Algoritmo eficiente en la generación de una tabla de primalidad de números usando Programación Funcional*[1]

[An application of a simple and efficient algorithm to generate a prime number table using Functional Programming]

Omar Iván TREJOS BURITICÁ[2]

Recibo: 23.01.2014 – Aprobación: 19.06.2014

Resumen

En el presente artículo se acude a la Programación Funcional para generar una tabla de análisis de primalidad de números en un rango dado a partir del uso de un algoritmo que, por las necesidades del mismo objetivo, tiene características de ser eficiente. Se plantea la fundamentación de dicho algoritmo y además se aprovecha su estructura lógica para resolver el problema propuesto. El propósito de este artículo es mostrar una arista útil de la eficiencia algorítmica teniendo en cuenta las características tecnológicas modernas y los problemas que la matemática provee. Se hace uso del lenguaje de programación Scheme y se aprovechan sus potencialidades para manejo, tamaño y cálculo de datos. Se demuestra que, acudiendo a algoritmos eficientes y a una lógica muy simple, la tecnología computacional moderna puede ser de una inmensa utilidad para resolver problemas matemáticos.

* **Modelo para la citación de este artículo**:
TREJOS BURITICÁ, Omar Iván (2014). Estado Algoritmo eficiente en la generación de una tabla de primalidad de números usando Programación Funcional. En: Ventana Informática No. 31 (jul-dic). Manizales (Colombia): Facultad de Ciencias e Ingeniería, Universidad de Manizales. p. 131-145. ISSN: 0123-9678

1 Artículo de investigación científica y tecnológica proveniente del proyecto *Desarrollo de contenidos y metodología para un curso de Introducción a la Programación basado en el paradigma de Programación Funcional para estudiantes de primeros semestres de Ingenierías utilizando actividades y técnicas de Active Learning* tramitado y aprobado ante la Vicerrectoría de Investigaciones, Innovación y Extensión de la Universidad Tecnológica de Pereira..

2 PhD en Ciencias de la Educación. Docente de Planta – Programa Ingeniería de Sistemas y Computación, Universidad Tecnológica de Pereira (Pereira, Colombia). Correo electrónico: omartrejos@utp.edu.co

Palabras Clave: *Algoritmo, eficiencia, matemáticas, números primos, programación funcional*

Abstract

In this article, we use Functional Programming to generate an analytic table of prime numbers in a specific range using an efficient algorithm. You can find the foundation and we use its logical structure to solve the problem. The proposal of this article is to show the useful face of the applied efficient algorithmic knowing the modern technologies y the math problems. We use Scheme as a programming language and we use its potentialities to manage, to storage and calculate data. We demonstrate that, using efficient algorithms and a simple logic, the computational technologies are very useful solving math problems.

Keywords: *Algorithm, efficiency, maths, prime numbers, Functional programming*

Introducción

La computación se ha convertido en el espacio excelso para encontrar soluciones a problemas que tradicionalmente han tenido que resolverse por otros métodos. Principios como la complejidad y la eficiencia, son fundamentos teóricos sobre los cuales se construyen las soluciones computacionales modernas y, con ello, estos principios han permitido que la programación de computadores evolucione en su concepción y aplicación.

Debe tenerse claro que la complejidad lógica se entiende como la necesidad de encontrar en los recursos más excelsos de las matemáticas, los caminos que permiten el logro de unos objetivos, independiente de los conceptos que subyacen a la solución. La eficiencia por su parte se entiende como el apropiado uso de los recursos que provee la tecnología y, especialmente, la computación de forma que las soluciones sean de bajo costo, en el sentido más amplio.

El problema que se plantea consiste en establecer, en un rango definido, cuáles números cumplen con la característica de *primalidad*[3] y cuáles no, señalándolos y contabilizándolos. Aprovechando los conceptos

3 «*Se conoce como primalidad la propiedad que tiene un número de ser, precisamente, un número primo. Debido a que el único número primo par es el número 2 (dado que cumple con la definición) entonces, con frecuencia, se utiliza el término número primo impar para referirse a cualquier número primo mayor que 2*» (Burns citado por Trejos, 2011, 277).

de eficiencia y simplicidad, se procede en este artículo a plantear una solución al problema formulado capitalizando las bondades de la programación funcional y las facilidades de codificación y entronque matemático del lenguaje de programación *Scheme*. En el desarrollo del algoritmo se explicará porqué este puede considerarse de alta eficiencia y simplicidad.

