UNIVERSIDAD
DE GRANADA

Facultad de Ciencias / Escuela Técnica Superior de Ingeniería
Informática y Telecomunicaciones

Doble Grado en Matemáticas e Ingeniería
Informática

TRABAJO DE FIN DE GRADO

Primalidad en tiempo polinomial

Presentado por:
Francisco Gallego Salido

Tutor:
Francisco Torralbo Torralbo
Departamento de Geometría y Topología

Curso académico 2021-2022

Primalidad en tiempo polinomial

Francisco Gallego Salido

Francisco Gallego Salido *Primalidad en tiempo polinomial*.
Trabajo de fin de Grado. Curso académico 2021-2022.

Responsable de tutorización

Francisco Torralbo Torralbo
Departamento de Geometría y Topología

Doble Grado en Matemáticas e Ingeniería Informática

Facultad de Ciencias / Escuela Técnica Superior de Ingeniería Informática y Telecomunicaciones

Universidad de Granada

Declaración de originalidad

D./Dña. Francisco Gallego Salido

Declaro explícitamente que el trabajo presentado como Trabajo de Fin de Grado (TFG), correspondiente al curso académico 2021-2022, es original, entendida esta, en el sentido de que no ha utilizado para la elaboración del trabajo fuentes sin citarlas debidamente.

En Granada a 17 de noviembre de 2021

Fdo: Francisco Gallego Salido

Dedicatoria

A mi familia por haberme apoyado incluso en los momentos más duros y a mi pareja por ser el mayor apoyo en mi vida.

Índice general

Agradecimientos

A mi tutor por ayudarme a solucionar todas mis dudas a lo largo del trabajo y a mis amigos por haberme acompañado en este viaje de explicarles el trabajo.

Summary

Prime numbers are of special importance when it comes to Mathematics in general and, in specific, the branch of Number Theory. Their applications go from purely theoretic results to practical uses like criptography, which is the base of the security on the Internet.

Primality testing has been extensively studied throughout history, and specially during the second half of the 20^{th} and 21^{st} centuries with the formalitation of complexity theory by *Alan Turing*.

There has been many attempts to come up with efficient techniques to prove the primality of a number. The definition of prime numbers provides by itself a primality test: check if some number below $\sqrt{n}$ divides n. This test has complexity $O(\sqrt{n})$, which is far from ideal. We want a test that runs in logarithmic time. A great attempt for that is the *Little Fermat's Theorem*, which states that if n is prime, then $a^n \equiv a \mod (n)$ for every $a \in \mathbb{Z}$. With that, we can check some values of a and see if the congruence holds. If it doesn't hold for some value, then n is definitely composite. Otherwise it is probably prime. This almost gives us an efficient test which runs in $\Omega(\log(n))$.

Unfortunately, there exists some numbers for which the congruence holds for every value of a. They are called *Charmichael Numbers*. Therefore, this test is not valid, but we can make it work with a generalization of the *Little Fermat's Theorem*.

Let $n > 1$ and $a \in \mathbb{Z}$. Then n is prime if, and only if,

$$(X + a)^n \equiv X^n + a \mod (n)$$

where X is an indeterminate variable.

This property leads us to a general, deterministic and inconditional primality test: try the congruence for some a and check if it holds. The problem with this approach is that it gives us a test with complexity $\Omega(n)$ due to the fact that we need to evaluate n coefficients in the left hand side of the congruence.

We can speed up the process if we reduce the amount of coefficients to evaluate by restricting the congruence to the ring $\mathbb{Z}_n[X]/(X^r - 1)$, where r is sufficiently small. This way, the congruence above is transformed into the next one below

$$(X + a)^n \equiv X^n + a \mod (n, X^r - 1)$$

This congruence still holds if n is prime for every $a \in \mathbb{Z}$ and every r. However, it also holds for some values of a and r when n is composite. This property can be restored if we appropriately choose r and test it for some values of a. We are going to prove that r and a are $O(\log^c(n))$ for some constant c, which leads us to a deterministic polynomial algorithm.

The algorithm is of great interest when it comes to the theory, as it is the first polynomial, deterministic, general and uncondicional primality test. This opens the door to the development of better algorithms that also run in polynomial time.

However, this algorithm falls behind some other tests that are currently used. For example, the *Miller-Rabin* test is of probabilistic nature, but its runtime is superior, which makes it more eligible when it comes to test for primality in branches like cryptography, where we need to test really big numbers (normally bigger than 1024 bits) really fast.

Even other primality tests that are deterministic and non-polynomial, like the ones based in elliptic curves, perform better than the **AKS** in most useful cases.

An empirical study and comparison with other probabilistic tests is going to let us jump to that conclusion.

The algorithm is easy to implement, but some care must be taken when dealing with polynomial multiplication. A good algorithm for polynomial multiplication is needed so that the test is not completely useless. We will see that a bad algorithm for polynomial multiplication can lead to an efficiency of $O^{\sim}(\log^{31/2}(n))$ instead of $O^{\sim}(\log^{21/2}(n))$. This is going to make the test struggle for inputs bigger than 16 bits.

The implementation uses C++ as the main programming language for its raw speed and control over the memory. For multiprecision, **GMP** is the library that we are going to use to implement the algorithm, as it has been extensively tested and is one of the most used libraries. It is written in C, and it has a C++ API, which makes the integration easier.

Introducción

En este trabajo vamos a estudiar un test de primalidad general, polinómico, determinista e incondicional llamado **AKS** en honor a sus autores.

1. Antecedentes

Durante la historia, la búsqueda de métodos eficientes para comprobar la primalidad de un número ha sido intensa.

Empezando por tests como el basado en el *Pequeño Teorema de Fermat* y pasando por generalizaciones del mismo, como el test de *Miller-Rabin*, se ha intentando buscar tests polinómicos y deterministas que no dependan de resultados no probados.

Existen otros intentos de tests deterministas que, a pesar de ser muy rápidos y eficientes, fallan en que no son polinómicos para todas las entradas, como pueden ser los tests basado en curvas elípticas.

Gracias a una generalización del *Pequeño Teorema de Fermat*, un grupo de matemáticos fue capaz de encontrar una manera de adaptar dicha generalización, de modo que el test resultante tuviera complejidad polinómica y fuera determinista.

Se trata del test **AKS**, el cual ha sido el primero con todas las características deseables en un test de primalidad: general, determinista, polinómico e incondicional.

El test no solo cumple con todos los requisitos para ser un test ideal, sino que además no requiere de herramientas avanzadas de matemáticas para probar su validez, ya que la prueba se basa casi exclusivamente en propiedades de los polinomios ciclotómicos y de los cuerpos.

Este test sienta las bases para la búsqueda de tests más eficientes que puedan ejecutarse en un tiempo polinómico.

2. Objetivos

Los objetivos de este trabajo son varios.

Primero realizaremos una análisis matemático del algoritmo, cuyos objetivos propuestos son los siguiente:

- Presentar las herramientas matemáticas necesarias para poder estudiar el algoritmo.
- Presentar los antecedentes que han llevado al desarrollo de dicho algoritmo.

- Dar una descripción clara y concisa del algoritmo, describiendo cada uno de sus pasos.
- Probar la validez del algoritmo. Esto es, comprobar que el algoritmo determina que su entrada es un número primo si, y solo si, dicha entrada representa un número que es primo.
- Comprobar que el algoritmo tiene una complejidad algorítmica polinómica en la cantidad de cifras de la entrada.

Una vez sentadas las bases del algoritmo **AKS**, pasaremos a analizar el algoritmo de manera empírica. Nuestros objetivos serán:

- Presentar las herramientas informáticas para desarrollar dicho algoritmo.
- Realizar una implementación lo más eficiente posible de dicho algoritmo.
- Realizar comparaciones de dicho algoritmo con otros tests de primalidad usados en la actualidad y comprobar cómo se comporta respecto a ellos.

3. Técnicas utilizadas

Para el desarrollo de este trabajo serán necesarias herramientas básicas de matemáticas, entre las que se incluyen:

- Espacios matemáticos como anillos, grupos, cuerpos, etc.
- Resultados básicos de álgebra como el máximo común divisor, combinatoria o aritmética modular.
- Propiedades básicas de los polinomios ciclotómicos.
- Definición del comportamiento asintótico de funciones.

Añadidas a estas herramientas, necesitaremos también herramientas para poder desarrollar una implementación, de manera que podamos analizar el algoritmo empíricamente. Entre estas se incluyen:

- Conocimiento de programación. En específico, usaremos el lenguaje de programación C++.
- Sistemas de compilación el código fuente. En nuestro caso usaremos CMake, ya que hace la portabilidad más fácil.
- Dependencias externas. En específico, usaremos el manejador de paquetes Conan, el cual nos servirá para poder integrar fácilmente librerías que usaremos, como GMP, MPFR o NTL. Estas librerías serán la base del algoritmo **AKS**
- Necesitaremos también software para generar gráficas, las cuales ayudan a visualizar mejor lso resultados obtenidos. Usaremos *Gnuplot*, aunque existen otros como la librería *Matplotlib* de *Python*.

4. Fuentes Principales

Este trabajo está basado en el trabajo de los matemáticos *Manindra Agrawal*, *Neeraj Kayal* y *Nitin Saxina*: "PRIMES is in P" [KS04].

En este paper se muestra todo lo relacionado con el algoritmo **AKS** (en honor a las iniciales de los tres autores), desde la prueba de su validez y su complejidad polinómica, terminando en mejoras que podrían mejorar aún más el algoritmo propuesto.

Parte I.

Análisis Matemático del Algoritmo AKS

En esta parte vamos a comprobar que el algoritmo **AKS** es correcto junto con la correspondiente demostración de ello.

Primero explicaremos un poco las herramientas que utilizaremos para poder entender la demostración del algoritmo.

Luego hablaremos de los tests de primalidad en general, su utilidad y describiendo algunos de los que se usan hoy en día.

Finalmente nos centraremos en la descripción del algoritmo **AKS**, explicando su historia y su demostración, así como algunas mejoras que se han hecho del mismo desde su publicación.

1. Herramientas Matemáticas

En este primer capítulo vamos a describir herramientas básicas del álgebra que nos van a servir para entender mejor los conceptos y demostraciones que presentaremos más adelante.

Empezaremos dando una introducción a distintos espacios de trabajo como los cuerpos, los grupos, los anillos, etc.

Después introduciremos los conceptos de combinatoria y álgebra modular junto con algunas propiedades que nos serán imprescindibles para presentar el trabajo de la manera más clara posible.

Haremos también una pequeña presentación de los polinomios ciclotómicos, los cuales son de vital importancia y nos serán muy útiles en la demostración del algoritmo **AKS**.

Haremos una breve introducción a la *Hipótesis Generaliza de Riemann*, la cual nos servirá para ver resultados mejorados a los que presentaremos.

Finalmente haremos una introducción a la notación que usaremos para estudiar la complejidad algorítmica.

1.1. Estructuras Algebraicas

Para trabajar con muchos de los elementos que presentaremos a continuación, es necesario hacerlo bajo diversas estructuras matemáticas con ciertas propiedades.

Presentaremos las que más nos servirán en el desarrollo del trabajo.

1.1.1. Anillos

Sea R un conjunto no vacío y sean dos aplicaciones $(+), (\cdot)$ definidas por

$$\begin{aligned} (+) : R \times R &\to R \\ (a, b) &\mapsto a + b, \\ (\cdot) : R \times R &\to R \\ (a, b) &\mapsto ab, \end{aligned}$$

Dichas aplicaciones las llamaremos suma y producto respectivamente.

Definición 1.1. La tupla $(R, +, \cdot)$ es un anillo si cumple las siguientes propiedades:

- **Asociatividad de la suma**. Para todo $a, b, c \in R$, se cumple que $(a + b) + c = a + (b + c)$.

- **Conmutatividad de la suma**. Para todo $a, b \in R$, se cumple que $a + b = b + a$.
- **Elemento neutro para la suma**. Existe $e \in R$ tal que $a + e = a$ para todo $a \in R$. Dicho elemento se suele representar con el número cero, 0.
- **Inverso para la suma**. Para todo $a \in R$ existe $b \in R$ tal que $a + b = 0$. Dicho elemento se suele conocer como el opuesto de a, y se representa con $-a$.
- **Asociatividad del producto**. Para todo $a, b, c \in R$, se cumple que $(ab)c = a(bc)$.
- **Elemento neutro para el producto**. Existe $e \in R$ tal que $ae = ea = a$ para todo $a \in R$. Dicho elemento se suele representar con el número uno, 1.
- **Distributividad de la suma respecto del producto**. Para todo $a, b, c \in R$, se cumplen:

$$a(b + c) = ab + ac$$
$$(a + b)c = ac + bc$$

Además, se dice que $(R, +, \cdot)$ es conmutativo si cumple:

- **Conmutatividad del producto**. Para todo $a, b \in R$, se cumple que $ab = ba$.

Vamos ahora a definir las unidades de un anillo.

Definición 1.2. Sea A un anillo conmutativo. Diremos que $a \in A$ es una unidad si tiene inverso respecto del producto. Esto es que existe $b \in A$ tal que $ab = ba = 1$ (dicho elemento se conoce como el inverso de a, y se suele representar con a^{-1}).

Al conjunto de las unidades de un anillo se le denota por $\mathcal{U}(A)$. Se dice que $a \in A$ es un divisor de cero si existe $b \in A \setminus \{0\}$ tal que $ab = 0$.

A la operación de sumar el opuesto podemos llamarla *resta*, y se representa con el símbolo $-$ (es decir, $a + (-b) = a - b$). A la operación de multiplicar por el inverso podemos llamarla *dividir*, y se representa con el símbolo $/$ (es decir, $ab^{-1} = a/b$).
Ahora vamos a dar algunas propiedades de los anillos.

Proposición 1.1. *Sea A un anillo. Se cumplen entonces:*

- *Los elementos neutros tanto para la suma como para el producto son únicos.*
- *El opuesto de cada elemento es único.*
- *Para todo $a \in A$, se cumple que $a0 = 0$.*
- *Para todo $a, b \in A$, se cumple que $(-a)b = -(ab) = a(-b)$.*
- *Sea $a \in \mathcal{U}(A)$, entonces su inverso es único.*
- *Sean $a, b \in \mathcal{U}(A)$, entonces $ab \in \mathcal{U}(A)$ y $(ab)^{-1} = b^{-1}a^{-1}$. Esta propiedad implica además que $\mathcal{U}(A)$ es un grupo, concepto que explicaremos más adelante.*

Dadas estas propiedades de los anillos, vamos a pasar a definir dos estructuras con propiedades muchos más deseables.

Definición 1.3. Sea A un anillo conmutativo. Diremos que A es un *dominio de integridad* si el elemento neutro para la suma, 0, es el único divisor de cero.

Esto implica que A es un dominio de integridad si, y solo si, dados $a, b \in A$ con $ab = 0$, entonces se cumple que $a = 0$ ó $b = 0$.

Definición 1.4. Sea A un anillo conmutativo. Diremos que A es un *cuerpo* si todo elemento no nulo es unidad, es decir, $\mathcal{U}(A) = A \setminus \{0\}$.

Esto implica que A es un cuerpo si, y solo si, todo elemento de $A \setminus \{0\}$ tiene inverso.

Ahora vamos a presentar algunos ejemplos de anillos.

Ejemplo 1.1. $\mathbb{Z}$ es un anillo junto con las operaciones suma y producto clásicas.

Ejemplo 1.2. Sea $n\mathbb{Z}$ el conjunto de los múltiplos de n en $\mathbb{Z}$. Es un anillo con las operaciones heredadas de $\mathbb{Z}$.

Ejemplo 1.3. Sea $\mathbb{Z}/n\mathbb{Z}$ una clase de equivalencia tal que dos elementos de $\mathbb{Z}$ son iguales si, y solo si, son múltiplos de n. A este conjunto se le suele denotar con $\mathbb{Z}_n$ y también es un anillo con las siguientes operaciones.

$$\begin{aligned} (+) : \mathbb{Z}_n \times \mathbb{Z}_n &\to \mathbb{Z}_n \\ (a, b) &\mapsto res(a + b; n), \\ (\cdot) : \mathbb{Z}_n \times \mathbb{Z}_n &\to \mathbb{Z}_n \\ (a, b) &\mapsto res(ab; n), \end{aligned}$$

donde $res(a; n)$ es el resto de dividir $a \in \mathbb{Z}$ entre n.

Así pues, si tomamos por ejemplo $\mathbb{Z}_5$, entonces 6 y 11 son equivalentes en este anillo, pues el resto de dividirlos entre 5 es el mismo.

A estos anillos se les suele conocer como *anillos modulares*.

Ejemplo 1.4. Sea A un anillo y $f \in A[x]$ un polinomio con coeficientes en A. Entonces la clase de equivalencia $A[x]/f(x)$ donde dos polinomios con coeficientes en A son equivalentes si tienen el mismo resto al dividirlos por $f(x)$, es un anillo. En particular, si A es un cuerpo y f es irreducible en A, entonces $A[x]/f(x)$ es un cuerpo.

Por ejemplo, $\mathbb{Z}_p/(x + 1)$ con p primo es un cuerpo.

Ejemplo 1.5. Sea A un anillo conmutativo, entonces el conjunto de los polinomios con coeficientes en A, $A[x]$, es también un anillo conmutativo.

Ejemplo 1.6. $\mathbb{Q}$, $\mathbb{R}$ y $\mathbb{C}$ son ejemplos de cuerpos infinitos.

Ejemplo 1.7. Todos los cuerpos finitos se denominan por $\mathbb{F}_q$ con $q = p^k$ con p primo y $k \geq 1$. Además, dos cuerpos finitos con el mismo cardinal son equivalentes.

En el desarrollo del trabajo, usaremos sobre todo los anillos modulares y los anillos modulares de polinomios.

También haremos uso del conjunto de las unidades de los anillos modulares, es decir, $\mathcal{U}(\mathbb{Z}_n)$, a veces también notados como $\mathbb{Z}_n^*$ o $\mathbb{Z}_n^\times$.

Definido el conjunto de las unidades del anillo y sabiendo que $\mathbb{Z}_n$ es un anillo, es natural definir entonces la *Función ϕ de Euler*.

Definición 1.5. Sea $n > 1$. Se define la *Función ϕ de Euler* como

$$\phi(n) = |\mathcal{U}(\mathbb{Z}_n)|,$$

donde la operación $|\cdot|$ es el cardinal de un conjunto.

Una propiedad que nos será útil más adelante sobre los cuerpos es la relacionada con las raíces de los polinomios con coeficientes en un cuerpo. Enunciamos pues la siguiente proposición.

Proposición 1.2. *Sea $f \in F[x]$ con F un cuerpo. Entonces f tiene a lo sumo tantas raíces distintas como el grado de f.*

Demostración. Haremos esta prueba por inducción sobre el grado de f, siendo este n. Entonces:

- Sea $f(x) = a \neq 0$, entonces f no tiene raíces. Del mismo modo, si $f(x) = ax + b$ con $a \neq 0$, entonces $-a^{-1}b$ es la única raíz de f.
- Supongamos ahora la proposición cierta para todos los polinomios de grado n y sea $f(x) = a_0 + a_1x + \ldots + a_nx^n + a_{n+1}x^{n+1}$ con $a_{n+1} \neq 0$, luego de grado $n+1$. Si f no tiene raíces en F, entonces la proposición se cumple trivialmente, así que supongamos que f tiene al menos una raíz $c \in F$. Entonces tenemos que $\exists g \in F[x]$ tal que $f(x) = (x - c)g(x)$.

  Es entonces claro que el grado de g es n, y por hipótesis de inducción tenemos que g tiene como mucho n raíces distintas, luego f tiene como mucho $n+1$ raíces distintas.

□

1.1.2. Grupos

Sea G un conjunto no vacío y sea $(\cdot)$ una operación interna en G definida como

$$\begin{aligned}(\cdot) : G \times G &\to G \\ (x, y) &\mapsto xy,\end{aligned}$$

a la cual llamaremos producto. Damos entonces la siguiente definición.

Definición 1.6. La pareja $(G, \cdot)$ es un grupo si se cumplen las siguientes propiedades:

- **Asociatividad**. Para todo $x, y, z \in G$, se tiene que $(xy)z = x(yz)$.

- **Elemento neutro**. Existe $e \in G$ tal que $ex = xe = x, \forall x \in G$. Dicho elemento se suele representar con el número uno, 1.
- **Inverso**. Para todo $x \in G$ existe $y \in G$ tal que $xy = yx = 1$. Dicho elemento se suele conocer como el inverso de x, y se representa con el símbolo x^{-1}.

Además, se dice que $(G, \cdot)$ es un grupo abeliano si cumple:

- **Conmutatividad**. Para todo $x, y \in G$, se tiene que $xy = yx$.

Al cardinal del conjunto G lo denominaremos *orden del grupo* G, y lo representamos por $|G|$. En palabras más simples, se trata de la cantidad de elementos distintos que contiene el grupo. Si $|G| < \infty$, entonces decimos que se trata de un *grupo finito*.

Denominaremos *orden* de un elemento al menor $k \geq 1$ tal que dicho elemento multiplicado k veces es igual a 1. Se denota por $or(a)$ con $a \in G$ un grupo cualquiera.

Algunas propiedades inmediatas y fáciles de comprobar son las siguientes.

Proposición 1.3. *Sea $(G, \cdot)$ un grupo. Entonces:*

- *El elemento neutro es único.*
- *Para cada $x \in G$, su inverso x^{-1} es único.*
- ***Involución**. Para cada $x \in G$, $(x^{-1})^{-1} = x$.*
- *Si $xx = x$ con $x \in G$, entonces $x = 1$.*
- ***Cancelación**. Sean $x, y, z \in G$, entonces:*

$$xy = xz \Rightarrow y = z$$
$$yx = zx \Rightarrow y = z$$

- *El inverso del elemento neutro es él mismo.*
- *Para todo $x, y \in G$, se cumple que $(xy)^{-1} = y^{-1}x^{-1}$.*
- *Para todo $x, y \in G$, existen únicos $u, v \in G$ tales que:*

$$xu = y$$
$$vx = y$$

Existen muchos ejemplos de grupos, como por ejemplo $\mathbb{Z}$, $\mathbb{Q}$, $\mathbb{R}$ y $\mathbb{C}$ bajo la operación de la suma.

Además de los anteriores, existen muchas categorías de grupos, entre los que podemos encontrar los grupos de permutaciones, los grupos diédricos, los cuaternios, etc. No vamos a centrarnos en ellos más, pues no los necesitaremos más adelante.

Sin embargo, y como ya dijimos anteriormente, dado un anillo conmutativo A, el conjunto de la unidades de dicho anillo, $\mathcal{U}(A)$, es un grupo. En especial, nos vamos a centrar en los anillos $\mathbb{Z}_n$ con $n > 1$ y sus correspondiente grupos de unidades, $\mathcal{U}(\mathbb{Z}_n) = \mathbb{Z}_n^*$, también conocido como grupo multiplicativo de $\mathbb{Z}_n$.

Es importante destacar lo siguiente:

Proposición 1.4. *Sea $n \in \mathbb{N}$ con $n > 1$. Entonces $|\mathbb{Z}_n^*| = \phi(n)$, donde ϕ es la función de Euler.*

1.2. Combinatoria

En esta sección vamos a presentar algunos resultados en el campo de la combinatoria, los cuales serán útiles más adelante.

Empecemos por definir la operación del binomio, la cual aparece en la fórmula de los coeficientes del binomio de Newton.

Definición 1.7. Sean $n, k \in \mathbb{Z}$ con $n \geq k \geq 0$. Entonces definimos el binomio de la forma

$$\binom{n}{k} = \frac{n!}{k!(n-k)!},$$

donde $n! = 1 \cdot 2 \cdots n$ es la operación factorial de n.

Existen muchas propiedades de los binomios, pero solo presentaremos algunas que utilizaremos en el desarrollo de la teoría.

Proposición 1.5. *Se cumplen:*

1. $$\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1} \quad \forall n \geq k > 0$$

2. $$\binom{n}{k} = \binom{n}{n-k} \quad \forall n \geq k \geq 0$$

3. ***Identidad del Palo de Hockey**. Sean n, k tales que $n \geq k \geq 0$. Entonces*

$$\sum_{i=k}^{n} \binom{i}{k} = \binom{n+1}{k+1}$$

Ahora vamos a presentar el *Binomio de Newton*, propiedad que nos vendrá muy bien en algunas demostraciones.

Teorema 1.1. *(Binomio de Newton) Sean $x, y \in \mathbb{Z}$ (a nosotros nos vale con $\mathbb{Z}$, pero x e y pueden pertenecer a otros espacios más generales), y sea $n \in \mathbb{Z}$ no negativo. Entonces se cumple*

$$(x+y)^n = \sum_{k=0}^{n} \binom{n}{k} x^k y^{n-k}$$

La demostración se puede hacer por inducción sobre n, por lo que no la vamos a detallar.

Ahora veremos algunos resultados que usaremos más adelante.

Lema 1.1.

$$\binom{2n+1}{n} > 2^{n+1} \quad \forall n \geq 2$$

Demostración. Haremos una inducción sobre n. Sea entonces pues $n = 2$, y tenemos

$$\binom{2\cdot 2+1}{2} = \binom{5}{2} = \frac{5!}{2!3!} = 10 > 8 = 2^{2+1}$$

Supuesto cierto para n, comprobemos la desigualdad para $n+1$:

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

En la penúltima desigualdad hemos aplicado la hipótesis de inducción sobre n, y la última desigualdad se deduce de que $\frac{(2n+3)}{(n+2)} = 2 - \frac{1}{n+2} > 1$. □

1.3. Máximo Común Divisor y Mínimo Común Múltiplo

En esta sección vamos a introducir un concepto que es una de las bases de la *Teoría de Números* y del *Álgebra* en general.

Damos entonces la siguiente definición.

Definición 1.8. Sean $a, b \in \mathbb{Z}$. Diremos que $d \in \mathbb{Z}$ con $d \geq 0$ es el *Máximo Común Divisor* de a y b si se cumplen:

- $d \mid a$ y $d \mid b$.
- Sea $d' \in \mathbb{Z}$ tal que $d' \mid a$ y $d' \mid b$, entonces $d' \mid d$.

Ha dicho d se le suele denotar por $mcd(n)$ (siglas en español), $\gcd(a, b)$ (siglas del nombre en inglés, *Greatest Common Divisor*) ó (a, b). Esta última suele ser la más utilizada por no depender del idioma, y será la que usaremos a lo largo del trabajo.

Como veremos más adelante en el *Algoritmo de Euclides* 1, necesitamos saber qué ocurre cuando uno de los dos parámetros es 0. Dado entonces $a \in \mathbb{Z}$, se tiene que $(a, 0) = (0, a) = |a|$ por la propia definición. Además, por convención, se tiene que $(0, 0) = 0$.

En esencia, el *Máximo Común Divisor* se puede entender como el mayor número entero positivo de manera que divide a dos números dados. Existe también el concepto opuesto, llamado *Mínimo Común Múltiplo*, que es la contraparte del *Máximo Común Divisor*, el cual podemos definir de la siguiente manera.

Definición 1.9. Sean $a, b \in \mathbb{Z}$. Diremos que $d \in \mathbb{Z}$ con $d \geq 0$ es el *Mínimo Común Múltiplo* de a y b si se cumplen:

- $a \mid d$ y $b \mid d$.
- Sea $d' \in \mathbb{Z}$ tal que $a \mid d'$ y $b \mid d'$, entonces $d \mid d'$.

Ha dicho d se le suele denotar por $mcm(n)$ (siglas en español), $lcm(a,b)$ (siglas del nombre en inglés, *Least Common Multiplier*) ó $[a,b]$. Esta última suele ser la más utilizada por no depender del idioma, y será la que usaremos a lo largo del trabajo.

La conexión entre ambos conceptos viene dada por el siguiente teorema.

Teorema 1.2. *Sean $a, b \in \mathbb{Z}$, entonces se cumple*

$$(a,b)[a,b] = |ab|$$

Del mismo modo que pasa con el *Máximo Común Divisor*, el 0 se comporta mal con esta definición, luego por convención se denota $[a,0] = [0,a] = [0,] = 0$ para todo $a \in \mathbb{Z}$.

En esencia es el menor entero de manera que es múltiplo tanto de ambos números.

Vamos a dar ahora algunas propiedades del *Máximo Común Divisor*.

