Revista Digital: Matemática, Educación e Internet
ISSN: 1659-0643
revistadigitalmatematica@itcr.ac.cr
Instituto Tecnológico de Costa Rica
Costa Rica

Miramontes de León, Gerardo
Análisis computacional a “Una fórmula que genera números primos”
Revista Digital: Matemática, Educación e Internet,
vol. 23, núm. 1, 2022, Agosto-Febrero, pp. 1-13
Instituto Tecnológico de Costa Rica
Cartago, Costa Rica

Revista digital
Matemática, Educación e Internet
https://revistas.tec.ac.cr/index.php/matematica
Vol 23, No 1. Agosto, 2022 – Febrero, 2023 ISSN 1659 -0643

Análisis computacional a "Una fórmula que genera números primos"

Computational analysis to "A formula that generates prime numbers"

Gerardo Miramontes de León
gmiram@ieee.org
Universidad Autónoma de Zacatecas
México

Recibido: 27 setiembre 2021 Aceptado: 1 abril 2022

Resumen: Se analiza el código computacional de "Una fórmula que genera números primos", la cual fue publicada en el Vol. 22, No. 1 de la Revista digital Matemática, Educación e Internet y que fue presentada como una función $a(n)$ dada por:

$$a(n) = n \left\lfloor \frac{2}{n - \sum_{i=1}^{n} \left\lceil \{\frac{n}{i}\} \right\rceil} \right\rfloor$$

Aquí se muestra que, para cada valor de n, esa fórmula se reduce a un bucle de la prueba de primalidad más simple, es decir, a la prueba de primalidad por división. Paso a paso se muestra que $a(n)$ incluye operaciones que se pueden evitar, como la extracción de la parte fraccionaria, y dos operaciones de redondeo. Se concluye que esa "fórmula que genera números primos" es en realidad una prueba de primalidad por división no optimizada, pues, por ejemplo, no evita probar valores pares de n.

Palabras Clave: función generadora de primos, prueba de primalidad, secuencia de primos.

Abstract: The computational code of "A formula that generates prime numbers" is analyzed, which was published in Vol. 22, No. 1 of Revista digital Matemática, Educación e Internet and which was presented as a function $a(n)$ given by:

$$a(n) = n \left\lfloor \frac{2}{n - \sum_{i=1}^{n} \left\lceil \{\frac{n}{i}\} \right\rceil} \right\rfloor$$

Here it is shown that, for each value of n, that formula reduces to a loop of the simplest primality test, that is, by trial division. It is shown, step by step, that $a(n)$ includes operations that can be avoided, such as extracting the fractional part, and two rounding operations. It is concluded that this "formula that generates prime numbers" is actually a non-optimized proof of primality by trial division, since, for example, it does not avoid testing even values of n.

Keywords: prime generating function, primality test, prime secuence.

1. Introducción

La búsqueda de una fórmula que genere números primos ha sido el anhelo de los matemáticos desde los tiempos antiguos. De acuerdo con Ribenboim [1]

> El problema es encontrar una "buena función" $f : \mathbb{N} \to \{\text{números primos}\}$. Esta función debe ser lo más fácil posible de calcular y, sobre todo, debe ser representable por funciones previamente conocidas.

Hay dos preguntas básicas que, de una manera u otra, siempre deberán ser respondidas. Ya sea que se desee una fórmula generadora de números primos, o si nos interesa saber si un número es primo, estas dos preguntas son:

1. ¿Hay una función que indique cómo se pueden generar números primos?
2. ¿Cómo se puede saber si un número dado es primo?

Lo deseable sería que una fórmula $f(n)$ entregue el n-ésimo número primo, es decir, $f(1) = p_1 = 2$, $f(2) = p_2 = 3$, $f(3) = p_3 = 5, \ldots, f(n) = p_n = n$-ésimo primo, donde p_n es el número primo en un orden natural. Hasta ahora la primera pregunta es la más difícil de responder.

Una de las fórmulas sencillas más conocidas es la fórmula de Euler,

$$p = n^2 + n + 41 \tag{1}$$

la cual genera, para cada número natural $n < 40$, un cierto número de primos. Para $n = 40$, resulta 1681, el cual no es primo. Aún cuando se tenga una fórmula que genere números primos, siempre habrá la necesidad de aplicar una prueba de primalidad sobre todo si n es muy grande.

Para la segunda pregunta, hay fórmulas que se pueden considerar como generadoras de números primos o como pruebas de primalidad. Por ejemplo, se tiene el siguiente teorema

Teorema 1 (Wilson)

Si p es un número primo, entonces $(p-1)! \equiv -1(mod\, p)$

El Teorema de Wilson se emplea en algunas funciones generadoras de números primos, como en la fórmula de Willans donde emplea una expresión equivalente a $(p-1)! \equiv -1(modp)$, es decir $((p-1)!+1)/p$, la cual es más una prueba de primalidad que una función generadora de números primos. Esta prueba no es práctica si p es grande, debido a la operación factorial, además de que se conocen pruebas más eficientes, como la prueba de primalidad por división, la cual se emplea en la fórmula que se analiza en este trabajo.

