmo como ejercicio para el lector (debiera correr en tiempo $O((\log a)(\log b))$.

Lema 3

1. *$(\frac{a}{b}) = 0$ si y sólo si $(a, b) \neq 1$.*
2. *Para todo a, b, c vale $(\frac{ab}{c}) = (\frac{a}{c})(\frac{b}{c})$*
3. *Para b fijo, el símbolo $(\frac{a}{b})$ es periódico en a. Tiene período b si $b \not\equiv 2(\bmod 4)$, y $4b$ en caso contrario.*
4. *Para $a \neq 0$ fijo, $(\frac{a}{b})$ es periódico en b. Tiene período $|a|$ si $a \equiv 0(\bmod 4)$, y $4|a|$ en caso contrario.*
5. *Para a, b impares positivos, vale la ley de reciprocidad $\left(\frac{a}{b}\right)\left(\frac{b}{a}\right) = (-1)^{(a-1)(b-1)/4}$, y $(\frac{-1}{b}) = (-1)^{(b-1)/2}$, y $(\frac{2}{b}) = (-1)^{(b^2-1)/8}$.*

1.3 Algunos resultados sobre números primos

Presentamos algunos teoremas y definiciones que necesitaremos más adelante. Por completitud añadimos algunos resultados de interés.

Prehistoria. El tema de los números primos no es nuevo, como muestran los siguientes resultados:

- $11, 13, 17, 19$ son números interesantes. (Africa Central, autor desconocido, cerca de 20.000 A.C.)
- Si p es primo, entonces $2^p - 2$ es divisible por p. (China, autor desconocido, aprox. 500 A.C.)
- Hay infinitos números primos. (Euclides, aprox. 300 A.C.) *Demostración.* Supongamos que no. Sea n el primo más grande. Considere el producto $n! + 1$. Todos los divisores de este número son mayores que n. Luego hay un primo mayor que n, contradicción.

Distribución de los números primos. ¿Cuan frecuentemente aparece los números primos en intervalos de números naturales? ¿Cómo están distribuídos? Si denotamos por $\pi(x)$ el número de primos menores o iguales que x, en 1852 Chebyshev demostró que $\pi(x) \in \Theta(\frac{x}{\log x})$. De hecho, es posible demostrar el siguiente resultado más preciso:[11]

Teorema 1 *Para $n \geq 1$ se cumple:*

$$\frac{n}{6 \log n} \leq \pi(n) \leq \frac{8n}{\log n}.$$

En algún momento necesitaremos un resultado similar sobre la distribución de cierto tipo de números primos. Denotemos por $P(n)$ el mayor divisor primo de n.

Teorema 2 (Fouvry; Baker, Harman) *El número de primos p menores que x tal que $P(p-1) > x^{\frac{2}{3}}$ es $\Theta(\frac{x}{\log x})$.*

El siguiente resultado lo usaremos en el algoritmo AKS.

Proposición 1 *Existen constantes positivas c_1, c_2 tal que existe un número primo r en el intervalo $[c_1(\log n)^6, c_2(\log n)^6]$ tal que $r-1$ tiene un factor primo $q \geq 4\sqrt{r}\log_2 n$ y que divide el orden de n módulo r.*

Demostración. Usando los Teoremas 1 y 2 no es difícil deducir (ejercicio) que existen c_1, c_2 tal que el número de primos r en $[c_1(\log n)^6, c_2(\log n)^6]$ tales que $P(r-1) > (c_2(\log n)^6)^{\frac{2}{3}} > r^{\frac{2}{3}}$ es mayor que $c\frac{(\log n)^6}{\log\log n}$ para una constante $c > 0$ y n suficientemente grande.

Sea $x = c_2(\log n)^6$ y consideremos el producto

$$\Pi = (n-1)(n^2-1)\cdots(n^{x^{\frac{2}{3}}}-1).$$

Usando el hecho de que este producto tiene a lo más $x^{\frac{2}{3}}\log n$ factores primos, y de que $x^{\frac{2}{3}}\log n < c\frac{(\log n)^6}{\log\log n}$, podemos concluir que existe al menos un primo especial, denotémoslo por r, que no divide el producto Π. Este es el primo requerido pues $r-1$ tiene un factor $q > r^{\frac{2}{3}} > 4\sqrt{r}\log n$ (el 4 resulta de suponer que $c_1 \geq 4^6$) y q divide al orden de n módulo r). $\odot$

2 Algoritmos para verificar Primalidad

Antes de entrar de lleno en el tema de primalidad, conviene destacar que los problemas aparentemente similares Primalidad (ver Problema 1) y Factorización (ver Problema 2) se comportan muy distintamente desde el punto de vista de la complejidad. Para

[11] Es lo que necesitaremos más adelante. Un resultado más fuerte aún es el celebrado teorema del número primo, demostrado por Hadamard y de la Vallée Poussin en 1896, que afirma que $\lim_{x\to\infty} \frac{\pi(x)\log x}{x} = 1$.

FACTORIZACIÓN no se conoce ningún algoritmo eficiente hoy día.[12] Por el contrario, hoy sabemos que PRIMALIDAD está en P.