Proposición 1.6. *Sean $a, b \in \mathbb{Z}$. Entonces se tiene:*

1. ***Existencia y Unicidad**. Existe un único $d \in \mathbb{Z}$ tal que $(a,b) = d$.*
2. ***Identidad de Bézout**. Existen $x, y \in \mathbb{Z}$ tales que $ax + by = (a,b)$.*
3. *$(ca, cb) = c(a,b)$ para todo $c > 0$.*
4. *Sea $c > 0$ tal que $c \mid a$ y $c \mid b$. Entonces $\left(\frac{a}{c}, \frac{b}{c}\right) = \frac{(a,b)}{c}$.*
5. *$(a,b) = (b,a) = (a,-b) = (a, b+ac)$ para todo $c \in \mathbb{Z}$.*
6. *Para todo $c \in \mathbb{Z}$ tal que $b \mid ac$, se tiene que $b \mid (a,b)c$.*
7. *Si $(a,b) = 1$ y $b \mid ac$ para algún $c \in \mathbb{Z}$, entonces $b \mid c$.*
8. *Si $b \mid a$ y sea $c \in \mathbb{Z}$ tal que $c \mid a$ y $(b,c) = 1$, entonces se tiene que $bc \mid a$.*
9. *Sea $c \in \mathbb{Z}$, entonces $(a, bc) = 1$ si, y solo si, $(a,b) = (a,c) = 1$.*

Una manera elemental de calcular el *Máximo Común Divisor* de dos números es calcular los factores primos de ambos, seleccionar aquellos que coincidan y multiplicarlos. Este método es fácil de enseñar y aplicar. Sin embargo, a medida que las entradas se hacen más grandes, calcular los factores primos puede convertirse en una tarea difícil. Es por ello que *Euclides* desarrolló un algoritmo que acelera este proceso, conocido como *Algoritmo de Euclides*. Antes de mostrar el algoritmo, vamos a dar un par de propiedades que prueban su validez.

Proposición 1.7. *Sean $a, b \in \mathbb{Z}$. Se tienen:*
Si $a, b \neq 0$ y $b \mid a$, entonces $(a,b) = |b|$.
Si $a = qb + r$ para algunos $q, r \in \mathbb{Z}$, entonces $(a,b) = (b,r)$.

Estas dos propiedades nos permiten mostrar un algoritmo para calcular el *Máximo Común Divisor* que no requiere de calcular los factores primos de ambos números, y solo se basa en los restos de la división de enteros. Mostramos entonces el algoritmo a continuación.

Algorithm 1 Algoritmo de Euclides

```
1: procedure GCD(a, b)                    ▷ Calcula el máximo común divisor de a y b
2:     if a = b = 0 then
3:         return 0
4:     end if
5:     if a = 0 then
6:         return |a|
7:     end if
8:     if b = 0 then
9:         return |b|
10:    end if
11:    return GCD(b, res(a; b))
12: end procedure
```

Este algoritmo junto con la definición de *Máximo Común Divisor* será fundamental en todo el trabajo.

1.4. Aritmética Modular

En este apartado nos vamos a centrar en la aritmética modular tanto con enteros como con polinomios. La mayoría de propiedades son las mismas, y solo distinguiremos entre ambos cuando sea necesario. En general, nos referiremos a aritmética de enteros, pero era necesario aclarar que dichas propiedades serán equivalente para polinomios (por tratarse ambos de anillos conmutativos).

Empecemos por definir lo que es una congruencia.

Definición 1.10. Sean $a, b \in \mathbb{Z}$ y $n \in \mathbb{N} \setminus \{0\}$. Diremos que a y b son *congruentes módulo* n si el resto de dividir ambos por n es el mismo.

Esto lo denotaremos por $a \equiv b \pmod{n}$, $a \equiv b \mod (n)$ ó $a \equiv_n b$. De la propia definición se sobreentiende que $b \equiv a \mod (n)$.

Es importante destacar que esta operación es una relación de equivalencia:

- **Reflexividad**. $a \equiv a \mod (n)$ para todos a, n.
- **Simetría**. Se cumple $a \equiv b \mod (n)$ y $b \equiv a \mod (n)$ para todos a, b, n.
- **Transitividad**. Si $a \equiv b \mod (n)$ y $b \equiv c \mod (n)$, entonces $a \equiv c \mod (n)$ para todos a, b, c, n.

De la definición podemos deducir varias propiedades inmediatas.

Proposición 1.8. *Sea $n \in \mathbb{N} \setminus \{0\}$. Se cumplen entonces:*

1. *Si* $a \equiv b \mod (n)$, *entonces* $a + k \equiv b + k \mod (n)$ *para todo* k.
2. *Si* $a \equiv b \mod (n)$, *entonces* $ka \equiv kb \mod (n)$ *para todo* k
3. *Si* $a \equiv b \mod (n)$ *y* $c \equiv d \mod (n)$, *entonces* $a + c \equiv b + d \mod (n)$.
4. *Si* $a \equiv b \mod (n)$ *y* $c \equiv d \mod (n)$, *entonces* $a - c \equiv b - d \mod (n)$.
5. *Si* $a \equiv b \mod (n)$ *y* $c \equiv d \mod (n)$, *entonces* $ac \equiv bd \mod (n)$.
6. *Si* $a \equiv b \mod (n)$, *entonces* $a^k \equiv b^k \mod (n)$ *para todo* k
7. *Si* $a \equiv b \mod (n)$ *y* $p \in \mathbb{Z}[x]$, *entonces* $p(a) \equiv p(b) \mod (n)$.
8. *Si* $a \equiv b \mod (\phi(n))$ *y un* k *tal que* $(k, n) = 1$, *entonces* $k^a \equiv k^b \mod (n)$.
9. *Si* $a + k \equiv b + k \mod (n)$ *para algún* k, *entonces* $a \equiv b \mod (n)$.
10. *Si* $ka \equiv kb \mod (n)$ *para algún* k *tal que* $(k, n) = 1$, *entonces* $a \equiv b \mod (n)$.
11. *Si* $ka \equiv kb \mod (kn)$ *para algún* k, *entonces* $a \equiv b \mod (n)$.
12. *Existe un único* a^{-1} *tal que* $aa^{-1} \equiv 1 \mod (n)$ *si, y solo si,* $(a, n) = 1$. *A* a^{-1} *se le llama el inverso multiplicativo de* a *módulo* n
13. *Si* $a \equiv b \mod (n)$ *y* $(a, n) = (b, n) = 1$, *entonces* $a^{-1} \equiv b^{-1} \mod (n)$.
14. *Si* $ax \equiv b \mod (n)$ *con* $(a, n) = 1$, *entonces* $x \equiv a^{-1}b \mod (n)$ *es solución de la ecuación.*

Demostración. Vamos a demostrar algunas de estas propiedades, pues muchas de ellas son casi triviales.

6 Si $a \equiv b \mod (n)$, entonces $a = qn + b$ para algún $q \in \mathbb{Z}$. Elevamos a k y nos queda por Teorema 1.1 lo siguiente

$$a^k = (qn + b)^k = \sum_{i=0}^{k} \binom{k}{i} q^i n^i b^{k-i} = b^k + \sum_{i=1}^{k} \binom{k}{i} q^i n^i b^{k-i} = b^k + n \sum_{i=0}^{k-1} \binom{k}{i} q^{i+1} n^i b^{k-i-1}$$

Llamando $q' = \sum_{i=0}^{k-1} \binom{k}{i} q^{i+1} n^i b^{k-i-1}$, tenemos que $a^k = q'n + b^k$, luego $a^k \equiv b^k \mod (n)$.

7 Se deduce de 2 y de 6.

8 Si $a \equiv b \mod (\phi(n))$, entonces $a = q\phi(n) + b$ para algún $q \in \mathbb{Z}$, luego nos queda

$$k^a = k^{q\phi(n)+b} = \left(k^{\phi(n)}\right)^q k^b \equiv k^b \mod (n)$$

En la última equivalencia hemos usado la propiedad 2 y Teorema 1.4, ya que $(k, n) = 1$.

12 Vamos a probar ambas implicaciones.

⇒ Si $aa^{-1} \equiv 1 \mod (n)$, entonces $aa^{-1} + n(-q) = 1$ para algún $q \in \mathbb{Z}$. Puesto que $ax + ny = (a, n)$ para algunos $x, y \in \mathbb{Z}$ por la *Identidad de Bézout* Proposición 1.6, entonces podemos asegurar que $(a, n) = 1$.

$\Leftarrow$ Como $(a, n) = 1$, por la *Identidad de Bézout* Proposición 1.6 tenemos que $\exists x, y \in \mathbb{Z}$ tales que $ax + ny = 1 \Rightarrow ax \equiv 1 \mod (n)$, luego hemos probado la existencia.

Supongamos ahora que existe un $b \not\equiv a^{-1} \mod (n)$ tal que $ab \equiv 1 \mod (n)$. Se tiene entonces

$$\begin{aligned} b &\equiv b(aa^{-1}) \mod (n) \\ &\equiv (ba)a^{-1} \mod (n) \\ &\equiv a^{-1} \mod (n) \end{aligned}$$

Llegamos así a una contradicción, luego a^{-1} es único.

□

Presentaremos ahora una propiedad interesante de las congruencias.

Lema 1.2. *Para todo $a, b \in \mathbb{Z}$ y para todo p primo, se tiene que $(a + b)^p \equiv a^p + b^p \mod (p)$*

Demostración. Por un lado, sabemos que $(a + b)^p = \sum_{i=0}^{p} \binom{p}{i} a^i b^{p-i}$.

Sabiendo eso, consideremos los binomios dentro de la sumatoria, pero excluyendo los casos donde $i = 0$ e $i = p$:

$$\binom{p}{i} = \frac{p!}{i!(p-i)!}$$

Como p es primo, entonces $p \nmid k!$ para todo $0 < k < p$ ó, lo que es lo mismo, $k!$ no contiene el número p en su factorización. Como $0 < i < p$ y, en consecuencia, $0 < p - i < p$, tenemos que ni $i!$ ni $(p - i)!$ contienen en su factorización a p, y por lo tanto no lo contiene el producto.

Como el binomio es un número entero, tenemos entonces que $\binom{p}{i}$ contiene en su factorización a p o, lo que es lo mismo, que es múltiplo de p. Esto último implica que, para $0 < i < p$, tenemos

$$\binom{p}{i} a^i b^{p-i} \equiv 0 \mod (p)$$

Así tenemos que

$$(a + b)^p = \sum_{i=0}^{p} \binom{p}{i} a^i b^{p-i} \equiv a^p + b^p \mod (p)$$

□

Teorema 1.3. *(Pequeño Teorema de Fermat) Sean $n \geq 0$ y p primo. Entonces se cumple que $n^p \equiv n \mod (p)$.*

Demostración. Procederemos usando inducción sobre n.

Para el caso $n = 0$ tenemos que $0^p \equiv 0 \mod (p)$, que es trivialmente cierto.

Aplicamos ahora inducción y suponemos que se cumple para n, por lo que vamos a comprobarlo para $n + 1$.

$$(n+1)^p \equiv n^p + 1^p \mod (p)$$

Usando la hipótesis de inducción sobre n y que $1^p = 1$, tenemos

$$(n+1)^p \equiv n + 1 \mod (p)$$

Es justo lo que queríamos probar. □

Del teorema que acabamos de demostrar, es evidente comprobar que, dado $n \in \mathbb{Z}$, entonces $n^{p-1} \equiv 1 \mod (p)$ para todo p primo. Este hecho nos da una pista de una generalización del pequeño teorema de Fermat, la cual fue descubierta por Euler.

Teorema 1.4. *(Teorema de Euler) Sean $n, p > 1$ con n y p coprimos, es decir, $(n, p) = 1$. Entonces se cumple que $n^{\phi(p)} \equiv 1 \mod (p)$, siendo ϕ la función de Euler.*

Aquí podemos ver que el Pequeño Teorema de Fermat es un caso particular de este teorema, pues $\phi(p) = p - 1$ si, y solo si, p es primo.

Vamos a dar ahora una definición que está íntimamente relacionada con estos resultados y que será de vital importancia más adelante.

Definición 1.11. Sean $n > 1$ y $a \in \mathbb{Z}$ tales que $(a, n) = 1$. Se define el *orden de a módulo n*, como el menor $k > 0$ tal que $a^k \equiv 1 \mod (n)$. Dicho k se denota por $ord_n(a)$

Por el *Teorema de Euler*, podemos deducir claramente que $ord_n(a) \mid \phi(n)$ para todo $a \in \mathbb{Z}$ tal que $(a, n) = 1$.

1.5. Polinomios Ciclotómicos

Lo primero que vamos a hacer es dar una propiedad de las raíces de los polinomios del estilo $x^n - a \in \mathbb{C}[x]$ con $a \in \mathbb{C}$ y $n \geq 1$.

Lema 1.3. *Sea $a \in \mathbb{C}$ y $n \geq 1$. Entonces el polinomio $x^n - a \in \mathbb{C}[x]$ tiene exactamente n raíces distintas*

Demostración. Si $n = 1$, entonces a es la única raíz, luego supongamos $n > 1$.

Supongamos ahora que existe $r \in \mathbb{C}$ raíz doble de $x^n - a$. Entonces debe existir un $g \in \mathbb{C}[x]$ tal que $x^n - a = (x - r)^2 g(x)$. Derivando ambas partes obtenemos lo siguiente

$$nx^{n-1} = (x - r)(2g(x) + (x - r)g'(x))$$

Luego tanto $x^n - a$ como nx^{n-1} son divisibles por $x - r$, luego $x - r \mid (x^n - a, nx^{n-1})$. Pero $(x^n - a, nx^{n-1}) = a$ y $x - r \nmid a$, lo cual es una contradicción y por lo tanto r no puede ser una raíz doble, luego todas las raíces de $x^n - a$ son distintas. □

A estos n números complejos (raíces de $x^n - a$) vamos a llamarlos **raíces n-ésimas** de a. Si $n = 2$, les llamamos **raíces cuadradas**, o **raíces cúbicas** si $n = 3$. Si $a = 1$, se les llama **raíces n-ésimas de la unidad**.

Para cada $n \geq 1$, dichas raíces conforman un subgrupo, $\mathbb{C}_n$, del grupo multiplicativo de los complejos, $\mathbb{C}^\times$, definido tal que:

$$\mathbb{C}_n = \{\zeta \in \mathbb{C}^\times : \zeta^n = 1\} = \left\{\cos\left(\frac{2k\pi}{n}\right) + i\ \sin\left(\frac{2k\pi}{n}\right) : k = 0, ..., n-1\right\}$$

Entre estas raíces, $\zeta_n = \cos\left(\frac{2\pi}{n}\right) + i\ \sin\left(\frac{2\pi}{n}\right)$ es llamada la **raíz n-ésima primitiva de la unidad**. A partir de aquí, es evidente comprobar que $\mathbb{C}_n = \langle\zeta_n\rangle$, lo cual lo hace un grupo cíclico de orden n generado por ζ_n.

Por otro lado, ζ_n^k es un generador de $\mathbb{C}_n$, o lo que es lo mismo, $or(\zeta_n^k) = n$ si, y solo si, $(n, k) = 1$. Por lo tanto definimos el conjunto de los generadores de $\mathbb{C}_n$ como:

$$Gen(\mathbb{C}_n) = \{\zeta \in \mathbb{C}_n : or(\zeta) = n\} = \left\{\zeta_n^k : 1 \leq k \leq n, (n, k) = 1\right\}$$

Es evidente ver que $\mathbb{C}_n$ tiene $\phi(n)$ generadores. Hacemos entonces la siguiente definición.

Definición 1.12. Sea $n \geq 1$, se define el n-ésimo polinomio ciclotómico Φ_n de la siguiente manera:

$$\Phi_n(x) = \prod_{\zeta \in Gen(\mathbb{C}_n)} (x - \zeta) = \prod_{\substack{1 \leq k \leq n \\ (n,k)=1}} (x - \zeta_n^k)$$

Dicho de otro modo, Φ_n es el polinomio mónico de grado $\phi(n)$ donde las raíces n-ésimas de la unidad son de orden n. Ahora vamos a pasar a dar algunas propiedades de estos polinomios:

Proposición 1.9. *Los n-ésimos polinomios ciclotómicos cumplen las siguientes propiedades:*

- $\Phi_n \in \mathbb{Z}[x]$
- Φ *es irreducible en* $\mathbb{Q}[x]$
- $x^n - 1 = \prod_{d|n} \Phi_d(x)$. *En particular, Φ_n es el polinomio irreducible en $\mathbb{Z}[x]$ de mayor grado que divide a $x^n - 1$ y no divide a $x^k - 1$ con $1 \leq k < n$.*
- *Si restringimos los coeficientes de Φ_n a $\mathbb{Z}_p$ con p primo y tal que $(p, n) = 1$, tenemos que Φ_n se puede factorizar en $\frac{\phi(n)}{d}$ polinomios irreducibles de grado d, donde $d = ord_n(p)$.*

1.6. Hipótesis Generalizada de Riemann

En la rama del Análisis Matemático y, en específico, la rama del Análisis en Variable Compleja, existe una conjetura muy importante propuesta por *Bernhard Riemann* en 1859, cuya popularidad es debida a su inclusión entre uno de los Problemas del Milenio por el *Clay Mathematics Institute*. Para enunciar dicha conjetura, definamos primero la *Función Zeta de Riemann*, $\zeta(s)$, con $s \in \mathbb{C}$:

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

Esta función se sabe que converge cuando la parte real de s es mayor que 1. Para los casos en los que la parte real de s sea menor o igual que 1, lo que se hace es extender analíticamente la función ζ de la siguiente manera:

$$\zeta(s) = 2^s \pi^{s-1} \sin\left(\frac{\pi s}{2}\right) \Gamma(1-s)\zeta(1-s)$$

Esta función está definida en todo $\mathbb{C} \setminus \{1\}$ (en $s = 1$ hay lo que se conoce como un *polo*). La función Γ extiende el concepto de factorial al plano complejo. Se define de la siguiente manera para s con parte real positiva:

$$\Gamma(s) = \int_0^{\infty} t^{s-1} e^{-t}\, dt$$

Como podemos ver en la propia definición de ζ, dicha función se anula para todos los enteros negativos pares. Estos ceros son más conocidos como los *ceros triviales* de la función ζ. Existen también valores de s cuya parte real se encuentra entre 0 y 1 (no incluidos) tales que $\zeta(s)$ también se anula. Estos valores son conocidos como los *ceros no triviales* de la función ζ.

Armados con este conocimiento, pasamos a enunciar la conjetura, también conocida como *Hipótesis de Riemann*.

Conjetura 1.1. *Todos los ceros no triviales de la función ζ tienen parte real igual a $\frac{1}{2}$.*

Esta conjetura, de ser cierta, implicaría profundos resultados en el ámbito de los números primos. En específico, existe una generalización de dicha conjetura, también denominada *Hipótesis Generalizada de Riemann*, la cual se enuncia para un conjunto específico de funciones llamado *Funciones-L de Dirichlet* y los *Caracteres de Dirichlet*, los cuales no vamos a definir en este trabajo. El enunciado de la conjetura es el siguiente.

Conjetura 1.2. *Sea χ un Carácter de Dirichlet. Se define L como una función-L de Dirichlet para todo $s \in \mathbb{C} \setminus \{1\}$ de la siguiente forma:*

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

Entonces, si $L(\chi, s) = 0$ y la parte real de s está entre 0 y 1 (no incluidos), la parte real de s es igual a $\frac{1}{2}$.

Es evidente comprobar que si tomamos $\chi(n) = 1$, tenemos la *Hipótesis de Riemann* 1.1.

Más adelante mencionaremos esta conjetura, cuya veracidad implicaría mejoras en la complejidad del algoritmo **AKS**.

1.7. Complejidad Algorítmica

Para poder estudiar la complejidad algorítmica del test **AKS**, tenemos que entender qué es la complejidad algorítmica como tal. Para ello usaremos la notación asintótica O, Ω y Θ.

Estas tres notaciones nos sirven para dar forma al concepto de crecimiento asintótico de una función.

Además, estas notaciones nos van a servir también para dar forma a la idea intuitiva de que el único término necesario en el comportamiento asintótico es aquel que crece más rápido.

1.7.1. Notación O

Empezaremos con el concepto intuitivo de que una función domina asintóticamente a otra según la entrada crece. Para ello damos la siguiente definición.

Definición 1.13. Sean f y g dos funciones definidas en $\mathbb{N}$, y cuyas imágenes pertenecen a $\mathbb{R}^+$. Diremos que f es de orden g, notado como $O(g(n))$, si, y solo si, $\exists k \in \mathbb{N}$ y $\exists C \in \mathbb{R}^+$ tales que se cumple lo siguiente:

$$f(n) \leq Cg(n) \quad \forall n \in \mathbb{N};\ n \geq k$$

Esta definición nos dice que una función domina a otra dada si la primera multiplicada por una constante es mayor que la segunda para toda entrada a partir de cierto punto. Veamos ahora algunos ejemplos:

Ejemplo 1.8. Probar que $f(n) = 3n^2 + 1$ es $O(n^2)$.

Tomando $k = 1$ y $C = 4$, podemos ver fácilmente usando inducción sobre n que $3n^2 + 1 \leq 4n^2\ \forall n \geq 1$, luego podemos asegurar que $3n^2 + 1 = O(n^2)$.

- Si $n = 1$, entonces $3 \cdot 1^2 + 1 = 4 \leq 4$, luego se cumple el caso inicial.
- Suponiendo cierto para n, comprobemos para $n + 1$. Entonces $3(n+1)^2 + 1 = 3n^2 + 6^n + 3 + 1 \leq 4n^2 + 6n + 3 \leq 4n^2 + 8n + 4 = 4(n+1)^2$, luego hemos probado lo que queríamos.

1.7.2. Notación Ω

Intuitivamente, el concepto de la notación Ω es el opuesto al concepto de la notación O. Lo vemos más rápido en la definición.

Definición 1.14. Sean f y g dos funciones definidas en $\mathbb{N}$, y cuyas imágenes pertenecen a $\mathbb{R}^+$. Diremos que f es de orden g, notado como $\Omega(g(n))$, si, y solo si, $\exists k \in \mathbb{N}$ y $\exists C \in \mathbb{R}^+$ tales que se cumple lo siguiente:

$$f(n) \geq Cg(n) \quad \forall n \in \mathbb{N};\ n \geq k$$

Viendo la definición, es inmediato ver que, dadas dos funciones $f, g : \mathbb{N} \to \mathbb{R}^+$, entonces $f(n) = O(g(n)) \Leftrightarrow g(n) = \Omega(f(n))$. Algunos ejemplos son:

Ejemplo 1.9. $3^n = \Omega(2^n)$

Ejemplo 1.10. $n^3 + 2n + 3 \neq \Omega(n^4)$

Realmente este concepto es exactamente igual que el anterior, solo que la acotación la hacemos por debajo en vez de por arriba.

1.7.3. Notación Θ

Este concepto no es más que una manera de indicar que dos funciones se acotan asintóticamente, o lo que es lo mismo, que crecen con la misma rapidez. También se le conoce como el "orden exacto". Para ser más exactos, esta es la definición.

Definición 1.15. Sean f y g dos funciones definidas en $\mathbb{N}$, y cuyas imágenes pertenecen a $\mathbb{R}^+$. Diremos que f es de orden exacto g, notado como $\Theta(g(n))$, si, y solo si

$$f(n) = O(g(n)) \ \wedge \ f(n) = \Omega(g(n))$$

1.7.4. Notación $O^{\sim}$

Algunas veces es complicado calcular la complejidad exacta, y puede que nos baste simplemente probar que nuestro algoritmo pertenece a una clase que sigue siendo polinómica. Por ello hacemos la siguiente definición:

Definición 1.16. Sea $f : \mathbb{N} \to \mathbb{R}^+$ y definimos $O^{\sim}(f(n)) = O(f(n) \cdot poly(\log(f(n))))$, donde $poly(n)$ es una función polinómica en n.

Con esta definición, tenemos que $O^{\sim}(\log^k(n)) = O(\log^k(n) \cdot poly(\log(\log^k(n))) = O(\log^{k+\epsilon}(n))$.

Consideramos de ahora en adelante $\log(n)$ como el logaritmo en base 2 de n, y $\ln(n)$ como el logaritmo natural (o en base e) de n.

2. Tests de Primalidad

En este capítulo vamos a dedicarnos a explicar qué son los tests de primalidad y cuál es su utilidad, además de presentar algunos de ellos.

Primero daremos una descripción general de los tests de primalidad, incluyendo su utilidad en ramas como la criptografía, la cual es de vital importancia para la seguridad en Internet.

Después haremos un repaso por la historia de los tests de primalidad, y presentaremos distintos tipos de tests de primalidad, entre los que incluiremos tanto los tests deterministas como los tests probabilísticos.

Por último nos pararemos a detallar un poco los tests de *Miller-Rabin* y *Solovay-Strassen*, el cual usaremos en nuestra comparación con el test **AKS**.

2.1. Introducción a los Tests de Primalidad

Un número decimos que es primo cuando sus únicos divisores son una unidad y el mismo número. En caso contrario diremos que es compuesto. Esto excluye a 1 y -1, pues estos son considerados unidades del conjunto de los números enteros, y no son ni primos ni compuestos. El número 0 también queda obviamente excluido. Además, considerar 1 ó -1 primos implica que hay infinitas factorizaciones de cualquier número, lo cual supone incumplir el *Teorema Fundamental de la Aritmética*. Dicho teorema afirma que todo número se puede escribir de manera única (salvo el orden) como producto de números primos.

Los números primos son de vital importancia en las matemáticas y, especialmente en el área de la Teoría de Números. De vital importancia son sobre todo las propiedades que nos permiten determinar cuándo un número es primo. Dichas propiedades son explotadas en el campo de la criptografía, rama en la que se apoya la seguridad en Internet.

Distintos tipos de tests se han ido descubriendo a lo largo de la historia. De hecho, la propia definición de un número primo nos da una manera de comprobar que un número es primo. Sea ese número n.

1. Comprobar todos los números hasta $\lfloor \sqrt{n} \rfloor$.
2. Si alguno divide a n, entonces n es compuesto.
3. Si ninguno divide a n, entonces n es primo.

Este test, aunque simple, es extremadamente lento a medida que crece el número de cifras del número a testear. Es por ello que el estudio de los tests de primalidad se centra en

encontrar tests mucho más rápidos.

Existen otras propiedades para los números primos que nos pueden indicar tests de primalidad. Una de las más conocidas, y base para muchos otros tests de primalidad, es el conocido *Pequeño Teorema de Fermat* 1.3, el cual que afirma que si p es primo, entonces $a^p \equiv a \mod (p)$ para todo $a \in \mathbb{Z}$. Un test que podemos aplicar a un número n es probar distintos valores de a y ver si se cumple la congruencia. En caso de que encontremos un a para el que la congruencia no se cumple, n será compuesto. En caso contrario, diremos que n es probablemente primo.

Este test es mucho más rápido que el anterior. De hecho es polinómico. Sin embargo no es determinista. De hecho el test falla siempre en un conjunto de números compuestos denominados de *Charmichael*, los cuales siempre cumplen el *Pequeño Teorema de Fermat*.

Dicho test, a pesar de no ser correcto, es base de muchos otros tests, como por ejemplo el test de *Miller-Rabin*, el test de *Lucas* (Apéndice B) o el test *AKS*.

2.2. Tipos de Tests de Primalidad

Existen distintos tipos de tests de primalidad según las certezas que nos dan sobre el resultado del mismo. En este ámbito se pueden destacar tres propiedades:

- **General**. Un test decimos que es general si se puede aplicar a cualquier número.
- **Determinista**. Un test se dice que es determinista si determina que un número es primo si, y solo si, dicho número es primo. En caso contrario, se suele decir que el test es probabilístico o no determinista.
- **Incondicional**. Un test se dice que es incondicional si no depende de un resultado no probado aún para ser correcto. En caso contrario, se dice que está condicionado.
- **Polinómico**. Un test se dice polinómico si tiene una complejidad polinómica en el número de dígitos de la entrada.

El primer test que vimos en el apartado anterior es general, determinista e incondicional. Desafortunadamente falla en que no es polinómico, por lo que su utilidad práctica es nula.

Sin embargo, el test basado en el *Pequeño Teorema de Fermat* es general, incondicional y polinómico, pero no es determinista (de hecho falla siempre en ciertos valores).

El famoso test de *Miller-Rabin* aleatorizado es general, incondicional y polinómico. Falla en que es probabilístico; pero con suficientes rondas de aplicar el test, se puede asegurar una probabilidad bastante baja de que el test falle. Además, *Miller* dio una variante determinista, cuya validez depende de la *Hipótesis Generalizada de Riemann* 1.2, lo cual lo hace condicionado. Explicaremos más adelante todo lo relacionado con este test.

Un test similar desarrollado por *Solovay* y *Strassen* basado en una propiedad de los números primos y el símbolo de Jacobi, $(-)$, proporciona un test probabilístico, general, incondicional

y polinómico. Este test también se puede hacer determinista si se cumple la *Hipótesis Generalizada de Riemann*. Explicaremos también este test más adelante.

Existen otros tipos de tests, como por ejemplo los basados en curvas elípticas. Dichos tests suelen ser generales, incondicionales y deterministas. También son polinómicos para la mayoría de los casos, pero no en general. Además, dichos tests proporcionan lo que se conocen como certificados de primalidad (lo normal era proporcionar certificados de composición). Explicaremos en otra sección más adelante en qué consisten los certificados.

