UNIVERSIDAD DE GRANADA

Facultad de Ciencias

Grado en Matemáticas

TRABAJO DE FIN DE GRADO

Factorización y pruebas de primalidad de enteros

Presentado por:
Adrián González Gutiérrez

Tutor:
Pedro A. García Sánchez
Departamento de Álgebra

Curso académico 2021-2022

Factorización y pruebas de primalidad de enteros

Adrián González Gutiérrez

Adrián González Gutiérrez *Factorización y pruebas de primalidad de enteros*.
Trabajo de fin de Grado. Curso académico 2021-2022.

Responsable de tutorización

Pedro A. García Sánchez
Departamento de Álgebra

Grado en Matemáticas
Facultad de Ciencias
Universidad de Granada

Declaración de originalidad

D. Adrián González Gutiérrez

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 14 de junio de 2022

Fdo: Adrián González Gutiérrez

Índice general

Agradecimientos

En primer lugar, a mi tutor Pedro A. García Sánchez por plantearme este trabajo, de cuya elaboración he disfrutado y por animarme a realizar ciertas partes del mismo que son nuevas para mi y que he me han hecho aprender mucho.

En segundo lugar, a mi familia y amigos sin los que todo esto se hubiera hecho muy cuesta arriba. Por apoyarme siempre a lo largo de todo el grado con las subidas y bajadas que este conlleva.

Por último y en especial quiero agradecer a mi madre todo el esfuerzo que ha hecho durante todo mi educación, siempre intentando ayudar en lo que necesitara. Estoy seguro que ella siempre ha tenido mucho que ver con mi éxito académico.

A todos ellos les agradezco y dedico este trabajo que simboliza la finalización del Grado en Matemáticas.

Summary

Prime numbers are known since ancient times, despite that, they still hide many of their mysteries from us. Since Euclid proved that there were infinitely many of them no one has been able to find the pattern by which they are governed. Moreover, nowadays there are still a large number of unsolved problems in which prime numbers are involved in one way or another, such as the famous Riemann hypothesis.

Over the last few decades, with the advent of telecommunications and the Internet, it has become necessary for our communications and web browsing to be secure in order to protect our privacy. It is here where the different cryptosystems come into play.

The main purpose of this work is to develop the theory needed to be able to describe different factorization methods, primality tests and ultimately to briefly present some applications in cryptography of what has been previously studied. That is why along the way we will introduce some concepts that are of vital importance in number theory. For all this it will be essential what we have studied in the courses Algebra I, II and III as well as in the course Theory of numbers and cryptography from the Mathematics degree.

In addition, what has been learned in Computer science I and II will also be crucial, as we also present a complementary repository to this memory in Github, which can be accessed by clicking here. It contains most of the algorithms described in this document programmed in the Python language. There you can also consult a Jupyter notebook (which we have made interactive using Binder) where we have intended to illustrate the use of the different functions implemented.

In the first chapter, the main methods of integer factorization will be presented, starting with the simplest ones and going on to give a few hints about one of the most efficient methods for this purpose at present. In each of them we will follow the following structure: development of the theory of the method, exposition of its pseudocode and finally an example of its application.

Then, in the second chapter, the most important primality tests, both deterministic and probabilistic, will be introduced. Again, we will start with the basics and end with the most commonly used tests for this purpose today. Here we will follow the same pattern as in the first chapter, development of theory, pseudocode of the test and an example to conclude.

Finally, in the third chapter, as mentioned above, some concepts that are vital for cryptography will be briefly discussed.

Having done all this, we can say that we have achieved the initial objectives of this work. The following texts, [CP05], [Kob94] and [Yan02], have been of great help in the development of this document.

Introducción

Los números primos son conocidos desde la antigüedad, a pesar de esto aún nos esconden muchos de sus misterios. Desde que Euclides probó que existe un número infinito de ellos nadie ha sido capaz de encontrar cual es el patrón por el cual estos se rigen. Además, en la actualidad aún existen un gran número de problemas sin resolver en los cuales intervienen los números primos de una u otra forma, por ejemplo la famosa hipótesis de Riemann.

Desde hace unas décadas, con la llegada de las telecomunicaciones e internet, se ha hecho necesario que nuestras comunicaciones y la navegación web sean seguras con el fin de proteger nuestra privacidad, y es aquí donde entran en juego los distintos criptosistemas.

El propósito de este trabajo es desarrollar la teoría necesaria para poder describir distintos métodos de factorización, pruebas de primalidad, y en última instancia presentar de forma breve algunas aplicaciones en criptografía de lo estudiado anteriormente. Es por eso que en el camino introduciremos algunos conceptos que son de vital importancia en teoría de números. Para todo esto será esencial lo estudiado en las materias Álgebra I, II y III así como en la asignatura Teoría de números y criptografía del Grado en Matemáticas.

Además, también será crucial lo aprendido en Informática I y II, ya que también presentamos un repositorio complementario a esta memoria en Github, al que se puede acceder pulsando aquí. En él se encuentran la mayoría de los algoritmos que se describen en este documento programados en el lenguaje Python. En este también se puede consultar un bloc de Jupyter (el cual hemos hecho interactivo utilizando Binder) donde se ha tratado de ilustrar el uso de las distintas funciones implementadas.

En el primer capítulo se expondrán los principales métodos de factorización de enteros, comenzando por los más simples y llegando a dar algunas pinceladas sobre uno de los métodos más eficientes para este propósito en la actualidad. En cada uno de ellos seguiremos la siguiente estructura: desarrollo de la teoría del método, exposición de su pseudocódigo y un ejemplo de aplicación del mismo.

A continuación, en el segundo capítulo, se introducirán las pruebas de primalidad más importantes, tanto deterministas como probabilísticas. De nuevo empezaremos por lo básico para terminar con las pruebas más utilizadas a día de hoy para este propósito. Aquí seguiremos la misma pauta que en el primer capítulo, desarrollo de teoría, pseudocódigo de la prueba y ejemplo.

Por último, en el tercer capítulo se verán de forma breve, como antes se dijo, algunos conceptos básicos en criptografía.

Habiendo hecho todo esto podemos decir que hemos conseguido los objetivos iniciales propuestos para este trabajo. A lo largo del mismo han resultado de gran ayuda los siguientes textos, [CP05], [Kob94] y [Yan02], siendo los más utilizados en la elaboración de la presente memoria.

1. Métodos de factorización en enteros

En este capítulo se describirán distintos métodos de factorización de enteros, empezando por algoritmos de propósito específico, es decir, en los que se requiere que el entero a factorizar tenga determinadas propiedades para el funcionamiento óptimo de éste. Comenzaremos por la división por tentativa, seguiremos con el método de Fermat y los métodos ρ y $p-1$ de Pollard. Tras esto desarrollaremos algoritmos de propósito general, esto es, su eficiencia solo depende del número de cifras del entero que se esté tratando. En este apartado se expondrán el método de factorización de Dixon y la criba cuadrática para finalizar.

1.1. División por tentativa

Este método consiste en intentar dividir el número entero al que este es aplicado, llamémoslo n, por todos los primos menores que su raíz cuadrada. Como se puede apreciar, este método permite calcular todos los factores primos de n.

Notemos que solo es necesario probar la división por los primos menores o iguales que $\sqrt{n}$, cosa que probamos en el siguiente resultado.

Proposición 1.1. *Sea $n \in \mathbb{N}$. Si n es compuesto, entonces este tiene un factor primo menor o igual que $\sqrt{n}$.*

Demostración. Sabemos por el teorema fundamental de la aritmética que todo $n \in \mathbb{N}$ se descompone de forma única como un producto de primos. Ahora, si suponemos por reducción al absurdo que todos los factores primos de n son mayores que $\sqrt{n}$, entonces como n tiene al menos dos factores primos en su descomposición, se tiene que el producto de estos dos es mayor que n, lo cual es un absurdo que viene de suponer que todos los factores primos son mayores que $\sqrt{n}$. □

Mostramos a continuación el pseudocódigo de este método y tras esto veremos un ejemplo de aplicación del mismo.

Algorithm 1 División por tentativa

Input: $n \in \mathbb{Z}$
Output: L ▷ Lista con los divisores de n
  $j \leftarrow 2$
  $L \leftarrow \{\}$ ▷ Definimos una lista vacía
  for $j \leq \lfloor \sqrt{n} \rfloor$ **do**
    while $n \equiv 0 \pmod{j}$ **do**
      $L \leftarrow L \cup \{j\}$ ▷ Si j divide a n lo añadimos a L
      $n \leftarrow \frac{n}{j}$
    end while
  end for
  return L

Ejemplo 1.1. Aplicaremos este método al número $n = 492$. Empezamos probando con 2 y vemos que es un divisor, y el cociente de la división es 246, probamos otra vez con dos, y obtenemos que sigue siendo divisor, y ahora el cociente de la división es 123. Observamos que 2 ya no vuelve a ser factor, por lo cual pasamos al 3, y vemos que es divisor de 123 y su cociente es 41. Otra vez no podemos seguir con 3, probamos con 5 y tampoco, y ya habríamos terminado, ya que el siguiente primo, 7, es mayor que $\sqrt{41}$. La factorización de 492 es por tanto $492 = 2^2 \cdot 3 \cdot 41$.

Este método es muy sencillo y nos permite hallar la descomposición en factores primos de cualquier número entero. Sin embargo, es muy poco eficiente cuando el número de dígitos empieza a aumentar, como es el caso de los números utilizados en criptografía, que tienen factores primos muy grandes. En este caso este algoritmo es computacionalmente impracticable.

1.2. Método de Fermat

Consideramos $n \in \mathbb{Z}$ impar (si no fuera así dividimos por 2 hasta que resulte un entero impar). Si $n = pq$, con $p \leq q$ ambos impares, entonces considerando:

$$x = \frac{1}{2}(p+q),$$
$$y = \frac{1}{2}(q-p).$$

Observamos que $n = (x+y)(x-y) = x^2 - y^2$, o lo que es lo mismo, $x^2 - n = y^2$, y además se cumple que:

$$q = x + y,$$
$$p = x - y.$$

Por tanto, si queremos una factorización de n de la forma $n = pq$, esto es lo mismo que buscar una factorización del tipo $n = pq = (x+y)(x-y) = x^2 - y^2$, expresando n como diferencia de cuadrados.

El algoritmo consiste en encontrar un valor de x tal que $x^2 - n$ sea un cuadrado, así tomando $p = x - y$ y $q = x + y$ tendríamos, por lo discutido antes, que $n = pq$. Para encontrar valores de x que satisfagan la condición requerida se comienza probando con $x = \lceil \sqrt{n} \rceil$, donde $\lceil\ \rceil$ es la función techo. Si este no la cumple seguimos con $x = \lceil \sqrt{n} \rceil + 1$, y así sucesivamente hasta que $x^2 - n$ sea un cuadrado, entonces hacemos $y = \sqrt{x^2 - n}$ y obtenemos la factorización de n como $n = (x+y)(x-y)$.

Aquí ha sido necesario calcular $\lceil \sqrt{n} \rceil$. Para su cálculo de este valor podemos proceder de distintas formas, describimos ahora una de ellas utilizando el conocido método de Newton-Raphson. Lo hacemos en la siguiente página realizando su pseudocódigo (Algoritmo 2).

Notemos que este mismo método lo podemos utilizar para calcular $\lfloor \sqrt{n} \rfloor$, sustituyendo los $\lceil\ \rceil$ por $\lfloor\ \rfloor$ en dicho algoritmo, por lo que también puede ser aplicable en el caso de la sección sección 1.1 donde precisábamos de este valor.

Se expone también ahora el pseudocódigo del método de Fermat, (Algoritmo 3), donde se requiere como entrada un entero impar y una cota para el número máximo de iteraciones del método, asegurándonos así de que este pare en algún momento, y nos devuelve dos factores de dicho entero o fallo si se ha alcanzado la cota. Utilizamos una función llamada NRtecho(n), que calcula $\lceil \sqrt{n} \rceil$ utilizando el método que justo acabamos de describir y cuyo pseudocódigo también se puede ver a continuación.

Algorithm 2 NRtecho

Input: $n \in \mathbb{Z}$
Output: $\lceil \sqrt{n} \rceil$
  $x \leftarrow n$
  $y \leftarrow \lceil \frac{x+1}{2} \rceil$
  while $y < x$ **do**
    $x \leftarrow y$
    $y \leftarrow \lceil \frac{x+n/x}{2} \rceil$
  end while
  return x

Algorithm 3 Método de Fermat

Input: $n \in \mathbb{Z}$ impar, $M \in \mathbb{N}$ ▷ M es la cota antes mencionada
Output: $\{p, q\}$ ▷ $n = pq$
  $x \leftarrow NRtecho(n)$
  $j \leftarrow 1$
  while $x^2 - n$ no sea un cuadrado y $j \leq M$ **do**
    $x \leftarrow x + 1$ ▷ Si $x^2 - n$ no es cuadrado probamos con x+1
    $j \leftarrow j + 1$
  end while
  if $x^2 - n$ es un cuadrado **then**
    $y \leftarrow \sqrt{x^2 - n}$
    return $\{x + y, x - y\}$
  else
    return Fail
  end if

Ambos algoritmos se pueden ver programados en el lenguaje de programación Python en el repositorio particular del TFG. El Algoritmo 2 puede encontrarse en forma de función el fichero *algoritmos_complementarios*.

En el ejemplo que sigue se ha utilizado el código implementado para agilizar los cálculos necesarios.

Ejemplo 1.2. En la siguiente tabla se muestra el proceso de factorización mediante el método de Fermat del entero $n = 6721$.

i	x	$x^2 - n$
1	82	3
2	83	168
3	84	335
4	85	504
5	86	675
6	87	848
7	88	1023
8	89	1200
9	90	1379
10	91	1560
11	92	1743
12	93	1928
13	94	2115
14	95	2304

Cuando $x = 95$, después de 14 iteraciones, el algoritmo para, ya que $2304 = 48^2$, así que la factorización buscada es $6721 = (95 + 48) \cdot (95 - 48) = 143 \cdot 47$.

Para terminar con este método es importante observar que para un entero n cuyos factores están próximos entre sí y cercanos a la raíz cuadrada de n este método es muy rápido, sin embargo no es así si están medianamente lejos uno del otro, cuando los números se hacen muy grandes y sus factores están lejos de su raíz, este método puede ser realmente lento.

1.3. Método ρ de Pollard

Este método fue introducido por J. Pollard en 1975 y resulta muy rápido cuando se trata de factorizar enteros con factores pequeños.

Supongamos que $n \in \mathbb{Z}$ es compuesto, ahora consideramos una aplicación $f : \mathbb{Z}_n \longrightarrow \mathbb{Z}_n$ que sea fácil de evaluar, se suele trabajar con polinomios de la forma $x^2 + c$ que son las que más probabilidades de éxito tienen y un $x_0 \in \mathbb{Z}_n$ e iterar de la siguiente forma:

$$x_{j+1} = f(x_j).$$

Como esta aplicación está definida en un conjunto finito (de n elementos), entonces sabemos que antes de n iteraciones la sucesión $\{x_j\} \subseteq \mathbb{Z}_n$ debe entrar en un ciclo. La sucesión aparece como un círculo del que pende un tallo teniendo la apariencia de la letra griega ρ, por lo que el nombre del método está totalmente justificado. Cuando n es muy grande cabe pensar que este ciclo puede estar muy lejos del inicio de la sucesión, pero es aquí donde entra en juego la paradoja del cumpleaños la cual se puede consultar en [Kono7, pág 395], y que afirma que probablemente no son necesarias tantas iteraciones como en un inicio pudiéramos pensar para que encontremos una repetición en nuestra sucesión. Pues podríamos hacer lo siguiente, para cada término x_k, $k > 0$ de la sucesión, computar $\gcd(x_k - x_j, n) = d$, para todo $j < k$ y si este número es distinto de 1 entonces d es divisor de n, si no es así el método hace lo propio con x_{k+1} y así sucesivamente.

Éste proceso puede hacerse realmente largo cuando k aumenta lo suficiente. Es por eso que describimos ahora un método para el que solo se necesita calcular un máximo común divisor por cada k, el cual viene justificado por el resultado que sigue y que ha sido extraído de [Kob94].

Proposición 1.2. *Sean $x_{k_0} \equiv x_{j_0} \pmod{p}$, siendo p un divisor no trivial de n. Entonces se cumple la misma relación para k, j tales que $k - j = k_0 - j_0$, es decir:*

$$x_k \equiv x_j \pmod{p}.$$

Demostración. Sea $k = k_0 + m = j_0 + m$, entonces aplicando m veces el polinomio f a ambos lados de la congruencia $x_{k_0} \equiv x_{j_0} \pmod{p}$ que estemos considerando, se tiene que $x_k \equiv x_j \pmod{p}$ como queríamos. □

Ahora sí, describimos como funciona el método ρ de Pollard. Este utiliza el algoritmo de Floyd para la detección de ciclos (también llamado algoritmo de la tortuga y la liebre), que se puede consultar en [Nes12], y según este para cada paso k solo debemos calcular $\gcd(x_k - x_{2k}, n)$.

Mostramos a continuación el pseudocódigo de este algoritmo, donde se requieren como entrada el entero a factorizar, un valor inicial x_0 con el que se iniciará la sucesión $\{x_j\}$, la función f que construirá dicha sucesión y un número natural que será una cota para asegurarnos de que el algoritmo termina. Si se alcanza esta cota el programa dará fallo y no habrá sido posible factorizar el número.

Algorithm 4 Método ρ de Pollard

Input: $n \in \mathbb{Z}$, x_0, f, $M \in \mathbb{N}$ ▷ f es nuestra aplicación de $\mathbb{Z}_n$ en $\mathbb{Z}_n$
Output: $\{p, q\}$ o fail ▷ p, q son factores no triviales de n

```
a ← x_0
b ← x_0
i ← 1
d ← 1
while i < M y d = 1 do
    a ← f(a)
    b ← f(f(b))
    d ← gcd(b − a, n)
    i ← i + 1
end while
if d ≠ 1 y d ≠ n then                ▷ Si salimos del bucle, o bien d ≠ 1 o bien i = M
    return {d, n/d}
else
    return Fail
end if
```

Si el algoritmo retorna fallo, habría que considerar cambiar la aplicación f, el valor inicial x_0 o incluso los dos. Si sigue sin dar resultado tras un número considerable de iteraciones, sería bueno plantearse el uso de otros métodos.

Ejemplo 1.3. Presentamos ahora un ejemplo de uso del método ρ de Pollard. Vamos a factorizar el entero $n = 12247$ usando el polinomio $f(x) = x^2 - 1$ y eligiendo $x_0 = 2$. Exponemos los resultados en forma de tabla.

i	x_k	x_{2k}	$\gcd(x_k - x_{2k}, n)$
1	3	8	1
2	8	3968	1
3	63	886	1
4	3968	563	1
5	7628	7631	1
6	886	4697	37

Por tanto en 6 iteraciones hemos obtenido que 37 es un factor de n, así que $n = 12247 = 37 \cdot 331$.

Pasamos ahora a describir una versión mejorada de este método, en el que en lugar de utilizar el algoritmo de detección de ciclos de Floyd utiliza el de Brent que se puede consultar en [Nes12]. El funcionamiento de éste aplicado al método ρ de Pollard es el que sigue, a la vez que vamos calculando cada x_k procedemos de la siguiente manera. Si k es un entero de (h+1) bits, o lo que es lo mismo, $2^h \leq k < 2^{h+1}$, tomamos el menor entero de h+1 bits, $j = 2^h$, y computamos $\gcd(x_k - x_j, n)$. Si el resultado de esta operación es un entero distinto de 1, el algoritmo para. Si no es así se hace lo mismo para $k + 1$ y así sucesivamente.

Este método perfeccionado de Brent es aproximadamente un 24 % mas rápido que el método original. Exponemos a continuación su pseudocódigo, muy similar al anterior, con la misma entrada pero con la única diferencia de que ahora utilizamos el algoritmo de detección de ciclos de Brent.

Algorithm 5 Método ρ de Pollard (versión mejorada de Brent)

Input: $n \in \mathbb{Z}$, x_0, f, $M \in \mathbb{N}$
Output: $\{p, q\}$ o fail

```
a ← x_0
b ← x_0
i ← 1
k ← 2
d ← 1
while i < M y d = 1 do
    if i > k then                          ▷ Pasamos de la iteración 2^j
        k ← 2k                             ▷ Actualizamos k a 2^{j+1}
        b ← a                              ▷ Asignamos b = x_{2^j}
    end if
    a ← f(a)
    d ← gcd(a − b, n)
    i ← i + 1
end while
if d ≠ 1 y d ≠ n then
    return {d, n/d}
else
    return Fail
end if
```

Ejemplo 1.4. Procedemos a factorizar el entero $n = 25769$ usando el método de Brent tomando el polinomio $f = x^2 - 1 \pmod{n}$ y tomando $x_0 = 2$. Mostramos los pasos seguidos por el algoritmo en la siguiente tabla.

i	x_j	x_k	$\gcd(x_k - x_j, n)$
1	2	3	1
2	2	8	1
3	8	63	1
4	8	3968	1
5	3968	164	1
6	3968	1126	1
7	3968	5194	1
8	3968	23261	1
9	23261	2427	1
10	23261	14996	1
11	23261	19721	1
12	23261	12092	73

Por tanto, en 12 iteraciones hemos obtenido que 73 es divisor de nuestro entero, entonces $n = 25769 = 73 \cdot 353$.

Aquí también se ha utilizado el material del repositorio para resolver lo propuesto en el ejemplo.

