Taller 2

Alejandro Morales Contreras[1]

[1]Departamento de Ingeniería de Sistemas, Pontificia Universidad Javeriana
Bogotá, Colombia
a.moralesc@javeriana.edu.co

18 de agosto de 2022

Índice

1. Enunciado del problema

Escribir e implementar un algoritmo basado en la estrategia "dividir-y-vencerz compararlo contra su versión ïterativa".

El problema a trabajar es "a partir de un número natural, calcular su representación binaria inversa". Por ejemplo, la representación binaria del número 345 es 101011001, y su representación inversa es 100110101, que es el número 309.

2. Formalización del problema

Dado un número natural ($x \in \mathbb{N}$), su representación binaria se puede expresar así:

$$x = 2^{n-1} \cdot B_1 + 2^{n-2} \cdot B_2 + \cdots + 2^0 \cdot B_n$$

$$B = \langle a_i \in \langle 0, 1 \rangle \rangle \mid 1 \leq i \leq n$$

$$b = 10^{n-1} \cdot b_1 + 10^{n-2} \cdot b_2 + \cdots + 10^0 \cdot b_n$$

donde B es la representación binarira en forma secuencial y b es la representación binaria en decimal (base-10). Encontrar la representación binaria inversa de x consistiría invertir la suma de la multiplicación de B_i con las potencias de 2 de $[n-1, 0]$ a $[0, n-1]$ así:

$$y = 2^0 \cdot B_1 + 2^1 \cdot B_2 + \cdots + 2^{n-1} \cdot B_n$$

2.1. Definición del problema "representación binaria inversa"

El problema de hallar la representación binaria inversa de un número se define a partir de un número $x \in \mathbb{N}$, encontrar el número y cuya representación binaria resulta de invertir (leer de derecha a izquierda) la representación binaria de x.

- Entradas:
  - $x \in \mathbb{N} \mid x = 2^{n-1} \cdot B_1 + 2^{n-2} \cdot B_2 + \cdots + 2^0 \cdot B_n \wedge B = \langle a_i \in \langle 0, 1 \rangle \rangle \wedge 1 \leq i \leq n$
- Salidas:
  - $y \in \mathbb{N} \mid y = 2^0 \cdot B_1 + 2^1 \cdot B_2 + \cdots + 2^{n-1} \cdot B_n$

3. Algoritmos de solución

3.1. Iterativo

3.1.1. Idea general de la solución

Convertir un número x a su representación binaria puede hacerse dividiendo entre 2 repetidamente el número, y agarrando los residuos y el último cociente para generar la representación. En la figura 1, se presenta un ejemplo de este cálculo para el número $x = 13$.

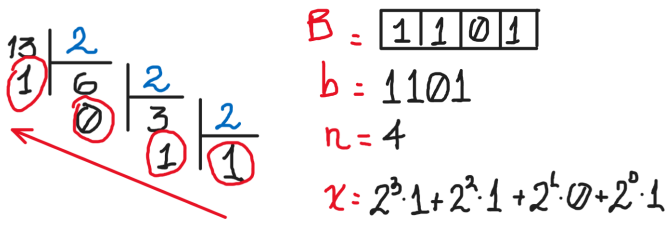

Figura 1: Calcular representación binaria de $x = 13$

La representación binaria de 13 sería entonces 1101. Calcular la representación binaria inversa sería ahora calcular la suma de la multiplicación de los dígitos de b con las potencias de 2 desde 0 hasta $n-1$. Para el ejemplo, podría expresarse como:

$$y = 2^0 \cdot 1 + 2^1 \cdot 1 + 2^2 \cdot 0 + 2^3 \cdot 1 = 11$$

La idea de solución consistiría entonces en dividir de dicha forma al número, guardando en un nuevo número b los dígitos generados y también calculando n (cantidad de dígitos). Finalmente, se extraerían los dígitos de b y se calcularía la suma de potencias.

Para hacer esta lógica de división y residuo, se utilizarían los operadores de división entera (también expresada como división piso) y el módulo. Así mismo, el proceso truncar el último dígito de un número, así como extraerlo, se haría utilizando estos operadores con 10 como cociente.