En la actualidad, tendencias como el análisis de complejidad, la eficiencia e, incluso, la simplicidad de un algoritmo, corresponden a tópicos que no han de descuidarse dada la gran incidencia que tiene la teoría de *computabilidad* y sus aplicaciones en el mundo moderno. Se ha acudido a una literatura que fundamenta la historia de las matemáticas, la historia de la computación, la teoría de los números, la teoría de la complejidad (específicamente la teoría de la *computabilidad*) y la programación funcional como base conceptual.

Desde lo académico, puede considerarse el presente artículo como un aporte para explicar tanto el concepto de eficiencia, en los términos que más adelante se aclaran, como para encontrar un punto de encuentro entre la matemática, desde la óptica de sus problemas tradicionales, y la computación, con tecnologías prácticas y conceptuales aplicadas. Para los estudiantes de Ingeniería de Sistemas será de gran utilidad tener un ejemplo claro de aplicación de la programación, al alcance de sus conocimientos, partiendo de los dos principios rectores de la programación moderna. Se ha acudido al lenguaje de programación *Scheme Entorno Dr Racket* debido a la sencillez con que se pueden implementar soluciones a problemas matemáticos, la facilidad de monitorear pruebas a las funciones, la prioridad que este lenguaje (y su paradigma asociado) le conceden a las funciones y la simplicidad para comprender las soluciones.

El objetivo general consiste en demostrar que se pueden solucionar problemas matemáticos con programación funcional y algoritmos de alta eficiencia así como complejidad moderada, y confirmar el sentido que tienen las tecnologías modernas en el desarrollo de soluciones a los problemas de la sociedad.

En términos generales, el programa aprovecha un algoritmo óptimo para detección de números primos y, en la medida en que va recorriendo un rango dado, va ubicando un señalador que indica la *primalidad* o no de los números comprendidos en dicho rango. Al final, se totalizan los resultados indicando cuántos números primos hay en el rango y cuántos no son primos, presentando una tabla resumen.

Como hipótesis se plantea que, aún en un momento infante de la formación de ingenieros de sistemas, es posible implementar soluciones

claramente eficientes y simples que resuelvan problemas de la matemática, al alcance de los estudiantes en sus primeras fases de formación.

Este artículo comienza presentando una teoría en relación con los temas que se involucran, posteriormente describe de manera detallada el programa y las funciones que se diseñaron para resolverlo. Se presentan los resultados obtenidos y, con base en ellos, se hace una discusión analítica para terminar con conclusiones producto de las mismas.

1. Fundamento teórico

La *primalidad* de un número, tema del cual se ocupa la Teoría de Números en las Matemáticas, es la característica que tiene un número natural de tener solamente dos divisores exactos que son el número 1 y el mismo número, según Giordano (2009, 37). Una tabla de *primalidad* de números primos consiste en la relación de todos los números naturales comprendidos en un rango de manera que en cada uno se identifique si cumple o no con dicha característica.

Darse a la tarea de encontrar todos los divisores exactos de un número representa recorrer el camino para confirmar que dicho número posee o no la característica de ser primo, dado que si la cantidad de divisores es mayor que dos significa que el número no es primo y si dicha cantidad es dos indica que es un número primo, de lo cual se puede suponer que los dos divisores son el 1 y el mismo número.

La definición de la *primalidad*[4] de un número, de acuerdo con Delvin (2002, 26), ha sido preocupación de los matemáticos de todas las épocas y para resolverlo se ha acudido a diferentes formas basada en técnicas y estrategias que provee la matemática. Desde hace más de 20.000 años, con los huesos de Ishango, el ser humano concibió un conjunto de números con una característica diferente a los demás. Los egipcios, los babilonios y los griegos, pueblos en donde floreció la matemática, también se ocuparon de perfilar la *primalidad* como una característica especial de los números y abrieron el camino para lo que hoy se conoce como Teoría de Números.