El método ρ de Pollard es una mejora de la división por tentativa en cuanto a eficiencia computacional se refiere, ya que se necesitan menos operaciones para hallar factores pequeños del numero tratado. Sin embargo tiene el inconveniente de que su tiempo de ejecución es solo un valor esperado, basado en probabilidad, en lugar de un tiempo definido de forma más rigurosa.

1.4. Método $p-1$ de Pollard

Este método fue ideado también por J. Pollard en 1974 y se basa fundamentalmente en el conocido pequeño Teorema de Fermat. Es por esto que pasamos a enunciarlo y demostrarlo propiamente, aunque este es fácil de encontrar en cualquier libro de teoría de números elemental.

Teorema 1.1. *(Pequeño Teorema de Fermat). Si p es un número primo, entonces para cada entero $a > 1$ verificando que $\gcd(a, p) = 1$, se cumple que $a^{p-1} \equiv 1 \pmod{p}$.*

Demostración. Probamos primero que $a^p \equiv a \pmod{p}$ para todo a positivo. Procedemos por inducción, vemos que el caso $a = 1$ es claro. Suponemos cierto para a y vemos que sucede con $a + 1$:

$$(a+1)^p \equiv a^p + 1 \equiv a + 1 \pmod{p}.$$

Aquí hemos utilizado el binomio de Newton, por el cual sabemos que los demás términos del desarrollo de $(a+1)^p$ son divisibles por p.

Tenemos por tanto que $a^p \equiv a \pmod{p}$, para todo $a \in \mathbb{N}$. Además utilizando nuestra hipótesis $\gcd(a, p) = 1$, podemos multiplicar ambos lados de la congruencia por el inverso de a obteniendo así el resultado deseado. □

El método $p-1$ de Pollard se basa en la siguiente idea. Supongamos que nuestro entero a factorizar es de la forma $n = pq$. Entonces por el pequeño teorema de Fermat (Teorema 1.1), sabemos que $a^{p-1} \equiv 1 \pmod{p}$ para cualquier $a \in \mathbb{Z}$. Consideramos ahora $p-1$, y supongamos que este es un factor de otro entero M, por tanto existe $K \in \mathbb{Z}$ tal que $M = (p-1)K$. Observamos por tanto que $a^M \equiv (a^{p-1})^K \equiv 1 \pmod{p}$, así $p|(a^M - 1)$, y a su vez sabemos que $p|n$, por lo que $\gcd(a^M - 1, n)$, que es un divisor de n, contiene a p en su factorización.

Como bien señala el autor de [Rie12] en la descripción de este método, llegados a este punto podemos formular la siguiente pregunta, ¿no sería más rápida la división por tentativa que ir probando hasta encontrar un entero M tal que $\gcd(a^M - 1, n) \neq 1$?, la respuesta es que sí, pero es aquí donde entra en juego la idea de Pollard. La idea es que $a^M - 1$ tenga muchos factores primos para así aumentar las posibilidades de que el máximo común divisor de éste y n sea distinto de 1. Supongamos que $p-1$ solo contiene en su factorización primos pequeños, entonces si calculamos $\gcd(a^M - 1, n)$ podríamos ser capaces de encontrar un factor de n relativamente pronto.

Hemos convertido nuestro problema inicial en un problema que consiste en generar enteros que solo contengan en su factorización números pequeños. Esto ya es una tarea más fácil de llevar a cabo, por ejemplo podemos considerar $M_k = k!$ que contiene a todos los primos menores o iguales que k. El método $p-1$ de Pollard consistiría en ir probando con estos M_k e ir comprobando si $\gcd(a^{M_k} - 1, n) \neq 1$.

Exponemos ahora el pseudocódigo del método, en el que como entrada requerimos el entero a factorizar y como siempre una cota que llamaremos H para asegurarnos de que el algoritmo finaliza en algún momento. El número a con el que hemos hecho la discusión suele tomarse como $a = 2$ y así lo haremos en el algoritmo siguiente.

Algorithm 6 Método $p-1$ de Pollard

Input: $n \in \mathbb{Z}$, $H \in \mathbb{N}$
Output: $\{p, q\}$ o fail

```
a ← 2
M ← 1
d ← 1
while M < H y d = 1 do
    a ← a^M (mod n)
    M ← M + 1
    d ← gcd(a − 1, n)
end while
if d ≠ 1 y d ≠ n then
    return {d, n/d}
else
    return Fail
end if
```

A continuación mostramos un ejemplo de funcionamiento del algoritmo presentado. Se ha utilizado el código que se presenta en el repositorio para realizar los cálculos necesarios en el proceso.

Ejemplo 1.5. Realizamos la factorización del número $n = 17278$ aplicando el Algoritmo 6. Mostramos los resultados obtenidos en cada iteración en la siguiente tabla:

i	$a^M \pmod n$	$\gcd(a^M - 1, n)$
1	2	1
2	4	1
3	64	1
4	278	1
5	240	1
6	2856	1
7	11776	1
8	9670	1
9	490	163

Obtenemos así, en la iteración 9, que 163 es un factor de nuestro entero y factoriza como $n = 163 \cdot 106$.

Este algoritmo puede ser fácilmente modificado eligiendo de una manera distinta los M_k antes mencionados. Ahora los vamos a definir como $M_k = \text{lcm}(1, ..., k)$, que también contienen en su factorización a todos los primos menores o iguales que k. Además son enteros más pequeños que los definidos anteriormente por lo que supondrán menos coste computacional.

Damos ahora una forma eficiente, extraída de [CP05], para calcular estos M_k para todo $k \in \mathbb{N}$. Sabemos que $M_1 = 1$, $M_2 = 2$, $M_3 = 6$, etc. Esta secuencia podemos construirla recursivamente de la siguiente forma. Supongamos que ya conocemos M_k para cierto k, entonces $M_{k+1} = M_k$ si $k+1$ no es primo, y cuando $k+1 = p^r$ con p primo se puede comprobar que $M_{k+1} = pM_k$.

Aquí se presenta el pseudocódigo de esta variación del método $p-1$ de Pollard. Las entradas requeridas son las mismas que para la versión original.

Algorithm 7 Método $p-1$ de Pollard (variación)

Input: $n \in \mathbb{Z}$, $H \in \mathbb{N}$
Output: $\{p, q\}$ o fail

```
a ← 2
M ← 1
d ← 1
while M < H y d = 1 do
    if M = p^r y p es primo then          ▷ Aplicamos el método de cálculo antes visto
        a ← a^p (mod n)
        d ← gcd(a − 1, n)
    end if
    M ← M + 1
end while
if d ≠ 1 y d ≠ n then
    return {d, n/d}
else
    return Fail
end if
```

Ejemplo 1.6. Factorizamos el mismo número del ejemplo anterior, $n = 12931$ aplicando esta vez el Algortimo 7. Mostramos los resultados obtenidos en cada iteración en la siguiente tabla:

i	$a^M \pmod n$	$\gcd(a^M - 1, n)$
1	2	1
2	4	1
3	64	1
4	4096	1
5	3238	1
6	3238	1
7	7518	1
8	11854	1
9	7446	1
10	7446	1
11	2011	67

Obteniendo así que 67 es un divisor de 12931, por tanto $n = 67 \cdot 193$, como ya habíamos visto en el ejemplo anterior, con la diferencia de que ahora el factor que nos ha dado el método es 67 y no 193 como aplicando el algoritmo original. Vemos que en este caso hemos encontrado un factor no trivial en 10 iteraciones, puede parecer por tanto que este algoritmo es más lento que el anterior. Sin embargo, no es así ya que en algunas iteraciones como la 6 o la 10 no hemos realizado cálculo alguno.

Terminamos aquí con el método $p - 1$ de Pollard y por tanto con los algoritmos de propósito específico. Nos centramos a partir de ahora en el estudio de los algoritmos de propósito general.

1.5. Método de factorización de Dixon

Fue introducido por John D. Dixon en 1981. Se trata de un algoritmo probabilístico, y asintóticamente es mucho mas rápido que cualquier otro algoritmo hasta la fecha de su publicación. La demostración de este hecho se puede ver en [Dix81], el artículo original donde se publica el método.

Este método y muchos otros se fundamentan en un método general por primera vez propuesto en el siglo XX por el matemático Maurice Kraitchik. Recogemos en el siguiente resultado una idea fundamental para su desarrollo.

Proposición 1.3. *Sea $n \in \mathbb{N}$. Si existen $x, y \in \mathbb{Z}$ cumpliendo que:*

a) $x \not\equiv \pm y \pmod n$,

b) $x^2 \equiv y^2 \pmod n$.

Entonces $d = \gcd(x \pm y, n)$ es un divisor no trivial de n.

Demostración. Por b) sabemos que $n \mid x^2 - y^2 = (x + y)(x - y)$ y si usamos a) obtenemos que:

$$n \nmid (x + y) \quad \text{y} \quad n \nmid (x - y).$$

Por otra parte, veamos que $d \neq 1$. Procedemos por reducción al absurdo. Si $d = \gcd(x - y, n) = 1$, en la congruencia $(x-y)(x+y) \equiv 0 \pmod{n}$ podemos multiplicar por el inverso módulo n de $x - y$ obteniendo así $x + y \equiv 0 \pmod{n}$, lo cual es imposible en vista de a).

Podemos asegurar entonces que $d = \gcd(x \pm y, n)$ es un divisor no trivial de n. □

Con la Proposición 1.3 en mente, el problema de factorización se reduce a encontrar $x, y \in \mathbb{Z}$ que satisfagan a) y b). Exponemos ahora el método siguiendo los pasos descritos en [Kono7].

Para encontrar estos pares de números podemos proceder de manera aleatoria, es decir, eligiendo al azar $s \in \mathbb{Z}_n$ y viendo que $u \equiv s^2 \pmod{n}$, con u un cuadrado perfecto. Si esto ocurre y además se cumple que $\sqrt{u} \not\equiv \pm s \pmod{n}$ tendríamos por la Proposición 1.3 un factor no trivial como queríamos. Si esto no ocurre, seguiríamos probando con distintos $s \in \mathbb{Z}_n$ hasta que así sea. Esto sería muy similar al método de Fermat desarrollado en la sección 1.2.

Podemos hacer lo anterior, pero este procedimiento puede hacerse sumamente largo. O bien podemos ingeniárnoslas de alguna forma para evitar esto y conseguir los pares (x, y) deseados de otra forma.

Elegimos de manera aleatoria un conjunto de, digamos, k enteros $\{s_i\} \subseteq \mathbb{Z}_n$ y calculamos sus cuadrados módulo n:

$$u_i \equiv s_i^2 \pmod{n}, \qquad 1 \leq i \leq k.$$

Describimos ahora el proceso general, desarrollado por Kraitchik, que se utiliza en todos los métodos de este tipo. La estrategia consiste en combinar de manera apropiada algunos de estos u_i para que al multiplicarlos obtengamos un cuadrado perfecto. O lo que es lo mismo, queremos que los primos que aparezcan en la descomposición de este producto tengan exponente par. Este proceso de combinación lo podemos hacer como sigue:

1) Sean $p_1 < p_2 < \ldots < p_m$ los primos que aparecen en la factorización de los u_i. Realizamos la factorización de $u_i \equiv s_i^2 \pmod{n} = \prod_{j=1}^{m} p_j^{e_{i,j}}$.

2) Formamos la matriz de dimensiones $k \times m$ con los exponentes resultantes de cada factorización, es decir:

$$\Psi = \begin{pmatrix} e_{1,1} & e_{1,2} & \cdots & e_{1,m} \\ e_{2,1} & e_{2,2} & \cdots & e_{2,m} \\ \vdots & \vdots & \ddots & \vdots \\ e_{k,1} & e_{k,2} & \cdots & e_{k,m} \end{pmatrix}.$$

Y ahora se trata de encontrar un vector $(x_1, x_2, ..., x_k) \in \mathbb{Z}_2^k$, que cumpla la siguiente condición:

$$\underbrace{(0, 0, 0, \ldots, 0)}_{longitud\ m} \equiv (x_1, x_2, \ldots, x_k)\Psi \pmod{2}.$$

Y esto es un problema sencillo de álgebra lineal que podemos resolver, por ejemplo, por eliminación gaussiana.

Tras todo este proceso podemos formar el siguiente par de números:

$$x^2 = \prod_{i=1}^{k} u_i^{x_i} = \prod_{j=1}^{m} p_j^{(\sum_{i=1}^{k} x_i e_{ij})}, \quad y^2 = \prod_{i=1}^{k} s_i^{2x_i}.$$

Obtenemos finalmente lo deseado, es decir, la congruencia $x^2 \equiv y^2 \pmod{n}$. Si se tiene que $x \not\equiv \pm y \pmod{n}$ el algoritmo ha resultado exitoso. En cambio, si no es así habría que cambiar el conjunto aleatorio inicial $\{s_i\}$ o buscar otro $(x_1, x_2, \ldots, x_k)$ solución de nuestro sistema.

Si analizamos el proceso anterior, vemos que es indispensable que $k > m$ para asegurar que existe una solución no trivial al sistema anterior. En la práctica para asegurar esto se hace lo siguiente:

1) Elegimos lo que se llama una base de factores, que no es mas que un conjunto formado por los m primeros primos, lo denotamos $P_m = \{p_1, p_2, ..., p_m\}$.

2) Cuando un entero se puede factorizar utilizando solo primos menores o iguales que cierto número B se dice que es un entero B-liso. Tratamos por tanto de encontrar al menos $m + 1$ enteros del conjunto $\{1, \ldots, n\}$ que cumplan que su cuadrado módulo n es p_m-liso, es decir, se puede factorizar utilizando solo los primos que aparecen en nuestra base de factores.

Como hemos visto el método tiene dos partes bien diferenciadas. Una primera que se basa en la búsqueda de los conjuntos $\{s_i\}$ y $\{u_i\}$. Y una segunda donde se trata con un problema de álgebra lineal, como bien decíamos antes, la cual termina con la consecución del par (x, y) deseado que nos dará la factorización de n.

Presentamos en la siguiente página el pseudocódigo de este método (Algoritmo 8). Para la inicialización del mismo se necesitarán como entrada el número a factorizar, un natural que llamaremos M que será la cota para la base de factores y otro parámetro que llamaremos N el cual nos marca el número de elementos aleatorios con los que trabajaremos.

Como bien dijimos antes y se puede observar en el Algoritmo 8, la búsqueda de los enteros que cumplan la condición de ser lisos para nuestra base de factores se realiza de manera aleatoria, es por esto que este método también se puede encontrar en muchos textos con el nombre de factorización aleatoria de Dixon. Al ser aleatorio, la eficiencia y éxito de este algoritmo residen en la elección de los parámetros N y M.

Queremos por tanto elegir N y M de tal forma que el algoritmo resulte exitoso para casi todas las listas L posibles y el número de operaciones requeridas para este propósito sea el menor posible. La elección que realiza el autor de este método es la siguiente:

$$M = \exp((2 \ln n \ln \ln n)^{1/2}), \quad N = \lfloor M^2 + 1 \rfloor.$$

Se puede demostrar que esta elección cumple las condiciones antes mencionadas. El lector interesado en obtener más detalles a cerca de esto puede consultar [Dix81], donde también se encuentra la demostración dada por Dixon.

Tras esto estamos en condiciones de ver una aplicación práctica del método presentado. En el siguiente ejemplo se presentarán los resultados obtenidos al ejecutar el código programado en Python que se puede consultar en nuestro repositorio. El programa solo nos ayuda con la primera parte del método comentada anteriormente, la generación de los conjuntos $\{s_i\}$ y $\{u_i\}$. Una vez tengamos esto procederemos a operar con los u_i de la manera antes descrita para terminar el algoritmo de manera exitosa.

Algorithm 8 Método de Dixon

Input: $n \in \mathbb{Z}$, $M \in \mathbb{N}, N \in \mathbb{N}$
Output: $\{p, q\}$ o fail

$L \leftarrow \{\}$
for $k = 1 : N$ **do** ▷ Generamos el conjunto L con N elementos (entre 1 y n)
  $L \leftarrow L \cup random[1, n]$ ▷ Elementos aleatorios
end for
$P \leftarrow \{p_1, ..., p_m\}$ ▷ Nuestra base de factores con $p_i < M$, $1 \leq i \leq m$
$B \leftarrow \{\}$ ▷ Definimos dos conjuntos vacíos
$Z \leftarrow \{\}$
$i \leftarrow 0$
while $i < n$ y $|B| \leq m$ **do** ▷ $|B|$ es el número de elementos de B
  $z_i \leftarrow L(i)$ ▷ $L(i)$ es el elemento i del vector L
  $w_i \leftarrow z_i^2 \pmod{n}$ ▷ El menor residuo positivo módulo n
  if w_i es p_m-liso con $w_i = \prod_{j=1}^{m} p_j^{a_{ij}}$ **then**
    $B \leftarrow B \cup (a_{i1}, ..., a_{im})$ ▷ Almacenamos los exponentes
    $Z \leftarrow Z \cup \{z_i\}$ ▷ Enteros cuyo cuadrado es p_m-liso módulo n
  end if
  $i \leftarrow i + 1$
end while
if $m < |B|$ **then** ▷ Condición para que exista solución no trivial
  Resolvemos por Gauss para x en la ecuación $\vec{0} \equiv xB \pmod 2$
  ▷ Vemos B como matriz y x es un vector de $|B|$ posiciones
  $a \leftarrow \prod_{j=1}^{|B|} Z(j)^{x_j}$ ▷ $Z(j)$ es el elemento j de Z
  $b \leftarrow \prod_{j=1}^{m} p_j^{(\sum_{i=1}^{|B|} x_i a_{ij})/2}$ ▷ Y se cumple que $a^2 \equiv b^2 \pmod n$
  if $a \not\equiv \pm b \pmod n$ **then**
    return $\{\gcd(a + b, n), \gcd(a - b, n)\}$ ▷ Ambos son factores de n
  else
    return Fail
  end if
else
  return Fail
end if

Ejemplo 1.7. En este caso el entero a factorizar es $n = 6213$, la base de factores se limita a los primeros cinco primos y hemos elegido el parámetro $N = 25$.

Aleatorio en Z_n	Descomposición	$11 - liso$
$3087^2 \equiv 5040 \pmod{n}$	$[2,2,2,2,3,3,5,7]$	*True*
$4861^2 \equiv 1282 \pmod{n}$	$[2,641]$	*False*
$1491^2 \equiv 5040 \pmod{n}$	$[2,2,2,2,3,3,5,7]$	*True*
$2037^2 \equiv 5298 \pmod{n}$	$[2,3,883]$	*False*
$4094^2 \equiv 4375 \pmod{n}$	$[5,5,5,5,7]$	*True*
$3556^2 \equiv 1681 \pmod{n}$	$[41,41]$	*False*
$2351^2 \equiv 3844 \pmod{n}$	$[2,2,31,31]$	*False*
$2301^2 \equiv 1125 \pmod{n}$	$[3,3,5,5,5]$	*True*
$3657^2 \equiv 3273 \pmod{n}$	$[3,1091]$	*False*
$917^2 \equiv 2134 \pmod{n}$	$[2,11,97]$	*False*
$5033^2 \equiv 688 \pmod{n}$	$[2,2,2,2,43]$	*False*
$5946^2 \equiv 2946 \pmod{n}$	$[2,3,491]$	*False*
$2310^2 \equiv 5346 \pmod{n}$	$[2,3,3,3,3,3,11]$	*True*
$2087^2 \equiv 256 \pmod{n}$	$[2,2,2,2,2,2,2,2]$	*True*
$1224^2 \equiv 843 \pmod{n}$	$[3,281]$	*False*
$1961^2 \equiv 5887 \pmod{n}$	$[7,29,29]$	*False*
$1464^2 \equiv 6024 \pmod{n}$	$[2,2,2,3,251]$	*False*
$420^2 \equiv 2436 \pmod{n}$	$[2,2,3,7,29]$	*False*
$1727^2 \equiv 289 \pmod{n}$	$[17,17]$	*True*
$2802^2 \equiv 4185 \pmod{n}$	$[3,3,3,5,31]$	*False*
$279^2 \equiv 3285 \pmod{n}$	$[3,3,5,73]$	*False*
$1354^2 \equiv 481 \pmod{n}$	$[13,37]$	*False*
$6145^2 \equiv 4624 \pmod{n}$	$[2,2,2,2,17,17]$	*True*
$3225^2 \equiv 63 \pmod{n}$	$[3,3,7]$	*True*

Vemos que en este caso hemos podido obtener 6 enteros que cumplen la condición deseada. Si obtuviéramos 5 o menos tendríamos que ejecutar de nuevo el programa para asegurar la condición $k > m$ de la que hablábamos en la descripción del método.

A continuación observamos cuales serían los exponentes en nuestra base de factores de los enteros elegidos.

u_i	$Exponente \pmod 2$
5040	$(0,0,1,1,0)$
4375	$(0,0,0,1,0)$
1125	$(0,0,1,0,0)$
256	$(0,0,0,0,0)$
5346	$(1,1,0,0,1)$
63	$(0,0,0,1,0)$

Solo con observar la tabla podemos ver que algunos elementos ya son cuadrados por lo que en este caso no haría falta combinar con los otros elementos. Por ejemplo podríamos tomar el cuarto y tendríamos que $256 \equiv 2087^2 \pmod{6213}$ y además $16 \not\equiv \pm 2087 \pmod{6213}$, por lo que $\gcd(2087 - 16, 6213) = 2071$ y $\gcd(2087 + 16, 6213) = 3$ son factores de $n = 6213$.

