Escritura del problema del número binario inverso

María Camila Aguirre Collante[1] Jessica Tatiana Naizaque Guevara[1]

[1]Departamento de Ingeniería de Sistemas, Pontificia Universidad Javeriana
Bogotá, Colombia
{aguirrec.mcamila, j.naizaque}@javeriana.edu.co

16 de agosto de 2022

Resumen

En este documento se presenta la formalización del problema del número binario inverso, junto con la descripción de dos algoritmos que lo solucionan. Además, se presenta el orden de complejidad y la invariante para cada uno de los algoritmos. **Palabras clave:** número decimal, número binario, invertir, algoritmo, formalización, complejidad.

Índice

1. Introducción

Los números binarios representan un número de forma alternativa usando solo dos valores: $0, 1$. Utilizando variedades de combinaciones de estos dos símbolos se pueden generar números decimales teniendo en cuenta las potencias de dos. Por lo que, si se toma este número binario de forma inversa, se genera un número decimal distinto. En este documento es posible encontrar dos algoritmos que posibiliten encontrar la inversa de un número binario y, así, poder conocer cuál sería el número decimal "escondido" detrás de estos símbolos invertidos. A lo largo del presente informe se mostrará: la formalización del problema (sección 2) y la escritura formal de dos algoritmos (sección 3).

2. Formalización del problema

Dado un número $a \in \mathbb{N}$, encontrar su representación binaria que es posible definirla teniendo en cuenta:

- $a = 2^0 * b_1 + 2^1 * b_2 + 2^2 * b_3 + ... + 2^{n-1} * b_n$,
- $b = \langle b_i \in (0,1) \rangle \mid 1 \leq i \leq n$

donde a es el número decimal que se forma teniendo en cuenta el valor de los símbolos $(0,1)$ en cada una de las posiciones $2^0...2^n$, n es la cantidad de potencias de 2 que se requieren para representar al número a en un binario y b es la secuencia como número binario que representa al número decimal a. Considerando lo anterior, encontrar la representación binaria inversa del número a y determinar su valor decimal, que se podría representar como $d = 2^{n-1} * b_1 + 2^{n-2} * b_2 + ... + 2^0 * b_n$.

2.1. Definición del problema del "número binario inverso"

Así, el problema del número binario inverso se define a partir de:

1. un número decimal $a \in \mathbb{N}$ que
2. puede ser representado de forma binaria así $b = 2^0 * b_1 + 2^1 * b_2 + 2^2 * b_3 + ... + 2^{n-1} * b_n$

generar el número binario inverso a b y, así mismo, encontrar su respectivo número decimal.

- Entradas:
  - $a \in \mathbb{N} \mid a = 2^0 * b_1 + 2^1 * b_2 + 2^2 * b_3 + ... + 2^{n-1} * b_n$.
- Salidas:
  - $revBin \mid 2^{n-1} * b_1 + 2^{n-2} * b_2 + ... + 2^0 * b_n$
  - $d \in \mathbb{N} \mid d = 2^{n-1} * b_1 + 2^{n-2} * b_2 + ... + 2^0 * b_n$.

3. Algoritmos de solución

3.1. Iterativo

La idea de este algoritmo es: en una primera instancia, convertir el número decimal que se recibe a un número binario. De esta forma, se comienza con un ciclo en el que se ejecutan las operaciones necesarias para ir generando el número binario inverso al obtenido anteriormente. Esta serie de pasos permite lograr el objetivo de revertir el binario puesto que se realiza haciendo uso de las operaciones módulo y división. Luego de encontrar este inverso, se convierte a decimal nuevamente, por lo que, finalmente, retorna la tupla que se conforma por el número binario invertido y su respectivo valor en número decimal.

3.1.1. Análisis de complejidad

Por inspección de código: En cada uno de los métodos es posible observar un ciclo while, por lo tanto, es posible indicar que se tiene un órden de complejidad $O(3n)$. Sin embargo, en un órden de complejidad no se tienen en cuenta las constantes, por lo que su complejidad final sería $O(n)$.

3.1.2. Invariante

La variable *binaryInv* indica el número binario invertido teniendo en cuenta cada uno de los dígitos que hacen parte del binario original.

1. Inicio: $binaryInv = 0$, no se ha invertido el binario.
2. Iteración: Se realiza un ciclo en el que va invirtiendo las posiciones de cada uno de los dígitos del número binario original, entonces cada nueva iteración llevará el i-ésimo elemento del número a su posición adecuada.
3. Terminación: $binary = 0$, el binario se recorrió completamente y su inverso ya se ha encontrado.

