# RSA

José Galaviz Casas

*Facultad de Ciencias, UNAM*

# Índice general

# 1. Introducción

## 1.1. Motivación

En un sistema criptográfico de clave o llave pública, cada usuario posee la capacidad de enviar mensajes cifrados a cualquier otro usuario y sin embargo sólo el destinatario del mensaje tiene la capacidad de recuperar su contenido. Por supuesto el remitente conoce el mensaje, conoce el texto cifrado que le corresponde y la clave con la que se cifró, pero a pesar de ello no posee lo necesario para leer otros mensajes cifrados para el mismo destinatario. Un requisito indispensable para la existencia de un sistema así es la existencia de dos claves diferentes por cada usuario: una *pública* a la que cualquier otro usuario tiene acceso, utilizada para cifrar mensajes destinados a él y la otra, *privada*, que sólo él mismo conoce, para poder descifrarlos. En este peculiar sistema el conocer la clave para cifrar mensajes no proporciona información relevante para obtener la otra, la usada para descifrar; a pesar de estar íntimamente ligadas.

Para construir un sistema así, se requiere de la existencia de una función de un sólo sentido (*one-way function*): una función invertible para la que el cálculo de la función misma es simple, mientras que el cálculo de la inversa es difícil, computacionalmente hablando. Ya en 1913 William Jevons prefigura la función de un sentido que será usada en los 70 para construir el más célebre de los sistemas criptográficos de clave pública. En [1] (Cap. VII, Pág. 122-123), Jevons escribe:

> "Dados cualesquiera dos números nosotros podemos, mediante un procedimiento simple e infalible, obtener su producto; pero cuando nos es dado un número grande es algo completamente distinto determinar sus factores. ¿Puede el lector decir cuál es el valor de los dos números [primos] que multiplicados producen el número 8,616,4600,799? Yo creo que probablemente nadie, salvo yo mismo, los conocerá alguna vez"

Ciertamente, dados dos números primos cualesquiera $p$ y $q$, es simple obtener $n = p \times q$ y sin embargo dado el valor de $n$, es en general computacionalmente costoso encontrar $p$ y $q$. Este costo computacional excesivo es lo que provee de seguridad al sistema conocido como RSA y que abordamos en este documento. Antes de proceder a describir y analizar RSA es necesario asegurarnos de recordar algunos conceptos y procedimientos preliminares.

## 1.2. Algoritmo de Euclides

Es el algoritmo más antiguo del que tenemos noticia. Sirve para calcular el máximo común divisor de dos números enteros positivos. Se denotará como $\text{MCD}\,(r_0, r_1)$. En la primaria lo calculamos como se muestra en el ejemplo 1.2.1.

**Ejemplo 1.2.1.** Para calcular el máximo común divisor de un par de enteros, solíamos dividir simultáneamente ambos números por los números primos en orden creciente, hasta que ya no fuera posible encontrar divisor primo común.

$$\begin{array}{rr|l} 84 & 30 & 2 \\ 42 & 15 & 3 \\ 14 & 5 & \end{array}$$

De donde $\text{MCD}\,(84, 30) = 2 \cdot 3 = 6$

$\triangleleft$

Como vemos esto significa, esencialmente, factorizar, lo que es impensable para números grandes[1].

Pero, suponiendo que $r_0 > r_1$, es posible construir un método para calcular el MCD basándose en que: $\text{MCD}\,(r_0, r_1) = \text{MCD}\,(r_0 - r_1, r_1)$, como podemos ver en ejemplo 1.2.2.

**Ejemplo 1.2.2.** Recalculando nuestro caso previo:

[1] Se considera que encontrar los factores primos de un número arbitrario es una operación computacionalmente difícil.

| | |
|---|---|
| $\text{MCD}\,(84, 30)$ | 84 - 30 = 54 |
| $\text{MCD}\,(54, 30)$ | 54 - 30 = 24 |
| $\text{MCD}\,(30, 24)$ | 30 - 24 = 6 |
| $\text{MCD}\,(24, 6)$ | 24 - 6 = 18 |
| $\text{MCD}\,(18, 6)$ | 18 - 6 = 12 |
| $\text{MCD}\,(12, 6)$ | 12 - 6 = 6 |
| $\text{MCD}\,(6, 6)$ | |

De donde $\text{MCD}\,(84, 30) = \text{MCD}\,(6, 6) = 6$

Es posible formular recursivamente el proceso ejemplificado, como se ve en el algoritmo 1.

**Algoritmo 1** Máximo común divisor: versión con diferencias.

```
1: function MCD-R(r_0, r_1)
2:     if r_0 > r_1 then
3:         return MCD-R(r_0 − r_1, r_1)
4:     else if r_0 < r_1 then
5:         return MCD-R(r_1 − r_0, r_0)
6:     else
7:         return r_0
8:     end if
9: end function
```

Se puede, claro, abreviar el proceso usando la división y considerando el residuo de ella, como puede verse en el algoritmo 2, probablemente la versión más popular del *algoritmo de Euclides*.

Lo que también puede expresarse de manera iterativa, como en la tercera versión mostrada en el algoritmo 3.