Sin embargo supongamos que no hemos tenido la suerte de encontrar elementos que sean cuadrados, entonces habría que buscar una combinación que si lo fuera. En este caso

podríamos proceder como sigue. Tomamos el primero, segundo y tercer elementos y vemos que la suma de sus exponentes es $(0,0,2,2,0)$, por lo que $5040 \cdot 4375 \cdot 1125$ es un cuadrado perfecto. Como consecuencia obtendríamos que $5040 \cdot 4375 \cdot 1125 \equiv (3087 \cdot 4094 \cdot 2301)^2 \pmod{6213}$. Además se cumple que $\underbrace{\sqrt{5040 \cdot 4375 \cdot 1125}}_{2175} \not\equiv \pm \underbrace{3087 \cdot 4094 \cdot 2301}_{4038} \pmod{6213}$.

Finalmente solo queda computar $\gcd(4038 - 2175, 6213) = 3$ y $\gcd(4038 + 2175, 6213) = 2071$, y vemos que $n = 6213 = 2071 \cdot 3$.

Si por lo contrario no podemos encontrar una combinación cumpliendo los requisitos impuestos deberíamos ejecutar de nuevo el algoritmo, ya que como es aleatorio nos dará unos resultados diferentes con los que podríamos tener más suerte.

Como podemos observar, para enteros relativamente pequeños este método no supone una mejora de los métodos ya mencionados, sino que es más lento. Sin embargo cuando nos encontramos con uno de mayor tamaño que no contiene en su factorización primos pequeños si que reduce el número de operaciones necesarias.

Concluimos así con el método de Dixon. A continuación se expondrá uno de los métodos de factorización más rápidos en la actualidad, la criba cuadrática, que no es más que una mejora en las ideas aquí aplicadas.

1.6. La criba cuadrática

El algoritmo de la criba cuadrática (QS del inglés *quadratic sieve*) es un algoritmo de propósito general de factorización de enteros desarrollado por Carl Pomerance en 1982. En la actualidad es el segundo método más eficiente para la factorización de enteros (solo por detrás de la criba general del cuerpo de números), y sigue siendo el más rápido para números de menos de 100 cifras.

Este método también se basa en la idea de Kraitchik y como bien señala su autor en [CP05] debe mucho a otros métodos de factorización anteriores como el de las fracciones continuas. Se puede indagar más sobre la historia de la criba cuadrática y de la criba general del cuerpo de números (el algoritmo de factorización más rápido actualmente) en [Pom96].

Como ya se dijo, la criba cuadrática gana eficiencia respecto al método de Dixon ya presentado. Aquí se refinan algunas ideas en la búsqueda de enteros p-lisos que lo hacen notablemente más rápido que su antecesor. El resto del proceso es similar al ya descrito en la sección anterior.

Antes de desarrollar el método es primordial introducir un concepto muy utilizado en teoría de números y que será usado posteriormente.

Definición 1.1. Se dice que $x \in \mathbb{Z}_N$ es un residuo cuadrático de N si x tiene una raíz cuadrada módulo N, esto es, si existe un $y \in \mathbb{Z}_N$ tal que $y^2 \equiv x \pmod{N}$.

Reducimos nuestro estudio al caso en el que N es primo, ya que el Teorema Chino del Resto nos permite centrarnos en este caso concreto.

Proposición 1.4. *Sea p un primo impar y a un entero no divisible por p. Entonces la congruencia*

$$x^2 \equiv a \pmod{p}$$

o bien no tiene solución o tiene dos soluciones congruentes módulo p.

Demostración. Si x, y son soluciones entonces $x^2 \equiv y^2 \pmod{p}$, por tanto $p|(x^2 - y^2)$. Entonces $p|(x-y)(x+y)$, por lo que $p|(x-y)$ o $p|(x+y)$, así que $x \equiv \pm y \pmod{p}$. □

Sabiendo todo esto podemos hacer entonces la siguiente definición.

Definición 1.2. (Símbolo de Legendre). Sea a un entero y sea $p > 2$ un primo. Definimos el simbolo de Legendre $\left(\frac{a}{p}\right)$ como sigue:

$$\left(\frac{a}{p}\right) = \begin{cases} 0, & \text{si } p|a, \\ 1, & \text{si a es un residuo cuadrático módulo p,} \\ -1, & \text{si a no es un residuo cuadrático módulo p.} \end{cases}$$

En el siguiente resultado damos una forma fácil de calcular el símbolo de Legendre, es decir, de identificar cuando a es un residuo cuadrático módulo p. La demostración que damos del mismo ha sido extraída de [Kob94].

Teorema 1.2. *(Criterio de Euler). Sea $p > 2$ primo y $a \in \mathbb{Z}$. Entonces:*

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

Demostración. Si a es divisible por p la identidad se cumple trivialmente. Supongamos ahora que $p \nmid a$. Por el pequeño Teorema de Fermat (Teorema 1.1), podemos afirmar que el cuadrado de $a^{\frac{p-1}{2}}$ es 1 en $\mathbb{Z}_p$ por lo que $a^{\frac{p-1}{2}}$ debe ser igual a ± 1. Sea g un elemento primitivo de $\mathbb{Z}_p^*$ (que existe por ser $\mathbb{Z}_p^*$ cíclico, ver [Con18], donde se exponen 8 demostraciones distintas de este hecho), entonces existe j tal que $a = g^j$. Por otra parte es claro que a es un residuo cuadrático si y solo si j es par. Así, tenemos que $a^{\frac{p-1}{2}} = g^{\frac{j\cdot(p-1)}{2}}$ es 1 si y solo si $\frac{j\cdot(p-1)}{2}$ es divisible por $p-1$, o lo que es lo mismo, si y solo si j es par. □

El siguiente resultado proporciona una serie de propiedades útiles sobre el símbolo de Legendre.

Proposición 1.5. *Sean a, b enteros y p un entero primo. Se cumplen las siguientes propiedades.*

1) $\left(\frac{0}{p}\right) = 0.$

2) $\left(\frac{1}{p}\right) = 1.$

3) *Si* $a \equiv b \pmod{n}$, *entonces* $\left(\frac{a}{p}\right) = \left(\frac{b}{p}\right).$

4) $\left(\frac{ab}{p}\right) = \left(\frac{a}{p}\right)\left(\frac{b}{p}\right).$

Demostración. 1) Por el criterio de Euler vemos que $0^{(p-1)/2} \equiv 0 \pmod{p}$.
2) También por el criterio de Euler vemos que $1^{(p-1)/2} \equiv 1 \pmod{p}$.
3) Claramente, si $a \equiv b \pmod{p}$ entonces también se tiene que $a^{(p-1)/2} \equiv b^{(p-1)/2} \pmod{p}$, y otra vez por el criterio de Euler se tiene lo deseado.
4) $\left(\frac{ab}{p}\right) = (ab)^{(p-1)/2} \pmod{p} = a^{(p-1)/2}b^{(p-1)/2} \pmod{p} = \left(\frac{a}{p}\right)\left(\frac{b}{p}\right).$ □

Aprovechamos la ocasión para enunciar y demostrar uno de los resultados más importantes en teoría de números, la Ley de reciprocidad cuadrática, siguiendo lo descrito en [Shoo9]. Antes de esto necesitamos algunos resultados para proceder de manera más cómoda con la demostración. En el proceso probaremos un resultado muy útil para calcular símbolos de Legendre de la forma $\left(\frac{2}{p}\right)$.

Lema 1.1. *(Lema de Gauss). Sea $p > 2$ primo y $a \in \mathbb{Z}$ tal que $\gcd(a, p) = 1$. Además, sea ω el número de enteros en el conjunto*

$$\Omega = \left\{1a, 2a, 3a, \ldots \frac{(p-1)}{2}a\right\}$$

cuyos residuos módulo p son negativos (o mayores que $p/2$), entonces

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

Demostración. Sean $b_1, b_2, \ldots, b_t$ los elementos del conjunto Ω cuyo residuo módulo p es menor que $p/2$ y $c_1, c_2, \ldots, c_\omega$ los elementos con residuo mayor que $p/2$. Entonces $t + \omega = \frac{(p-1)}{2}$. Consideramos ahora la secuencia $0 < b_1, b_2, \ldots, b_t, p - c_1, p - c_2, \ldots, p - c_\omega < p/2$.

Cada uno de estos es diferente. Claramente $b_i \neq b_j$ y $c_i \neq c_j$ cuando $i \neq j$. Además, si $b_i = p - c_j$, sea $b_i = rq$ y $c_j = sq$. Entonces $r + s = 0$, lo cual es imposible porque $0 < r, s < p/2$.

Entonces la secuencia $b_1, b_2, \ldots, b_t, p - c_1, p - c_2, \ldots, p - c_\omega$ se corresponde con los valores $1, 2, \ldots, (p-1)/2$, por tanto:

$$a^{\frac{p-1}{2}}\left(\frac{p-1}{2}\right)! = a \cdot (2a) \cdots \left(\frac{a(p-1)}{2}\right) = b_1 \cdot b_2 \cdots b_t \cdot c_1 \cdot c_2 \cdots c_\omega$$

$$= (-1)^{\omega} \cdot b_1 \cdot b_2 \cdots b_t \cdot (p - c_1) \cdot (p - c_2) \cdots (p - c_\omega) = (-1)^{\omega}\left(\frac{p-1}{2}\right)!.$$

Solo nos queda dividor por $(\frac{p-1}{2})!$ y finalmente por el criterio de Euler, $\left(\frac{a}{p}\right) = (-1)^{\omega}$. □

Proposición 1.6. *Sea $p \in \mathbb{Z}$ primo impar y sea $a \in \mathbb{Z}$ tal que $\gcd(p, a) = 1$. Entonces $\left(\frac{a}{p}\right) = (-1)^m$, donde*

$$m = \left\lfloor \frac{a}{p} \right\rfloor + \left\lfloor 2\frac{a}{p} \right\rfloor + \cdots + \left\lfloor \frac{p-1}{2}\frac{a}{p} \right\rfloor.$$

Además se satisface que $\left(\frac{2}{p}\right) = (-1)^{\frac{p^2-1}{8}}$.

Demostración. Para cada $i \in \{1, 2, \ldots, (p-1)/2\}$, consideramos la ecuación $ia = p\left\lfloor \frac{ia}{p} \right\rfloor + r_i$, con $0 < r_i < p$. Observando la prueba del Lema de Gauss vemos que los r_i coinciden con los b_i y c_i que entonces considerábamos.

Sumando las ecuaciones para $i \in \{1, 2, \ldots, (p-1)/2\}$ obtenemos:

$$\sum_{j=1}^{\frac{p-1}{2}} aj = \sum_{j=1}^{\frac{p-1}{2}} p\left\lfloor \frac{ja}{p} \right\rfloor + \sum_{j=1}^{t} b_j + \sum_{j=1}^{\omega} c_j. \tag{1.1}$$

Además, atendiendo a la demostración del Lema de Gauss sabemos que:

$$\sum_{j=1}^{\frac{p-1}{2}} j = \sum_{j=1}^{t} b_j + \sum_{j=1}^{\omega}(p - c_j) = \sum_{j=1}^{t} b_j + \sum_{j=1}^{\omega}(p - c_j) = \omega p + \sum_{j=1}^{t} b_j - \sum_{j=1}^{\omega} c_j. \tag{1.2}$$

Restando (1.2) a (1.1) conseguimos lo siguiente:

$$(a-1)\sum_{j=1}^{\frac{p-1}{2}} j = p\left(\sum_{j=1}^{(p-1)/2} \left\lfloor \frac{ja}{p} \right\rfloor - \omega\right) + 2\sum_{j=1}^{\omega} c_j.$$

Esto es lo mismo que:

$$(a-1)\frac{p^2-1}{8} \equiv \sum_{j=1}^{(p-1)/2} \left\lfloor \frac{ja}{p} \right\rfloor - \omega \pmod 2. \tag{1.3}$$

Vemos entonces que si a es impar obtenemos lo siguiente:

$$\omega \equiv \sum_{j=1}^{(p-1)/2} \left\lfloor \frac{ja}{p} \right\rfloor \pmod 2.$$

Consideramos el caso particular en el que $a = 2$. En este caso $\left\lfloor \frac{2j}{p} \right\rfloor = 0$ para $j \in \{1, 2, \ldots, (p-1)/2\}$, por tanto, de (1.3) deducimos que:

$$\omega \equiv \frac{p^2-1}{8} \pmod 2.$$

□

Teorema 1.3. *(Ley de reciprocidad cuadrática) Si p, q son dos primos distintos e impares, entonces:*

$$\left(\frac{p}{q}\right)\left(\frac{q}{p}\right) = (-1)^{(p-1)(q-1)/4}.$$

O lo que es lo mismo:

1) $\left(\frac{p}{q}\right) = -\left(\frac{q}{p}\right)$ *si* $p, q \equiv 3 \pmod 4$.

2) $\left(\frac{p}{q}\right) = \left(\frac{q}{p}\right)$ *si* $p \equiv 1 \pmod 4$ *o* $q \equiv 1 \pmod 4$.

Demostración. Por el Lema de Gauss sabemos que $\left(\frac{p}{q}\right) = (-1)^{\omega}$ y $\left(\frac{q}{p}\right) = (-1)^{\mu}$ para ciertos ω y μ. Ahora por la Proposición 1.6 sabemos que $\omega \equiv m \pmod 2$ con $m = \sum_{i=1}^{\frac{p-1}{2}} \left\lfloor \frac{qi}{p} \right\rfloor$ y $\mu \equiv n \pmod 2$ con $n = \sum_{j=1}^{\frac{q-1}{2}} \left\lfloor \frac{pj}{q} \right\rfloor$. Podemos concluir entonces lo siguiente:

$$\left(\frac{p}{q}\right)\left(\frac{q}{p}\right) = (-1)^{(n+m)}.$$

A la luz de esto solo resta probar que $n+m=(p-1)(q-1)/4$. Para esto utilizaremos un elegante argumento geométrico debido a Eisenstein, este es como sigue.

Consideremos el rectángulo con vértices en $(0,0)$, $(p/2,q/2)$, $(p/2,0)$ y $(0,q/2)$.

Veamos ahora el número de puntos con coordenadas enteras que hay en el interior del rectángulo, es decir, puntos que cumplen que $0<x\leq(p-1)/2$ y $0<y\leq(q-1)/2$. Vemos rápidamente que el número de puntos cumpliendo estas condiciones es $(p-1)(q-1)/4$.

Procedemos ahora a contar estos puntos de manera distinta. Trazamos la recta diagonal r que une $(0,0)$ con $(p/2,q/2)$. El segmento que une $(0,0)$ con $(p/2,q/2)$ no contiene ningún punto con coordenadas enteras ya que los puntos con coordenadas enteras en r tienen la forma (up,uq) con $u\in\mathbb{Z}$. Determinaremos ahora la cantidad de puntos por encima y por debajo de r, que deberá coincidir con $(p-1)(q-1)/4$.

Sabiendo que $r=\{(x,y)/0\leq x\leq p/2, 0\leq y\leq q/2\}$, los puntos con coordenadas enteras por debajo de r son aquellos (i,l) tales que $1\leq l\leq qj/p$, por tanto el número de puntos cumpliendo esto es $\sum_{i=1}^{\frac{p-1}{2}}\left\lfloor\frac{iq}{p}\right\rfloor$.

Procedemos de igual manera para los puntos por encima de r y obtenemos esta vez $\sum_{j=1}^{\frac{q-1}{2}}\left\lfloor\frac{jp}{q}\right\rfloor$.

Por tanto obtenemos finalmente la siguiente igualdad:

$$(p-1)(q-1)/4=\sum_{i=1}^{\frac{p-1}{2}}\left\lfloor\frac{iq}{p}\right\rfloor+\sum_{j=1}^{\frac{q-1}{2}}\left\lfloor\frac{jp}{q}\right\rfloor.$$

Obteniendo así lo que buscábamos, acabando por tanto la demostración. □

Después de este paréntesis, ahora sí, pasamos a describir el algoritmo de la criba cuadrática. Lo primero que cambia Pomerance es la aleatoriedad que imperaba en el de Dixon. En este caso, en lugar de ir probando con números aleatorios comienza tomando enteros cercanos a $m=\lceil\sqrt{n}\rceil$. Vamos a considerar la siguiente aplicación:

$$q(x)=(m+x)^2-n=m^2+2mx+x^2-n\approx x^2+2mx,\ x=0,1,2,\ldots$$

Por lo que el cuadrado en $\mathbb{Z}_n$ de los elementos que ahora tomamos es del orden de $\lceil\sqrt{n}\rceil$, con lo que es razonable que se pueda factorizar usando primos pequeños.

También será de vital importancia la siguiente observación. Nuestro propósito es que $q(x)$ sea divisible por los primos en nuestra base de factores, supongamos entonces que p primo divide a $q(x)$. Tenemos entonces lo siguiente:

$$q(x)\equiv 0 \pmod{p},$$

o lo que es lo mismo

$$q(x)\equiv(x+m)^2-n\equiv 0 \pmod{p},$$

obteniendo así que n es residuo cuadrático módulo p. Es por esto que en nuestra base de factores solo consideraremos primos tales que $\left(\frac{n}{p}\right)=1$.

Estas dos mejoras hacen que aumente mucho la eficiencia en comparación con el método de Dixon, en el que el azar jugaba un papel fundamental y se necesitaba un mayor número de operaciones para encontrar números p-lisos.

Pasamos a dar el esquema general de la criba cuadrática. Como ya se dijo antes, este

también sigue el patrón de actuación propuesto por Kraitchik. En este caso el conjunto $\{s_i\}$ del que hablábamos con el método de Dixon es el que sigue:

$$S = \left\{ \lceil \sqrt{n} \rceil + x, x = 0, 1, 2, \ldots, M_1 \right\}.$$

Aquí M_1 es una cota impuesta a priori. Para la inicialización del algoritmo también se requiere otra cota, M_2, para la formación de nuestra base de factores, que vendrá dada por el siguiente conjunto:

$$B = \left\{ 2 < p \leq M_2 | \left(\frac{n}{p} \right) = 1 \right\} \cup \{2\}.$$

Una vez tenemos definidos los dos conjuntos procedemos de manera similar a como lo hacíamos con Dixon, es decir, tomamos un elemento s de nuestro conjunto S y calculamos $u \equiv s^2 \pmod{n}$ que en este caso coincide con $u = s^2 - n$. Por lo que nos es más fácil proceder de la primera forma, teniendo en cuenta que esto solo se cumple para $s < \lceil \sqrt{2n} \rceil$.

Seguidamente comprobamos si u es B-liso (descompone en B), y si es así almacenamos s, u y además calculamos su exponente (en nuestra base de factores) módulo 2, al igual que hacíamos en el método de Dixon.

Continuamos este proceso hasta obtener $|B| + 1$ elementos que satisfagan esto, o bien hasta que nos quedemos sin elementos en S.

Una vez hemos conseguido los $|B| + 1$ elementos procedemos como en la segunda parte del método de Dixon. Usamos el álgebra lineal para encontrar una combinación con la que obtener la congruencia de cuadrados deseada.

Una observación interesante es que en el conjunto S podemos considerar, en lugar de un intervalo a partir de $\lceil \sqrt{n} \rceil$, un intervalo simétrico en $\lceil \sqrt{n} \rceil$. Es decir, podríamos considerar

$$S = \left\{ \lceil \sqrt{n} \rceil + x, x = 0, \pm 1, \pm 2, \ldots, \pm M_1 \right\}.$$

No obstante, como bien señala el autor en [CP05], es importante ver que si hacemos esto, al computar $s^2 - n$ obtendremos un valor negativo en algunas ocasiones.

Esto tiene fácil solución, solo habría que añadir una componente a nuestros vectores exponente que sea 0 si nuestro número es positivo y 1 si es negativo (equivalente a añadir el elemento -1 a nuestra base de factores). Pero entonces aumenta el cardinal de B y se hace necesario encontrar otro elemento B-liso. Esta es una desventaja a tener en cuenta, pero por otra parte podemos aprovechar que si consideramos este nuevo conjunto obtendremos más números pequeños que tienen altas posibilidades de ser B-lisos.

La elección del conjunto S se puede hacer indistintamente de una manera u otra, y no afecta sobremanera el funcionamiento del algoritmo, en este caso nosotros nos quedaremos con $S = \{ \lceil \sqrt{n} \rceil + x, x = 0, \pm 1, \pm 2, \ldots, \pm M_1 \}$.

Cabe destacar que esto es una primera aproximación al método de la criba cuadrática, ya que este es un método muy complejo, con muchas variantes y al que se le pueden hacer distintas mejoras para aumentar su eficiencia. El lector interesado en profundizar sobre el tema puede consultar más en [CP05] y [Bre12]. En este último se puede ver un ejemplo práctico en el que el autor lo explica todo con mucho detalle.

Tras esto procedemos a probar el algoritmo en un ejemplo. Como en el método de Dixon elegimos de nuevo un entero con pocos dígitos para no extenderlo mucho, ya que solo nos interesa ver la idea del método, y esta es la misma tanto para números pequeños como para números grandes.

Inmediatamente después se puede ver el pseudocódigo de la criba cuadrática Algoritmo 9.

Ejemplo 1.8. Procedemos a factorizar el entero $n = 7381$, elegimos el parámetro $M_1 = 20$ y $M_2 = 19$. Ejecutando el programa de Python, cuyo código se puede consultar en nuestro repositorio, conseguimos obtener 9 enteros que cumplen lo deseado, es decir, factorizan completamente en nuestra base de factores, que en este caso se puede comprobar que es la siguiente:

$$B = \{-1, 2, 3, 5, 13, 19\}.$$

La siguiente tabla muestra los resultados obtenidos, al igual que hacíamos en el ejemplo 1.7.