Algoritmo 1 Iterativo (1/3)

```
1: procedure DecBinary(a)
2:     bin ← 0
3:     prod ← 1
4:     if a = 0 ∨ a = 1 then
5:         return a
6:     else
7:         while a ≠ 0 do
8:             rem ← MOD(a,2)
9:             bin ← bin + (rem * prod)
10:            a ← a ÷ 2
11:            prod ← prod * 10
12:        end while
13:        return bin
14:    end if
15: end procedure
```

Algoritmo 2 Iterativo (2/3)

```
1: procedure BinDecimal(n)
2:     a ← 0
3:     i ← 0
4:     if n = 0 ∨ n = 1 then
5:         return n
6:     else
7:         while n ≠ 0 do
8:             rem ← MOD(n, 10)
9:             n ← n ÷ 10
10:            a ← a + (rem * (2^i))
11:            i ← i + 1
12:        end while
13:        return a
14:    end if
15: end procedure
```

Algoritmo 3 Iterativo (3/3)

```
1: procedure ReverseBinary(a)
2:     binary ← DecBinary(a)
3:     binaryInv ← 0
4:     while binary > 0 do
5:         rem ← MOD(a, 10)
6:         binaryInv ← binaryInv * 10 + rem
7:         a ← a ÷ 10
8:     end while
9:     newNum ← BinDecimal(binaryInv)
10:    return [binaryInv, newNum]
11: end procedure
```

3.2. Basado en la estrategia "dividir-y-vencer"

La idea de este algoritmo es: en primer lugar, hallar la representación binaria del número natural que se recibe. Luego de esto, se realiza la inversión del número binario ejecutando diferentes operaciones necesarias, como el módulo y la división y haciendo uso de la estrategia dividir y vencer y teniendo como pivote el

promedio de los datos. Este último método resultará en el número binario de forma invertida, que debe ser convertido a número decimal de nuevo, para que, finalmente, sea posible obtener el nuevo valor decimal. Considerando lo anterior, el resultado final será una tupla que contenga el nuevo número decimal y su respectiva representación binaria.

Lo anterior, puede ser explicado de mejor forma si se supone que el número binario podría observarse como una secuencia, así:

1. Dado el número $a = 345$, se guarda su representación binaria 101011001 en la secuencia *bin*. Tal como se puede observar en la Tabla 1.

1	0	1	0	1	1	0	0	1

Tabla 1: Representación binaria

2. Se divide *bin* en dos partes teniendo en cuenta la variable q , la cual tiene el papel de "pivote". Además, se establecen las variables b y e, estas representan el inicio y el final de la secuencia, respectivamente. Se puede observar en la Tabla 2, donde q se encuentra en la posición obtenida de la operación matemática $\lfloor (b + e) \div 2 \rfloor$.

1	0	1	0	1	1	0	0	1

Tabla 2: Variables generadas

3. El paso anterior se repite recursivamente obteniendo los pares presentados en la Tabla 4

1	0	1	0	1	1	0	0	1

Tabla 3: Pares generados

4. Para cada una de las parejas se invierten los valores, por lo cual se obtiene:

0	1	0	1	1	1	1	0	0

Tabla 4: Invertir valores *"primer nivel"*

0	1	0	1	1	0	0	1	1

Tabla 5: Invertir valores *"segundo nivel"*

1	0	0	1	1	0	1	0	1

Tabla 6: Invertir valores *"nivel final"*

5. Finalmente, se obtiene el número inverso 309 a partir de la unión final.

Algoritmo 4 Dividir-y-vencer (1/4)

```
1:  procedure DecBinary(a)
2:      bin ← 0
3:      prod ← 1
4:      while a ≠ 0 do
5:          rem ← MOD(a, 2)
6:          bin ← bin + (rem ∗ prod)
7:          a ← a ÷ 2
8:          prod ← prod ∗ 10
9:      end while
10:     return bin
11: end procedure
```

Algoritmo 5 Dividir-y-vencer (2/4)

```
1:  procedure BinDecimal(n)
2:      a ← 0
3:      i ← 0
4:      while n ≠ 0 do
5:          rem ← MOD(n, 10)
6:          n ← n ÷ 10
7:          a ← a + (rem ∗ (2^i))
8:          i ← i + 1
9:      end while
10:     return a
11: end procedure
```

3.2.1. Análisis de complejidad

Para este caso, es preciso utilizar el "Teorema maestro". Inicialmente, por inspección de código se tiene que

$$\mathbf{T(n)} = 2T\left(\frac{n}{2}\right) + O\left(n\right)$$

De acuerdo con esto, el segundo caso del teorema aplicado es el apropiado

$$f(n) \in \Theta\left(n^{\log_b a} \log_2^k n\right) \longrightarrow T(n) \in \Theta\left(n^{\log_b a} \log_2^{k+1} n\right)$$

Teniendo en cuenta lo anterior, y reemplazando los valores correspondientes se afirma que

$$n \longrightarrow \left(n^{\log_2 2} \log_2^0 n\right)$$

En consecuencia, se asevera que el orden de complejidad de este algoritmo es $\Theta\left(n \log_2 n\right)$.

