Mejora al algoritmo optimizado para la detección de un número primo usando programación funcional

Improve to optimized algorithm for prime number detection using functional programming.

Orlando Arboleda Molina
Facultad de Ingeniería, Departamento de Operaciones y Sistemas, Universidad Autónoma de Occidente, Colombia
oarboleda@uao.edu.co

***Resumen*— Tomando como base una optimización realizada en Scheme del algoritmo simple para determinar si un número de entrada es primo, se propone una mejora que incrementa su desempeño y garantiza su correctitud.**

***Palabras clave*— Número primo, Test de primalidad, Programación funcional, Scheme, Racket, complejidad**

***Abstract*— Taking as base a realized optimization in Scheme of the simple algorithm to determine if a number of entry is prime, I propose an improvement that increases his performance and guarantees his correctitude.**

***Key Word* — Prime number, Primality test, Functional programming, Scheme, Racket, Asymptotic notation**

I. INTRODUCCIÓN

El análisis de los números primos y la búsqueda de algoritmos que los detecten fácilmente ha sido siempre una preocupación a nivel de programación de computadores. Primeramente porque las mismas características matemáticas de los números primos han sido un factor que ha seducido no solo ha matemáticos a lo largo de la historia sino a programadores en los sesenta años que lleva la programación como área definida de conocimiento. De otra parte la gran cantidad de aplicaciones que se le han encontrado a los números primos, precisamente por sus características matemáticas, ha permitido encontrar en ellos el soporte para poder hacer que algunos procesos sean mucho más seguros y que temas tan modernos como la encriptación tengan componentes matemáticos de gran solidez [1].

Los test (o chequeos) de primalidad son algoritmos para determinar si un número de entrada es primo. Aun a niveles puramente académicos, se espera que estos algoritmos sean sencillos de entender, eficientes y correctos. *Eficientes* en el sentido que su ejecución se pueda dar en tiempos razonables (que su orden de ejecución sea preferiblemente polinomial) y *correctos* en el hecho que las salidas sean las esperadas para los valores de entrada dados [4].

En la literatura han sido documentadas propuestas algorítmicas para test de primalidad del algoritmo simple que es de muy fácil entendimiento, acrecentando su comprensión al ser implementado en el lenguaje funcional Scheme [2], al cual posteriormente se le propusieron dos mejoras graduales de optimización del tiempo de desempeño, pero no se realizó el estudio de correctitud ni se formalizo su complejidad [1].

En el presente artículo se toma como base la segunda optimización del algoritmo simple, para proporcionar una mejora que hará que el nuevo algoritmo mantenga su eficiencia, se garantice su correctitud y se proporcione su análisis de complejidad. El algoritmo presentado es implementado usando el paradigma de programación funcional usando el lenguaje de programación multi-paradigma Racket (sucesor del lenguaje de programación Scheme), usando como entorno de desarrollo DrRacket versión 5.2.1, configurado como estudiante avanzado

II. ALGORITMO BASE PARA LA OPTIMIZACION DE LA DETECCION DE UN NUMERO PRIMO

Un número primo es aquel número natural que tiene solamente como divisores exactos al número 1 y el mismo número en cuestión. De la anterior definición se desprende que no debería existir ningún otro divisor en el rango [2,n-1], el cual es el fundamento del test de primalidad del algoritmo que denominaremos simple [2].

Una optimización a resaltar para el test de primalidad del algoritmo simple, fue propuesta por el matemático y astrónomo marroquí *Ibn Al Banna*, quien logro determinar que basta con

Fecha de Recepción: 17 de Enero de 2013
Fecha de Aceptación: 28 de Diciembre de 2014

determinar que no tenga divisores primos en el rango [2, √n]. A nivel computacional, se reduce el rango de búsqueda, pero se plantea el hecho de seguir computando los primos inferiores al número a revisar [3].