En los días en que no se sabia si PRIMALIDAD era posible resolverlo eficientemente, se atacaba el problema de diferentes puntos de vista, comprometiendo algunos de los aspectos deseados de toda solución: eficiencia, generalidad y correctitud.

1. Comprometer eficiencia. Esto generalmente se produce cuando se ingresa información extra o se usan procedimientos poco eficientes.
2. Comprometer generalidad: Restringirse a números de ciertas formas específicas.
3. Comprometer correctitud: descansar en evidencia estadística. Es posible hacer esto de al menos un par de maneras:
   (a) Algoritmos Probabilísticos: sabremos que la salida será correcta con una alta probabilidad, pero no con certeza.
   (b) Algoritmos que usan conjeturas no demostradas: tendremos certeza sólo si la conjetura es cierta.

2.1 Métodos generales (e ineficientes)

Folklore Para cada $a \leq \sqrt{n}$ probar si a divide a n. Este método ingenuo tiene el defecto de tomar mucho tiempo: $\mathcal{O}(\sqrt{n})$ divisiones en el peor caso. Pero tiene otro defecto: si al terminar dictamina que n es primo nos quedamos sin ninguna *prueba* de que realmente lo es. Para verificar el resultado, debemos hacer todo el proceso de nuevo.

Criba de Eratóstenes (circa 250 A.C.) Este es un método para encontrar *todos* los primos menores que un número n. Funciona así:

1. Haga una lista de los enteros menores que n y mayores que 1.
2. Tome el primer número de la lista no tarjado o marcado y márquelo PRIMO.
3. Tarje todos los múltiplos del número anterior que aparezcan en la lista. Luego vuelva a 2.

Test de Wilson. Este es un test sencillísimo que usa el teorema de Wilson. El único problema es que no se conoce una forma eficiente de calcular $a!$ módulo n (Este es un problema abierto como mencionamos en la Nota 1).

Teorema 3 (Wilson) *n es primo si y sólo si* $(n-1)! \equiv -1 \bmod n$.

[12]Hablamos de computación clásica. Shor mostró en 1994 que en el marco de la computación cuántica es posible factorizar un número en tiempo polinomial. Hoy día (2002) este es un resultado que aún no tiene consecuencias prácticas.

Test $n-1$. Mostraremos un resultado muy interesante que determina la primalidad de n en tiempo polinomial si es que conocemos todos los divisores primos de $n-1$. Obviamente esto último hace que el test no tenga aplicación práctica general por la discusión que hicimos al comienzo de esta sección.

Teorema 4 (Lucas, Proth, Kraitchik, Lehmer) *Sea $n > 1$. Entonces n es primo si y sólo si existe a tal que $a^{n-1} \equiv 1 (\mathrm{mod}\, n)$ y $a^{(n-1)/q} \not\equiv 1 (\mathrm{mod}\, n)$ para todos los primos q que dividen $n-1$.*

Demostración. Sea n primo. Entonces un generador a de $\mathbb{Z}/(n)$ cumple las condiciones del teorema.

En la otra dirección, para mostrar que n es primo necesitamos demostrar sólo que $\varphi(n) = n - 1$, o más sencillamente, que $n-1$ divide a $\varphi(n)$.[13] Supongamos que esto no ocurre, es decir, que hay un primo q y un exponente $r > 0$ tal que q^r divide $n-1$ pero no $\phi(n)$. Para este primo q debe haber un entero a que satisfaga las condiciones del enunciado. Sea m el orden de a módulo n. Entonces m divide a $n-1$ (la primera condición), pero no a $(n-1)/q$ (la segunda condición). Luego q^r divide m que divide $\phi(n)$, una contradicción. ⊙

Nota 2 *La esencia de esta demostración es considerar el grupo $(\mathbb{Z}/(n))^*$. El tamaño de este grupo indica si n es primo (si $|(\mathbb{Z}/(n))^*| = n-1$ entonces n es primo.) Luego recoge información (las dos condiciones) para mostrar que el grupo tiene el tamaño correcto. Esta es la base de todos los tests de primalidad modernos. Lo que varía es el tipo de grupo con que se trabaja.*

Nota 3 *El Teorema 4 permite producir un "certificado" de que n es primo que es puede verificar eficientemente. El certificado consta del número a y de los primos q que dividen a $n-1$ (que son $\mathcal{O}(\log n)$). Claro que necesitamos también certificar que los q de nuestro certificado son primos. Para ello usamos este mismo método recursivamente. El lector puede comprobar que tanto el tamaño del certificado como el tiempo para su comprobación son polinomiales.*

2.2 Métodos para casos particulares

Encontrar algoritmos eficientes para tipos particulares de números tiene algunas ventajas. Por ejemplo para encontrar primos grandes. Entre los más conocidos están los números de la forma $F_k = 2^{2^k} + 1$ (*números de Fermat*), y los números de la forma $M_p = 2^p - 1$ (*números de Mersenne*).