En otro ejemplo, de acuerdo con Mills [2], la función

$$f(n) = \lfloor A^{3^n} \rfloor \tag{2}$$

entrega valores primos para un número $A > 1$ y $n \in \mathbb{N}$. En este caso, el primer problema es encontrar el valor de A, y para ello se requiere de contar con una buena cantidad de números primos. Además, la mayoría de estas fórmulas no entregan una secuencia ordenada de números primos, ya que entre un número primo y el siguiente puede darse una gran falta de números primos no encontrados por dicha fórmula. La constante se calcula a partir de los números primos, en lugar de que los primos se generen a partir de la constante. Se dice que tales fórmulas codifican el conocimiento existente de números primos, en lugar de generar nuevos números primos. Al mismo tiempo, hay fórmulas que más que generar un número primo, en realidad son pruebas de primalidad y otras son tan poco prácticas que se aceptan como curiosidades matemáticas.

2. Objetivo

El objetivo de este artículo es mostrar que el código presentado en "Una fórmula que genera números primos" (ver [3] y [4]), incluye y se puede reducir a la prueba de primalidad más simple, la prueba de primalidad por división. Para tal efecto, se analiza el código en Mathematica©, propuesto por [3] pero en su versión con código en GNU Octave [5]. En adelante nos referimos a esa fórmula como "fórmula Camacho". También se comparan los tiempos de ejecución en cada caso.

3. Análisis de la fórmula Camacho

Como las pruebas de primalidad son necesarias o incluso están incluidas en algunas fórmulas que generan números primos, en esta sección se incluye la prueba de primalidad más simple, es decir, la prueba de primalidad por división. Después se analiza la (3) desde el punto de vista de cómputo.

3.1. Encontrar si n es primo

Es sabido que si n es un número muy grande, probar que n no es divisible por ningún número excepto 1 y n, nos llevaría a hacer un número nada razonable de cálculos [6]. Sin embargo, parece ser una de las mejores opciones, hasta ahora.

La prueba de primalidad por división se basa en que un numero es primo, si ningún número (diferente de 1 y de él mismo) menor o igual que su raíz cuadrada lo divide. La razón es que si n tiene un factor mayor a su raíz cuadrada, también tendrá un factor menor a ésta. En el Algoritmo 1 se muestra, en pseudocódigo, una función la cual entrega un resultado igual 1 si n es primo, y 0 en caso contrario.

Algoritmo 1: función esprimo

Ingresa: n
Regresa: p
Datos: Prueba $\frac{n}{i}$ con $i = 2$ hasta $i = \sqrt{n}$
Funcion `esprimo(`n`)`**:**

```
si n es igual a 2 o 3 entonces
    devolver p=1                                          // termina aquí
si no, si n/2 es entero entonces
    devolver p=0                                  // termina; es divisible entre 2
en otro caso
    For i_impar=3 hasta √n                              // prueba de 3 en adelante
    si n/i es entero entonces
        devolver p=0
    en otro caso
        devolver p=1
```

En el Apéndice A se incluye este y otros códigos en Octave desarrollados para este trabajo. Además de probar, por división, desde 2 hasta $\sqrt{n}$, se eliminan en la prueba los números pares mayores a 2. Eso permite reducir el número de operaciones no necesarias.

Puede notar que si multiplicamos el valor entregado por la función `esprimo` por el valor de n, se puede decir que tal operación "genera" un número primo y un cero si n no es primo. La operación sería la siguiente:

$$n \times \text{esprimo(n)} = \begin{cases} n \times 1 = n, & \text{si es primo} \\ n \times 0 = 0, & \text{si no es primo} \end{cases}$$

Esta pequeña nota sirve de base para el análisis siguiente.

3.2. Análisis de la fórmula

De acuerdo a [3, 4] la ecuación (3) es una nueva fórmula , "que permite comprobar si un número es primo":

$$a(n) = n \left\lfloor \frac{2}{n - \sum_{i=1}^{n} \left\lceil \{\frac{n}{i}\} \right\rceil} \right\rfloor \tag{3}$$

donde $\lfloor \cdot \rfloor$ indica redondeo descendente, $\lceil \cdot \rceil$ redondeo ascendente y $\{\cdot\}$ indica parte fraccionaria. Si n es primo, entonces $a(n) = n$, si n no es primo $a(n) = 0$. Es importante resaltar que tal fórmula sólo opera sobre un valor de n dado; además, la suma de las partes fraccionarias en el denominador implica un ciclo de 1 hasta n. No es equivalente, por ejemplo, a la criba de Eratóstenes, la cual entrega todos los números primos que sean menor o igual a una n dada. Además, si n es grande, se requerirán n divisiones, es decir, más divisiones que las que requiere la prueba de primalidad del Algoritmo 1, en el cual el número de divisiones es siempre menor o igual a la raíz cuadrada de n.