Ese principio fue usado en el algoritmo de optimización de detección de un numero primo, que usaremos como base, el cual *no* valida que no exista ningún divisor primo en el rango [2, √n] (que sería lo optimo), pero *si* valida que no exista ningún otro divisor en ese mismo rango. El algoritmo que puede apreciarse en la Figura 1, fue implementado utilizando programación funcional y al lenguaje DrScheme, pero no se realizó el análisis de su complejidad (por que no era el fin en su momento), ni se garantizó la correctitud [1].

```
(define (esmulti a b)
  (if (= (remainder a b) 0 )
      1
      0 ))

(define (cuentamulti n d)
  ( if (= d ( sqrt n) )
       0
       (if (= (esmulti n d) 1)
           1
           (+ 0 (cuentamulti n (+ d 1))))))

(define (primo n)
  (cuentamulti n 2 ) )

(define (esprimo n)
  (if (= (primo n) 1)
      ;; Se evalúa si se encontró el 1er divisor
      (display "no es primo")
      (display "es primo")
      ))
```

Figura 1. Algoritmo de optimización base para la detección de un número primo.

El algoritmo mostrado no es correcto en el cómputo de ciertos valores. Específicamente, se dan los siguientes casos:

No detección de algunos primos
Números primos como los indicados en la Figura 2, son reportados por el algoritmo como que no lo son.

2, 3, 5, 7, 11, 13, 17, 19 ….

Figura 2. Algunos números primos reportados incorrectamente.

Esto se da, porque en la función *cuentamulti* el computo de la raíz del número de entrada, produce un número real con parte decimal diferente a cero[1], por tanto la comparación de ese *resultado obtenido* y el valor entero *d* siempre será falsa, lo cual provocará que al ser primo el número de entrada, se invoque sucesivamente esta función con todos los valores sucesivos en el rango [2,**n**], aumentándose el tiempo de ejecución del algoritmo por que se realizan siempre (n-1) comparaciones.

El computo de (*esprimo* 17)
Tendrá las siguientes invocaciones y dará como salida:

(*cuentamulti* 17 2) salida = 0+salida$_1$
(*cuentamulti* 17 3) salida$_1$ = 0+ salida$_2$
(*cuentamulti* 17 4) salida$_2$ = 0+ salida$_3$
(*cuentamulti* 17 5) salida$_3$ = 0+ salida$_4$
…
(*cuentamulti* 17 16) salida$_{14}$ = 0+ salida$_{15}$
(*cuentamulti* 17 17) salida$_{15}$ = 1

Salida = 0 + 0 + …..+ 0 + salida$_{15}$ = 1

Figura 3. Invocaciones recursivas para un valor de ejemplo no detectado realmente como primo.

Tomando como ejemplo el caso mostrado en la Figura 3, en donde se describen todos los llamados recursivos realizados, se logra apreciar que luego de las (n-1) comparaciones, la invocación inicial a la función *cuentamulti* tendrá como valor de salida 1, lo cual hará que la función *esprimo* indicará que el valor de entrada *n* no es primo.

Reporta primos que no lo son
Estos se dan con valores como los indicados en la Figura 4.

4, 9, 25, ….

Figura 4. Algunos números reportados como primos sin serlo.

Todos estos valores tienen una norma y es que su divisor es precisamente su raíz. El detalle es que la función *cuentamulti* cuando recibe como posible divisor un valor *d* que corresponde a la raíz del número *n* al que se le calcula la primalidad, termina su ejecución inmediatamente indicando que no existe divisor, sin verificar que si *n* es o no múltiplo de *d.*

El computo de (*esprimo* 25)
desencadenaría las siguientes invocaciones:

(*cuentamulti* 25 2) salida = 0+salida$_1$
(*cuentamulti* 25 3) salida$_1$ = 0+ salida$_2$
(*cuentamulti* 25 4) salida$_2$ = 0+ salida$_3$
(*cuentamulti* 25 5) salida$_3$ = 0

Salida = 0 + 0 + …..+ 0 = 0

Figura 5. Invocaciones recursivas para un valor de ejemplo reportado como primo sin serlo.

Tomando como ejemplo el caso mostrado en la Figura 5, en donde se describen todos los llamados recursivos realizados, se logra apreciar que estas si se dan en el rango [2, √n], pero la invocación inicial a la función *cuentamulti* tendrá como valor

[1] Por ejemplo, la raíz cuadrada de 17 es 4.123105625617661

de salida 0, lo cual hará que la función *esprimo* indicará que el valor de entrada *n* si es primo.