Los dos teoremas siguientes (cuyas demostraciones pueden encontrarse en cualquier texto de la bibliografía) dan resultados que sin mayores complicaciones pueden transformarse en algoritmos eficientes. Para el test de Pepin hay que usar exponenciación modular y para el test de Lucas-Lehmer basta observar que la sucesión S_j tiene un largo $\mathcal{O}(\log n)$.

[13] Recordemos que $\varphi(n)$ es la función de Euler que indica la cantidad de números de $0, 1, \ldots, n-1$ que son primos con n. En particular $\varphi(p) = p - 1$ para todos los primos p.

Teorema 5 (Test de Pepin) *Para $k \geq 2$, F_k es primo si y sólo si* $5^{(F_k-1)/2} \equiv -1 (\bmod F_k)$.

Teorema 6 (Test de Lucas-Lehmer) *Sea $p > 2$ un primo y definamos $n = 2^p - 1$. Definamos $S_1 = 4$ y $S_{k+1} = S_k^2 - 2$ para $k \geq 1$. Entonces n is primo si y sólo si $S_{p-1} \equiv 0 (\bmod n)$.*

2.3 Métodos probabilísticos

Un algoritmo *probabilístico* (o *aleatorio*) es aquel cuya respuesta puede errar con cierta probabilidad. Los más comunes son aquellos que se equivocan "en un solo lado de la respuesta", es decir si responden afirmativamente lo hacen con certeza, y si responden negativamente, pueden equivocarse con alguna probabilidad.

Recordemos el siguiente teorema clásico:

Teorema 7 (Pequeño Teorema de Fermat) *Si p es primo y $\gcd(a,p) = 1$, entonces $a^{p-1} \equiv 1 (\bmod p)$.*

Demostración. Considere los enteros $a, 2a, \ldots, (p-1)a$. Observe que que ninguno de estos enteros es divisible por p y ningún par de ellos son congruentes entre ellos módulo p. Por lo tanto este es un sistema completo de residuos módulo p. Luego

$$a \cdot 2a \cdots (p-1)a \equiv 1 \cdot 2 \cdots (p-1) (\bmod p).$$

Por consiguiente $a^{p-1}(p-1)! \equiv (p-1)! (\bmod p)$, de donde es fácil deducir la afirmación del teorema. $\odot$

Si el recíproco de este teorema fuese cierto, tendríamos un interesante test de primalidad. Pero:

1. El recíproco es falso: Contraejemplo: $4^{14} \equiv 1 (\bmod 15)$, y 15 no es primo.

2. Peor aún: hay un número infinito de números compuestos n tal que $a^{n-1} \equiv 1 (\bmod n)$ para cada a con $\gcd(a,n) = 1$. Estos números se llaman *números de Carmichael.* El menor de ellos es $561 = 3 \cdot 11 \cdot 17$.

Decimos que la pareja (m,a) satisface la *condición de Fermat* si y sólo si $a^{m-1} \equiv 1 (\bmod m)$. Con esta notación, el Pequeño Teorema de Fermat dice que si m es primo entonces todos los números $a = 1, \ldots, m-1$ forman parejas (m,a) que satisfacen la condición de Fermat. Hagamos notar que verificar que una pareja (m,a) satisface o no la condición de Fermat es posible hacerlo en tiempo polinomial (¿por que?).

Ahora bien, supongamos que la siguiente afirmación es válida: "si m no es primo entonces la *mayoría* de los a violan la condición de Fermat." Entonces tendríamos las ideas esenciales para un algoritmo aleatorio polinomial para verificar si un número n es primo, que funcionaría así: *elegir a aleatoriamente y determinar si (n,a) satisface la condición de Fermat.* Si (n,a) no la satisface, entonces tenemos la certeza que n no es primo. Si (n,a) la satisface, con probabilidad de error menor que $1/2$ podemos afirmar que n es primo. Luego podemos repetir el procedimiento t para disminuir la probabilidad de error a menor que 2^{-t}.

Desgraciadamente la afirmación entre comillas no es cierta debido a la existencia de los números de Carmichael. Luego hay que refinar la idea.

Algoritmo de Solovay-Strassen. El algoritmo se basa en un importante teorema de Lehmer que caracteriza los números primos en función de algunas propiedades del grupo multiplicativo $(\mathbb{Z}/(n))^*$. Sea

$$E(n) = \{a \in (\mathbb{Z}/(n))^* : \left(\frac{a}{n}\right) \equiv a^{(n-1)/2} (\operatorname{mod} n)\}, \tag{1}$$

donde $(\frac{a}{n})$ es el símbolo de Jacobi.

Teorema 8 (Lehmer) *Para $n \geq 3$, n es primo si y sólo si $E(n) = (\mathbb{Z}/(n))^*$.*

Demostración. Si n es un primo impar, entonces $(\frac{a}{n}) \equiv a^{(n-1)/2} (\operatorname{mod} n)$ (Ejercicio).).

Para demostrar el recíproco, supongamos que $E(n) = (\mathbb{Z}/(n))^*$ pero que n es compuesto. Entonces

$$a^{n-1} \equiv \left(\frac{a}{n}\right)^2 \equiv 1 (\operatorname{mod} n)$$

para todo $a \in (\mathbb{Z}/(n))^*$. Luego n es Carmichael, luego n no puede tener cuadrados.[14] Luego podemos escribir $n = pr$, con p primo, $r > 1$ y $\operatorname{mcd}(p, r) = 1$. Sea g un no-residuo cuadrático módulo p y sea $a \equiv g (\operatorname{mod} p)$ y $a \equiv 1 (\operatorname{mod} r)$.