Las fracciones egipcias, señalan Rey & Pastor (2008, 52), que se identificaban porque el numerador siempre era el número 1 y los denominadores números primos, se usaban para expresar cualquier cantidad en forma de fraccionario, bajo un método similar al de descomposición por factores primos que se utiliza en la actualidad, lo cual indica que

4 Debe anotarse que, por definición, el número 1 no es un número primo y que el infinito conjunto de los números primos se denota con *P*.

los egipcios tenían una noción bastante sólida de la *primalidad* como característica de algunos números naturales.

Euclides, en 300 A.C., ideó una forma confiable de probar la existencia de los números primos y fue capaz de demostrar que el conjunto de ellos es infinito, como lo consignó en su libro Elementos Tomo VII. Estas bases, asegura Krantz (2010, 86), fueron retomadas por varios matemáticos como el monje Mersenne.

Eratóstenes diseñó su criba, como otra forma algorítmica de resolver el problema a partir de la interacción del concepto de números primos, con la idea de una matriz de números[5]. Si bien este método es bastante dispendioso, no se puede descartar pues, con las tecnologías modernas, puede ser aprovechado como algoritmo ineficiente pero efectivo en la solución del hallazgo de los números primos.

Fermat, Euler, Mersenne y Riemann, entre otros matemáticos, han intentado caracterizar los números primos haciendo a un lado las sumas y restas para acudir a la formulación matemática de alto nivel con el ánimo de hacer más eficiente dicha caracterización, de forma que el trabajo de papel y lápiz de otros tiempos sea más llano, más confiable y, por ende, menos agotador.

En este punto es, según Grant, Palmer & Smith (2011, 22), donde la tecnología computacional moderna permite encontrar números primos con muchos dígitos, como los que usa la empresa estadounidense RSA para el desarrollo de sus claves públicas en transmisión confiable de información por los canales de Internet. La gran ventaja que proporcionan las tecnologías modernas, y sus dispositivos asociados, es poner a disposición estructuras electrónicas de alta velocidad, procesamiento eficiente y gran aprovechamiento de recursos.

Cormen & Leiserson (2009, 55), consideran que la computación ha refinado, para efectos del diseño de algoritmos eficientes, dos conceptos: - la complejidad computacional, se ocupa del análisis de los recursos computacionales involucrados en la implementación de una solución a un problema computable, y - la complejidad algorítmica, aborda el análisis detallado de los algoritmos, de los recursos que requiere y de la manera de optimizarlos, señala Mackay (2006, 163), para que su codificación se haga con eficiencia.

El mismo concepto de algoritmo, a pesar de que aparece hacia el año 800 con Al-Juarismi, se potencializa, mejora, optimiza y refina con la intervención de la tecnología computacional y su utilización en la implementación de soluciones. Algunos matemáticos como Napier, Kepler,

5 Un conjunto de números organizados en forma de filas y columnas

Pascal y Babbage, se ocuparon de diseñar y construir dispositivos que pudieran resolver problemas matemáticos de una manera más ágil y confiable.

Los sistemas electrónicos mejoraron su rendimiento y, hacia la mitad del siglo XX, el mundo cambió con la aparición de las altas velocidades y los dispositivos de alto rendimiento, donde el computador como protagonista, potencia la obtención de soluciones donde eficiencia y complejidad confluyen. Para Brassard (2009, 30), en la actualidad el desarrollo de la tecnología ha llevado a dispositivos de alta integración, gran almacenamiento y alto procesamiento, las tres características que identifican el mundo de los avances modernos.

Las necesidades de la sociedad se suplen desde las nuevas tecnologías con sus dos principales características: veracidad y oportunidad, que según expresan Gerequeta & Vallecito (2007, 111), sin que tenga que ser una regla, se puede asociar mucho más la veracidad con el software y la oportunidad con el hardware, como base clave para soluciones a problemas computacionales.

De acuerdo con Sedgewick & Wayne (2011, 37), a partir de la maduración de las tecnologías computacionales, aparece la complejidad, característica de lo complejo, que en sí mismo encierra lo complicado, y que puede definirse como aquello que requiere un estudio detenido y profundo para ser entendido en su funcionamiento como en su composición.