Finalmente, llegamos al algoritmo AKS, el cual cumple las cuatro propiedades que deseamos en un test de primalidad. Es general, incondicional, determinista y polinómico, lo cual resuelve un problema que lleva muchos años abierto. Dicho algoritmo veremos más adelante que su utilidad práctica no es tanta, pues el test suele tardar demasiado comparado con otros tests presentados anteriormente. Sin embargo, su utilidad teórica es vital, pues implica que la búsqueda de tales tests es útil.

2.3. Certificados

Los tests de primalidad están diseñados para determinar cuándo un número es primo o no. Dichos tests suelen proporcionar un "testigo" de que un número es compuesto. También existen testigos de que un número es primo, pero son menos comunes y más difíciles de obtener.

A dichos testigos se les suele conocer como certificados de que un número es compuesto o primo. Ahora vamos a detallar cada tipo, cómo se pueden usar y qué tipos de certificados proporcionan los distintos algoritmos.

2.3.1. Certificados de Composición

Los certificados para comprobar que un número es compuesto nos permiten determinar rápidamente cuándo un número es compuesto.

Un ejemplo claro de certificado de que un número es compuesto es uno de sus factores no triviales. Dado un número y un factor suyo, podemos comprobar rápidamente que dicho número es, de hecho, compuesto simplemente dividiéndolo por dicho factor y comprobar que el resto es cero.

Estos certificados no tienen que ser únicamente factores primos no triviales. Un certificado de composición para n puede ser por ejemplo un $a \in \mathbb{Z}$ para el que no se cumple el *Pequeño Teorema de Fermat* 1.3, es decir, para el que $a^n \not\equiv a \mod (n)$. El test de *Miller-Rabin* proporciona un certificado similar.

Vamos a poner algunos ejemplos de ello.

Ejemplo 2.1. Sea $n = 48941 = 449 \cdot 109$. Un certificado para comprobar que n es compuesto es uno de sus factores, como por ejemplo 109. Simplemente dividiendo n por 109 podemos comprobar que, efectivamente, es compuesto.

Ejemplo 2.2. Sea $n = 341$. Un certificado de composición para n es $a = 3$, pues $a^n \equiv 168 \mod (n) \not\equiv a \mod (n)$, lo cual implica por el *Pequeño Teorema de Fermat* que $n = 341$ no puede ser primo.

Más adelante veremos también cómo se puede obtener un certificado para el test de *Miller-Rabin*.

2.3.2. Certificados de Primalidad

También existen los certificados de primalidad, esto es, un testigo de que un número es primo, de manera que podamos comprobar rápidamente que dicho número es, de hecho, primo. Estos no son tan comunes, y suelen producirse en algoritmos que utilizan curvas elípticas, *ECPP*.

En el caso de *ECPP* (*Elliptic Curves for Primality Proving*), un certificado de primalidad se puede generar de manera recursiva. La generación de dicho certificado es lo que más tiempo consume en el algoritmo. Dicho certificado, como ya hemos explicado, permite comprobar muy rápidamente si el número es primo o no.

2.4. Tests de Miller-Rabin y Solovay-Strassen

En esta sección vamos a detallar dos tests que usaremos más adelante para comparar con el algoritmo **AKS**. Ambos tests son de naturaleza probabilística, aunque si se cumple la *Hipótesis Generalizada de Riemann* 1.2 y eligiendo adecuadamente los números para los que realizar los tests, se pueden convertir en deterministas.

Ambos se basan en congruencias, y los vamos a describir a continuación, aunque nos centraremos más en el test de Miller-Rabin.

2.4.1. Test de Miller-Rabin

El test de *Miller-Rabin* es uno de los más usados actualmente tanto por su velocidad como por su fiabilidad en el campo de la criptografía y seguridad en Internet [dig13].

Dicho test se basa en una serie de relaciones de congruencias, las cuales se cumplen siempre cuando se trata de un número primo impar. Demos pues la siguiente definición.

Definición 2.1. Sea $n > 2$ y sean $s, d > 0$ con d impar tales que $n = 2^s d + 1$. Sea a un entero con $0 < a < n$ al que llamaremos *base*. Diremos entonces que n es un primo probable fuerte en base a si se cumple alguna de las siguientes congruencias:

$$\begin{aligned} a^d &\equiv 1 \mod (n) \\ a^{2^r d} &\equiv -1 \mod (n) \text{ con } 0 \leq r < s \end{aligned} \tag{2.1}$$

Si n no es primo y encontramos un a para el que no es un primo probable fuerte, entonces diremos que a es un testigo de su composibilidad. En específico, es un certificado de composición de n.

Si n no es primo y es un primo probable fuerte para algún a, entonces diremos que a es un *mentiroso fuerte*.

La idea del test yace en que, si n es un primo impar, entonces pasa el test por las siguientes dos razones.

- Como n es primo, entonces $a^{n-1} \equiv 1 \mod (n)$ por el *Pequeño Teorema de Fermat* 1.3.
- Las únicas raíces cuadradas de 1 son 1 y -1.

Vamos a empezar probando esto último.

Proposición 2.1. *Si n es un primo impar, entonces la ecuación de congruencia $x^2 \equiv 1 \mod (n)$ tiene como únicas soluciones 1 y -1.*

Demostración. Por un lado, sabemos que 1 y -1 son soluciones de la ecuación, luego nos queda ver que no hay más. Pero $x^2 - 1$ tiene como mucho 2 raíces distintas en el cuerpo $\mathbb{Z}_n$ por Proposición 1.2. Como 1 y -1 son ambos raíces de $x^2 - 1$, tienen que ser las únicas raíces, luego no puede haber más. □

Teniendo esto, enunciamos la siguiente proposición.

Proposición 2.2. *Si n es un primo impar, entonces es un primo probable fuerte en base a.*

Demostración. Sea $n = d2^s + 1$ con $d \geq 1$ impar y $s \geq 1$. Por el *Pequeño Teorema de Fermat* 1.3 sabemos que $a^{d2^s} \equiv 1 \mod (n)$.

Es claro que cada a^{d2^r} con $0 \leq r < s$ es la raíz cuadrada de $a^{d2^{r+1}}$. Como $a^{d2^s} \equiv 1 \mod (n)$, tenemos dos casos por Proposición 2.1:

- $a^{d2^{s-1}} \equiv -1 \mod (n)$. En este caso hemos acabado y n es primo probable fuerte en base a.
- $a^{d2^{s-1}} \equiv 1 \mod (n)$. En este caso, volvemos a iterar con la raíz cuadrada del término actual.

Al terminar, o encontramos algún término de la sucesión tal que la congruencia se cumpla para -1 o todas se cumplen para 1, y en particular para a^d, lo cual concluye nuestra prueba. □

Veamos ahora un ejemplo.

Ejemplo 2.3. Sea $n = 221$ y sean $d = 55$ y $s = 2$ de modo que $n = 221 = d2^s + 1 = 55 \cdot 2^2 + 1$ y sea, por ejemplo, $a = 174$. Entonces

$$a^{d2^0} = 174^{55} \equiv 47 \mod (n) \not\equiv 1 \mod (n) \not\equiv 220 \mod (n)$$
$$a^{d2^1} = 174^{110} \equiv 220 \mod (n)$$

Tenemos entonces que n es un primo probable fuerte o a es un mentiroso fuerte. Sea ahora $a = 137$.

$$a^{d2^0} = 137^{55} \equiv 188 \mod (n) \not\equiv 1 \mod (n) \not\equiv 220 \mod (n)$$
$$a^{d2^1} = 137^{110} \equiv 205 \mod (n) \not\equiv 220 \mod (n)$$

Tenemos entonces que n no ha pasado el test en base 137, luego $a = 137$ es un testigo de la composibilidad de n y 174 es en realidad un mentiroso fuerte.

Es importante notar que si n no es primo, entonces existirá alguna base para la que no se cumplan las congruencias (2.1).

Descrita ya la parte fundamental del test, vamos a describir las dos variantes de dicho test: la probabilística y la determinista condicionada.

2.4.1.1. Versión Probabilística

La versión probabilística del test de *Miller-Rabin* es muy sencilla y es probablemente uno de los tests de primalidad más utilizados en la seguridad de las comunicaciones en Internet.

La idea es simplemente comprobar si se cumplen las congruencias (2.1) para varias bases elegidas aleatoriamente. Si n pasa el test para todas ellas, diremos que n es probablemente primo con un cierto grado de fiabilidad.

Si por el contrario encontramos una base para la que n no pasa el test, entonces podemos asegurar que n es compuesto y dicha base será un testigo de su composición.

La elección se hace de manera aleatoria porque no se sabe con certeza la distribución de los testigos para un número compuesto. La cantidad de bases a probar depende del grado de fiabilidad que queramos obtener con el test. Se puede probar que si n es compuesto, entonces hay como mucho una cuarta parte de las bases para las que a es un mentiroso fuerte [Rab80], por lo que para cada base que comprobamos, la probabilidad de encontrarnos con un mentiroso fuerte es de 4^{-1}, luego si probamos k bases, obtenemos que la probabilidad de encontrar un mentiroso fuerte es 4^{-k}. Esta es la principal razón por la que el test de *Miller-Rabin* es más usado que otros tests probabilísticos que también son muy rápidos. En el caso del test de *Solovay-Strassen*, dicha probanilidad es de 2^{-k}.

En [dig13] Anexo C.3, podemos encontrar la cantidad de bases mínimas a probar para obtener una fiabilidad suficiente de la primalidad. Para números de 1024 bits se recomienda probar 40 bases, 56 para números de 2048 bits y 64 para números de 3072 bits. Estos números son algo menores si también aplicamos el test de *Lucas*.

En Apéndice A detallaremos que la complejidad de este test es $O(k \log^3(n))$ con multiplicación tradicional, ó $O^{\sim}(k \log^2(n))$ con versiones que hacen uso de la *Transformada Inversa de Fourier* (k es la cantidad de bases a probar).

2.4.1.2. Versión Determinista Condicionada

Una manera de hacer el test de *Miller-Rabin* determinista consiste en simplemente probar todas las bases entre 0 y n. Dicho test es muy ineficiente, por lo que lo ideal sería probar una cantidad de bases igual a $O(\log^{O(1)}(n)$ para poder asegurar una complejidad polinómica en el logaritmo de n.

Existe una versión que fue descubierta por *Miller* la cual hace uso de la *Hipótesis Generalizada de Riemann* 1.2. Dicha idea se basa en que, si n es compuesto, entonces el conjunto de los mentirosos fuertes a tales que $(a, n) = 1$ es un subgrupo del grupo multiplicativo de $\mathbb{Z}_n$, $\mathbb{Z}_n^*$. De este modo, si probamos todos los a de un conjunto que genere $\mathbb{Z}_n^*$, uno de ellos debe quedarse fuera del subgrupo mencionado anteriormente, luego sería un testigo de la composibilidad de n.

Asumiendo la veracidad de La *Hipótesis de Riemann Generalizada* Conjetura 1.2, se puede demostrar que las bases a probar son menores que $k = O(c \ln^2(n))$. La constante c se puede demostrar que es 2, y por lo tanto la versión determinista solo debe comprobar las congruencias (2.1) para $2 \leq a \leq \min\{n-2, \lfloor 2\ln^2(n)\rfloor\}$.

Esta versión, puesto que $k = O(\ln^2(n))$, tiene complejidad $O^{\sim}(\log^4(n))$. Para una descripción más exacta, ir a Apéndice A.

2.4.2. Test de Solovay-Strassen

Este test, aún habiendo sido muy usado, ha sido reemplazado por otros más fiables, como el ya mencionado test de *Miller-Rabin*.

Dicho test se basa también en una propiedad de las congruencias para los números primos. Dicha congruencia está basada en el *Símbolo de Jacobi*, $(-)$, el cual vamos a definir a continuación. Para ello necesitamos un par de conceptos previos.

Definición 2.2. Se dice que a es un residuo cuadrático módulo p si la ecuación $x^2 \equiv a \mod (p)$ tiene solución.

Definición 2.3. Sean a, p donde p es un primo impar. Se define el *Símbolo de Legendre* de la siguiente forma.

$$\left(\frac{a}{p}\right) = \begin{cases} 1 & \text{si } a \text{ es un residuo cuadrático módulo } p \text{ y } a \not\equiv 0 \mod (p) \\ -1 & \text{si } a \text{ no es un residuo cuadrático módulo } p \\ 0 & \text{si } a \equiv 0 \mod (p) \end{cases}$$

Con estas dos definiciones, podemos definir el *Símbolo de Jacobi*.

Definición 2.4. Sean a, n con n impar y donde $n = p_1^{e_1} \cdots p_k^{e_k}$ es su factorización. Se define el *Símbolo de Jacobi* de la siguiente forma.

$$\left(\frac{a}{n}\right) = \left(\frac{a}{p_1}\right)^{e_1} \cdots \left(\frac{a}{p_k}\right)^{e_k}$$

Cada factor $\left(\frac{a}{p_i}\right)$ es el *Símbolo de Legendre* 2.3.

Teniendo estas definiciones, podemos pasar a enunciar la congruencia que es la base de este test.

Proposición 2.3. *Si p es cualquier primo y a cualquier entero, entonces se cumple*

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

donde $\left(\frac{a}{p}\right)$ es el Símbolo de Legendre 2.3.

Puesto que el *Símbolo de Jacobi* es la generalización del *Símbolo de Legendre* para cualquier n impar, podemos comprobar si se cumple la congruencia

$$a^{\frac{(n-1)}{2}} \equiv \left(\frac{a}{n}\right) \mod (n), \tag{2.2}$$

para varias bases a con $(a, n) = 1$. Como ya vimos antes, si n es primo, entonces (2.2) se cumple para todo a. Si encontramos una base para la que no se cumple la congruencia, podemos asegurar que n es compuesto.

Si encontramos una base a para la que n no pasa el test, diremos que a es un *testigo de Euler* de la composibilidad de n. Del mismo modo, si n es compueto y pasa el test para una base a, diremos que dicha base es un *mentiroso de Euler*.

De modo parecido a como ocurría con el test de *Miller-Rabin*, al menos la mitad de las bases $a \in \mathbb{Z}_n^*$ son *testigos de Euler*. Esto da lugar a dos tests: uno probabilístico y uno determinista condicionado. Vamos a describirlos a continuación.

2.4.2.1. Versión Probabilística

Al igual que se hizo con el test de *Miller-Rabin*, el test de *Solovay-Strassen* tiene una versión probabilística.

Dicha versión funciona de la misma manera que el test de *Miller-Rabin*. Elegimos una base a de manera aleatoria y comprobamos si se cumple (2.2). Realizamos este proceso una cantidad determinada de veces. Si encontramos una base a para la que no se cumple la congruencia, podemos asegurar que n es compuesto. En caso contrario, n es probablemente primo.

Como vimos antes, al menos la mitad de las bases son *testigos de Euler*, luego en cada iteración tenemos una probabilidad de 2^{-1} de que a sea un *mentiroso de Euler*, luego la probabilidad después de k rondas es 2^{-k}, mucho mayor en contraste con la de *Miller-Rabin*, 4^{-k}.

La complejidad de este test es, usando multiplicación de enteros usando la *Transformada Rápida de Fourier* y *Exponenciación Rápida*, $O^{\sim}(k \log^2(n))$ como veremos en Apéndice A. Usando multiplicación tradicional tendríamos una complejidad de $O(k \log^3(n))$.

2.4.2.2. Versión Determinista Condicionada

Usando la misma idea que en el test de *Miller-Rabin*, en caso de que la *Hipótesis Generalizada de Riemann* sea cierta, podemos asegurar que existe un *testigo de Euler* menor o igual que

$2\log^2(n)$ [Bac90].

Podemos entonces modificar el algoritmo probabilístico para que pruebe todas las bases hasta $2\log^2(n)$. Si n es compuesto, al menos una de esas bases será un *testigo de Euler*, por lo que se determinará correctamente la primalidad de n.

En Apéndice A se comprueba que la complejidad de este test es $O^{\sim}(\log^4(n))$

3. Test AKS. El Algoritmo y su Validez

Esta parte vamos a dedicarla por completo al algoritmo **AKS**.

Primero daremos una introducción a la historia del algoritmo, indicando cómo se llegó a su descubrimiento.

Después presentaremos el algoritmo en pseudocódigo, especificando claramente cada uno de sus pasos.

Luego pasaremos a comprobar que el algoritmo es correcto. Esto es, demostrar que dicho algoritmo solo determina que su entrada se trata de un número primo si, y solo si, dicha entrada representa un número primo.

Finalmente presentaremos mejoras que se han realizado al algoritmo desde su publicación.

3.1. Historia del Algoritmo

Como ya vimos en el capítulo anterior, de entre los distintos tests de primalidad, el *Pequeño Teorema de Fermat* es la base de muchos de los utilizados hoy en día. De hecho, el propio teorema casi nos daba un test eficiente, pero que desafortunadamente falla siempre en un conjunto de números, denominados de *Charmichael*.

Para poder intentar conseguir un test que sea determinista, será necesario encontrar una propiedad que nos proporcione mayores garantías sobre las propiedades de los números primos. Es por ello que presentamos la siguiente identidad, la cual es una generalización del *Pequeño Teorema de Fermat*.

Proposición 3.1. *Sean $a \in \mathbb{Z}$ y $n \in \mathbb{N} \setminus \{0\}$ con $n > 1$ tales que $(a, n) = 1$. Sean además $(X + a)^n, X^n + a \in \mathbb{Z}[X]$. Entonces n es primo si, y solo si, se cumple*

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

Demostración. Por el teorema del binomio, sabemos que para $0 < i < n$, el coeficiente X^i del polinomio $(X + a)^n - (X^n + a)$ es $\binom{n}{i} a^{n-i}$.

- $\Rightarrow$) Supongamos que n es primo. Sabemos que $(X + a)^n \equiv X^n + a^n \mod (n)$ por Lema 1.2, y como n es primo, tenemos que $(X + a)^n \equiv X^n + a \mod (n)$ por el *Pequeño Teorema de Fermat* 1.3.
- $\Leftarrow$) Supongamos que $(X + a)^n \equiv X^n + a \mod (n)$ y que n es compuesto. Dado que n es compuesto, sea q un factor primo de n y sea k tal que $q^k | n$ y $q^{k+1} \nmid n$. Entonces tenemos que $q^k \nmid \binom{n}{q}$ dado que $q \nmid m$ con $n - q < m < n$, lo que implica que en el numerador de $\binom{n}{q}$, solo n contiene factores q (en específico k de ellos). Como en el

denominador hay al menos un factor q, tenemos que el resultado de dicho binomio es divisible, como mucho, por q^{k-1}.

Además tenemos que $(a^{n-q}, q^k) = 1$ por hipótesis, luego nos queda que $n \nmid a^{n-q}$ ni $n \nmid \binom{n}{q}$, luego el coeficiente de X^q no puede ser divisible por n, luego no es nulo en $\mathbb{Z}_n$. Esto contradice que $(X+a)^n - (X^n + a)$ sea nulo.

□

Dicha identidad así presentada nos proporciona un test de primalidad determinista: dado un $n > 1$, comprobamos si la congruencia se cumple. El problema de este test es que es muy ineficiente, pues nos obliga a evaluar, en el peor de los casos, n coeficientes del polinomio $(X+a)^n$, luego tendría eficiencia $\Omega(n)$.

Una idea para reducir la cantidad de coeficientes a evaluar está en reducir la congruencia (3.1) módulo un polinomio del tipo $X^r - 1$ para un r que haya sido convenientemente elegido. Esto nos ayuda a que la cantidad de coeficientes que tenemos que evaluar sea mucho menor. En esencia, queremos reducir (3.1) a la siguiente congruencia.

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

Dicho de otro modo, lo que queremos comprobar es que (3.1) se satisface en el anillo $\mathbb{Z}_n[X]/(X^r - 1)$ en vez de en $\mathbb{Z}_n[X]$.

Es evidente por Proposición 3.1 que si n es primo, entonces (3.2) se sigue cumpliendo en $\mathbb{Z}_n[X]/(X^r - 1)$. Sin embargo no ocurre igual al revés. Existen números compuestos n para los que la congruencia (3.2) se cumple para algunos valores de a [KS04].

Para resolver este problema, probaremos que eligiendo un r de manera conveniente y tal que si (3.2) se satisface para varios valores de a, tenemos entonces que n es una potencia de un número primo.

Además probaremos que tanto r como a tienen un tamaño polinómico en el $\log(n)$, lo cual nos permitirá probar en la segunda parte del trabajo que el algoritmo es, de hecho, polinómico.

3.2. El Algoritmo

En esta sección vamos a presentar el algoritmo **AKS**.

Algorithm 2 Algoritmo **AKS**

procedure IsPrimeAKS(n) ▷ Comprobar si $n > 1$ es un número primo
  if $n = a^b$ para algún $a \in \mathbb{N}$ y $b > 1$ **then** ▷ Paso 1
    return COMPUESTO
  end if

  Encontrar el menor r tal que $ord_r(n) > \log^2(n)$. ▷ Paso 2

  if $1 < (a, n) < n$ para algún $a \leq r$ **then** ▷ Paso 3
    return COMPUESTO
  end if

  if $n \leq r$ **then** ▷ Paso 4
    return PRIMO
  end if

  for $a = 1$ hasta $\lfloor \sqrt{\phi(r)} \log(n) \rfloor$ **do** ▷ Paso 5
    if $(X + a)^n \not\equiv X^n + a \mod (n, X^r - 1)$ **then**
      return COMPUESTO
    end if
  end for

  return PRIMO ▷ Paso 6
end procedure

Con el algoritmo ya presentado, vamos a enunciar el siguiente teorema.

Teorema 3.1. *El algoritmo 2 devuelve PRIMO si, y solo si, n es primo.*

En la siguiente sección nos dedicaremos a probar dicho teorema, lo cual probaría que el algoritmo determina correctamente y de manera determinista si un número es primo.

3.3. Validez del Algoritmo AKS

Vamos a empezar probando una de las implicaciones del teorema.

Lema 3.1. *Si n es un número primo, entonces el algoritmo devuelve PRIMO*

Demostración. Puesto que n es primo, el primer paso es imposible que devuelva COMPUESTO, pues implicaría que $n = a^b$ con $a \in \mathbb{Z}$ y $b > 1$.

Del mismo modo, como n es primo, el tercer paso es imposible que devuelva COMPUESTO porque implicaría que existe un $a \in \mathbb{Z}$ tal que $1 < (a, n) < n$, lo cual es imposible.

Finalmente, el quinto paso tampoco puede devolver COMPUESTO por Proposición 3.1.

Por lo tanto, el algoritmo solo termina en el cuarto paso o en el sexto, devolviendo así PRIMO. □

Para comprobar la otra implicación del teorema, debemos comprobar qué ocurre cuando el algoritmo termina en el sexto paso, pues si el algoritmo termina en el cuarto paso, n debe ser primo. Esto es debido a que, en caso contrario, el paso 3 habría encontrado un divisor no trivial de n.

Es por ello que en el resto de esta sección nos centraremos en el segundo y el quinto paso, que son los dos principales para demostrar la validez.

Lo primero que vamos a hacer ahora es dar una cota para r. Para ello necesitamos los siguientes dos lemas previos.

Lema 3.2. *Sea $m \geq 1$ y definimos $LCM(m)$ como el mínimo común múltiplo de $1, ..., m$. Entonces se cumple para todo $m \geq 7$*

$$LCM(m) \geq 2^m$$

Demostración. Definamos $I_{k,m}$ para $1 \leq k \leq m$ enteros tal que

$$I_{k,m} = \int_0^1 x^{k-1}(1-x)^{m-k}\,dx$$

Donde se tiene que, para ver que está bien definida, lo siguiente:

$$\begin{aligned} I_{1,1} &= \int_0^1 dx = [x]_0^1 = 1 \\ I_{1,m} &= \int_0^1 (1-x)^{m-1}\,dx = \left[-\frac{(1-x)^m}{m}\right]_0^1 = \frac{1}{m} \\ I_{m,m} &= \int_0^1 x^{m-1}\,dx = \left[\frac{x^m}{m}\right]_0^1 = \frac{1}{m} \end{aligned}$$

Tenemos entonces la siguiente cadena de igualdades:

$$\begin{aligned} I_{k,m} &= \int_0^1 x^{k-1}(1-x)^{m-k}\,dx = \\ &= \int_0^1 \sum_{i=0}^{m-k} \binom{m-k}{i} (-1)^i x^{k+i-1}\,dx = \\ &= \left[\sum_{i=0}^{m-k} \binom{m-k}{i} (-1)^i \frac{x^{k+i}}{k+i}\right]_0^1 = \\ &= \sum_{i=0}^{m-k} \binom{m-k}{i} \frac{(-1)^i}{k+i} \end{aligned}$$

Como $k+i \mid LCM(m)$ para todo $0 \leq i \leq m-k$, podemos entonces asegurar que $I_{k,m}LCM(m) \in \mathbb{Z}$ para todo $1 \leq k \leq m$. Por otro lado tenemos lo siguiente si usamos integración por partes:

$$\begin{aligned}
I_{k,m} &= \int_0^1 x^{k-1}(1-x)^{m-k}\,dx \\
&= \left[-\frac{x^{k-1}(1-x)^{m-k+1}}{m-k+1}\right]_0^1 + \frac{k-1}{m-k+1}\int_0^1 x^{k-2}(1-x)^{m-k+1} \\
&= \left[\frac{x^{k-1}(1-x)^{m-k+1}}{m-k+1}\right]_0^1 + \frac{k-1}{m-k+1} I_{k-1,m} \\
&= \frac{k-1}{m-(k-1)} I_{k-1,m} = \frac{(k-1)(k-2)}{(m-(k-1))(m-(k-2))} I_{k-2,m} = \dots \\
&= I_{1,m} \prod_{i=1}^{k-1} \frac{i}{m-i} = \frac{1}{k\binom{m}{k}}
\end{aligned}$$

Tenemos entonces que $k\binom{m}{k} \mid LCM(m)$ por ser $\frac{1}{k\binom{m}{k}} \leq 1$ para todo $1 \leq k \leq m$ y porque $I_{k,m} LCM(m) \in \mathbb{Z}$. Por lo tanto tenemos lo siguiente para $k \geq 1$:

$$\begin{aligned}
k\binom{2k}{k} &\mid LCM(2k) \\
(2k+1)\binom{2k}{k} = (k+1)\binom{2k+1}{k+1} &\mid LCM(2k+1)
\end{aligned}$$

Como además tenemos que $LCM(2k) \mid LCM(2k+1)$ y que $k \nmid (2k+1)$, podemos asegurar que $k(2k+1)\binom{2k}{k} \mid LCM(2k+1)$. Por lo tanto nos queda para $k \geq 4$:

$$LCM(2k+1) \geq k(2k+1)\binom{2k}{k} \geq k4^k \geq 2^{2k+2} \geq 2^{2k+1}$$

La segunda desigualdad se deduce usando que $\binom{2k}{k+i} = \binom{2k}{k-i} \leq \binom{2k}{k}$ para todo $0 \leq i \leq k$ y usando el *Teorema del Binomio de Newton* 1.1 de la siguiente forma:

$$4^k = (1+1)^{2k} = \sum_{i=0}^{2k}\binom{2k}{i} \leq \sum_{i=0}^{2k}\binom{2k}{k} = \binom{2k}{k}(2k+1)$$

Por otro lado, es evidente que $LCM(2k+2) \geq LCM(2k+1) \geq 2^{2k+2}$ para todo $k \geq 4$. Esto nos deja con que $LCM(m) \geq 2^m$ para todo $m \geq 9$. Los casos $m = 8$ y $m = 7$ se comprueban a mano, obteniendo así la afirmación que queríamos. □

Lema 3.3. $\lfloor \log(\lceil \log^5(k) \rceil) \rfloor + \frac{1}{2}\left(\log^4(k) - \log^2(k)\right) \leq \log^4(k)$ *para todo* $k \geq 2$.