**Ejemplo 1.2.3.** De acuerdo con el algoritmo de Euclides, el proceso para calcular el máximo común divisor de 1106 y 497 es el siguiente:

$$\begin{array}{rcllcl} 1106 & = & 2 \cdot 497 + 112 & \Longrightarrow \text{MCD}\,(1106, 497) & = & \text{MCD}\,(497, 112) \\ 497 & = & 4 \cdot 112 + 49 & \Longrightarrow \text{MCD}\,(497, 112) & = & \text{MCD}\,(112, 49) \\ 112 & = & 2 \cdot 49 + 14 & \Longrightarrow \text{MCD}\,(112, 49) & = & \text{MCD}\,(49, 14) \\ 49 & = & 3 \cdot 14 + 7 & \Longrightarrow \text{MCD}\,(49, 14) & = & \text{MCD}\,(14, 7) \\ 14 & = & 2 \cdot 7 + 0 & \Longrightarrow \text{MCD}\,(14, 7) & = & \text{MCD}\,(7, 0) \end{array}$$

◁

**Algoritmo 2** Máximo común divisor: versión con divisiones.

**Require:** $r_0 > r_1$

```
1: function MCD-D(r_0, r_1)
2:     if r_1 = 0 then
3:         return r_0
4:     else
5:         return MCD-D(r_1, r_0 (mód r_1))
6:     end if
7: end function
```

**Algoritmo 3** Máximo común divisor: versión iterativa.

**Require:** $r_0 > r_1$

```
1: function MCD(r_0, r_1)
2:     while r_1 > 0 do
3:         aux ← r_0 (mód r_1)
4:         r_0 ← r_1
5:         r_1 ← aux
6:     end while
7:     return r_0
8: end function
```

## 1.3. Algoritmo de Euclides Extendido

El algoritmo de Euclides presentado arriba es ya útil dado que nos permite determinar el máximo común divisor y por tanto, si dos números dados son primos relativos[2]. Pero puede ser aún más útil si lo modificamos un poco.

**Ejemplo 1.3.1.** Si despejamos los residuos de las expresiones del ejemplo previo (1.2.3) obtenemos:

$$\begin{aligned} 112 &= 1106 - 2 \cdot 497 \quad (1) \\ 49 &= 497 - 4 \cdot 112 \quad (2) \\ 14 &= 112 - 2 \cdot 49 \quad (3) \\ 7 &= 49 - 3 \cdot 14 \quad (4) \end{aligned}$$

De la línea 4:

[2]Recuérdese que dos números $a, b \in \mathbb{Z}$ son primos relativos si $\text{MCD}\,(a, b) = 1$.

$$
\begin{aligned}
7 &= 49 - 3 \cdot 14 \\
&= 49 - 3 \cdot [112 - 2 \cdot 49] && \text{usando (3)} \\
&= 7 \cdot 49 - 3 \cdot 112 \\
&= 7 \cdot [497 - 4 \cdot 112] - 3 \cdot 112 && \text{usando (2)} \\
&= 7 \cdot 497 - 31 \cdot 112 \\
&= 7 \cdot 497 - 31 \cdot [1106 - 2 \cdot 497] && \text{usando (1)} \\
&= 69 \cdot 497 - 31 \cdot 1106
\end{aligned}
$$

La última línea expresa el máximo común divisor de 1106 y 497 como combinación lineal de ellos mismos. Podemos modificar el algoritmo de Euclides para que, además de entregar el MCD, también nos proporcione los coeficientes de esta combinación.

Conviene etiquetar los números importantes que aparecen en el proceso mostrado en el ejemplo 1.2.3.

| $i$ | $r_i$ | $q_i$ |
|---|---|---|
| 0 | 1106 | |
| 1 | 497 | 2 |
| 2 | 112 | 4 |
| 3 | 49 | 2 |
| 4 | 14 | 3 |
| 5 | 7 | 2 |
| 6 | 0 | |

Observaciones:

- Residuos.
  - $r_2 = 112 = 1106 \pmod{497}$
  - $r_3 = 49 = 497 \pmod{112}$
  - $r_4 = 14 = 112 \pmod{49}$
  - $r_5 = 7 = 49 \pmod{14}$
  - $r_6 = 0 = 14 \pmod{7}$
- Cocientes.

- $q_1 = 2 = (1106 - 112)/497$
- $q_2 = 4 = (497 - 49)/112$
- $q_3 = 2 = (112 - 14)/49$
- $q_4 = 3 = (49 - 7)/14$
- $q_5 = 2 = (14 - 0)/7$

$\triangleleft$

Podemos generalizar las observaciones con objeto de obtener una variante del algoritmo de Euclides que nos entregue como salida los coeficientes de la combinación lineal que expresa el MCD en términos de sus argumentos además del MCD mismo.

De la observación hecha sobre los residuos:

$$r_i = r_{i-2} \pmod{r_{i-1}} \tag{1.1}$$

De la hecha sobre los cocientes:

$$q_{i-1} = \frac{r_{i-2} - r_i}{r_{i-1}} \tag{1.2}$$

El proceso que llevamos a cabo para obtener el MCD, suponiendo que este resulta ser el $k$-ésimo residuo es el siguiente:

$$\begin{aligned}
r_0 &= q_1 r_1 + r_2 \\
r_1 &= q_2 r_2 + r_3 \\
&\vdots \quad\quad \vdots \\
r_{i-2} &= q_{i-1} r_{i-1} + r_i \\
&\vdots \quad\quad \vdots \\
r_{k-2} &= q_{k-1} r_{k-1} + r_k \\
r_{k-1} &= q_k r_k + 0
\end{aligned} \tag{1.3}$$

De la expresión 1.3 se obtiene:

$$r_i = r_{i-2} - q_{i-1}\ r_{i-1} \tag{1.4}$$

Y queremos usar estas expresiones para obtener las que nos interesan, de la forma:

$$r_i = s_i\ r_0 + t_i\ r_1 \tag{1.5}$$

Ya que al expresar así a $r_k$ tendremos al MCD como combinación lineal de $r_0$ y $r_1$, los argumentos de entrada al proceso.