La Teoría de la Complejidad se ocupa de analizar el todo y sus partes y cuando este todo tiene que ver con solución a problemas computacionales se habla de Teoría de la Complejidad Computacional, considerada por Trejos (2010a, 155), como un área de la Teoría de la Computación que se ocupa de analizar, comprender y clasificar los problemas computacionales en su dificultad propia (nivel de dificultad relacionado con su naturaleza innata) y las diferentes relaciones que se presentan. La complejidad algorítmica corresponde al análisis de la cantidad de recursos que se requieren temporalmente para hacer efectiva la solución de un problema computable.

El concepto de eficiencia, incorporado en el título de este artículo, según Trejos (2010b, 155), hace referencia a la capacidad de disponer de los recursos computacionales necesarios para alcanzar un objetivo determinado de manera que los resultados sean óptimos con menor consumo de dichos recursos. Lo deseable, señala Trejos (1999, 39), es que todo algoritmo sea altamente eficiente en términos computacionales, la cual involucra la complejidad computacional como una de sus bases. Entre la eficiencia y la complejidad, se ha desarrollado una

solución que permite generar una tabla de análisis de *primalidad* de números en un rango dado apoyados en la programación funcional, que se convierte en una vertiente de la programación declarativa y, en ella, el núcleo de la programación es la función, incluyendo su definición, desde lo matemático y las relaciones que se establecen.

Se escogió la programación funcional y el lenguaje de programación *Scheme* debido a su alta cercanía con la notación matemática y a lo proclive que es este lenguaje para facilitar la solución de problemas de orden matemático. Se ha acudido al gran potencial que provee la programación funcional desde el concepto de función y bajo la estrategia de la recursividad que permite plantear soluciones de gran rendimiento computable. En este programa se utilizó recursividad nivel III, es decir, funciones recursivas que se apoyan en otras del mismo tipo para lograr su objetivo, en términos de Trejos (2011b, 276).

2. Metodología

2.1 Descripción

El programa diseñado consiste en la aplicación de un algoritmo eficiente para la generación de una tabla de *primalidad* de números en un rango dado usando programación funcional:

```
;; =====================================================
;; DESPLIEGUE DE UNA TABLA DE PRIMALIDAD DE NÚMEROS
NATURALES
;; EN UN RANGO ;; DADO A PARTIR DE UN ALGORITMO DE EVALUACIÓN
;; OPTIMIZADO
;; Recibe un rango y evalúa cuales números naturales den-
tro de ese
;; rango son primos y cuáles no, de forma que construye
una tabla
;; que despliega con la información pertinente totalizando
los resultados
;; =====================================================

;; Función que determina si un num es múltiplo de otro
(define (divisor a b)   ;; Definición de la función
(if (= (remainder a b) 0)      ;; Si el 2o num divide exac-
tamente al 1o
 1 ;; entonces retorne un valor 1 (true)
 0 ;; Sino retorne un valor 0 (False)
)) ;; Fin Condicional - Fin Función
```

```
;; Función que cuenta los div exactos de un num
;; en el rango (2,raiz cuad num)
(define (cuentadivisores num div)      ;; Definición de la
función
 (if(= div (floor(/ (sqrt num)1)))     ;; Si se llegó a la
raíz cuad
 ;; entonces reciba lo que devuelva la función divisor
enviándole
 ;; como argumento el num y la raiz cuadrada del num
 (divisor num (floor ( / (sqrt num) 1)))

 ;; Sino, entonces sume lo que retorne la función divisor
mas
 ;; lo que retorne la misma función cuentadivisores
 ;; incrementando en 1 el valor de divisor
 ( + (divisor num div)
 (cuentadivisores num (+ div 1)))
 ))                                    ;; Fin Condicional
- Fin Función


;; Función que determina si un num es primo (optimizada)
(define (esprimo n) ;; Definición de la función
(if (= (cuentadivisores n 2) 0) ;; Si el num tiene 0
divisores
 1 ;; retorne 1 (Verdadero)
 0 ;; retorne 0 (Falso)
)) ;; Fin Condicional - Fin Función

;; Función que recibe el rango de evaluación
(define (rangonum ini tope)     ;; Definición de función
 (if (> ini tope)               ;; Si se llegó al tope
 (begin        ;; Mostrar línea doble que indique final
 (newline)
 (display "=============================="))
 (begin                                ;; sino
 (newline)
 (display " ")
 (display ini)                  ;; mostrar el valor evaluado
 ;; Dependiendo de si es primo o no, ubicar una x en
 ;; la columna correspondiente
 (if (= (esprimo ini) 1)
 (begin
 (display " X")
 (rangonum (+ ini 1) tope))
```

```
 (begin
 (display " X")
 (rangonum (+ ini 1) tope))
 ) ) ) )                        ;; Fin Función


;; Función que cuenta la cantidad de números primos
(define (cuentaprimos ini tope)       ;;  Definición  de
función
 (if (= ini tope)      ;; si se ha llegado al tope
 (esprimo ini) ;; entonces evaluar el tope
 ;; sino sumar lo que retorne la función esprimo
 ;; mas lo que retorne la misma función cuentaprimos
 ;; incrementando el valor inicial
 (+ (esprimo ini)
 (cuentaprimos (+ ini 1) tope))
 ))                    ;; Fin Condicional - Fin Función


;; Función que cuenta la cantidad de números no primos
(define (cuentanoprimos ini tope)     ;;  Definición  de
función
 (+ (- (- tope ini)    ;; calcular la cantida de
 (cuentaprimos ini tope))      ;; nums no primos que hay
 1)     ;; en el rango
 ) ;; Fin Condicional - Fin Función

;; Función que inicia el proceso
(define (interfaz ini tope)     ;; Definición de función
 (newline)      ;; Línea en blanco
 (display "Número Primo NoPrimo") ;; Poner títulos
 (rangonum ini tope)   ;; Desplegar tabla de primalidad
 (newline)      ;; Línea en blanco
 (display "Totales ")  ;; Titulo de Totales
 (display (cuentaprimos ini tope))    ;; Mostrar cant de
primos
 (display " ") ;; Dejar espacios en blanco
 (display (cuentanoprimos ini tope)) ;; Mostrar cant de
no primos
 )      ;; Fin Función
```

