Escritura del problema de la representación binaria inversa

David Enrique Palacios García[1] Karen Sofia Coral Godoy[1]

[1]Departamento de Ingeniería de Sistemas, Pontificia Universidad Javeriana
Bogotá, Colombia
{david_palacios, corallg_ksofia}@javeriana.edu.co

18 de agosto de 2022

Resumen

En este documento se presenta la formalización del problema de la inversión de una cadena, junto con la descripción de dos algoritmos que lo solucionan. Además, se presenta un análisis experimental de la complejidad de esos dos algoritmos. **Palabras clave:** inversión, algoritmo, formalización, experimentación, complejidad, cadena.

Índice

1. Introducción

Invertir una cadena puede llegar a ser útil para encontrar similitudes entre datos, tal así, que existe un gran número de formas para hacerlo. El ejemplo práctico que se tratará en este documento será tomar un número natural N, convertirlo a cadena binaria, invertir esta cadena y, finalmente, obtener el valor entero de la cadena invertida. Para resolver este problema se presentan dos algoritmos que lo solucionan, con el objetivo de mostrar: la formalización del problema (sección 2), la escritura formal de ambos algoritmos (sección 3) y un análisis experimental de la complejidad de cada uno de ellos (sección 4).

2. Formalización del problema

Cuando se piensa en *invertir una cadena* la solución inmediata puede ser muy simplista: inocentemente, se piensa en leer una cadena de atrás hacia adelante. Sin embargo, con un poco más de reflexión, hay tres preguntas que pueden surgir:

1. ¿Cadena es únicamente de tipo `String`?
2. ¿Cómo se guardan esas cadenas en memoria?
3. ¿Solo se pueden ordenar letras?

Recordemos que, en memoria, una cadena de tipo `String` es un arreglo de datos de tipo caracter. Además, es necesario entender que cualquier caracter utilizado por una máquina en realidad tiene un valor `ASCII`. Esto significa que cualquier letra, símbolo o número representado por una máquina, se puede escribir como una secuencia de caracteres. Basado en esta premisa, entendemos que convirtiendo cualquier dato en un arreglo de caracteres, podemos invertir su secuencia y obtener un nuevo dato.

2.1. Definición del problema de la "inversión de la cadena"

Así, el problema de invertir se define a partir de:

1. una secuencia S de elementos $a \in \mathbb{N}$

producir una nueva secuencia S' cuyos elementos cumplan con la relación $a_n,\ a_{n-1},\ ...\ a_0$.

- Entradas:
  - $S = \langle a_i \in \mathbb{N} \rangle \mid 1 \leq i \leq n$.
- Salidas:
  - $S' = \langle e_i \in Sm \rangle \mid e_n,\ e_{n-1} \forall i \in [1, n)$.

3. Algoritmos de solución

3.1. Iterativo

La idea de este algoritmo es: invertir la cadena recorriendo todos elementos desde la última hasta la primera posición.

Algoritmo 1 Binario inverso Iterativo: Convertir un entero a cadena de caracteres binarios

Require: $n \in \mathbb{N}$

```
1: procedure DecimalToBinary(n)
2:     binary ← [*]char
3:     i ← 0
4:     while n! = 0 do
5:         i ← i + 1
6:         binary[i] ← (char)(n %2)
7:         n ← Floor(n/2)
8:     end while
9:     return binary
10: end procedure
```

Algoritmo 2 Binario inverso Iterativo: Invertir una cadena de caracteres

```
1: procedure InvertChain(S)
2:     if |S| = 1 then
3:         return S[1]
4:     end if
5:     inverted ← [*]char
6:     for i ← |S| to 1 step −1 do
7:         inverted[i] ← S[i]
8:     end for
9:     return inverted
10: end procedure
```

Algoritmo 3 Binario inverso Iterativo: Convertir una cadena de caracteres binarios a un entero

Ensure: $n \in \mathbb{N}$

```
1: procedure BinaryToDecimal(binary)
2:     decimal ← 0
3:     i ← 0
4:     invertedChain ← InvertChain(binary)
5:     for i, ← 1 to |invertedChain| do
6:         p ← Pow((int)invertedChain[i] ∗ 2, i)
7:         decimal ← decimal + p
8:     end for
9:     return decimal
10: end procedure
```

3.1.1. Análisis de complejidad

Por inspección de código: Se puede observar que, cada paso de la resolución del problema tiene un único ciclo, por lo que la complejidad de cada uno es $O(n)$. Vale la pena resaltar que el algoritmo que realmente nos interesa es el `Algoritmo 2`, pues es quien invierte cualquier cadena. El `Algoritmo 1 y 3` realmente son auxiliares para esos sub-problemas (convertir enteros a binarios y viceversa) en particular.