$s_i^2 \equiv u_i \pmod{n}$	Descomposición	$B - liso$
$86^2 \equiv 15 \pmod{n}$	$[3, 5]$	*True*
$87^2 \equiv 188 \pmod{n}$	$[2, 2, 47]$	*False*
$85^2 \equiv -156 \pmod{n}$	$[-1, 2, 2, 3, 13]$	*True*
$88^2 \equiv 363 \pmod{n}$	$[3, 11, 11]$	*False*
$84^2 \equiv -325 \pmod{n}$	$[-1, 5, 5, 13]$	*True*
$89^2 \equiv 540 \pmod{n}$	$[2, 2, 3, 3, 3, 5]$	*True*
$83^2 \equiv -492 \pmod{n}$	$[-1, 2, 2, 3, 41]$	*False*
$90^2 \equiv 719 \pmod{n}$	$[719]$	*False*
$82^2 \equiv -657 \pmod{n}$	$[-1, 3, 3, 73]$	*False*
$91^2 \equiv 900 \pmod{n}$	$[2, 2, 3, 3, 5, 5]$	*True*
$81^2 \equiv -820 \pmod{n}$	$[-1, 2, 2, 5, 41]$	*False*
$92^2 \equiv 1083 \pmod{n}$	$[3, 19, 19]$	*True*
$80^2 \equiv -981 \pmod{n}$	$[-1, 3, 3, 109]$	*False*
$93^2 \equiv 1268 \pmod{n}$	$[2, 2, 317]$	*False*
$79^2 \equiv -1140 \pmod{n}$	$[-1, 2, 2, 3, 5, 19]$	*True*

Seguidamente construimos otra tabla con los exponentes en nuestra base B de los enteros obtenidos.

u_i	Exponente $\pmod 2$
15	$(0, 0, 1, 1, 0, 0)$
-156	$(1, 0, 1, 0, 1, 0)$
-325	$(1, 0, 0, 0, 1, 0)$
540	$(0, 0, 1, 1, 0, 0)$
900	$(0, 0, 0, 0, 0, 0)$
1083	$(0, 0, 1, 0, 0, 0)$
-1140	$(1, 0, 1, 1, 0, 1)$

Aquí, al igual que pasaba en el ejemplo anterior tenemos la suerte de que 900 es un cuadrado perfecto. Ocurre entonces que $91^2 \equiv 900 \pmod{n}$ y además $91 \not\equiv 30 \pmod{n}$. Sabiendo esto solo falta hacer $\gcd(91 - 30, n) = 61$ y $\gcd(91 + 30, n) = 121$, por lo que obtenemos que $n = 7381 = 61 \cdot 121$.

Sin embargo, si no hubiéramos corrido la misma suerte tendríamos que recurrir al álgebra lineal para encontrar una combinación con la que obtengamos un cuadrado perfecto. En este caso podemos tomar el primer y cuarto elementos de la tabla, es decir, 15 y 540 y sabemos que $15 \cdot 540 \equiv (86 \cdot 89)^2 \pmod{n}$. También se cumple que $90 \not\equiv 273 \pmod{n}$. Finalmente solo falta hacer $\gcd(273 - 90, n) = 61$ y $\gcd(273 + 90, n) = 121$ y como es lógico obtenemos de nuevo $n = 7381 = 61 \cdot 121$.

Al igual que pasaba con el método de Dixon, para números pequeños no supone una gran diferencia respecto a métodos como el ρ de Pollard o $p-1$ de Pollard. No obstante, la criba cuadrática toma mayor valor cuando trabajamos con números de mayor tamaño, y hasta 100 dígitos es el algoritmo de factorización más rápido hoy en día.

Algorithm 9 Criba cuadrática (QS)

Input: $n \in \mathbb{Z}$, $M_1 \in \mathbb{N}, M_2 \in \mathbb{N}$
Output: $\{p, q\}$ o fail

```
S ← {}
B ← {2}
for k = 0 : M_1 do                                   ▷ Generamos el conjunto S
    S ← S ∪ {⌈√n⌉ ± k}
end for
for 2 < p ≤ M_2 primo do                            ▷ Generamos la base de factores
    if (n/p) = 1 then
        B ← B ∪ {p}
    end if
end for
L ← {}                                               ▷ Definimos dos conjuntos vacíos
Z ← {}
i ← 0
while i ≤ 2M_1 + 1 y |L| ≤ |B| do                    ▷ En S hay 2M_1 + 1 elementos
    z_i ← S(i)                                       ▷ S(i) es el elemento i del vector L
    w_i ← z_i^2 − n
    if w_i = ∏_{j=1}^{|B|} B(j)^{a_ij} then          ▷ Descompone en B
        L ← L ∪ (a_i1, ..., a_i|B|)                  ▷ Almacenamos los exponentes
        Z ← Z ∪ {z_i}
    end if
    i ← i + 1
end while
if |B| < |L| then                                    ▷ Condición para que exista solución no trivial
    Resolvemos por Gauss para x en la ecuación 0⃗ ≡ xL (mod 2)
    a ← ∏_{j=1}^{|Z|} Z(j)^{x_j}                     ▷ Z(j) es el elemento j de Z
    b ← ∏_{j=1}^{|B|} B(j)^{(Σ_{i=1}^{|L|} x_i a_ij)/2}   ▷ Y se cumple que a^2 ≡ b^2 (mod n)
    if a ≢ ±b (mod n) then
        return {gcd(a + b, n), gcd(a − b, n)}
    else
        return Fail
    end if
else
    return Fail
end if
```

2. Pruebas de primalidad en enteros

A lo largo de este capítulo se expondrán distintas pruebas de primalidad, es decir, algoritmos que nos permiten concluir si un número es primo, o por el contrario compuesto. Podemos distinguir entre dos tipos de pruebas o test de primalidad. Por un lado los test determinísticos, que nos permiten clasificar un número como primo o compuesto con total certeza. Y por otro lado nos encontramos con los test de primalidad probabilísticos, los cuales nos permiten distinguir los números primos con una probabilidad tan alta como queramos.

2.1. Pruebas deterministas

En esta sección se estudiarán las siguientes pruebas de primalidad: la clásica criba de Eratóstenes, la prueba de Wilson, el test de Lucas y para terminar el test de primalidad AKS, que demuestra que el problema de determinar la primalidad de un número es de clase P.

2.1.1. Criba de Eratóstenes

Este conocido método fue desarrollado, como bien indica su nombre, por el matemático griego Eratóstenes en el siglo III a.C. Este proporciona un algoritmo que permite obtener todos los primos menores que un cierto número dado.

Básicamente, lo que hace el algoritmo es marcar sucesivamente los múltiplos de los primos, empezando por 2. Tras esto, el primer número no marcado tras el 2 será considerado primo, y procedemos de igual forma para este. El proceso termina cuando todos los números están marcados o son primos.

Mostramos el proceso para los 50 primeros naturales en la siguiente tabla:

	2	3	~~4~~	5
~~6~~	7	~~8~~	~~9~~	~~10~~
11	~~12~~	13	~~14~~	~~15~~
~~16~~	17	~~18~~	19	~~20~~
~~21~~	~~22~~	23	~~24~~	~~25~~
~~26~~	~~27~~	~~28~~	29	~~30~~
31	~~32~~	~~33~~	~~34~~	~~35~~
~~36~~	37	~~38~~	~~39~~	~~40~~
41	~~42~~	43	~~44~~	~~45~~
~~46~~	47	~~48~~	~~49~~	~~50~~

2.1.2. Prueba de Wilson

Presentamos un test basado en el siguiente teorema proporcionado por J.Wilson cuyo recíproco también veremos que es cierto. Ambas demostraciones se basan en lo estudiado en [FR07].

Teorema 2.1. *(Teorema de Wilson). Sea un número entero $p > 1$. Se tiene que p es primo si se satisface:*

$$(p-1)! \equiv -1 \pmod{p}.$$

Demostración. Sabemos que $(p-1)! = 1 \cdot 2 \cdots (p-2) \cdot (p-1)$. Notamos que al ser p primo $\mathbb{Z}_p$ es un cuerpo por lo que cada $x \in \{1, 2, \ldots, p-1\}$ tiene un inverso multiplicativo módulo p. Supongamos además que $x = x^{-1}$ en $\mathbb{Z}_p$, entonces $x^2 = 1 \pmod{p}$ lo que implica que $(x+1)(x-1) = 1$ en $\mathbb{Z}_p$ implicando esto último que $x = 1$ o $x = -1$ por ser $\mathbb{Z}_p$ un dominio de integridad. Hemos obtenido como consecuencia que solo 1 y $-1 \equiv p-1 \pmod{p}$ son sus propios inversos multiplicativos en $\mathbb{Z}_p$.

Viendo el producto anterior $1 \cdot 2 \cdots (p-2) \cdot (p-1)$ en $\mathbb{Z}_p$ y utilizando lo anterior observamos que cada elemento podemos asociarlo con su inverso excepto 1 y $p-1$, por lo que $1 \cdot 2 \cdots (p-2) \cdot (p-1) \equiv p-1 \equiv -1 \pmod{p}$. □

Vamos ahora con el recíproco.

Teorema 2.2. *Sea $p > 1$ un entero que cumple que $(p-1)! \equiv -1 \pmod{p}$, entonces p es primo.*

Demostración. Supongamos que $(p-1)! \equiv -1 \pmod{p}$. Si p fuera compuesto entonces podríamos escribirlo como $p = mk$ con $1 < k, m < p-1$. Si $m \neq k$, entonces los dos son factores de $(p-1)!$, por lo que $p|(p-1)!$ y $(p-1)! \equiv 0 \pmod{p}$ entrando en contradicción con lo supuesto.

Por otra parte, si $p = m^2$ con $m > 2$, podemos ver que $2m < p$. Esto nos indica que m y $2m$ dividen a $(p-1)!$ por lo que $2p|(p-1)!$ y por tanto $(p-1)! \equiv 0 \pmod{p}$. Por tanto, p deber ser primo. El caso restante, si $m = k = 2$, entonces $p = 4$ y $(p-1)! \equiv 6 \pmod{p}$, que no es congruente con -1 módulo 4. □

Viendo estos resultados se puede intuir la idea del método. Es simple, consiste en computar $(n-1)! + 1$ y comprobar si es divisible por n. Si es así, tenemos que n es primo, y si no es compuesto.

Como el método, el pseudocódigo es muy sencillo, se presenta a continuación.

Algorithm 10 Prueba de Wilson

Input: $n \in \mathbb{Z}$
  $a \leftarrow (n-1)! + 1$
  if $n|a$ **then**
    return True
  else
    return False
  end if

Ponemos unos ejemplos sencillos de utilización de este test.

Ejemplo 2.1. En primer lugar lo aplicamos a $n = 97$ y obtenemos que efectivamente obtenemos que es primo ya que $(97-1)! + 1 \equiv 0 \pmod{n}$. En cambio para $n = 27$ vemos que

$(27-1)!+1 \equiv 1 \pmod{n}$ por lo que obtenemos que es compuesto. Todo esto se ha hecho utilizando el código en Python de la prueba, que se puede consultar, como los demás, en nuestro repositorio.

Es importante señalar que el interés de este algoritmo es puramente teórico, debido a que en la práctica el cálculo del factorial tiene una alta complejidad computacional. Esto podría mejorarse empleando un algoritmo específico para calcular el factorial módulo n pero aún así no merece la pena por la existencia de algoritmos mucho más rápidos en la actualidad, algunos de los cuales veremos a lo largo de este capítulo.

2.1.3. Test de Lucas

El test se basa en una idea atribuida a E. Lucas, propuesta en 1876. Antes de presentarla vamos a enunciar y demostrar uno de los teoremas más importantes en teoría de números, como es el Teorema de Euler, que nos servirá posteriormente para probar el teorema en el que se basa este test.

Este teorema utiliza la conocida función φ de Euler (también llamada función totiente de Euler). Lo que esta función indica, dado un entero positivo n, es la cantidad de números menores o iguales que n coprimos con el.

Teorema 2.3. *(Teorema de Euler). Si a y n son enteros primos relativos entonces:*

$$a^{\varphi(n)} \equiv 1 \pmod{n}.$$

Demostración. Consideramos el conjunto $\mathcal{U}(\mathbb{Z}_n) = \{x \in \mathbb{Z}_n / \exists x^{-1}\}$, las unidades del anillo $\mathbb{Z}_n$. Observamos que por la identidad de Bezout esto es lo mismo que el conjunto $\mathcal{U}(\mathbb{Z}_n) = \{x \in \mathbb{Z} / \gcd(x,n) = 1\}$. Sabemos entonces que el cardinal del conjunto $\mathcal{U}(\mathbb{Z}_n)$ es $\varphi(n)$. Sea $a \in \mathcal{U}(\mathbb{Z}_n)$ definimos la siguiente aplicación:

$$\begin{array}{rccc} f\colon & \mathcal{U}(\mathbb{Z}_n) & \longrightarrow & \mathcal{U}(\mathbb{Z}_n) \\ & x & \longrightarrow & ax \end{array}$$

Podemos ver entonces que $\prod_{u\in\mathcal{U}(\mathbb{Z}_n)} u \equiv \prod_{u\in\mathcal{U}(\mathbb{Z}_n)} f(u) \equiv a^{\varphi(n)} \prod_{u\in\mathcal{U}(\mathbb{Z}_n)} u \pmod{n}$. Y aquí multiplicando por el inverso de $\prod_{u\in\mathcal{U}(\mathbb{Z}_n)} u$ obtenemos que $a^{\varphi(n)} \equiv 1 \pmod{n}$. □

Ahora si podemos presentar el teorema fundamental en este test.

Teorema 2.4. *(Teorema de Lucas). Sean $a, n \in \mathbb{Z}$ con $a, n > 1$ cumpliendo que $a^{n-1} \equiv 1 \pmod{n}$, pero $a^{(n-1)/q} \not\equiv 1 \pmod{n}$ para todo divisor primo de $n-1$, entonces n es primo.*

Demostración. La condición nos dice que el orden de a en $\mathbb{Z}_n$ es un divisor de $n-1$. Por otra parte la condición $a^{(n-1)/q} \not\equiv 1 \pmod{n}$ implica que el orden de a no es un divisor propio de $n-1$, sino que es $n-1$. También sabemos por el Teorema de Euler (Teorema 2.3) que el orden de a divide a $\varphi(n)$, así que $n-1 < \varphi(n)$.

Si suponemos por reducción al absurdo que n es compuesto y un factor primo suyo es p, entonces p y n son enteros que no son coprimos con n, esto implica que $\varphi(n) \leq n-2$, pero esto entra en contradicción con que $n-1 < \varphi(n)$. Esta contradicción viene de suponer que n es compuesto por lo que debe ser primo. □

También será de utilidad para este test el siguiente resultado elemental de teoría de grupos.

Proposición 2.1. *Si existe $a \in \mathbb{Z}_n$ tal que su orden multiplicativo es $n-1$, entonces n es primo.*

Demostración. Sabemos por hipótesis que el grupo $\mathbb{Z}_n^*$ es un grupo cíclico con elemento primitivo a. Por tanto $\mathbb{Z}_n$ es un cuerpo, lo que implica que n es primo. □

Una vez conocemos estos resultado, podemos adivinar como será el test. Inicialmente comprobamos que $a^{n-1} \equiv 1 \pmod{n}$, condición necesaria para que sea primo (por el PTF, Teorema 1.1), si no ocurre podemos decir que n es compuesto. Seguidamente computamos $a^{(n-1)/q} \pmod{n}$ para cada uno de los divisores primos de $n-1$ (estos suelen ser complicados de obtener si el número tiene muchos dígitos). Si el resultado es distinto de 1 para todos entonces el teorema anterior Teorema 2.4 nos permite concluir que n es primo. En cambio, si para algún q el resultado es 1 entonces debemos seguir probando con $a+1$ hasta, en caso necesario llegar a $n-1$. Si llegamos a $n-1$ y aún así no podemos concluir nada entonces devolvemos que n es compuesto por la Proposición 2.1.

Aquí vemos el pseudocódigo del test. Este requiere como entrada el número al que se le hará el test, y además los divisores primos de $n-1$. Si alcanzamos $a = n-1$ y no hemos podido concluir que n es primo entonces debe ser compuesto por el resultado anterior.

Algorithm 11 Test de Lucas

Input: $n, q_1, \ldots, q_r \in \mathbb{Z}$
```
a ← 2
while a < n do
    bool ← True
    i ← 1
    if a^{n-1} (mod n) ≠ 1 then          ▷ Si esto no se cumple sabemos que no es primo
        return False
    end if
    while 1 ≤ i ≤ r y bool = True do
        if a^{(n-1)/q_i} (mod n) = 1 then          ▷ Si es igual a 1 tenemos que
            bool ← False                           ▷ salir del bucle para seguir con a+1
        end if
        i ← i + 1
    end while
    if bool=True then          ▷ Si salimos del bucle con el valor True entonces es primo
        return True
    end if
    a ← a + 1
end while
return False                                   ▷ Si llegamos hasta aquí es que es compuesto
```

Vamos ahora con unos ejemplos.

Ejemplo 2.2. ▪ Queremos ver si $n = 131$ es primo. Lo primero según el Algoritmo 11 es calcular $2^{n-1} \pmod{n}$. Al hacerlo comprobamos que efectivamente el resultado es 1. Procedemos entonces con la segunda parte del algoritmo, dados los divisores de 130, $[2, 5, 13]$. Comprobamos uno a uno si se cumple la condición deseada, vemos que $2^{130/2} \pmod{n} = 130$, $2^{130/5} \pmod{n} = 53$ y $2^{130/13} \pmod{n} = 107$. Obtenemos que se cumple la condición por lo que $n = 131$ es primo.

- Probamos con un entero de mayor tamaño, por ejemplo $n = 18181$. Vemos que de nuevo $2^{n-1} \pmod{n} = 1$. Seguimos con el test, los divisores de $n-1$ son $[2, 3, 5, 101]$. Vemos que $2^{18180/2} \pmod{n} = 18180$, $2^{18180/3} \pmod{n} = 6623$, $2^{18180/5} \pmod{n} = 9261$ y $2^{18180/101} \pmod{n} = 7454$. De nuevo podemos concluir que $n = 18181$ es un número primo.
- Vemos por último el caso del entero $n = 1009491929$. Vemos que $2^{n-1} \pmod{n} = 1$. En este caso los divisores de $n-1$ son $[2, 61, 107, 19333]$. Observamos que $2^{(n-1)/2} \pmod{n} = 1$, tenemos que probar entonces con 3. Ahora sí, se puede comprobar que $3^{(n-1)/2} \pmod{n} = 1009491928, 3^{(n-1)/61} \pmod{n} = 45793833, 3^{(n-1)/107} \pmod{n} = 720580793$ y $3^{(n-1)/19333} \pmod{n} = 540096076$. Podemos concluir entonces que 1009491929 es un número primo.

Este ejemplo se ha realizado con la ayuda del código programado en Python y que como siempre se puede encontrar en nuestro repositorio.

Para terminar con esta prueba comentamos brevemente la eficiencia de la misma. La necesidad de conocer los divisores primos de $n-1$ es la principal desventaja del test, ya que como vimos en el primer capítulo es una tarea difícil cuando el número tiene gran tamaño. Si por cualquier razón disponemos de los divisores es un método muy conveniente. Si esto no ocurre, para números menores de 30 cifras podemos utilizar el método ρ de Pollard (1.3), que para estos números funciona satisfactoriamente en la mayoría de los casos. En cambio, si el número es aún mayor lo ideal sería usar la criba cuadrática (1.6) o algún método más sofisticado. Es debido a esto que el test de Lucas no es muy utilizado.

Por otra parte podemos pensar que realizar las potencias del tipo $a^{(n-1)/q} \pmod{n}$ pueden tener un gran coste computacional. Este problema se puede solventar fácilmente recurriendo a algoritmos de cálculo como el método de los cuadrados iterados (se puede consultar en [CP05, pág 458]), que abaratan mucho este proceso. Es de este modo como lo hemos programado en nuestro repositorio. En concreto el algoritmo de los cuadrados iterados se encuentra en el fichero *algoritmos_complementarios*.

2.1.4. Test de primalidad AKS

Fue diseñado por M. Agrawal, N. Kayal y N.Saxena en 2002. Este supuso un gran avance por ser el primer algoritmo determinista que decide la primalidad de un número en tiempo polinomial.

Antes de su publicación no se sabía si quiera si era posible la invención de un algoritmo determinista que resolviera el problema de la primalidad en tiempo polinomial. Esto demuestra que el problema de determinar la primalidad de un número esta en la clase de complejidad P.

El siguiente resultado es fundamental para el test. Se trata de una generalización del Pequeño Teorema de Fermat (Teorema 1.1). El teorema es el que sigue:

Teorema 2.5. *Sean dos enteros $a, n \in \mathbb{Z}_n$ tal que $\gcd(a, n) = 1$. Se cumple que n es primo si y solo si se satisface la siguiente congruencia en $\mathbb{Z}_n[x]$,*

$$(x + a)^n \equiv x^n + a.$$

Demostración. Para $0 < i < n$, el coeficiente de x^i en $((x-a)^n - (x^n - a))$ es $\binom{n}{i} a^{n-i}$.

Por tanto, si n es primo vemos que $\binom{n}{i} \equiv 0 \pmod{n}$. Así obtenemos que todos los coeficientes son 0.

Supongamos ahora que n es compuesto. Consideramos un factor primo q de n y supongamos que $q^k | n$. Entonces q^k no divide a $\binom{n}{q}$ y es coprimo con a^{n-q}, por lo que el coeficiente de x^q no es $0 \pmod{n}$. Así $((x-a)^n - (x^n - a))$ no es 0 módulo n. □

El teorema anterior proporciona una herramienta que puede utilizarse para construir un test de primalidad. Dado un entero n tomamos otro entero cualquiera a y comprobamos que se cumple la congruencia del teorema. No obstante, esto requiere de muchas operaciones, ya que habría que evaluar todos los coeficientes de nuestro polinomio.