2.2 Aplicación

El esquema funcional de la solución algorítmica representa la estructura lógica que permite observar las interacciones entre las funciones y el flujo mismo del programa (Figura 1), donde la función interfaz llama a tres funciones: *rangonuma*, *cuentaprimos* y *cuentanoprimos* a las cuales les envía los argumentos *ini* y *tope*. *rangonum* es una función recursiva y llama a la función *esprimo* (a la cual le envía el valor de *ini*), que a su vez llama a la función *cuentadivisores* (que recibe dos argumentos *num* y *div*) y esta, a la función *divisor* (que recibe dos argumentos a y b) que retorna un valor 1 o 0 dependiendo de una condición.

La función *cuentaprimos* llama a la función *esprimo* que realiza la misma secuencia anterior. Esta función llama a la función *cuentadivisores* (a la cual le envía los argumentos *num* y *div*) y que a su vez llama a la función *divisor* (que recibe los argumentos *a* y *b*), la cual devuelve un valor 1 o 0 dependiendo de una condición. La función *cuentanoprimos* llama a la función *cuentaprimos* y esta le retorna un resultado producto de una resta.

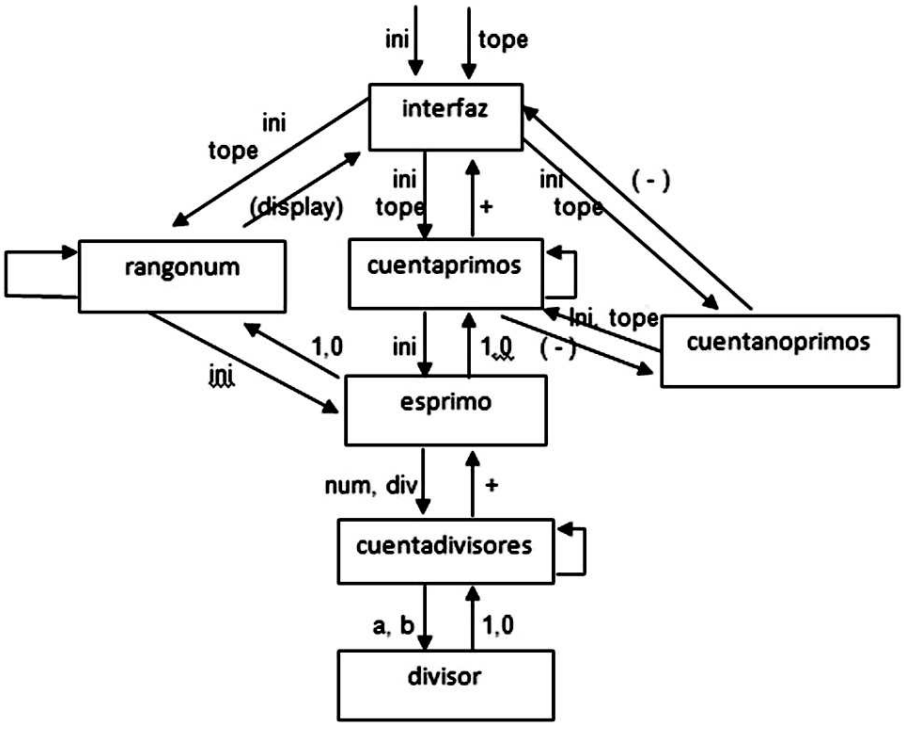

Figura 1. Esquema Funcional

3. Resultados

3.1 Descripción de resultados

En su parte aplicativa, al momento de ejecutar al programa se obtienen los siguientes resultados:

```
> (interfaz 1000 1010)
Número Primo NoPrimo
 1000 X
 1001 X
 1002 X
 1003 X
 1004 X
 1005 X
 1006 X
 1007 X
 1008 X
 1009 X
 1010 X
==============================
Totales 1 10
```

En este primer caso se evalúan los números primos comprendidos entre 1000 y 1010. El programa genera la tabla correspondiente encontrando 1 número primo (1009) y 10 números no primos dentro del rango de los 11 números evaluados.

```
> (interfaz 1000005 1000015)
Número Primo NoPrimo
 1000005 X
 1000006 X
 1000007 X
 1000008 X
 1000009 X
 1000010 X
 1000011 X
 1000012 X
 1000013 X
 1000014 X
 1000015 X
==============================
Totales 0 11
```

En este segundo ejemplo, se establece como rango de evaluación los números 40 a 60, para el cual se genera la tabla con cinco primos y 16

