PRIMALIDAD

Paula Muñoz Ramírez

PRIMALIDAD

Paula Muñoz Ramírez

Memoria presentada como parte de los requisitos para la obtención del título de Grado en Matemáticas por la Universidad de Sevilla.

Tutorizada por

Prof. Tutor José María Tornero Sánchez

Índice general

Abstract

The aim of this work is to present a deterministic method that establishes with absolute certainty whether a given number is a prime or not. In order to give an answer to this question, we will study in depth the AKS algorithm, which gives a theoretical polynomial–time solution to the Primality problem.

The first chapter is devoted to give an introduction of what being a prime means, as well as presenting the Primality problem. We will talk about some subjects, from both Mathematics and Computer Science, where this problem plays an essential role. Moreover, we will make a distinction between probabilistic primality tests and deterministic primality tests, giving an example for each case.

In the second chapter, we will introduce some notations that we will use through the text. In addition to this, we will also show and recall several previous concepts that will help us in the sequel.

During the third chapter we will be presenting the AKS algorithm as well as developing the ideas and mathematical concepts behind this test that verifies its correctness. We will use not only modular arithmetic but also group theory in order to build some results that allow us to prove the primality of a certain number in a satisfactory way.

Throughout the fourth chapter we will be studying the complexity of AKS algorithm using basic notions of complexity theory.

Eventually, in the last chapter we will give an example of the AKS algorithm, providing us a detailed study of each step.

Resumen

El objetivo de este trabajo es exponer un método determinístico para, dado un cierto entero n, poder afirmar con total certeza si es o no primo. Para ello estudiaremos en profundidad el algoritmo determinístico AKS [PinP] que da una solución *computacionalmente eficaz* (esto es, en tiempo logarítmico) al problema de la Primalidad.

Dedicaremos el primer capítulo a introducir el concepto de número primo, así como el problema de la Primalidad. Veremos algunas áreas, tanto de las matemáticas como de las ciencias de la computación, en las que tanto dicho problema como los números primos en sí mismos juegan un papel fundamental. Por otra parte haremos una distinción entre tests de primalidad probabilísticos y tests de primalidad determinísticos, tomando algunos algoritmos como ejemplo en cada caso.

En el segundo capítulo introduciremos algunas notaciones que adoptaremos a lo largo del texto, además de exponer o recordar diversos conceptos previos para el desarrollo de nuestro tema.

A lo largo del tercer capítulo, expondremos el algoritmo AKS y desarrollaremos las ideas y conceptos matemáticos que se encuentran detrás de él y que verifican su corrección. Haremos uso de la aritmética modular y de la teoría de grupos para construir una serie de resultados que permitan demostrar la primalidad de un cierto número de manera satisfactoria.

Siguiendo con el algoritmo AKS, en el cuarto capítulo se estudia la complejidad del mismo, haciendo uso de resultados básicos de la teoría de la complejidad.

Finalmente, en el último capítulo emplearemos el algoritmo AKS para determinar la primalidad de un cierto valor n, haciendo un estudio detallado de cada paso.

1 El problema de la Primalidad.

El número 73 es claramente un número primo, pues es divisible únicamente por el número 1 y por sí mismo, de hecho es el vigésimo primer número primo. Además, si a este número le damos la vuelta se convierte en el número 37 que también es primo, el duodécimo número primo. Es decir, el 73 se encuentra en el lugar 21 de la lista de los números primos mientras que el 37 se encuentra en el lugar 12. Curioso, ¿no? Por otra parte, el 73 también es parte de un par de primos gemelos junto con el número 71, que también es primo. ¿Tendrá este número alguna propiedad curiosa más?

Estos números sirven mucho más que para darnos anécdotas interesantes que contar y han tenido un gran peso en las matemáticas a lo largo de la historia e incluso lo siguen teniendo en la actualidad. Los números primos han sido, desde la antigüedad, una herramienta de gran interés en muchos ámbitos. En la antigua Grecia, muchos matemáticos dedicaron parte de sus estudios a estos. Algunos, como Euclides, enunciaron y estudiaron sus propiedades más básicas. Otros, como Arquímedes o Eratóstenes, desarrollaron métodos para encontrar todos los números primos menores que un cierto valor n dado. Ya desde entonces, y hasta nuestros días, los números primos han sido un importante objeto de estudio desde el punto de vista matemático y, gracias a ello, se han desarrollado diversos teoremas y conjeturas. Un claro ejemplo de esto es el *teorema de los números primos* de Hadamard o la *conjetura sobre la densidad de los primos de Sophie Germain*.

En relación a la densidad de los números primos y cómo se distribuyen en $\mathbb{Z}$ se conocen muchas propiedades, pero hay así mismo una gran cantidad de conjeturas o hipótesis, pues dicha distribución es un tanto ambigua. Sabemos que no podemos encontrar números primos consecutivos, salvo en el caso del número 2 y el 3; pero que sí existen primos que se diferencian en 2 unidades (los llamados *primos gemelos*) como por ejemplo el 3 y el 5 o el 5 y el 7; y también sabemos que podemos encontrar *desiertos de primos* tan grandes como queramos. Pero las claves últimas de esta distribución se

nos escapan.

El interés en relación a dichos números da paso al estudio del problema de la Primalidad. Dicho problema, que se define como aquel consistente en determinar si un número n dado es o no primo, es una cuestión primordial en campos como la teoría de números o ciencias de la computación. Se sabe, gracias a [PinP], que pertenece a la clase **P** de complejidad, que es aquella que contiene a todos los problemas que pueden resolverse en un tiempo polinomial con una máquina de Turing determinista.

1.1 Áreas de influencia.

El problema de la Primalidad juega un importante rol en diversas áreas de estudio. Algunas de ellas las veremos a continuación.

En primer lugar, cabe destacar su importancia, como ya hemos mencionado antes, en la teoría de números. Todo número natural puede descomponerse como producto de potencias de números primos, por el *teorema fundamental de la aritmética*. Dicha propiedad, aunque parezca obvia, no es nada trivial. Su estudio partió de las investigaciones de Euclides y fue el persa Kamal al-Din Al-Farisi quien lo enunció más o menos formalmente por primera vez.

Esta rama de las matemáticas también se plantea cómo se distribuyen los números primos dentro del conjunto de los números naturales. Como ya hemos mencionado anteriormente parece no existir una clara regla en dicha distribución. Sin embargo, en el *teorema de los números primos* se establece que hay una cantidad infinita de dichos números. Por otra parte, la llamada función *Zeta de Riemann*,

$$\zeta(s) = \sum_{n=1}^{\infty} \frac{1}{n^s},$$

está íntimamente relacionada con la distribución de estos números como se puede ver en [TAN]. La famosa *hipótesis de Riemann*, uno de los Problemas del Milenio sin resolver, conjetura propiedades de esta función.

En áreas como la estadística o las ciencias de la computación los números primos juegan un papel de gran importancia a la hora de generar números aleatorios. Dichos números son ideales para ello pues, al no estar compuestos por otros números, es decir, al ser divisibles únicamente por el 1 y por ellos mismos, no pueden generarse

mediante productos de otros números, son menos predecibles y por tanto idóneos para la generación aleatoria de números. En estadística esto es vital para diversos menesteres como los muestreos aleatorios, simulaciones, cálculo de intervalos de confianza o prueba de hipótesis, entre otros. Por ejemplo, para simular sucesos aleatorios que se rigen según una determinada distribución probabilística se emplean métodos de generación de números aleatorios especializados para dicha distribución, o en el caso del remuestreo, que se generan aleatoriamente muestras a partir de una original para llevar a cabo estimaciones de intervalos de confianza o inferencias estadísticas. En el ámbito de las ciencias de la computación también se emplean dichos métodos de generación para diversos algoritmos o programas.

Es más, los números primos y el problema de la primalidad son claves en el área de la criptografía, esto es, en los algoritmos dedicados a la encriptación de mensajes. De hecho, en muchos de estos algoritmos su efectividad y seguridad radican en la dificultad de factorizar números lo suficientemente grandes en producto de primos. Así mismo, para la generación de claves, se requiere poder crear números primos de gran tamaño, lo cual evidentemente implica poder resolver el problema de la primalidad de forma eficiente.

Un primer ejemplo de esto es el caso del protocolo *Diffie–Hellman* de intercambio de claves. En este protocolo en concreto se puede encontrar el siguiente problema: dado un primo p y dos enteros a y g menores que p, hallar (si existe) $x \in \mathbb{Z}$ verificando

$$a = g^x \mod p,$$

conocido también por el *problema del logaritmo discreto* (PLD). Este problema es el inverso de la exponenciación modular, que se puede resolver de manera eficiente, pero a diferencia de éste, no se conocen algoritmos en tiempo polinomial que resuelvan el PLD [TCYC][1].

Otro claro ejemplo del uso de números primos dentro de esta rama es su importancia en algoritmos de encriptación asimétrica, que son aquellos donde cada usuario posee dos claves distintas: una pública (que puede conocer cualquier usuario del protocolo) y una privada, que es conocida exclusivamente por el usuario para la que se genera. La primera se usa para encriptar y la segunda para desencriptar, de forma que todo el mundo puede cifrar mensajes, pero solo los legítimos destinatarios pueden descifrar los que se hayan cifrado con su clave pública.

[1] En realidad en el protocolo Diffie-Hellamn se usa un problema muy similar, que se asume como equivalente al PLD, pero a día de hoy no se ha demostrado esto.

Entre estos modelos destaca el protocolo *RSA*, que se estudia también en [TCYC]. Para generar la clave pública de un usuario se toman dos números primos arbitrarios, p y q, y se multiplican para obtener un entero N. A su vez se escoge e de tal forma que sea coprimo con $\phi(N)$ y así, el par (N, e) forma la clave pública que se emplea para encriptar los datos. La clave privada se obtiene a partir de la pública buscando un cierto valor d que verifique

$$e \cdot d \equiv 1 \mod \phi(N).$$

Dicha d será la clave privada que se empleará para descifrar el mensaje y conocerla equivale, de facto, a conocer una factorización de N (que es donde radica la seguridad del sistema). La utilización de los números primos en estos casos y los problemas computacionales que presentan hacen estos cifrados mucho más seguros y menos susceptibles a ataques.

Hablando acerca de la seguridad de diversos protocolos, cabe destacar también el papel de los números primos en la firma digital, una herramienta criptográfica que se emplea para verificar la autenticidad de un cierto mensaje. Un claro ejemplo de ello es la firma *DSS*, estudiada en [TCYC]. En este caso el usuario, para crear sus parámetros para la firma digital, toma en primer lugar un número primo, llamémoslo Q, del orden de 2^{160}, y a partir de este toma un segundo número primo, P, del orden de 2^{1024} tal que verifique que

$$P \equiv 1 \mod Q.$$

Una vez conocemos Q y P, tomamos $w \in \mathbb{Z}$ de forma aleatoria y calculamos $w^{(P-1)/Q} \pmod P$. Como es claro que $\mathbb{F}_P^*$ tiene un único subgrupo cíclico de orden Q, si este cálculo es distinto de 1, podemos tomar $G = w^{(P-1)/Q}$ como un generador de dicho subgrupo cíclico (si no es el caso, probamos con otros valores hasta que se cumpla). Finalmente escogemos n, satisfaciendo $0 < n < Q$, como clave privada y como clave pública tomamos (Q, P, G, E), donde $E = G^n \pmod P$. Con esta clave pública / privada y una cierta clave efímera, $h > 0$, el usuario puede generar su firma digital que verificará de manera unívoca su identidad.