Para comprobar la fórmula, se desarrolló una función para Octave, según el Algoritmo 2. Esta función recibe como parámetro de entrada el valor de n a probar, y entrega un valor n si es primo y 0 en caso contrario. Además requiere de un paso intermedio para extraer la parte fraccionaria de las divisiones de n/i, donde i es el vector $i = [1, \ldots, n]$.

Algoritmo 2: función FCamacho

Ingresa: n
Regresa: p
Datos: Prueba $a(n)$
Def `subFraccion(`x`)`**:**
    fraccion = x-parte entera de x `// Define subrutina`
    devolver fraccion

Funcion `FCamacho(`n`)`**:**
    $i \leftarrow [1, \ldots, n]$ `// prepara vector` i
    f=subFraccion(n/i) `// extrae parte fraccionaria`
    d=redondeaArriba(f)
    a=redondeaAbajo$\left(\frac{2}{n-sum(d)}\right)$

Queda claro que, en esa fórmula, las operaciones adicionales de extración de parte fraccionaria, redondeo ascendente y descendente no son operaciones ordinarias, es decir, operaciones aritméticas o algebraicas, y que requieren para su ejecución rutinas programadas en algún lenguaje de cómputo.

Para generar una secuencia de números primos como la secuencia OEIS A061397 [7], se debe aplicar la (3) sucesivamente para cada valor de n. Por ejemplo, para valores de n desde 1 a 700 se aplicaría 700

veces. Para este caso, en el Apéndice B se muestra su ejecución sólo para los primeros 100 números naturales. Además, en los Apéndices se incluyen, para que el lector pueda comprobarlo, los Códigos para Octave y algunos resultados que pueden ser comparados con las secuencias de OEIS A061397 [7].

Tanto la prueba de Wilson como la fórmula (3) no son una buena opción desde el punto de vista computacional, ya que requieren, en el primer caso, calcular un factorial y en el segundo, dividir n/i, con $i = 1, \ldots, n$, es decir un bucle, además de las funciones de redondeo y una función para extraer la parte fraccionaria del cociente $[\frac{n}{i}]$.

Al revisar la (3) se encuentra que el término del lado derecho de la ecuación, despúes de n, sólo tiene dos valores, 0 y 1. Desde este momento se observa la similitud con la prueba de primalidad de la función dada en el Código 1.

3.3. Simplificando el código

Cabe hacer notar que primero se reprodujo el resultado de la (3) para dar contexto a un análisis desde el punto de vista computacional. Bajo este contexto, se vio la posibilidad de reducir o eliminar, en el código, algunas operaciones. La primera modificación es reducir el rango de valores de $i = 1, \ldots, n$ al rango $i = 2, \ldots, n-1$, ya que sólo requiere contar aquellos valores que no son divisores de n.

Se puede observar que la parte importante de (3) es el denominador, así que sólo es necesario probar si se cumple

$$d(n) = n - \sum_{i=2}^{n-1} \left\lceil \left\{ \frac{n}{i} \right\} \right\rceil = \begin{cases} 2, & \text{si } n \text{ es primo} \\ > 2, & \text{si } n \text{ no es primo} \end{cases} \tag{4}$$

que se puede ver de la siguiente manera:

Observación 1

Si n es un número primo, sólo tiene dos divisores (enteros) y todo el resto de los divisores con $i = 2, \ldots, n-1$ serán fraccionarios, así que la suma de divisores fraccionarios será $n-2$, por lo tanto, para n primo, $d(n) = 2$ siempre. [a]

[a] Note que $\frac{n}{1}$ y $\frac{n}{n}$ son divisibles, y no hace falta hacer esa prueba, como la hace (3).

Una explicación, paso a paso (por el código que es necesario) de esa prueba simplificada, que se hace más adelante, es: dividir $\frac{n}{i}$, con $i = 2, \ldots, n-1$, extraer la parte fraccionaria, hacer un redondeo ascendente (que entregará 1's cuando no sea cero) y sumar, para finalmente comprobar si es igual a 2. En pocas palabras la pregunta clave es:

¿n menos el número de veces que n no es divisible es igual a 2?

Esta prueba funciona para toda $n > 1$. Se muestran algunos ejemplos para comprobar que sólo se necesita calcular $d(n)$ y no $a(n)$ como lo propone la (3).

La ecuación (4) se muestra con los siguientes Ejemplos.