Si usamos expresiones de la forma de 1.5 para $r_{i-2}$ y $r_{i-1}$ en 1.4, tenemos:

$$\begin{aligned} r_i &= r_{i-2} - q_{i-1}\ r_{i-1} \\ &= [s_{i-2}\ r_0 + t_{i-2}\ r_1] - q_{i-1}\ [s_{i-1}\ r_0 + t_{i-1}\ r_1] \\ &= r_0\ [s_{i-2} - q_{i-1}\ s_{i-1}] - r_1\ [t_{i-2} - q_{i-1}\ t_{i-1}] \end{aligned}$$

De esta última expresión, de la forma 1.5, podemos extraer los valores para $s_i$ y $t_i$:

$$s_i = s_{i-2} - q_{i-1}\ s_{i-1} \tag{1.6}$$

$$t_i = t_{i-2} - q_{i-1}\ t_{i-1} \tag{1.7}$$

Conjuntando los resultados de las expresiones 1.1, 1.2, 1.6 y 1.7, podemos finalmente escribir la versión extendida del algoritmo de Euclides (algoritmo 4).

**Algoritmo 4** Máximo común divisor: versión extendida del algoritmo de Euclides.

**Require:** $r_0 > r_1$

```
1:  procedure MCD-EXT(r_0, r_1)
2:      s_0 ← 1
3:      t_0 ← 0
4:      s_1 ← 0
5:      t_1 ← 1
6:      i ← 1
7:      repeat
8:          i ← i + 1
9:          r_i ← r_{i-2} (mód r_{i-1})
10:         q_{i-1} ← (r_{i-2} − r_i) / r_{i-1}
11:         s_i ← s_{i-2} − q_{i-1} s_{i-1}
12:         t_i ← t_{i-2} − q_{i-1} t_{i-1}
13:     until r_i = 0
14:     return {r_{i-1} = MCD(r_0, r_1), s = s_{i-1}, t = t_{i-1}}
15: end procedure
```

La utilidad del algoritmo extendido de Euclides es la de calcular el inverso multiplicativo de un elemento en los enteros módulo $m$, suponiendo que dicho inverso exista. En $\mathbb{Z}_m$ sabemos que tienen inverso multiplicativo sólo

aquellos elementos que sean primos relativos al tamaño del módulo ($m$). Es decir, para un elemento $k \in \mathbb{Z}_m$ existe $k^{-1} \in \mathbb{Z}_m$ si y sólo si $\text{MCD}\,(k, m) = 1$.

**Ejemplo 1.3.2.** Tomemos como ejemplo $\mathbb{Z}_{21}$, hay que recordar que en los enteros módulo $m$ se enrollan todos los enteros, incluso los negativos, así que en nuestro conjunto, módulo 21, el -1 es equivalente al 20, -2 al 19, -3 al 18 y así sucesivamente.

El número 10 está en $\mathbb{Z}_{21}$, y además $\text{MCD}\,(21, 10) = 1$, así que debe tener inverso. ¿Quién es?

Con el algoritmo extendido de Euclides podemos obtener la combinación lineal:

$$\text{MCD}\,(21, 10) = 1 = 1 \cdot 21 - 2 \cdot 10$$

Si tomamos esta expresión módulo 21:

$$1 \equiv 1 \cdot 21 - 2 \cdot 10 \pmod{21}$$

pero $1 \cdot 21$ es congruente con cero módulo 21. Así que:

$$1 \equiv -2 \cdot 10 \pmod{21}$$

Así que el inverso multiplicativo del 10 en $\mathbb{Z}_{21}$, es el -2 o mejor dicho el 19:

$$10^{-1} = 19 \in \mathbb{Z}_{21}$$

$\lhd$

## 1.4. Exponenciación eficiente

El algoritmo más ingenuo para elevar un número $b$, a una potencia $n$, no negativa, es el 5. Pero no es el más eficiente, en número de operaciones es $O(n)$. Se puede hacer mejor.

**Ejemplo 1.4.1.** Digamos que queremos obtener $3^{11}$, el 11 en binario se escribe

$$1011_2 = 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$$

Así que podríamos escribir:

$$\begin{aligned} 3^{1011_2} &= 3^{1\times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0} \\ &= 3^{8+2+1} \\ &= 3^8 \times 3^2 \times 3^1 \\ &= ((3^2)^2)^2 \times 3^2 \times 3^1 \end{aligned}$$

**Algoritmo 5** Algoritmo simple para obtener $b^n$.

**Require:** $n \geq 0$

```
1: function POTENCIA(b, n)
2:     p ← 1
3:     if n > 0 then
4:         for i ← 1 to n do
5:             p ← p · b
6:         end for
7:     end if
8:     return p
9: end function
```

Supongamos que queremos obtener ahora $3^{14}$, el 14 en binario es

$$1110_2 = 1 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0$$

Así que haciendo lo mismo que antes:

$$\begin{aligned} 3^{1110_2} &= 3^{1\times 2^3 + 1\times 2^2 + 1\times 2^1 + 0\times 2^0} \\ &= 3^{8+4+2} \\ &= 3^8 \times 3^4 \times 3^2 \\ &= ((3^2)^2)^2 \times (3^2)^2 \times 3^2 \end{aligned}$$

Los dos casos mostrados ilustran las dos posibilidades para el bit menos significativo, si este es 1 el primer factor a la derecha es justamente la base que hay que elevar, si es cero podemos decir que el primer factor de la derecha no aparece porque es uno.

Luego aparece un patrón: para obtener el factor que corresponde al $i$-ésimo bit (de derecha a izquierda) sólo tenemos que elevar el previo al cuadrado. Estos factores se ven acumulando, multiplicándose, conforme se recorre la expresión hacia la izquierda.