Es claro, como ya hemos observado, que tanto los números primos como el problema de la primalidad son claves para el correcto desarrollo de diversos protocolos criptográficos y su seguridad. Sin embargo, estos también son de gran utilidad en los ataques a los mismos, es decir, son la base en diversas técnicas empleadas para romper sistemas criptográficos. El estudio de dichas técnicas se denomina criptoanálisis y su

interés radica en el desarrollo de ciertos métodos o algoritmos para tratar de factorizar números compuestos relativamente grandes en el producto de sus factores primos para poder lograr romper los sistemas de encriptación mencionados anteriormente. El criptoanálisis de algoritmos basados en el empleo de los números primos es muy importante pues, una vez conocemos cómo se pueden romper dichos criptosistemas podemos reforzarlos para que no sean tan fácil de atacar.

En resumidas cuentas, el problema de la primalidad es una cuestión de vital importancia tanto en las matemáticas más puras como en sus aplicaciones más básicas, además de en algunas no tan básicas. La seguridad de numerosos sistemas criptográficos y la confidencialidad de la información que se maneja dependen, por lo general, en mayor o menor medida del problema de conseguir descomponer ciertos números bastante grandes en productos de factores primos. El estudio de la primalidad ha iniciado pues abundantes investigaciones y considerables avances en numerosos campos que a día de hoy siguen siendo una vía activa de estudio.

1.2 Algunos algoritmos de primalidad.

Los algoritmos de primalidad se separan en dos grandes bloques, los test probabilísticos y los test determinísticos. Los algoritmos que pertenecen a la familia de los test de primalidad probabilísticos son aquellos que no garantizan dar la respuesta correcta con total certeza, sino que dan una cierta respuesta con una mayor o menor probabilidad de error. Algunos algoritmos destacables dentro de esta familia son el test de Miller-Rabin o el test de Solovay-Strassen.

Estudiemos someramente el segundo de ellos pues es bastante interesante. Para ello introduciremos algunos conceptos que nos serán útiles en el estudio del algoritmo.

Definición 1.1. *Sea p un número primo y $a \in \mathbb{Z}$, definiremos el símbolo de Legendre (de a sobre p) como*

$$\left(\frac{a}{p}\right) = \begin{cases} 0 & si \quad p \mid a \\ 1 & si \quad a \text{ tiene raíz cuadrada} \quad \text{mód } p \\ -1 & si \quad a \text{ no tiene raíz cuadrada} \quad \text{mód } p \end{cases}$$

Además, si tenemos $n \in \mathbb{Z}$, tal que la descomposición de n en factores primos es

$n = p_1^{\alpha_1} \cdot \ldots \cdot p_s^{\alpha_s}$, denominamos símbolo de Jacobi (de a sobre n) a

$$\left(\frac{a}{n}\right) = \left(\frac{a}{p_1}\right)^{\alpha_1} \cdots \left(\frac{a}{p_s}\right)^{\alpha_s}.$$

Una vez conocemos esta definición, podemos enunciar un resultado basado en una modificación del *Teorema de Fermat* empleando el *símbolo de Jacobi.*

Lema 1.1. Dados $a \in \mathbb{Z}$ y p un número primo, se verifica:

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

Basándonos en la definición anterior y este último resultado consideramos la siguiente definición.

Definición 1.2. *Dados $a \in \mathbb{Z}$ y n un número compuesto e impar, diremos que n es un pseudoprimo de Euler en base a si ambos verifican las dos condiciones siguientes:*

$$\gcd(a, n) = 1, \qquad \left(\frac{a}{n}\right) = a^{\frac{n-1}{2}} \mod n.$$

Una vez conocemos los anteriores resultados es bastante sencillo de comprender el test de primalidad de Solovay-Strassen [TCYC]. Sea n un número el cuál queremos ver si es o no primo.

En primer lugar debemos tomar aleatoriamente un cierto número a tal que $a \in \{2, 3, \ldots, n-1\}$ y nos aseguramos de que $\gcd(a, n) = 1$.

Una vez conocido dicho valor, comprobamos si se verifica

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

Si n no es un pseudoprimo de Euler en base a podemos afirmar que n es un número compuesto, en caso afirmativo, repetiríamos el proceso y lo haremos un cierto número fijo, m, de veces. Si, tras reiterar el desarrollo m veces, para ningún a cumpliendo lo anterior se ha verificado que n no es un pseudoprimo de Euler en base a, ciertos resultados sobre la distribución de estos pseudoprimos nos permiten asegurar con una probabilidad de $1 - (1/2)^m$ que n es primo. De esta forma, variando m podemos conseguir una elevada probabilidad de que nuestro candidato n sea primo (o bien la certeza de que no lo es).

La otra gran familia de test de primalidad es la de los algoritmos determinísticos. Estos son aquellos que aportan una respuesta precisa a la pregunta de si es n primo. Dichos protocolos presentan la solución, al contrario que los test probabilísticos, sin ningún margen de error. Entre ellos podemos encontrar la criba de Eratóstenes o el test de AKS.

La criba de Eratóstenes fue diseñada por el matemático griego Eratóstenes, que le da su nombre, y tiene como función encontrar todos los números primos menores que un cierto n dado. Veamos como funciona.

En primer lugar debemos escribir en una lista los números naturales comprendidos entre el 2 y n, incluyendo a estos. Comenzaremos escogiendo el primer número no marcado de la lista, que es el número 2 y marcaremos todos los múltiplos de 2, exceptuando el mismo, pues es primo. A continuación seleccionaremos el siguiente elemento no marcado de nuestro registro, q, que será el siguiente número primo y volveremos a marcar aquellos que sean múltiplos de este, exceptuando el propio número q. Dicho procedimiento se reiterará hasta haber seleccionado en alguna ocasión todos aquellos elementos no marcados de la lista. Una vez finalizado esto, los números que quedan sin marcar en nuestro registro tras haber aplicado la criba son todos los números primos menores que el cierto número n dado.

Este algoritmo se basa en el hecho de que todo número compuesto debe ser dividido por algún número primo, luego por ello vamos eliminando aquellos naturales que son divididos por los primos que vamos seleccionando. Es más, aunque no parezca claro en el desarrollo del algoritmo nos bastará con reiterar los pasos hasta llegar a $\sqrt{n}$, pues de ser n compuesto, deberá tener al menos un factor primo menor que $\sqrt{n}$.

Por otra parte, otro algoritmo determinístico interesante es el test AKS, el cual es relativamente nuevo, fue desarrollado a principios del siglo XXI por Manindra Agrawal, profesor del Departamento de Ciencias de la Computación del *Indian Institute of Technology*, junto con Neeraj Kayal y Nitin Saxena, estudiantes en la misma institución, como se comenta en [BFE]. Dicho algoritmo fue presentado por primera vez en [PinP] y su importancia reside en su capacidad de concluir, en tiempo logarítmico, si un cierto número es o no primo con total seguridad, de manera determinística, a diferencia de los tests probabilísticos mencionados anteriormente, que no nos dan el resultado con total seguridad. Esto supuso un gran avance dentro del campo de las ciencias de la computación, pues la discusión de que el problema de la Primalidad se encontrase en la clase **P** de complejidad era un problema abierto, hasta que Agrawal,

Kayal y Saxena desarrollaron su algoritmo, conocido actualemente como AKS.

Dicho algoritmo demostró que es posible probar si un número es primo o no en un tiempo polinomial. Sin embargo, en la práctica no es muy útil para números considerablemente grandes, luego su interés es meramente teórico por ahora y, por ende, en la práctica sigue haciéndose uso de los algoritmos probabilísticos que son mucho más rápidas.

A lo largo de este trabajo estudiaremos las ideas y técnicas matemáticas que se encuentran tras este algoritmo y que lo hacen tan interesante. Nos basaremos en [PinP] para ello.

2 Notación y preliminares matemáticos

2.1 Notación

- Usaremos $q^k \parallel n$ para notar que $q^k \mid n$ pero $q^{k+1} \nmid n$.
- Para el estudio de la complejidad emplearemos la notación de la *soft-*$\mathcal{O}$, esto es $f(n) = \widetilde{\mathcal{O}}(g(n))$ que equivale a $f(n) = \mathcal{O}\left(g(n)\log^k(g(n))\right)$ (para un cierto k).
- También usaremos la *big omega*, siendo $f(n) = \Omega(g(n))$ equivalente a $g(n) = \mathcal{O}(f(n))$. Esto es, f está acotada inferiormente por g de forma asintótica,

$$\exists k > 0,\ \exists n_0 \mid \forall n > n_0 \text{ se tiene que } f(n) \geq k \cdot g(n).$$

- Por último, usaremos en lo sucesivo $\log(\cdot)$ para denotar el logaritmo en base 2, como es habitual en teoría de la complejidad.

2.2 Preliminares

En esta sección recordaremos diversas definiciones y teoremas que hemos visto a lo largo de la carrera y que nos serán de gran utilidad a la hora de desarrollar el algoritmo AKS.

Definición 2.1. *Sean $r \in \mathbb{N}$, $a \in \mathbb{Z}$ tal que $\gcd(a, r) = 1$. El orden de a módulo r es el menor k tal que $a^k \equiv 1 \pmod{r}$. Lo denotaremos por $o_r(a)$.*

Definición 2.2. *Sea $r \in \mathbb{Z}_{\geq 0}$. La función phi de Euler, que notaremos por $\phi(r)$, da el número de enteros positivos menores o iguales que r que son primos relativos con r.*

Observación 2.1. Sea $r \in \mathbb{N}$, para todo $a \in \mathbb{Z}$ tal que $\gcd(a, r) = 1$, se cumple que $o_r(a) \mid \phi(r)$. Esto es consecuencia directa del Teorema de Lagrange.