no primos para completar los 21 números naturales del rango evaluado. Es de anotar que el rango de evaluación puede ser tan amplio como se desee. Para efectos del espacio disponible en este artículo se han seleccionado solamente tres ejemplos con rangos breves. Un llamado totalmente posible de este programa podría ser:

```
>(interfaz 500000 500100)
```

Y los totales que originaría, luego de la tabla de *primalidad*, son los siguientes:

```
==============================
Totales 6 95
```

Lo cual indica que en el rango (500000, 500100) existen seis números primos y 95 no primos[6].

3.2 Discusión

Lo primero que puede destacarse de esta solución es el código muy sencillo y fácil de entender, lo cual posibilita que un código, con estas características, pueda intervenirse, ajustarse, optimizarse y darle mantenimiento de una manera simple y ágil. La codificación simple se nota en funciones como la siguiente:

```
;; Función que determina si un num es múltiplo de otro
(define (divisor a b) ;; Definición de la función
(if ( = (remainder a b) 0);; Si el 2o num divide exacta-
mente al 1o
 1      ;; entonces retorne un valor 1 (true)
 0                    ;; Sino retorne un valor 0 (false)
))      ;; Fin Condicional - Fin Función
```

En este tipo de funciones, es muy sencillo comprender su lógica y establecer que si el residuo de dividir el valor del argumento *a* entre el valor del argumento b es *Verdadero*, entonces se retornará 1 y si es *Falso*, se retornará el valor 0. La simplicidad de esta función hace que su utilización y aplicación sea muy sencilla a partir de la lógica expuesta.

Se aprovecha en buena medida la potencialidad de la recursividad pues, a la luz tanto del esquema funcional como del código, el programa solución se apoya en tres funciones recursivas (*rangonum*, *cuentapri-*

6 Por razones de espacio se ha suprimido la relación puntual de los números comprendidos en el rango con su respectivo análisis y ubicación de *primalidad*.

mos y *cuentadivisores*) que, si bien implican un consumo de recursos computacionales, son solución sencilla y simple de comprender, dentro del contexto del programa presentado y de las posibilidades de las tecnologías modernas. Tal como se explicó, en este programa se acude a la recursividad nivel III, es decir, funciones recursivas que se apoyan en otras funciones recursivas.