Demostración. Primero vamos a transformar la desigualdad en otra más fuerte, que será la que probaremos:

$$\begin{aligned}
&\lfloor \log(\lceil \log^5(k) \rceil) \rfloor + \frac{1}{2}\left(\log^4(k) - \log^2(k)\right) \leq \log^4(k) \\
\iff &\lfloor \log(\lceil \log^5(k) \rceil) \rfloor - \frac{1}{2}\left(\log^4(k) + \log^2(k)\right) \leq 0 \\
\iff &2\lfloor \log(\lceil \log^5(k) \rceil) \rfloor \leq \log^4(k) + \log^2(k) \\
\iff &2(\log(\log^5(k)) + 1) \leq \log^4(k) + \log^2(k)
\end{aligned}$$

En la última implicación hemos usado la siguiente cadena de desigualdades asumiendo que $k \geq 2$:

$$\lfloor \log(\lceil \log^5(k) \rceil) \rfloor \leq \log(1 + \log^5(k)) = \log(\log^5(n)) + \log(1 + \frac{1}{\log^5(n)}) \leq \log(\log^5(n)) + 1$$

Dicha desigualdad se cumple para $k = 2$ y $k = 3$. Para comprobar que también se cumple para $k > 3$, vamos a calcular las derivadas de ambas partes, así que sea $f(x) = 2(\log(\log^5(x)) + 1)$ y sea $g(x) = \log^4(x) + \log^2(x)$. Entonces:

$$\begin{aligned}
f'(x) &= \frac{10}{x \ln(2) \ln(x)} \\
g'(x) &= \frac{2\ln(x)\left(2\ln^2(x) - \ln^2(2)\right)}{x \ln^4(2)}
\end{aligned}$$

Por lo tanto tenemos que queremos probar lo siguiente para todo $x \geq 3$:

$$\begin{aligned}
&\frac{10}{x \ln(2) \ln(x)} = \frac{10 \ln^3(2) \frac{1}{\ln(x)}}{x \ln^4(2)} \leq \frac{2\ln(x)\left(2\ln^2(x) - \ln^2(2)\right)}{x \ln^4(2)} \\
\iff &10 \ln^3(2) \frac{1}{\ln(x)} \leq 2\ln(x)\left(2\ln^2(x) - \ln^2(2)\right)
\end{aligned}$$

Podemos comprobar que la parte izquierda es decreciente y la derecha creciente, y como $f'(3) \leq g'(3)$, podemos asegurar que $f(k) \leq g(k)$ para todo $k > 3$, y como ya comprobamos que la desigualdad principal se cumple para $k = 2$ y $k = 3$, tenemos que la desigualdad se cumple para todo $k \geq 2$, como queríamos. □

Teniendo estos dos lemas, podemos pasar a enunciar el lema que nos dará una cota superior para r.

Lema 3.4. *Existe* $r \leq \max\{3, \lceil \log^5(n) \rceil\}$ *tal que* $ord_r(n) > \log^2(n)$.

Demostración. Para empezar, si $n = 2$, tenemos que $r = 3$ cumple las condiciones del lema, luego supongamos $n > 2$. Sea $B = \lceil \log^5(n) \rceil > 10$ y escogemos r de manera que sea el menor entero que no divida al siguiente producto:

$$n^{\lfloor \log(B) \rfloor} \prod_{i=1}^{\lfloor \log^2(n) \rfloor} (n^i - 1) \tag{3.3}$$

Tenemos entonces la siguiente cadena de desigualdades:

$$n^{\lfloor \log(B) \rfloor} \prod_{i=1}^{\lfloor \log^2(n) \rfloor} (n^i - 1) < n^{\lfloor \log(B) \rfloor + \frac{1}{2}(\log^4(n) - \log^2(n))} \leq n^{\log^4(n)} = 2^{\log^5(n)} \leq 2^B \leq LCM(B)$$

En la tercera desigualdad hemos usado Lema 3.3, y en la ultima hemos usado Lema 3.2. Puesto que (3.3) es menor que $LCM(B)$, debe haber algún valor $d \leq B$ de modo que no divida a (3.3). Si no fuera así, es decir, que (3.3) es divisible por todo $d \leq B$, tendríamos que (3.3) es un múltiplo común de todos esos números, luego tenemos que (3.3) es mayor o igual que $LCM(B)$, lo cual contradice la cadena anterior. Por haber elegido r como el menor número de manera que no divida a (3.3), podemos entonces asegurar que $r \leq B$.

Teniendo esto, hagamos ahora una pequeña observación. Si $m^k \leq B$ con $m \geq 2$ y $k \geq 0$, tenemos que el mayor valor posible de k sería $\lfloor \log(B) \rfloor$ (el caso en el que $m = 2$).

Habiendo hecho esta observación, entonces tenemos que (r, n) no puede ser divisible por todos los factores primos de r. En caso de que sí, sea $r = p_1^{e_1} p_2^{e_2} \cdots p_s^{e_s}$ con p_i primo y $0 \leq e_i \leq \lfloor \log(B) \rfloor$ para $1 \leq i \leq s$. Observamos que $e_i \leq \lfloor \log(B) \rfloor$ porque $r \leq B$ y por la observación hecha anteriormente. Cada $p_i \mid (r, n)$, luego $p_i \mid n$, lo cual implica que $p_i^{e_i} \mid n^{\lfloor \log(B) \rfloor}$ (esto por ser $e_i \leq \lfloor \log(B) \rfloor$). Esto implica que $r \mid n^{\lfloor \log(B) \rfloor}$, lo cual es imposible por la elección del r.

Teniendo esto, sabemos que $\frac{r}{(r,n)}$ tampoco puede dividir a (3.3), pues hay algún factor primo de r que no divide a (r, n), y por lo tanto tampoco a n. Pero r era el menor elemento que no dividía a dicho producto, y como $\frac{r}{(r,n)} \leq r$, no queda más remedio que $\frac{r}{(r,n)} = r$, luego $(r, n) = 1$.

Finalmente, como r no divide a ningún $n^i - 1$ para $1 \leq i \leq \lfloor \log^2(n) \rfloor$, se tiene que $ord_r(n) > \log^2(n)$, como queríamos. □

Este lema que acabamos de demostrar nos asegura que podemos elegir r de manera que su tamaño sea polinómico en el logaritmo. Esto será esencial cuando probemos que el algoritmo es polinómico.

Una vez encontrado r, puesto que $ord_r(n) > \log^2(n) \geq 1$, sabemos que debe existir un p factor primo de n de forma que $ord_r(p) > 1$ (si no existiera dicho p, entonces $n \equiv 1 \mod (r)$, lo cual sería una contradicción).

Además, tenemos que destacar dos propiedades:

- $p > r$, pues de lo contrario, el paso 3 o el paso 4 determinarían la primalidad de n.
- $(n, r) = 1$, pues de lo contrario, el paso 3 o el paso 4 determinarían la primalidad de n. Esto implica que $n, p \in \mathbb{Z}_r^\times$.

De ahora en adelante, consideramos p y r fijos. Ahora vamos a hablar de una propiedad que nos será de utilidad para la prueba, llamada *introspección*.

Antes de introducir el concepto de *introspección*, vamos a definir $\ell = \lfloor \sqrt{\phi(r)} \log(n) \rfloor$. Sabemos además que el paso 5 no devuelve COMPUESTO, por lo que se debe cumplir que, para todo $0 \leq a \leq \ell$,

$$(X+a)^n \equiv X^n + a \mod (n, X^r - 1)$$

Esto implica para todo $0 \leq a \leq \ell$, dado que p es un factor primo de n, que

$$(X+a)^n \equiv X^n + a \mod (p, X^r - 1) \tag{3.4}$$

Por Proposición 3.1, tenemos que, para todo $0 \leq a \leq \ell$

$$(X+a)^p \equiv X^p + a \mod (p, X^r - 1) \tag{3.5}$$

De estas dos últimas, deducimos que, para todo $0 \leq a \leq \ell$

$$(X+a)^{n/p} \equiv X^{n/p} + a \mod (p, X^r - 1)$$

Esto último lo comprobamos utilizando la siguiente cadena de congruencias:

$$\begin{aligned}
(X+a)^{n/p} &\equiv (X^p + a)^{n/p} \mod (p, X^r - 1) \\
&\equiv [(X+a)^p]^{n/p} \mod (p, X^r - 1) \\
&\equiv (X+a)^n \mod (p, X^r - 1) \\
&\equiv X^n + a \mod (p, X^r - 1) \\
&\equiv (X^p)^{n/p} + a \mod (p, X^r - 1) \\
&\equiv X^{n/p} + a \mod (p, X^r - 1)
\end{aligned}$$

En la segunda equivalencia usamos (3.5). En la cuarta usamos (3.4)

Lo que podemos comprobar de estas congruencias es que tanto n como $\frac{n}{p}$ se comportan como p. Damos entonces una definición a esta propiedad.

Definición 3.1. Sea $f \in \mathbb{Z}[X]$ un polinomio y sea $m \in \mathbb{N}$. Diremos que m es *introspectivo* para f si

$$[f(X)]^m \equiv f(X^m) \mod (p, X^r - 1)$$

De las congruencias anteriores, es evidente ver que tanto p como $\frac{n}{p}$ son introspectivos para $X + a$ con $0 \leq a \leq \ell$.

Vamos ahora a dar dos características que prueban que esta propiedad es cerrada para la multiplicación, tanto para los números como para los polinomios.

Lema 3.5. *Se cumplen:*

1. *Dados m, m' introspectivos para f un polinomio, entonces mm' es introspectivo para f.*
2. *Dados f, g polinomios para los que m es introspectivo, entonces m es introspectivo para fg.*

Demostración. Vamos a demostrar cada una por separado.

1. Por un lado, puesto que m es introspectivo para f, se tiene que

$$f(X)^{mm'} \equiv f(X^m)^{m'} \mod (p, X^r - 1)$$

Por otro lado, como m' es introspectivo para f, tenemos que

$$f(X^m)^{m'} \equiv f(X^{mm'}) \mod (p, X^{mr} - 1) \Rightarrow f(X^m)^{m'} \equiv f(X^{mm'}) \mod (p, X^r - 1)$$

La implicación se sigue de que $X^r - 1 | X^{mr} - 1$. Uniendo entonces ambas congruencias, obtenemos la siguiente

$$f(X)^{mm'} \equiv f(X^{mm'}) \mod (p, X^r - 1)$$

2. Se tiene lo siguiente

$$[f(X)g(X)]^m \equiv f(X)^m g(X)^m \equiv f(X^m)g(X^m) \mod (p, X^r - 1)$$

□

Con estos dos resultados, y sabiendo que p y $\frac{n}{p}$ son introspectivos para $X + a$ con $0 \leq a \leq \ell$, podemos afirmar que todos los elementos del conjunto

$$I = \left\{ p^i \left(\frac{n}{p}\right)^j | i, j \geq 0 \right\} \tag{3.6}$$

son introspectivos para todos los elementos del conjunto

$$P = \left\{ \prod_{a=0}^{\ell} (X + a)^{e_a} | e_a \geq 0 \right\}$$

Vamos ahora a definir dos grupos que serán de vital importancia en la demostración. Empecemos por el primero de ellos.

$$G = \{a \in \mathbb{Z}_r \mid b \equiv a \mod (p, X^r - 1), \ \ b \in I\} \tag{3.7}$$

Este grupo consiste básicamente en los restos de dividir los elementos de I por r. Como tenemos que $(n, r) = (p, r) = 1$, es claro entonces que G es un subgrupo del grupo multiplicativo de $\mathbb{Z}_r$, $\mathbb{Z}_r^\times$. Definamos $t = |G|$. Es claro que G está generado por n y p (pues $n = \frac{n}{p}p$, producto de dos elementos de I) y, sabiendo esto y que $ord_r(n) > \log^2(n)$, es claro que $t > \log^2(n)$.

Definamos ahora el segundo grupo. Para ello, sea $\Phi_r \in \mathbb{Z}_p[X]$ el r-ésimo polinomio ciclotómico con coeficientes en $\mathbb{Z}_p$. Entonces sabemos que Φ_r factoriza en polinomios irreducibles

de grado $ord_r(p)$. Sea entonces $h \in \mathbb{Z}_p[X]$ uno de esos factores irreducibles, cuyo grado será mayor que 1 al ser $ord_r(p) > 1$. Sea el cuerpo $\mathbb{F} = \mathbb{Z}_p[X]/(h(X))$. Definimos pues el segundo grupo.

$$\mathcal{G} = \{f \in \mathbb{F} \mid g \equiv f \mod (p, X^r - 1),\ \ g \in P\} \tag{3.8}$$

Este grupo consiste en los restos de dividir los elementos de P entre $h(X)$ y p. Es claro que $\mathcal{G}$ está genarado por $X + a$ con $0 \leq a \leq \ell$ en el cuerpo $\mathbb{F}$, y es claro entonces que $\mathcal{G}$ es un subgrupo del grupo multiplicativo de $\mathbb{F}$, $\mathbb{F}^\times$.

Nuestra tarea ahora va a ser dar cotas para el grupo $\mathcal{G}$ recién definido. Para ello, enunciamos el siguiente lema, el cual nos da una cota inferior.

Lema 3.6.

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

Demostración. Para empezar vamos a comprobar que todos los polinomios en P de grado menor que t son distintos en $\mathbb{F}$. Para ello, supongamos $g, f \in P$ (3.8) distintos de grados menor que t tales que $f(X) \equiv g(X)$ en $\mathbb{F}$. Tomemos $m \in I$. Sabemos que $f(X)^m \equiv g(X)^m$ en $\mathbb{F}$, y como m es introspectivo para f y g, entonces tenemos

$$f(X^m) \equiv g(X^m)$$

Esta equivalencia se cumple para $\mathbb{Z}_p[X]/(X^r - 1)$, por lo que naturalmente se cumple para $\mathbb{F}$ al darse que $h(X) \mid X^r - 1$. Esto implica que X^m es una raíz del polinomio $f(Y) - g(Y)$ para todo $m \in G$. Puesto que G es un subgrupo de $\mathbb{Z}_p^\times$, es obvio que $(m, r) = 1$ y, por lo tanto, X^m es una raíz r-ésima primitiva de la unidad (todas ellas distintas por ser todos los m distintos). Por lo tanto, el polinomio $f(Y) - g(Y)$ tendrá al menos t raíces distintas en $\mathbb{F}$. Esto es una contradicción, pues el grado de $f(Y) - g(Y)$ es menor que t por la elección de ambos, luego llegamos a una contradicción, luego $f \not\equiv g$ en $\mathbb{F}$.

Por otro lado sabemos que $\ell = \lfloor \sqrt{\phi(r)} \log(n) \rfloor < \sqrt{r} \log(n) < r < p$ (la penúltima desigualdad viene de que $r > \log^2(n)$). Por lo tanto, los polinomios $X + a$ con $0 \leq a \leq \ell$ son todos distintos en $\mathbb{F}$. Como además el grado de h es mayor que 1, tenemos que $X + a \not\equiv 0$ en $\mathbb{F}$ con $0 \leq a \leq \ell$. Por lo tanto tenemos que existen $\ell + 1$ polinomios de grado 1. Teniendo en cuenta que todos los polinomios en P de grado menor que t son distintos en $\mathcal{G}$, solo tenemos que calcular todas estas combinaciones usando combinatoria.

Por un lado, sabemos que la cantidad de polinomios de grado exactamente $k \in \mathbb{N}$ que podemos construir con $\ell + 1$ polinomios de grado 1 es equivalente a $\binom{\ell+k}{\ell}$. Tomando $0 \leq k \leq t - 1$, tenemos que la cantidad total de polinomios de grado menor que t que podemos construir con $\ell + 1$ polinomios de grado 1 viene dada por

$$\sum_{k=0}^{t-1} \binom{\ell + k}{\ell} = \sum_{k=\ell}^{\ell+t-1} \binom{k}{\ell} = \binom{\ell + t}{\ell + 1} = \binom{t + \ell}{t - 1}$$

Donde en la segunda desigualdad hemos usado la *Identidad del Palo de Hockey* 1.5. Por lo tanto tenemos que $|\mathcal{G}| \geq \binom{t+\ell}{t-1}$, como queríamos. □

Teniendo esta cota inferior, ahora vamos a calcular una cota superior, la cual solo es cierta cuando n no es una potencia de p.

Lema 3.7. *Si n no es una potencia de p, entonces se tiene que $|\mathcal{G}| \leq n^{\sqrt{t}}$.*

Demostración. Vamos a considerar el siguiente subconjunto de I (3.6) definido de la siguiente manera:

$$\hat{I} = \left\{ \left(\frac{n}{p}\right)^i p^j \mid 0 \leq i, j \leq \lfloor \sqrt{t} \rfloor \right\}$$

Es evidente que si n no es una potencia de p, la cantidad de elementos distintos en $\hat{I}$ es equivalente a $(\lfloor \sqrt{t} \rfloor + 1)^2 > t$. Dado que sabemos que $|G| = t$, por el principio del palomar, existen al menos dos elementos en $\hat{I}$ de manera que su resto módulo r es el mismo. Dicho de otro modo, sean $m_1, m_2 \in \hat{I}$ con $m_1 > m_2$ (esto sin perder generalidad), de manera que $X^{m_1} \equiv X^{m_2} \mod (X^r - 1)$. Sea entonces $f \in P$, y se tiene lo siguiente por ser m_1, m_2 introspectivos para f:

$$\begin{aligned} f(X)^{m_1} &\equiv f(X^{m_1}) \mod (p, X^r - 1) \\ &\equiv f(X^{m_2}) \mod (p, X^r - 1) \\ &\equiv f(X)^{m_2} \mod (p, X^r - 1) \end{aligned}$$

Así tenemos que $f(X)^{m_1} \equiv f(X)^{m_2} \mod (p, h(X))$. Por lo tanto, es evidente que $f \in \mathcal{G}$ es una raíz del polinomio $Y^{m_1} - Y^{m_2}$ en el cuerpo $\mathbb{F}$. Puesto que f es un elemento arbitrario de $\mathcal{G}$, sabemos que el polinomio $Y^{m_1} - Y^{m_2}$ debe tener al menos $|\mathcal{G}|$ raíces distintas. Siendo tal el caso, y teniendo que el grado de $Y^{m_1} - Y^{m_2}$ es m_1, nos queda

$$|\mathcal{G}| \leq m_1 \leq \left(\frac{n}{p} p\right)^{\lfloor \sqrt{t} \rfloor} = n^{\lfloor \sqrt{t} \rfloor} \leq n^{\sqrt{t}} \qquad \square$$

Con estos dos último lemas hemos conseguido acotar el tamaño de $\mathcal{G}$ cuando n no es una potencia de p. Teniendo esto en cuenta, enunciamos ya el resultado final.

Lema 3.8. *Si el algoritmo 2 devuelve PRIMO, entonces n es primo.*

Demostración. Supongamos que el algoritmo devuelve PRIMO. Sea pues $t = |G|$ y $\ell = \lfloor \sqrt{\phi(r)} \log(n) \rfloor$. Entonces tenemos la siguiente cadena de desigualdades:

$$\begin{aligned} |\mathcal{G}| &\geq \binom{t+\ell}{t-1} \\ &\geq \binom{\ell + 1 + \lfloor \sqrt{t} \log(n) \rfloor}{\lfloor \sqrt{t} \log(n) \rfloor} \\ &\geq \binom{2\lfloor \sqrt{t} \log(n) \rfloor + 1}{\lfloor \sqrt{t} \log(n) \rfloor} \\ &> 2^{\lfloor \sqrt{t} \log(n) \rfloor + 1} \geq 2^{\sqrt{t} \log(n)} \geq n^{\sqrt{t}} \end{aligned}$$

La primera igualdad se tiene por Lema 3.6. La segunda porque $t > \log^2(n) \Leftrightarrow \sqrt{t} > \log(n) \Leftrightarrow t > \sqrt{t} \log(n)$. La tercera porque $\ell = \lfloor \sqrt{\phi(r)} \log(n) \rfloor \geq \lfloor \sqrt{t} \log(n) \rfloor$. La cuarta se

tiene por Lema 1.1.

Por Lema 3.7 y dado que $|\mathcal{G}| > n^{\sqrt{t}}$, tiene que darse que n sea una potencia de p. Es decir, $n = p^k$ con $k > 0$. Puesto que en el primer paso eliminamos todas las potencias donde $k > 1$, no queda más remedio que ser $n = p$, luego n es primo. □

Finalmente, Teorema 3.1 se deduce por Lema 3.1 y por Lema 3.8, lo cual concluye la prueba de la validez del algoritmo 2.

4. Conclusiones y Vías Futuras

Una vez demostrada la validez del algoritmo, vamos a dar algunas conclusiones sobre la demostración, así como algunas mejoras que podrían mejorar aún más el algoritmo.

4.1. Conclusiones

Al final hemos conseguido uno de los objetivos principales del trabajo: demostrar que el algoritmo propuesto determina que un número es primo si, y solo si, dicho número es primo. El análisis de su complejidad lo realizaremos en la siguiente parte del trabajo.

Una de las cosas más destacables de la demostración de la validez del algoritmo **AKS** es el uso de herramientas elementales de matemáticas.

En las primeras versiones del trabajo sobre el algoritmo **AKS** [KS04], los autores hacían uso de herramientas más avanzadas de *Teoría de Números* para poder realizar las acotaciones de (3.8).

Gracias a la introducción de (3.7) y usando propiedades de las raíces de los polinomios ciclotómicos en un cuerpo, se puede evitar el uso de herramientas no elementales para la demostración del algoritmo.

Esto resulta en una demostración elegante y al alcance de la mayoría de los matemáticos.

4.2. Mejoras del Algoritmo AKS

Una vez probada la validez de nuestro algoritmo, es en la siguiente parte donde comprobaremos que dicho algoritmo tiene complejidad polinómica. En específico, veremos que usando buenos algoritmos de multiplicación y división de números enteros y polinomios, la complejidad es $O^{\sim}(\log 21/2(n))$.

Esta eficiencia se deduce de que $r = O(\log^5(n))$ por Lema 3.4 y que la cantidad de iteraciones a realizar en el paso 5 es $\lfloor \sqrt{\phi(r)} \log(n) \rfloor$.

Mejorando la cota para r o reduciendo la cantidad de iteraciones del paso 5, podremos asegurar una mejor complejidad. Vamos a discutir ambos casos.

4.2.1. Cota de r

Como ya demostramos en Lema 3.4, podemos encontrar $r = O(\log^5(n))$. En el mejor de los casos, y dado que $ord_r(n) > \log^2(n)$, podremos encontrar un $r = O(\log^2(n))$, lo cual

reduce significativamente la complejidad del algoritmo. Existen dos conjeturas de las cuales se puede deducir dicha afirmación.

Conjetura 4.1. *(**Conjetura de Artin**). Sea $n \in \mathbb{N}$ de manera que no es una potencia perfecta. Entonces, la cantidad de primos q tales $q \leq m$ para algún $m > 0$ y tales que $ord_q(n) = q - 1$ es asintóticamente $A(n)\frac{m}{\ln(n)}$, donde $A(n) > 0.35$ es la constante de Artin.*

Esta conjetura, en caso de que se cumpliera para un cierto $m = O(\log^2(n))$, implicaría la existencia de un $r = O(\log^2(n))$ cumpliendo las condiciones necesarias. Esta conjetura también es cierta en caso de que la *Hipótesis Generalizada de Riemann* 1.2 sea cierta.

Para asegurar dicho m, enunciamos esta segunda conjetura.

Conjetura 4.2. *(**Conjetura de la Densidad de Sophie-Germain**). La cantidad de primos q con $q \leq m$ para algún $m > 0$ tales que $2q + 1$ también es primo es asintóticamente $\frac{2C_2 m}{\ln^2(m)}$. $C_2 \simeq 0.66$ es la constante de los números primos gemelos.*

Los números primos con esta propiedad se les suele conocer como Primos de Sophie-Germain.

Si esta segunda conjetura fuera cierta, entonces podemos asegurar que existe un $m = O(\log^2(n))$ que cumple la conjetura de Artin, luego concluiríamos que existe un $r = O^{\sim}(\log^2(n))$ tal que $ord_r(n) > \log^2(n)$, como queríamos.

A pesar de que se sigue realizando trabajo en demostrar estas dos conjeturas, una versión más débil fue demostrada por el matemático *Fouvry* en [Fou85].

Lema 4.1. *Existen c, n_0 con $c > 0$ tales que, para todo $x \geq n_0$, se tiene*

$$|\left\{q \mid p \text{ es primo}, q \leq x \text{ y } P(q-1) > q^{2/3}\right\}| \geq c\frac{x}{\ln(x)}$$

Usando este lema, se puede mejorar la cota de r.

Teorema 4.1. *Existe $r = O(\log^3(n))$ tal que $ord_r(n) > \log^2(n)$.*

4.2.2. Iteraciones del Paso 5

Otra manera de optimizar aún más el algoritmo es reduciendo la cantidad de iteraciones que hay que realizar en el paso 5, donde la implementación actual realiza $\lfloor \sqrt{\phi(r)} \log(n) \rfloor$ iteraciones.

La razón de estas iteraciones se basa en que necesitamos asegurar que el tamaño del grupo $\mathcal{G}$ sea suficientemente grande. Si podemos encontrar alguna manera de demostrar que un $\mathcal{G}$ puede estar generado por menos polinomios del estilo $X + a$, la cantidad de iteraciones también se reduciría, lo cual mejoraría la eficiencia del algoritmo.

4.2.3. Otras mejoras

Se puede mejorar aún más la complejidad del algoritmo hasta $O^{\sim}(\log^3(n))$ si la siguiente conjetura es cierta.

Conjetura 4.3. *Sea r un número primo que no divide a n tal que se cumple*

$$(X-1)^n \equiv X^n - 1 \mod (n, X^r - 1)$$

Entonce n es primo o $n^2 \equiv 1 \mod (r)$.

Esta congruencia se puede determinar en tiempo $O^{\sim}(r\log^2(n))$, y como dicho r podemos encontrarlo en el intervalo $[2, 4\log(n)]$, concluimos con el análisis.

Existen argumentos heurísticos que afirman que dicha conjetura probablemente no sea cierta. Sin embargo, algunas modificaciones de la conjetura, como exigir que $r > \log(n)$, pueden seguir siendo ciertas.

Parte II.

Análisis Teórico y Empírico del Algoritmo AKS

En esta parte vamos a comprobar que el algoritmo **AKS** tiene una complejidad algorítmica polinómica en el número de dígitos.

Mostraremos también una implementación de dicho algoritmo, con las decisiones tomadas en cada paso del mismo y qué herramientas se han utilizado para ello.

Finalmente, haremos una comparación del tiempo de ejecución del algoritmo **AKS** con otros algoritmos conocidos en cuanto a tests de primalidad.

5. Complejidad Algorítmica del test AKS

En este capítulo nos encargaremos de analizar la complejidad algorítmica de cada uno de los pasos del test AKS, la cual queremos ver que es polinómica en la cantidad de dígitos de la entrada o, lo que es lo mismo, que es logarítmica.

Al final de esta sección, tomaremos el paso cuya complejidad sea mayor, y comprobaremos que, efectivamente, el algoritmo AKS está dentro de la clase de complejidad polinomial para algoritmos deterministas.

5.1. Operaciones básicas

Para hablar de complejidad de algoritmos complejos, es necesario conocer la complejidad de las operaciones más básicas en matemáticas. Las que vamos a usar en este análisis son sobre todo la multiplicación y división de números enteros y la multiplicación de polinomios.

Sean entonces pues las siguientes funciones:

- $M(n)$ la cantidad de pasos a realizar para multiplicar/dividir dos números enteros de tamaño n bits.
- $P(n, m)$ la cantidad de pasos a realizar para multiplicar dos polinomios de grado n con coeficientes de tamaño m bits.

Es bien sabido ya que $O(M(n)) = O(n^2)$ y que $O(P(n, m)) = O(n^2 m^2)$ usando los métodos elementales. Para demostrar que el test **AKS** tiene complejidad polinómica, estas complejidades serán suficientes para la prueba.

No obstante, la complejidad se puede reducir aún más usando algoritmos más sofisticados. Sabemos que $O(M(n)) = O^{\sim}(n)$ y que $O(P(n, m)) = O^{\sim}(nm)$ usando algoritmos basados en la *Transformada Rápida de Fourier* [GG09]. Estas últimas complejidades serán las que consideraremos al calcular eficiencias, aunque las obtenidas por métodos elementales funcionan perfectamente para probar que el algoritmo AKS es polinomial.

5.2. Pasos del algoritmo AKS

En esta sección nos vamos a dedicar a estudiar cada paso del algoritmo y calcular las complejidades de cada uno.

Es importante destacar que no es necesario calcular las mejores complejidades posibles. En algunos pasos podemos permitirnos perder un poco de eficiencia (siempre manteniendo la polinomialidad de la misma) sin que esto afecte al hecho de que el algoritmo se ejecuta en tiempo polinómico.

5.2.1. Paso 1: Potencias Perfectas

El primer paso del algoritmo consiste en comprobar si la entrada es una potencia perfecta, en cuyo caso es evidente que el número es compuesto.

Vamos a describir un algoritmo básico, que aunque no es el más óptimo, será suficiente, pues la verdadera complejidad algorítmica se encuentra en el paso 5, donde comprobamos las identidades polinómicas.

Sea entonces pues el siguiente algoritmo:

Algorithm 3 Potencia Perfecta

```
1: procedure IsPerfectPower(n)                ▷ Comprobar si n = x^y con x, y > 1
2:     for cada k ≤ log(n) do
3:         x = ⌊n^{1/k}⌋
4:         if n = x^k then
5:             return True
6:         end if
7:     end for
8:     return False
9: end procedure
```

Hay variantes que en vez de tomar todos los números $k \leq \log(n)$ en el algoritmo 3, lo que hacen es tomar todos los $p \leq \log(n)$ primos, de manera que usando *Teoría de Cribas* se puede conseguir una complejidad de $O^{\sim}(\log^3(n))$ [BS89]. Nosotros no vamos a hacer dicha selección, de modo que nuestro algoritmo tendrá una complejidad de $O(\log^4(n))$, la cual sigue siendo polinómica en el número de cifras, pero es mucho más sencilla de probar.