3.1.2. Escritura del algoritmo

Algoritmo 1 Calcular representación binaria inversa de forma iterativa.

```
1:  procedure ReverseBinaryRepresentationIterative(x)
2:      b ← 0
3:      factor2 ← 1 ∧ factor10 ← 1
4:      while x > 0 do
5:          b ← b + Mod(x, 2) * factor10
6:          x ← ⌊x ÷ 2⌋
7:          factor2 ← factor2 * 2
8:          factor10 ← factor10 * 10
9:      end while
10:     y ← 0
11:     while b > 0 do
12:         factor2 ← factor2 ÷ 2
13:         y ← y + Mod(b, 10) * factor2
14:         b ← ⌊b ÷ 10⌋
15:     end while
16:     return y
17: end procedure
```

3.1.3. Análisis de complejidad

Por inspección de código: hay un solo ciclo *mientras-que* anidado que siempre debe durar tantas *bits* iteraciones como tenga la representación binaria del número; entonces, este algoritmo es $O(n)$ donde n es la cantidad de bits que tiene la representación binaria de x.

3.1.4. Invariante

Después de cada iteración, b tiene los k primeros bits de la representación binaria.

1. Inicio: $b = 0$, no se ha determinado la representación binaria.
2. Iteración: $1 \leq k < n \mid 0 < x$, si se supone que los k primeros bits ya están en b, entonces la nueva iteración llevará un nuevo bit y los $k + 1$ primeros bit estarán en b.
3. Terminación: b corresponde a los n bits que son la representación binaria del número.

3.2. Dividir y vencer

3.2.1. Idea general de la solución

Para resolver el problema mediante un algoritmo dividir y vencer, se puede dividir el problema en tres partes: convertir el número natural en su representación binaria, invertir la representación binaria y convertir la representación binaria invertida de nuevo a número natural. Para la primera y tercera parte, se puede utilizar la misma lógica definida en la sección 3.1.1, de dividir repetidamente el número entre 2 y "agarrar"los

residuos y el último cociente, y de hacer la suma de multiplicación de potencias 2 con los dígitos del número invertido.

El problema de invertir un número (mediante un algoritmo dividir y vencer) se puede asemejar al de invertir un árbol binario. En la figura 2 se presenta un ejemplo de este ejercicio. Enfoncándose solo en los nodos hoja, se puede ver como invertir los nodos hermanos desde el nivel más profundo hacia arriba, lleva a invertir todo el árbol.

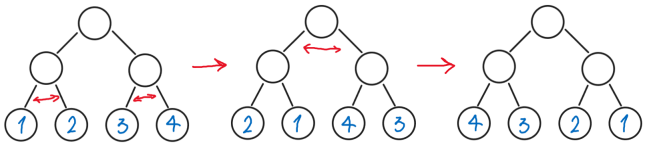

Figura 2: Invertir un árbol binario

Este proceso se puede asemejar al de invertir el número, donde invertir desde los grupos de 2 elementos (y a dígitos) más pequeños posibles hasta los más grandes, lleva a invertir todo el número. En la figura 3 se presenta un ejemplo de este proceso.

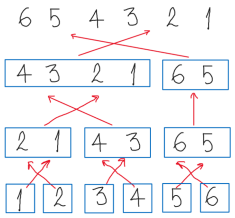

Figura 3: Invertir un número mediante un algoritmo dividir y vencer

El proceso ahora se reduce a hallar una forma de dado un número, invertir sus dígitos desde un dígito en la posición *beg* hasta un dígito en la posición *end*, con un pivote en la posición q. Una primera forma que se podría pensar, es la de utilizar una secuencia donde cada elemento es un dígito del número. Sin embargo, no hay necesidad de hacer esto si se utilizan las operaciones de truncar y reducir un número con ayuda de la división piso y módulo con cociente 10^a.