$\triangleleft$

El algoritmo esbozado en el ejemplo previo recibe el nombre en inglés de *Square and Multiply* y puede escribirse formalmente como se muestra en el algoritmo 6. Siguiendo este procedimiento hacemos a lo más tantas multiplicaciones como dos veces el número de bits del exponente, es decir: $O(\log_2 n)$.

**Algoritmo 6** Algoritmo eficiente para obtener $b^n$.

**Require:** $n \geq 0$

```
1: function POTENCIA-SQM(b, n)
2:     binexp ← aBinario(n)
3:     s ← b
4:     p ← 1
5:     if n > 0 then
6:         if binexp[0] = 1 then
7:             p ← b
8:         end if
9:         for j ← 1 to binexp.length − 1 do
10:            s ← s^2
11:            if binexp[j] = 1 then
12:                p ← p · s
13:            end if
14:        end for
15:    end if
16:    return p
17: end function
```

## 1.5. El teorema pequeño de Fermat

**Teorema 1** (Pequeño de Fermat)**.** *Si $p$ es un número primo entonces, para cualquier número natural $a$:*

$$a^p \equiv a \pmod{p} \tag{1.8}$$

*Demostración.* Por inducción sobre el número natural $a$.

**Caso base.** Para $a = 1$ se tiene que: $1^p = 1 \equiv 1 \pmod{p}$ para cualquier primo $p$, lo que satisface el teorema.

**Hipótesis.** Suponemos que $a^p \equiv a \pmod{p}$ (i.e. $p \mid (a^p - a)$).

**Paso inductivo.** Debemos probar que $(a+1)^p \equiv a+1 \pmod{p}$, es decir: $p \mid (a+1)^p - (a+1)$. Por el teorema del binomio:

$$(a+1)^p = a^p + \binom{p}{1}a^{p-1} + \binom{p}{2}a^{p-2} + \cdots + \binom{p}{p-1}a + 1$$

De donde:

$$(a+1)^p - a^p - 1 = \binom{p}{1}a^{p-1} + \binom{p}{2}a^{p-2} + \cdots + \binom{p}{p-1}a$$

$p$ es un factor de cada sumando del lado derecho de esta expresión, así que $p$ divide al lado derecho y por tanto al izquierdo:

$$p \mid (a+1)^p - a^p - 1$$

Si sumamos a esta expresión algo divisible por $p$ el resultado seguirá siendo divisible por $p$, así que, usando la hipótesis de inducción:

$$p \mid (a+1)^p - a^p - 1 + a^p - a = (a+1)^p - (a+1)$$

□

Se puede formular una versión alterna del teorema anterior, de hecho, la versión original escrita por Fermat (sin demostración).

Si fuera posible garantizar que $p \nmid a$ entonces $a \pmod p \neq 0$ y podríamos dividir ambos lados de la expresión 1.8:

$$\frac{a^p}{a} \equiv \frac{a}{a} \pmod p$$

Esto ocurre, ciertamente, cuando $a < p$, así que en $\mathbb{Z}_p$ es válida la siguiente versión.

**Teorema 2** (Pequeño de Fermat, versión original, octubre de 1640)**.** *Si $p$ es un número primo entonces para toda $a$ natural, que no es múltiplo de $p$ ocurre:*

$$a^{p-1} \equiv 1 \pmod p \tag{1.9}$$

El teorema inverso, que haría de este un "si y sólo si" y por tanto una caracterización de los números primos, no es cierto. Hay números compuestos que satisfacen la congruencia previa.

**Definición 1.** Un número de Carmichael, $n$, es un número impar compuesto (i.e. no primo) tal que para cualquier $a \in \mathbb{N}$ que satisface $\text{MCD}\,(a, n) = 1$, ocurre:

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

El $561 = 3 \times 11 \times 17$ es el primer número de Carmichael.

## 1.6. El teorema de Euler

Fermat no demostró su teorema, quien lo hizo fue Leonhard Euler en 1736 y lo generalizó mediante una función que después usaremos.

**Definición 2.** La función $\varphi$ de Euler, también llamada *función indicatriz* de Euler (también conocida como *función totient*[3]), evaluada en un número natural $m$, es el número total de enteros en $\mathbb{Z}_m$ que resultan ser primos relativos con $m$.

**Ejemplo 1.6.1.** Sea $m = 6$; $\mathbb{Z}_6 = \{0, 1, 2, 3, 4, 5\}$, tenemos que:

$$\text{MCD}\,(0, 6) = 6$$
$$\text{MCD}\,(1, 6) = 1$$
$$\text{MCD}\,(2, 6) = 2$$
$$\text{MCD}\,(3, 6) = 3$$
$$\text{MCD}\,(4, 6) = 2$$
$$\text{MCD}\,(5, 6) = 1$$

Como se puede ver, los únicos primos relativos con 6 menores que él son el 1 y el 5, así que $\varphi(6) = 2$

En cambio si consideramos $m = 5$; $\mathbb{Z}_5 = \{0, 1, 2, 3, 4\}$ y tenemos que:

$$\text{MCD}\,(0, 5) = 5$$
$$\text{MCD}\,(1, 5) = 1$$
$$\text{MCD}\,(2, 5) = 1$$
$$\text{MCD}\,(3, 5) = 1$$
$$\text{MCD}\,(4, 5) = 1$$

De donde: $\varphi(5) = 4$

$\triangleleft$

Euler proporcionó un mecanismo para calcular $\varphi(m)$ sin tener que verificar cada número menor que $m$. De todos modos no es un método eficiente porque involucra factorizar $m$, lo que no es trivial si $m$ es grande. El siguiente teorema establece este método para calcular $\varphi$, hace uso de lo que se denomina *factorización canónica* de $m$ lo que significa que cada factor primo aparece una sola vez en la expresión, si es múltiple, esta multiplicidad se expresa mediante un exponente, y los primos aparecen en orden creciente.