Ejemplo 1

Sea $n = 5$, entonces $\{\frac{5}{2}, \frac{5}{3}, \frac{5}{4}\} = \{2.5000, 1.6667, 1.2500\}$
parte fraccionaria = $\{0.50000, 0.66667, 0.25000\}$
redondeo ascendente = $\{1, 1, 1\}$
$\#\{n$ no divisible$\}$ = suma del redondeo = 3
$n -$ suma $= 5 - 3 = 2$. Por lo tanto $n = 5$ es primo.

Ejemplo 2

Sea $n = 15$, entonces $\{\frac{15}{2}, \frac{15}{3}, \frac{15}{4}, \ldots, \frac{15}{14}\} =$
$\{7.5000, 5.0000, 3.7500, 3.0000, 2.5000, 2.1429, 1.8750, 1.6667, 1.5000, 1.3636, 1.2500, 1.1538, 1.0714\}$
parte fraccionaria = $\{0.5000,\ 0.0000,\ 0.7500,\ 0.000,\ 0.5000$
$0.14286,\ 0.8750,\ 0.66667,\ 0.500,\ 0.36364,\ 0.250,\ 0.15385,\ 0.07143\}$
redondeo ascendente = $\{1,\ 0,\ 1,\ 0,\ 1,\ 1,\ 1,\ 1,\ 1,\ 1,\ 1,\ 1,\ 1\}$

$\#\{n$ no divisible$\}$ = suma del redondeo = 11
$n -$ suma $= 15 - 11 = 4$. Por lo tanto $n = 15$ NO es primo.

Aunque toda prueba de primalidad debería comenzar por verificar si $n > 2$ y además impar[1], esta prueba, la ecuación (4), resiste una entrada con n par, como se muestra en el siguiente Ejemplo:

Ejemplo 3

Sea $n = 4$, entonces $\{\frac{4}{2}, \frac{4}{3}\} = \{2.000,\ 1.3333\}$
parte fraccionaria = $\{0.0000,\ 0.3333\}$
redondeo ascendente = $\{0,\ 1\}$
$\#\{n$ no divisible$\}$ = suma del redondeo = 1
$n -$ suma $= 4 - 1 = 3$. Por lo tanto $n = 4$ NO es primo.

Cabe aclarar que la forma simplificada de (3), dada en (4), todavía requiere, además de dividir $\frac{n}{i}$ con $i = 2, \ldots, n-1$: una función para extraer la parte fraccionaria, un redondeo ascendente y su suma, para verificar finalmente si es igual a 2.

Una segunda simplificación al código que evita la función de extracción de la parte fraccionaria y la operación de redondeo ascendente en la "fórmula" simplificada (4) se puede escribir en una sola línea de código utilizando dos funciones incluidas en Octave, una es la función de suma (llamada `sum`) y la otra la función residuo (llamada `rem`); esta última requiere el valor de n y el divisor a probar, la cual entrega 0 si es un divisor entero. Para dar una clara idea de la simplificidad del código, se considera conveniente mostrar su realización en Octave quedando así: `d=n-sum((rem(n,i)~=0))`, donde el signo `~=` significa "a condición de $\neq 0$", y ya sólo hay que verificar si d es igual a 2. En el Apéndice C se incluye el código correspondiente, el cual funciona para toda $n > 1$.

3.4. Nueva simplificación a (3)

Para evitar los redondeos y la extracción de la parte fraccionaria, se intentó otro enfoque:

1. Contar lo contrario, es decir, el número de veces que $[\frac{n}{i}]$ sí es divisible[2].

[1] Recuerde que $n = 2$ es el primero y único número primo par.
[2] Es decir, tiene parte fraccionaria cero.

2. Calcular el complemento de esa suma, para regresar a la misma condición de la (4).

Ahora se hace la suma de residuos que son cero. Con eso se obtiene el número de veces que $[\frac{n}{i}]$ sí es divisible. Para calcular el complemento de la suma de la ecuación (4), se requiere cumplir:

$$n - \text{longitud}(i) - \sum(\text{residuos} = 0) = 2 \tag{5}$$

Este cambio en el código funciona del mismo modo que el anterior, es decir, requiere comprobarse si es igual a 2, como se muestra en el siguiente ejemplo:

Ejemplo 4

Para $n = 5$ se obtiene:

$$\begin{aligned} &5 - \text{ longitud}(\ [2,3,4]\ ) \ - \textstyle\sum_{i=2}^{4} \left([\frac{5}{i}] \text{ con residuo } 0\right) = 2 \\ &5 - 3 - 0 = 2 \\ &2 = 2 \end{aligned} \tag{6}$$

Por lo tanto n=5 sí es primo.