Para demostrar esto, se asume el ejemplo de invertir los dígitos del número 123456 con $n = 6$ cantidad de dígitos, $beg = 2$ dígito de inicio, $end = 5$ dígito de fin y $q = 3$ pivote. El resultado debería ser 145236. Para empezar, se deben "cortar"todos los dígitos que estén antes de *beg* y después de *end*.

$$num\_left = \lfloor 123456 \div 10^{6-2+1} \rfloor = 1$$
$$num\_right = \text{Mod}(123456, 10^{6-5}) = 6$$

Ahora, se calculan los dígitos que se ubican entre *beg* y *end* inclusive

$$num\_mid = \text{Mod}(\lfloor 123456 \div 10^{6-5} \rfloor, 10^{5-2+1}) = 2345$$

Después, se dividen los dígitos de la mitad en dos grupos: los que están a la izquierda del pivote inclusive, y los que están a la derecha del pivote.

$$left = \lfloor 2345 \div 10^{5-3} \rfloor = 23$$
$$right = \text{Mod}(2345, 10^{5-3}) = 45$$

Luego, se pasan los dígitos de la derecha a la izquierda. Esto se puede hacer fácilmente multiplicando los dígitos de la derecha por la potencia de 10 a la pivote dígitos.

$$num\_mid = 45 * 10^{5-3} + 23 = 4523$$

Finalmente, se unen las tres partes.

$$reversed = 1 * 10^{6-2+1} + 4523 * 10^{6-5} + 6 = 145236$$

3.2.2. Escritura del algoritmo

Algoritmo 2 Calcular representación binaria inversa de forma dividir y vencer.

```
1: procedure ReverseBinaryRepresentationDC(x)
2:     b ← 0 ∧ n ← 0
3:     factor10 ← 1
4:     while x > 0 do
5:         b ← b + Mod(x, 2) * factor10
6:         x ← ⌊x ÷ 2⌋
7:         n ← n + 1
8:         factor10 ← factor10 * 10
9:     end while
10:    b2 ← ReverseNumber(b, n, 1, n)
11:    y ← 0
12:    factor2 ← 1
13:    while b2 > 0 do
14:        y ← y + Mod(b2, 10) * factor2
15:        b2 ← ⌊b2 ÷ 10⌋
16:        factor2 ← factor2 * 2
17:    end while
18:    return y
19: end procedure
```

Algoritmo 3 Invertir un número.

```
1: procedure ReverseNumber(num, n, beg, end)
2:     if beg < end then
3:         q ← ⌊(beg + end) ÷ 2⌋
4:         num ← ReverseNumber(num, n, beg, q)
5:         num ← ReverseNumber(num, n, q + 1, end)
6:         return ReverseNumberAux(num, n, beg, q, end)
7:     else
8:         return num
9:     end if
10: end procedure
```

Algoritmo 4 Invertir un número desde el dígito *beg* hasta el dígito *end* en un pivote dígito *q*.

```
1: procedure ReverseNumberAux(num, n, beg, q, end)
2:     pow1 ← Pow(10, n − beg + 1)
3:     pow2 ← Pow(10, n − end)
4:     pow3 ← Pow(10, end − q)
5:     num_left ← ⌊num ÷ pow1⌋
6:     num_right ← Mod(num, pow2)
7:     num_mid ← Mod(⌊num ÷ pow2⌋, pow1 ÷ pow2)
8:     left ← ⌊num_mid ÷ pow3⌋
9:     right ← Mod(num_mid, pow3)
10:    num_mid ← right * pow3 + left
11:    return num_left * pow1 + num_mid * pow2 + num_right
12: end procedure
```

3.2.3. Análisis de complejidad

El algoritmo de solución consite en tres partes: la conversión del número en binario, la inversión del número, y el posterior cálculo del numero binario invertido en decimal, $O = O_1 + O_2 + O_3$, en donde solo se preserva la de mayor complejidad. La complejidad de O_1 y O_3 está dada por un ciclo *mientras-que* que dura tantas iteraciones como *bits* tenga el número en su representación binaria. Entonces, estas complejidades son $O_1(n)$ y $O_3(n)$.

Por su parte, la inversión del número binario es un algoritmo dividir y vencer, y su complejidad puede ser expresada así y hallada a través del teorema maestro:

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

Utilizando las tres ecuaciones, se encuentra:

1. $1 = n^{\log_b(a-\epsilon)} = n^{\log_2(2-\epsilon)} \wedge \epsilon = 1 \rightarrow \Theta(n^{\log_b a}) = \Theta(n)$
2. $1 = n^{\log_b a} \log_2^k n = n^{\log_2 2} \log_2^k n \wedge k =?$
3. $1 = n^{\log_b(a-\epsilon)} = n^{\log_2(2+\epsilon)} \wedge \epsilon = -1$

Por ende la complejidad de invertir el número binario es $\Theta_2(n)$. Debido a que todas las partes son equivalentes, la complejidad del algoritmo es $\Theta(n)$ donde n es la cantidad de bits que tiene la representación binaria de x.