[3]El término *totient* proviene del latín y significa *contabilizar*. El primero en usarla fue Gauss, Euler la estudió exhaustivamente y el término totient se lo debemos a Sylvester.

**Teorema 3.** *Si $m$ es un entero cualquiera cuya factorización canónica es:*

$$m = p_1^{e_1} \cdot p_2^{e_2} \cdot \ldots p_n^{e_n}$$

*entonces:*

$$\varphi(m) = \prod_{i=1}^{n} (p_i^{e_i} - p_i^{e_i - 1}) \tag{1.10}$$

**Ejemplo 1.6.2.** Sea $m = 240$, entonces, como $m = 2^4 \cdot 3 \cdot 5$, sabemos que:

$$\begin{aligned} \varphi(240) &= (2^4 - 2^3) \cdot (3^1 - 3^0) \cdot (5^1 - 5^0) \\ &= (16 - 8) \cdot (3 - 1) \cdot (5 - 1) \\ &= 8 \cdot 2 \cdot 4 \\ &= 64 \end{aligned}$$

◁

Como caso particular del teorema previo tenemos que si $p$ es un número primo entonces $\varphi(p) = p - 1$

Otra propiedad muy útil de la función de Euler es que es multiplicativa: si $m$ y $n$ son primos relativos, entonces $\varphi(m\,n) = \varphi(m)\,\varphi(n)$.

**Teorema 4** (de la indicatriz de Euler)**.** *Si $a$ y $m$ son enteros positivos tales que* $\text{MCD}\,(a, m) = 1$ *entonces:*

$$a^{\varphi(m)} \equiv 1 \pmod{m} \tag{1.11}$$

*Demostración.* Recordemos que $\varphi(m)$ es la cardinalidad del subconjunto de $\mathbb{Z}_m$ de elementos primos relativos con $m$.

Sea $A = \{n_1, n_2, \ldots, n_{\varphi(m)}\} \pmod{m}$ (tomar el módulo de cada elemento), el conjunto de elementos en $\mathbb{Z}_m$ que son primos relativos con $m$ (Nota: como no necesariamente $a < m$, no necesariamente $a \in A$).

Para todo $n_i \in A$, como $\text{MCD}\,(n_i, m) = 1$ y además $\text{MCD}\,(a, m) = 1$, entonces $\text{MCD}\,(a\,n_i, m) = 1$. Así que para los elementos de

$$B = \{a\,n_1, a\,n_2, \ldots, a\,n_{\varphi(m)}\} \pmod{m}$$

podemos afirmar que:

- Son primos relativos con $m$.
- Son estrictamente menores que $m$ (por el módulo) por lo que están en $\mathbb{Z}_m$.

Por lo que la cardinalidad de $A$ y $B$ es la misma. Además en ambos cada elemento aparece sólo una vez y sólo hay primos relativos con $m$. Sabemos que en $A$ están *todos* los primos relativos menores que $m$, así que en $B$ también: por cada elemento de $A$ hay uno de $B$ congruente con él y viceversa. Entonces:

$$n_1\, n_2\, \ldots n_{\varphi(m)} \equiv a\, n_1\ a\, n_2\ \ldots a\, n_{\varphi(m)} \pmod{m}$$

es decir:

$$a^{\varphi(m)}\ (n_1\, n_2\, \ldots n_{\varphi(m)}) \equiv n_1\, n_2\, \ldots n_{\varphi(m)} \pmod{m}$$

Dividiendo[4] por el producto de las $n_i$:

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

□

**Ejemplo 1.6.3.** Sea $m = 12$

$$\varphi(12) = (2^2 - 2^1)(3^1 - 3^0) = (4-2)(3-1) = 4$$

Por tanto:

$$5^{\varphi(12)} = 5^4 = 625 \equiv 1 \pmod{12}$$

◁

[4]Esta división es válida dado que, cada $n_i$ y por tanto el producto de ellas, es primo relativo con $m$, no congruente con cero módulo $m$, lo que invalidaría la división.

# 2. RSA

RSA es el sistema criptográfico de llave pública más usado en el mundo. Fue creado por Ron Rivest, Adi Shamir y Leonard Adleman, en el MIT en 1977. Ahora sabemos que fue independientemente descubierto unos cuatro años antes por Clifford Cocks, un criptógrafo de los *Government Communications Head Quarters* (GCHQ) de la inteligencia británica, pero su trabajo fue desclasificado hasta 1997.

RSA usa la factorización de un número como la función de un sólo sentido que le proporciona la seguridad al sistema.

## 2.1. Generación de llave

Para poner en marcha RSA es necesario primero llevar a cabo algunas tareas.

1. Elegir dos números primos grandes $p$ y $q$ (de al menos 512 bits de tamaño).
2. Calcular el producto $n = p\,q$.
3. Calcular $\varphi(n) = (p-1)(q-1)$ la función indicatriz de Euler.
4. Elegir $e$, $1 < e < \varphi(n)$, tal que $\text{MCD}\,(e, \varphi(n)) = 1$.
5. Calcular $d$, el inverso multiplicativo de $e$ módulo $\varphi(n)$. Es decir, encontrar $d$ tal que:

$$d \cdot e \equiv 1 \pmod{\varphi(n)} \tag{2.1}$$

el número $e$ será usado como llave para cifrar y $d$ para descifrar. La llave pública es la pareja $(n, e)$, la llave privada es $(n, d)$ aunque realmente $n$ no es privada, se necesita para descifrar. Se deben mantener en secreto $p$, $q$, y $\varphi(n)$.