Usando el Lema 3 obtenemos

$$\left(\frac{a}{n}\right) = \left(\frac{a}{pr}\right) = \left(\frac{a}{p}\right)\left(\frac{a}{r}\right) = \left(\frac{g}{p}\right)\left(\frac{1}{r}\right) = (-1)(+1) = -1.$$

Puesto que supusimos que $(\frac{a}{n}) \equiv a^{(n-1)/2} (\operatorname{mod} n)$ para todo $a \in (\mathbb{Z}/(n))^*$ (recordar definición de $E(n)$ en Ec. 1) se debe tener que $a^{(n-1)/2} \equiv -1 (\operatorname{mod} r)$. Pero esto contradice $a \equiv 1 (\operatorname{mod} r)$. $\odot$

Nota 4 *Obsérvese que si n no es primo, entonces $E(n)$ es un subgrupo propio de $(\mathbb{Z}/(n))^*$, y luego $|E(n)| \leq |(\mathbb{Z}/(n))^*|/2$.*

Test de Solovay-Strassen. Elegir aleatoriamente un número a con $1 \leq a < n$.

- **if** $\gcd(a, n) \neq 1$ **return** COMPUESTO.
- **if** $(\frac{a}{n}) \not\equiv a^{(n-1)/2} (\operatorname{mod} n)$ **return** COMPUESTO.
- Si no, **return** PRIMO.

[14] No lo demostraremos aquí, pero los números de Carmichael son impares, libres de cuadrados y divisibles por al menos 3 primos distintos. La demostración puede encontrarse en las referencias.

Observe que si p no es primo, el algoritmo se equivocará con probabilidad menor o igual a $1/2$. Repitiendo el test varias veces esta probabilidad puede hacerse tan pequeña como se quiera.

El algoritmo es eficiente pues computar $a^{(n-1)/2}(\bmod n)$ toma sólo aproximadamente $(\lg n)^3$ pasos elementales.

Test de Rabin-Miller. Otra forma de mejorar el resultado de Fermat que evita el cálculo del símbolo de Jacobi, que aunque polinomial es costoso, es la siguiente idea de Miller.

Escribamos el número $m-1$ de la forma $2^k M$ donde M es impar. Diremos que a satisface la *condición de Miller* si al menos uno de los números

$$a^M - 1, a^M + 1, a^{2M} + 1, a^{4M} + 1, a^{2^{k-1}M} + 1$$

es divisible por m.

Nótese que el producto de estos números es $a^{m-1}-1$, luego todos los números que satisfacen la condición de Miller también satisfacen la condición de Fermat. Más aún, si m es primo, entonces m divide este producto, y por lo tanto divide a uno de estos factores, i.e., cada a $(1 \leq a < m)$ satisface la condición de Miller. Pero si m es compuesto, podría suceder que algunos a satisfagan la condición de Fermat pero no la de Miller (m puede ser un divisor de un producto sin ser divisor de ninguno de sus factores).

Es posible demostrar el siguiente resultado que entrega inmediatamente la base para un algoritmo aleatorio eficiente. (Nótese que verificar la condición de Miller puede hacerse eficientemente.)

Teorema 9 *Si m es un número compuesto, entonces al menos la mitad de los números $1, \ldots, m-1$ viola la condición de Miller.*

2.4 Hipótesis Extendida de Riemann y tests de primalidad.

La aleatoriedad en el test de Solovay-Strassen puede ser eliminada usando la siguiente conjetura que se puede deducir de la Hipótesis Extendida de Riemann.[15]

Conjectura. Si n es un número impar compuesto, entonces existe un entero positivo $a \leq 2(\lg n)^2$ para el cual, o bien (1) $\gcd(a,n) \neq 1$, o bien (2) $(\frac{a}{n}) \not\equiv a^{(n-1)/2}(\bmod n)$.

La conjetura dice que el conjunto de a's para el cual es necesario verificar la condición del test de Solovay-Strassen es pequeño (tamaño $\mathcal{O}((\lg n)^2)$). Luego en vez de tomar aleatoriamente los a, los verificamos todos. El test entonces toma tiempo $\mathcal{O}((\lg n)^5)$ (ejercicio).

[15]La *Hipótesis de Riemann* afirma que la función compleja Zeta $\zeta(s) = \sum_{n\geq 1} \frac{1}{n^s}$ no se anula en el semiplano $Re(s) > 1/2$. Es posible generalizar esta función utilizando *caracteres* sobre el grupo multiplicativo $(\mathbb{Z}/(n))^*$ (es decir, homomorfismos de $(\mathbb{Z}/(n))^*$ en el círculo unitario) y extendiéndolos naturalmente a los números naturales. Se define entonces la L-función de Dirichlet correspondiente al carácter χ como $L(s,\chi) = \sum_{n\geq 1} \frac{\chi(n)}{n^s}$. La *Hipótesis de Riemann Extendida* afirma que todos los ceros de esta función tienen parte real $1/2$.