Esto podemos solucionarlo evaluando cada lado de la congruencia módulo un polinomio de la forma $x^r - 1$, escogiendo un r adecuado. Tras esto, lo que en realidad vamos a comprobar es que se satisface la siguiente ecuación:

$$(x^n + a) = x^n + a \pmod{x^r - 1, n}.$$

Cuando aquí escribimos $\pmod{x^r - 1, n}$ nos referimos al anillo cociente $\mathbb{Z}_n[x]/(x^r - 1)$.

Por el Teorema 2.5 sabemos que si n es primo se cumple la igualdad anterior. El problema lo tenemos si existen números compuestos que la verifiquen para algunos valores de a y r (y estos existen). Sin embargo, los autores del método prueban en [AKS04] el siguiente resultado Teorema 2.6, que es el pilar fundamental del test.

En este teorema designaremos por $o_n(a)$ al orden multiplicativo del un elemento a en el grupo $\mathbb{Z}_n$.

Teorema 2.6. *(Agrawal, Kayal, Saxena). Supongamos que $n \in \mathbb{Z}$ con $2 \leq n$, r es un entero coprimo con n de forma que $o_r(n) > \log_2^2(n)$ y se cumple que*

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

para todo entero $0 \leq a \leq \sqrt{\varphi(r)}\log_2(n)$. Si n tiene un factor primo p mayor que $\sqrt{\varphi(r)}\log_2(n)$ entonces $n = p^m$ para algún entero positivo m. En particular, si n no tiene factores primos entre 1 y $\sqrt{\varphi(r)}\log_2(n)$ y tampoco es potencia de un primo, entonces n es primo.

Demostración. Por extensión, complejidad y tecnicidad de la misma no se hará la demostración aquí. Sin embargo, la demostración de este hecho se puede encontrar tanto en [CP05] como en [AKS04], siendo este último el artículo donde se publica el test. □

Ahora sí, una vez dicho esto, estamos en disposición de presentar el pseudocódigo del test de primalidad AKS, que se puede consultar en la siguiente página.

A diferencia de los otros algoritmos presentados en esta memoria, en este caso no dispondremos de el código en Python de este test en el repositorio. Esto es debido principalmente a dos factores, el primero de ellos es la complejidad de programarlo en Python y el segundo es que considero que el valor de este test es principalmente teórico, ya que prueba que el problema de la primalidad está en P como se discutió anteriormente, y es a esto mayormente a lo que debe su fama.

Algorithm 12 Test AKS

Input: $n \in \mathbb{Z}$
  $bool \leftarrow True$
  if $n = p^m$ con p primo **then** ▷ Verificamos que no sea potencia de un primo
    return False
  end if
  $r \leftarrow 1$
  while $bool = True$ **do** ▷ Aquí obtenemos el valor de r
    $r \leftarrow r + 1$
    $bool2 \leftarrow True$
    $k \leftarrow 1$
    while $k \leq \lfloor \log_2(n) \rfloor$ y $bool2 = True$ **do** ▷ Si para todos esos k es distinto de 1,
      if $n^k \equiv 1 \pmod{r}$ **then** ▷ entonces ese es el r que buscamos
        $bool2 \leftarrow False$
      end if
      $k \leftarrow k + 1$
    end while
    if $bool2 = True$ **then**
      $bool \leftarrow False$
    end if
  end while ▷ Aquí ya tendríamos el r
  for $1 \leq p \leq \sqrt{\varphi(r)} \log_2(n)$ **do** ▷ Vemos que se cumple otra de las hipótesis
    if p primo y $p|n$ **then**
      return False
    end if
  end for
  for $1 \leq a \leq \sqrt{\varphi(r)} \log_2(n)$ **do** ▷ Comenzamos con el cálculo de las congruencias
    if $(x + a)^n \not\equiv x^n + a \pmod{x^r - 1, n}$ **then**
      return False ▷ Si alguna falla, significa que n no es primo
    end if
  end for
  return True

2.2. Pruebas probabilísticas

Comenzamos esta sección con una de las pruebas más simples y conocidas, la prueba de Fermat. Tras esto presentamos con la prueba de Sovolay-Strassen, para lo cual introduciremos el símbolo de Jacobi, una extensión del símbolo de Legendre muy utilizada en teoría de números. Acto seguido describimos el test de Miller-Rabin, uno de los más utilizados en la práctica. Finalizamos el capítulo con el test Baillie–PSW que, como veremos, se basa en una combinación de otras pruebas de primalidad.

2.2.1. Prueba de Fermat

De acuerdo con el Pequeño Teorema de Fermat (Teorema 1.1), sabemos que si n es primo entonces se cumple que

$$a^{n-1} \equiv 1 \pmod{n} \tag{2.1}$$

para cualquier a tal que $\gcd(a,n)=1$.

Sabemos que el recíproco no es cierto, sabemos que si se cumple la igualdad anterior entonces n puede no ser primo.

Definición 2.1. Sean n un entero compuesto e impar y a un entero tal que $\gcd(a,n)=1$. Si se cumple que $a^{n-1}\equiv 1 \pmod{n}$ se dice que n es un pseudoprimo de Fermat para la base a.

Con la información de la que disponemos hasta ahora sabemos que si un número no cumple la igualdad (2.1), entonces es compuesto. Por desgracia, utilizando este resultado no podemos determinar si es primo por muchas veces que se cumpla la igualdad para diferentes bases.

Queremos saber cual es la probabilidad estimada de que, cumpliéndose (2.1) para un cierto número de bases, este sea primo. Para ello nos ayudamos del siguiente resultado.

Proposición 2.2. *Sea n un entero compuesto e impar.*

1) *El entero n es un pseudoprimo de Fermat para la base a, con $\gcd(a,n)=1$, si y solo si el orden de a en el grupo $\mathbb{Z}_n^*$ divide a $n-1$.*

2) *Si n es un pseudoprimo de Fermat para las bases a_1 y a_2, donde $\gcd(a_1,n)=\gcd(a_2,n)=1$, entonces lo es para las bases a_1a_2 y $a_1a_2^{-1}$.*

3) *Si n no pasa el test para una determinada base a, de nuevo $\gcd(a,n)=1$, entonces no lo pasa para al menos la mitad de las posibles bases en $\mathbb{Z}_n$.*

Demostración. 1) Es claro que si $a^{n-1}\equiv 1 \pmod{n}$, entonces por el Teorema de Lagrange de grupos el orden de a en $\mathbb{Z}_n^*$ debe dividir a $n-1$. Recíprocamente es trivial que si el orden de a divide a $n-1$, digamos que $n-1 = o_n(a)\cdot k$ para cierto entero k, entonces $a^{n-1}\equiv (a^{o_n(a)})^k \equiv 1 \pmod{n}$.

2) Sabemos que $a_i^{n-1}\equiv 1 \pmod{n}$ para $i\in\{1,2\}$. Por tanto vemos que $(a_1a_2)^{n-1}\equiv a_1^{n-1}a_2^{n-1}\equiv 1 \pmod{n}$. Por otra parte es conocido que el orden de el elemento inverso de un elemento de un grupo es igual al de este. Sabiendo esto $(a_1a_2^{-1})^{n-1}\equiv a_1^{n-1}(a_2^{-1})^{n-1}\equiv 1 \pmod{n}$.

3) Sea $\{a_1,\dots,a_s\}$ el conjunto de todos los enteros para los que n es pseudoprimo de Fermat. Sea ahora b una base para la que n no es pseudoprimo. Si n fuera pseudoprimo para alguna de las bases ba_i, entonces por 2) sería pseudoprimo para b, ya que $b\equiv (ba_i)a_i^{-1} \pmod{n}$, pero esto no ocurre. Por tanto para $\{ba_1,\dots,ba_s\}$, n no pasa el test, por tanto podemos concluir que existen al menos las mismas bases para las que n es pseudoprimo que para las que no. □

Podemos decir entonces, que a menos que n sea pseudoprimo para todos los $a\in\mathbb{Z}_n$ tal que $\gcd(a,n)=1$ tenemos una probabilidad del 50 % de que n no pase el test para una base aleatoria. Si consideramos sucesos independientes un número k de elecciones aleatorias de bases entonces la probabilidad de que sea un pseudoprimo de Fermat para todas ellas es 2^{-k}. Todo esto a no ser que cumpla que es pseudoprimo de Fermat para todas las bases, pero este caso lo discutimos después.

Ahora describimos la prueba de Fermat, que sabiendo todo esto es fácil de concretar. Esta consiste en elegir k números enteros aleatorios en el intervalo $[2,n-1]$. Estas serán las bases de las que hemos estado hablando. Si para alguna de ellas a no se cumple la igualdad modular (2.1), entonces podemos concluir que es compuesto. No obstante, si la cumple para

las k bases elegidas entonces podemos decir que n es un posible primo de Fermat. Es obvio por lo mencionado anteriormente que a más grande sea k, mayor será la probabilidad de que n sea primo.

En el Algoritmo 13 podemos ver el pseudocódigo de la prueba. Como entradas se requieren el número n al que se le realizará la prueba y el número de bases k con las que probaremos el test.

Algorithm 13 Prueba de Fermat

Input: $n, k \in \mathbb{Z}, k > 0$
  for $1 \leq j \leq k$ **do**
    $a \leftarrow random[2, n-1]$ ▷ Número aleatorio entre 2 y n-1
    if $a^{n-1} \not\equiv 1 \pmod{n}$ **then**
      return False ▷ Si no pasa el test es compuesto
    end if
  end for
  return True ▷ Pasa todos los test, entonces es probable primo

Seguimos ahora con unos ejemplos de aplicación de la prueba.

Ejemplo 2.3.

- Vamos a comprobar la primalidad del número $n = 7109$. Probamos a aplicar el test tomando la base $a = 2$. Utilizando el algoritmo de la exponenciación binaria, cuyo código encontramos en el repositorio de la memoria, obtenemos que $2^{n-1} \equiv 1 \pmod{n}$, por lo que n es un probable primo para esta base. Si elegimos las bases 3, 5 y 11 se puede comprobar que obtenemos el mismo resultado. Sabiendo esto podemos decir que muy probablemente $n = 7109$ es primo. En efecto, se puede comprobar (utilizando algún test determinista) que de hecho lo es.

- Estudiamos ahora el caso del entero $n = 6071$. Si probamos de nuevo con $a = 2$, al realizar los cálculos nos queda que $a^{n-1} \equiv 5964 \pmod{n}$. Podemos concluir entonces que $n = 6071$ no es primo.

- Por último probamos el test con el caso del número $n = 561$. Si tomamos las bases $2, 5, 7$ y 13 vemos que pasa el test para todas y cada una de ellas. Esto nos diría que la probabilidad de que 561 sea primo es muy alta, sin embargo no lo es, ya que 3 es un divisor suyo.

Pasamos ahora a discutir el caso en el que n pasa el test para todas las bases a tales que $\gcd(a, n) = 1$.

Definición 2.2. Un número n de Carmichael es un entero compuesto que cumple que $a^{n-1} \equiv 1 \pmod{p}$ para todo entero a con $\gcd(a, n) = 1$. Notamos inmediatamente que esto es lo mismo que $a^n \equiv a \pmod{n}$ para todo entero a.

A continuación enunciamos y demostramos un resultado especialmente interesante sobre números de Carmichael que ha sido extraído de [CP05].

Proposición 2.3. *(Teorema de Korselt). Un entero n es un número de Carmichael si y solo si es positivo, compuesto, libre de cuadrados y para cada p divisor de n se tiene que $p - 1 | n - 1$.*

Demostración. Supongamos que n es un número de Carmichael, por tanto compuesto. Sea p un factor primo de n. De la igualdad $p^n \equiv p \pmod{n}$, vemos que p^2 no divide a n, lo que

nos dice que n está libre de cuadrados. Sea ahora a un elemento primitivo módulo p. Por una parte sabemos que $a^n \equiv a \pmod{n}$, por tanto $a^n \equiv a \pmod{p}$, por lo que vemos que $a^{n-1} \equiv 1 \pmod{p}$. Pero como a es primitivo módulo p, entonces sabemos que tiene orden $p-1$, por lo que $p-1|n-1$.

Recíprocamente supongamos que n es compuesto, libre de cuadrados y para cada factor primo de n se tiene que $p-1|n-1$. Nuestro propósito es ver que para todo entero a se cumple que $a^n \equiv a \pmod{n}$. Como n es libre de cuadrados basta con demostrar (gracias al Teorema Chino del Resto) que $a^n \equiv a \pmod{p}$ para cada entero a y cada primo p divisor de n. Supongamos por tanto que a es cualquier entero y que $p|n$. Si a no es divisible por p entonces $a^{p-1} \equiv 1 \pmod{p}$ y como por hipótesis $p-1|n-1$ obtenemos que $a^{n-1} \equiv 1 \pmod{p}$. Esto nos dice que $a^n \equiv a \pmod{p}$. Además esta congruencia también se cumple cuando $p|a$ por lo que conseguimos lo deseado. □

Hemos encontrado una caracterización para los números de Carmichael. Después de esto nos surge la pregunta, ¿existirá un número infinito de ellos? En 1994, los matemáticos William R. Alford, Andrew Granville y Carl Pomerance publicaron la respuesta a esta pregunta [AGP94], que desafortunadamente es sí. Como curiosidad los 4 primeros números de Carmichael son $561, 1105, 1729$ y 2465 (extraídos de [Wik22a]).

Sin embargo también consiguen dar una cota superior para la cantidad de números de Carmichael menores que un cierto número x. Esta nos permite concluir que son infinitos pero no muy abundantes, lo cual nos alivia, ya que estos números pasan el test para cualquier base (a no ser que tengamos suerte y elijamos una base a tal que $\gcd(a,n) > 1$) a pesar de no ser primos.

2.2.2. Prueba de Solovay-Strassen

Creado por Robert M. Solovay y Volker Strassen en 1977. En la actualidad este ha sido reemplazado por otros test más eficientes como la prueba de primalidad de Baillie-PSW y la prueba de primalidad de Miller-Rabin, que también estudiaremos en este capítulo.

Este test se basa también en el PTF (Teorema 1.1), el Criterio de Euler (Teorema 1.2) y además utiliza el símbolo de Jacobi, el cual introducimos a continuación.

Definición 2.3. (Símbolo de Jacobi). Supongamos que n es un entero compuesto e impar cuya descomposición en primos es

$$n = \prod_{i=1}^{k} p_i^{e_i}.$$

Sea a un entero. El símbolo de Jacobi $\left(\frac{a}{n}\right)$ se define como:

$$\left(\frac{a}{n}\right) = \prod_{i=1}^{k} \left(\frac{a}{p_i}\right)^{e_i}.$$

Nos percatamos de que el símbolo de Jacobi no es más que una generalización del símbolo de Legendre. Debido a esto, los propiedades vistas en la Proposición 1.5 también son válidas aquí de manera inmediata.

También son ciertas algunas generalizaciones para el símbolo de Jacobi, como el resultado que sigue y la Ley de reciprocidad cuadrática. La prueba de la Ley de reciprocidad cuadrática que se presenta se ha extraído de [Kob94].

Proposición 2.4. *Sea $n \in \mathbb{Z}$ impar. Se cumple que:*

$$\left(\frac{2}{n}\right) = (-1)^{(n^2-1)/8} = \begin{cases} -1 & si \quad n \equiv \pm 3 \pmod 8, \\ 1 & si \quad n \equiv \pm 1 \pmod 8. \end{cases}$$

Demostración. Definimos $f(n) = (-1)^{(n^2-1)/8}$. Solo tenemos que probar que $f(ab) = f(a)f(b)$ con a, b impares, ya que entonces la Proposición 1.5 garantiza el resultado . Basta ver que $\frac{a^2-1}{8} + \frac{b^2-1}{8} \equiv \frac{a^2b^2-1}{8} \pmod 2$.

Si $a = 2k+1$ es impar, como $k(k+1)$ es par tenemos que

$$a^2 = 4k^2 + 4k + 1 = 4k(k+1) + 1 = 8t + 1 \equiv 1 \pmod 8.$$

La penúltima igualdad es debida a la paridad de $k(k+1)$, por tanto existe un entero t cumpliendo lo escrito.

Entonces, si a, b son impares, $a^2 - 1$, $b^2 - 1$, $a^2b^2 - 1$ y $\frac{(a^2-1)(b^2-1)}{8}$ son múltiplos de 8. Tenemos entonces que:

$$\frac{a^2b^2-1}{8} - \left(\frac{a^2-1}{8} + \frac{b^2-1}{8}\right) = \frac{(a^2-1)(b^2-1)}{8} \equiv 0 \pmod 8.$$

Al ser 8 múltiplo de dos tenemos que $\frac{a^2b^2-1}{8} \equiv \left(\frac{a^2-1}{8} + \frac{b^2-1}{8}\right) \pmod 2$. □

Teorema 2.7. *(2^a Ley de reciprocidad cuadrática). Si m, n son dos enteros distintos e impares, entonces:*

$$\left(\frac{m}{n}\right)\left(\frac{n}{m}\right) = (-1)^{(m-1)(n-1)/4}.$$

O lo que es lo mismo:

1) $\left(\frac{m}{n}\right) = -\left(\frac{n}{m}\right)$ *si* $m, n \equiv 3 \pmod 4$.

2) $\left(\frac{m}{n}\right) = \left(\frac{n}{m}\right)$ *si* $m \equiv 1 \pmod 4$ *o* $n \equiv 1 \pmod 4$.

Demostración. Primero notamos que si $\gcd(m, n) > 1$, entonces se sigue de la definición símbolo de Legendre y de Jacobi que $\left(\frac{m}{n}\right) = \left(\frac{n}{m}\right) = 0$.

Supongamos entonces que $\gcd(m, n) = 1$. Sean $m = \prod_{i=1}^{r} p_i$ y $n = \prod_{j=1}^{s} q_j$ las correspondientes factorizaciones en primos (pueden existir factores repetidos si m o n no son libres de cuadrados).

Entonces sabemos, por definición, que $\left(\frac{m}{n}\right) = \prod_{i,j=1}^{r,s} \left(\frac{p_i}{q_j}\right)$ y $\left(\frac{n}{m}\right) = \prod_{i,j=1}^{r,s} \left(\frac{q_j}{p_i}\right)$.

Para pasar de una a otra debemos aplicar la Ley de reciprocidad cuadrática (Teorema 1.3) rs veces. El número de (-1) que obtenemos es justo el número de veces que $p_i, q_j \equiv 3 \pmod 4$, o lo que es lo mismo, el producto del número de primos congruentes con $3 \pmod 4$ en las factorizaciones de m y n. Vemos entonces que $\left(\frac{m}{n}\right) = \left(\frac{n}{m}\right)$ a menos que el número de primos congruentes a $3 \pmod 4$ en ambas factorizaciones es impar, en cuyo caso $\left(\frac{m}{n}\right) = -\left(\frac{n}{m}\right)$.

Pero números que son producto de impares como m y n son congruentes con $3 \pmod 4$ si y solo si contienen en su factorización un número impar de primos congruentes con 3 $\pmod 4$. Por tanto, concluimos que $\left(\frac{m}{n}\right) = \left(\frac{n}{m}\right)$ a menos que $m, n \equiv 3 \pmod 4$, como queríamos probar. □

Teniendo en cuenta estos resultados acerca del símbolo de Jacobi podemos dar un método de cálculo del mismo, lo exponemos en el Algoritmo 14. Lo hacemos utilizando la recursividad, ya que el algoritmo se presta a que así sea.

Este algoritmo se puede encontrar en el documento *algoritmos_complementarios* de nuestro repositorio.

Algorithm 14 Símbolo de Jacobi

Input: $a, n \in \mathbb{Z}$ impares
  $sol \leftarrow 0$
  if $a = 0$ **then**
    if $n = 1$ **then**
      return 1
    else
      return 0 ▷ Por la Proposición 1.5
    end if
  end if
  if $a = 2$ **then** ▷ Aplicamos la Proposición 2.4
    $r \leftarrow n \pmod 8$
    if $r = 7$ o $r = 1$ **then**
      return 1
    end if
    if $r = 5$ o $r = 3$ **then**
      return -1
    end if
  end if
  if $a \geq n$ **then**
    $sol \leftarrow Jacobi(a \pmod n, n)$ ▷ Por la Proposición 1.5
  else
    if $a \equiv 0 \pmod 2$ **then**
      $sol \leftarrow Jacobi(2, n) \cdot Jacobi(a/2, n)$ ▷ De nuevo por la Proposición 1.5
    else ▷ Aplicamos la Ley de reciprocidad cuadrática
      if $a \equiv 3 \pmod 4$ o $n \equiv 3 \pmod 4$ **then**
        $sol \leftarrow -Jacobi(n, a)$
      else
        $sol \leftarrow Jacobi(n, a)$
      end if
    end if
  end if
  return sol

Después de este importante inciso vamos con la prueba. Si en la Definición 2.3 hacemos que n sea primo, obtenemos el símbolo de Legendre, por lo que atendiendo al Criterio de Euler (Teorema 1.2) que

$$a^{(n-1)/2} \equiv \left(\frac{a}{n}\right) \pmod n. \tag{2.2}$$

Sin embargo, no se cumple el recíproco, es decir, si se cumple la identidad anterior no podemos concluir que n sea primo.

De aquí podemos obtener un algoritmo probabilístico que nos permita concluir si un dado número es primo o no. Empezamos con la siguiente definición.

Definición 2.4. Sea n un entero compuesto e impar y a un entero tal que $\gcd(a, n) = 1$. Diremos que n es un pseudoprimo de Euler para la base a si se cumple (2.2).