## 2.2. Operación

Un mensaje $m$ tal que $0 \leq m < n$ se cifra como:

$$c = m^e \pmod{n} \tag{2.2}$$

Para descifrar se hace:

$$c^d = (m^e)^d = m^{e \cdot d} \equiv m \pmod{n} \tag{2.3}$$

Para probar que RSA funciona, teniendo en cuenta la ecuación 2.1, debemos probar que es cierta la expresión 2.3. Recordemos que:

$$\varphi(n) = (p-1)(q-1)$$

así que 2.1 es equivalente a:

$$e \cdot d - 1 = k\,(p-1)(q-1) \tag{2.4}$$

para alguna $k$ entera no negativa.

Hay que recordar también que, de acuerdo con el teorema chino del residuo, si $a \equiv b \pmod{s}$ y $a \equiv b \pmod{t}$ entonces $a \equiv b \pmod{s \cdot t}$, así que para probar la congruencia módulo $n$ lo haremos primero con $p$ y, por simetría, la tendremos también para $q$ y por tanto para el producto de ellos.

Ahora bien, para $p$ hay dos casos a considerar:

- $m \equiv 0 \pmod{p}$. En este caso $m^{ed}$ es múltiplo de $p$, así que $m^{ed} \equiv 0 \equiv m \pmod{n}$
- $m \not\equiv 0 \pmod{p}$. En este caso:

  $$m^{ed} = m \cdot m^{ed-1}$$

  por 2.4:

  $$m^{ed} = m \cdot m^{k(p-1)(q-1)} = m \cdot (m^{p-1})^{k(q-1)}$$

  por el pequeño teorema de Fermat:

  $$m^{p-1} \equiv 1 \pmod{p}$$

  así que:

  $$m \cdot (m^{p-1})^{k(q-1)} \equiv m \cdot 1^{k(q-1)} \equiv m \pmod{p}$$

Alternativamente se puede probar lo mismo usando el teorema de Euler, dado que $e \cdot d = 1 + k\varphi(n)$ entonces:

$$\begin{aligned} m^{ed} &= m^{1+k\varphi(n)} \\ &= m \cdot m^{k\varphi(n)} \\ &\equiv m \cdot \left(m^{\varphi(n)}\right)^k \\ &\equiv m \cdot 1^k \\ &\equiv m \pmod{n} \end{aligned}$$

**Ejemplo 2.2.1.** Hagamos un ejemplo de juguete con números pequeños.

Sean $p = 7$, $q = 13$, por lo que $n = 7 \times 13 = 91$ y $\varphi(n) = 72$

Supongamos que $e = 5$, lo que satisface $\text{MCD}\,(5, 72) = 1$ por lo que se encuentra:

$$d = 29 = 5^{-1} \pmod{72}$$

- Sea $m = 10$ el mensaje a enviar. Al cifrarlo obtenemos:

  $$c = m^e = 10^5 \equiv 82 \pmod{91}$$

  Y al descifrar:

  $$c^{29} \equiv 82^{29} \pmod{91} \equiv 10 \pmod{91}$$

- Si hacemos $m_u = 21$ obtendremos:

  $$c_u = m_u^e = 21^5 \equiv 21 \pmod{91}$$

  ¡El mensaje cifrado $c_u$ es igual al mensaje claro!

$\lhd$

Como se puede ver en el ejemplo 2.2.1 existen mensajes que no pueden ocultarse. A estos se les denomina *inocultables* en RSA (*unconcealed* en inglés). De hecho, dado que dependen de una relación entre $e - 1$, $p - 1$ y $q - 1$ (todos ellos pares) el número de mensajes inocultables es, al menos, 9 y su número en general es:

$$(1 + \text{MCD}\,(e - 1, p - 1)) \cdot (1 + \text{MCD}\,(e - 1, q - 1))$$

Cuanto mayores sean $p$ y $q$ será más improbable que un mensaje claro válido casualmente sea un inocultable y en todo caso, las implementaciones serias de RSA realizan lo que se denomina relleno o *padding* para garantizar que no ocurran fenómenos extraños. Otra de las garantías que ofrece el relleno es que un mensaje claro no se cifra dos veces igual, sin importar que los valores de $p$, $q$ y $e$ sean los mismos.

## 2.3. Generación de $p$ y $q$

Hay que aclarar cómo se encuentran $p$ y $q$ en RSA. Para cada uno de ellos el procedimiento es:

1. Se genera un número aleatorio grande, de al menos 512 bits.
2. Se procede a probar si es primo o no.

Si fracasa la prueba del segundo paso se repite el proceso hasta encontrar uno que la pase. Cada uno de los pasos tiene características importantes.

### 2.3.1. CSPRNG

Aquello para lo que existe un algoritmo no es, por definición, aleatorio. De allí que se les llame tradicionalmente *pseudoaleatorios* a los números producidos con una llamada a un método o subrutina en una computadora digital. Los generadores de números pseudoaleatorios (*Pseudo Random Number Generator* o PRNG) que solemos utilizar para programas de aplicación no son muy complejos. Se utiliza comúnmente un generador de congruencias lineal (*Linear Congruential Generator* o LCG). Al generador se le alimenta con un número llamado *semilla*, al que identificaremos con $X_0$. Los siguientes números pseudoaleatorios son los términos de una secuencia determinada por la expresión:

$$X_{n+1} = (a\,X_n + c) \pmod{m}$$

El objetivo es que la secuencia resultante, a pesar de ser completamente determinista, estadísticamente se comporte como una secuencia aleatoria. Sin embargo la simplicidad de estos métodos los haría presa fácil del criptoanálisis: la secuencia generada es finita, claro, y comienza a repetirse luego de un tiempo. No podríamos pensar en usarlos en aplicaciones criptográficas.