Teorema 2.1 (Pequeño teorema de Fermat). *Sea $a \in \mathbb{Z}$ tal que $a > 0$ y p primo. Entonces $a^p \equiv a \pmod{p}$. En particular, si $p \nmid a$, $a^{p-1} \equiv 1 \pmod{p}$.*

Definición 2.3. *Dado un cuerpo K definimos como el polinomio ciclotómico de orden n sobre K, denotado $\Phi_n(X)$, como el polinomio mónico de $K[X]$ que tiene por raíces todas las raíces primitivas de la unidad de orden n en K.*

Para el cálculo de $\Phi_n(X)$ consideraremos el polinomio $X^n - 1$, cuyas raíces son todas las raíces n-ésimas de la unidad. Cada una de estas raíces, a su vez, es una raíz j-ésima primitiva de la unidad para un cierto j, divisor de n. Del mismo modo, si consideramos Φ_j con $j \mid n$, todas sus raíces también son raíces de $X^n - 1$. Luego de esto deducimos que

$$X^n - 1 = \prod_{j \mid n} \Phi_j(X).$$

Despejando de la igualdad anterior, obtenemos un procedimiento recursivo para la obtención de polinomios ciclotómicos:

$$\Phi_n(X) = \frac{X^n - 1}{\prod_{j \mid n, j \neq n} \Phi_j(X)}.$$

Teorema 2.2. *Dado un cuerpo K, consideremos el cuerpo ciclotómico $K^{(n)}$, que es la extensión algebraica simple del cuerpo K obtenida al adjuntarle las raíces n-ésimas de la unidad.*

Si $K = \mathbb{F}_q$, con $\gcd(n, q) = 1$, entonces Φ_n factoriza en $\phi(n)/o_n(q)$ distintos polinomios mónicos irreducibles en $K[X]$, todos ellos de grado $o_n(q)$.

Podemos encontrar una prueba de este teorema en [FFA].

2.3 Resultados de interés

A continuación enunciaremos y demostraremos una serie de lemas previos que emplearemos a lo largo del siguiente capítulo en el desarrollo matemático del algoritmo.

En primer lugar, denotaremos por d_n al mínimo común múltiplo de los primeros n números enteros positivos, que nos será útil en el siguiente lema.

Lema 2.1. Para $n \geq 7$, $d_n \geq 2^n$.

Demostración. Para $1 \leq m \leq n$, consideremos la integral:

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

que se tiene directamente del binomio de Newton (la prueba se puede encontrar en [CIP]). Haciendo integración por partes de forma reiterada tenemos:

$$\begin{aligned} I(m,n) &= \int_0^1 x^{m-1}(1-x)^{n-m}dx = 0 + \frac{m-1}{n-m+1} \cdot \int_0^1 x^{m-2} \cdot (1-x)^{n-m+1}dx \\ &= 0 + \frac{(m-1)\cdot(m-2)}{(n-m+1)\cdot(n-m+2)} \cdot \int_0^1 x^{m-3} \cdot (1-x)^{n-m+2}dx = \ldots = \\ &= \frac{(m-1)\cdot(m-2)\cdot\ldots\cdot 1}{(n-m+1)\cdot(n-m+2)\cdot\ldots\cdot n} = \frac{m\cdot(m-1)\cdot(m-2)\cdot\ldots\cdot 1}{m\cdot(n-m+1)\cdot(n-m+2)\cdot\ldots\cdot n} \\ &= \frac{1}{m\cdot\binom{n}{m}} \end{aligned}$$

Claramente $I(m,n)\cdot d_n \in \mathbb{Z}$ pues, fijándonos en la parte derecha de la expresión de $I(m,n)$, vemos que $\binom{n-m}{r}\cdot(-1)^r$ es entero para cualquier r y, además, $m+r$ (donde $r \in \{0,\ldots,n-m\}$) se puede simplificar en cada sumando con algún factor de d_n. Luego podemos concluir que $I(m,n)\cdot d_n \in \mathbb{Z}$.

De esta forma, como $I(m,n)\cdot d_n \in \mathbb{Z}$, tenemos que $m\cdot\binom{n}{m} \mid d_n$ para cualquier m, con $1 \leq m \leq n$. Como consecuencia de ello, tomando unos valores específicos para m y n, vamos a poder acotar inferiormente d_n. En particular, si tomamos $n = 2n$ y $m = n$, tenemos que

$$n \cdot \binom{2n}{n} \mid d_{2n},$$

y, como por definición, $d_{2n} \mid d_{2n+1}$, se observa que:

$$n \cdot \binom{2n}{n} \mid d_{2n+1}. \tag{I}$$

Por otra parte, haciendo $n = 2n+1$ y $m = n+1$ en $m\cdot\binom{n}{m} \mid d_n$ obtenemos

$$(n+1)\cdot\binom{2n+1}{n+1} \mid d_{2n+1}.$$

Vamos a analizar un poco más el término de la derecha. Si desarrollamos el número combinatorio $\binom{2n}{n}$ y multiplicamos por $2n+1$ nos queda la siguiente igualdad:

$$\begin{aligned}(2n+1)\cdot\binom{2n}{n} &= \frac{(2n+1)\cdot 2n\cdot(2n-1)\cdot\ldots\cdot(2n-n+1)}{n\cdot(n-1)\cdot\ldots\cdot 1}\\ &= \frac{(n+1)\cdot(2n+1)\cdot 2n\cdot\ldots\cdot(n+1)}{(n+1)\cdot\ldots\cdot 1} = (n+1)\cdot\binom{2n+1}{n+1}.\end{aligned}$$

Así pues

$$(2n+1)\cdot\binom{2n}{n} \mid d_{2n+1}. \tag{II}$$

Como $\gcd(n, 2n+1) = 1$, podemos deducir de (I) y (II) que

$$n\cdot(2n+1)\cdot\binom{2n}{n} \mid d_{2n+1}.$$

Entonces

$$d_{2n+1} \geq n\cdot(2n+1)\cdot\binom{2n}{n} \geq n\cdot 4^n,$$

donde la última desigualdad se sigue del hecho de que $\binom{2n}{n}$ es el mayor de los $2n+1$ términos de la expresión de $(1+1)^{2n}$.

Si $n \geq 2$, entonces $d_{2n+1} \geq 2^{2\cdot n+1} = 4^n \cdot 2$. Por otra parte, si $n \geq 4$, entonces $d_{2n+2} \geq d_{2n+1} \geq 2^{2n+2}$. Por tanto $d_n \geq 2^n$, como queríamos demostrar.

Probemos ahora la siguiente desigualdad.

Lema 2.2. Para $n \in \mathbb{N}$ se cumple que

$$n\prod_{i=1}^{\lceil\log^2(n)\rceil}(n^i-1) < n^{\log^4(n)}.$$

Demostración. Supondremos que $n > 3$, ya que los casos restantes se comprueban inmediatamente. Por un lado tenemos

$$n\prod_{i=1}^{\lceil\log^2(n)\rceil}(n^i-1) < n\prod_{i=1}^{\lceil\log^2(n)\rceil}\left(n^{\lceil\log^2(n)\rceil}-1\right) = n\cdot\left(n^{\lceil\log^2(n)\rceil}-1\right)^{\lceil\log^2(n)\rceil}$$

Desarrollemos ahora empleando el binomio de Newton

$$\begin{aligned} n \cdot \left(n^{\lceil \log^2(n) \rceil} - 1\right)^{\lceil \log^2(n) \rceil} &= n \cdot \sum_{i=0}^{\lceil \log^2(n) \rceil} \binom{\lceil \log^2(n) \rceil}{i} \cdot n^{\lceil \log^2(n) \rceil - i} \cdot (-1)^i \\ &< n \cdot \sum_{i=0}^{\lceil \log^2(n) \rceil} \binom{\lceil \log^2(n) \rceil}{i} \cdot n^{\lceil \log^2(n) \rceil - i} \\ &< n \cdot \sum_{i=0}^{\lceil \log^2(n) \rceil} \binom{\lceil \log^2(n) \rceil}{i} \cdot n^{\lceil \log^2(n) \rceil} \\ &= n^{\lceil \log^2(n) \rceil + 1} \cdot \sum_{i=0}^{\lceil \log^2(n) \rceil} \binom{\lceil \log^2(n) \rceil}{i} \end{aligned}$$

Ahora bien, como ya sabemos, $\sum_{i=0}^{x} \binom{x}{i} = (1+1)^x = 2^x$, luego haciendo $x = \lceil \log^2(n) \rceil$ tenemos

$$\begin{aligned} n^{\lceil \log^2(n) \rceil + 1} \cdot \sum_{i=0}^{\lceil \log^2(n) \rceil} \binom{\lceil \log^2(n) \rceil}{i} &\leq n^{\log^2(n)+2} \cdot 2^{\lceil \log^2(n) \rceil} \\ &\leq n^{\log^2(n)+2} \cdot 2^{\log^2(n)+1} \\ &= n^{\log^2(n)+2} \cdot 2 \cdot \left(2^{\log(n)}\right)^{\log(n)} \\ &= n^{\log^2(n)+2} \cdot 2 \cdot n^{\log(n)} \\ &= 2 \cdot n^{\log^2(n)+\log(n)+2} \end{aligned}$$

Como $n > 3$ y $\log(n) \geq 2$, tenemos

$$\begin{aligned} 2 \cdot n^{\log^2(n)+\log(n)+2} &\leq n^{\log^2(n)+\log(n)+2+1} \leq n^{\log^2(n)+\log(n)+\log(n)+1} \\ &= n^{\log^2(n)+2\log(n)+1} = n^{(\log(n)+1)^2} \\ &\leq n^{(\log(n)+\log(n))^2} = n^{\log^4(n)}. \end{aligned}$$

Luego se tiene la desigualdad como queríamos probar. ∎

Finalmente, estudiemos el siguiente resultado combinatorio.

Lema 2.3. Para $n \in \mathbb{Z}$ se cumple que

$$\binom{2n+1}{n} > 2^{n+1}. \tag{2.1}$$

Demostración. Realizaremos esta prueba por inducción. Para $n = 2$, tenemos directamente

$$\binom{5}{2} = \frac{5 \cdot 4 \cdot 3 \cdot 2 \cdot 1}{1 \cdot 2} = 60 > 2^3 = 8.$$

Supongámoslo cierto para n y probémoslo para $n + 1$ viendo que se cumple

$$\binom{2(n+1)+1}{n+1} > 2^{(n+1)+1}, \text{ o sea, } \binom{2n+3}{n+1} > 2^{n+2}.$$

$$\binom{2(n+1)+1}{n+1} = \frac{(2n+3)!}{(2n+3-(n+1))! \cdot (n+1)!} = \frac{(2n+3)!}{(n+1)! \cdot (n+2)!} =$$

$$= \frac{(2n+3) \cdot (2n+2) \cdot (2n+1)!}{(n+1) \cdot (n+2) \cdot n! \cdot (n+1)!}$$

Usando la hipótesis de inducción tenemos:

$$\frac{(2n+3) \cdot (2n+2) \cdot (2n+1)!}{(n+1) \cdot (n+2) \cdot n! \cdot (n+1)!} > \frac{(2n+3) \cdot (2n+2)}{(n+1) \cdot (n+2)} \cdot 2^{n+1} = \frac{2n+3}{n+2} \cdot 2^{n+2} > 2^{n+2},$$

que es lo que queríamos probar.

3 El algoritmo AKS y su corrección

Para estudiar el desarrollo matemático del algoritmo AKS seguiremos el artículo [PinP].

3.1 Antes de empezar

El test de primalidad que expondremos en la siguiente sección se basa en una generalización del pequeño teorema de Fermat, reflejada en el siguiente lema.