La relación entre las funciones y el desempeño de cada una es óptima en un nivel alto (más no absoluto), pues de una parte es claro que el enlace de ida de las funciones se capitaliza a partir de los argumentos y el enlace de venida se aprovecha desde los valores de retorno. Se ha de notar que, tal como se aprecia en el esquema funcional, los argumentos enviados corresponden a los argumentos originales que inician el programa o a algún componente aritmético de ellos. De otra parte los valores de retorno son el resultado de operaciones simples (sumas o restas) o valores 1 o 0, todavía más simple.

Debe anotarse que para los números 2 y 3, el programa requerirá ajustes muy especiales, puesto que ellos no estarían contemplados dentro de la función *cuentadivisores*, ya que esta depende de que se establezca un rango entero no vacío ni igual a cero entre 2 y la raíz cuadrada del número, asumiendo eso sí, que este segundo valor es superior o igual al número 2. Precisamente, para los valores citados (2 o 3) no se cumple esta condición y, por tanto, el rango quedaría con un valor inicial inferior al tope del mismo, lo cual haría inconsistente la función. Un posible ajuste de la función *cuentadivisores* que, omitiendo comentarios, aparece así:

```
(define (cuentadivisores num div)
 (if (= div (floor ( / (sqrt num) 1)))
 (divisor num (floor ( / (sqrt num) 1)))
 ( + (divisor num div) (cuentadivisores num (+ div 1)))
 ))
```

Podría ser el siguiente:

```
(define (cuentadivisores num div)
 (if (= div (floor ( / (sqrt num) 1)))
 (divisor num (floor ( / (sqrt num) 1)))
 ( begin                        ;; Inicio de la modificación
 (if (or (= num 2) (= num 3) ) ;; Si es el número 2 o 3
 0                              ;; Entonces retornar 0
 (+ (divisor num div)  ;; Sino realizar operación
 (cuentadivisores num (+ div 1)))
 )                              ;; Fin de la modificación
 ))                             ;; Fin de la Función
```

Puede notarse que en la línea en la cual se retorna un valor 0, esto confirma que el número 2 o el número 3 son primos, de acuerdo a la evaluación en la función *esprimo*:

```
;; Función que determina si un num es primo (optimizada)
(define (esprimo n)             ;; Definición de la función
(if (= (cuentadivisores n 2) 0)       ;; Si el num tiene
0 divisores
 1      ;; retorne 1 (Verdadero)
 0      ;; retorne 0 (Falso)
))      ;; Fin Condicional - Fin Función
```

La función *cuentadivisores* puede optimizarse aún más, puesto que, tal como está, la evaluación de cada número *n* se hace en el rango (2, raíz cuadrada de n). Haciendo una mirada detenida del algoritmo puede pensarse que si el número fuera 1000000 (un millón) no tendría que hacerse todo el recorrido de evaluación desde 2 hasta 1000 (que es su raíz cuadrada) dado que, con el solo hecho de encontrar que el número 2 es un divisor exacto del número, es suficiente para detectar que el número 1000000 no es un número primo. Esto aceleraría notoriamente la evaluación de los números sin desconocer que, como está planteada, es una buena solución.

Finalmente, vale la pena acotar que el concepto de eficiencia radica en tres factores dentro de este programa: - el diseño de las funciones en el cual es acude a la simplicidad, - la relación entre funciones, paso de argumentos y retornos enteros y simples de comprender y manipular, y - los rangos de evaluación y el bueno uso de la recursividad cuando se hace necesario.

4. Conclusiones

Se pueden desarrollar algoritmos que provean soluciones a diferentes problemas bajo un enfoque de alta eficiencia y suficiente complejidad de forma que se capitalicen las características de diseño y rendimiento del hardware y el software moderno.

Vale la pena analizar el concepto de optimización en el diseño de algoritmos al momento de diseñar un programa que resuelva un problema, especialmente cuando este sea de orden matemático.