Se han establecido requisitos más fuertes para un *Generador de Números Pseudoaleatorios Criptográficamente Seguro* o CSPRNG por sus sigla en inglés (BSI, Alemania):

- Con muy baja probabilidad genera secuencias idénticas de números.
- La secuencia debe ser estadísticamente indistinguible de una verdaderamente aleatoria.
- Dada una secuencia cualquiera, generada por el método, debe ser imposible determinar el término siguiente.

- Dado un estado del generador, debe ser imposible calcular los estados previos del generador y los términos previos de la secuencia.

Para juzgar el segundo punto se han establecido diversas pruebas estadísticas:

- Monobit test. En la secuencia considerada en binario, hay igual número de ceros que de unos.
- Poker test. Dados los términos de la secuencia, no debe haber repeticiones frecuentes del mismo dígito: 959, 883, 575, 344, no es una buena secuencia, en cada término hay un dígito repetido.
- Kolmogorov-Smirnov. La distribución acumulativa de los términos de la secuencia observada debe coincidir con los de la secuencia teórica (normalmente uniforme).
- Run test. La secuencia no debe tener patrones de "corridas": dos subsecuencias crecientes y tres decrecientes, por ejemplo, o tres términos arriba de la media seguidos de tres por debajo de la media o tres corridas crecientes de longitud 8 y luego dos decrecientes de longitud 6, etcétera.
- Autocorrelación. Los términos de la secuencia deben ser independientes.

En 1982 Andrew Yao probó que es fundamental que no exista un algoritmo polinomial para determinar, en la secuencia, el bit siguiente con probabilidad mayor a 0.5.

Existen varios diseños diferentes para este tipo de generadores, los más eficientes están basados en funciones hash criptográficas o en cifrados de bloque como DES o AES. A la función de bloque (hash o cifrado) se le suministra una semilla y se procede luego a iterar la función en la modalidad llamada "de contador": recibiendo como entrada 1, 2, 3, etc. cada nuevo resultado se suma al previo.

Hay diseños más complejos basados en problemas matemáticos o en operaciones sobre curvas elípticas. Uno de estos diseños fue publicado por el NIST como estándar en 2007 (NIST-800-90A). Algunos criptógrafos sospecharon que poseía una "puerta trasera" que podía hacer predecible la secuencia. En 2013, entre la información que Edward Snowden hizo pública, quedó claro que la NSA había puesto una puerta trasera en el estándar.

### 2.3.2. Pruebas de primalidad

Una vez que se genera un número grande con un procedimiento pseudo-aleatorio criptográficamente seguro, se debe probar si es primo o compuesto. En principio deseamos que esta prueba sea:

- General. Que sirva para cualquier número.
- Incondicional. Que su corrección no dependa de alguna conjetura no probada.
- Determinista. Que proporcione una total certidumbre acerca de su respuesta.
- Polinomial. Que sea un algoritmo de complejidad tratable.

En 2002, Agrawal, Kayal y Saxena probaron que existe un procedimiento polinomial para determinar si un número dado es primo o no. La prueba de Agrawal-Kayal-Saxena o AKS posee todas las cualidades listadas arriba. Sin embargo su costo, aunque polinomial, es muy alto: $O(\log^{12} n)$, donde $n$ es el número que se desea probar si es primo o no. Se han hecho mejoras al algoritmo AKS, actualmente su complejidad es $O(\log^6 n)$, pero sigue siendo costoso en comparación con otras pruebas de primalidad, que no garantizan la lista de requisitos, pero son eficientes y suficientemente confiables, por lo que, a pesar de poseer un algoritmo con 100 % de certidumbre, se prefieren usar estas pruebas probabilísticas.

Para empezar ¿Qué tan fácil es atinar a un número primo? La *función de conteo de primos*, denotada tradicionalmente como $\pi(x)$ (valuada en $\mathbb{R}^+$) nos dice cuantos números primos menores o iguales a $x$ existen. El teorema de los números primos establece la distribución asintótica de los primos como:

$$\pi(x) \approx \frac{x}{\ln x}$$

Entonces la probabilidad de atinarle a un primo es:

$$p(x \text{ es primo}) \approx \frac{1}{\ln x}$$

si nos restringimos a buscarlo entre los impares:

$$p(x \text{ es primo}) \approx \frac{2}{\ln x} \tag{2.5}$$

Para encontrar uno en el rango que nos interesa, de unos 512 bits:

$$p(x \text{ es primo}) \approx \frac{2}{\ln 2^{512}} = \frac{2}{512 \ln 2} \approx \frac{1}{177}$$

Existen varias pruebas históricamente importantes para probar, con cierto grado de confiabilidad, que un número es primo. La de Fermat, basada en el teorema pequeño que probamos antes, es la más antigua y como sabemos los números de Carmichael mienten en ella: la prueba diría que son primos cuando en realidad no lo son.

La siguiente prueba relevante fue la de Solovay-Strassen, como todas estas heurísticas, es un proceso iterativo en el que, cuantas más repeticiones se hagan, mayor certidumbre de adquiere de una respuesta correcta. Por supuesto también hay números que mienten en la prueba de Solovay-Strassen, todos ellos mienten también en la de Fermat, pero no viceversa. La probabilidad de que la prueba de Solovay-Strassen diga que un número es primo y que en realidad no lo sea es: $\frac{1}{2^t}$ donde $t$ es el número de iteraciones del proceso.

La prueba de primalidad más popular hoy en día es la de Miller-Rabin. Nuevamente los mentirosos de Miller-Rabin están contenidos propiamente en los de Solovay-Strassen. El algoritmo 7 muestra el procedimiento de la prueba. Dado el número $n$ que se desea probar si es primo o no, la prueba se inicia buscando un par de números $s$ y $r$, con $r$ impar, tal que $n - 1 = 2^s\, r$.