El resultado que se expone a continuación afirma que, a diferencia que ocurría con los pseudoprimos de Fermat, no existe un análogo a los números de Carmichael para los pseudoprimos de Euler. La mayor parte de la demostración que se puede ver a continuación ha sido extraída de [Pla].

Teorema 2.8. *(Solovay-Strassen). Sea $n \geq 3$ un entero compuesto e impar. Entonces al menos la mitad de los elementos de $\mathbb{Z}_n^*$ no satisfacen la igualdad* (2.2).

Demostración. Lo haremos en tres pasos. Primero probaremos que si n es un pseudoprimo de Euler para la base a, pero pasa el test para otra base b, entonces n es pseudoprimo de Euler para la base ab. Tras esto veremos que esto implica que si existe al menos una base para el que n es pseudoprimo de Euler entonces lo es para el menos la mitad de los elementos de $\mathbb{Z}_n^*$. Y por último probaremos que siempre existe tal base para el que n es pseudoprimo de Euler.

Para probar lo primero, supongamos que n no pasa el test para la base a, entonces $a^{(n-1)/2} \equiv -\left(\frac{a}{n}\right) \pmod{n}$. Veamos que pasa con la base ab.

$$(ab)^{(n-1)/2} \pmod{p} \equiv a^{(n-1)/2} b^{(n-1)/2} \pmod{p} = -\left(\frac{a}{n}\right)\left(\frac{b}{n}\right) = -\left(\frac{ab}{n}\right).$$

Por lo que vemos que se cumple lo perseguido.

Vamos con la segunda parte. Supongamos que existe una base a tal que n es pseudoprimo de Euler. Entonces si para otra base b si que pasa el test, sabemos por la primera parte que n falla el test para la base ab, por lo que al menos hay tantas bases para las que n pasa el test como para las que no.

Solo nos falta la última parte, ver que siempre existe un elemento para el que el test falla. Para ello distinguiremos dos casos, cuando n está libre de cuadrados y cuando no.

Comenzamos suponiendo que está libre de cuadrados (basta con suponer que en su factorización hay un primo con multiplicidad 1). Sea entonces p un factor primo de n (con multiplicidad 1) y sea b un no-residuo cuadrático módulo n. Usando el Teorema Chino del Resto elegimos un a tal que:

$$\begin{aligned} a &\equiv b \pmod{p}, \\ a &\equiv 1 \pmod{\frac{n}{p}}. \end{aligned}$$

El símbolo de Jacobi lo podemos calcular como sigue:

$$\left(\frac{a}{n}\right) = \left(\frac{a}{p}\right)\left(\frac{a}{n/p}\right) = (-1) \cdot 1 = -1.$$

Ahora supondremos que $a^{(n-1)/2} \equiv -1 \pmod{n}$ y llegaremos a una contradicción. Esta suposición implica que $a^{(n-1)/2} \equiv -1 \pmod{\frac{n}{p}}$. Sin embargo, como $a \equiv 1 \pmod{\frac{n}{p}}$, entonces $a^{(n-1)/2} \equiv 1 \pmod{\frac{n}{p}}$. Llegamos así a que $1 \equiv -1 \pmod{\frac{n}{p}}$.

Para terminar vemos el caso en el que n tiene factores primos repetidos. Sea p primo tal

que $p^2|n$ y consideramos $a = 1 + \frac{n}{p}$. Por el binomio de Newton:

$$a^p = \left(1 + \frac{n}{p}\right)^p = 1 + \binom{p}{1}\left(\frac{n}{p}\right) + \binom{p}{2}\left(\frac{n}{p}\right)^2 + \cdots + \binom{p}{p}\left(\frac{n}{p}\right)^p \equiv 1 \pmod{n}.$$

Esto nos dice que el orden multiplicativo de a módulo n es p, por lo que $a^{n-1} \not\equiv 1 \pmod{n}$, ya que p no divide a $n-1$. Por otra parte,

$$\left(\frac{a}{n}\right) = \left(\frac{1+n/p}{n}\right) = \left(\frac{1+n/p}{p}\right)\left(\frac{1+n/p}{n/p}\right) = \left(\frac{1}{p}\right)\left(\frac{a}{n/p}\right) = 1.$$

Podemos concluir entonces que n es un pseudoprimo de Euler para esta base, que es lo que buscábamos. □

A la luz de este teorema vemos que efectivamente aquí no disponemos de algo similar a los números de Carmichael, lo cual es una buena noticia. También hemos logrado ver que, al cabo de k elecciones aleatorias de bases, obtenemos que la probabilidad de que sea un pseudoprimo de Euler para todas ellas es de 2^{-k}.

Podemos ahora describir en que consiste la prueba de Solovay-Strassen. Esta consiste en elegir un número k de bases aleatorias en el intervalo $[2, n-1]$ y comprobar si se cumple la congruencia (2.2). Si no se cumple para alguna base podemos asegurar que n es compuesto. En cambio, sabemos que si se cumple no podemos decir que sea primo, lo único que podemos decir si n pasa el test para todas las bases elegidas es que es un probable primo.

El pseudocódigo de la prueba es como sigue. Las entradas pedidas son el número n sobre el que se quiere averiguar la primalidad y el número de bases aleatorias con las que se quiera probar el test.

Algorithm 15 Prueba de Solovay-Strassen

Input: $n, k \in \mathbb{Z}, k > 0$
  for $1 \leq j \leq k$ **do**
    $a \leftarrow random[2, n-1]$ ▷ Número aleatorio entre 2 y n-1
    $x \leftarrow Jacobi(a, n)$
    if $a^{(n-1)/2} \not\equiv x \pmod{n}$ **then**
      return False ▷ Si no pasa el test es compuesto
    end if
  end for
  return True ▷ Pasa todos los test, entonces es probable primo

Procedemos ahora a probar el algoritmo con algunos ejemplos. Los cálculos, como de costumbre, se han realizado usando el código en Python de la prueba que se puede encontrar en el repositorio.

Ejemplo 2.4. ▪ Comprobamos la primalidad del entero 24371 usando esta prueba. Obtenemos los siguientes resultados para 6 bases aleatorias.

a	$\left(\frac{a}{n}\right)$	$a^{(n-1)/2}$
5540	-1	-1
20605	1	1
19952	-1	-1
260	1	1
5602	-1	-1
842	1	1

A la vista de esto podemos afirmar que muy probablemente $n = 24371$ sea primo (con una probabilidad de al menos $1 - 2^{-6} = 0.984375$), y de hecho lo es.

- Probamos con un entero de mayor tamaño, por ejemplo $n = 25630757$. Los resultados son los siguientes:

a	$\left(\frac{a}{n}\right)$	$a^{(n-1)/2}$
2994460	1	1
17298765	1	1
22324763	1	1
14850294	1	1
509506	-1	-1
19407003	1	1

El test nos indica de nuevo que la probabilidad de que n sea primo es grande (0.984375), y efectivamente así es.

- Vemos el caso de $n = 91$, que es pseudoprimo de Euler para las bases $9, 10$ y 17 y sin embargo no para la base 11. Se tiene lo siguiente:

a	$\left(\frac{a}{n}\right)$	$a^{(n-1)/2}$
9	1	1
10	-1	-1
17	-1	-1
11	-1	8

Por tanto 91 es compuesto. Esto nos indica la importancia de probar con bases distintas, ya que podemos concluir de manera errónea que cierto número es primo cuando no lo es

2.2.3. Test de Miller-Rabin

Este test, también llamado de pseudoprimos fuertes, fue propuesto por M. Rabin en 1980. Este surge como una modificación de un algoritmo determinista propuesto por G. Miller en 1975, que asumiendo la no demostrada Hipótesis de Riemann generalizada se ejecutaba en tiempo polinomial. En la actualidad se trata de uno de los test de primalidad más utilizados por su eficacia y baja probabilidad de error. Es debido a esto que es el test estándar que incorporan la mayoría de paquetes informáticos.

Describiremos la mayor parte de la teoría necesaria para el test ayudándonos de lo estudiado en [Yan02].

Comenzamos con una proposición básica pero fundamental para el test.

Proposición 2.5. *Si p es un entero primo, entonces la ecuación*

$$x^2 \equiv 1 \pmod{p}$$

tiene como soluciones $x \equiv \pm 1 \pmod{p}$.

Demostración. Como $x^2 \equiv 1 \pmod{p}$, esto es lo mismo que $(x-1)(x+1) \equiv 0 \pmod{p}$. Esto implica que $p|(x-1)(x+1)$, por lo que, como p es primo, $p|(x-1)$ o $p|(x+1)$. Entonces $x - 1 \equiv 0 \pmod{p}$ o $x + 1 \equiv 0 \pmod{p}$. Finalmente esto nos hace ver que $x \equiv 1 \pmod{p}$ o $x \equiv -1 \pmod{p}$.

También podemos ver esto como consecuencia de que $\mathbb{Z}_p$ es un cuerpo cuando p es primo, y por tanto la ecuación $x^2 \equiv 1 \pmod{p}$ solo admite dos soluciones. □

Esto implica que si existe una ecuación distinta de ± 1 (no trivial) a la ecuación $x^2 \equiv 1 \pmod{p}$ entonces p es compuesto.

Vamos con un resultado que es consecuencia directa de esta proposición y es la piedra angular del test de Miller-Rabin.

Teorema 2.9. *Sea n un entero primo y tal que $n - 1 = 2^j d$, con d impar y j un entero positivo. Definimos la a-sucesión de n como el siguiente conjunto:*

$$\left\{ a^d \pmod{n}, a^{2d} \pmod{n}, a^{2^2 d} \pmod{n}, \ldots, a^{2^{j-1} d} \pmod{n}, a^{2^j d} \pmod{n} \right\}.$$

Se tiene entonces que la recién definida a-sucesión tiene una de las dos formas siguientes:

$$\{\pm 1, 1, \ldots, 1, 1\}$$

$$\{?, ?, \ldots, ?, -1, 1, \ldots, 1\}$$

para cualquier $1 < b < n$. (El símbolo '?' denota cualquier número distinto de ± 1.)

Demostración. Por el PTF (Teorema 1.1) sabemos que al ser n primo el último término de la a-sucesión es 1. Entonces por la Proposición 2.5, sabemos que lo precede un 1 o un -1. En el caso que sea un 1 procedemos razonando de la misma manera con el término anterior. Si por lo contrario, es un -1 entonces el término de la a-sucesión que lo precede es distinto de ± 1, y por tanto todos los anteriores a este también lo son. Siguiendo este razonamiento tenemos el teorema. □

Sabemos entonces que si la a-sucesión de un dado $n \in \mathbb{Z}$ no se corresponde con ninguno de los dos modelos anteriores podemos concluir que n es compuesto. Sin embargo, como venimos advirtiendo en todos los test probabilísticos, si coincide con alguna de los dos patrones anteriores no podemos confirmar que se trate de un número primo, aunque veremos que la probabilidad es bastante alta.

Podemos ahora describir el test de Miller-Rabin. Supongamos que queremos determinar la primalidad de un entero n. El primer paso es descomponer $n - 1 = 2^j d$. Tras esto elegimos de manera aleatoria una base a en el intervalo $[2, n-1]$. Seguidamente calculamos $a^d \pmod{n}$. Si obtenemos ± 1 podemos concluir por el Teorema 2.9 que n es un probable primo y pasamos a elegir otro a aleatorio . Sin embargo, si $a^d \not\equiv \pm 1 \pmod{n}$ debemos seguir construyendo la a-sucesión, es decir, calculando $a^{2^i d} \pmod{n}$, con $i \in \{1, 2, \ldots, j-1\}$. Si en el proceso obtenemos un -1 y no estamos en la última posición, entonces concluimos que n es un

probable primo, de nuevo por el Teorema 2.9. Si nunca conseguimos un -1 podemos asegurar que n es compuesto.

Definición 2.5. Un entero compuesto n es llamado un pseudoprimo fuerte para la base a si ha pasado el test de Miller-Rabin para la base a.

Es fácil ver que que ser pseudoprimo fuerte implica ser pseudoprimo de Fermat. Sin embargo, no es trivial probar que pseudoprimo fuerte implica pseudoprimo de Euler. La demostración de este hecho es algo complicada y se puede encontrar en [Kob94].

De lo anterior deducimos que el test de Miller-Rabin funciona mejor que el test de Solovay-Strassen y el test de Fermat. Pero queremos cuantificar esta mejoría en términos probabilísticos. Esto nos lo da el siguiente teorema.

Teorema 2.10. *Sea $n > 1$ un entero compuesto e impar. Se tiene entonces que n pasa el test para, como mucho, $(n-1)/4$ bases a, con $1 \leq a < n$.*

Demostración. La demostración es compleja y extensa, por lo que daremos una aproximación de la misma extraída de [Yan02]. Esta se puede ver completa en [CP05].

Se puede probar que dados un primo impar p y α y q enteros positivos, el número de soluciones no congruentes de la ecuación

$$x^{q-1} \equiv 1 \pmod{p^\alpha}$$

es $\gcd(q, p^{\alpha-1}(p-1))$.

Sea $n-1 = 2^j d$, donde d es un entero impar y j un entero positivo. Supongamos que n es un pseudoprimo fuerte para la base b, entonces

$$b^d \equiv \pm 1 \pmod{n},$$

o bien

$$b^{2^i d} \equiv -1 \pmod{n},$$

para $0 < i < j-1$. En cualquier caso se tiene que $b^{n-1} \equiv 1 \pmod{n}$. Sea ahora la factorización de n como sigue:

$$n = p_1^{\alpha_1} p_2^{\alpha_2} \dots p_k^{\alpha_k}.$$

Sabemos entonces por lo expuesto al principio de la prueba que existen

$$\gcd(n-1, p_i^{\alpha_i}(p_i-1)) = \gcd(n-1, p_i - 1)$$

soluciones incongruentes a la ecuación $x^n - 1 \equiv 1 \pmod{p_i^{\alpha_i}}$.

Por el Teorema Chino del Resto sabemos que hay exactamente

$$\prod_{i=1}^{k} \gcd(n-1, p_i - 1) \tag{2.3}$$

soluciones de la ecuación $x^{n-1} \equiv 1 \pmod{n}$.

Para probar el teorema se distinguen dos casos:

- La factorización de n contiene una potencia de un primo con exponente mayor o igual que dos.

- La factorización de n esta formada por primos distintos.

A partir de aquí se usa (2.3) para probar lo deseado aplicándolo a cada uno de los casos. □

Esto nos dice que la probabilidad de que un entero compuesto e impar pase el test para una determinada base es $1/4$. Si consideramos independientes las elecciones aleatorias de las bases podemos afirmar que si nuestro entero pasa el test para k bases distintas, la probabilidad de que sea primo es $1 - 4^{-k}$, por lo que con pocas bases podemos conseguir una probabilidad muy alta.

Presentamos en la siguiente página el pseudocódigo (Algoritmo 16) del test de Miller-Rabin. Como entradas se requerirán el número al que se le aplicará el test y un entero positivo que marcará el número de bases aleatorias para las que se desea que n pase el test.

Mostramos también el funcionamiento de las distintas etapas del test, ayudándonos como siempre del código en Python del repositorio, aplicándolo a distintos números en un ejemplo.

Ejemplo 2.5.

- Vemos que ocurre con $n = 4657$. Observamos que $n - 1 = 2^4 \cdot 291$. Si en el Algoritmo 16 elegimos $k = 6$ el resultado es el siguiente:

a	$a^{291} \pmod n$	$a^{2\cdot 291} \pmod n$	$a^{2^2\cdot 291} \pmod n$	$a^{2^3\cdot 291} \pmod n$
1957	-1	–	–	–
3616	2202	867	1912	-1
4064	-1	–	–	–
2436	497	188	2745	-1
439	4469	2745	-1	–
985	188	2745	-1	–

Como consecuencia el test devuelve que $n = 4657$ es un posible primo, y ciertamente este lo es.

- Seguimos con $n = 1541$, por tanto $n - 1 = 2^2 \cdot 385$. En este caso elegimos $k = 3$ obteniendo lo siguiente:

a	$a^{385} \pmod n$	$a^{2\cdot 385} \pmod n$
1348	-1	–
713	736	805

Concluimos entonces que n es compuesto.

- Por último vemos estudiamos el caso $n = 21037369$, con $n - 1 = 2^3 \cdot 2629671$. Elegimos $k = 6$ y la salida es:

a	$a^{2629671} \pmod n$	$a^{2\cdot 2629671} \pmod n$	$a^{2^2\cdot 2629671} \pmod n$
20372818	1	–	–
2068518	13603672	5179166	-1
14583472	13603672	5179166	-1
17447029	1	–	–
2678119	-1	–	–
6618812	7433697	5179166	-1

Podemos concluir entonces que n es un probable primo, y efectivamente se puede comprobar que lo es.

Algorithm 16 Test de Miller-Rabin

Input: $n, k \in \mathbb{Z}, k > 0$
  $j \leftarrow 0$
  $d \leftarrow n - 1$
  while $d \equiv 0 \pmod 2$ **do** ▷ Descomponemos $n-1$ de la manera deseada
    $j \leftarrow j + 1$
    $d \leftarrow d/2$
  end while
  for $1 \leq r < k$ **do** ▷ Repetimos el proceso k veces
    $a \leftarrow random[2, n-1]$ ▷ Elección de base aleatoria
    $e \leftarrow a^d \pmod n$
    if $e \neq \pm 1$ **then** ▷ Si $e = \pm 1$ entonces pasamos a la siguiente base
      $i \leftarrow 1$
      $bool \leftarrow True$
      while $i < j$ y $bool = True$ **do** ▷ Vemos si en la $a-$sucesión hay algún -1
        $e \leftarrow e^2 \pmod n$
        if $e = -1$ **then**
          $bool \leftarrow False$
        end if
        $i \leftarrow i + 1$
      end while
      if $bool = True$ **then** ▷ Si no hay ningún -1 entonces n es compuesto
        return False
      end if
    end if
  end for
  return True ▷ Ha pasado el test para todas las bases y entonces es posible primo

2.2.4. Test de Baillie-PSW

El nombre de la prueba se debe a Robert Baillie, Carl Pomerance, John L. Selfridge y Samuel S. Wagstaff. Al igual que la prueba de Miller-Rabin, también es una de las más utilizadas en la actualidad. Esta consiste en una combinación entre una prueba de Miller-Rabin para la base 2 y una prueba fuerte de Lucas.

La prueba surge a partir de [BW80], donde se introduce el concepto de pseudoprimo fuerte de Lucas y se observa que los primeros 50 números de Carmichael y otros muchos pseudoprimos fuertes no eran pseudoprimos fuertes de Lucas. Es por esto que en [PSW80] los autores deciden combinar ambas pruebas para obtener un nuevo test de primalidad.

Pasamos a describir la prueba fuerte de Lucas. Para ello debemos introducir las sucesiones de Lucas. Lo haremos siguiendo el camino marcado por [Yan02].

Sean $P, Q \in \mathbb{Z}^*$ y $D = P^2 - 4Q$. Consideramos ahora la ecuación $x^2 - Px + Q = 0$. Su discriminante es D y supongamos que α y β son sus dos soluciones:

$$\alpha = \frac{P + \sqrt{D}}{2},$$
$$\beta = \frac{P - \sqrt{D}}{2},$$

por tanto:

$$\begin{aligned} \alpha + \beta &= P, \\ \alpha\beta &= Q. \end{aligned}$$

Definimos ahora las sucesiones:

$$\begin{aligned} U_k(P,Q) &= \frac{\alpha^k - \beta^k}{\alpha - \beta}, \\ V_k(P,Q) &= \alpha^k + \beta^k. \end{aligned}$$

En particular, $U_0(P,Q) = 0, U_1(P,Q) = 1$ y $V_0(P,Q) = 2, V_1(P,Q) = P$. Para $k \geq 2$ se tiene:

$$\begin{aligned} U_k(P,Q) &= PU_{k-1} - QU_{k-2}, \\ V_k(P,Q) &= PV_{k-1} - QV_{k-2}. \end{aligned}$$

Las sucesiones:

$$\begin{aligned} U(P,Q) &= (U_k(P,Q))_{k\geq 0}, \\ V(P,Q) &= (U_k(P,Q))_{k\geq 0}, \end{aligned}$$

se denominan sucesiones de Lucas asociadas al par (P,Q).

Podemos obtener rápidamente un método de cálculo de U_{2k} y V_{2k} a partir de U_k y V_k (algo que nos será útil posteriormente). Lo hacemos de la siguiente forma:

$$U_{2k} = \frac{\alpha^{2k} - \beta^{2k}}{\alpha - \beta} = U_k V_k, \tag{2.14}$$

$$V_{2k} = \alpha^{2k} + \beta^{2k} = V_k^2 - 2\alpha^k\beta^k = V_k^2 - 2Q^k.$$

Presentamos ahora el teorema fundamental sobre el que se construye la prueba de Lucas.

Teorema 2.11. *(Teorema de Lucas). Sean n un primo impar, $P, Q \in \mathbb{Z}$ y consideramos $D = P^2 - 4Q$. Además, sea $\varepsilon(n)$ el símbolo de Jacobi $\left(\frac{D}{n}\right)$ y $\delta(n) = n - \varepsilon(n)$. Entonces, si $\gcd(n,Q) = 1$, entonces se cumple la siguiente igualdad modular:*

$$U_{\delta(n)} \equiv 0 \pmod{n}. \tag{2.15}$$

Demostración. La demostración requiere algo más de teoría sobre las sucesiones de Lucas y se puede consultar en [CP05]. □

Definición 2.6. Sea $n \in \mathbb{Z}$ compuesto e impar, diremos que n es un pseudoprimo de Lucas para el par (P,Q) cuando se cumple la igualdad (2.15).