3.1.2. Invariante

Después de cada iteración controlada, el elemento i queda en el lugar invertido que le corresponde, es decir $|S| - 1$.

1. Inicio: $i = 0$, la secuencia vacía está invertida.
2. Iteración: $1 \leq i < |S|$, si se supone que los $i - 1$ elementos ya están en su posición, entonces la nueva iteración llevará los i-ésimo elementos a su posición adecuada, es decir al inicio de la secuencia.
3. Terminación: $i = 0$, los $|S|$ elementos están en su posición, entonces la secuencia está invertida completamente.

3.2. Dividir y vencer

La idea de este algoritmo es: en primera medida, entender que la inversión de una cadena con un único elemento, es igual a la cadena inicial, este siendo el caso base. Por otra parte, eligiendo un pivote en el medio de la secuencia, permite dividirla en subsecuencias de carácteres, que a su vez, son invertidas de derecha a izquierda. Para este caso, también se cuenta con los `algoritmos 1 y 3`, para convertir números enteros a cadenas binarias y viceversa.

Algoritmo 4 Binario inverso DyV

```
1:  procedure InvertChain(S, l, r)
2:      if l = r then
3:          return S[i]
4:      end if
5:      q ← Floor((l + r)/2)
6:      left ← InvertChain(S,l,q)
7:      right ← InvertChain(S,q+1,r)
8:      result ← [*]char
9:      for i ← 1 to |right| do
10:         result[i] ← (right[i])
11:     end for
12:     for i ← 1 to |left| do
13:         result[i] ← (left[i])
14:     end for
15:     return result
16: end procedure
```

3.2.1. Análisis de complejidad

El algoritmo ¨InvertChain de dividir y vencer¨ tiene orden de complejidad $O(1)$ cuando hay un elemento en la secuencia S, es decir $n = 1$.

Cuando $n > 1$ la complejidad del algoritmo según el teorema maestro está dada por:

$$T(n) = 2T\ \frac{n}{2} + O(n) \tag{1}$$

El valor de $a = 2$
El valor de $b = 2$
La complejidad de $C(n) = O(n)$

Esto resulta en un órden de complejidad similar a `MergeSort`, con una complejidad de $O(n)$ para su función auxiliar. Esto hace que la complejidad algorítimica de este algoritmo sea $O(nlogn)$; siendo más lento que su contraparte iterativa.

3.2.2. Invariante

La invariante esta dada por: en cada paso de división se realiza el intercambio de la secuencia desde el pivote hacia los extremos.

4. Análisis experimental

En esta sección se presentarán los resultados del experimento para confirmar los órdenes de complejidad de los dos algoritmos presentados en la sección 3.

4.1. Múltiplos de 10

Acá se presentan los experimentos cuando los algoritmos se ejecutan con números que son múltiplos de 10, es decir, si contiene a 10 varias veces exactamente. Además, en cada iteración cada número va incrementando en un digito.

4.1.1. Protocolo

1. Definir una muestra de 8 números, cada número debe incrementar en un digito a el anterior respectivamente.
2. Cada algoritmo se ejecutará 10 veces con cada número y se guardará el tiempo promedio de ejecución.
3. Se genera el gráfico necesario para comparar los algoritmos.