Teorema 5.1. *El Algoritmo 3 tiene complejidad* $O(\log^4(n))$.

Demostración. Para calcular $\lfloor n^{1/k} \rfloor$ usando una búsqueda binaria, tenemos que elevar las sucesivas aproximaciones a k.

La operación de elevar tiene un coste de $O(\log^2(n))$ usando el algoritmo de cuadrados repetidos, y como dicha operación la realizamos $O(\log(n))$ veces (debido a la naturaleza de la búsqueda binaria), podemos concluir que cada iteración del algoritmo ocupa $O(\log^3(n))$.

Como tenemos que realizar $O(\log(n))$ iteraciones en total, podemos concluir entonces que la complejidad del Algoritmo 3 es $O(\log^4(n))$. □

Como ya dijimos anteriormente, no vamos a intentar buscar un algoritmo mucho más eficiente, pues el tiempo de ejecución de este paso es básicamente nulo comparado con el tiempo de ejecución del quinto paso, que es en el que nos vamos a centrar más a fondo.

5.2.2. Paso 2: Encontrar el menor r tal que $ord_r(n) > \log^2(n)$

Para este paso no necesitamos realmente calcular el orden exacto para cada r. Nos bastaría simplemente con probar que $n^k \neq 1 \mod (r)$ para todo $k \leq \log^2(n)$, pues si se cumplen

todas esas igualdades, podemos asegurar que $ord_r(n) > \log^2(n)$ para ese r en específico.

Lo anterior junto con el hecho de que $r \leq \max\{3, \lceil \log^5(n) \rceil\}$,
Teniendo esto en mente, podemos enunciar el teorema.

Teorema 5.2. *El paso 2 del algoritmo AKS tiene complejidad* $O^{\sim}(\log^7(n))$.

Demostración. Por Lema 3.4, tenemos que $r \leq max\{3, \lceil \log^5(n) \rceil\}$, es decir, solo hay que comprobar $O(\log^5(n))$ valores de r como mucho.

Por otro lado, fijado ya r, comprobar $ord_r(n) > \log^2(n)$ es equivalente a comprobar que se cumple $n^k \neq 1 \mod (r)$ para todo $k \leq \log^2(n)$. Esto equivale a hacer $O(\log^2(n))$ comprobaciones para cada r, ya que para cada igualdad solo realizamos una multiplicación, que al ser módulo r, nos queda que la comprobación tenga complejidad $O(\log^2(n)\log(r)) = O^{\sim}(\log^2(n))$.

Por lo tanto, tenemos que hay que comprobar $O(\log^5(n))$ valores de r, y comprobar para cada r cuesta $O^{\sim}(\log^2(n))$, luego la complejidad del paso 2 es $O^{\sim}(\log^7(n))$. □

Este paso es importante en el sentido de que la complejidad total del algoritmo depende de la cota que podamos obtener para r. A menor r, menor complejidad algorítmica.

5.2.3. Paso 3: Comprobar si $1 < (a, n) < n$ para algún $a \leq r$

Primero tenemos que calcular una eficiencia para el algoritmo de Euclides. Para ello vamos primero a calcular la cantidad de pasos que toma el algoritmos de Euclides.

Primero vamos a hacer un par de comprobaciones. Supongamos que queremos calcular (a, b) con $a > b$. Sabemos que $(a, b) = (b, c)$ con $a = k_1 b + c$ para algún $k_1 \in \mathbb{N}$ o, dicho en términos más claros, c es el resto de dividir a entre b ($c = a\,\%b$). Por otro lado, $(b, c) = (c, d)$ con $b = k_2 c + d$ para algún $k_2 \in \mathbb{N}$.

Puesto que $b > c$, es claro entonces que $k_2 \geq 1$, luego $b = k_2 c + d \geq c + d$, y en consecuencia, $a > c + d$. Sumamos estas dos últimas expresiones y obtenemos $a + b > 2(c + d)$.

Estas dos observaciones nos dan a entender que, cada dos pasos, el tamaño del problema se reduce a algo menos de la mitad. Con eso en las manos y asumiendo que a, b tienen como mucho k bits, podemos asegurar que el algoritmo da $O(k)$ pasos.

Sabiendo esto y que en cada paso debemos realizar una división, llegamos a que la eficiencia del algoritmo para dos números de tamaño k bits es $O(kM(k)) = O^{\sim}(k^2)$, es decir, $O^{\sim}(\log^2(n))$ para el número completo.

El algoritmo de Euclides debemos aplicarlo r veces, y sabemos que $r = O(\log^5(n))$, luego el tiempo de ejecución de este paso es $O^{\sim}(\log^7(n))$.

5.2.4. Paso 4: Comprobar si $n \leq r$

Este paso es probablemente el más sencillo de todos, pues solo tenemos que hacer una comparación.

Como las comparaciones solo requieren comparar los bits, podemos asegurar que este paso tiene complejidad $O(\log(n))$.

5.2.5. Paso 5: Comprobar identidades polinómicas

En este paso tenemos un bucle. La complejidad entonces será el tamaño del bucle multiplicado por la complejidad de cada iteración.

Primero tenemos que el número de iteraciones es $\lfloor \sqrt{\phi(r)} \log(n) \rfloor$. Como sabemos que $\phi(r) = r - 1$ si r es primo y que $r = O(\log^5(n))$, podemos entonces asegurar lo siguiente:

$$O\left(\lfloor \sqrt{\phi(r)} \log(n) \rfloor \right) = O(\sqrt{r} \log(n)) = O(\log^{5/2}(n) \log(n)) = O(\log^{7/2}(n))$$

Teniendo esto claro, ahora tenemos que comprobar lo que nos cuesta cada iteración. Para ello nos basta primero con saber que la exponenciación requiere de $O(\log(n))$ multiplicaciones de polinomios. Aunque tengamos que realizar dos exponenciaciones ($(X^n + a)^n$ y X^n), esto no cambia el hecho de que la cantidad de multiplicaciones de polinomios siga siendo $O(\log(n))$.

Ahora veamos lo que nos cuesta cada multiplicación de polinomios. Dado que esta exponenciación se hace módulo $(X^r - 1, n)$, sabemos que el grado de los polinomios va a ser $O(r)$ y el tamaño de los coeficientes $O(\log(n))$. Como ya vimos anteriormente, la multiplicación de polinomios con grado r y coeficientes de tamaño $O(\log(n))$ bits tiene complejidad $O(P(r, \log(n)) = O^{\sim}(r \log(n)) = O^{\sim}(\log^6(n))$.

Con todas estas piezas, tenemos que la complejidad del paso 5 es $O^{\sim}(\log^{7/2}(n) \log(n) \log^6(n)) = O^{\sim}(\log^{21/2}(n))$.

5.3. Resultado final

Habiendo comprobado las eficiencias de todos los pasos, y sabiendo que el paso 6 no afecta en absoluto, tenemos que las eficiencias de cada paso son:

1. $O(\log^4(n))$
2. $O^{\sim}(\log^7(n))$
3. $O^{\sim}(\log^7(n))$
4. $O(\log(n))$
5. $O^{\sim}(\log^{21/2}(n))$

Puesto que la complejidad del quinto paso es superior a la de los 4 anteriores, podemos concluir con que la complejidad del algoritmo **AKS** es $O^{\sim}(\log^{21/2}(n))$.

Cabe destacar que, si hubiésemos considerado que $O(M(n)) = O(n^2)$ y que $O(P(n,m)) = O(n^2m^2)$, la complejidad del quinto paso sería $O(\log^{33/2}(n))$, la cual es mucho peor que la obtenida, pero sigue siendo polinómica.

5.4. Cotas del algoritmo

En el análisis recién realizado, cabe destacar que la eficiencia del quinto paso (y la del algoritmo en general) depende del valor r encontrado en el segundo paso. Dicho valor sabemos que está acotado superiormente por $\max\{\lceil 3, \log^5(n)\rceil\}$, y es lo que nos ha proporcionado la complejidad $O^{\sim}(\log^{21/2}(n))$.

Como sabemos que dicho valor tiene que ser mayor que $\log^2(n)+1$ por lo explicado anteriormente, podemos entonces asegurar que $r \in [\log^2(n)+2, \lceil \log^5(n)\rceil]$. De esta manera podemos acotar la complejidad exacta del algoritmo, teniendo así que $O^{\sim}(\log^{21/2}(n))$ y $\Omega^{\sim}(\log^6(n))$, y concluyendo que $\Theta(\log^x(n))$ con $x \in [6, 21/2]$.

6. Implementación del test AKS

En este capítulo nos vamos a dedicar a implementar el algoritmo AKS usando un lenguaje de programación.

Por un lado mostraremos las herramientas que nos ayudarán a ello, y por otro iremos paso a paso explicando cada detalle y decisión a la hora de implementarlo.

6.1. Herramientas de desarrollo

En primer lugar vamos a mostrar las herramientas elegidas tanto para poder implementar el algoritmo AKS, como para poder crear el programa y poder ejecutarlo de manera sencilla.

6.1.1. Lenguaje de programación: C++

El lenguaje elegido para la implementación es C++ y, en específico, la revisión del año 2020: C++20.

C++ es un lenguaje de programación multiparadigma diseñado por el profesor Bjarne Stroustrup [bja20] basado en el ya conocido C. De hecho, C++ fue pensado en un principio para ser 100 % compatible con C, aunque la divergencia en los últimos años es cada vez más notable.

Las principales razones por la que C++ es el lenguaje elegido son su velocidad y su capacidad de poder abstraer conceptos fácilmente. La versión utilizada es la del año 2020 por ser la última y por estar ampliamente soportada entre los principales compiladores: GCC, Clang y MSVC.

El compilador a usar no está definido, pues al ser una librería, debería haber libertad para poder usar el compilador que uno prefiera. En este proyecto, todos los tests y mediciones que se hagan serán usando GCC o Clang, pero se podría usar cualquier otro que al menos soporte C++20 y tenga buena integración con el build system (el cual ahora veremos).

6.1.2. Build system: CMake

No nos basta solo con elegir un lenguaje junto con un compilador. Según crece el proyecto, también necesitaremos una herramienta para poder compilarlo todo automáticamente sin tener que hacerlo a mano. Es por eso que usaremos un build system para ello.

El build system usado para compilar este proyecto es CMake [Kit].

CMake es lo que se conoce como un meta build system. Su diferencia principal con otros sistemas como *Makefile* o *Ninja* es que CMake se encarga de generar estos últimos. Podemos decir que CMake es realmente un generador de build systems.

Su principal ventaja con respecto a los build systems tradicionales es que CMake está pensado para ser multiplataforma, por lo que un mismo CMake puede servir para compilar tanto en Linux como en Windows o Apple. Esto se debe a que puede generar archivos de los build systems nativos de estos sistemas operativos.

Es muy versátil y es el sistema más usado para proyectos desarrollados en C o C++.

6.1.3. Manejo de dependencias: Conan

A medida que se desarrolla un proyecto, suele ser necesario usar funcionalidad que otras personas ya han hecho con anterioridad y que nos alivia a nosotros de ese trabajo.

Es lo que normalmente conocemos como librerías, y en el caso de C++, hay varias estrategias y herramientas entre las que elegir.

Para este proyecto se ha decidido utilizar el manejador de dependencias Conan [JFr].

Las razones para elegir Conan es que tiene una integración sencilla con CMake, contiene todas las librerías que se van a usar y está bastante estandarizado en el ecosistema de C++.

Otras opciones podrían haber sido Vcpkg de Microsoft o simplemente instalar las dependencias a mano (esto último puede llegar a ser muy tedioso para el usuario final).

6.1.4. Librerías

Para el desarrollo será necesario el uso de varias librerías.

Para la implementación del test AKS en específico haremos uso de las siguiente:

- **GMP**: Implementada en C. Es la librería por defecto para usar cuando queremos trabajar con números de precisión arbitraria.

  Usaremos esta librería tanto para poder testear números muy grandes (que normalmente no caben en los registros de 64 bits), como para utilizar algunas funciones que nos serán muy útiles en la implementación del algoritmo.

  Además, esta librería contiene un wrapper para C++, lo cual nos será muy útil a la hora de definir la interfaz.

- **MPFR**: Implementada en C. Esta librería es utilizada para trabajar con números en coma flotante de precisión arbitraria.

Hay algunos puntos en la implementación del test AKS donde necesitaremos funciones que nos permitan controlar bien la precisión para calcular cotas lo más fieles posibles.

Esta librería tiene buena integración con **GMP**, por lo que es una candidata perfecta para este proyecto.

- **NTL**: Implementada en C++. Más conocida como *Number Theory Library*, será nuestra opción a la hora de implementar las identidades polinómicas.

  Tiene una alta cantidad de módulos entre los que elegir, y la interfaz es mucho más clara al estar escrita en C++.

  La única pega de esta librería es que no está incluida en el manejador de paquetes, por lo que el usuario deberá instalarla en el sistema para poder obtener todos sus beneficios.

Además de las librerías para implementar el test AKS, también necesitaremos algunas otras para testing, benchmarks, etc:

- **Catch2**: Implementada en C++. Esta librería es una de las más utilizadas en el entorno de C++ para testear código. Nos será muy útil para implementar los tests de todos los pasos del algoritmo.
- **Google Benchmark**: Implementada en C++. Esta librería la usaremos sobre todo para evitar que el compilador optimice los resultados de algunas llamadas, y de esa manera conseguir que los tiempos de ejecución sean lo más fieles posibles a los reales.

6.1.5. Analizadores estáticos: Cppcheck y Clang-tidy

A pesar de que el compilador suele atrapar muchos errores y avisarnos de posibles bugs, estos no son infalibles y pueden pasar por alto muchos bugs que, en el sentido técnico de la palabra, son perfectamente válidos en la gramática del lenguaje.

Es por ello que usaremos dos analizadores estáticos durante el desarrollo para atrapar la mayor cantidad de errores posibles y así agilizar el desarrollo: Cppcheck [Mar] y Clang-tidy [Fou].

6.1.6. Generador de Gráficas: gnuplot

Puesto que en este trabajo queremos analizar los tiempos de ejecución de distintos algoritmos, la mejor manera de representarlos es haciendo uso de gráficas. Dichas gráficas nos ayudarán a analizar de manera visual los tiempos de ejecución de manera más fácil.

La herramienta que usaremos para generar dichas gráficas será **gnuplot**. Esta suele ya venir incluida en la mayoría de distribuciones de Linux.

6.1.7. IDE: Visual Studio Code

Para poder manejar todas estas herramientas de forma más cómoda, suele ser recomendable utilizar un entorno de desarrollo especializado o un editor de texto.

El abanico de entornos de desarrollo es muy basto, y cada desarrollador suele adaptarse mejor a unos que otros.

En este caso, el entorno de desarrollo a utilizar será Visual Studio Code [Mic16].

Este IDE (Integrated Development Environment) tiene una gran integración con C++, y hace que la navegación por el código y la detección de errores sea mucho más amena que con un editor común.

6.2. Implementación

Una vez presentadas todas las herramientas, vamos a describir cómo se ha implementado cada paso del algoritmo.

A pesar de que describiremos todas las partes por separado, la parte en la que más tiempo vamos a dedicar será el paso 5, pues es ahí donde se encuentra la mayor complejidad y donde más esfuerzo se va a tener que invertir para conseguir un buen resultado.

6.2.1. Estructura

Antes de explicar la estructura física del proyecto, vamos a mostrar un diagrama donde podremos ver cada uno de los componentes del proyecto y cómo se relacionan entre ellos.

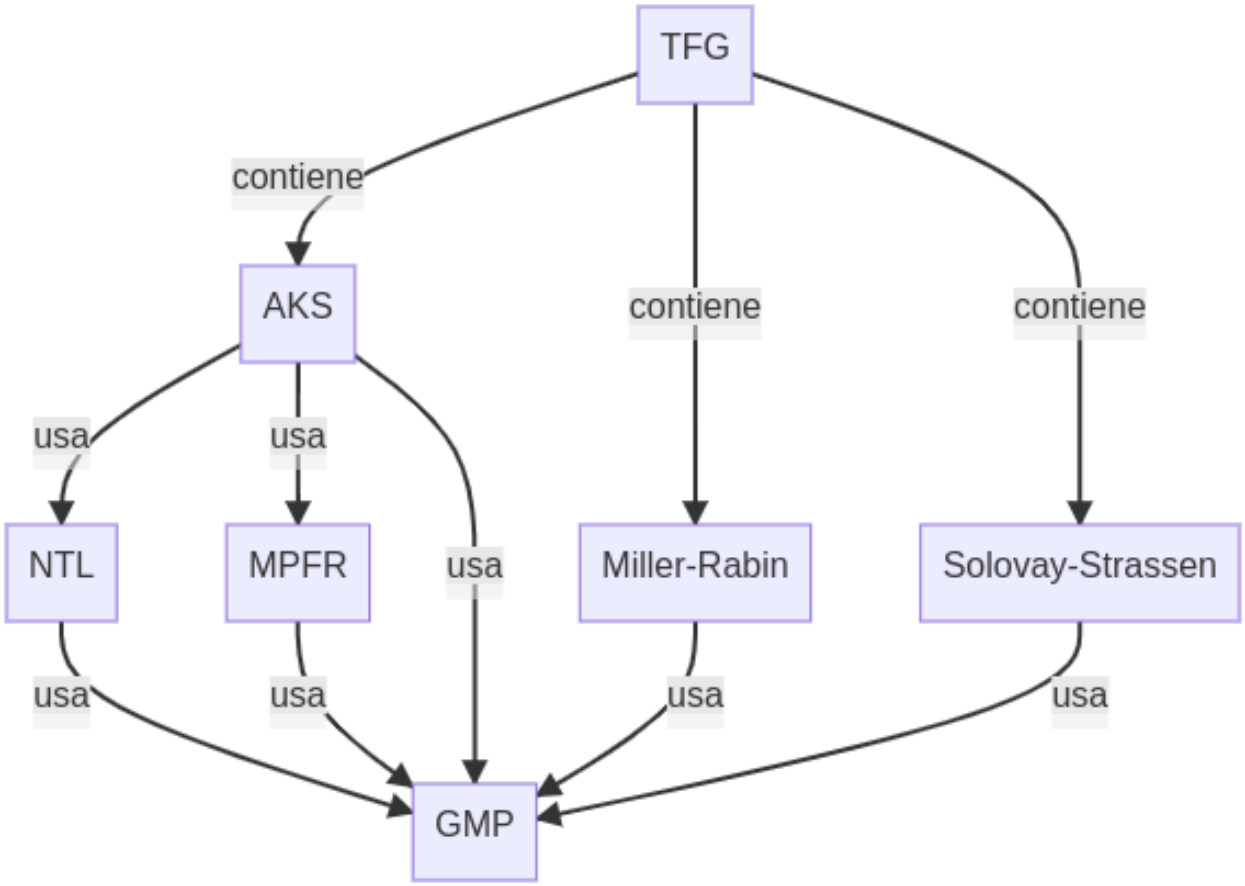

Figura 6.1.: Diagrama de relaciones de los componentes del proyecto

Todos los tests que se implementan en el proyecto hacen uso de la librería **GMP** para manejar números de precisión arbitraria. Además el algoritmo **AKS** hace uso de las librerías **NTL** (multiplicación de polinomios) y **MPFR** (cálculo preciso de cotas). Estas dos últimas además hacen uso de **GMP**. Finalmente tenemos el paquete **TFG** que incluye todos los tests de primalidad implementados. Ahora vamos a explicar cómo se ha estructurado físicamente el proyecto.

El código fuente está incluido todo en una carpeta a la que hemos llamado **TFG**. Dentro de esta carpeta tenemos varios archivos:

- **.clang-tidy**: Control sobre los warnings que emite Clang-tidy.
- **suppressions.txt**: Control sobre los warnings que emite Cppcheck.
- **CMakeLists.txt**: Fichero con todas las órdenes necesarias para compilar el proyecto usando CMake.
- **conanfile.py**: Archivo Python donde se añaden las dependencias de Conan.

- **graphs.gp**: Archivo de **gnuplot** que usaremos para generar las gráficas comparativas de los tiempos de ejecución de los tests de primalidad.

Luego tenemos varias carpetas:

- **include**: Cabeceras públicas de las funciones (API).
- **src**: Implementación de las funciones y cabeceras privadas.
- **tests**: Archivos con los tests unitarios.
- **examples**: Ejemplos para ejecutar los algoritmos implementados.
- **cmake**: Archivos auxiliares que usa el build system.

Todas las funciones de la librería se encuentran en un único namespace, llamado **tfg**, y en el archivo **include/TFG.hpp**.

Las funciones relativas al test **AKS** se encuentran en el namespace **tfg::aks**, y los pasos en el namespace **tfg::aks::steps**. La API se encuentra en el archivo **include/AKS.hpp**, y la implementación en **src/AKS.hpp**.

Las funciones que actúan como wrapper de las funciones de **GMP** están en el archivo **src/GMPWrappers.hpp** (no se exponen como parte de la librería) y las implementaciones en **src/GMPWrappers.cpp**. Todas estas funciones se encuentran en el namespace **tfg::gmp**.

6.2.2. Comprobar potencia perfecta

En este apartado vamos a presentar varias decisiones que se han tomado para implementar este algoritmo.

A pesar de que probamos que este algoritmo tiene complejidad $O^{\sim}(\log^4(n))$ y se presentará una posible implementación, la implementación final hará uso de la función ya implementada por la librería **GMP**.

Dicho esto, pasaremos a comprobar las distintas implementaciones.

6.2.2.1. Implementación $O(\log^4(n))$

En esta sección vamos a exponer de manera resumida el código en C++ necesario para implementar esta versión. Tampoco vamos a explicarlo mucho en profundidad, ya que en las siguientes secciones expondremos alternativas más eficientes y rápidas.

Primero presentamos la función *isPowerOf*. Esta función toma dos valores (n, p) y comprueba si existe algún a tal que $n = a^p$.

```
auto isPowerOf(mpz_class n, size_t p) -> bool {
        auto lower = 0_mpz;
        auto upper = n;

        while (lower <= upper) {
```

```
                auto middle = mpz_class{(lower + upper) / 2};
                auto value = pow(middle, p);

                if (value < n)
                        lower = middle + 1;
                else if (value > n)
                        upper = middle - 1;
                else
                        return true;
        }

        return false;
}
```

La función puede ser optimizada un poco mejor si la primera cota de la cota superior (upper) es un poco más baja. Por ejemplo $2^{\lfloor \log(n)/p \rfloor + 1}$ sería válida, pero de momento la dejamos más simple.

El algoritmo es muy simple. Simplemente calculamos la mitad de ambas cotas, y ese número lo elevamos a p. Si el resultado es menor que n, actualizamos la cota inferior (lower); si es mayor, actualizamos la superior; y si es igual, devolvemos true. Si el bucle acaba, la búsqueda binaria no ha encontrado el valor, luego devolvemos false.

Ahora presentamos el algoritmo *isPerfectPower*. Esta función toma un valor n y comprueba si existen dos valores $a, p > 1$ tales que $n = a^p$.

```
auto isPerfectPower(mpz_class n) -> bool {
        auto top = floorLog2(n);

        for (auto p = size_t{2}; p <= top; ++p)
                if (isPowerOf(n, p))
                        return true;

        return false;
}
```

La función **floorLog2** se implementa como el número de bits que ocupa n menos 1 (*mpz_sizeinbase(n.get_mpz_t(), 2) - 1*). Dicha cota sabemos que es la mayor porque $\lfloor \log(b) \rfloor$ es el mayor exponente posible suponiendo que n pueda ser una potencia de 2. Luego simplemente se comprueba para cada valor de p si $n = a^p$ para algún $a > 1$.

Esta implementación es relativamente simple. Ahora pasaremos a explicar optimizaciones que podemos hacer a esta implementación para reducir la complejidad a $O^{\sim}(\log^3(n))$, tal y como se explica en [BS89], Theorem 3.1.

6.2.2.2. Implementación $O^{\sim}(\log^3(n))$

En esta implementación simplemente realizaremos algunas pequeñas modificaciones a la anterior para poder reducir la complejidad.

En primer lugar, la primera optimización que haremos será reducir la primera cota superior en la función *isPowerOf*. Por tanto, el único cambio es el siguiente:

```
auto upper = pow(2_mpz, floorLog2(n) / p + 1);
```

De este modo, la complejidad de *isPowerOf* pasa a ser $O(\log^3(n)/p)$, en contraste con la implementación del apartado anterior que era $O(\log^3(n))$. Ahora tenemos que optimizar la función *isPerfectPower*.

Para ello, en vez de calcular una cota superior, simplemente vamos a usar el algoritmo de la Criba de Eratóstenes para calcular todos los primos menores que $\log(n)$. Suponiendo que tenemos dicho algoritmo implementado, la nueva versión quedaría tal que así:

```
auto isPerfectPower(mpz_class n) -> bool {
        auto primes = eratosthenesSieve(floorLog2(n));

        for (auto prime : primes)
                if (isPowerOf(n, p))
                        return true;

        return false;
}
```

La función *eratostheneSieve* básicamente calcula todos los $p \in \mathbb{N}$ primos tales que $p \leq \lfloor \log(n) \rfloor$.

Luego simplemente iteramos cada primo y hacemos la comprobación con la búsqueda binaria.

Este algoritmo tiene complejidad $O^{\sim}(\log^3(n))$ [BS89].

6.2.2.3. Implementación GMP

La versión que finalmente se ha implementado es la que ya proporciona la librería **GMP**. Dicha función se llama *mpz_perfect_power_p*. Recibe un único argumento n de tipo *mpz_t* y devuelve un entero distinto de 0 si n es una potencia perfecta.

La función se ha encapsulado en su correspondiente wrapper llamado **isPerfectPower** en el namespace **tfg::gmp**. La implementación es la siguiente:

```
auto isPerfectPower(mpz_class const& n) -> bool {
        return mpz_perfect_power_p(n.get_mpz_t()) != 0;
}
```

Este algoritmo utiliza el método de Newton para calcular raíces. Una explicación un poco más detallada se puede encontrar en Apéndice B.

6.2.3. Encontrar menor r tal que $ord_r(n) > \log^2(n)$

En este paso vamos a calcular el valor de r que luego usaremos en el paso 5. Explicaremos tanto la manera en que calculamos la cota como el cálculo de $ord_r(n)$ de manera eficiente.

6.2.3.1. Calcular $\log^2(n)$

Para calcular esta cota de manera fiable y lo más baja posible, usaremos la librería **MPFR**, ya que con **GMP** no podemos asegurar una cota tan precisa. Para calcular la cota, este es el código:

```
auto log2Sqr(mpz_class n) -> size_t {
        mpfr_t thresholdMPFR;
        mpfr_init_set_z(thresholdMPFR, n.get_mpz_t(), MPFR_RNDU);
        mpfr_log2(thresholdMPFR, thresholdMPFR, MPFR_RNDU);
        mpfr_sqr(thresholdMPFR, thresholdMPFR, MPFR_RNDU);

        return mpfr_get_ui(thresholdMPFR, MPFR_RNDD);
}
```

Primero cabe destacar que la cota no debería sobrepasar los 64 bits de capacidad (pues luego tendremos que reservar memoria acorde a esta cota), luego podemos devolver un entero sin signo cuyo valor máximo nunca será menor que la cantidad de memoria del ordenador.

Primero inicializamos una variable de tipo *mpfr_t* (número en coma flotante con precisión arbitraria) con la función *mpfr_init_set_z*, que toma nuestro entero de **GMP** y lo redondea hacia $+\infty$.

Después realizamos *mpfr_log2* y *mpfr_sqr* sucesivamente, que son las funciones logaritmo en base 2 y elevar al cuadrado respectivamente (todo esto reutilizando la misma memoria). Seguimos redondeando a $+\infty$.

Finalmente devolvemos la cota que hemos calculado en coma flotante como si fuera un entero redondeado hacia $-\infty$ (o lo que es lo mismo, $\lfloor \log_2(n)^2 \rfloor$).

6.2.3.2. Comprobar que $ord_r(n) > \log^2(n)$

Para este paso, como ya explicamos anteriormente, no necesitamos calcular explícitamente $ord_r(n)$, sino simplemente comprobar que $n^k \neq 1 \mod (r)$ para todo $k \leq \log^2(n)$. Este es el código:

```
auto isOrderBiggerThan(mpz_class n, size_t r, size_t threshold) -> bool {
        auto temp = 1_mpz;

        for (auto i = std::size_t{1}; i <= threshold; ++i) {
                temp *= n;
                temp %= r;

                if (temp == 1)
                        return false;
        }

        return true;
}
```

Las entradas son n (número cuya primalidad queremos testear), r (el valor actual que estamos comprobando) y threshold (cota previamente calculada).

Es importante remarcar que, aunque sabemos que la cota es $\log^2(n)$ y, por lo tanto, podríamos calcularla dentro del bucle, es preferible calcularla fuera una vez y pasarla simplemente a esta función cada vez.