Manteniendo la misma filosofía que en la definición de pseudoprimo fuerte tenemos el siguiente teorema, vital para la prueba fuerte de Lucas.

Teorema 2.12. *Sean n un primo impar, $P, Q \in \mathbb{Z}$ y consideramos $D = P^2 - 4Q$. Además, sea $\varepsilon(n)$ el símbolo de Jacobi $\left(\frac{D}{n}\right)$ y $\delta(n) = n - \varepsilon(n) = 2^j d$ para ciertos $j, d \in \mathbb{Z}$. Si $\gcd(n,Q) = 1$, entonces se satisface una de las condiciones siguientes:*

- $U_d \equiv 0 \pmod{n}$,
- $V_{2^r d} \equiv 0 \pmod{n}$, *para algún* $0 \leq r < s$.

Demostración. Por el Teorema 2.11 sabemos que $U_{\delta(n)} \equiv 0 \pmod{n}$. Como n es primo esto implica, gracias a (2.14), que $U_{2^{j-1}d} = 0 \pmod{n}$ o $V_{2^{j-1}d} = 0 \pmod{n}$. Si $V_{2^{j-1}d} = 0 \pmod{n}$ hemos terminado, si no es así seguiríamos dividiendo por 2 y aplicando (2.14) hasta llegar a que $U_d \equiv 0 \pmod{n}$ o $V_d \equiv 0 \pmod{n}$ por lo que llegaríamos a lo deseado. □

Definición 2.7. Sea $n \in \mathbb{Z}$ compuesto e impar, y supongamos que $\delta(n) = 2^s d$ con d impar ($\delta(x)$ como en el Teorema 2.11), diremos que es un pseudoprimo fuerte de Lucas para el par (P, Q) si se cumple una de las condiciones siguientes:

- $U_d \equiv 0 \pmod{n}$.
- $V_{2^r d} \equiv 0 \pmod{n}$, para algún $0 \leq r < s$.

Conociendo esto, el lector puede vislumbrar cual será el funcionamiento del test fuerte de Lucas para un entero n, el cual es como sigue:

1) El primer paso es elegir un entero D tal que $\left(\frac{D}{n}\right) = -1$ y a partir de este formar el par (P, Q). Para llevar a cabo esto, en [PSW80] se dan dos procedimientos. Aquí adoptaremos el preferido de Selfridge. Este consiste en elegir D como el primer entero en la sucesión $\{5, -7, 9, -11, \dots\}$ tal que $\left(\frac{D}{n}\right) = -1$ y a partir de este definir $P = 1$ y $Q = (1 - D)/4$.
2) Ahora podemos construir las sucesiones de Lucas asociadas a (P, Q). Si escribimos $\delta(n) = n + 1 = 2^s d$ tendremos que calcular la sucesión U hasta el término d, si este es 0 módulo n devolvemos que n es un probable primo fuerte de Lucas.
3) En otro caso, comprobamos si $V_{2^r d} \equiv 0 \pmod{n}$ para algún $0 \leq r < s$, si es así, de nuevo devolvemos que n es un probable primo fuerte de Lucas. Si no es así, devolvemos que n es compuesto.

Para el test fuerte de Lucas también existe un resultado análogo a el Teorema 2.10. Dicha probabilidad es a lo sumo $\frac{4n}{15}$, siendo n el número al que se le aplica el test. Este resultado con la respectiva prueba, que es compleja y larga, se puede encontrar en [Arn97].

Expresamos esto en forma de pseudocódigo en el Algoritmo 18. En el mismo se puede observar que el cálculo de las sucesiones de Lucas no es ni por asomo el más eficiente. Sin embargo, el código en Python del repositorio se ha programado intentando hacerlo lo más eficiente posible, por lo que para calcular los términos U_d y V_d antes mencionados se ha seguido otro procedimiento distinto al descrito en el Algoritmo 18. El algoritmo utilizado se ha extraído de los apuntes de la asignatura *Teoría de números y criptografía* y se puede consultar en [Gar].

Presentamos ahora distintos ejemplos del funcionamiento de la prueba fuerte de Lucas.

Ejemplo 2.6.

- Comenzamos con el entero $n = 492343$. Vemos que $\left(\frac{5}{n}\right) = -1$, por lo que $D = 5$. Por tanto $P = 1$ y $Q = -1$, además $\delta(n) = n + 1 = 2^3 \cdot 61543$. Si calculamos la sucesión de Lucas hasta el término $d = 61543$ vemos que $U_d \equiv 191397 \pmod{n}$ y $V_d \equiv 173780 \pmod{n}$. Por tanto aún no podemos concluir nada.

Al calcular $V_{2\cdot d} \pmod n$ obtenemos el valor $153468 \neq 0$. Vemos que $V_{2^2\cdot d} = 214931 \pmod n$, también distinto de 0, por lo que concluimos que $n = 492343$ es compuesto.

- Consideramos ahora $n = 25630771$. Haciendo cálculos vemos que $\left(\frac{D}{n}\right) = 1$ para $D \in \{5, -7, 9\}$, pero $\left(\frac{-11}{n}\right) = -1$ por lo que tomamos $D = -11$. Así, $P = 1$ y $Q = 3$. Por otro lado $\delta(n) = n + 1 = 2^2 \cdot 6407693$. Al computar las sucesiones de Lucas obtenemos que, haciendo $d = 6407693$, $U_d \equiv 22928620 \pmod n$ y $V_d \equiv 15329387 \pmod n$. Por tanto no podemos decir nada todavía sobre la primalidad de n.

  Sin embargo al calcular V_{2d} obtenemos 7319843 módulo n, por lo que concluimos que n es compuesto.

- Por último vemos el caso de $n = 25630777$. Se puede comprobar que $\left(\frac{5}{n}\right) = -1$, por lo que elegimos $D = 5$, y por tanto $P = 1$ y $Q = -1$. Observamos que $\delta(n) = n + 1 = 2 \cdot 12815389$. Las sucesiones de Lucas en el término $d = 12815389$ tienen un valor de $U_d \equiv 0 \pmod n$ y $V_d \equiv 25004885 \pmod n$. Viendo esto podemos concluir que efectivamente n es un probable primo (se puede comprobar ciertamente lo es).

Como siempre los cálculos del ejemplo han sido posibles gracias al código que se puede consultar en el repositorio.

Tras introducir el test fuerte de Lucas ya nos encontramos con todas las herramientas necesarias para describir la prueba de primalidad de Baillie-PSW. Acto seguido mostramos los pasos que sigue la prueba:

1) Ejecutar la división por tentativa hasta un cierto límite para testear que n no tiene factores pequeños. Si tuviera alguno, devolvemos que n es compuesto, si no es así pasamos a el siguiente paso.

2) Aplicamos el test de Miller-Rabin a n tomando la base 2. Si el resultado es que n es compuesto hemos terminado.

3) Si por el contrario es posible primo, llevamos a cabo la prueba fuerte de Lucas y devolvemos el resultado.

Mostramos esto en forma de pseudocódigo en el Algoritmo 17 como venimos haciendo durante toda la memoria.

Algorithm 17 Prueba de Baillie-PSW

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

```
if Division_por_tentativa(n, 1000) = False then        ▷ Si nos devuelve False,
    return False                                       ▷ es que hay algún factor menor de 1000,
                                                       ▷ por tanto devolvemos compuesto
else
    if Miller_Rabin(n, 2) = False then                 ▷ Ejecutamos Miller-Rabin con base 2
        return False
    else
        return TestF_Lucas(n)      ▷ Devolvemos el resultado de la prueba fuerte de Lucas
    end if
end if
```

Algorithm 18 Test fuerte de Lucas

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

```
D, i ← 5, 1
while Jacobi(D, n) = 1 do                    ▷ Bucle para elegir el valor de D
    D ← −D + 2(−1)^i
    i ← i + 1
end while
P ← 1                                        ▷ Definimos P y Q
Q ← (1 − D)/4
j, d ← 0, n + 1
while d ≡ 0 (mod 2) do                       ▷ Descomponemos n + 1 de la manera deseada
    j ← j + 1
    d ← d/2
end while
U_0, U_1 ← 0, 1                              ▷ Valores iniciales de U y V
V_0, V_1 ← 2, P
for 2 ≤ k ≤ d do                             ▷ Cálculo de U_d y V_d
    U ← PU_1 − QU_0 (mod n)
    U_0 ← U_1
    U_1 ← U
    V ← PV_1 − QV_0 (mod n)
    V_0 ← V_1
    V_1 ← V
end for
if U = 0 o V = 0 then                        ▷ Si alguno de los dos es 0, n es posible primo
    return True
else                                         ▷ Si no seguimos probando con los V_{2^r d}
    r ← 1
    bool ← True
    while r < s y bool = True do
        V ← V^2 − 2Q^{2^{r−1} d} (mod n)     ▷ Utilizamos la fórmula V_{2k} = V_k^2 − 2Q^k
        if V = 0 then
            return True
        end if
    end while
    return False
end if
```

En la actualidad aún no se ha encontrado número compuesto alguno que pase con éxito la prueba de Baillie-PSW. Un dato curioso es que los autores de [PSW80] ofrecen al final de este mismo artículo un premio de 30$ a quien encuentre un contraejemplo a esto, o bien demuestre que tal número existe. Todo esto nos lleva a establecer la siguiente conjetura.

Conjetura 2.1. *Si n es un entero positivo mayor que 1 que pasa con éxito la prueba de primalidad de Baillie-PSW, entonces n es primo.*

Para terminar ilustramos la prueba con algunos ejemplos.

Ejemplo 2.7.

- Comenzamos con el caso de $n = 25326001$. Lo primero sería comprobar si existe algún factor menor que 1000. En este caso se puede comprobar que no es así.

  Realizamos entonces la prueba de Miller-Rabin para la base 2. Lo primero es factorizar $n - 1 = 2^4 \cdot 1582875$. Si calculamos $2^{1582875}$ observamos que es congruente con -1 módulo n. Por lo que Miller-Rabin nos dice que es un posible primo. Seguimos entonces con la prueba.

  Pasamos a realizar la prueba fuerte de Lucas. Vemos que el primer entero D de la sucesión $\{5, -7, 9, \dots\}$ que cumple que $\left(\frac{D}{n}\right) = -1$ es $D = -11$. Así $P = 1$ y $Q = 3$. Factorizamos $n + 1 = 2 \cdot 12663001$. Haciendo cálculos obtenemos lo siguiente: $U_{12663001} \equiv 5170797 \pmod{n}$ y $V_{12663001} \equiv 17549793 \pmod{n}$. Podemos concluir entonces que $n = 25326001$ no es primo. En este caso notamos que n era un pseudoprimo fuerte para la base 2 pero no es un pseudoprimo fuerte de Lucas, por lo que Baillie-PSW da la respuesta correcta.

- Estudiamos ahora el caso de $n = 6843227$. De nuevo no encontramos factores por debajo de 1000.

  Pasamos a Miller-Rabin. Factorizamos de manera conveniente $n - 1$ y nos queda $n - 1 = 2 \cdot 3421613$ y vemos que $2^{3421613} \equiv 2814770 \pmod{n}$ por lo que podemos concluir que n no es primo.

- Por último analizamos el caso de $n = 1664614493$. Otra vez este entero no tiene enteros menores que 1000 en su factorización.

  Pasamos a aplicar Miller-Rabin. Tenemos que $n - 1 = 2^2 \cdot 416153623$. Al calcular $2^{416153623}$ obtenemos 152892113 módulo n. Y si hacemos $2^{2 \cdot 416153623}$ vemos que es congruente con -1 módulo n. Por tanto es un probable primo.

  Ejecutamos ahora el test fuerte de Lucas. Vemos que el primer entero D de la sucesión $\{5, -7, 9, \dots\}$ que cumple que $\left(\frac{D}{n}\right) = -1$ es $D = 5$. Por lo que $P = 1$ y $Q = -1$. Factorizando obtenemos $n + 1 = 2 \cdot 832307247$. Si calculamos los términos correspondientes de las sucesiones de Lucas vemos que $U_{832307247} \equiv 0 \pmod{n}$ por lo que concluimos que n es primo. Además en este caso debe ser cierto ya que, hasta la fecha de elaboración de esta memoria, no se ha encontrado ningún pseudoprimo para el test de Baillie-PSW.

3. Aplicaciones en criptografía

En este capítulo final se estudiarán de manera muy breve algunas de las muchas aplicaciones de la teoría de números al mundo de la seguridad en las telecomunicaciones. Comenzaremos exponiendo algunos algoritmos de generación aleatoria de enteros y de números primos. Tras esto, introduciremos el concepto de criptosistema y terminaremos describiendo algunos de los más utilizados en la actualidad como son RSA, ElGamal y el protocolo Diffie-Hellman.

> To the layman, a lot of math (like primality testing and factoring large numbers) may seem a frivolous waste of time. However, this research often pays off unexpectedly years later. Factoring and primality testing have become important because of the need to make electronic communications secure...So, what used to be an esoteric playground for mathematicians has become applicable research.
>
> *(David Gries and Fred B. Schneider,*
> *A logical aproach to discrete math [GS13])*

3.1. Generación de números y primos aleatorios

La generación de números aleatorios o RNG (*random number generation* en inglés) es un recurso muy útil hoy en día. Tiene aplicaciones en campos como la criptografía, la estadística, análisis numérico... Una sucesión numérica se dice aleatoria cuando cada uno de los números de la misma no depende de los anteriores, es decir, no podemos predecir los números de la misma.

Conseguir números puramente aleatorios es algo muy complicado (casi imposible), aunque en la actualidad se está utilizando la mecánica cuántica para hacerlo posible. Generalmente cuando se habla de números aleatorios en realidad nos referimos a números pseudoaleatorios.

Guiándonos por [Yano2], en esta sección se describirán métodos basados en congruencias lineales, suelen conocerse como LCG (*linear congruence generator* en inglés), que generan números que pudieran confundirse con aleatorios. Estos métodos fueron introducidos por primera vez por D. H. Lehmer en 1949. A continuación explicaremos su funcionamiento.

Para iniciar el proceso de generación necesitamos los siguientes ingredientes:

- Un entero positivo, n (módulo).
- x_0 cumpliendo $0 \leq x_0 < n$ (semilla).
- a tal que $0 \leq a \leq n$ (multiplicador).
- b que satisfaga $0 < b < n$ (incremento).

A partir de estos elementos construimos la siguiente sucesión de manera recursiva:

$$x_j \equiv ax_{j-1} + b \pmod{n},\ j > 0,$$

para $1 \leq j \leq l$, donde l es el menor valor para el que ocurre que $x_{l+1} \equiv x_j \pmod{n}$ para algún $j \leq l$. A l lo llamamos el periodo del generador. El mejor generador se da cuando $l = n$, que obviamente es el mayor valor que este puede alcanzar. En [Knu14] D. E. Knuth da una condición necesaria y suficiente para alcanzar este objetivo.

Teorema 3.1. *Una LGC tiene periodo $l = n$ si y solo si $\gcd(b, n) = 1$, $a \equiv 1 \pmod{p}$ para todo primo p tal que $p|n$ y $a \equiv 1 \pmod{4}$ si $4|n$.*

Demostración. Es larga para poder exponerla aquí, pero se puede encontrar como bien se dijo antes en [Knu14]. □

Mostramos un ejemplo, también extraído de [Yan02] del funcionamiento del algoritmo.

Ejemplo 3.1. Sean $x_0 = 5$, $a = 11$, $b = 73$ y $n = 1399$. Los números generados son entonces:
$x_0 = 5$
$x_1 = ax_0 + b \pmod{n} = 128$
$x_2 = ax_1 + b \pmod{n} = 82$
$x_3 = ax_2 + b \pmod{n} = 975$
$x_4 = ax_3 + b \pmod{n} = 1005$
$x_5 = ax_4 + b \pmod{n} = 1335$
$x_6 = ax_5 + b \pmod{n} = 768$
$x_7 = ax_6 + b \pmod{n} = 127$
$x_8 = ax_7 + b \pmod{n} = 71$
$x_9 = ax_8 + b \pmod{n} = 854$
$x_{10} = ax_9 + b \pmod{n} = 1073$
$\vdots$
$x_{232} = ax_{231} + b \pmod{n} = 121$
$x_{233} = ax_{232} + b \pmod{n} = 5$
$x_{234} = ax_{233} + b \pmod{n} = 128.$
Observamos entonces que la longitud o periodo de esta sucesión es $l = 233$.

Podemos preguntarnos que hacer si lo que nos interesa es obtener un número aleatorio con un cierto número de cifras. En el caso de la criptografía el caso de interés es generar números con un cierto número de bits, digamos r. Para ello podemos ejecutar un algoritmo de generación de números aleatorios con el que podamos obtener r números aleatorios. Tras esto formamos el número deseado simplemente obteniendo el residuo módulo 2 de cada uno de los números generados.

Tras esto nos centramos en la generación aleatoria de números primos. Esto es de vital importancia en el criptosistema RSA, del que hablaremos más tarde.

Para generar primos simplemente podemos, tras generar un número aleatorio impar (bastaría con tener un 1 al final de su representación en binario), hacer que este pase un test de primalidad (generalmente probabilístico). Si lo pasa, tenemos nuestro primo, y sino generamos otro número aleatorio y repetimos el proceso.

En la actualidad, lo más común es aplicar el test de Miller-Rabin o Baillie-PSW. En el capítulo anterior vimos que tras pasar para un cierto número de bases el test de Miller-Rabin, la probabilidad de que nuestro entero sea primo es muy alta. Por otra parte, con Baillie-PSW tendríamos una fiabilidad del 100 % como ya se comentó.

También existe otro algoritmo muy extendido de generación de primos que se basa en un resultado debido a Pocklington, este se puede consultar en [Yan02].

3.2. Criptosistemas

En criptografía, un criptosistema es un conjunto de algoritmos criptográficos necesarios para implementar un servicio de seguridad particular, generalmente para conseguir confidencialidad. A continuación damos la definición de criptosistema de acuerdo con [Wik21].

Podemos definir un criptosistema como una tupla $(\mathcal{P}, \mathcal{C}, \mathcal{K}, \mathcal{E}, \mathcal{D})$ que cumple las siguientes propiedades:

- $\mathcal{P}$ es un conjunto denominado espacio de texto plano. Sus elementos se llaman textos planos.
- $\mathcal{C}$ es un conjunto denominado espacio de texto cifrado. Sus elementos se denominan textos cifrados.
- $\mathcal{K}$ es un conjunto denominado el espacio de clave. Sus elementos se denominan claves.
- $\mathcal{E} = \{E_k : k \in \mathcal{K}\}$ es un conjunto de funciones $E_k : \mathcal{P} \to \mathcal{C}$. Sus elementos se denominan funciones de encriptación (o de cifrado).
- $\mathcal{D} = \{D_k : k \in \mathcal{K}$ es un conjunto de funciones $D_k : \mathcal{C} \to \mathcal{P}\}$. Sus elementos se denominan funciones de desencriptación (o de descifrado).

Además se debe satisfacer que para cada $e \in \mathcal{K}$, debe existir $d \in \mathcal{K}$ tal que $D_d(E_e(p)) = p$ para todo $p \in \mathcal{P}$.

Esencialmente podemos distinguir dos tipos de criptosistemas.

1) **Criptosistemas simétricos** (o de clave privada): este tipo de criptosistemas se caracteriza por usar la misma clave para el cifrado que para el descifrado, es decir, se cumple que $e = d$. Esto implica que una persona que pueda cifrar el mensaje también puede descifrarlo. Como ejemplos de este tipo de criptosistemas podemos encontrar AES (*Advanced Encryption Standard*) y DES (*Data Encryption Standard*).

2) **Criptosistemas asimétricos** (o de clave pública): a diferencia de los anteriores utiliza claves distintas para el cifrado y para el descifrado. En este caso la clave de cifrado es pública pero no lo es la de descifrado. Ambas dos están relacionadas matemáticamente, pero es extremadamente complicado obtener la clave privada a partir de la pública (al menos para que sea un buen criptosistema). Como ejemplos tenemos el conocido criptosistema RSA (el más usado actualmente) y el criptosistema ElGamal.

3.2.1. Criptosistema RSA

Publicado en 1978 ([RSA78]) por tres investigadores del MIT (*Massachusetts Institute of Technology*), Ronald L. Rivest, Adi Shamir y Leonard Adleman, es en la actualidad el criptosistema de clave pública más utilizado a nivel mundial.

El éxito del criptosistema se basa en la siguiente premisa: es sencillo encontrar dos primos de grandes pero es muy complicado factorizar un entero de gran tamaño dando sus factores primos.

Pasamos a detallar ahora el proceso de generación de claves de RSA y las funciones de cifrado y descifrado siguiendo [Bla16].

1) **Generación de claves.** El primer paso es generar aleatoriamente dos números primos grandes y distintos, que llamaremos p y q. Para ello podemos usar distintas pruebas de primalidad, siendo las más usadas Miller-Rabin (2.2.3) y Baillie-PSW (2.2.4).

   Tras esto se calcula $n = pq$ y $\Phi = \varphi(n) = (p-1)(q-1)$ (por ser p y q primos) y se busca un entero $1 < c < \Phi$ tal que $\gcd(c, \Phi) = 1$.

   Seguidamente calculamos el inverso multiplicativo de c módulo Φ (mediante el algoritmo de Euclides extendido), es decir, un entero d tal que $cd \equiv 1 \pmod{\Phi}$.

   Tras esto ya hemos generado la clave pública (n, c), y la clave privada (n, d). Una vez disponemos de ellas procedemos a describir el proceso de cifrado y descifrado.