Lema 3.1. Sea $a \in \mathbb{Z}$, $n \in \mathbb{N}$, $n \geq 2$ y $\gcd(a, n) = 1$. Entonces n es primo si y sólo si

$$(X + a)^n \equiv X^n + a \pmod{n}. \tag{3.1}$$

Demostración. Para $0 < i < n$, los coeficientes de X^i en $(X + a)^n - (X^n + a)$ son $\binom{n}{i} \cdot a^{n-i}$. Distingamos 2 casos: cuando n es primo y cuando n es compuesto.

Sea n primo. En este caso tenemos

$$\binom{n}{i} = \frac{n \cdot (n-1) \cdot \ldots \cdot (n-i+1)}{i!} = n \cdot M \equiv 0 \pmod{n},$$

donde $M \in \mathbb{Z}$, ya que $(i!) \nmid n$ para $i > 1$, para $i = 1$ se tiene trivialmente. Entonces $\binom{n}{i} \equiv 0 \pmod{n}$ para todo $0 < i < n$.

Sea n compuesto. Consideremos q un factor primo de su descomposición y sea $q^k \parallel n$. Tenemos que

$$\begin{aligned} \binom{n}{q} &= \frac{n \cdot (n-1) \cdot \ldots \cdot (n-q+1)}{q!} \\ &= \frac{p_1^{\alpha_1} \cdot p_2^{\alpha_2} \cdot \ldots \cdot p_l^{\alpha_l} \cdot (n-1) \cdot \ldots \cdot (n-q+1)}{q!}, \end{aligned}$$

donde $n = p_1^{\alpha_1} \cdot p_2^{\alpha_2} \cdot \ldots \cdot p_l^{\alpha_l}$ es la descomposición de n en factores primos y $p_i^{\alpha_i} = q^k$ para cierto $i \in \mathbb{N}$.

Luego se sigue

$$\begin{aligned}\binom{n}{q} &= \frac{p_1^{\alpha_1} \cdot \ldots \cdot q^k \cdot \ldots \cdot p_l^{\alpha_l} \cdot (n-1) \cdot \ldots \cdot (n-q+1)}{q \cdot (q-1) \cdot \ldots \cdot 2 \cdot 1} \\ &= \frac{p_1^{\alpha_1} \cdot \ldots \cdot q^{k-1} \cdot \ldots \cdot p_l^{\alpha_l} \cdot (n-1) \cdot \ldots \cdot (n-q+1)}{(q-1) \cdot \ldots \cdot 2 \cdot 1}\end{aligned}$$

Entonces es claro que $q^k \nmid \binom{n}{q}$. Además, q^k es coprimo con a^{n-q} dado que, por hipótesis n y a son coprimos y $q^k \mid n$, por tanto $q^k \nmid a$ y, por ende, no divide a ninguna potencia de a.

Así, el coeficiente de X^q no es 0 módulo n. De esta forma, $(X+a)^n - (X^n + a)$ no es idénticamente nulo en $\mathbb{Z}/\mathbb{Z}n$.

La identidad anterior sugiere un test de primalidad bastante simple: dado una entrada, n, escogemos un determinado a y estudiamos si (3.1) se satisface.

Sin embargo, esto nos toma un tiempo del orden de $\Omega(n)$, pues debemos evaluar n coeficientes en la parte izquierda de la ecuación, en el peor de los casos. Para reducir esta complejidad podemos evaluar (3.1) módulo un polinomio de la forma $X^r - 1$ para un r lo suficientemente pequeño, previamente escogido. Esto es, veremos si se satisface la siguiente ecuación:

$$(X+a)^n \equiv X^n + a \pmod{X^r - 1, n} \tag{3.2}$$

Por el Lema 3.1 es claro que todo primo n satisface (3.2) para todos los valores de a o r. El problema radica en que algunos n compuestos también satisfacen la ecuación para ciertos a y r.

No obstante, podemos reescribir, como veremos, la caracterización anterior obteniendo que, para un r escogido adecuadamente, si (3.2) se satisface para unos determinados a, entonces necesariamente n debe ser potencia de un primo.

El número de valores necesario para a y el valor apropiado de r vienen ambos acotados por un polinomio en $\log(n)$ y, de esta forma, conseguiremos un algoritmo en tiempo polinomial que comprueba la primalidad del n dado.

3.2 El algoritmo (I)

En esta sección plantearemos el algoritmo de nuestro test de primalidad.

Algoritmo:

Input: integer $n > 1$

1. If $(n = a^b$ for $a \in \mathbb{N}$ and $b > 1)$, output COMPOSITE.
2. Find the smallest r such that $o_r(n) > \log^2(n)$.
3. If $1 < \gcd(a, n) < n$ for some $a \leq r$, output COMPOSITE.
4. If $n \leq r$, output PRIME. (3.3)
5. For $a = 1$ to $\lfloor \sqrt{\phi(r)} \log(n) \rfloor$ do
   if $(X + a)^n \neq X^n + a \pmod{X^r - 1, n}$,
   output COMPOSITE.
6. Output PRIME.

Nuestro estudio consistirá en tratar de probar el siguiente teorema, mediante la prueba de una serie de lemas, que nos implicarán inmediatamente la corrección de nuestro algoritmo.

Teorema 3.1. *El algoritmo que acabamos de enunciar devuelve* PRIME *si y sólo si n es primo.*

La prueba de la implicación de derecha a izquierda viene dada trivialmente por el siguiente lema.

Lema 3.2. Si n es primo, el algoritmo devuelve PRIME.

Demostración. Si n es primo, entonces es claro que los puntos 1 y 3 del algoritmo nunca pueden devolver COMPOSITE. Esto es así dado que en el punto 1 jamás podrá expresarse n tal que $n = a^b$, con $b > 1$ por la propia definición de número primo y, para el punto 3, $\gcd(a, n) = 1$ si $a \neq n$ por ser primo, por lo que no cumpliría la condición para que fuese compuesto.

Por otra parte, por el Lema 3.1, el bucle del punto 5 jamás podrá devolver como resultado COMPOSITE. Así, el algoritmo identificará n como número primo en el paso 4 o 6.

Ahora pasaremos a demostrar la implicación contraria: si el algoritmo devuelve PRIME, entonces n es primo. Esta implicación no es inmediata como la anterior y requiere un poco más de trabajo.

Es claro que en el paso 4, si se devuelve PRIME, n ha de ser un número primo pues, en caso contrario, habríamos encontrado en el paso 3 un cierto factor primo no trivial, q, en la descomposición de n tal que, claramente, $\gcd(q,n) \neq 1$ y $\gcd(q,n) \neq n$. Falta pues estudiar cuándo se devuelve PRIME en el paso 6.

Para que esto suceda, tienen un papel principal los puntos 2 y 5 del algoritmo. En el punto 2 encontramos un valor de r adecuado y en el paso 5 veremos si se verifica (3.2) para ciertos valores de a.

En primer lugar probaremos la existencia de un cierto valor de r tal que cumpla la siguiente cota superior.

Lema 3.3. Existe un $r \leq \text{máx}\left\{3, \left\lceil \log^5(n) \right\rceil\right\}$ tal que $o_r(n) > \log^2(n)$.

Demostración. Para el caso $n = 2$ y $r = 3$ se tiene trivialmente, ya que

$$3 \leq \text{máx}\left\{3, \left\lceil \log^5(n) \right\rceil\right\}, \qquad o_3(2) > \log^2(2).$$

Asumamos pues $n > 2$. Así $\left\lceil \log^5(n) \right\rceil > 10$. Podemos entonces aplicar el Lema 2.1, luego tenemos $d_n \geq 2^n$.

Sean $r_1, r_2, \ldots, r_t$ los números tales que, bien $o_{r_i}(n) \leq \log^2(n)$, bien $r_i \mid n$. Notemos que, por definición, solo existe una cantidad finita de enteros r_i que verifiquen esta condición.

Notemos por

$$M = n \cdot \prod_{i=1}^{\lceil \log^2(n) \rceil} (n^i - 1).$$

Es claro que los r_j anteriores dividen a M pues, si $r_j \mid n$ también divide a M ya que $n \mid M$, y si $o_{r_j}(n) \leq \log^2(n)$ tenemos que, dado que $o_{r_j}(n)$ es el mínimo k tal que $n^k - 1 \equiv 0 \pmod{r_j}$, deducimos que $r_j \mid (n^k - 1)$, con $k \leq \left\lceil \log^2(n) \right\rceil$. Luego efectivamente $r_j \mid M$.

Por un lado tenemos la siguiente cadena de desigualdades, donde la primera de ellas se tiene por 2.2:

$$M = n \cdot \prod_{i=1}^{\lceil \log^2(n) \rceil} (n^i - 1) < n^{\log^4(n)} \leq 2^{\log^{\log^4(n)}(n)} \leq 2^{\log^5(n)}.$$

Por otra parte tenemos que, por el Lema 2.1,

$$d_{\lceil \log^5(n) \rceil} \geq 2^{\lceil \log^5(n) \rceil},$$

y, así, debe existir un número $s \leq \lceil \log^5(n) \rceil$ tal que $s \notin \{r_1, r_2, \dots, r_t\}$. Esto es debido a que, como $r_1, \dots, r_t \mid M$, entonces

$$\operatorname{lcm}\{r_1, \dots, r_t\} \mid M < 2^{\log^5(n)},$$

pero

$$\operatorname{lcm}\left\{x \mid x < \lceil \log^5(n) \rceil\right\} \geq 2^{\lceil \log^5(n) \rceil}.$$

Si $\gcd(s, n) = 1$, entonces $o_s(n) > \log^2(n)$ y hemos terminado (el propio s nos sirve). En caso contrario, si $\gcd(s, n) > 1$, entonces como $s \nmid n$ y $\gcd(s, n) \in \{r_1, \dots, r_t\}$, tendremos que $r = s/\gcd(s, n) \notin \{r_1, \dots, r_t\}$ y así $o_r(n) > \log^2(n)$, con lo que habríamos terminado (el entero r nos sirve).

Observación 3.1. Una vez sabemos esto, como $o_r(n) > 1$ debe existir p primo tal que $p \mid n$ y $o_r(p) > 1$. Ahora bien, tenemos $p > r$ pues, en caso contrario, habríamos obtenido una respuesta satisfactoria en los pasos 3 o 4, donde se considera $a \leq r$. Además, $\gcd(n, r) = 1$ (pues de otra forma tendríamos ya respuesta al problema de la primalidad) y por tanto $n, p \in (\mathbb{Z}/\mathbb{Z}r)^*$.

Sigamos entonces con el algoritmo y definamos l como $l = \left\lfloor \sqrt{\phi(r)} \cdot \log(n) \right\rfloor$.