El algoritmo simplemente comprueba si en algún momento $n^k \equiv 1 \mod (r)$ y, en dicho caso, devolver false (pues el orden entonces es menor que la cota que hemos pasado).

Si acabamos el bucle, significa que $ord_r(n) > \log^2(n)$, luego devolvemos true.

6.2.3.3. Bucle para probar valores de r

Con las dos funciones anteriores, estamos preparados para ejecutar el segundo paso. Este es el código:

```
auto step2(mpz_class n) -> size_t {
        auto const threshold = log2Sqr(n);

        for (auto r = threshold + 2;; ++r)
                if (isOrderBiggerThan(n, r, threshold))
                        return r;
}
```

Lo importante a destacar aquí es que el primer r que probamos es $\log^2(n) + 2$, pues este es el primer r para el que es posible que se cumpla que $ord_r(n) > \log^2(n)$.

La razón e que, dado que $n^{\phi(r)} \equiv 1 \mod (r)$ para todo $n, r \geq 1$, si $r \leq \log^2(n) + 1 \Rightarrow ord_r(n) \leq \phi(r) \leq r - 1 \leq \log^2(n)$, luego sería imposible que $ord_r(n) > \log^2(n)$.

Más allá de esa aclaración, el código es fácil de entender. Simplemente vamos probando varios valores de r hasta que encontremos uno que cumple la condición, el cual ya sabemos que existe por Lema 3.4.

6.2.4. Comprobar si $1 < (a, n) < n$ para algún $a \leq r$

Este paso consiste simplemente en calcular el máximo común divisor repetidamente para valores de $a \leq r$. El código para ello es el siguiente:

```
auto checkGCD(mpz_class n, size_t r) -> bool {
        for (auto a = size_t{2}; a <= r; ++a) {
                auto const result = gmp::gcd(a, n);

                if (1 < result && result < n)
                        return true;
        }
        return false;
}
```

Como dijimos anteriormente, usamos **size_t/std::size_t** para el tipo de r (pues luego lo usaremos para reservar memoria).

La función *gmp::gcd* simplemente es un wrapper de la función de **GMP**, el cual ya explicamos al principio de este capítulo.

Si encontramos un a de manera que $1 < (a, n) < n$, devolvemos true. En caso contrario, devolvemos false.

6.2.5. Comprobar si $n \leq r$

Este paso es el más fácil de todos, y ocupará poco espacio en nuestro análisis.

En este paso simplemente vamos a añadir la siguiente condición, la cual se encuentra entre el paso 3 y el paso 5:

```
if (n <= r)
        return true;
```

Podemos optimizar esto un poco más si tenemos en cuenta que este paso solo es necesario, ya que la condición $n \leq r$ solo se cumple si $n \leq 5.690.034$, pues $\lceil \log^5(r) \rceil < r$ para todo $n > 5.690.034$.

Esta optimización puede ser útil cuando el número de cifras crezca mucho, aunque tampoco va a afectar demasiado, pues volvemos a insistir que la complejidad real está en el paso 5.

6.2.6. Comprobar identidades polinómicas

Este apartado será en el que invirtamos más tiempo, pues es en el que realmente tenemos que optimizar donde sea posible para poder tener un tiempo de ejecución razonable.

Lo 4 pasos anteriores no suponen ningún problema de eficiencia con números relativamente grandes. El tener que manejar memoria en este paso puede suponer un auténtico problema si no lo hacemos adecuadamente, pues muchas reservas de memoria pueden resultar en un tiempo de ejecución muy lejos de lo que aspiramos conseguir.

En este apartado discutiremos 2 implementaciones posibles. Cada una tiene sus ventajas e inconvenientes, los cuales detallaremos a continuación:

- **Implementación directa**. Esta implementación es la más directa, sencilla de integrar en el código fuente (ya que no hace uso de librerías externas) y la que nos da más flexibilidad al tomar distintas decisiones.

  La mayor desventaja es que será muy complicado lograr una eficiencia parecida a la que otras librerías ya han conseguido, pues mientras que esta implementación se puede realizar en un tiempo relativamente corto de desarrollo, optimizarla puede suponer un tiempo innecesariamente largo.

Además de lo mencionado, para que el algoritmo sea realmente rápido, habría que implementar la versión de la multiplicación polinómica que hace uso de la *Transformada Rápida de Fourier* (FFT). En nuestro caso usaremos el método clásico para centrarnos más en la que nos proporciona la librería siguiente.

- **NTL**. Esta librería ya tiene implementadas funciones para poder trabajar con anillos de polinomios y módulos, además de haber sido optimizada. Además, la interfaz es en C++, lo cual facilita su integración.

  Sin embargo esta librería no viene incluida con el manejador de paquetes de Conan, por lo que será necesario instalar dicha librería en el sistema o compilarla a mano, lo cual puede resultar engorroso para el usuario final.

Dicho esto, empecemos con el análisis de la implementación del paso 5.

6.2.6.1. Cálculo de cota superior para el bucle

La primera parte del paso es calcular el valor para saber cuántas iteraciones tenemos que realizar. Esta cota es $\lfloor \sqrt{\phi(r)} \log(n) \rfloor$. Para ello, recurriremos de nuevo a la librería **MPFR** para conseguir una cota lo más fiel y baja posible.

Primero necesitamos una implementación para la función ϕ de Euler, la cual podemos ver en el siguiente código. Aquí no usamos el tipo de **GMP**, pues sabemos que r cabe en el tipo *size_t*:

```
auto phi(size_t n) -> size_t {
        auto const top = size_t{std::sqrt(n)};
        auto result = n;

        for (auto p = size_t{2}; p <= top; ++p) {
                if (n %p == 0) {
                        while (n %p == 0)
                                n /= p;

                        result -= result / p;
                }
        }

        if (n > 1)
                result -= result / n;

        return result;
}
```

Esta función es una implementación sencilla que simplemente va calculando el valor de $\phi(n)$ a medida que va factorizando n. Esta implementación además evita el uso de números en coma flotante, lo cual ayuda a obtener resultados exactos sin recurrir a aproximaciones.

Ahora pasamos a explicar la función que calcula la cota. El código para ello es el siguiente:

```
auto upperBoundStep5(mpz_class n, size_t r) -> size_t {
        mpfr_t result;
```

```
        mpfr_init_set_z(result, n.get_mpz_t(), MPFR_RNDU);
        mpfr_log2(result, result, MPFR_RNDU);

        mpfr_t sqrtPhiR;
        mpfr_init_set_ui(sqrtPhiR, phi(r), MPFR_RNDU);
        mpfr_sqrt(sqrtPhiR, sqrtPhiR, MPFR_RNDU);

        mpfr_mul(result, result, sqrtPhiR, MPFR_RNDU);

        return mpfr_get_ui(result, MPFR_RNDD);
}
```

Empezamos calculando $\log(n)$, lo cual lo hacemos fácilmente con las tres primeras líneas. Para ello inicializamos una variable de tipo *mpfr_t* con n (**MPFR** admite conversiones desde tipos de **GMP**) y luego usamos *mpfr_log2* para aplicarle el logaritmo en base 2 y así obtener $\log(n)$.

Lo siguiente es calcular $\sqrt{\phi(r)}$ en las tres siguientes líneas. Inicializamos otra variable de tipo *mpfr_t* con el valor de llamar a la función *phi* con r, teniendo así $\phi(r)$. Ahora usamos la función *mpfr_sqrt* para calcularle la raíz cuadrada, obteniendo así $\sqrt{\phi(r)}$.

Después calculamos $\sqrt{\phi(r)}\log(n)$ usando la función *mpfr_mul* y acumulando el resultado en la primera variable que creamos (para no reservar más memoria).

Finalmente devolvemos $\lfloor\sqrt{\phi(r)}\log(n)\rfloor$ usando el resultado de la llamada a *mpfr_get_ui*, que devuelve el valor del resultado como un entero sin signo y redondeando hacia $-\infty$.

Ahora vamos a pasar a explicar las implementaciones del bucle principal del algoritmo. Es solo esta parte en la que divergen varias implementaciones en todo el algoritmo. Esta separación nos servirá luego para poder comparar ambas implementaciones y ver las ventajas de una sobre la otra.

Destacar que ambas implementaciones residen en el namespace **tfg::aks::steps::impl** y están expuestas públicamente. La primera se llama *step5Direct*, y la segunda *step5NTL*. En la implementación final se usa por defecto la segunda por ser más eficiente.

6.2.6.2. Bucle: Implementación Directa

Ahora vamos a centrarnos en cómo podríamos hacer una implementación directa sin hacer uso de librerías externas para el bucle principal del algoritmo **AKS**.

Para ello necesitamos implementar la operación principal de la identidad: exponenciación rápida de un polinomio módulo otro polinomio y un entero. Esta operación puede parecer aparentemente sencilla, pero requiere de un buen manejo de la memoria para no estar reservando memoria constantemente en cada iteración del bucle.

Vamos a presentar entonces dos clases que nos ayudarán a la hora de la implementación:

- **AKSCoefficient**: Esta clase es simplemente un wrapper de *mpz_class* para trabajar con aritmética modular más fácilmente.

El único atributo de dichos objetos es una variable de tipo *mpz_class*, donde las operaciones de suma, resta y multiplicación se han adaptado al anillo $\mathbb{Z}_n$.

Además, para que todos los objetos de dicha clase tengan el mismo módulo, se ha añadido una variable estática de clase (común a todos los objetos) que indicará el anillo en el que nos encontramos.

- **AKSPolynomial**: Esta clase representa un polinomio con coeficientes en $\mathbb{Z}_n$ y módulo $X^r - 1$ o, dicho de otro modo, el anillo $\mathbb{Z}_n[X]/(X^r - 1)$. Agruparlo así nos servirá para controlar mejor el uso de memoria.

  Consta de tres atributos:

  - Dos buffers de tamaño $2r$. Uno para almacenar los coeficientes del polinomio. El otro es un buffer auxiliar que nos servirá para evitar reservas de memoria repetidas cada vez que hagamos una operación sobre los polinomios.
  - Un entero sin signo indicando el grado actual del polinomio.

  Además, hay una variable estática de clase que indicará el grado del polinomio que marca el módulo (si el polinomio es $X^r - 1$, nosotros guardamos el valor de r).

Para ambas clases implementaremos lo justo para poder usarlas con el algoritmo **AKS**, y además no estarán expuestas en la interfaz pública, por lo que no necesitamos una API extremadamente versátil.

La implementación de **AKSCoefficient** es simplemente un wrapper con las operaciones elementales adaptadas al anillo $\mathbb{Z}_n$, por lo que no vamos a ocupar mucho tiempo en ella.

Decir que los operadores que se implementan son +=, -= y *=. Esto es para evitar que se hagan muchas reservas de memoria y simplemente actualizar los valores. Además se implementa el operador == para poder comparar dos objetos de esta clase, constructores que aceptan variables de tipo *mpz_class* y una pareja getter/setter para cambiar el módulo del anillo, es decir, indicar en qué $\mathbb{Z}_n$ estamos.

La clase **AKSPolynomial** es la que va a hacer el trabajo pesado, pues es la que se va a encargar de implementar las operaciones polinómicas. Vamos a indicar varias características de esta clase:

- Una característica importante de esta clase es que no se puede copiar. Tiene el constructor y la asignación por copia eliminados explícitamente. Esto nos va a ayudar a que no se hagan copias accidentalmente.
- Tiene un constructor que acepta dos variables de tipo **AKSCoefficient**, que indican los dos primeros coeficientes. Esto es porque en el algoritmo solo necesitamos construir polinomios de esa manera y no con tantos coeficientes como queramos.
- Los únicos operadores que se implementan son -= y *=. El primero acepta una variable de tipo **AKSCoefficient** (básicamente restar un escalar al polinomio). El segundo acepta otro polinomio, y es donde se implementará la operación de multiplicación de polinomios.

- Se implementa la operación *pow*, que acepta una variable de tipo *mpz_class* y eleva el polinomio a dicho valor. Además acepta un segundo parámetro que es donde se guardará el resultado de la operación (esto con el fin de evitar reservar memoria repetidas veces).

Empezamos viendo la operación *pow*. El código es el siguiente:

```
auto AKSPolynomial::pow(mpz_class exp, AKSPolynomial& result) const -> void {
        result.setCoefficient(0, 1_mpz);
        result.m_currentDegree = 0;

        for (auto const& bit : exp.get_str(2))
        {
                result *= result;
                if (bit == '1')
                        result *= *this;
        }
}
```

El algoritmo simplemente va tomando los bits del número que se le pasa con la función *get_str*. Los bits se recorren de más significativo a menos. Simplemente elevamos al cuadrado en cada iteración el resultado y, si el bit es 1, multiplicamos por el valor que estamos elevando. Este algoritmo es bien conocido y realiza $O(\log(n))$ multiplicaciones.

Ahora vamos a presentar una función auxiliar, *adjustDegree*, que sirve para adaptar el grado del polinomio:

```
auto AKSPolynomial::adjustDegree() -> void {
        while (getCoefficient(getDegree()) == 0_mpz && getDegree() > 0)
                --m_currentDegree;
}
```

Las funciones *getCoefficient* y *getDegree* son getters para obtener un coeficiente específico y obtener el grado del polinomio respectivamente.

Simplemente va actualizando el grado del polinomio hasta que se encuentra un coeficiente distinto de 0 o llega al grado 0.

La siguiente operación que vamos a presentar es la división del polinomio por el módulo:

```
auto AKSPolynomial::dividePolMod() -> void {
        if (getDegree() >= getModuleDegree()) {
                for (auto i = getDegree(); i >= getModuleDegree(); --i)
                        m_coeffs[i - getModuleDegree()] += getCoefficient(i);

                m_currentDegree = getModuleDegree() - 1;
        }
        adjustDegree();
}
```

Antes de explicar la función, destacar que la función *getModuleDegree* devuelve el grado del polinomio $X^r - 1$, pues para el algoritmo no necesitamos el polinomio entero y podemos realizar ciertas optimizaciones basadas en ello. La variable *m_coeffs* es el buffer que contiene

los coeficientes.

Básicamente, lo que hacemos es aplicar el algoritmo clásico de división de polinomios. Como sabemos que el polinomio siempre es de la forma $X^r - 1$, podemos simplemente actualizar los $n - r$ primeros coeficientes desde el de más grado hasta el de menos. Finalmente reajustamos el grado del polinomio resultante. De este modo, la eficiencia es $O(n - r)$ o $O(r)$ teniendo en cuenta que el polinomio siempre será de grado $O(r)$.

Finalmente vamos a presentar la multiplicación. Cabe destacar que aquí usamos el algoritmo elemental. Para conseguir una mejor eficiencia será necesario usar la versión que hace uso de la *Transformada Rápida de Fourier*. Esto no lo haremos aquí ya que puede ser complicado implementarla correctamente y las próximas versiones ya harán ese trabajo por nosotros.

```
auto AKSPolynomial::operator*=(AKSPolynomial const& rhs) -> AKSPolynomial& {
        auto const newDegree = getDegree() + rhs.getDegree() + 1;

        for (auto i = std::size_t{0}; i <= newDegree; ++i)
                m_coeffsAux[i] = 0_mpz;

        for (auto i = std::size_t{0}; i <= getDegree(); ++i) {
                for (auto j = std::size_t{0}; j <= rhs.getDegree(); ++j) {
                        auto product = getCoefficient(i);
                        product *= rhs.getCoefficient(j);
                        m_coeffsAux[i + j] += product;
                }
        }

        m_currentDegree = newDegree;
        std::swap(m_coeffs, m_coeffsAux);

        dividePolMod();

        return *this;
}
```

Explicamos cada paso con detalle:

1. Primero calculamos el grado final del polinomio resultante (la suma de ambos grados más 1).
2. Actualizamos los valores del buffer auxiliar, *m_coeffsAux* a 0 (preparando el terreno para acumular el resultado).
3. Bucle doble donde básicamente aplicamos el algoritmo elemental de multiplicación de polinomios y acumulamos el resultado en el buffer auxiliar.
4. Actualizamos el grado del polinomio actual al del producto.
5. Intercambiamos los buffers, de modo que el buffer principal contenga el resultado de la multiplicación.
6. Finalmente aplicamos la división por el módulo, la cual ya ajusta el grado del resultado final.

La parte que deberíamos optimizar es el bucle anidado donde realizamos el algoritmo de multiplicación, pero no lo haremos en este apartado ya que será complicado hacerlo correctamente.

Finalmente, y haciendo uso de lo que acabamos de explicar, podemos implementar el quinto paso de la siguiente manera:

```
auto step5Direct(mpz_class n, size_t r) -> bool {
        auto const top = calculateUpperBound(n, r);

        detail::AKSCoefficient::setModule(n);
        detail::AKSPolynomial::setModuleDegree(r);

        auto lhs = detail::AKSPolynomial{};
        auto rhs = detail::AKSPolynomial{};

        auto temp = detail::AKSPolynomial{0_mpz, 1_mpz};

        temp.pow(detail::AKSCoefficient::getModule(), rhs);

        for (auto a = std::size_t{1}; a <= top; ++a) {
                temp.setCoefficient(0, mpz_class{a});
                temp.pow(detail::AKSCoefficient::getModule(), lhs);
                lhs -= mpz_class{a};

                if (lhs != rhs)
                        return true;
        }

        return false;
}
```

Antes de explicar cada paso en detalle, es importante aclarar que la identidad que vamos a comprobar la vamos a modificar ligeramente. En vez de comprobar $(X+a)^n \equiv X^n + a \mod (X^r - 1, n)$, vamos a comprobar $(X+a)^n - a \equiv X^n \mod (X^r - 1, n)$. Esto nos permite evitar calcular el polinomio de la parte derecha en cada iteración, ya que no dependerá de la variable de iteración a. Ahora explicamos en detalle cada paso:

1. Primero calculamos la cota del bucle con la función **calculateUpperBound**.
2. Indicamos a las clases el módulo $(X^r - 1, n)$ en el que vamos a trabajar.
3. Creamos los dos polinomios que usaremos para comprobar las identidades: *lhs* para $(X+a)^n - a$ y *rhs* para X^n. Además creamos uno extra que nos servirá para almacenar el resultado de elevar la parte izquierda y no reservar memoria repetidas veces.
4. Almacenamos en *rhs* el resultado de $X^n \mod (X^r - 1, n)$.
5. Empezamos el bucle, y en cada iteración, almacenamos en *lhs* el resultado de $(X+a)^n - a \mod (X^r - 1, n)$. Comparamos *lhs* con *rhs*, y devolvemos true si no coinciden.
6. Si llegamos al final del bucle, devolvemos false (Es decir, que se han cumplido las identidades).

La complejidad de esta implementación está entre $O^{\sim}(\log^8(n))$ y $O^{\sim}(\log 31/2(n))$ (según el valor de r). Esto se debe a que la multiplicación de enteros viene dada por la librería **GMP**,

que implementa la dicha operación con complejidad $O^{\sim}(\log(n))$; y porque el algoritmo de multiplicación de polinomios implementado sin uso de librerías externas es $O^{\sim}(r^2 \log(n))$.

6.2.6.3. Bucle: Implementación con NTL

En este apartado nos vamos a centrar en implementar el paso 5 haciendo uso de la librería **NTL**.

Esta implementación hace uso de algoritmos para realizar la multiplicación de polinomios con complejidad $O^{\sim}(nm)$ en vez de $O^{\sim}(n^2 m)$, como en el apartado anterior.

Presentamos entonces una implementación del paso 5 haciendo uso de la librería **NTL**:

```
auto step5NTL(mpz_class n, size_t r) -> bool {
        auto const top = calculateUpperBound(n, r);

        auto const nNTL = NTL::conv<NTL::ZZ>(n.get_str().c_str());
        NTL::ZZ_p::init(nNTL);

        auto const module = NTL::ZZ_pXModulus{NTL::ZZ_pX{r, 1} - 1};

        auto const rhs = [&nNTL, &module] {
                auto result = NTL::ZZ_pX{1, 1};
                NTL::PowerMod(result, result, nNTL, module);

                return result;
        }();

        for (auto a = std::size_t{1}; a <= top; ++a) {
                auto lhs = NTL::ZZ_pX{1, 1};
                lhs += a;
                NTL::PowerMod(lhs, lhs, nNTL, module);
                lhs -= a;

                if ((lhs != rhs) != 0)
                        return true;
        }

        return false;
}
```

Primero, al igual que en la implementación anterior, calculamos la cota superior del bucle haciendo uso de la función *calculateUpperBound*.

Después convertimos la entrada (entero de **GMP**, *mpz_class*) a un entero de **NTL**, *NTL::ZZ*, para poder usarlo en los algoritmos de esta librería. Hecho eso, inicializamos el módulo $\mathbb{Z}_n$ con la llamada a *NTL::ZZ_p::init*. Esto hará que todas las operaciones en enteros sean módulo n.

Ahora declaramos el polinomio $X^r - 1$ como el módulo que usaremos para exponenciar.

Después calculamos $X^n \mod (X^r - 1, n)$ con la función *NTL::PowMod*. El segundo parámetro es el polinomio a exponenciar (base). El tercer parámetro es el exponente. El cuarto parámetro es el polinomio cuyo módulo vamos a aplicar.El resultado se guarda en el primer

parámetro.

Finalmente ejecutamos el bucle, y en cada iteración calculamos $(X+a)^n - a \mod (X^r - 1, n)$. Si alguna identidad no se cumple, devolvemos true.

Finalmente, si llegamos al final del bucle, devolvemos false (pues todas las identidades se han cumplido).

La complejidad de esta implementación está entre $O^{\sim}(\log^6(n))$ y $O^{\sim}(\log 21/2(n))$ (según el valor de r). Esto se debe a que la multiplicación de enteros viene dada por la librería **GMP**, que implementa la dicha operación con complejidad $O^{\sim}(\log(n))$; y porque el algoritmo de multiplicación de polinomios implementado por **NTL** tiene complejidad $O^{\sim}(r\log(n))$.

6.2.7. Paso 6: Devolver true

Este paso simplemente se implementa como una función a parte para ser más fiel al algoritmo original y estar en concordancia con el resto de pasos. Consiste en una función que devuelve true.

6.3. Comparación Implementación Directa/NTL

En esta sección vamos a justificar con resultados gráficos la elección de usar la librería externa **NTL** a la hora de elegir una implementación definitiva del paso 5 del algoritmo **AKS**.

Como ya explicamos anteriormente, este paso es el único en el que usamos implementaciones distintas, por lo que el resto de pasos serán comunes en la comparación y solo mediremos el tiempo de ejecución del quinto paso.

Para la comparación vamos a usar los mayores primos que ocupan una cantidad determinada de bits (desde 2 bits hasta 16 bits). Nuestro conjunto de prueba será el siguiente:

$$\{3, 7, 13, 31, 61, 127, 251, 509, 1021, 2039, 4093, 8191, 16381, 32749, 65521\}$$

No usamos primos más grandes porque, como veremos en las gráficas, el tiempo que invierte la implementación directa es muy alto para números pequeños.

No usamos números compuestos porque necesitamos comparar la eficiencia del paso 5, el cual se ejecuta por completo cuando la entrada se trata de un número primo. Es por ello que aquí no tiene mucho sentido usar números compuestos. Por esta razón, en la comparación solo vamos a ejecutar los pasos 2 y 5, ya que necesitamos el valor de r calculado en el segundo paso para poder ejecutar el quinto.

Además de presentar los tiempos de ejecución de ambas implementaciones, también vamos a representar las gráficas de las eficiencias teóricas ajustadas con la función *fit* de *gnuplot*.

Ambos ejes de las gráficas están en escala logarítmica en base 2, para que se puedan apreciar mejor los resultados.

Esta gráfica muestra los tiempos de ejecución de la implementación directa junto con sus eficiencias teóricas.

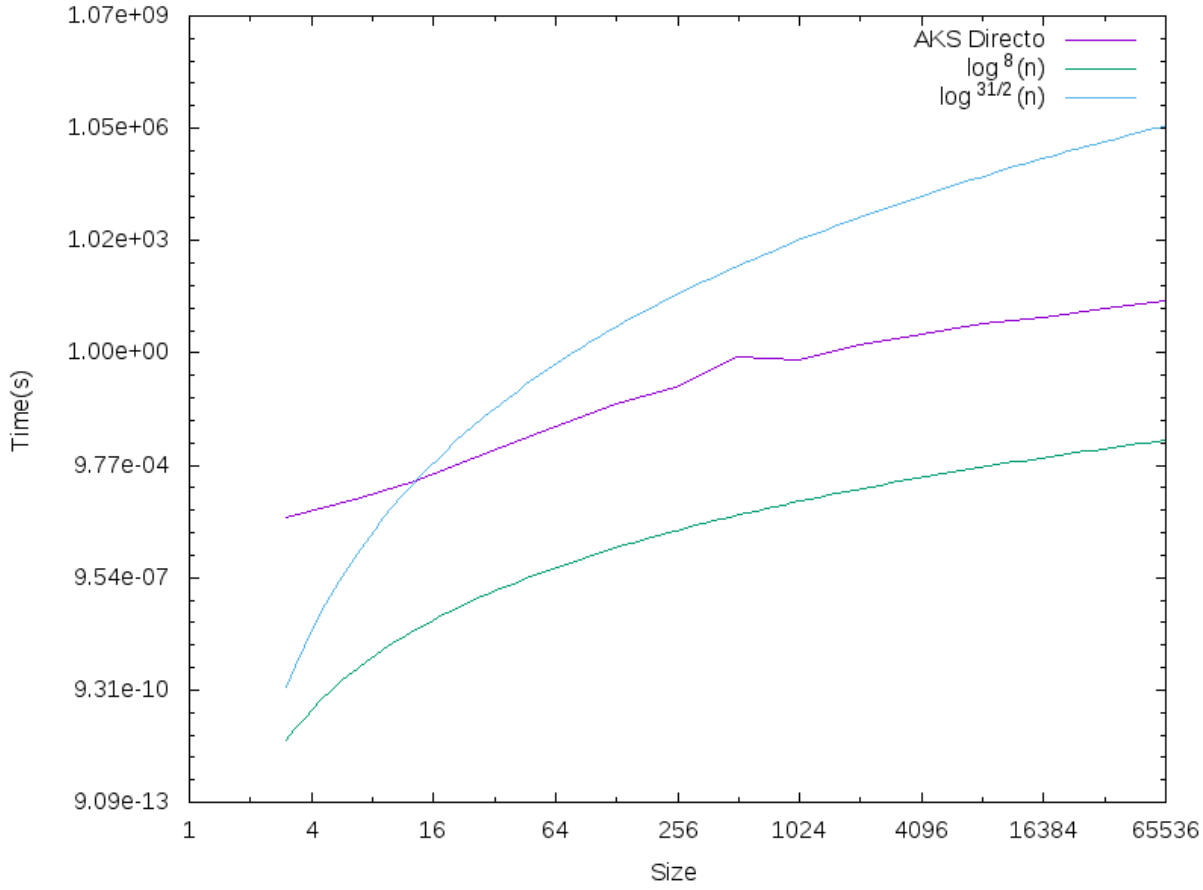

Figura 6.2.: Gráfica AKS con implementación directa

Esta gráfica muestra los tiempos de ejecución de la implementación usando **NTL** junto con sus eficiencias teóricas.