2) **Cifrado y descifrado.** Definimos el conjunto de claves como

$$K = \{(n, p, q, c, d) : n = pq,\ p,\ q \text{ primos},\ cd \equiv 1 \pmod{\varphi(n)}\}.$$

   Dada una clave concreta $K = (n, p, q, c, d)$ y $x \in \mathbb{Z}_n$ tenemos que las funciones de cifrado y descifrado son respectivamente

$$E_c(x) = x^c \pmod{n},$$

$$D_d(x) = x^d \pmod{n}.$$

   Además vemos que están bien definidas ya que

$$D_d(E_c(x)) = (x^c)^d = x^{cd} = x^{1+t\Phi} = x(x^\Phi)^t = x, \text{para cierto } t \in \mathbb{Z}.$$

   Toda la cadena de igualdades anteriores ha realizado en $\mathbb{Z}_n$ lógicamente. La última igualdad se debe al Teorema de Euler.

Vemos que las funciones de cifrado y descifrado son muy sencillas y elegantes. Observamos que si la factorización de n fuera conocida podríamos descifrar el mensaje cifrado, ya que si hallamos p y q sería fácil obtener $\varphi(n)$ y al tener este podríamos calcular el inverso multiplicativo de c obteniendo así d.

Es debido a esto que los primos generados deben ser de gran tamaño y suficientemente separados entre si, ya que de lo contrario n podría romperse usando el método de factorización de Fermat (1.2), y por tanto no exista una manera eficiente de descomponer n y así poder hallar la clave privada.

Seguidamente presentamos una variación de RSA, en la que en lugar de utilizarse la función totiente de Euler se utiliza la función lambda de Carmichael que definimos ahora.

Definición 3.1. La función de Carmichael de un entero positivo n, denotada como $\lambda(n)$, se define como el menor entero m tal que $a^m \equiv 1 \pmod{n}$, para cada entero a coprimo con n.

Sean ahora p, q primos, entonces sabemos por el PTF (Teorema 1.1) que para cualquier múltiplo k de $p-1$ y $q-1$ se cumple que $a^k \equiv 1 \pmod{p}$ y $a^k \equiv 1 \pmod{q}$. Además, por el Teorema Chino del Resto tendríamos que $a^k \equiv 1 \pmod{pq}$. Es fácil ver por este razonamiento que $\lambda(pq) = \operatorname{lcm}(p-1, q-1)$.

Sabiendo esto vamos a probar que podemos sustituir sin problema la función totiente de Euler, φ, por la función de Carmichael, λ. Veamos que $D_d(E_c(x)) = x$.

$$D_d(E_c(x)) = (x^c)^d = x^{1+t\lambda(n)} = x(x^\lambda)^t = x \pmod{n}.$$

Por tanto, vemos que es lícito el uso de la función de Carmichael en RSA. Es inmediato darse cuenta de que $\lambda(n) \leq \varphi(n)$, lo que nos puede ahorrar algunos cálculos, pero en la práctica la diferencia de cómputo entre una variante y otra es casi indistinguible.

Por otra parte, es importante destacar que el criptosistema RSA no solo nos permite garantizar la confidencialidad de una comunicación sino que también proporciona otros servicios como bien puede ser la firma digital, de la que hablamos a continuación. Describimos el proceso conforme a lo obtenido de [Upm].

Supongamos que el usuario Alicia quiere enviar un mensaje al usuario Bruno. Pero además Alicia quiere firmar el mensaje de forma que Bruno pueda estar seguro de que el mensaje que ha recibido lo ha enviado Alicia. Los pasos que debe seguir Alicia para conseguir su propósito son los siguientes:

1) Utilizar una función hash (o función resumen) para crear un resumen de su mensaje. En pocas palabras una función hash permite obtener un resumen de tamaño fijo de un mensaje que puede ser muy largo. Además de esto una función hash debe cumplir una serie de requisitos adicionales (ver [Wik22b]).

2) Representa este resumen como un entero M entre 0 y $n-1$.

3) Utiliza su clave privada (n, d) para hacer $S = M^d \pmod{n}$.

4) Envía S a Bruno junto con el mensaje original cifrado.

Cuando Bruno ha recibido esta información debe comprobar que efectivamente el mensaje proviene de Alicia, es decir, verificar la firma digital. Para hacer esto Bruno actúa de la siguiente forma:

1) Utiliza la clave pública de Alicia (n, c) para computar $V = S^c \pmod{n}$.

2) De V obtenemos el resumen que había realizado Alicia del mensaje.

3) De forma paralela, Bruno aplica la correspondiente función hash al mensaje que le ha llegado (ya descifrado), obteniendo otro resumen.

4) Si ambos resúmenes coinciden, queda verificada la firma. Entonces Bruno puede estar seguro de que el mensaje procede de Alicia y que además el mensaje ha llegado íntegro, es decir, sin ser alterado.

Con la firma digital podemos garantizar la autenticación del origen, la integridad del mensaje y también el no-repudio en origen, puesto que si Bruno ha podido recuperar el resumen del mensaje utilizando la clave pública de Alicia, la firma recibida junto con el mensaje sólo puede haber sido generada con la clave privada de Alicia.

En la actualidad no se conoce algoritmo alguno (al menos uno que se pueda implementar de forma eficiente) que permita factorizar enteros en tiempo polinomial, por lo que la seguridad está garantizada. Sin embargo, en estos últimos años se ha empezado a hablar de computación cuántica y de que su llegada puede ser devastadora para RSA. Esto se debe a que en 1994 el matemático Peter Shor publicó (véase [Sho94]) un algoritmo cuántico que factoriza enteros en un tiempo polinomial. Si este llega a implementarse en un ordenador cuántico podría suponer el fin de RSA.

3.2.2. Protocolo Diffie-Hellman

Propuesto en 1976 por Whitfield Diffie y Martin Hellman en [DH76]. Este artículo fue un gran avance en criptografía ya que en él se introducía la criptografía asimétrica o de clave pública.

El protocolo Diffie-Hellman fue uno de los primeros algoritmos de criptografía de clave pública. No se trata de un criptosistema de clave pública al uso sino más bien un protocolo de establecimiento de claves.

Este basa su seguridad en el problema del logaritmo discreto, que hace referencia al cálculo del logaritmo en grupos cíclicos finitos, una tarea ciertamente compleja (incluso más que la factorización de enteros) para algunos de estos grupos (para más información véase [Kob94]). En el Apéndice A describimos uno de los algoritmos más conocidos para el cálculo del logaritmo discreto, que además es muy simple, se trata del algoritmo paso de bebé-paso de gigante.

Siguiendo con el protocolo, lo primero que deben hacer ambos usuarios es fijar un primo p tal que el problema del logaritmo discreto sea complejo en $\mathbb{Z}_p^*$. Tras esto debemos elegir un elemento primitivo de dicho grupo, que llamaremos g.

Una vez disponemos de estos elementos, el protocolo esta basado en la siguiente conjetura por primera vez propuesta por Diffie y Hellman. Esta se conoce también como el problema de Diffie-Hellman.

Conjetura 3.1. *Dados* $g^a \pmod p$ *y* $g^b \pmod p$, *calcular* $g^{ab} \pmod p$ *es computacionalmente equivalente a calcular uno de los logaritmos* $a = \log_g g^a \pmod p$ *o* $b = \log_g g^b \pmod p$.

Describimos ahora el proceso mediante el que se crea la clave compartida.

1) Un usuario llamado Alicia elige un número aleatorio a tal que $1 \leq a \leq p-1$. Tras esto envía $g^a \pmod p$ al usuario Bruno.

2) Bruno hace lo mismo y toma un número aleatorio b tal que $1 \leq b \leq p-1$. Tras esto envía $g^b \pmod p$ a Alicia.

3) Finalmente Bruno calcula $(g^a)^b \pmod p$ y Alicia $(g^b)^a \pmod p$. Por tanto $g^{ab} \equiv (g^a)^b \equiv (g^b)^a \pmod p$ será la clave compartida que hemos generado con el protocolo.

Tras esto Alicia y Bruno comparten una clave que podrá ser utilizada en comunicaciones futuras.

3.2.3. Criptosistema ElGamal

El criptosistema ElGamal surge en el año 1985, cuando es publicado en [ElG85] por Tahel ElGamal.

El problema del logaritmo discreto también juega un papel decisivo aquí, ya que es en lo que se basa la seguridad.

Como hicimos con RSA primero veremos cómo se realiza la generación de claves y tras ello las funciones de cifrado y descifrado.

1) **Generación de claves.** Lo primero es generar un número primo p tal que el problema del logaritmo discreto sea complejo en $\mathbb{Z}_p$. Tras esto se elige un elemento primitivo a del grupo $\mathbb{Z}_p$. Por último tomamos un entero t cumpliendo $1 < t < p-1$.

Con esto obtendríamos la clave pública, que sería (p, a, b) donde $b = a^t \pmod{p}$ y la clave privada es t.

2) **Cifrado y descifrado.** Dados p, a, t, b tales como los que acabamos de describir definimos el conjunto de claves como sigue:

$$K = \{(p, a, t, b) : b \equiv a^t \pmod{p}\}.$$

Para una clave dada $K = (p, a, t, b)$ y un elemento $m \in \mathbb{Z}_p - \{1\}$ generado aleatoriamente las funciones de cifrado y descifrado son respectivamente:

$$E_t(m) = (a^t, mb^t),$$

$$D_t(x, y) = yx^{-t}.$$

Probamos que efectivamente están bien definidas:

$$D_t(E_t(m)) = D_t(a^t, mb^t) = mb^t(a^t)^{-t} = m(a^t)^t(a^t)^{-t} = m.$$

Como se dijo en un inicio la fortaleza de ElGamal se base en la complejidad para calcular $\log_a(b)$ con lo que podríamos calcular la clave privada t y descifrar por tanto el mensaje. Hoy en día este es uno de los criptosistemas más utilizados por su efectividad ya que no existe algoritmo conocido que sea capaz de calcular el logaritmo discreto de forma eficiente.

Analizando, tanto el protocolo de Diffie-Hellman como el criptosistema ElGamal, vemos que necesitamos encontrar un elemento primitivo del grupo $\mathbb{Z}_p^*$, lo cual puede convertirse en una tarea complicada. Describimos a continuación un procedimiento para poder hallarlo sin mucho trabajo.

Sabemos que un elemento $a \in \mathbb{Z}_p^*$ es primitivo si cumple que $a^k \not\equiv 1 \pmod{p}$ para $k \in \{1, \ldots, p-1\}$. Esta condición equivale a que $a^d \not\equiv 1 \pmod{p}$ siendo d un divisor maximal de $p-1$. Es decir, basta probar que $a^{(p-1)/p_i} \not\equiv 1 \pmod{p}$ con p_i divisor de p ya que si no es congruente con 1 para ninguno, entonces tampoco lo será para ningún divisor de los mismos, ya que si lo fuera eso implicaría que $a^{(p-1)/p_i} \equiv 1 \pmod{p}$.

Por tanto, para encontrar un elemento primitivo debemos conocer la factorización de $p-1$, cosa que no suele pasar. Como alternativa, lo que se suele hacer es buscar p primo de tal forma que $(p-1)/2$ también sea primo. Si conseguimos esto, encontrar a elemento primitivo se reduce a comprobar que $a^2 \not\equiv 1 \pmod{p}$ y que $a^{(p-1)/2} \not\equiv 1 \pmod{p}$. Esto último, como consecuencia del Criterio de Euler (Teorema 1.2), es lo mismo que decir que el símbolo de Legendre $\left(\frac{a}{p}\right) = -1$.

Una vez conseguimos un primo p a medida procedemos a elegir aleatoriamente $a \in \{2, \ldots, p-2\}$ y verificar si se cumple lo deseado. Tras unos pocos intentos conseguiremos generar un elemento primitivo de $\mathbb{Z}_p^*$ (pues la probabilidad de encontrar a en cada intento es de $1/2$).

Ahora solo queda obtener un primo p que cumpla que $(p-1)/2$ también es primo. Esto podemos hacerlo de forma aleatoria (pero lógicamente siempre eligiendo un candidato impar) comprobando con el test de Miller-Rabin (2.2.3) que p y $(p-1)/2$ son primos. Si el entero p no satisface las condiciones impuestas probamos con $p+2$ y continuamos así el proceso hasta encontrar lo buscado.

Como conclusión cabe decir que al igual que ocurría con el problema de factorización también existen algoritmos cuánticos para resolver de manera eficiente el problema del

logaritmo discreto.

Es debido a esta nueva *etapa cuántica* de la computación por lo que se ha empezado a trabajar en la criptografía postcuántica. Esta consiste en generar algoritmos criptográficos que sean resistentes a ataques efectuados mediante computación cuántica, para más información se puede consultar [Wik22c].

A. Algoritmo de paso de bebé-paso de gigante (Shanks)

También conocido como algoritmo de Shanks, el primero que lo publicó, pero hay autores que afirman que era conocido anteriormente. Es uno de los algoritmos de cálculo del logaritmo discreto más conocidos. Introducimos el algoritmo utilizando los apuntes de la asignatura Teoría de números y criptografía.

Antes que nada mostraremos su pseudocódigo y luego probaremos que efectivamente calcula el logaritmo discreto (si es que existe). Supongamos que queremos calcular $\log_b(h)$ en un grupo G. El algoritmo toma como entradas b y h y devuelve $\log_b(h)$ o *Fail*. Si devuelve *Fail*, es que no existe el logaritmo buscado.

Algorithm 19 Algoritmo paso de bebé-paso de gigante

Input: $b \in G$ de orden n y $h \in \langle b \rangle$
  $f \leftarrow \lceil \sqrt{n} \rceil$
  $table \leftarrow \{\}$
  for $0 \leq i \leq f-1$ **do**
    $table \leftarrow table \cup (i, b^i)$
  end for
  $b^{-f} \leftarrow b^{n-f}$
  $h_0 \leftarrow h$
  for $0 \leq i \leq f-1$ **do**
    if $(j, h_i) \in table$ para algún j **then**
      return $j + if$
    else
      $h_{i+1} \leftarrow h_i b^{-f}$
    end if
  end for
  return Fail

Teorema A.1. *El Algoritmo 19 calcula correctamente el logaritmo discreto, si existe.*

Demostración. Observamos que $h_i = hb^{-if}$. Por tanto, si $h_i = b^j$ tenemos que

$$h = hb^{-if}b^{if} = b^j b^{if} = b^{j+if},$$

de donde $j + if = \log_b(h)$. Por tanto, si el logaritmo existe, el Algoritmo 19 da la respuesta correcta.

Por otra parte, si $h = b^m$, dividimos m entre f, obteniendo $m = if + j$ donde $0 \leq j < f$. Dado que $m < n$, necesariamente $i < f$, por lo que

$$h_i = hb^{-if} = b^m b^{-if} = b^{if+j} b^{-ij} = b^j$$

es decir, si existe el logaritmo discreto el Algoritmo 19 debe encontrarlo. □

Ejemplo A.1. Mostramos ahora un ejemplo de aplicación del algoritmo de Shanks, vamos

a calcular $\log_3(109) \pmod{211}$. Por tanto, los datos para el Algoritmo 19 son $G = \mathbb{F}_{211}^*, b = 3, h = 109, f = \lceil \sqrt{211-1} \rceil = 15$. Construimos ahora la tabla del algoritmo:

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
b^i	1	3	9	27	81	32	96	77	20	60	180	118	143	7	21

Calculamos $b^{-f} = b^{211-1-f} = b^{195} = 67 \pmod{211}$. A continuación calculamos los h_i tal como se describe en el algoritmo.

- Vemos que $h_0 = h = 109$ y no está en la tabla.
- Calculamos $h_1 = h_0 b^{-f} = 129$ pero tampoco está en la tabla.
- Lo mismo para $h_2 = 203$, pero no está en la tabla.
- Ahora $h_3 = 97$ sigue sin estar en la tabla.
- Tenemos entonces $h_4 = 169$ que no está en la tabla.
- Calculando $h_5 = 140$ vemos que tampoco lo encontramos en la tabla.
- Finalmente $h_6 = 96$ está en la tabla. Entonces, siguiendo el Algoritmo 19, el resultado del logaritmo es $6 + 6 \cdot 15 = 96$.

Se puede comprobar facilmente que el resultado es correcto ya que $3^{96} \equiv 103 \pmod{211}$.

Bibliografía

[AGP94] William R Alford, Andrew Granville, and Carl Pomerance. There are infinitely many carmichael numbers. *Annals of Mathematics*, 139(3):703–722, 1994. [Citado en pág. 32]

[AKS04] Manindra Agrawal, Neeraj Kayal, and Nitin Saxena. Primes is in p. *Annals of mathematics*, pages 781–793, 2004. [Citado en pág. 28]

[Arn97] François Arnault. The rabin-monier theorem for lucas pseudoprimes. *Mathematics of computation*, 66(218):869–881, 1997. [Citado en pág. 43]

[Bla16] Blas Torrecillas. El criptosistema RSA, 2016. Granada. [Citado en pág. 49]

[Bre12] David M Bressoud. *Factorization and primality testing*. Springer Science & Business Media, 2012. [Citado en pág. 20]

[BW80] Robert Baillie and Samuel S Wagstaff. Lucas pseudoprimes. *Mathematics of Computation*, 35(152):1391–1417, 1980. [Citado en pág. 41]

[Con18] Keith Conrad. Cyclicity of $(z/(p))\times$, 2018. [Citado en pág. 16]

[CP05] Richard E Crandall and Carl Pomerance. *Prime numbers: a computational perspective*, volume 20052. Springer, 2005. [Citado en págs. xi, xiii, 9, 15, 20, 27, 28, 31, 39, and 42]

[DH76] Whitfield Diffie and Martin Hellman. New directions in cryptography. *IEEE transactions on Information Theory*, 22(6):644–654, 1976. [Citado en pág. 52]

[Dix81] John D Dixon. Asymptotically fast factorization of integers. *Mathematics of computation*, 36(153):255–260, 1981. [Citado en págs. 10 and 12]

[ElG85] Taher ElGamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE transactions on information theory*, 31(4):469–472, 1985. [Citado en pág. 52]

[FR07] Benjamin Fine and Gerhard Rosenberger. *Number theory*. Springer, 2007. [Citado en pág. 24]

[Gar] Enrique R. Aznar García. Cálculo de las sucesiones de lucas. `http://wpd.ugr.es/~eaznar/presentaciones/calLucas.htm#/`. Recurso online. Accedido por última vez el 5 de mayo de 2022. [Citado en pág. 43]

[GS13] David Gries and Fred B Schneider. *A logical approach to discrete math*. Springer Science & Business Media, 2013. [Citado en pág. 47]

[Knu14] Donald E Knuth. *Art of computer programming, volume 2: Seminumerical algorithms*. Addison-Wesley Professional, 2014. [Citado en pág. 48]

[Kob94] Neal Koblitz. *A course in number theory and cryptography*, volume 114. Springer Science & Business Media, 1994. [Citado en págs. xi, xiii, 5, 16, 32, 39, and 52]

[Kon07] Alan G Konheim. *Computer security and cryptography*. John Wiley & Sons, 2007. [Citado en págs. 4 and 11]

[Nes12] A Yu Nesterenko. Cycle detection algorithms and their applications. *Journal of Mathematical Sciences*, 182(4):518–526, 2012. [Citado en págs. 5 and 6]

[Pla] Solovay-strassen test. `http://planetmath.org/solovaystrassentest`. Recurso online. Accedido por última vez el 3 de mayo de 2022. [Citado en pág. 35]

[Pom96] Carl Pomerance. A tale of two sieves. In *Notices Amer. Math. Soc*. Citeseer, 1996. [Citado en pág. 15]

[PSW80] Carl Pomerance, John L Selfridge, and Samuel S Wagstaff. The pseudoprimes to 25×10^9. *Mathematics of Computation*, 35(151):1003–1026, 1980. [Citado en págs. 41, 43, and 46]

[Rie12] Hans Riesel. *Prime numbers and computer methods for factorization*, volume 126. Springer Science & Business Media, 2012. [Citado en pág. 8]

[RSA78] Ronald L Rivest, Adi Shamir, and Leonard Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978. [Citado en pág. 49]

[Sho94] Peter W Shor. Algorithms for quantum computation: discrete logarithms and factoring. In *Proceedings 35th annual symposium on foundations of computer science*, pages 124–134. Ieee, 1994. [Citado en pág. 51]

[Sho09] Victor Shoup. *A computational introduction to number theory and algebra*. Cambridge university press, 2009. [Citado en pág. 17]

[Upm] El uso de rsa en la seguridad de la información. http://www.dma.fi.upm.es/recursos/aplicaciones/matematica_discreta/web/aritmetica_modular/rsa2.html. Recurso online. Accedido por última vez el 11 de junio de 2022. [Citado en pág. 51]

[Wik21] Wikipedia contributors. Cryptosystem — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Cryptosystem&oldid=1042466837, 2021. [Internet; descargado 26-mayo-2022]. [Citado en pág. 49]

[Wik22a] Wikipedia contributors. Carmichael number — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Carmichael_number&oldid=1063216884, 2022. [Online; accessed 31-May-2022]. [Citado en pág. 32]

[Wik22b] Wikipedia contributors. Cryptographic hash function — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Cryptographic_hash_function&oldid=1086890848, 2022. [Online; accessed 11-June-2022]. [Citado en pág. 51]

[Wik22c] Wikipedia contributors. Post-quantum cryptography — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Post-quantum_cryptography&oldid=1088363432, 2022. [Online; accessed 31-May-2022]. [Citado en pág. 54]

[Yan02] Song Y Yan. *Number theory for computing*. Springer Science & Business Media, 2002. [Citado en págs. xi, xiii, 37, 39, 41, 47, and 48]