Tenemos que en el paso 5 se verifican l ecuaciones en el peor de los casos. Si el algoritmo no devuelve `COMPOSITE` en este paso, tenemos (el caso $a = 0$ se satisface trivialmente):

$$(X + a)^n = X^n + a \pmod{X^r - 1, n}, \qquad \forall\, 0 \leq a \leq l.$$

Esto implica:

$$(X + a)^n \equiv X^n + a \pmod{X^r - 1, p}, \qquad \forall\, 0 \leq a \leq l, \tag{3.4}$$

con p un primo cualquiera en la descomposición de n como producto de factores primos. Por el Lema 3.1 tenemos:

$$(X+a)^p \equiv X^p + a \pmod{X^r - 1, p}, \qquad \forall\, 0 \leq a \leq l. \tag{3.5}$$

De (3.4) y (3.5) deducimos entonces lo siguiente:

$$(X+a)^{n/p} = X^{n/p} + a \pmod{X^r - 1, p}, \qquad \forall\, 0 \leq a \leq l.$$

Así, n y n/p se comportan como el primo p en la ecuación anterior. Pasemos a darle nombre a esta propiedad.

3.3 Interludio introspectivo

Definición 3.1. *Sean p un primo y r un entero que consideraremos fijados, $f(X) \in \mathbb{Z}[X]$ un polinomio y sea $m \in \mathbb{N}$. Decimos que m es introspectivo para $f(X)$ si*

$$f(X)^m \equiv f(X^m) \pmod{X^r - 1, p}.$$

Una vez dada la definición vamos a probar que los números introspectivos son cerrados bajo la multiplicación.

Lema 3.4. Si m y n son números introspectivos para $f(X)$, se tiene que mn también lo es.

Demostración. En primer lugar, como m es introspectivo para $f(X)$, tenemos:

$$f(X)^{mn} \equiv f(X^m)^n \pmod{X^r - 1, p}.$$

Además, como n también es introspectivo:

$$f(X^m)^n \equiv f(X^{mn}) \pmod{X^{mr} - 1, p}.$$

Y como $X^r - 1$ divide a $X^{mr} - 1$,

$$f(X^m)^n \equiv f(X^{m\cdot n}) \pmod{X^r - 1, p},$$

luego podemos concluir:

$$f(X)^{mn} \equiv f(X^{mn}) \pmod{X^r - 1, p}.$$

Es más, para un m fijado, el conjunto de polinomios para los cuales m es introspectivo es también cerrado bajo la multiplicación. Veámoslo en el siguiente lema:

Lema 3.5. Si m es introspectivo para $f(X)$ y para $g(X)$, entonces también lo es para $f(X) \cdot g(X)$.

Demostración. Tenemos

$$\big(f(X) \cdot g(X)\big)^m = f(X)^m \cdot g(X)^m \equiv f(X^m) \cdot g(X^m) = (f \cdot g)(X^m) \pmod{X^r - 1, p},$$

pues m es introspectivo para $f(X)$ y $g(X)$. ∎

3.4 El algoritmo (II)

Los dos últimos lemas y las observaciones que hemos hecho al final de la sección 3.2 implican que todo número del conjunto

$$I = \left\{ \left(\frac{n}{p}\right)^i \cdot p^j \mid i, j \geq 0 \right\}$$

es introspectivo para cada polinomio del conjunto

$$P = \left\{ \prod_{a=0}^{l} (X + a)^{e_a} \mid e_a \geq 0 \right\},$$

ya que p y n/p lo son para cada $(X + a)$. Definamos pues dos grupos basándonos en los conjuntos anteriores.

El primer grupo que definiremos será el conjunto de todos los residuos de elementos de I, módulo r. Esto es, claramente, subgrupo de $(\mathbb{Z}/\mathbb{Z}r)^*$ pues, como ya hemos visto, $\gcd(n, r) = \gcd(p, r) = 1$. Notemos por G a dicho grupo y fijemos $|G| = t$. Además, n y p (sus clases módulo r, para ser precisos) generan G y, como $o_r(n) > \log^2(n)$, tenemos que $|G| = t > \log^2(n)$.

Definamos ahora el segundo grupo, que denotaremos W. Para ello necesitamos conocer ciertas nociones básicas sobre polinomios ciclotómicos, que se encuentran reflejadas en la sección 2.2. Sea $Q_r(X)$ el r-ésimo polinomio ciclotómico sobre $\mathbb{F}_p$. El polinomio $Q_r(X)$ divide a $X^r - 1$ y factoriza por tanto en factores irreducibles de grado $o_r(p)$, por el teorema 2.2. Sea $h(X)$ uno de esos factores irreducibles.

Como tenemos que $o_r(p) > 1$ por la observación 3.1, el grado de $h(X)$ debe ser mayor que uno. Una vez sabemos esto, definiremos W como el conjunto de los residuos de los polinomios del conjunto P módulo $(h(X), p)$. Es claro que W está generado como grupo multiplicativo por

$$\left\{ X, X+1, X+2, \ldots, X+l \right\} \subset \mathbb{F} = \mathbb{F}_p[X]/\langle h(X) \rangle$$

y, por tanto, W se puede entender como un subgrupo de $\mathbb{F}^*$. A continuación daremos una cota inferior para el cardinal del grupo W.

Lema 3.6. En las condiciones anteriores, $|W| \geq \binom{t+l}{t-1}$.

Demostración. Como $h(X)$ es un factor del polinomio ciclotómico $Q_r(X)$, X es una raíz r-ésima de la unidad en $\mathbb{F}$, pues $h(X) \mid Q_r(X)$ y en $\mathbb{F}$, $Q_r(X) \equiv 0$.

Ahora vamos a probar que dados cualesquiera dos polinomios distintos de grado menor que t en P, ambos se corresponden con elementos diferentes de W. Sean $f(X)$ y $g(X)$ dos polinomios que cumplan las dos condiciones. Supongamos, por reducción al absurdo, que $f(X) = g(X)$ en $\mathbb{F}$ y consideremos $m \in I$.

Tenemos, pues, $f(X)^m = g(X)^m$ en $\mathbb{F}$. Como todo elemento de I es introspectivo para cada elemento de P, tenemos en particular que m es introspectivo para f y g, y como $h(X)$ divide a $X^r - 1$, tenemos $f(X^m) = g(X^m)$ en $\mathbb{F}$. Por lo tanto X^m es raíz del polinomio $q(Y) = f(Y) - g(Y)$, $\forall m \in G$.

Como $\gcd(m, r) = 1$, ya que G es un subgrupo de $(\mathbb{Z}/\mathbb{Z}r)^*$, cada X^m es una raíz r-ésima de la unidad. De esta forma tendremos $|G| = t$ raíces distintas de $q(Y)$ en $\mathbb{F}$. Sin embargo, el grado de $q(Y)$ ha de ser menor que t debido a la elección de f y g. Pero esto es una contradicción con el hecho de que tiene t raíces distintas. Luego $f(X) \neq g(X)$ en $\mathbb{F}$.

Sea entonces $i \neq j$ en $\mathbb{F}_p$ con $1 \leq i, j \leq l$, siendo

$$l = \left\lfloor \sqrt{\phi(r)} \cdot \log(n) \right\rfloor < \sqrt{r} \log(n) < r$$

y $p > r$. Entonces $X, X+1, X+2, \ldots, X+l$ son todos elementos distintos en $\mathbb{F}$. Por otra parte, $X + a \neq 0$ en $\mathbb{F}$, para todo a, con $0 \leq a \leq l$, pues el grado de h es mayor que 1. Por lo tanto existen al menos $l+1$ polinomios de grado uno en W. De esta forma, existen al menos $\binom{t+l}{t-1}$ polinomios distintos de grado menor que t en W. Luego concluimos que $|W| \geq \binom{t+l}{t-1}$. ∎

Ahora bien, en el caso de que n no sea potencia de un primo p, además de estar $|W|$ acotado inferiormente, también podemos acotarlo superiormente.

Lema 3.7. Si n no es potencia de p, entonces $|W| \leq n^{\sqrt{t}}$.

Demostración. Consideremos el subconjunto $J \subseteq I$ dado por

$$J = \left\{ \left(\frac{n}{p}\right)^i p^j \quad \middle| \quad 0 \leq i, j \leq \left\lfloor \sqrt{t} \right\rfloor \right\}.$$

Si $n \neq p^k$ para algún k, entonces J contiene $\left(\lfloor \sqrt{t} \rfloor + 1\right)^2 > t$ números distintos, pues $\left\lfloor (\sqrt{t})^2 + 2 \cdot \sqrt{t} + 1 \right\rfloor > t$. Como $|G| = t$, al menos dos elementos en J deben ser iguales módulo r, pues el número de elementos de J es mayor que el de G. Sean dichos números m_1 y m_2 tal que $m_1 > m_2$. Así, tenemos:

$$X^{m_1} \equiv X^{m_2} \pmod{X^r - 1}.$$

Sea $f(X) \in P$. Entonces:

$$f(X)^{m_1} \equiv f(X^{m_1}) \equiv f(X^{m_2}) \equiv f(X)^{m_2} \pmod{X^r - 1, p}.$$

Por lo que tenemos:

$$f(X)^{m_1} \equiv f(X)^{m_2}$$

en $\mathbb{F}$. De esta forma, $f(X) \in W$ es una raíz del polinomio $q(Y) = Y^{m_1} - Y^{m_2}$ en $\mathbb{F}$. Como hemos tomado $f(X)$ como un elemento arbitrario de W, $q(Y)$ ha de tener al menos $|\, W \,|$ raíces distintas en $\mathbb{F}$. Por lo tanto, el grado de $q(Y)$ es m_1, pero

$$m_1 \leq \left(\frac{n}{p} \cdot p\right)^{\lfloor \sqrt{t} \rfloor} \leq n^{\sqrt{t}}.$$

Esto prueba que $|W| \leq n^{\sqrt{t}}$. ∎

Una vez hemos probado estas cotas que estiman el tamaño de W, estamos listos para probar la corrección del algoritmo. Demostremos pues la implicación que falta.

Lema 3.8. Si el algoritmo devuelve `PRIME`, entonces n es primo.

Demostración. Supongamos pues que el algoritmo devuelve `PRIME`. Por 3.6, para $t = |G|$ y $l = \left\lfloor \sqrt{\phi(r)} \cdot \log(n) \right\rfloor$, tenemos:

$$|W| \geq \binom{t+l}{t-1} \geq \binom{l + 1 + \lfloor \sqrt{t} \log(n) \rfloor}{\lfloor \sqrt{t} \log(n) \rfloor}$$

pues $t > \sqrt{t} \cdot \log(n)$. Ahora bien, como $l = \left\lfloor \sqrt{\phi(r)} \cdot \log(n) \right\rfloor \geq \left\lfloor \sqrt{t} \cdot \log(n) \right\rfloor$:

$$\binom{l+1+\lfloor \sqrt{t} \cdot \log(n) \rfloor}{\lfloor \sqrt{t} \cdot \log(n) \rfloor} \geq \frac{\left(l+1+\left\lfloor \sqrt{t} \cdot \log(n) \right\rfloor\right) \cdot \ldots \cdot (l+2)}{\left\lfloor \sqrt{t} \cdot \log(n) \right\rfloor \cdot \ldots \cdot 2 \cdot 1} \geq$$

$$\geq \frac{\left(2 \cdot \left\lfloor \sqrt{t} \cdot \log(n) \right\rfloor + 1\right) \cdot \ldots \cdot \left(\left\lfloor \sqrt{t} \cdot \log(n) \right\rfloor + 2\right)}{\left\lfloor \sqrt{t} \cdot \log(n) \right\rfloor \cdot \ldots \cdot 2 \cdot 1} \geq \binom{2\lfloor \sqrt{t} \cdot \log(n) \rfloor + 1}{\lfloor \sqrt{t} \cdot \log(n) \rfloor}$$

Teniendo en cuenta que $\left\lfloor \sqrt{t} \cdot \log(n) \right\rfloor > \left\lfloor \log^2(n) \right\rfloor \geq 1$, y haciendo uso de 2.3, se sigue:

$$\binom{2\lfloor \sqrt{t} \cdot \log(n) \rfloor + 1}{\lfloor \sqrt{t} \cdot \log(n) \rfloor} > 2^{\left\lfloor \sqrt{t} \cdot \log(n) \right\rfloor} \geq n^{\sqrt{t}},$$

con lo que tenemos $|W| > n^{\sqrt{t}}$.

