Escribir Un Algoritmo Para Encontrar Un Elemento En Una Secuencia*

Juan Pablo Gonzalez Rincon[1], Matias Felipe Gonzalez Valencia.[1], Laura Isabel Montero Blanco[1]

[a] *Pontificia Universidad Javeriana, Bogota, Colombia*

Abstract

Este documento presenta un algoritmo basado en "dividir y vencer"para calcular la representación booleana de un número natural. El algoritmo descompone recursivamente el número en sus componentes binarios y combina los resultados para obtener la secuencia final de bits. Se implementa en Python y se analiza su complejidad temporal y espacial.

Keywords: Dividir y vencer, representación booleana, algoritmos recursivos, complejidad algorítmica.

1. Análisis del Problema

Dado un número natural $n \in \mathbb{N}$, se calculará su representación booleana B_n.

Formalmente, la representación booleana de un número n se define como la secuencia de bits que representa a n en el sistema binario, es decir:

$$B_n = \{b_i \mid b_i \in \{0,1\},\, 0 \leq i < k,\, k = \lceil \log_2(n+1) \rceil\}$$

donde:

- n es un número natural.
- B_n es la representación booleana de n, la cual es un vector de bits.
- k es la longitud de la secuencia booleana necesaria para representar n en binario.
- b_i representa los bits individuales de la secuencia booleana, tales que cada b_i es igual a 0 o 1.

*Este documento presenta la escritura formal de un algoritmo.

Email addresses: gonzalez_juanp@javeriana.edu.co (Juan Pablo Gonzalez Rincon), matias_gonzalez@javeriana.edu.co (Matias Felipe Gonzalez Valencia.), Limontero@javeriana.edu.co (Laura Isabel Montero Blanco)

El objetivo es construir la secuencia B_n a partir de n, utilizando operaciones de división y residuo para obtener los bits b_i de manera recursiva, de modo que la representación B_n cumpla con las propiedades del sistema binario para representar a n de forma exacta.

2. Algoritmos de solución

2.1. Algoritmo Recursivo

El algoritmo recursivo descompone un número X en rangos de bits, calculando su representación binaria dentro de un intervalo definido por b y e. Se basa en la técnica de "dividir y vencer", donde X se descompone mediante llamadas recursivas para calcular los bits individuales en el rango especificado. Finalmente, combina los bits obtenidos para formar la secuencia binaria completa.

Algorithm 1 Conversión Binaria Auxiliar con Rango

procedure CB(X, b, e)
    if $e < b$ **then**
        return []
    else
        $q \leftarrow (b + e)//2$
        $d \leftarrow (X//(2^q))\,\%2$
        $l \leftarrow \mathrm{CB}(X, b, q - 1)$
        $h \leftarrow \mathrm{CB}(X, q + 1, e)$
        return $h + [d] + l$
    end if
end procedure

2.1.1. Invariante

Las sentencias de retorno de la función siempre devolverán la secuencia binaria del número ingresado.

- Inicio: Caso base: Se retorna la secuencia vacía.
- Avance: Cualquier número con una secuencia no vacía: Se hace la transofrmación a binario y se retorna la secuencia booleana.

2.1.2. Análisis de Complejidad

La complejidad del algoritmo se analiza en términos de la ecuación de recurrencia: $\mathrm{T(n)} = \mathrm{T}\left(\frac{n}{2}\right) + O(n)$

donde T(n) es el tiempo de ejecución para un número con n posiciones en su traducción booleana y la partición del algoritmo tiene una completidad de O(n) en el peor de los casos.

2.2. Algoritmo Iterativo

El algoritmo iterativo convierte el número natural n en su representación binaria utilizando un bucle. Calcula los bits uno por uno, almacenándolos en una lista, hasta que el número se reduce a 0.

Algorithm 2 Conversión Iterativa a Binario

```
procedure CBI(n)
    res ← []
    while n > 0 do
        r ← n%2
        res ← [r] + res
        n ← n//2
    end while
    if res = [] then
        return [0]
    end if
    return res
end procedure
```

2.2.1. Invariante

Las sentencias de retorno de la función siempre devolverán la secuencia binaria del número ingresado.

- **Inicio**: Caso base: Si n es 0, la lista de bits `res` estará vacía, y se retornará `[0]`.
- **Avance**: Para cualquier $n > 0$, se calcula el residuo r de n dividido por 2, se almacena en la lista `res` y luego n se divide por 2. El proceso continúa hasta que n se reduce a 0, construyendo la representación binaria en la lista `res`.

2.2.2. Análisis de Complejidad

La complejidad del algoritmo se analiza en términos del número de iteraciones del ciclo `while`. En cada iteración, n se reduce a la mitad, lo que da como resultado una complejidad de $O(\log n)$ para el número de pasos necesarios.

- **Complejidad Temporal**: El tiempo de ejecución del algoritmo es $O(\log n)$ debido a que el número de iteraciones es proporcional al logaritmo en base 2 de n.