III. ALGORITMO PROPUESTO

En el presente artículo se toma como base la segunda optimización del algoritmo simple para determinar la primalidad de un número, y se proporciona la mejora que puede apreciarse en la Figura 6, que hará que el nuevo algoritmo mantenga o incremente su eficiencia, se garantice su correctitud, y se proporciona su correspondiente análisis de complejidad. El algoritmo presentado también es implementado usando el paradigma de programación funcional usando el lenguaje de programación multi-paradigma Racket (sucesor del lenguaje de programación Scheme), usando como entorno de desarrollo DrRacket versión 5.2.1, configurado como estudiante avanzado

En el algoritmo propuesto se retoman muchas de las funciones del algoritmo base y se incluyen nuevas funciones que garantizaran su correctitud y mejoran su tiempo de desempeño.

El algoritmo propuesto, que corresponde a la función *esprimo* obtenida del algoritmo base, tiene como consigna determinar que el número proporcionado es primo si no logra tener un divisor en el rango [2, √n], siendo no primo al confirmarse el primero de sus divisores en dicho rango.

Con el fin de facilitar su entendimiento y sentar las bases para el posterior análisis de correctitud y de complejidad, se detallan cada una de las funciones del algoritmo propuesto:

Función: esmulti
Tomada del algoritmo base, es la función más básica del algoritmo propuesto, que a partir de un par de números enteros de entrada, determina si el segundo de ellos es divisor del primer número ingresado, en cuyo caso retorna 1. De no serlo, su salida es 0.

Función: tienedivisores
Es una nueva función recursiva que corrige el mal comportamiento de la función c*uentamulti.* Sobre ella recae todo el peso del algoritmo propuesto, al ser la encargada de verificar la primalidad de un número *n* revisando los posibles divisores desde un límite inferior denominado *d*. Esta función tiene una respuesta positiva (retorna 1) si *n* tiene un divisor en el rango [d, √n] y una respuesta negativa (retorna 0) en caso de no tenerlo.

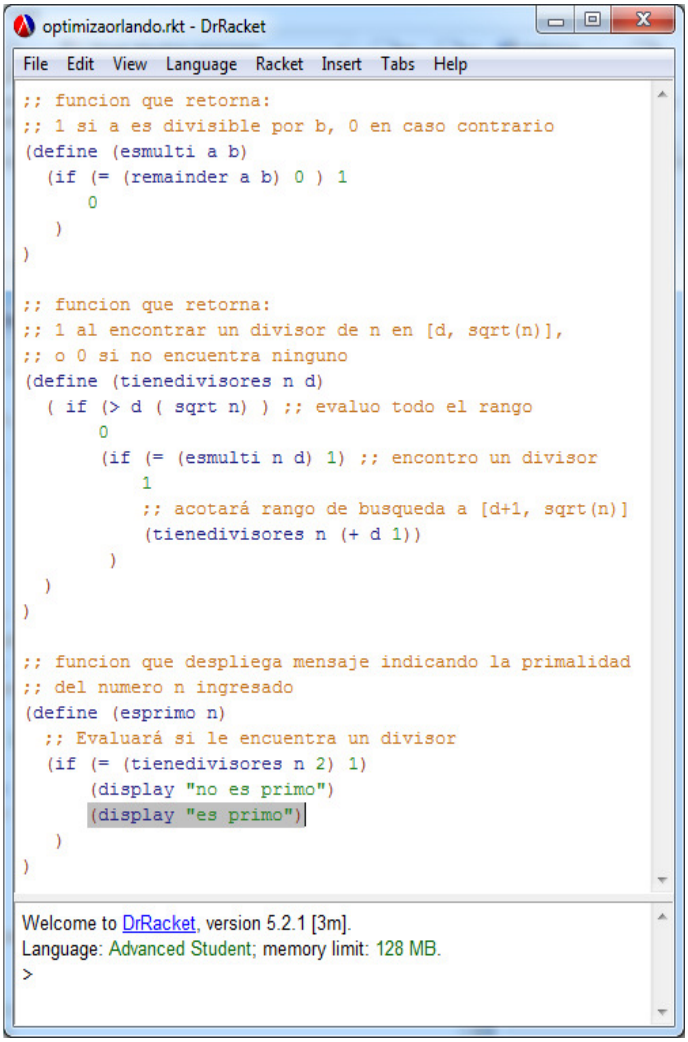

Figura 6. Algoritmo propuesto para determinar primalidad.