La probabilidad de error de la prueba de Miller-Rabin es la mitad de la de Solovay-Strassen, es decir, la prueba dice que $n$ es primo cuando en realidad no lo es, con probabilidad $\frac{1}{4^t}$.

**Ejemplo 2.3.1.** Sea $n = 91 = 7 \cdot 13$.

$$91 - 1 = 90 = 2 \cdot 45$$

así que $s = 1$ y $r = 45$.

Supongamos que escogemos $a = 9$, como $9^r = 9^{45} \equiv 1 \pmod{91}$ entonces la prueba diría que 91 es primo y estaría en un error.

$\lhd$

En una computadora digital es particularmente fácil encontrar la $r$ y la $s$ de la línea 2. Por supuesto, dado que se desea encontrar un valor de $n$ primo, lo menos que podemos pedir es que sea impar ya que, salvo el 2, el resto de los primos lo son. Dado entonces el valor de $n$ impar, su expresión binaria tendrá su bit menos significativo en 1 y por tanto el valor de $n - 1$ tendrá la misma expresión binaria, salvo justamente el bit menos significativo, que ahora será 0 necesariamente, dado que $n - 1$ es par. El procedimiento pide encontrar el valor de una $r$ impar que habrá que multiplicar por 2 $s$ veces para obtener el $n - 1$ cuya expresión binaria estamos considerando. Como multiplicar por 2 un número binario consiste en desplazar sus bits un lugar

**Algoritmo 7** Prueba de primalidad de Miller-Rabin.

**Require:** $n \geq 3$, $t \geq 1$
**Ensure:** false si $n$ no es primo, true si parece serlo

```
1: function MILLER-RABIN(n, t)
2:     Encontrar s y r, con r impar, tal que n − 1 = 2^s r
3:     for i ← 1 to t do
4:         Elegir a ∈ {2, ..., n − 2}, aleatoriamente
5:         y ← a^r (mód n)
6:         if y ≠ 1 ∧ y ≠ n − 1 then
7:             j ← 1
8:             while j ≤ s − 1 ∧ y ≠ n − 1 do
9:                 y ← y^2 (mód n)
10:                if y = 1 then
11:                    return false
12:                end if
13:                j ← j + 1
14:            end while
15:            if y ≠ n − 1 then
16:                return false
17:            end if
18:        end if
19:    end for
20:    return true
21: end function
```

a la izquierda, rellenando con cero los huecos menos significativos que se generan, encontrar el valor de $s$ consiste en contar cuantos ceros a la derecha consecutivos tiene la expresión binaria de $n-1$, sabemos que $s > 1$, porque el número es par. Y entonces, el número que nos quedaría quitando todos estos ceros menos significativos es justamente el valor de $r$.

**Ejemplo 2.3.2.** Sea $n = 89$ que realmente es un número primo.

La expresión binaria de $n$ es $n_2 = 1011001$ que, por supuesto, por ser impar tiene el bit menos significativo en 1. Así que $(n-1)_2 = 1011000 = 88_{10}$. Este número tiene tres bits en cero a la derecha, así que es como sí hubiéramos multiplicado al $1011 = 11$ por $2^3 = 8$. Por lo que $s = 3$ y $r = 11$. ◁

La complejidad de la prueba, si se requiere una respuesta con la precisión de $t$ iteraciones, es decir con una probabilidad de $\frac{1}{4^t}$ de error, es de $O(t \log^3 n)$

## 2.4. Ataques a RSA

Los ataques a RSA se pueden clasificar en tres grandes grupos:

- Ataques al protocolo. Esto es, ataques a la manera en que se usa o implementa RSA. Por ejemplo, no usar el relleno adecuado que garantice que un mensaje no se cifre como él mismo, o usar primos débiles: $p$ es un primo débil si $p-1$ tiene un factor primo $f$ relativamente pequeño. De ser así no es descabellado pensar en encontrar $f$ y por tanto $k$ tal que $p-1 = kf$, de donde $p = kf+1$ y si tenemos $p$ entonces tenemos $q$ y por tanto $\varphi(n)$, con lo que se puede calcular $d$ a partir de $e$.

- Se ha probado que romper la seguridad de RSA es equivalente a factorizar un número compuesto producto de dos primos[1] así que se ha invertido mucho trabajo en lograr mejores algoritmos o heurísticos para factorizar. En 1977 Martin Gardner publicó en su columna de *Scientific American* un reto RSA: un número de 129 dígitos cuyos factores primos había que encontrar. En aquel entonces se etimaba que el trabajo requeriria $4 \times 10^{13}$. El número fue factorizado, sin embargo, en 1994 usando 1600 computadoras de voluntarios a través de Internet. Se han hecho muchos avances en el estudio del problema de factorización, de hecho, motivados por RSA.

[1] No se ha probado si es equivalente al problema de factorizar en general.

- Ataques de canal lateral (side-channel). Consisten en que, una vez que se posee una implementación de RSA en una tarjeta inteligente o en un chip, es posible obtener algunos o todos los bits de la clave privada que se esté usando si se pueden medir las diferencias de voltaje entre diferentes terminales o *pines* del chip.

# Bibliografía


[1] Jevons, W.S. *The principles of science: A Treatise on Logic and Scientific Method*, 2a. Ed., Londres, Macmillan and Co., 1913.

[2] Katz, J. y Lindell Y., *Introduction to Modern Cryptography*, 2a Ed., Chapman & Hall / CRC, 2014.

[3] Menezes, A.J., van Oorschot P.C. y Vanstone S.A., *Handbook of Applied Cryptography*, CRC Press, 1996.

[4] Paar, C. y Pelzl J., *Understanding Cryptography*, Springer, 2010.