Nota 5 *Podemos utilizar los algoritmos aleatorios de verificación de primos recién aprendido para encontrar un número primo de n dígitos binarios. Escogemos un número k del intervalo* $[2^{n-1}, 2^n - 1]$ *y verificar si es primo, con una probabilidad de error de a lo más* $2^{-100}/n$*. Si lo es, paramos. Si no lo es, escogemos un nuevo número k. De la teoría de números primos sabemos que en este intervalo el número de primos es bastante grande: asintóticamente es* $(\log e)2^{n-1}/n$*, i.e., un número de n dígitos escogido aleatoriamente será un primo con probabilidad* $1.44/n$*. Repitiendo entonces este experimento* $\mathcal{O}(n)$ *veces encontraremos un número primo con muy buena probabilidad. Análogamente es posible escoger un primo aleatorio en cualquier intervalo suficientemente grande, por ejemplo en el intervalo* $[1, 2^n]$*.*

2.5 Uso de cuerpos ciclotómicos y curvas elípticas

En 1980 Adleman, Pomerance y Rumely presentaron el primer algoritmo de primalidad determinístico que funcionaba en tiempo casi polinomial. Cohen y Lenstra mejoraron el algoritmo logrando que corriera en tiempo $\mathcal{O}((\log n)^{c \log\log\log n}$ donde $c > 0$ es una constante. Aunque no es un algoritmo polinomial, para todos los efectos prácticos el exponente $c \log\log\log n$ funciona como una constante. De hecho, permite verificar primalidad para números grandes –1000 decimales. La idea del algoritmo APRCL es verificar congruencias del tipo Fermat, pero en ciertos cuerpos de números (cuerpos ciclotómicos).

El otro tipo de algoritmos "modernos" para primalidad está basado en curvas elípticas sobre cuerpos finitos. La idea fué formulada por Goldwasser y Kilian (1986) quienes formularon un algoritmo aleatorio y fué modificada por Atkin y Morain (1993) para convertirlo en un test práctico (ECPP). En vez de usar los grupos multiplicativos usuales, of de $\mathbb{Z}/(n^d)$, se usa el grupo de puntos de curvas elípticas sobre $\mathbb{Z}/(n)$.

Referimos al lector a las referencias debido a que es imposible presentar los detalles en el marco de estos apuntes. El espíritu está señalado en la Nota 2.

3 Primalidad está en P

En la práctica el algoritmo de Rabin-Miller funciona bastante bien si certeza absoluta no es un imperativo. Para números grandes, ECPP y APRCL funcionan bien. En particular, APRCL provee un método "práctico" para *verificar* primalidad.

La situación hasta el 2 de Agosto de 2002 era esta: el problema de determinar si existe un algoritmo (determinístico, i.e. no aleatorio) para verificar primalidad en tiempo polinomial estaba abierto; el único algoritmo conocido que era "casi" polinomial (APRCL) exige bastante teoría algebraica de números. Ese día Agrawal, Kayal y Saxena presentan un algoritmo que permite verificar primalidad en tiempo polinomial (aunque el algoritmo no es practicable como está) y de una simpleza extraordinaria (prácticamente usa sólo álgebra elemental).

Probablemente el mejor resumen del trabajo *Primes is in P* de Agrawal, Kayal y Saxena lo haya dado el magazine indio *Frontline* de la segunda quincena de Agosto del 2002:

> A team of young researchers at the Indian Institute of Technology, Kanpur, solves a problem in number theory and computer science that had eluded the best minds for decades.

El 6 de Agosto de 2002 pusieron el paper en Internet, antes de enviarlo a referear en alguna revista. A poco andar, todo el mundo comprobó asombrado la corrección del algoritmo.

3.1 Idea general

El algoritmo está basado en la siguiente identidad de números primos:

Proposición 2 *Supongamos que* $(a,p)=1$. *Entonces* p *es primo si y sólo si*

$$(x-a)^p \equiv (x^p - a) \ (\operatorname{mod} p). \tag{2}$$

Demostración. Para $i<p$, el coeficientes de x^i en $((x-a)^p-(x^p-a))$ es $(-1)^i\binom{p}{i}a^{p-i}$ Ahora bien, si p es primo, $\binom{p}{i} \equiv 0(\operatorname{mod} p)$ y por lo tanto todos los coeficientes son cero. Si p es compuesto, consideremos un factor primo q de p y sea k maximal tal que $q^k|p$. Entonces q^k no divide $\binom{p}{q}$ y $(q^k, a^{p-q})=1$, y luego el coeficiente de x^q no es cero módulo p. Luego $((x-a)^p-(x^p-a))$ no es idénticamente cero módulo p. $\odot$

La idea es la siguiente: dado p como entrada, uno podría tomar un polinomio $P(x) = x-a$ y computar si la congruencia de la (2) se satisface o no. El único problema es que este procedimiento pudiera tomar $\Omega(p)$, pues en el peor caso necesitamos evaluar p coeficientes en el lado izquierdo.