3.2.2. Invariante

El número binario invertido debe tener los mismos caracteres que el número binario original.

1. Inicio: Al inicio, el número binario debe estar organizado como originalmente se obtuvo, por lo que no se le ha realizado ninguna operación.
2. Iteración: Cuando se recorren los elementos del número binario, en el método *ReverseBinary_Aux*, mediante las operaciones de módulo y división, se van obteniendo las parejas encontradas en orden invertido. Por lo que, si encuentra un 10, lo retornará como 01 y así por cada pareja de elementos o por cada elemento.
3. Terminación: Cuando se realiza la unión de ambas partes de la división, se retorna el resultado final.

Algoritmo 6 Dividir-y-vencer (3/4)

```
1: procedure ReverseBinary_Aux(bin, b, e, tam)
2:     devolver ← declarevariable
3:     num ← declarevariable
4:     if (e − b) ≤ 1 then
5:         num ← bin÷ POW(10, tam − e − 1)
6:         if (e − b) = 0 then
7:             devolver ← MOD(num, 10)
8:             return devolver
9:         else
10:            devolver ← MOD(num, 10) ∗10
11:            num ← num ÷ 10
12:            devolver ← devolver+MOD(num, 10)
13:            return devolver
14:        end if
15:    else
16:        q ← ⌊(b + e) ÷ 2⌋
17:        left ← ReverseBinary_Aux(bin, b, q, tam)
18:        right ← ReverseBinary_Aux(bin, q + 1, e, tam)
19:        l ← ToString(left)
20:        r ← ToString(right)
21:        result ← r + l
22:        return ToNumber(result)
23:    end if
24: end procedure
```

Algoritmo 7 Dividir-y-vencer (4/4)

```
1: procedure ReverseBinary(a)
2:     if a = 0 ∨ a = 1 then
3:         return [a, a]
4:     else
5:         bin ← DecBinary(a)
6:         revBin ←ReverseBinary_Aux(bin, 1, |bin|, |bin|)
7:         newNum ← BinDecimal(revBin)
8:         return [revBin, newNum]
9:     end if
10: end procedure
```

4. Análisis experimental

En esta sección se presentarán algunos los experimentos para confirmar los órdenes de complejidad de los tres algoritmos presentados en la sección 3.

4.1. Secuencias aleatorias

Acá se presentan los experimentos cuando los algoritmos se ejecutan con secuencias de entrada de orden aleatorio.

4.1.1. Protocolo

1. Cargar en memoria un archivo de, al menos, 200Kb.

2. Definir un rango $(b, e, s) \in \mathbb{N}^3$, donde: b es un tamaño inicial, e es un tamaño final y s es un salto. Se

generarán secuencias, a partir del archivo de entrada, de diferentes tamaños desde b hasta e, adicionando cada vez s elementos.

3. Cada algoritmo se ejecutará 10 veces con cada secuencia y se guardará el tiempo promedio de ejecución.

4. Se generan los gráficos necesarios para comparar los algoritmos.

4.2. Secuencias ordenadas

Acá se presentan los experimentos cuando los algoritmos se ejecutan con secuencias de entrada ordenadas de acuerdo al orden parcial $a < b$.

4.2.1. Protocolo

1. Definir un rango $(b, e, s) \in \mathbb{N}^3$, donde: b es un tamaño inicial, e es un tamaño final y s es un salto. Se generarán secuencias aleatorias de diferentes tamaños desde b hasta e, adicionando cada vez s elementos.

2. Se usará el algoritmo `sort(S)`, disponible en la librería básica de python, para ordenar dicha secuencia.

3. Cada algoritmo se ejecutará 10 veces con cada secuencia ordenada y se guardará el tiempo promedio de ejecución.

4. Se generan los gráficos necesarios para comparar los algoritmos.

4.3. Secuencias ordenadas invertidas

Acá se presentan los experimentos cuando los algoritmos se ejecutan con secuencias de entrada ordenadas de forma invertida de acuerdo al orden parcial $a < b$.

4.3.1. Protocolo

1. Definir un rango $(b, e, s) \in \mathbb{N}^3$, donde: b es un tamaño inicial, e es un tamaño final y s es un salto. Se generarán secuencias aleatorias de diferentes tamaños desde b hasta e, adicionando cada vez s elementos.

2. Se usará el algoritmo `sort(S)`, disponible en la librería básica de python, para ordenar dicha secuencia.

3. Cada algoritmo se ejecutará 10 veces con cada secuencia ordenada y se guardará el tiempo promedio de ejecución.

4. Se generan los gráficos necesarios para comparar los algoritmos.