Por otra parte, por 3.7 sabemos que $|W| \leq n^{\sqrt{t}}$, si $n \neq p^k$ para algún k. Luego, ha de existir cierto $k > 0$ para el que se cumple $n = p^k$. Pero si $k > 1$ el algoritmo devuelve `COMPOSITE` en el paso 1, por tanto $k = 1$. Entonces podemos concluir que $n = p$ y el algoritmo devuelve `PRIME`.

Por tanto, gracias a los lemas 3.8 y 3.2 hemos probado el teorema 3.1 que nos garantiza que el algoritmo devuelve `PRIME` si y sólo si n es primo.

4 Análisis de la complejidad

En esta sección estudiaremos la complejidad del algoritmo. Para ello tendremos en cuenta, según puede observarse en [MCA], que:

- Sumar dos números binarios n y m, donde la longitud de n es mayor a la de m, es $\mathcal{O}(\log(n))$,
- Multiplicar n por m tiene una complejidad de $\mathcal{O}(\log(n)\log(m))$, y
- Dividir n entre m tiene una complejidad de $\mathcal{O}(\log^2(n))$.

Además, la complejidad de estas operaciones entre un par de polinomios de grado t es de $O(t \cdot \log^2(n))$.

Sin embargo, en el artículo original se considera que la complejidad tanto de la suma como del producto y división de dos números binarios n y m, siendo n el de mayor longitud, es $\widetilde{\mathcal{O}}(\log(n))$. Es más, debido a esto se observa que esas mismas operaciones entre dos polinomios de grado t con coeficientes del orden de $\mathcal{O}(\log(n))$ pueden ser realizadas en un tiempo de $\mathcal{O}(t \cdot \log(n))$. Debido a todo esto se enuncia el siguiente teorema:

Teorema 4.1. *La complejidad asintótica del algoritmo es* $\widetilde{\mathcal{O}}\left(\log^{21/2}(n)\right)$.

La prueba de este teorema puede verse en [PinP]. Nosotros probaremos el siguiente teorema, análogo al anteriormente enunciado, bajo las condiciones expuestas al comienzo de la sección. Dicho teorema nos da una complejidad levemente mayor a la ya expuesta.

Teorema 4.2. *La complejidad del algoritmo es de* $\mathcal{O}\left(\log^{23/2}(n)\right)$.

Demostración. El primer paso del algoritmo es:

Si $n = a^b$ para ciertos $a \in \mathbb{N}$ y $b > 1$, entonces `COMPOSITE`.

Para comprobar que n es igual a a^b para ciertos a y b debemos calcular aproximadamente b raíces para $b \leq \log(n)$. Lo que nos da una complejidad de [PPT]

$$\mathcal{O}\left(\log^3(n)\log\log\log(n)\right).$$

Sin embargo, dicha complejidad puede mejorarse si en lugar de evaluar las b raíces comprobamos si n es una potencia perfecta módulo un primo pequeño. Luego podemos concluir que este paso se lleva a cabo en un tiempo de $\mathcal{O}(\log^2(n))$ [PPT].

El segundo paso consiste en encontrar el menor r tal que $o_r(n) > \log^2(n)$. La idea es probar con distintos valores de r e ir comprobando si se cumple que

$$n^k \not\equiv 1 \pmod{r},\ \forall k \leq \log^2(n).$$

En primer lugar, como se observa en [TCYC], sabemos que hacer $a^n \pmod{m}$ tiene una complejidad de

$$\mathcal{O}\left(\log(n)\log^2(m)\right).$$

En nuestro caso debemos calcular, para cada valor de r, $n^k \pmod{r}$, por tanto tenemos:

$$\mathcal{O}\left(\log(k)\log^2(r)\right).$$

luego, en nuestro caso, como $k \leq \log^2(n)$ tenemos que

$$\mathcal{O}\left(\log\left(\log^2(n)\right)\log^2(n)\right) = \mathcal{O}\left(\log^4(n)\right).$$

Además, por el lema 3.3, tenemos que sólo hay que probar, a lo más, con $O(\log^5(n))$ valores diferentes de r. Por ende, la complejidad total de este paso será

$$\mathcal{O}\left(\log^4(n)\cdot r\right) = \mathcal{O}\left(\log^4(n)\log^5(n)\right) = \mathcal{O}\left(\log^9(n)\right).$$

El tercer paso consiste en comprobar si $1 < \gcd(a,n) < n$ para algún $a \leq r$ y, en caso afirmativo, devolver `COMPOSITE`. En este caso, debemos hacer el máximo común divisor de r números a lo sumo, pues $a \leq r$. Para cada máximo común divisor debemos realizar $\mathcal{O}(\log(n))$ divisiones y, como cada división tiene una complejidad de $\mathcal{O}(\log^2(n))$, concluimos que la complejidad de cada máximo común divisor es de $\mathcal{O}(\log(n)\log^2(n)) = \mathcal{O}(\log^3(n))$ [TCYC]. Por tanto la complejidad de este paso será de

$$\mathcal{O}\left(r\cdot\log^3(n)\right) = \mathcal{O}\left(\log^5(n)\cdot\log^3(n)\right) = \mathcal{O}\left(\log^8(n)\right),$$

por el lema 3.3.

El cuarto paso consiste en verificar si $n \leq r$ y, en caso afirmativo el algoritmo devolverá PRIME. Luego la complejidad es simplemente $\mathcal{O}(\log(n))$, pues he de hacer una única comprobación.

En el quinto paso tenemos el bucle siguiente:

Para $a = 1$ hasta $\lfloor \sqrt{\phi(r)} \log(n) \rfloor$
  Si $(X + a)^n \not\equiv X^n + a \pmod{X^r - 1, n}$
  Entonces COMPOSITE.

En este paso debemos comprobar si se verifican $\lfloor \sqrt{\phi(r)} \log(n) \rfloor$ ecuaciones. En cada una de dichas ecuaciones necesitamos $\mathcal{O}(\log(n))$ multiplicaciones de grado r y coeficientes de tamaño $\mathcal{O}(\log(n))$ [PCR]. Luego, por un razonamiento similar al del paso 3 tenemos que resolver cada ecuación, lo que requiere una complejidad de

$$\mathcal{O}\left(r \log(n) \log^2(n)\right) = \mathcal{O}\left(r \log^3(n)\right).$$

Además, como tenemos $\lfloor \sqrt{\phi(r)} \log(n) \rfloor$ ecuaciones, la complejidad de este paso será:

$$\begin{aligned} \mathcal{O}\left(r\sqrt{\phi(r)} \log(n) \log^3(n)\right) &= \mathcal{O}\left(r\sqrt{\phi(r)} \log^4(n)\right) = \mathcal{O}\left(r^{\frac{3}{2}} \log^4(n)\right) \\ &= \mathcal{O}\left((\log^5(n))^{3/2} \log^4(n)\right) = \mathcal{O}\left(\log^{23/2}(n)\right), \end{aligned}$$

empleando el lema 3.3 y la definición de $\phi(r)$.

Por tanto, podemos concluir que la complejidad del algoritmo ha de ser:

$$\mathcal{O}\left(\log^2(n)\right) + \mathcal{O}\left(\log^9(n)\right) + \mathcal{O}\left(\log^8(n)\right) + \mathcal{O}\left(\log(n)\right) + \mathcal{O}\left(\log^{23/2}(n)\right),$$

esto es, $\mathcal{O}\left(\log^{23/2}(n)\right)$. ∎

Los autores exponen que la complejidad del algoritmo puede ser disminuida si se mejora la estimación de r. De hecho, la cota óptima se dará cuando $r = \mathcal{O}(\log^2(n))$ y, en este caso, la complejidad del algoritmo pasará a ser $\widetilde{\mathcal{O}}(\log^6(n))$. Para ello se apoyan en dos conjeturas muy conocidas, la *conjetura de Artin* y la *conjetura de densidad de primos de Sophie Germain*, las cuales nosotros únicamente vamos a comentar, sin entrar en mucho detalle.

La *conjetura de Artin* nos dice que dado cualquier $n \in \mathbb{N}$ tal que no es un cuadrado perfecto, el número de primos $q \leq m$ tal que $o_q(n) = q - 1$ es asintóticamente

$A(n)(m/\log(n))$, donde $A(n)$ es una constante denominada la *constante de Artin*. En el caso de que la conjetura fuese cierta lo único que se sabe acerca de dicha constante es que $A(n) > 0.35$.

Los *primos de (Sophie) Germain* son aquellos primos q tal que $2q + 1$ es también primo. La *conjetura de densidad de primos de Germain* afirma que el número de primos de Germain menores que m es asintóticamente $2C_2 m/\log^2(m)$, donde C_2 es una constante denominada la *constante de primos gemelos*, que se estima que tenga un valor aproximado de 0.66.

5 | Un ejemplo concreto

En esta sección haremos, a partir de determinado número n, un seguimiento en profundidad del funcionamiento de nuestro algoritmo. Veremos lo que sucede explícitamente en cada paso para llegar finalmente a saber si n es o no primo.

Consideremos $n = 197$.

Primer paso

Lo que nuestro algoritmo hace en primer lugar a la hora de enfrentarse al problema es tratar de encontrar un cierto $a \in \mathbb{N}$ y un cierto $b > 1$ tal que $n = 197 = a^b$. Si encuentra dichos valores devuelve `COMPOSITE`, en caso contrario llevamos a cabo el siguiente paso.

Para ello, para cada valor de a debemos calcular $\log_a(197)$ (con un método numérico estándar), cuyo entero más próximo, si hay alguno razonablemente cerca, tomaremos como b, y luego comprobaremos si $a^b = 197$. Si se cumple dicha igualdad tenemos que n es compuesto. Si no, seguiremos el mismo procedimiento para el siguiente valor de a. Es claro que debemos realizar dichos cálculos para $a < \sqrt{197} \approx 14$, pues $a^b > 197$ para $a > 14$ y $b > 1$. Veámoslo:

En primer lugar, es claro que, $a \neq 1$. Luego razonemos para $a \geq 2$.

Sea $n = 197$.