Para hacer la idea practicable evaluaremos ambos lados de (2) módulo un polinomio de la forma x^r-1 para un r "adecuado". Una iteración del algoritmo consistirá en evaluar si la siguiente congruencia se cumple:

$$(x-a)^p \equiv (x^p-a)(\operatorname{mod} x^r-1, p). \tag{3}$$

(Esto es determinar si $(x-a)^p = (x^p-a)$ en $(\mathbb{Z}/(p))[x]/(x^r-1)$.) Claramente, todos los primos p satisfacen la congruencia (3) para todos los valores de a. El resultado principal del trabajo de Agrawal, Kayal y Saxena es probar que una forma de inverso de este resultado también es cierto, a saber, si para todo a "pequeño" $(\mathcal{O}(\sqrt{r}\log p)$ la congruencia (3) se cumple, entonces p es primo.

El algoritmo toma entonces tiempo $\mathcal{O}(r^2\log^3 p)$ (¿por que?). Los autores demuestran que es posible elegir un r "adecuado" de tamaño $\mathcal{O}(\log^6 p)$, y luego el algoritmo corre en tiempo polinomial.

3.2 El algoritmo

El algoritmo AKS recibe como entrada un número natural n y retorna COMPUESTO o PRIMO dependiendo que propiedad posee n.

AKS(n)

1. **if** ($n = a^b$) **return** COMPUESTO;
2. $r \leftarrow 2$;
3. **while** ($r < n$) **do**
4.   **if** r es primo **do**
5.   **if** $r|n$ **return** COMPUESTO;
6. $q \leftarrow$ el mayor factor primo de $r - 1$;
7.   **if** $q \geq 4\sqrt{r}\log_2 n$ **and** $n^{(r-1)/q} \not\equiv 1(\bmod r)$ **break**;
8. $r \leftarrow r + 1$;
9. **for** $a = 1$ **to** $2\sqrt{r}\log_2 n$ **do**
10.   **if** $(x-a)^n \not\equiv x^n - a \ (\bmod (x^r - 1, n))$ **return** COMPUESTO
11. **return** PRIMO

Comentarios. El algoritmo se basa esencialmente en dos bucles. En el primero (líneas 3-8) elige el r "adecuado". En el segundo (líneas 9-10) verifica la identidad para los a's pequeños. La línea 1 elimina los casos en que n es una potencia perfecta (que se necesitará en la prueba de corrección). En la línea 7, la expresión $n^{(r-1)/q} \not\equiv 1(\bmod r)$ es equivalente a afirmar que q divide el orden de n módulo r. Todo el resto es completamente autoexplicativo.

3.3 Demostración de corrección

El siguiente es el resultado fundamental que hacer funcionar el algoritmo de Agrawal-Kayal-Saxena:

Teorema 10 (Fundamental) *Sea n un entero positivo. Sea r primo y q el mayor factor primo de $r-1$. Sea s entero positivo. Supongase que:*

1.

$$\binom{q+s-1}{s} > n^{2\lfloor\sqrt{r}\rfloor}. \tag{4}$$

2. n no tiene factores primos $\leq s$ y que $n^{(r-1)/q} \bmod r \notin \{0,1\}$ (Es decir, r no divide a n ni q divide al orden de n módulo r).

3. $(x-b)^n = x^n - b(\bmod x^r - 1, n)$ para todo $1 \leq b \leq s$.

Entonces n es una potencia de un número primo.

Demostración. Como $n^{(r-1)/q} \bmod r \notin \{0,1\}$, entonces existe un factor primo p tal que $p^{(r-1)/q} \bmod r \notin \{0,1\}$. De aquí en adelante trabajaremos en $\mathbb{F}_p[x]$. Por construcción el orden de p módulo r es más grande que q.

Consideremos un factor irreducible $h(x)$ del r-ésimo polinomio ciclotómico $\Phi_r(x) = x^{r-1} + x^{r-2} + \cdots + 1$.

Dividiremos la demostración usaremos una serie de lemas simples.

Lema 4 *El grado de h es al menos q.*

Demostración. Sea d el grado de $h(x)$. De la teoría de cuerpos finitos, sabemos que $h(X)$ divide a $x^{p^d} - x$. Como $h(x)$ divide a $x^r - 1$, se deduce que $h(x)$ divide a $\mathrm{mcd}(x^r - 1, x^{p^d} - 1) = x^{\mathrm{mcd}(r, p^d - 1)} - 1$. Si $d < q$, entonces $h(x)|(x-1)$, luego son iguales. Pero entonces $\Phi_r(1) = r \not\equiv 0 (\mathrm{mod}\, p)$, una contradicción. $\odot$

Sea $F = \mathbb{F}_p[x]/(h(x))$, que es un cuerpo finito. Denotemos por G el subgrupo de F^* generado por los polinomios $(x-k)^e$ para $1 \leq k \leq s$.