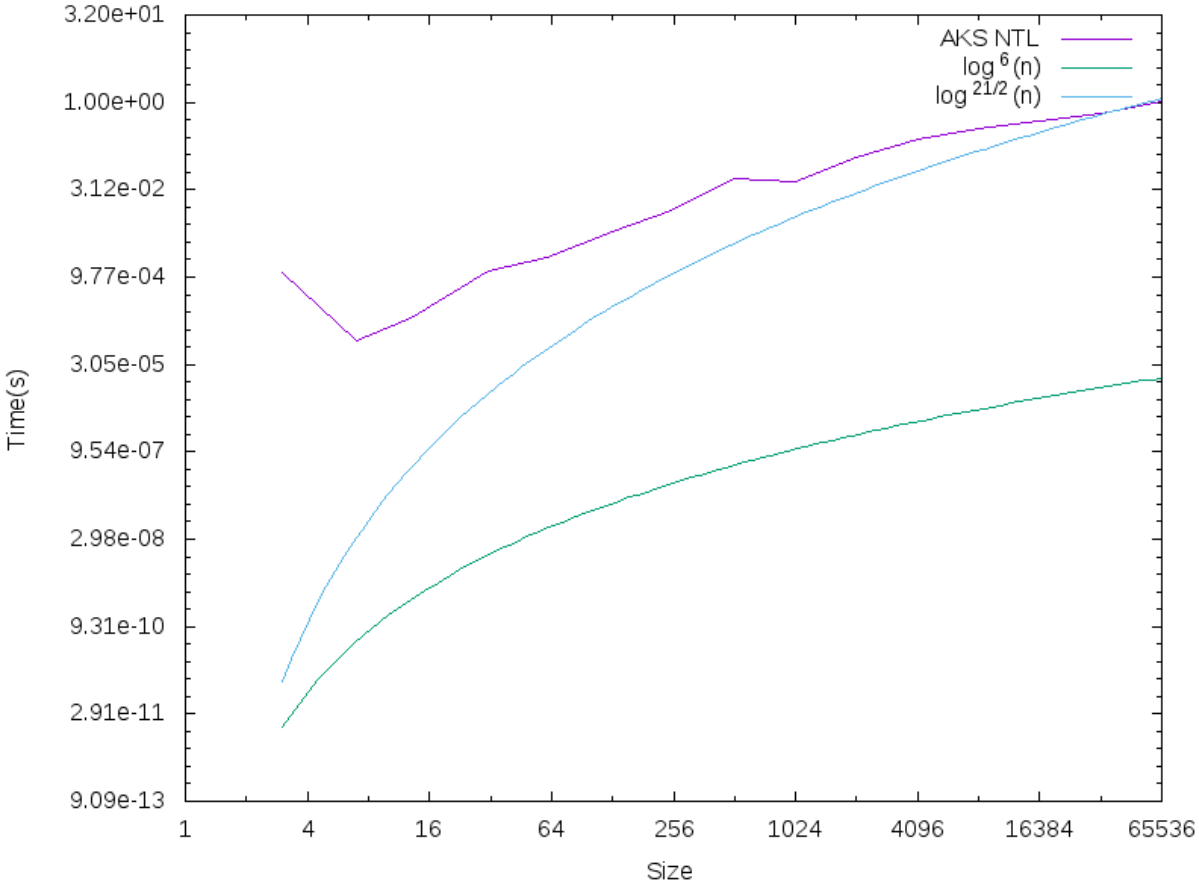

Figura 6.3.: Gráfica AKS usando NTL

Finalmente mostramos la comparación de ambas implementaciones.

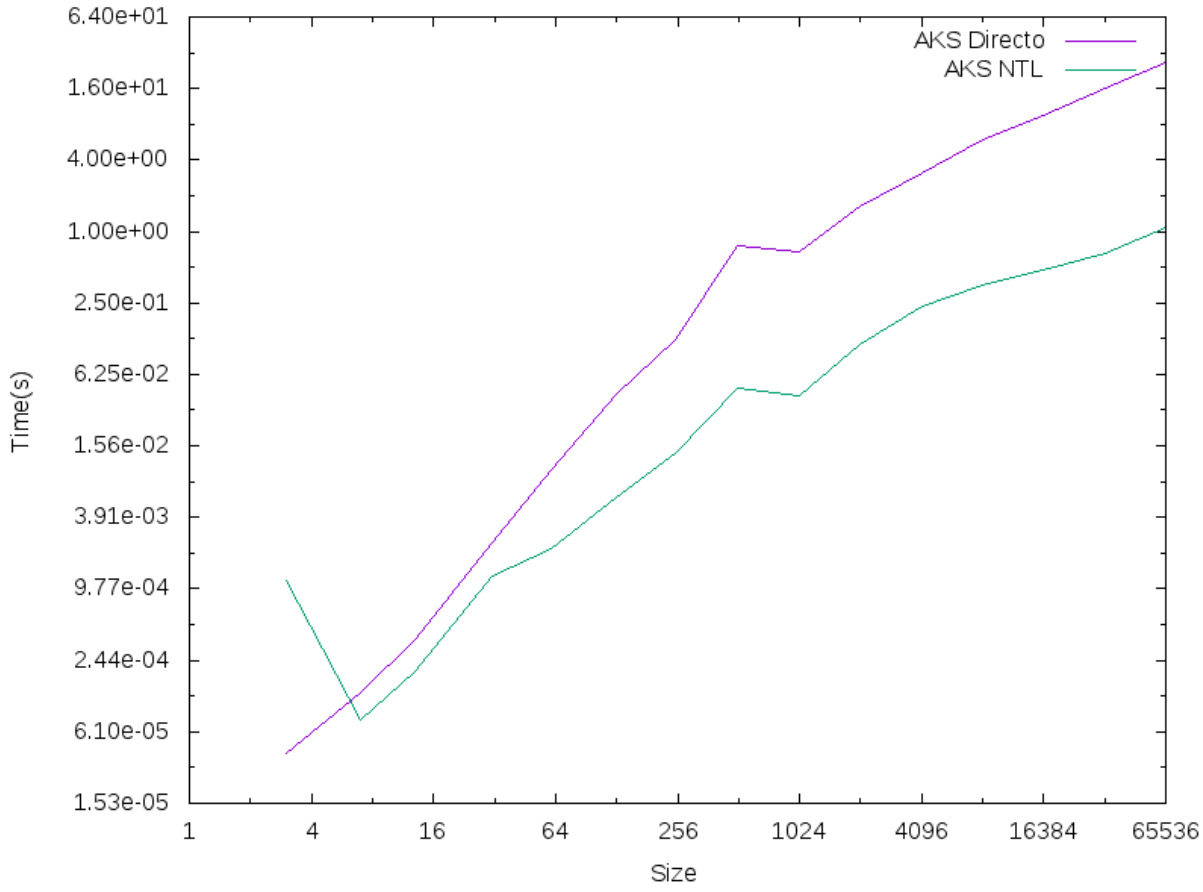

Figura 6.4.: Comparación ambas implementaciones AKS

Como podemos comprobar, el tiempo de ejecución de la implementación directa es mucho mayor que la implementación usando la librería **NTL**. Esto es debido al algoritmo de multiplicación polinómica usado en ambos casos, lo cual resalta su importancia a la hora de implementar el algoritmo **AKS**.

Puesto que la implementación usando **NTL** es superior, será la que usaremos en el siguiente apartado para comparar el algoritmo **AKS** con los tests probabilísticos de *Miller-Rabin* y *Solovay-Strassen*.

7. Comparación con algoritmos probabilísticos

En este capítulo vamos a implementar algunos tests de primalidad probabilísticos para poder comparar su tiempo de ejecución con la implementación descrita anteriormente del test **AKS**.

En específico, vamos a implementar dos tests: *Miller-Rabin* y *Solovay-Strassen*. Una vez implementados estos dos tests, haremos varias comparaciones con el test **AKS** usando distintos conjuntos de números:

- Números primos.
- Potencias de primos.
- Números compuestos no potencias de primos.

El análisis de cada conjunto irá acompañado de gráficas que representen visualmente los tiempos de ejecución de los distintos tests. Para cada conjunto analizaremos los resultados correspondientes y sacaremos conclusiones respecto a la eficiencia de cada test.

7.1. Tests Probabilísticos

En esta sección vamos a presentar las dos implementaciones de los tests probabilísticos que vamos a usar, además de explicarlos brevemente.

La base teórica de ambos test ya la explicamos anteriormente, y en esta solo nos vamos a centrar en la implementación de los mismos.

En ambos casos, la entrada consiste de tres parámetros, descritos a continuación en el mismo orden:

1. Número cuya posible primalidad queremos comprobar.
2. Número de rondas del test probabilístico a realizar.
3. (Opcional) Generador de números aleatorio. Esto puede ser útil a la hora de testear el código de manera determinista. En caso de que no se pase ninguno, se usará uno con una semilla generada aleatoriamente.

7.1.1. Test de Miller-Rabin

Ahora vamos a presentar una implementación del test de *Miller-Rabin*. Dicha implementación está expuesta públicamente y se encuentra en el namespace *tfg::miller_rabin*:

```
auto isProbablyPrime(mpz_class n, mpz_class k, gmp_randclass &prng) -> bool {
        if (n == 2_mpz || n == 3_mpz)
                return true;

        if (n < 2 || n %2 == 0)
                return false;

        auto const [r, d] = [&n] {
                auto dResult = mpz_class{n - 1};
                auto rResult = mpz_scan1(dResult.get_mpz_t(), 0);
                mpz_fdiv_q_2exp(dResult.get_mpz_t(), dResult.get_mpz_t(), rResult);

                return std::make_pair(rResult, dResult);
        }();

        for (auto i = 0_mpz; i < k; ++i) {
                auto const a = mpz_class{prng.get_z_range(n - 3) + 2};
                auto x = gmp::powMod(a, d, n);

                if (x != 1 && x != n - 1) {
                        for (auto j = 0_mpz; j < r - 1; ++j) {
                                x = gmp::powMod(x, 2, n);

                                if (x == n - 1)
                                        break;
                        }

                        if (x != n - 1)
                                return false;
                }
        }

        return true;
}
```

Primero nos libramos de los múltiplos de dos con las dos primeras condiciones. Además manejamos el caso $n = 3$ para asegurar que el test de *Miller-Rabin* solo lo aplicamos a enteros impares mayores que 3.

Después encontramos r, d tales que $n = 2^r d + 1$.

Después ejecutamos el test de *Miller-Rabin* el número de rondas que le hemos pasado y, para cada ronda, generamos un número aleatorio entre 2 y $n - 2$ (ambos inclusive), el cual usaremos para comprobar las congruencias (2.1).

Si en alguna ronda no se pasa el test, se devuelve false (el número es compuesto). Si llegamos al final del bucle, entonces devolvemos true (el número es probablemente primo).

7.1.2. Test de Solovay-Strassen

Ahora vamos a presentar una implementación del algoritmo de Solovay-Strassen. Dicha implementación está expuesta públicamente y se encuentra en el namespace *tfg::solovay_strassen*:

```
auto isProbablyPrime(mpz_class n, mpz_class k, gmp_randclass &prng) -> bool {
        if (n == 2)
```

```
                return true;

        if (n < 2 || n %2 == 0)
                return false;

        for (auto i = 0_mpz; i < k; ++i) {
                auto const a = mpz_class{prng.get_z_range(n - 2) + 2};
                auto const x = [&a, &n] {
                        auto result = gmp::jacobiSymbol(a, n);
                        return (result < 0) ? n + result : result;
                }();

                if (x == 0 || gmp::powMod(a, (n - 1)/2, n) != x)
                        return false;
        }

        return true;
}
```

Primero nos libramos de los múltiplos de 2.

Una vez hecho eso, simplemente ejecutamos el test el número de rondas que se ha pasado con números aleatorios generados entre 2 y $n - 1$. El *Símbolo de Jacobi* 2.4 lo hallamos usando la función que nos proporciona **GMP** para ello (usando el wrapper que hemos creado para C++).

Igual que con el test de *Miller-Rabin*, si no se pasa el test para alguna ronda, devolvemos false (compuesto). Si llegamos al final, devolvemos true (probablemente primo).

7.2. Comparaciones

En esta sección vamos a comparar estos dos tests probabilísticos con la implementación usando la librería **NTL** descrita anteriormente del algoritmo **AKS**.

Para ello prepararemos números primos cuya cantidad de bits es creciente, y así poder tener una idea de cómo se comportan los algoritmos a medida que crecen la cantidad de bits de las entradas.

Dichas entradas serán ejecutadas en los distintos algoritmos cinco veces, y se hará una media aritmética de los tiempos de ejecución para obtener un resultado más fiable.

Todas estas mediciones se realizarán en una máquina cuya CPU tiene una frecuencia de $1.7GHz$, $16GB$ de memoria RAM y $240GB$ de memoria sólida o SSD. Las mediciones se realizarán en una única hebra para obtener resultados aún más fiables.

Los números primos que usaremos serán los mayores para una cantidad determinada de bits. Por ejemplo: 3 es el mayor primo que ocupa 2 bits, 7 para 3 bits, 31 para 5 bits, 65521 que ocupa 16 bits, etc.

La generación de dichos primos se encuentra en Apéndice C.

Las gráficas se presentan con ambos ejes en escala logarítmica en base 2, para poder apreciar mejor los resultados. Además, las gráficas están generadas con *Gnuplot*. Más información en Apéndice C.

Como ya explicamos anteriormente, la cantidad de rondas a ejecutar en los tests probabilísticos será 40 según [dig13]. Esto solo aplica al test de *Miller-Rabin*. Para el test de *Solovay-Strassen*, puesto que queremos que ambas implementaciones tengan aproximadamente las mismas probabilidades de fallar, ejecutaremos el doble de rondas (ya que este test tiene el doble de posibilidades de fallar, como explicamos anteriormente).

Los tests probabilísticos aceptan, además del número cuya primalidad queremos probar y la cantidad de rondas, un generador de números aleatorios. Esto nos va a permitir que, al realizar las mediciones, obtengamos los mismos resultados siempre y cuando utilicemos el mismo generador en el mismo estado en cada ejecución. Es por ello que utilizaremos la misma semilla para inicializar el generador de números aleatorios en todas las ejecuciones. Dicho generador es el conocido *Mersenne-Twister*. Se puede encontrar más información de dicho generador en Apéndice B.

7.2.1. Números Primos

En esta sección vamos a realizar una comparación cuando las entradas son números primos. Esta comparación es la más importante, pues es la que de verdad nos va a dar una idea del tiempo de ejecución de los distintos tests en el peor de los casos (cuando la entrada es un número primo).

Como ya explicamos antes, las entradas que usaremos serán los mayores primos que ocupan una cantidad determinada de bits. En específico, llegaremos hasta los 32 bits. La razón de este límite superior se debe a que el test **AKS**, con la implementación actual, tarda más de 20 minutos en ejecutarse para un primo de 32 bits, mientras que los otros dos tests no llegan al segundo.

Se ha considerado entonces que dicho conjunto de prueba es suficiente para el análisis que más adelante realizaremos.

Hecha esta introducción, veamos una gráfica de los tiempos de ejecución los tres test.

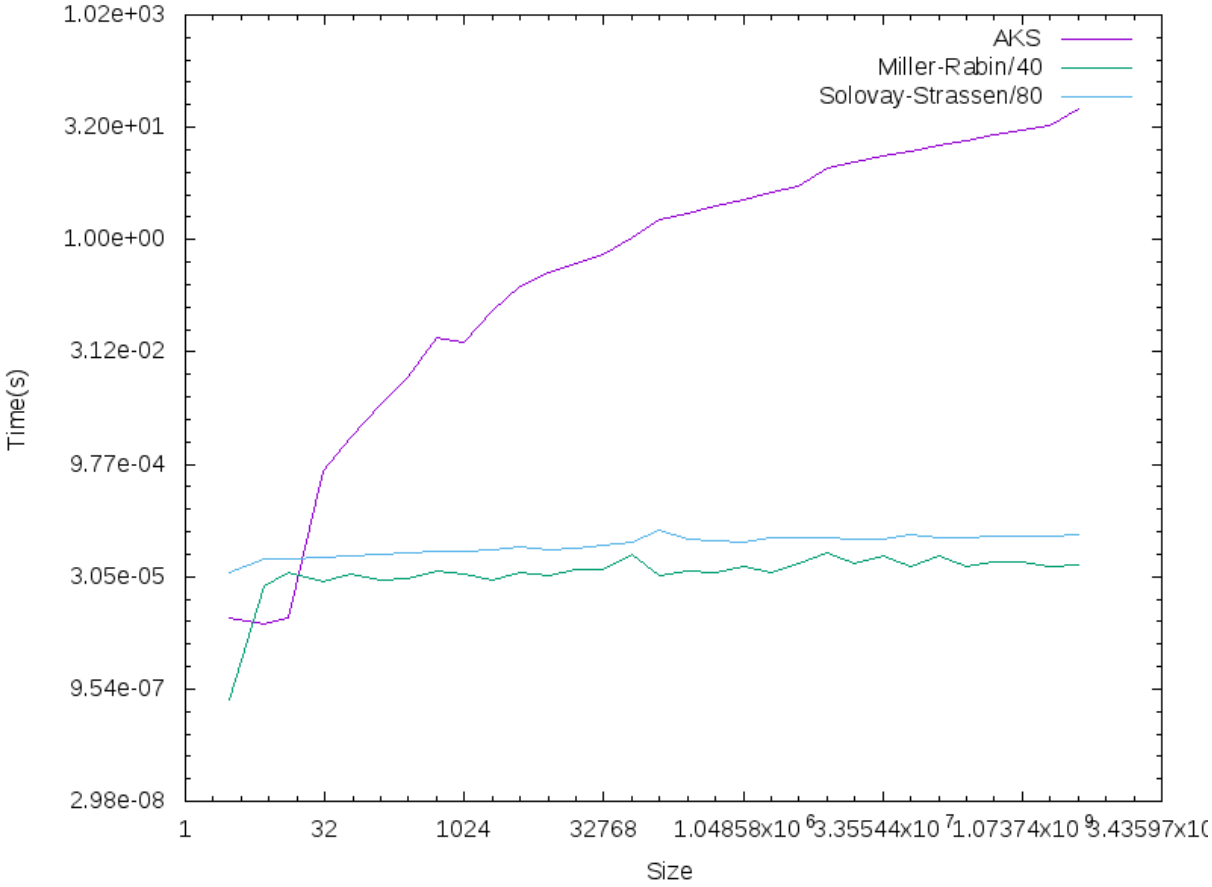

Figura 7.1.: Comparación AKS, Miller-Rabin/40 y Solovay-Strassen/80 con números primos

Como podemos comprobar, para entradas pequeñas, el algoritmo **AKS** funciona muy bien e, incluso, superando a los dos test probabilísticos. Sin embargo, vemos que entorno a $32 = 2^5$, el tiempo de ejecución se dispara, lo cual deja claro lo ineficiente del test **AKS**.

El test de *Solovay-Strassen* es un poco peor que el de *Miller-Rabin* porque realizamos el doble de rondas para asegurar probabilidades similares.

En conclusión, los tests probabilísticos funcionan mucho más rápido, lo cual es muy útil cuando estamos tratando con números muy grandes, además de que sus posibilidades de dar una respuesta errónea son prácticamente nulas debido a la cantidad de rondas.

7.2.2. Potencias de Primos

Puesto que el primer paso del test **AKS** es comprobar si la entrada es una potencia perfecta, es interesante ver cómo se comporta frente a los algoritmos probabilísticos con entradas que son potencias de primos.

Para esta comparación vamos a usar dos conjuntos distintos de prueba:

- Potencias grandes de primos pequeños. En específico, primos de hasta 16 bits elevados a 100.
- Potencias pequeñas de primos grandes. En específico, primos de hasta 256 bits elevados a 5.

Primero empecemos con las potencias grandes de primos pequeños.

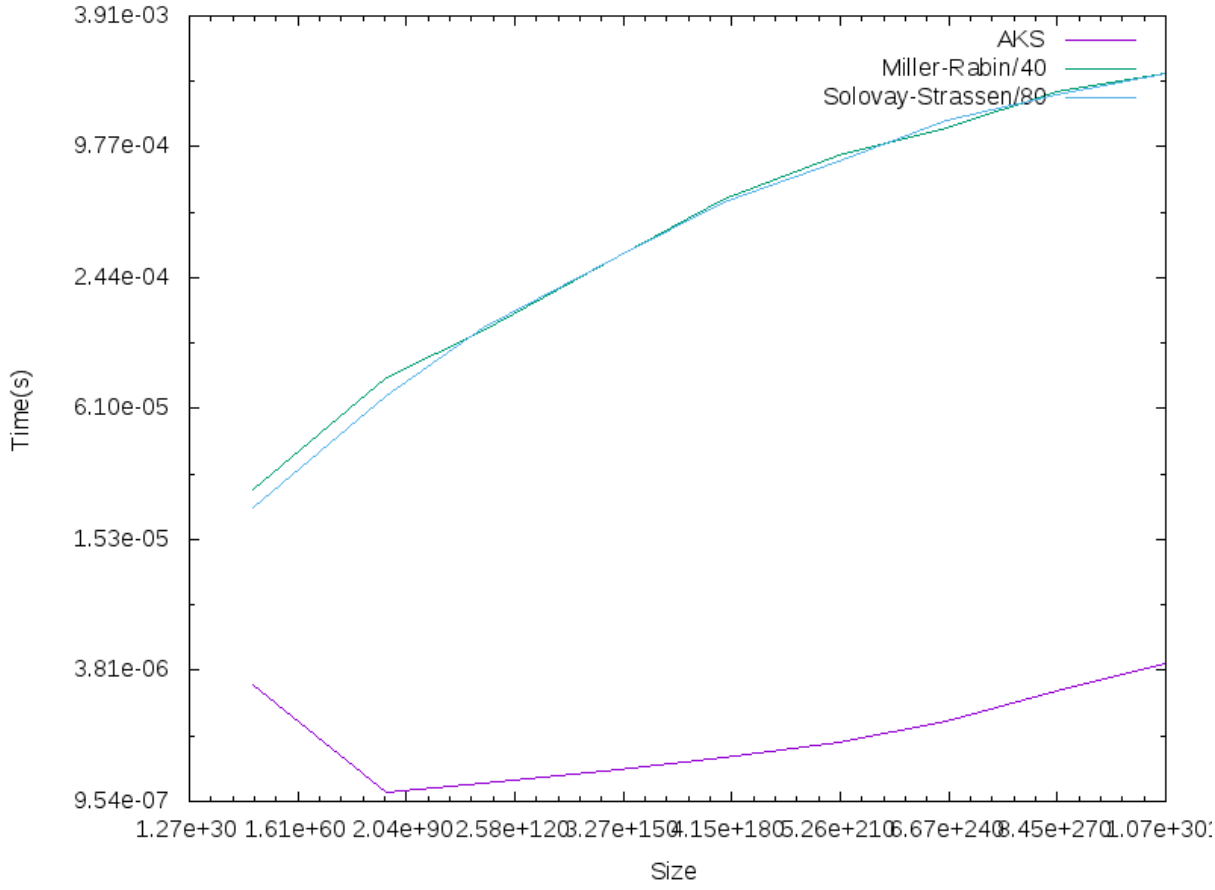

Figura 7.2.: Comparación AKS, Miller-Rabin/40 y Solovay-Strassen/80 con potencias grandes de primos pequeños

Y ahora la gráfica de potencias pequeñas de primos grandes.

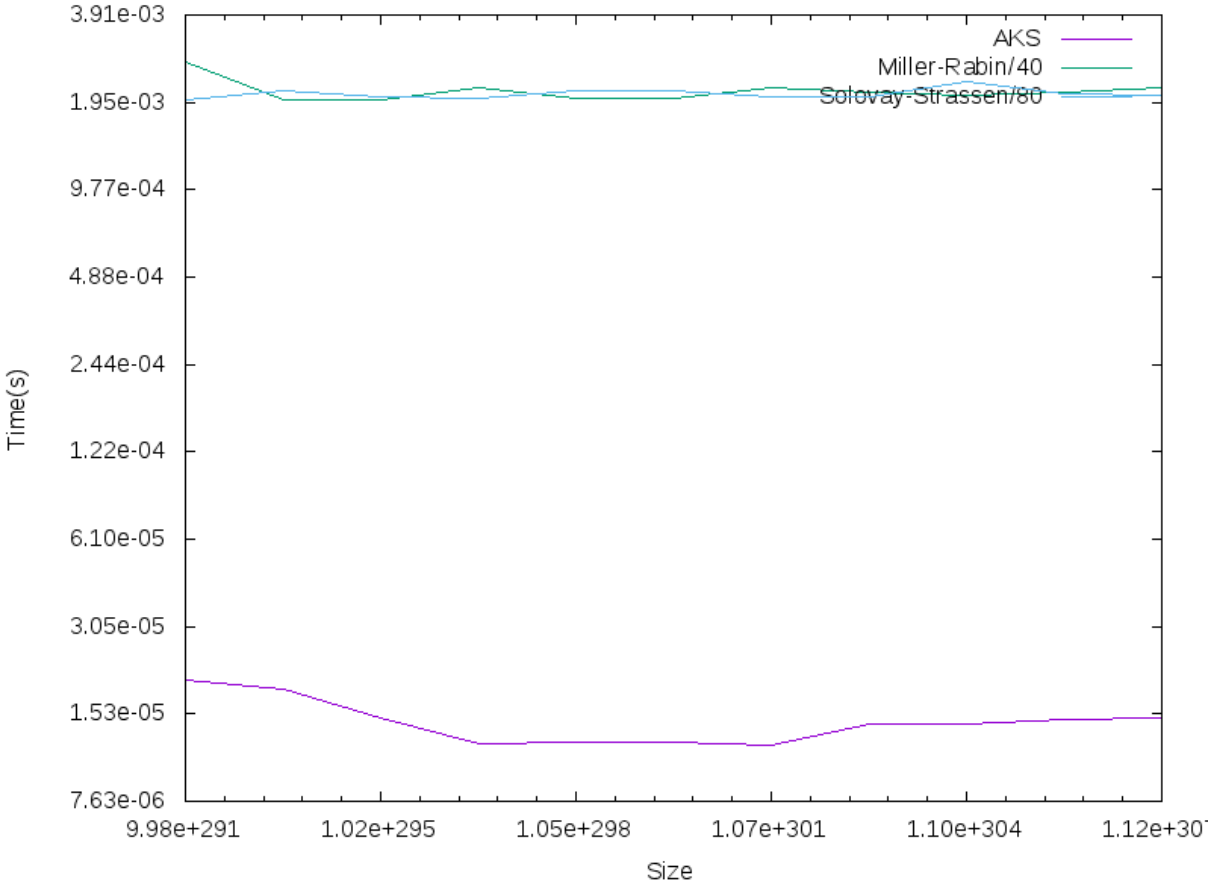

Figura 7.3.: Comparación AKS, Miller-Rabin/40 y Solovay-Strassen/80 con potencias pequeñas de primos grandes

En ambos casos, el análisis es claro. Comprobar si un número es una potencia perfecta es mucho más rápido que aplicar el test de *Miller-Rabin* o el de *Solovay-Strassen*.

Puesto que el test **AKS** maneja las potencias perfectas en el primer paso, y en vista de las gráficas anteriores, concluimos que el test **AKS** funciona mucho mejor que los test probabilísticos cuando la entrada se trata de una potencia perfecta.

7.2.3. Números Compuestos No Potencias de Primos

Habiendo hecho comparaciones con números primos y potencias de primos, es natural comparar usando números compuestos que no sean potencias de primos. Para ello usaremos números que son producto de dos o más factores primos grandes.

Esto nos ayudará a comprobar cómo se comportan los tres tests en los casos más sensibles, es decir, aquellos donde los factores primos son grandes. Determinar correctamente la composición de dichos números es de vital importancia en los protocolos de seguridad como

RSA, pues de lo contrario, la seguridad de dicho sistemas se podría ver comprometida.

Usaremos distintos conjuntos de prueba:

- Compuestos con factores primos grandes de magnitud similar. En particular, números compuestos que sean producto de un número de 32 bits y otro que tenga entre 33 bits y 42 bits.
- Compuestos con factores primos de magnitudes distintas. En particular, números compuestos con factores de 16 bits y otro que tenga entre 33 bits y 42 bits.

La decisión de no elegir factores más grandes se debe a que el algoritmo **AKS** empezaba a tardar mucho tiempo con números más grandes, y se ha considerado que estos conjuntos de prueba son suficientes para reafirmar la idea de que el test **AKS** es muy lento comparado con los probabilísticos. Vayamos ahora con la gráfica del primer conjunto de prueba.

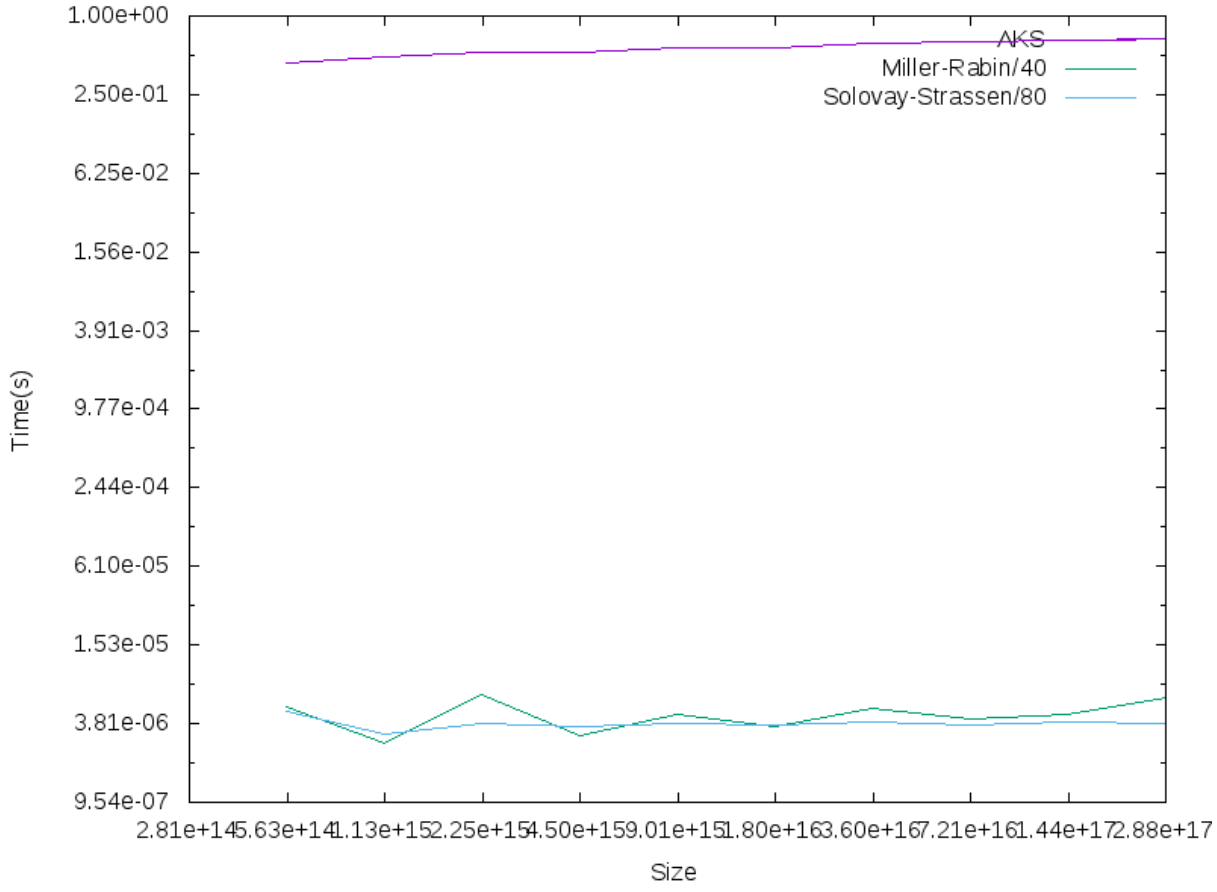

Figura 7.4.: Comparación AKS, Miller-Rabin/40 y Solovay-Strassen/80 con productos de primos de más de 32 bits y otro de 16 bits

Ahora veamos la gráfica del segundo.