Sea $a = 2$;
Tenemos que $\log_2(197) = 7.62$.
Pero 7.62 no es próximo a un entero $b \Rightarrow$ No existe b tal que $2^b = 197$.

Sea $a = 3$;

Tenemos que $\log_3(197) = 4.80$.
Pero 4.80 no es próximo a un entero $b \Rightarrow$ No existe b tal que $3^b = 197$.

Sea $a = 4$;
Tenemos que $\log_4(197) = 3.81$.
Pero 3.81 no es próximo a un entero $b \Rightarrow$ No existe b tal que $4^b = 197$.

Sea $a = 5$;
Tenemos que $\log_5(197) = 3.28$.
Pero 3.28 no es próximo a un entero $b \Rightarrow$ No existe b tal que $5^b = 197$.

Sea $a = 6$;
Tenemos que $\log_6(197) = 2.94 \approx 3$.
Pero $6^3 = 216 \Rightarrow$ No existe b tal que $6^b = 197$.

Sea $a = 7$;
Tenemos que $\log_7(197) = 2.71$.
Pero 2.71 no es próximo a un entero $b \Rightarrow$ No existe b tal que $7^b = 197$.

Sea $a = 8$;
Tenemos que $\log_8(197) = 2.54$.
Pero 2.54 no es próximo a un entero $b \Rightarrow$ No existe b tal que $8^b = 197$.

Sea $a = 9$;
Tenemos que $\log_9(197) = 2.40$.
Pero 2.40 no es próximo a un entero $b \Rightarrow$ No existe b tal que $9^b = 197$.

Sea $a = 10$;
Tenemos que $\log_{10}(197) = 2.29$.
Pero 2.29 no es próximo a un entero $b \Rightarrow$ No existe b tal que $10^b = 197$.

Sea $a = 11$;
Tenemos que $\log_{11}(197) = 2.20$.
Pero 2.20 no es próximo a un entero $b \Rightarrow$ No existe b tal que $11^b = 197$.

Sea $a = 12$;
Tenemos que $\log_{12}(197) = 2.12$.
Pero 2.12 no es próximo a un entero $b \Rightarrow$ No existe b tal que $12^b = 197$.

Sea $a = 13$;
Tenemos que $\log_{13}(197) = 2.06 \approx 2$.
Pero $13^2 = 169 \Rightarrow$ No existe b tal que $13^b = 197$.

Sea $a = 14$;
Tenemos que $\log_{14}(197) = 2.00$.
Pero $14^2 = 196 \Rightarrow$ No existe b tal que $14^b = 197$.

Por tanto no existe ningún $a \in \mathbb{N}$ ni $b > 1$ tal que $a^b = 197$. Luego pasamos al siguiente paso. Este paso lo hemos hecho por inspección, pero en la práctica habría que usar un algoritmo más eficaz tipo [DPP].

Tras realizar los cálculos de este paso con SAGE, observamos que emplea un tiempo de 6.11ms.

Segundo paso

Ahora debemos encontrar el menor r tal que $o_r(n) > \log^2(n)$, es decir, el menor r que verifique que el primer $k > 0$ con $197^k \equiv 1 \pmod{r}$ cumpla

$$k > \log^2(197) \approx 58.09.$$

Así pues, el r que buscamos ha de satisfacer que $o_r(197) > \log^2(197) \approx 58.09$. Como $o_r(n) \mid \#(\mathbb{Z}/\mathbb{Z}_r)^* = \phi(r)$, necesitamos que r sea tal que $|\phi(r)| > 58$.

Comencemos pues con $r = 59$, pues es claro que para $r \leq 58$ no se cumple dicha propiedad. Veámoslo:

Sea $r = 59$.
Como $\phi(59) = 58$,
Sus divisores son $1, 2, 29, 58$;
Luego basta probar con 2 y 29
Si $k = 2$, $197^2 = 46 \pmod{59}$.
Si $k = 29$, $197^{29} = 1 \pmod{59}$.
Pero $29 < \log^2(197)$.

Sea $r = 60$.
Como $\phi(60) < 58$, $o_{60}(197) < 58.09$.

Sea $r = 61$.
Como $\phi(61) = 60$,
Sus divisores son $1, 2, 3, 4, 5, 6, 10, 12, 15, 20, 30, 60$;

Luego basta probar con 20 y 30.
Si $k = 20$, $197^{20} = 13 \pmod{61}$.
Si $k = 30$, $197^{30} = 1 \pmod{61}$.
Pero $30 < \log^2(197)$.

Sea $r = 62$.
Como $\phi(62) = 30$, $o_{62}(197) < 58.09$.

Sea $r = 63$.
Como $\phi(63) = 36$, $o_{63}(197) < 58.09$.

Sea $r = 64$.
Como $\phi(64) = 32$, $o_{64}(197) < 58.09$.

Sea $r = 65$.
Como $\phi(65) = 48$, $o_{65}(197) < 58.09$.

Sea $r = 66$.
Como $\phi(66) = 20$, $o_{66}(197) < 58.09$.

Sea $r = 67$.
Como $\phi(67) = 66$,
Sus divisores son $1, 2, 3, 6, 11, 22, 33, 66$;
Luego basta probar con 6, 22, 33 y 66.
Si $k = 6$, $197^6 = 9 \pmod{67}$.
Si $k = 22$, $197^{22} = 29 \pmod{67}$.
Si $k = 33$, $197^{33} = 66 \pmod{67}$.
Si $k = 66$, $197^{66} = 1 \pmod{67}$.
Luego $66 > \log^2(197)$.

Por tanto hemos encontrado nuestro $r = 67$, tal que

$$o_{67}(197) > \log^2(197).$$

Tras realizar los cálculos de este paso con SAGE, observamos que emplea un tiempo de 2.83ms.

Tercer paso.

En este paso debemos comprobar si existe un $a \leq r$ tal que $1 < \gcd(a, n) < n$. Luego en nuestro caso trataremos de encontrar $a \leq 67$ tal que $1 < \gcd(a, 197) < 197$. En caso de encontrarlo, nuestro algoritmo devolverá `COMPOSITE`, de otra forma continuará con el siguiente paso.

Observación 5.1. Para todo n lo suficientemente grande se cumple que:

$$1, ..., r \leq \log^5(n) < n$$

donde la primera desigualdad se deduce del lema 3.3.

Sin embargo, en nuestro ejemplo se tiene que $n = 197 < \log^5(n) \approx 25725.24$, luego nos bastará con comprobar que $1 < \gcd(a, 197) < 197$ para $a \leq \sqrt{n} \approx 14$.

Comencemos para $a = 2$.

Si $a = 2$, $\gcd(2, 197) = 1$;
Si $a = 3$, $\gcd(3, 197) = 1$;
Si $a = 4$, $\gcd(4, 197) = 1$;
Si $a = 5$, $\gcd(5, 197) = 1$;
Si $a = 6$, $\gcd(6, 197) = 1$;
Si $a = 7$, $\gcd(7, 197) = 1$;
Si $a = 8$, $\gcd(8, 197) = 1$;
Si $a = 9$, $\gcd(9, 197) = 1$;
Si $a = 10$, $\gcd(10, 197) = 1$;
Si $a = 11$, $\gcd(11, 197) = 1$;
Si $a = 12$, $\gcd(12, 197) = 1$;
Si $a = 13$, $\gcd(13, 197) = 1$;
Si $a = 14$, $\gcd(14, 197) = 1$;

Luego no existe un $a \leq 197$ tal que $1 < \gcd(a, 197) < 197$. Por tanto no podemos concluir que n es compuesto y debemos pasar al siguiente paso del algoritmo.

Tras realizar los cálculos de este paso con SAGE, observamos que emplea un tiempo de 0.32ms.

Cuarto paso.

Este paso consiste en comprobar si nuestro n es menor que el r calculado en el paso 2. En caso de que esto sea cierto se devolverá `PRIME`, de otra manera se seguirá con el paso 5. Luego, como $n = 197 > 67 = r$, continuaremos con el siguiente paso.

Tras realizar los cálculos de este paso con SAGE, observamos que emplea un tiempo de 11.4μs.

Quinto paso.

Una vez llegado a este punto, sólo nos queda comprobar para cada a entre 1 y $\lfloor\sqrt{\phi(r)}\log(n)\rfloor$ si se tiene

$$(X+a)^n \not\equiv X^n + a \qquad (\text{mod } X^r - 1, n).$$

En caso de verificarse para un cierto valor de a, el algoritmo devolverá `COMPOSITE`. En caso contrario pasará al paso 6. En nuestro ejemplo tomaremos a entre 1 y $61 = \lfloor\sqrt{\phi(67)}\log(197)\rfloor$ para comprobar si se verifica que $(X+a)^{197} \not\equiv X^{197} + a$ (mod $X^{67} - 1, 197$) para todo valor de a.

Si $a = 1$;
$(X+1)^{197} \equiv X^{197} + 1 \pmod{X^{67} - 1, 197}$
Luego no se devuelve `COMPOSITE`.

Si $a = 2$;
$(X+2)^{197} \equiv X^{197} + 2 \pmod{X^{67} - 1, 197}$
Luego no se devuelve `COMPOSITE`.

Si $a = 3$;
$(X+3)^{197} \equiv X^{197} + 3 \pmod{X^{67} - 1, 197}$
Luego no se devuelve `COMPOSITE`.

Si $a = 4$;
$(X+4)^{197} \equiv X^{197} + 4 \pmod{X^{67} - 1, 197}$
Luego no se devuelve `COMPOSITE`.

Si $a = 5$;
$$(X + 5)^{197} \equiv X^{197} + 5 \pmod{X^{67} - 1, 197}$$
Luego no se devuelve `COMPOSITE`.

Si $a = 6$;
$$(X + 6)^{197} \equiv X^{197} + 6 \pmod{X^{67} - 1, 197}$$
Luego no se devuelve `COMPOSITE`.

Si $a = 7$;
$$(X + 7)^{197} \equiv X^{197} + 7 \pmod{X^{67} - 1, 197}$$
Luego no se devuelve `COMPOSITE`.

Si $a = 8$;
$$(X + 8)^{197} \equiv X^{197} + 8 \pmod{X^{67} - 1, 197}$$
Luego no se devuelve `COMPOSITE`.

Si $a = 9$;
$$(X + 9)^{197} \equiv X^{197} + 9 \pmod{X^{67} - 1, 197}$$
Luego no se devuelve `COMPOSITE`.

Si $a = 10$;
$$(X + 10)^{197} \equiv X^{197} + 10 \pmod{X^{67} - 1, 197}$$
Luego no se devuelve `COMPOSITE`.

Si $a = 11$;
$$(X + 11)^{197} \equiv X^{197} + 11 \pmod{X^{67} - 1, 197}$$
Luego no se devuelve `COMPOSITE`.

Si $a = 12$;
$$(X + 12)^{197} \equiv X^{197} + 12 \pmod{X^{67} - 1, 197}$$
Luego no se devuelve `COMPOSITE`.

Si $a = 13$;
$$(X + 13)^{197} \equiv X^{197} + 13 \pmod{X^{67} - 1, 197}$$
Luego no se devuelve `COMPOSITE`.