Pero tenemos que la longitud del vector i es igual a $n - 2$, entonces la ecuación (4) queda como

$$n - (n-2) - \sum_{i=2}^{n-1} \left(\left[\frac{n}{i}\right] \text{ con residuo } 0\right) = 2$$
$$2 - \sum_{i=2}^{n-1} \left(\left[\frac{n}{i}\right] \text{ con residuo } 0\right) = 2$$

Por simple álgebra, n es primo ssi

$$\sum_{i=2}^{n-1} \left(\left[\frac{n}{i}\right] \text{ con residuo } 0\right) = 0$$

es decir, n es primo si el número de veces que $\left[\frac{n}{i}\right]$ es divisible es cero. Esta simplificación es notable porque muestra que la (3) se reduce a la prueba de primalidad más simple, es decir, a la prueba de primalidad por división, y sólo eso. Incluso se apega completamente a la definición de número primo: p es primo si solamente es divisible entre 1 y entre p. Esta simplificación al código original de la fórmula se muestra en el Apéndice D.

3.5. Justificación matemática

Aunque se mostró que la simplificación del código propuesto como implementación en Mathematica por [3] de una "fórmula que genera números primos", se reduce a la prueba de primalidad por división, debe notarse que la razón matemática de la (3) se basa en la función divisor.

La función divisor $\sigma_x(n) = \sum_{d|n} d^x$ fue estudiada por Ramanujan [8]. Para el caso $x = 0$, $\sigma_o(n)$ se denota como $d(n)$ o $\tau(n)$. Cuando n es un número primo se tiene que $d(n) = 2$. Este valor es la base de la (3), de modo que sabiendo que para n primo, el denominador $d(n) = 2$, se debe cumplir $a(n) = n \times 1$, y se logra con el cociente $\left(\frac{2}{d(n)}\right) = 1$. En el caso en que n no es primo, $d(n) > 2$, y se requiere $a(n) = n \times 0$, lo cual se logra con la operación de redondeo descendente, puesto que $\left(\frac{2}{d(n)>2}\right)$ será un número menor a 1, que se redondea a 0.

Así pues, la propiedad de la función $d(n)$ se aprovecha de manera aritmética, mientras que en código se usaría una operación condicional `if`. Como última simplificación al cálculo de $a(n)$, se propone hacerlo con la función `isprime`, la cual, al igual que la parte derecha después de n de la fórmula (3), entrega un valor 1 o 0. Por ejemplo, al ejecutar `isprime(5)` el resultado es 1, mientras que `isprime(4)` entrega un resultado 0. Entonces "generar" números primos se puede lograr creando un vector $n = [1, \ldots, N]$, donde N será el mayor valor de n a probar, seguido de la instrucción `a=n*isprime(n)`. Esta sola instrucción genera la lista de números primos y ceros, como lo hace la (3). Para completar los códigos propuestos, en el Apéndice E, se muestra este último código.

Cabe hacer la siguiente aclaración: en aras de mantener la formulación presentada en [3], se siguió un enfoque computacional al analizar la realización o implementación de la (3), ya que esa fórmula requiere, necesariamente, su realización en un ambiente de programación. Sin embargo, se puede observar lo siguiente:

Observación 2

Por definición, un número primo sólo tiene dos divisores, así que dividir 2 entre el número de divisores de un número primo es obviamente igual a 1, y multiplicar 1 por n entrega al primo n. Así pues, la (3) se reduce a encontrar cuántos divisores tiene n, y una forma de hacerlo, sobre todo para números no demasiado grandes, es por simple división. Esta puede ser tan corta como dividir n entre los primeros números como el 2, el 3 y continuar hasta $\sqrt{n}$ si es necesario. Note que, la prueba de primalidad por división, no requiere el paso previo de extraer la parte fraccionaria de la división. En resumen, se muestra que la (3) es, en todo caso, un *detour* evitable.

4. Comparaciones de tiempo de cómputo

En esta sección se presentan dos tipos de pruebas para los diferentes códigos analizados. La primera de ellas trata sobre el tiempo de ejecución cuando se utiliza un solo valor de n. La razón de esta prueba es porque, como se ha explicado, las pruebas de primalidad requieren diferentes cantidades de operaciones, y estas dependen del valor de n.

Se propone la Figura 1, en lugar de una tabla, para mostrar una comparación de tiempos de ejecución para algunos números primos. `Ta(n)` denota la fórmula (3), `Txdiv` se refiere a la prueba por división, `Tisprime` emplea la función `isprime`, y `Tsimple` denota la simplificación propuesta como Código 5.

Mientras que la fórmula (3) requiere n divisiones, además de otras operaciones, la prueba por división requiere un número menor de operaciones, es decir, elimina de la prueba los valores pares y reduce los divisores i de $i = 2$ a $i \leq \sqrt{n}$.

El último número $n = 100005$ no es primo, y se puede notar que la prueba por división reduce significativamente su tiempo de ejecución ya que al ser divisible entre 3, el código termina la prueba en la primera instrucción `if`, con un tiempo mucho menor que las otras que requieren hacer las 100 mil divisiones. En la Figura 2 se incluye en la prueba un valor de 100 millones, entonces la diferencia entre tiempos de ejecución aumenta.