Conviene analizar hasta donde, en el diseño de un algoritmo, la complejidad de la solución es necesaria pues una solución como la que aquí se expone se distingue por su alta simplicidad y, si bien es cierto que al respecto de este problema existen diversas soluciones de diferentes niveles de complejidad, las soluciones simples son las más fáciles de

entender y, apoyados en el hardware moderno, podrían ser suficientes para resolver óptimamente muchos problemas.

Resulta ser de gran utilidad ahondar en la teoría de la *computabilidad* para determinar criterios claros y concretos que permitan establecer las soluciones en las cuales se justifica acudir a la complejidad y la eficiencia y en cuáles no.

Siempre van a existir dos caminos para resolver un problema computacional: uno que puede ser el complejo y eficiente, otro que puede ser simple e ineficiente. Es posible que algunas soluciones lleguen a ser complejas e ineficientes y otras lleguen a ser simples y eficientes y es ese precisamente el tema que atañe a la teoría de la *computabilidad*.

Referencias bibliográficas

BRASSARD, G. & BRADLEY, P. (2009). Fundamentos de Algoritmia. Montreal (Canadá): Prentice Hall. 579 p. ISBN: 848 966 000X

CORMEN, T.H.; LEISERSON, C.E.; RIVEST, R.L. & STEIN, C. (2009). Introduction to Algorithms. 3 ed. Massachusetts (USA): MIT Press. 1313 p. ISBN: 978-0262033848

DEVLIN, K. (2002). El lenguaje de las matemáticas. Barcelona (España): Ediciones Robinbook. 381 p. ISBN: 84-95601-71-0

GEREQUETA, R. & VALLECILLO, A. (2007). Técnicas de Diseño de Algoritmos. Málaga (España): Servicio de Publicaciones de la Universidad de Málaga. 315 p. ISBN: 9788474967845.

GIORDANO, P. (2009). La soledad de los números primos. Barcelona (España): Narrativa Salamandra. 288 p. ISBN: 9788498382051

GRANT, M.; PALMER, Z & SMITH, S. (2011). Principles of Programming Languages: Version 0.7 [on line]. San Francisco (CA, USA): Creative Commons Attribution-Share Alike 3.0. 181 p. <http://www.cs.jhu.edu/~scott/pl/book/dist/book/book.pdf> [consult: 27/03/2014]

KRANTZ, S.G. (2010). An Episodic History of Mathematics. Mathematical culture through problem solving. Washington D.C. (USA): Mathematical Association of America. 381 p. ISBN: 978-0883857663

MACKAY, D.C. (2006). Information Theory, Inference and Learning Algorithms. 4 ed. Cambridge (UK): Cambridge University Press. 640 p. ISBN: 978-0521642989

REY PASTOR, J. & BABINI, J. (2008). Historia de la Matemática. Barcelona (España): Gedisa Editorial. 224 p. ISBN: 9788497847810

SEDGEWICK, R. & WAYNE, K. (2011). Algorithms. Upper Saddle River (NJ, USA): Pearson Education, Inc. Princeton, USA. 937 p. ISBN: 978-0-321-57351-3

TREJOS B., O. I. (1999). La esencia de la lógica de programación. Manizales (Colombia): Centro Editorial Universidad de Caldas. 325 p. ISBN: 958-33-1125-1

TREJOS B., O. I. (2010a). Algoritmo de optimización para la detección de un número primo basado en programación funcional utilizando DrScheme. En: Revista Scientia et Technica, Vol. 17, No. 47. Pereira (Colombia). p. 276-289. ISSN: 0122-1701

TREJOS B., O. I. (2010b). Determinación simple de un número primo aplicando programación funcional a través de DrScheme. En: Revista Scientia et Technica. Vol. 16, No. 45. Pereira (Colombia). p. 155-160. ISSN: 0122-1701

TREJOS B., O. I. (2011a). Fundamentos de Programación. Pereira (Colombia): Editorial Papiro. 152 p. ISBN: 978-958-8236-10-0

TREJOS B., O.I. (2011b). Algoritmo de optimización para la detección de un número primo basado en programación funcional utilizando DrScheme [en línea]. En: Scientia et Technica, Vol. 17, No. 47 (abr). Pereira (Colombia): Universidad Tecnológica de Pereira. p. 276-280. ISSN: 0122-1701. <http://revistas.utp.edu.co/index.php/revistaciencia/article/view/1161/653> [consulta: 09/02/2014]