Lema 5 *El grupo G es cíclico y tiene al menos $\binom{q+s-1}{s}$ elementos.*

Demostración. G es cíclico pues es subgrupo de un grupo cíclico. Por otra parte, G contiene al menos todos los productos $(x-1)^{e_1}(x-2)^{e_2}\cdots(x-s)^{e_s}$ con $e_1 + \cdots e_s \leq q - 1 < \mathrm{gr}(h)$ y estos polinomios son todos distintos (recuérdese que n no tiene factores más pequeños que s). $\odot$

Como G es cíclico tiene un generador, sea $g(x) = (x-1)^{e_1}(x-2)^{e_2}\cdots(x-s)^{e_s}$.

Lema 6 *El polinomio $g(x)$ verifica que $g(x)^n = g(x^n)(\mathrm{mod}\, x^r - 1, n)$.*

Demostración. Por hipótesis, $(x-b)^n = x^n - b(\mathrm{mod}\, x^r - 1, n)$ para $1 \leq b \leq s$. Luego

$$g(x)^n = ((x-1)^{e_1}\cdots(x-s)^{e_s})^n \equiv (x^n - 1)^{e_1}\cdots(x^n - s)^{e_s} = g(x^n).$$

$\odot$

Denotaremos por T el conjunto de enteros $\{e : g(x)^e \equiv g(x)^e(\mathrm{mod}\, x^r - 1, p)\}$. Vimos más arriba que $n \in T$. Como p es primo, también $p \in T$. Por supuesto también $1 \in T$.

Lema 7 *El conjunto T es un monoide multiplicativo.*

Demostración. Si $f \in T$, entonces vale $g(x^e)^f \equiv g(x^{ef})(\mathrm{mod}\, x^{er} - 1, p)$. Esto implica que $g(x^e)^f \equiv g(x^{ef})(\mathrm{mod}\, x^r - 1, p)$ porque $x^r - 1$ divide a $x^{er} - 1$. Luego, si $e, f \in T$ se cumple $g(x)^{ef} = (g(x)^e)^f \equiv g(x^e)^f \equiv g(x^{ef})(\mathrm{mod}\, x^r - 1, p)$. $\odot$

Corolario 1 *El grupo T contiene todos los productos $n^i p^j$.*

Estamos listos para terminar la demostración del teorema. Consideremos todos los productos $n^i p^j$ con $0 \leq i, j \leq \lfloor\sqrt{r}\rfloor$. Hay $(\lfloor\sqrt{r}\rfloor + 1)^2 > r$ tales números. Existen entonces pares diferentes (i,j) y (k,l) tal que $n^i p^j \equiv n^k p^l (\mathrm{mod}\, r)$. Pongamos $t = n^i p^j$ y $u = n^k p^l$. Por construcción $g(x^t) \equiv g(x^u)(\mathrm{mod}\, x^r - 1)$, de donde obtenemos $g(x)^t \equiv g(x)^u(\mathrm{mod}\, x^r - 1)$. Luego $g(x)^t \equiv g(x)^u(\mathrm{mod}\, h(x))$, lo que dice que $t \equiv u(\mathrm{mod}\, \mathrm{ord}(g))$. Pero t y u están acotados por $n^{2\lfloor\sqrt{r}\rfloor} < \binom{q+s-1}{s} \leq \mathrm{ord}(g)$. Por consiguiente, $t = u$ y luego $n^{i-k} = p^{j-l}$. $\odot$

Nota 6 *Veamos las condiciones necesarias para obtener el resultado. De la condición (4) surgen los parámetros del algoritmo y las condiciones de la Proposición 1.*

Como $\binom{q+s-1}{s} > \binom{q}{s}^s$, *imponiendo* $q \geq 2s$ *obtenemos la inecuación* $2^s \geq n^{2\lfloor\sqrt{r}\rfloor}$. *Luego tenemos*

$$\frac{r-1}{2} \geq q \geq 2s \geq 4\lfloor\sqrt{r}\rfloor \log_2 n.$$

Notemos que esto implica que $n > r \geq 64(\log_2 n)^2$, *lo que significa* $n \geq 11689$.

3.4 Análisis de complejidad

Denotemos por $\bar{\mathcal{O}}(f(n))$ la expresión $\mathcal{O}(f(n)p(\log f(n)))$, donde $p(x)$ es un polinomio. La notación $\bar{\mathcal{O}}$ nos permite esencialmente desligarnos de los factores logarítmicos que pudieran aparecer.

Proposición 3 *La complejidad asintótica de tiempo del algoritmo AKS es* $\bar{\mathcal{O}}(\log^{19} n)$ *utilizando algoritmos clásicos de multiplicación y* $\bar{\mathcal{O}}(\log^{12} n)$ *utilizando multiplicación rápida.*

Demostración. El tiempo está dominado por el bucle de las líneas 9-10 que analizaremos al final.