Finalmente, en una prueba más completa se generaron secuencias $a(n)$ tipo OEIS A061397, con longitudes de 10 hasta 50 mil términos. Las secuencias no se muestran debido a su longitud, pero en cambio se comparan los tiempos de cómputo, en segundos, entre las versiones que se elaboraron en Octave. En la Tabla 1 se muestran los resultados y se puede destacar que el empleo de la función `isprime` ofrece los menores tiempos de cómputo cuando se tiene que generar toda la secuencia.

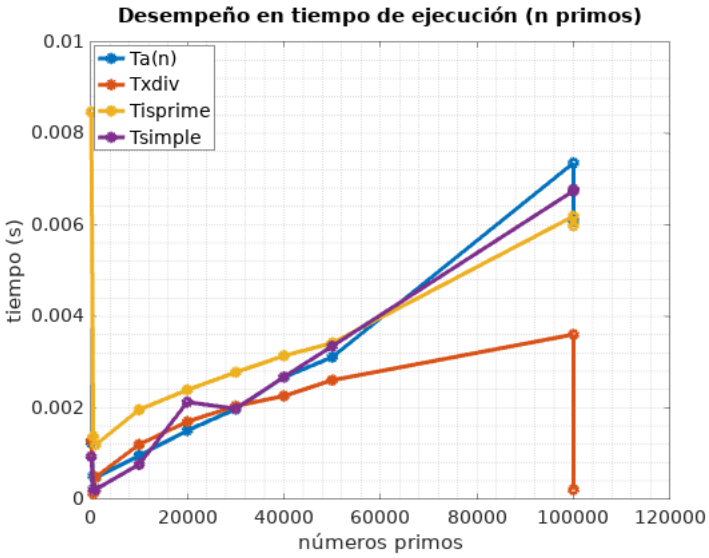

Figura 1: Tiempos de ejecución para varias versiones, n=[100 500 1009 10007 20011 30011 40009 50021 100003 100005]. (Elaboración propia).

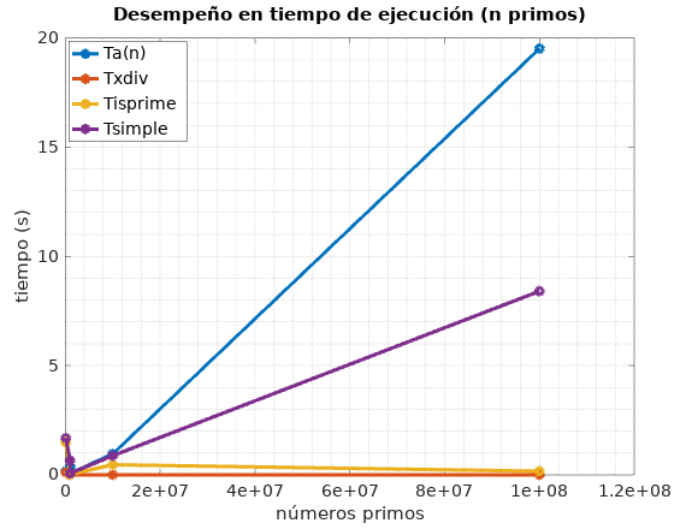

Figura 2: Tiempos de ejecución para varias versiones, n=[100 500 1009 10007 20011 30011 40009 50021 100003 1000003 1000005 10000003 100000055]. (Elaboración propia).

5. Conclusiones

En este trabajo se realizó un análisis computacional a una fórmula presentada en [3] como generadora de números primos.

Se modificó su código (anteriormente propuesto para Mathematica) y al mismo tiempo se pudo demostrar que tal fórmula se puede ver como una prueba de primalidad más básica, es decir, la prueba de primalidad por división. Al modificar el código se fue modificando la fórmula, demostrando que ella contiene más operaciones de las necesarias para realizar la prueba que se propone ejecutar, es decir, sólo comprueba si n es primo o no.