Básicamente, esta función valida si el límite inferior suministrado *d* es divisor de *n*, en cuyo caso detiene su ejecución y retorna la respuesta positiva, en caso contrario su respuesta dependerá de si *n* tiene un divisor solo en el rango [d+1, √n].

Funcionalmente esta función presenta dos mejoras notables:

1. Logra evaluar efectivamente como posible divisor a la raíz del número de entrada , lo cual se observa porque su condición de entrada es dejar de ejecutarse solo cuando *d* haya superado a la raíz del *n*.
2. No deja pendiente sumatorias inoficiosas de valores 0. Lo cual se refleja en que si no puede decidir para un valor *d* su salida será la que obtenga para la invocación para *d+1*, y así sucesivamente.

Función: esprimo
Corresponde al algoritmo y su intención es determinar si el número *n* ingresado es primo o no dependiendo de si se encontró o no respectivamente, un divisor de *n* en el rango [2, √n], como lo establece el algoritmo simple propuesto. La única diferencia con la función presente en el algoritmo base, es que ahora invoca es a la nueva función *tienedivisores*.

IV. CORRECTITUD Y COMPLEJIDAD DEL ALGORITMO PROPUESTO

Un aspecto fundamental en el diseño de un algoritmo, es que se verifique su correctitud y que su complejidad sea razonable para que sea viable su implementación, aun con los potentes equipos de cómputo que tenemos hoy en día.

Análisis de Correctitud

El algoritmo propuesto depende de la correctitud de las funciones auxiliares, con relación a lo que se espera de ellas, según lo definido en el apartado algoritmo propuesto.

Correctitud función: esmulti

La función *esmulti* es trivial y depende de la correctitud de la función *remainder* proporcionada por Racket la cual retorna el residuo de dividir el primer argumento por el segundo. En nuestro caso si no hay residuo (es decir este es 0) la función retorna 1 como se esperaba al describir previamente esta función auxiliar, dado que es solo invocada desde la función *tienedivisores* con *n* y *d* respectivamente, aseguramos que en todos los casos se esté usando para validar si *n* es divisible por un valor *d* que debiera estar siempre en el rango [d, √n].

Correctitud función: tienedivisores

La función *tienedivisores* es invocada inicialmente con *n* y *d.* Si el valor *d* es divisor de *n*, detiene su ejecución y retorna la respuesta positiva. Si el valor *d* no es un divisor intenta validar si *n* sigue teniendo divisores a partir del sucesor de *d*, hecho que puede generar invocaciones sucesivas con el próximo sucesor, y así sucesivamente, hasta que en el peor de los casos ese sucesor sea superior a √n, en cuyo caso detiene su ejecución y retorna una respuesta negativa. Si no se dio el peor caso, fue porque en una de esas llamadas uno de los sucesores de *d* fue divisor de *n*. El comportamiento descrito indica que el rango de búsqueda en el peor de los casos, se acota por las invocaciones enteras sucesivas con *d, d+1, d+2, ……..√n, √n+1* (punto de parada si en las sucesivas invocaciones no se encontró un divisor de n), siendo la función finita y cumpliendo con el propósito original de identificar posibles divisores en el rango [d, √n].

Para mostrar el comportamiento de esta función para valores que originalmente no se detectaban como primos, tomemos como ejemplo el caso mostrado en la Figura 7, en donde se describen para su comparación, todos los llamados recursivos realizados para una entrada similar a la de la Figura 3. Esta vez se logra apreciar que si se realizan √n invocaciones y que esta vez la salida de la función *tienedivisores* si es 0 indicando el número de divisores que encontró.

Para mostrar el comportamiento de esta función para valores que originalmente se detectaban como primos, tomemos como ejemplo el caso mostrado en la Figura 8, en donde se describen para su comparación, todos los llamados recursivos realizados para una entrada similar a la de la Figura 5. Esta vez se logra apreciar que luego de la invocación para √n se logra evaluar como posible divisor y al serlo la salida de la función *tienedivisores* será 1.