Veamos rápidamente las líneas 1 a 8. Para la línea 1, vimos que detectar potencias toma tiempo $\mathcal{O}(\log^3 n)$. El ciclo **while** (líneas 3-8) busca el r "adecuado" y por supuesto se repite a lo más r veces. Veamos cuanto cuesta cada iteración. Línea 4: la implementación de fuerza bruta toma tiempo $\mathcal{O}(\sqrt{r}\log^2 r)$. Línea 5: división $\mathcal{O}(\log^2 n)$. Línea 6: fuerza bruta toma $\mathcal{O}(\sqrt{r}\log^2 r)$ (ejercicio). Línea 7: esencialmente exponenciación y aritmética módulo r; tiempo $\mathcal{O}((\log r)^3 + (\log n)(\log r))$. Luego, el ciclo **while** (líneas 3-8) toma tiempo $\mathcal{O}(r^{3/2}\log^2 r)$.

Analizemos ahora el bucle de las líneas 9-10. Para cada valor de a, el cálculo de $(x-a)^n(\text{mod}(x^r-1,n)$ demanda $\mathcal{O}(\log n)$ multiplicaciones $A(x)B(x) \bmod (x^r-1)$ donde $A(x)$ y $B(x)$ tiene grado $\mathcal{O}(r)$. Si denotamos $t(n,r)$ el tiempo necesario para multiplicar dos polinomios de grado r con coeficientes en $\mathbb{Z}/(n)$, el tiempo del bucle 9-10 es $\mathcal{O}(\sqrt{r}(\log n)^2 t(n,r))$. Por otro lado, el tiempo $t(n,r)$ puede expresarse en número de operaciones sobre los enteros módulo n. Si $M(n)$ denota el tiempo de multiplicación de dos enteros módulo n, se tiene que $t(n,r) = r^2 M(n)$ si se utiliza el método clásico de multiplicación de polinomios, y $t(n,r) = r\log r M(n)$ si se utiliza la transformada rápida de Fourier (FFT). Similarmente, utilizando aritmética ingenua en $\mathbb{Z}/(n)$, $M(n) = \mathcal{O}(\log^2 n)$; si se utiliza FFT, tenemos que $M(n) = \bar{\mathcal{O}}(\log n)$. En resumen, si todo es ingenuo, obtenemos $\mathcal{O}(r^{5/2}\log^4 n)$. Si se utiliza FFT, $\bar{\mathcal{O}}(r^{3/2}\log^3 n)$.

Finalmente, de la Proposición 1 se deduce que $r = \mathcal{O}(\log^6 n)$, y por lo tanto obtenemos los resultados del enunciado. $\odot$

Consideraciones prácticas. El algoritmo como está no es práctico aún para números de tamaño "moderado".

Aunque es posible buscar el r adecuado de forma bastante eficiente si uno se preocupa de los detalles de implementación, no es evidente (al menos hoy día) cómo ejecutar rápidamente el bucle de las líneas 9-10. El problema son las evaluaciones de polinomios. Aunque uno puede hacer exponenciación eficientemente con métodos estándares, luego de algunas iteraciones se obtienen expresiones intermedias grandes. Por ejemplo para n de 512 bits, r será aproximadamente 3 millones. Debemos evaluar por ejemplo $(x-1)^n(\mathrm{mod}(x^r-1,n)$. Después de 22 iteraciones (de las 512 necesarias...) tendremos un polinomio con cerca de 3 millones de coeficientes, cada uno con 512 bits. No sabemos hoy día cómo trabajar con estos polinomios en forma eficiente.

Trabajo para la casa... Para finalizar, dejamos al lector la siguiente conjetura que dejaría al algoritmo AKS compitiendo en la práctica.

Conjetura [Bhattacharjee-Pandey 2001] Si r no divide n y si

$$(x-1)^n \equiv (x^n-1) \pmod{x^r-1, n}, \tag{5}$$

entonces n es primo o $n^2 \equiv 1 (\mathrm{mod}\, r)$.

Si esta conjetura es verdadera, es posible modificar el algoritmo de la siguiente forma: Primero, buscar un r que no divida n^2-1. Es posible encontrar tal r en el intervalo $[2, 4\log n]$ (debido a que el producto de números primos menores que x es al menos e^x.) Luego verificar si la congruencia (5) vale o no. Esto último toma tiempo $\bar{\mathcal{O}}(r\log^2 n)$ usando FFT. Luego el algoritmo tomaría tiempo $\bar{\mathcal{O}}(\log^3 n)$ y luego sería perfectamente practicable al menos para números muy grandes.

Referencias

[1] M. Agrawal, N. Kayal, N. Saxena, *PRIMES is in P*, Dept. of C.S. & Eng, Indian Inst. of Tech. Kanpur Aug 6, 2002 `http://www.cse.iitk.ac.in/primality.pdf`

[2] E. Bach, J. Shallit, *Algorithmic Number Theory, Volume 1: Efficient Algorithms*, MIT Press, 1996.

[3] H. Cohen, *A Course in Computational Algebraic Number Theory*, Springer Verlag, 1993.

[4] J. von zur Gathen, J. Gerhard, *Modern Computer Algebra*, Cambridge Univ. Press, 1999.

[5] F. Morain, *Primalité théorique et primalité pratique ou AKS vs. ECPP*, (en la Web).

[6] T. Cormen, Ch. Leisserson, R. Rivest. *Introduction to Algorithms*, MIT Press, 1990.

[7] C. Papadimitriou, *Computational Complexity*, Addison-Wesley Publ. Co., 1995.