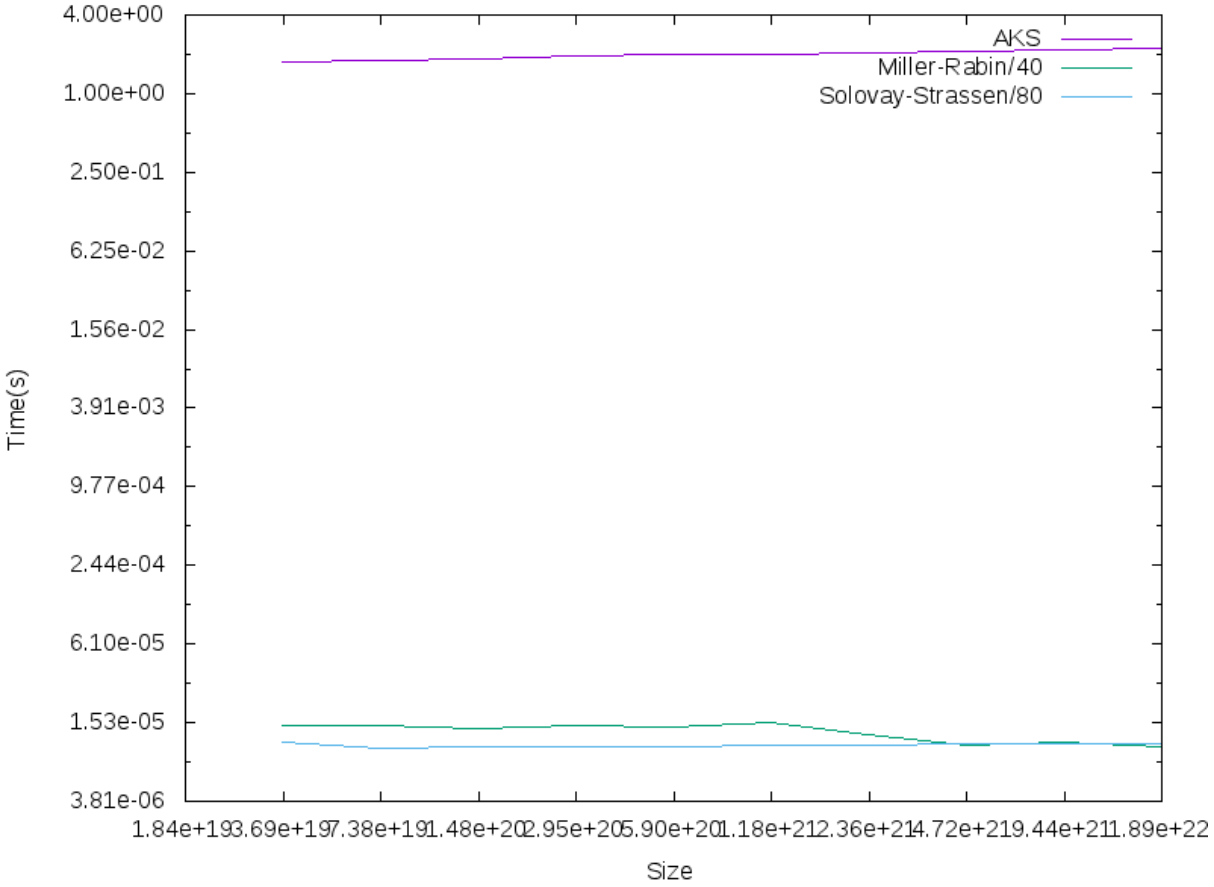

Figura 7.5.: Comparación AKS, Miller-Rabin/40 y Solovay-Strassen/80 con productos de primos de más de 32 bits y otro de 32 bits

Aparentemente, podemos comprobar que las gráficas son idénticas y que el tamaño de los factores no afecta en cómo se comportan el test **AKS** respecto de los otros.

Volvemos a comprobar, al igual que ya pasó con el caso en que la entrada eran primos, que el algoritmo **AKS** es bastante peor que sus contrapartes probabilísticas.

7.3. Conclusión

Como conclusión final de este apartado, y vistas las observaciones realizadas en esta comparación con los algoritmos probabilísticos, es que el algoritmo **AKS**, aún siendo general, polinómico, determinista e incondicional, es excesivamente lento para entradas no muy grandes, lo cual hace inviable su uso en otras aplicaciones de la criptografía.

Los algoritmos probabilísticos, aunque su respuesta no sea determinista, son suficientemente fiables en la mayoría de los casos, y su tiempo de ejecución los hace extremadamente convenientes cuando se trata de números con una gran cantidad de cifras.

8. Conclusiones y Vías Futuras

Una vez realizado el análisis teórico y empírico del algoritmo **AKS**, y hechas las comparaciones con los tests de *Miller-Rabin* y *Solovay-Strassen*, vamos a realizar algunas conclusiones y mejoras a realizar.

8.1. Conclusiones

Uno de los objetivos prioritarios del trabajo era comprobar que el algoritmo **AKS** tiene complejidad polinómica en el número de cifras de la entrada.

Conseguido eso, lo siguiente ha sido realizar una implementación del algoritmo que funcionase. Dicha implementación fue realizada en un primer lugar usando algoritmos elementales de multiplicación de polinomios, lo cual dio como resultado un test extremedamante lento.

La manera de solucionar dicho problema fue realizar otra implementación, pero usando la librería **NTL**, la cual ya proporciona buenos algoritmos de multiplicación polinómica. Esta nueva implementación, aún no siendo muy eficiente, era ya superior que la primera.

Realizada la implementación, procedimos a compararla con los algoritmos probabilísticos de *Miller-Rabin* y *Solovay-Strassen*. Entre nuestros objetivos no se encontraban comprobar que el test **AKS** es más rápido, sino más bien todo lo contrario.

De hecho, la conclusión a la que hemos podido llegar es que el test **AKS** tiene un gran interés teórico por ser el primer test general, determinista, polinómico e incondicional. Sin embargo, las constantes ocultas de dicho test son muy elevadas, lo que provoca que quede por detrás respecto de otros tests como los ya mencionados.

8.2. Ampliaciones futuras

A pesar de que los resultados en este trabajo son referentes al algoritmo **AKS** y sus comparaciones con otros algoritmos probabilísticos, podemos extender este análisis por varias ramas.

8.2.1. Algoritmo AKS Mejorado

Una ampliación que podemos realizar es implementar un test basado en Conjetura 4.3. Este test implica una ligera modificación al test original, donde r lo buscamos de manera que no divida a $n^2 - 1$.

Este test tiene un tiempo de ejecución esperado de $O^{\sim}(r \log^2(n))$ [KS04]. Puesto que dicho r lo podemos encontrar en el rango $[2, 4\log(n)]$, podemos entonces asegurar que la eficiencia

de dicho test sería $O^{\sim}(\log^3(n))$.

Esta implementación nos serviría para comprobar cómo se comporta una posible versión mejorada del algoritmo **AKS** y condicionada por Conjetura 4.3, cuyo tiempo de ejecución podría incluso ser comparable al de los tests probabilísticos estudiados.

8.2.2. Comparación con Tests Deterministas

En este trabajo hemos comparado el algoritmo **AKS** con tests probabilísticos. Sería interesante comprobar cómo se comporta dicho el algoritmo con otros test deterministas.

Podemos escoger dos test:

- *Test básico*. Se trata de comparar el algoritmo **AKS** con un test de primalidad básico y determinista que deriva directamente de la definición de primalidad. Simplemente se comprueba si algún número menor que $\sqrt{n}$ divide a n.
- *ECPP*. Test basado en curvas elípticas. Se trata de un test determinista que no es polinómico, el cual se ha llegado a utilizar con números con una cantidad considerable de cifras.

Añadir estas dos comparaciones puede aportar una mejor imagen del comportamiento del algoritmo **AKS** y, en su caso, de su posible versión mejorada.

A. Complejidad Tests Probabilísticos

En este apéndice vamos a describir algorítmicamente los test de *Miller-Rabin* y *Solovay-Strassen*.

Además comprobaremos también la complejidad de dichos algoritmos, tanto sus versiones probabilísticas como las deterministas asumiendo la veracidad de la *Hipótesis de Riemann Generalizada*.

A.1. Test de Miller-Rabin

Vamos a presentar primero la versión probabilística del algoritmo.

Algorithm 4 Algoritmo de *Miller-Rabin*

procedure ISPROBABLYPRIMEMILLERRABIN(n, k) ▷ Comprueba si n es probablemente primo aplicando k tests de *Miller-Rabin*
  Sea $n = d2^s + 1$ con $d \geq 1$ impar y $s \geq 1$.
  for $i = 1$ hasta k **do**
    Sea a un entero aleatorio en $[2, n-2]$
    $x = a^d \,\%n$
    if $x = 1$ ó $x = n - 1$ **then**
      Continuamos siguiente ronda
    end if
    for $j = 1$ hasta $s - 1$ **do**
      $y = x^2 \,\%n$
      if $y = 1$ **then**
        return COMPUESTO
      end if
      $x = y$
      if $x = n - 1$ **then**
        Continuamos siguiente ronda
      end if
    end for
    return COMPUESTO
  end for
  return PROBABLEMENTE_PRIMO
end procedure

Lo primero que vemos es que el test se ejecuta k veces, por lo que vamos a centrarnos en la complejidad de cada iteración.

El bucle interior realiza $s-1$ iteraciones, que por la manera en que hemos definido s sabemos que es $O(\log(n))$, por lo tanto el bucle interior realiza $O(\log(n))$ iteraciones. En cada iteración de dicho bucle tenemos que realizar una multiplicación módulo n, cuya complejidad es $O(M(\log(n))) = O^{\sim}(\log(n))$. Juntando todas las piezas, tenemos que la complejidad del algoritmo 4 es $O^{\sim}(k\log^2(n))$. Si usamos multiplicación elemental, es decir, que $O(M(\log(n))) = O(\log^2(n))$, la complejidad sería $O(k\log^3(n))$.

Para la versión determinista no vamos a presentar el algoritmo de nuevo entero, pues lo único que cambia es que comprobamos el test para todos los $a \in [2, \min\{n-2, \lfloor 2\ln^2(n)\rfloor\}]$. Puesto que en este caso hay que realizar $O(\ln^2(n))$ operaciones en vez de k, la complejidad del algoritmo 4 en su versión determinista es $O^{\sim}(\ln^2(n)\log^2(n)) = O^{\sim}(\log^4(n))$.

A.2. Test de Solovay-Strassen

Vamos a presentar la versión probabilística del algoritmo.

Algorithm 5 Algoritmo de *Solovay-Strassen*

```
1: procedure isProbablyPrimeSolovayStrassen(n, k)        ▷ Comprueba si n es probablemente primo aplicando k tests de Solovay-Strassen
2:     for i = 1 hasta k do
3:         Sea a un entero aleatorio en [2, n − 1]
4:         x = (a/n)
5:         if x = 0 ó a^((n−1)/2) ≢ x (mod n) then
6:             return COMPUESTO
7:         end if
8:     end for
9:     return PROBABLEMENTE_PRIMO
10: end procedure
```

La complejidad depende de k, luego nos vamos a centrar en la complejidad de cada iteración del test.

Hay dos aspectos importantes a destacar en cada iteración:

- Calcular el *Símbolo de Jacobi* de a y n.
- Hacer una exponenciación.

El *Símbolo de Jacobi* se puede calcular en $O(\log(a)\log(n)) = O(\log^2(n))$ [Coh93]. La exponenciación modular requiere de $O(\log(n))$ multiplicaciones, y cada multiplicación tiene complejidad $O(M(\log(n))) = O^{\sim}(\log(n))$ usando la *Transformada Rápida de Fourier*. Esto nos proporciona una complejidad total de cada iteración de $O^{\sim}(\log^2(n))$ ($O(\log^3(n))$ usando multiplicación elemental de enteros). Puesto que esta complejidad es mayor que la de calcular el *Símbolo de Jacobi*, será la que elijamos para calcular la complejidad final. Esto nos resulta en un test con complejidad $O^{\sim}(k\log^2(n))$ usando multiplicación rápida de enteros, ó $O(k\log^3(n))$ con multiplicación elemental.

Para la versión determinista, el test es exactamente el mismo, solo que en vez de elegir a de manera aleatoria k veces, comprobamos el test para varios valores hasta un límite. Dicho límite, como ya vimos anteriormente, es $2\log^2(n)$, de modo que el bucle principal se ejecuta $O(\log^2(n))$ veces, luego la complejidad del algoritmo en su versión determinista es $O^{\sim}(\log^4(n))$ usando multiplicación rápida de enteros, y $O(\log^5(n))$ usando multiplicación elemental.

B. Otros Algoritmos

En este apéndice vamos a detallar cómo se comportan algunas de las funciones de librerías externas para que quede más claro qué fundamentos matemáticos se utilizan.

B.1. Potencias Perfectas

En esta sección vamos a ver cómo funciona la implementación de **GMP** de la función *mpz_perfect_power_p*, la cual devuelve un entero distinto de 0 si su entrada es una potencia perfecta.

La idea está en usar el *Método de Newton* para encontrar raíces con la siguiente secuencia de iteraciones.

$$a_{i+1} = \frac{1}{n}\left(\frac{A}{a_i^{n-1}} + (n-1)a_i\right)$$

En esta fórmula recursiva, A es el número cuya raíz n-ésima queremos calcular. La primera aproximación, a_1, se calcula de modo que la raíz real se quede justo por debajo de la primera aproximación, asegurando así obtener la raíz real [gmp].

Esta secuencia de iteraciones se sabe que converge de manera cuadrática.

Finalmente, usando esta idea del *Método de Newton*, lo que se hace es calcular sucesivas raíces donde n es primo, y luego elevar para comprobar si coinciden los resultados. Esto nos proporciona una complejidad de $O(\log^3(n))$ [BS89].

B.2. Transformada Rápida de Fourier

Existen algoritmos elementales para multiplicar tanto enteros como polinomios. Estos algoritmos, aunque sencillos y fáciles de enseñar, tienen una complejidad algorítmica elevada. En el caso de la multiplicación de enteros, dicho método elemental tiene complejidad $O(M(n)) = O(n^2)$, donde n es la cantidad de bits que ocupa el número en cuestión. Del mismo modo, la multiplicación polinómica elemental tiene complejidad $O(P(n,m)) = O(n^2M(m))$, donde n es el grado del polinomio y m es el número de bits de sus coeficientes.

Estas operaciones pueden ser aceleradas haciendo uso de algoritmos más sofisticados. Uno de los más usados en este ámbito es el algoritmo de la *Transformada Rápida de Fourier* (*FFT* por las siglas del término en inglés, *Fast Fourier Transform*).

Para explicar la *Transformada Rápida de Fourier*, primero tenemos que entender qué es una *Transformada Discreta de Fourier* (*DFT* por las siglas del término en inglés, *Discrete Fourier Transform*). Damos entonces la siguiente definición.

Definición B.1. Sean $x_0, \ldots, x_{N-1} \in \mathbb{C}$. Se define la *Transformada Discreta de Fourier* de la siguiente forma:

$$X_k = \sum_{n=0}^{N-1} x_n e^{-i2\pi kn/N}, \quad \forall k \in \{0, \ldots, N-1\}$$

De la definición podemos ver la importancia de las raíces de la unidad. En este caso, $e^{i2\pi/N}$ es la N-ésima raíz primitiva de la unidad.

Si observamos, usando la definición tal cual está presentada, calcular una *Transformada Discreta de Fourier* requiere de realizar un total de $O(\log^2(N))$ operaciones. Esto supone que es necesario acelerar dicho proceso para poder mejorar dicha eficiencia. Dichas optimizaciones son más conocidas como *Transformadas Rápidas de Fourier*, y la implementación más usada es el algoritmo de *Cooley-Tukey* [CT69]. La complejidad de este algoritmo es $O^{\sim}(\log(N))$, y es la técnica usada para implementar multiplicación de números y polinomios de manera eficiente.

B.3. Generador Mersenne Twister

A la hora de diseñar tests probabilísticos, estos necesitan de una manera de generar números aleatorios. Puesto que la generación de números aleatorios puros es difícil por el determinismo de las máquinas, necesitamos una manera de generar números que al menos parezcan aleatorios. Es lo que se conoce como los números *pseudo-aleatorios*.

La manera de generar estos números no solo nos proporciona números que aparentemente son aleatorios, sino que además nos permite repetir experimentos de manera que los números generados aleatoriamente sean los mismo.

Normalmente, los generadores requieren de una posición inicial, conseguida mediante lo que se conoce como una semilla. Usar la misma semilla en distintas ejecuciones implica generar exactamente los mismos números en el mismo orden.

Existen mucho generadores de números *pseudo-aleatorios*, cada uno con sus ventajas y sus inconvenientes. Uno de los más usados, y el que hemos usado en este trabajo es el *Mersenne Twister* [MN98].

La implementación más extendida suele ser la basada en el número primo $2^{19937} - 1$, también conocida como *MT19937*, que usa semillas de longitud 32 bits. Entre algunas de sus ventajas, podemos destacar que tiene un periodo suficientemente largo. En específico, los valores se repiten tras $2^{19937} - 1$ iteraciones, lo cual lo hace muy conveniente. Además, la distribución de los números pasa algunos test de aleatoriedad. Además es considerablemente rápido.

Entre sus desventajas se encuentran que el estado del generador ocupa más memoria que otros generadores.

B.4. Otros Tests de Primalidad

En esta sección vamos a explicar brevemente algunos tests de primalidad utilizados en la actualidad, pero que no han sido usados en este trabajo.

B.4.1. Test de Lucas

El test probabilístico de *Lucas* es distinto respecto a los otros tests probabilísticos estudiados en el trabajo: *Miller-Rabin* y *Solovay-Strassen*. Mientras que estos últimos determinan si un número es probablemente primo, el test de *Lucas* funciona al revés, es decir, determina si un número es probablemente compuesto.

Si el test de *Lucas* determina que la entrada es un primo, entonces el número es definitivamente primo junto con su correspondiente certificado de primalidad. Sin embargo solo puede detectar que la entrada es definitivamente compuesta en determinados casos, mientras que en los demás solo determinará que es probablemente compuesto.

La idea del test está en que si existe un $a \in \{2, \dots, n-1\}$ donde n es el número cuya primalidad queremos comprobar, de modo que se cumple

$$a^{n-1} \equiv 1 \mod (n),$$

y para cada factor primo q de $n-1$ se tiene

$$a^{(n-1)/q} \not\equiv 1 \mod (n),$$

entonces n es primo. Si tal a no existe, entonces n es 1, 2 ó es compuesto. Como podemos comprobar, este test requiere conocer los factores primos de $n-1$, lo cual es otro problema distinto a tratar. Veamos un ejemplo.

Ejemplo B.1. Sea $n = 71$, luego $n - 1 = 70$ y cojamos aleatoriamente $a = 17$. Teniendo en cuenta que los factores primos de 70 son 2, 5 y 7. tenemos lo siguiente:

$$\begin{aligned} 17^{70} &\equiv 1 \mod (71) \\ 17^{35} \equiv 70 \mod (71) &\not\equiv 1 \mod (71) \\ 17^{14} \equiv 25 \mod (71) &\not\equiv 1 \mod (71) \\ 17^{10} &\equiv 1 \mod (71) \end{aligned}$$

Como $17^{10} \equiv 1 \mod (71)$, no podemos decir nada sobre la primalidad de 71. Sea ahora $a = 11$. Entonces nos queda:

$$\begin{aligned} 11^{70} &\equiv 1 \mod (71) \\ 11^{35} \equiv 70 \mod (71) &\not\equiv 1 \mod (71) \\ 11^{14} \equiv 54 \mod (71) &\not\equiv 1 \mod (71) \\ 11^{10} \equiv 32 \mod (71) &\not\equiv 1 \mod (71) \end{aligned}$$

Tenemos entonces que 71 es primo y su certificado de primalidad es $a = 11$.

Una versión probabilística de este test es probar distintos valores de a, y si encontramos alguno para el que n no pasa el test anterior, entonces n es primo. si por el contrario pasa el test para todas las rondas, diremos que es probablemente compuesto.

Este test es usado en la generación de números primos, junto con otros tests como el de *Miller-Rabin* [dig13].

B.4.2. Test de Baillie-PSW

El test probabilístico de *Billie-PSW* consiste en una combinación del test de *Miller-Rabin* y de los primos probables fuertes de *Lucas*.

La principal idea está en que el conjunto de los números compuestos que pasan el test de *Miller-Rabin* para la base 2 y el conjunto de los primos probables fuertes de *Lucas* no tienen elementos comunes. Hasta el momento no se ha encontrado un número compuesto que pertenezca a ambos conjuntos.

Esto significa que un número que pase ambos tests será primo con una alta probabilidad.

Este test es el que se implementa en la librería **GMP**.

C. Código Fuente

En este apéndice se van a explicar algunas cosas del código fuente que quedan fuera del trabajo principal. El código fuente se puede encontrar en [Sal].

C.1. Generación de Números Primos

En la última sección del trabajo nos dedicamos a comparar el test **AKS** con los test probabilísticos de *Miller-Rabin* y de *Solovau-Strassen*.

Para dichas comparaciones era necesario generar distintos números primos que nos ayudaran a presentar resultados con distintos conjuntos y combinaciones. Para generar los números primos, se ha considerado que la mejor manera de hacerlo es usando la función que proporciona **GMP** llamada *mpz_probab_prime_p*. Esta función devuelve 2 si el número es definitivamente primo, 1 si es probablemente primo y 0 si es definitivamente compuesto.

El conjunto de números primos generado son los mayores primos que ocupan k bits. Por ejemplo, 13 es el mayor primo que ocupa 4 bits. Se han generado primos hasta los 512 bits. El código para generarlos es el siguiente:

```
auto biggestNBitsPrime(std::size_t bits) -> mpz_class {
        assert(bits >= 2);

        auto n = 2_mpz;
        mpz_pow_ui(n.get_mpz_t(), n.get_mpz_t(), bits);
        --n;

        while (true) {
                if (mpz_probab_prime_p(n.get_mpz_t(), 50) > 0) {
                        return n;
                }
                n -= 2;
        }
}

auto generateTestPrimes(std::size_t n) -> std::vector<mpz_class> {
        auto primes = std::vector<mpz_class>(n);

        std::ranges::generate(primes, [bits = std::size_t(1)]() mutable {
                return biggestNBitsPrime(++bits);
        });

        return primes;
}
```

La función *biggestNBitsPrime* recibe la cantidad de bits y devuelve el mayor primo que ocupa esa cantidad de bits.

La función *generateTestPrimes* recibe hasta cuántos bits tiene que generar primos, y va generando el mayor primo para cada cantidad de bits hasta llegar a la indicada.

En el *main* se llama tal que así:

```
const auto primes = generateTestPrimes(512);
```

Tenemos así un vector con los primos descritos hasta 512 bits. Luego se combinan convenientemente según la comparación que se esté haciendo.

C.2. Generación de Gráficas

Las gráficas generadas en este trabajo son la combinación de dos herramientas.

Por un lado generar los datos, lo cual se hace en C++ y se encuentra en el archivo **examples/src/Benchmarks.cpp**. Dicho archivo fuente usa la librería desarrollada para generar los tiempos.

Una vez generados los tiempos de ejecución, los datos se almacenan en distintos archivos **.txt**. Estos archivos luego se usan en un script de *Gnuplot*, el cual se encarga de generar las imágenes que contienen las gráficas, y que han sido incluidas en este trabajo.

El comando más importante es el que nos permite pintar las gráficas. Dicho comando es *plot*, al cual le podemos pasar distintos archivos de datos o funciones juntos con distintos parámetros (como el estilo de las gráficas y el título), y las guarda en un archivo **.png**.

Para adaptar las eficiencias teóricas a los datos que hemos generado se ha usado el comando *fit*, el cual se encarga de adaptar los coeficientes para que sean más fieles a los datos.

Glosario

$\mathbb{N}$ Conjunto de números naturales.

$\mathbb{Z}$ Conjunto de números enteros.

$\mathbb{R}$ Conjunto de números reales.

$\mathbb{C}$ Conjunto de números complejos.

$|C|$ Cantidad de elementos distintos en el conjunto C o su cardinal.

$or(a)$ Orden del elemento a de un grupo G, es decir, el menor k tal que $a^k = 1$.

$ord_n(a)$ Orden de a módulo n, es decir, el menor k tal que $a^k \equiv 1 \mod (n)$.

Bibliografía

Las referencias se listan por orden alfabético. Aquellas referencias con más de un autor están ordenadas de acuerdo con el primer autor.

[Bac90] Eric Bach. Explicit bounds for primality testing and related problems. *Mathematics of Computation*, 55(191):355–380, 1990. [Citado en pág. 27]

[bja20] Bjarne stroustrup. https://es.wikipedia.org/wiki/Bjarne_Stroustrup, Jul 2020. [Citado en pág. 53]

[BS89] Eric Bach and Jonathan Sorenson. *Sieve algorithms for perfect power testing*. University of Wisconsin-Madison, Computer Science Dept., 1989. [Citado en págs. 48, 59, 60, and 91]

[Coh93] Henri Cohen. Algorithms for algebraic number theory ii. *A Course in Computational Algebraic Number Theory Graduate Texts in Mathematics*, page 297–359, 1993. [Citado en pág. 88]

[CT69] James Cooley and John Tukey. An algorithm for the machine calculation of complex fourier series. *Papers on Digital Signal Processing*, 1969. [Citado en pág. 92]

[dig13] Digital signature standard. 2013. [Citado en págs. 22, 24, 78, and 94]

[Fou] LLVM Foundation. Clang-tidy. https://clang.llvm.org/extra/clang-tidy/. [Citado en pág. 55]

[Fou85] Fouvry. Theoreme de brun-titchmarsh; application au theoreme de fermat. *Inventiones Mathematicae*, 79(2):383–407, 1985. [Citado en pág. 42]

[GG09] Joachim Von Zur Gathen and Jurgen Gerhard. Modern computer algebra. 2009. [Citado en pág. 47]

[gmp] GNU MP 6.2.1. https://gmplib.org/manual/. Recurso online. Accedido el 16 de noviembre de 2021. [Citado en pág. 91]

[JFr] JFrog. Conan: The open source C/C++ package manager for developers. https://conan.io/. [Citado en pág. 54]

[Kit] Kitware. CMake build system. https://cmake.org/. [Citado en pág. 53]

[KS04] Manindra Agrawal, Neeraj Kayal and Nitin Saxina. PRIMES is in P. https://www.cse.iitk.ac.in/users/manindra/algebra/primality_v6.pdf, 2004. [Citado en págs. xix, 30, 41, and 85]

[Mar] Daniel Marjamäki. Cppcheck. https://cppcheck.sourceforge.io/. [Citado en pág. 55]

[Mic16] Microsoft. Visual studio code - code editing. redefined. https://code.visualstudio.com/, Apr 2016. [Citado en pág. 56]

[MN98] Makoto Matsumoto and Takuji Nishimura. Mersenne twister. *ACM Transactions on Modeling and Computer Simulation*, 8(1):3–30, 1998. [Citado en pág. 92]

[Rab80] Michael O Rabin. Probabilistic algorithm for testing primality. *Journal of Number Theory*, 12(1):128–138, 1980. [Citado en pág. 24]

[Sal] Francisco Gallego Salido. Tfg. https://github.com/fgallegosalido/TFG. [Citado en pág. 95]