El computo de (*esprimo* 17)
Tendrá las siguientes invocaciones y dará como salida:

(*tienedivisores* 17 2)	salida = $salida_1$
(*tienedivisores* 17 3)	$salida_1$ = $salida_2$
(*tienedivisores* 17 4)	$salida_2$ = $salida_3$
(*tienedivisores* 17 5)	$salida_3$ = 0

Salida = 0

Figura 7. Invocaciones recursivas para un valor de ejemplo que es primo.

El computo de (*esprimo* 25)
desencadenaría las siguientes invocaciones:

(*tienedivisores* 25 2)	salida = $salida_1$
(*tienedivisores* 25 3)	$salida_1$ = $salida_2$
(*tienedivisores* 25 4)	$salida_2$ = $salida_3$
(*tienedivisores* 25 5)	$salida_3$ = 1

Salida = 1

Figura 8. Invocaciones recursivas para un valor de ejemplo reportado como primo sin serlo.

Correctitud función: esprimo

La función propuesta *esprimo* depende de una invocación inicial de la función *tienedivisores* con el *n* que se espera verificar es *primo o no*, y como límite inferior 2. Dado que la función *tienedivisores* es correcta y retorna 0 o 1, dependiendo de si no existían divisores en el rango [2, √n] o si encontró uno, respectivamente, está acorde con el mensaje desplegado finalmente que es el de ser primo o no serlo respectivamente, como se aprecia en la Figura 6.

Análisis de Complejidad

Dado que el algoritmo opera siempre en el rango [2, √n] su complejidad es realmente O(√n), complejidad superior a la obtenida en los algoritmos que suelen implementarse en los ámbitos académicos en los que se dan cursos de introducción a los algoritmos en los cuales los posibles divisores son buscados en el rango [2,n-1] o [2,n/2], siendo en ambos casos O(n).

Un aspecto que debe valorarse del diseño del algoritmo propuesto, en el que se prueban los sucesores a partir de 2, como posibles divisores de *n*, es que en el mejor de los casos, que se da por ejemplo cuando la entrada es par, en la única invocación a la función *esmulti* se determinara que el número ingresado es divisible por 2, e inmediatamente se desplegará que el número *no es primo*, siendo en este caso su complejidad θ(1). Este comportamiento es similar cuando el número evaluado es divisible por 3, 5, 7,…..etc. Nótese que en el mejor de los casos, no se evaluará por 4 porque de ser divisible por 4 lo fue también

por 2, detectando esto de manera prematura el algoritmo planteado.

V. CONCLUSIONES

Se logró identificar las causas del mal funcionamiento del algoritmo base optimizada para el cómputo de la primalidad de un número, y los casos en donde la complejidad se volvía lineal.

A partir del algoritmo base, se plantearon las modificaciones necesarias para generar un algoritmo finito, correcto y de buena complejidad para determinar si un número de entrada es primo o no.

Aunque el algoritmo planteado fue escrito utilizando el paradigma de programación funcional, es fácil tomarlo de referencia para construir el correspondiente algoritmo imperativo, cuyas versiones pueden ser usados para la enseñanza en los primeros cursos de algoritmia.

REFERENCIAS

[1] O. I. Trejos, "Algoritmo de optimización para la detección de un número primo basado en programación funcional utilizando Drscheme," Scientia et Technica, Año XVII, No. 47 , Apr. 2011.

[2] O. I. Trejos, "Determinación simple de un número primo aplicando programación funcional a través de DrScheme," Scientia et Technica, Año XVI, No. 45 , Aug. 2010.

[3] R. A. Mollin, "A Brief History of Factoring and Primality Testing B.C. (Before Computers)," Mathematics Magazine, Vol 75, No. 1 , pp. 18-29, Feb. 2002.

[4] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, "Introduction to algorithms," Massachusetts Institute of Technology, USA.

[5] C. Caldwell. (2012, Dec). *The Prime Pages: prime number research, records, and resources*. The University of Tennessee at Martin. [Online]. Available: http://primes.utm.edu/

[6] P. Van Toy, and S. Haridi, "Concepts, Techiques, and Models of Computer Programming,", Cambridge: The MIT Press, 2004.

[7] Racket. (2011, Dec). *Racket*. [Online]. Available: http://racket-lang.org