Se compararon los tiempos de ejecución y también se pudo demostrar que la prueba por división que considera divisores hasta $i \leq \sqrt{n}$, para los valores mostrados, requiere menos tiempo de ejecución, cuando se compara con la fórmula original, ya que reduce el número de operaciones necesarias. Al

Tabla 1: Tiempos de ejecución de generación de secuencias tipo OEIS A061397. (Elaboración propia).

n-términos	t_$a(n)$	t_xdiv	t_isprime	t_simplif
10	**0.003489**	**0.00194**	0.0089588	0.0019701
100	**0.025152**	**0.010294**	0.00058699	0.011077
1000	**0.30092**	**0.13473**	0.0014119	0.1401
2000	**0.61962**	**0.29723**	0.0047848	0.34142
3000	**1.0686**	**0.48229**	0.0016379	0.61265
4000	**1.4951**	**0.67442**	0.0022631	0.94331
5000	**1.9941**	**0.87963**	0.0020528	1.3135
6000	**2.536**	**1.0938**	0.0021122	1.7831
7000	**3.1969**	**1.3242**	0.0022399	2.2616
8000	**4.1602**	**1.5384**	0.002248	2.7995
9000	**5.331**	**1.7764**	0.0023069	3.4205
10000	**5.7463**	**2.0228**	0.0023279	4.1124
20000	**18.381**	**4.8359**	0.0034339	14.265
30000	**37.101**	**8.2784**	0.0046279	30.785
40000	**56.165**	**11.399**	0.0065022	53.742
50000	**86.483**	**15.024**	0.0066781	83.49

generar las secuencias tipo OEIS A061397, si n es muy grande, habrá muchos valores que no son primos, como se nota en la cada vez más grande cantidad de ceros; es ahí donde la prueba de primalidad por división toma ventaja sobre la fórmula (3).

Se puede concluir que la "fórmula que genera números primos" es en realidad una prueba de primalidad por división no optimizada, pues, por ejemplo, no evita la prueba de valores pares de n.

6. Bibliografía

[1] P. Ribenboim, *The New Book of Prime Number Records*, 3rd ed., 1995.

[2] W. Mills, "A prime-representing function," *Bull. Amer. Math. Soc.*, vol. 53, no. 604, 1947.

[3] J. d. J. Camacho, "Una fórmula que genera números primos," *Revista digital Matemática, Educación e Internet*, vol. 22, no. 1, 2022.

[4] J. de Jesús Camacho, "¿es posible encontrar una fórmula que permita comprobar si un número es primo?" https://www.masscience.com/2019/10/27/es-posible-encontrar-una-formula-que-permita-comprobar-si-un-numero-es-primo/.

[5] J. W. Eaton, D. Bateman, S. Hauberg, and R. Wehbring, "Octave version 4.2.1 manual: a high-level interactive language for numerical computations," https://www.gnu.org/software/octave/doc/v4.2.1/.

[6] W. F. Mora, *INTRODUCCIÓN a la TEORÍA DE NÚMEROS. Ejemplos y algoritmos*, 2nd ed., 2010.

[7] "A061397 characteristic function sequence of primes multiplied componentwise by n, the natural numbers," https://oeis.org/A061397, Sloane, N. J. A. (2021) On-Line Encyclopedia of Integer Sequences.

[8] S. Ramanujan, "On certain trigonometric sums and their applications in the theory of numbers," *Transactions of the Cambridge Philosophical Society*, vol. 22, no. 15, pp. 259–276, 1918.

Apéndices

A. Prueba de primalidad básica

Dado un valor deseado n, se realiza la prueba de primalidad por división (limitada a $\sqrt{n}$) ejecutando la función `esprimo(n)` y se obtiene 1 si es un número primo o 0 en caso contrario.

```
function p=esprimo(n)
% prueba de primalidad por divisiOn
if ((n==2)||(n==3)) p=1; return
  elseif ((rem(n,2)==0)||(n<2))
  p=0; return
endif
for i=3:2:round(sqrt(n))
  if rem(n,i)==0
  p=0; return
  endif
endfor
p=1;
return
```

Código 1: esprimo.m

B. Prueba de la fórmula (3)

Para comprobar la fórmula, se desarrolló una función para Octave, según el Código 2.

```
function a=funcionCamacho(n)
% Referencia: J. de J. Camacho Medina, (2019).
% Regresa a=n, si es primo, lo contrario a=0
% GMdeL (2021)
fraccion = @(x) x - fix(x);
i=[1:n];
d=ceil(fraccion(n./i));
a=(n*floor(2/(n-sum(d))));
```

Código 2: funcionCamacho.m

La `funcionCamacho` se puede llamar desde otro código, especificando un intervalo de valores y agregando una rutina de impresión de resultados. El Código 3 muestra el llamado a la función, donde se aplica para una $n = 2, \ldots, 100$.

```
% Prueba funcionCamacho para n de 1 a N
clear all
% Define longitud de la secuencia N
N=100;
% aplica funciOn para cada valor de n
a=[];
for n=2:N
  a(n)=funcionCamacho(n);
endfor
% imprime secuencia
for k=1:length(a)
printf("%i,",a(k))
endfor
```

Código 3: PruebafCamachoUNO.m

El resultado mostrado en el espacio de trabajo de Octave es:

```
 >> PruebafCamachoUNO
0,2,3,0,5,0,7,0,0,0,11,0,13,0,0,0,17,0,19,0,0,0,23,0,0,0,0,0,29,0,31,0,0,0,0,0,37,0,0,0,
41,0,43,0,0,0,47,0,0,0,0,0,53,0,0,0,0,0,59,0,61,0,0,0,0,0,67,0,0,0,71,0,73,0,0,0,0,0,79,
0,0,0,83,0,0,0,0,0,89,0,0,0,0,0,0,0,97,0,0,0,>>
```