Si $a = 14$;
$$(X + 14)^{197} \equiv X^{197} + 14 \pmod{X^{67} - 1, 197}$$
Luego no se devuelve `COMPOSITE`.

Si $a = 15$;
$$(X + 15)^{197} \equiv X^{197} + 15 \pmod{X^{67} - 1, 197}$$

Luego no se devuelve `COMPOSITE`.

Si $a = 16$;
$(X + 16)^{197} \equiv X^{197} + 16 \pmod{X^{67} - 1, 197}$
Luego no se devuelve `COMPOSITE`.

Si $a = 17$;
$(X + 17)^{197} \equiv X^{197} + 17 \pmod{X^{67} - 1, 197}$
Luego no se devuelve `COMPOSITE`.

Si $a = 18$;
$(X + 18)^{197} \equiv X^{197} + 18 \pmod{X^{67} - 1, 197}$
Luego no se devuelve `COMPOSITE`.

Si $a = 19$;
$(X + 19)^{197} \equiv X^{197} + 19 \pmod{X^{67} - 1, 197}$
Luego no se devuelve `COMPOSITE`.

Si $a = 20$;
$(X + 20)^{197} \equiv X^{197} + 20 \pmod{X^{67} - 1, 197}$
Luego no se devuelve `COMPOSITE`.

Si $a = 21$;
$(X + 21)^{197} \equiv X^{197} + 21 \pmod{X^{67} - 1, 197}$
Luego no se devuelve `COMPOSITE`.

Si $a = 22$;
$(X + 22)^{197} \equiv X^{197} + 22 \pmod{X^{67} - 1, 197}$
Luego no se devuelve `COMPOSITE`.

Si $a = 23$;
$(X + 23)^{197} \equiv X^{197} + 23 \pmod{X^{67} - 1, 197}$
Luego no se devuelve `COMPOSITE`.

Si $a = 24$;
$(X + 24)^{197} \equiv X^{197} + 24 \pmod{X^{67} - 1, 197}$
Luego no se devuelve `COMPOSITE`.

Si $a = 25$;
$(X + 25)^{197} \equiv X^{197} + 25 \pmod{X^{67} - 1, 197}$
Luego no se devuelve `COMPOSITE`.

Si $a = 26$;
$$(X + 26)^{197} \equiv X^{197} + 26 \pmod{X^{67} - 1, 197}$$
Luego no se devuelve `COMPOSITE`.

Si $a = 27$;
$$(X + 27)^{197} \equiv X^{197} + 27 \pmod{X^{67} - 1, 197}$$
Luego no se devuelve `COMPOSITE`.

Si $a = 28$;
$$(X + 28)^{197} \equiv X^{197} + 28 \pmod{X^{67} - 1, 197}$$
Luego no se devuelve `COMPOSITE`.

Si $a = 29$;
$$(X + 29)^{197} \equiv X^{197} + 29 \pmod{X^{67} - 1, 197}$$
Luego no se devuelve `COMPOSITE`.

Si $a = 30$;
$$(X + 30)^{197} \equiv X^{197} + 30 \pmod{X^{67} - 1, 197}$$
Luego no se devuelve `COMPOSITE`.

Si $a = 31$;
$$(X + 31)^{197} \equiv X^{197} + 31 \pmod{X^{67} - 1, 197}$$
Luego no se devuelve `COMPOSITE`.

Si $a = 32$;
$$(X + 32)^{197} \equiv X^{197} + 32 \pmod{X^{67} - 1, 197}$$
Luego no se devuelve `COMPOSITE`.

Si $a = 33$;
$$(X + 33)^{197} \equiv X^{197} + 33 \pmod{X^{67} - 1, 197}$$
Luego no se devuelve `COMPOSITE`.

Si $a = 34$;
$$(X + 34)^{197} \equiv X^{197} + 34 \pmod{X^{67} - 1, 197}$$
Luego no se devuelve `COMPOSITE`.

Si $a = 35$;
$$(X + 35)^{197} \equiv X^{197} + 35 \pmod{X^{67} - 1, 197}$$
Luego no se devuelve `COMPOSITE`.

Si $a = 36$;
$$(X + 36)^{197} \equiv X^{197} + 36 \pmod{X^{67} - 1, 197}$$

Luego no se devuelve `COMPOSITE`.

Si $a = 37$;
$$(X + 37)^{197} \equiv X^{197} + 37 \pmod{X^{67} - 1, 197}$$
Luego no se devuelve `COMPOSITE`.

Si $a = 38$;
$$(X + 38)^{197} \equiv X^{197} + 38 \pmod{X^{67} - 1, 197}$$
Luego no se devuelve `COMPOSITE`.

Si $a = 39$;
$$(X + 39)^{197} \equiv X^{197} + 39 \pmod{X^{67} - 1, 197}$$
Luego no se devuelve `COMPOSITE`.

Si $a = 40$;
$$(X + 40)^{197} \equiv X^{197} + 40 \pmod{X^{67} - 1, 197}$$
Luego no se devuelve `COMPOSITE`.

Si $a = 41$;
$$(X + 41)^{197} \equiv X^{197} + 41 \pmod{X^{67} - 1, 197}$$
Luego no se devuelve `COMPOSITE`.

Si $a = 42$;
$$(X + 42)^{197} \equiv X^{197} + 42 \pmod{X^{67} - 1, 197}$$
Luego no se devuelve `COMPOSITE`.

Si $a = 43$;
$$(X + 43)^{197} \equiv X^{197} + 43 \pmod{X^{67} - 1, 197}$$
Luego no se devuelve `COMPOSITE`.

Si $a = 44$;
$$(X + 44)^{197} \equiv X^{197} + 44 \pmod{X^{67} - 1, 197}$$
Luego no se devuelve `COMPOSITE`.

Si $a = 45$;
$$(X + 45)^{197} \equiv X^{197} + 45 \pmod{X^{67} - 1, 197}$$
Luego no se devuelve `COMPOSITE`.

Si $a = 46$;
$$(X + 46)^{197} \equiv X^{197} + 46 \pmod{X^{67} - 1, 197}$$
Luego no se devuelve `COMPOSITE`.

Si $a = 47$;
$$(X + 47)^{197} \equiv X^{197} + 47 \pmod{X^{67} - 1, 197}$$
Luego no se devuelve `COMPOSITE`.

Si $a = 48$;
$$(X + 48)^{197} \equiv X^{197} + 48 \pmod{X^{67} - 1, 197}$$
Luego no se devuelve `COMPOSITE`.

Si $a = 49$;
$$(X + 49)^{197} \equiv X^{197} + 49 \pmod{X^{67} - 1, 197}$$
Luego no se devuelve `COMPOSITE`.

Si $a = 50$;
$$(X + 50)^{197} \equiv X^{197} + 50 \pmod{X^{67} - 1, 197}$$
Luego no se devuelve `COMPOSITE`.

Si $a = 51$;
$$(X + 51)^{197} \equiv X^{197} + 51 \pmod{X^{67} - 1, 197}$$
Luego no se devuelve `COMPOSITE`.

Si $a = 52$;
$$(X + 52)^{197} \equiv X^{197} + 52 \pmod{X^{67} - 1, 197}$$
Luego no se devuelve `COMPOSITE`.

Si $a = 53$;
$$(X + 53)^{197} \equiv X^{197} + 53 \pmod{X^{67} - 1, 197}$$
Luego no se devuelve `COMPOSITE`.

Si $a = 54$;
$$(X + 54)^{197} \equiv X^{197} + 54 \pmod{X^{67} - 1, 197}$$
Luego no se devuelve `COMPOSITE`.

Si $a = 55$;
$$(X + 55)^{197} \equiv X^{197} + 55 \pmod{X^{67} - 1, 197}$$
Luego no se devuelve `COMPOSITE`.

Si $a = 56$;
$$(X + 56)^{197} \equiv X^{197} + 56 \pmod{X^{67} - 1, 197}$$
Luego no se devuelve `COMPOSITE`.

Si $a = 57$;
$$(X + 57)^{197} \equiv X^{197} + 57 \pmod{X^{67} - 1, 197}$$

Luego no se devuelve `COMPOSITE`.

Si $a = 58$;
$(X + 58)^{197} \equiv X^{197} + 58 \pmod{X^{67} - 1, 197}$
Luego no se devuelve `COMPOSITE`.

Si $a = 59$;
$(X + 59)^{197} \equiv X^{197} + 59 \pmod{X^{67} - 1, 197}$
Luego no se devuelve `COMPOSITE`.

Si $a = 60$;
$(X + 60)^{197} \equiv X^{197} + 60 \pmod{X^{67} - 1, 197}$
Luego no se devuelve `COMPOSITE`.

Si $a = 61$;
$(X + 61)^{197} \equiv X^{197} + 61 \pmod{X^{67} - 1, 197}$
Luego no se devuelve `COMPOSITE`.

Por lo tanto no podemos concluir que n sea compuesto, luego aplicamos el paso 6.

Tras realizar los cálculos de este paso con SAGE, observamos que emplea un tiempo de 9.08ms.

Sexto paso.

Finalmente concluimos que n es primo y, por ende, el algoritmo devuelve `PRIME`.

Es evidente que esta no es una implementación optimizada, pues el tiempo necesario para determinar si n es primo es muy elevado. Es más, es claro que el quinto paso es el que más tiempo consume pues depende del valor de r (a mayor r, más iteraciones deberemos realizar y por tanto dicho paso consumirá un tiempo mayor). Esto hace que el algoritmo, en la práctica, puede no ser muy útil para números considerablemente grandes.

Bibliografía

[PinP] M. AGRAWAL, N. KAYAL, N. SAXENA, *Primes is in P.* Annals of Mathematics, **160** (2004), 781–793.

[PCR] M. AGRAWAL, S. BISWAS, *Primality and identity testing via Chinese remaindering*. Journal of the ACM **50** (2003), 429–443.

[PPT] BACH, E., SORENSON, J. *Sieve algorithms for perfect power testing.* Algorithmica **9** (1993), 313–328.

[DPP] D.J. BERNSTEIN, *Detecting perfect powers in essentially linear time.* Mathematics of Computation **67** (1998), 1253-1283.

[BFE] F. BORNEMANN, *Primes is in P: A Breakthrough for "Everyman"*. Notices of the AMS, May 2003.

[TCYC] Dpto. de Álgebra (US), Notas de *Teoría de Códigos y Criptografía.* Universidad de Sevilla.

[TAN] Dpto. de Análisis Matemático (US), Notas de *Teoría Analítica de Números.* Universidad de Sevilla.

[FFA] R. LIDL, H. NIEDERREITER, *Introduction to Finite Fields and their Applications.* Cambridge Univ. Press, Cambridge, 1986.

[CIP] M. NAIR, *On Chebyshev-type inequalities for primes.* Amer. Math. Monthly **89** (1982), 126–129.

[MCA] J. VON ZUR GATHEN, J. GERHARD, *Modern Computer Algebra.* Cambridge Univ. Press, Cambridge, 1999.