4.1.2. Experimentación

1. Se utilizó el programa *run_experiment.cpp* el cual internamente define los 8 números requeridos, que son:
   - `10,340,1890,14870,354880,1254690,78465460,988641540`
2. El programa logró finalizar sin complicaciones, con un tiempo expresado en segundos.
3. Se obtuvieron los mismos 8 datos con el tiempo promedio de cada uno de los algoritmos (Ver tabla 1), logrando así crear la siguiente gráfica: (ver Figura 1)

N	BinaryInversedIterative (s)	BinaryInversedDyV (s)
10	0,0002705	0,0001263
340	0,0001051	0,0001246
1890	0,0001779	0,0001508
14870	0,0001377	0,0001778
354880	0,0001627	0,0002362
1254690	0,0001563	0,0002218
78465460	0,0001574	0,0002359
988641540	0,0001543	0,0002757

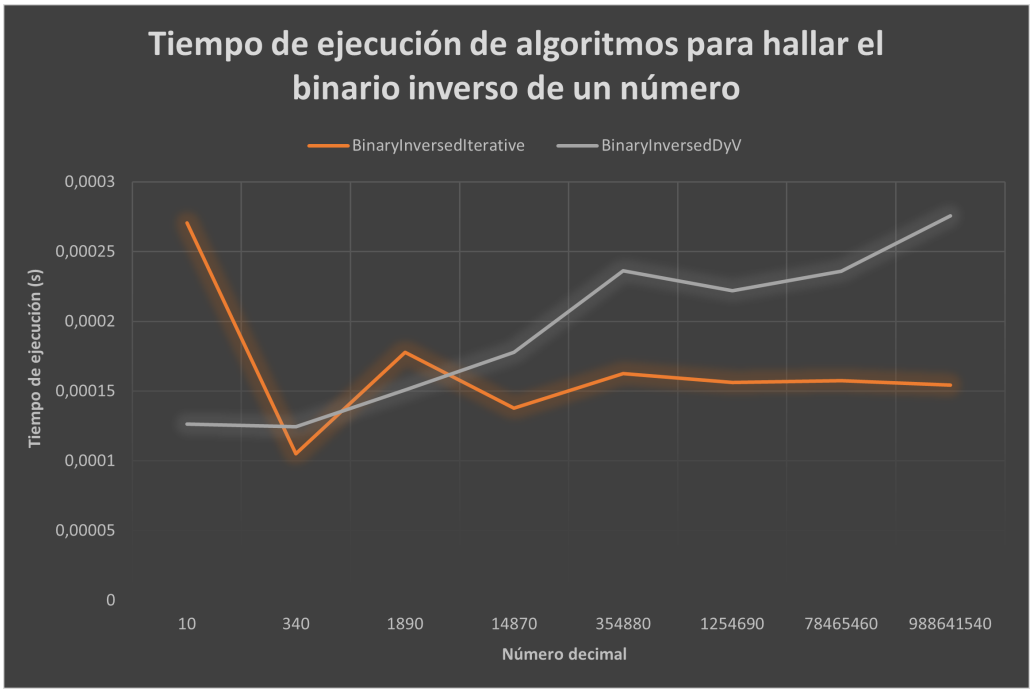

Figura 1: Tiempo de ejecución de los algoritmos

4.1.3. Análisis del experimento

Como se expreso previamente, el algoritmo iterativo tiene una complejidad $\Theta(n)$ y el algoritmo de dividir y vencer tiene una complejidad $\Theta(nlogn)$, por lo que se asume que el algortimo más rápido es el iterativo, sin embargo, a partir del segundo número múltiplo de 10 seleccionado, se empieza a notar esto. Por el contrario, para el caso del primer número no se cumple, puesto que el $\log(n)$ siendo $n \leq 10$ dará como resultado un número ≤ 1 para finalmente reducir el tiempo de ejecución del algortimo de dividir y vencer sobre el iterativo. Para los números > 10 el tiempo del algoritmo iterativo será menor, como se esperaba, al hallar las complejidades de cada algoritmo.

5. Conclusiones

El experimento fue útil para concluir:

1. El tiempo de ejecución para el caso promedio del algoritmo planteado de dividir y vencer tiene una complejidad igual al algoritmo de `MergeSort` siendo $\Theta(nlogn)$
2. Aunque se plantee un algoritmo de dividir y vencer, no implica que sea el adecuado para solucionar el problema.
3. Para determinar si un algoritmo es más rápido que otro, debe evaluarse cada caso individualmente, aunque su orden de complejidad sea menor no garantiza que en todos los casos el tiempo también lo sea.