C. Primera simplificación a la fórmula (3)

Considerando solamente el denominador, el código queda de la siguiente manera:

```
% COdigo simple de la (3) y (4)
n=input("Ingresa n a probar = ");
i=[2:n-1];
if (n-sum((rem(n,i)~=0)))==2
a=n;
else
a=0;
endif
```

Código 4: CamachoSimplePrimo.m

Note que la parte que calcula el denominador de la (3) es una línea de código, es decir, en la instrucción condicional `if`, con: `n-sum((rem(n,i~=0)))==2`.

D. Segunda simplificación al código

Apegándose a la fórmula (3), el código se reescribe como:

```
function a=SimpleMaxCamacho(n)
% Prueba por divisi0n de n/[2:n-1] si n es primo
% obtenida al simplificar Camacho
% GMdeL (junio 2021)

i=[2:n-1];
if sum(rem(n,i)==0)==0
a=n;
else
a=0;
end
```

Código 5: SimpleMaxCamacho.m

Esta función se llama para un solo valor de n. Todavía se realizan $n-2$ divisiones y encuentra si la suma de divisores es cero, en cuyo caso regresa `a=n`.

E. Uso de la instrucción `isprime` de Octave

```
function a=aprimes(N)
n=[1:N];
a=n.*(isprime(n));
```

Código 6: aprimes.m

El Código 6 se puede llamar sin usar un bucle `for`, como sí se requiere en los casos anteriores. El vector de 1 a N se crea en el interior de la función. Al ingresar el valor N, el código entrega toda la secuencia

de números primos y ceros hasta N. Cabe aclarar que la instrucción `Table` del código en Mathematica propuesto en [3] realiza un bucle `for` de 2 a 700, es decir, la fórmula prueba cada número, uno a la vez. Al ejecutar el Código 6, la instrucción `aprimes(100)` produce la misma secuencia que se obtuvo con el Código 3.

A manera de comprobación, se muestra la salida de la consola de trabajo de Octave de la secuencia de 50 mil términos, obtenida al ejecutar `aprimes(5e4)`. Sólo se muestran los valores para $n = 1001$ a 1500, aunque la lista podría continuarse hasta 50 mil. Puede observar la gran cantidad de ceros y comprobar si lo desea que el resto de los números son primos.

```
0,0,0,0,0,0,0,0,1009,0,0,0,1013,0,0,0,0,0,1019,0,1021,0,0,0,0,0,0,0,0,1031,0,1033,
0,0,0,0,0,1039,0,0,0,0,0,0,0,0,0,1049,0,1051,0,0,0,0,0,0,0,0,1061,0,1063,0,0,0,0,0,
1069,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1087,0,0,0,1091,0,1093,0,0,0,1097,0,0,0,0,0,
1103,0,0,0,0,0,1109,0,0,0,0,0,0,0,1117,0,0,0,0,0,1123,0,0,0,0,0,1129,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,1151,0,1153,0,0,0,0,0,0,0,0,1163,0,0,0,0,0,0,0,1171,0,0,0,
0,0,0,0,0,0,1181,0,0,0,0,0,1187,0,0,0,0,0,1193,0,0,0,0,0,0,0,1201,0,0,0,0,0,0,0,0,0,0,
0,1213,0,0,0,1217,0,0,0,0,0,1223,0,0,0,0,0,1229,0,1231,0,0,0,0,0,1237,0,0,0,0,0,0,0,
0,0,0,0,1249,0,0,0,0,0,0,0,0,0,1259,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1277,0,1279,0,0,
0,1283,0,0,0,0,0,1289,0,1291,0,0,0,0,0,1297,0,0,0,1301,0,1303,0,0,0,1307,0,0,0,0,0,
0,0,0,0,0,0,1319,0,1321,0,0,0,0,0,1327,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,1361,0,0,0,0,0,1367,0,0,0,0,0,1373,0,0,0,0,0,0,0,1381,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,1399,0,0,0,0,0,0,0,0,0,1409,0,0,0,0,0,0,0,0,0,0,0,0,0,1423,0,0,0,
1427,0,1429,0,0,0,1433,0,0,0,0,0,1439,0,0,0,0,0,0,0,1447,0,0,0,1451,0,1453,0,0,0,0,
0,1459,0,0,0,0,0,0,0,0,0,0,0,1471,0,0,0,0,0,0,0,0,0,1481,0,1483,0,0,0,1487,0,1489,0,
0,0,1493,0,0,0,0,0,1499,0,>>
```