3.2.4. Invariante

Después de cada llamado a $ReverseNumberAux$, el número tiene sus dígitos desde la posición beg hasta q y desde la posición $q + 1$ hasta end invertidos.

4. Análisis experimental

En esta sección se presentarán algunos los experimentos para comparar el algoritmo dividir y vencer contra el iterativo para determinar cuál se desempeña mejor. Así mismo, también servirán para confirmar los órdenes de complejidad de los dos algoritmos presentados en la sección 3. Así mismo,

4.1. Potencias de dos

La idea consiste en analizar las k primeras potencias de 2. Esto debido a que cada k-ésima potencia tiene $k + 1$ bits en su representación binaria.

4.1.1. Protocolo

1. Definir un número $k \in \mathbb{N}$. Se generarán las primeras k potencias de 2 desde 2^0 hasta 2^{k-1}.
2. Cada algoritmo se ejecutará 1000 veces con cada potencia y se guardará el tiempo promedio de ejecución.
3. Se generan los gráficos necesarios para comparar los algoritmos.

4.2. Números aleatorios

La idea consiste en analizar n números aleatorios de distinta cantidad de bits en su representación binaria.

4.2.1. Protocolo

1. Definir un número $n \in \mathbb{N}$. Se generarán n números aleatorios de distinta cantidad de dígitos.
2. Cada algoritmo se ejecutará 10 veces con cada número aleatorio y se guardará el tiempo promedio de ejecución.
3. Se compara el tiempo total promedio de ejecución de ambos algoritmos, correspondiente a la suma de los tiempos promedios de ejecución del punto 2.
4. Repetir desde el paso 1 tantas veces como se requiera

5. Resultados de los experimentos

Los algoritmos se implementan en C++, y sus experimentos se realizan siguiendo el protocolo definido sección 4. Los resultados de los experimentos esperan comparar ambos algoritmos, así probar (o refutar) los análisis de complejidad realizados para cada uno de ellos en la sección 3.

Debido a que la implementación se realiza en C++, y ambos algoritmos definen una forma de calcular el número binario en donde se almacena la representación en número base-10, existe un límite equivalente a la cantidad de bits máxima que una variable númerica puede almacenar en C++. Se utiliza el tipo de variable *unsigned long long*, que puede almacenar números de hasta 20 dígitos. Es por esto que el máximo número binario que se podría almacenar en esta variable es equivalente a $2^{19} = 524288$, el cual es un límite impuesto sobre el algoritmo.

Para el primer experimento, se espera hacer primeramente una inspección empírica de la gráfica resultante en donde se grafica el tamaño de bits de las potencias de dos contra el tiempo promedio que le toma a cada algoritmo determinar su representación binaria inversa en nanosegundos. Se esperaría que la gráfica resultante fuera una línea recta.

Después de la inspección de la gráfica, es necesario confirmar más detalladamente si realmente los resultados prueban el análisis de complejidad. Una buena forma de confirmar esto es mediante una regresión de los datos. Ya que ambos algoritmos presentan una complejidad de $O(n)$, una regresión lineal simple es la operación adecuada para confirmar los resultados.

La regresión lineal simple pretende ajustar una curva de la forma:

$$f(X) = aX + b$$

donde X representaría la variable independiente (cantidad de bits) y $f(X)$ la variable dependiente (tiempo que toma calcular la representación binaria inversa). Después de la regresión, se obtienen los coeficientes del polinomio a y b. La medida que se utiliza para determinar si la regresión se cumple es el coeficiente de determinación (R^2), el cual refleja que tan bien ajustado es el polinomio resultante a los datos de entrada. Esta se calcula así:

$$R^2 = \frac{\sum_{n=1}^{N}(\hat{Y_n} - \bar{Y})^2}{\sum_{n=1}^{N}(Y_n - \bar{Y})^2}$$

en donde:

- N representa la cantidad de registros de las variables
- $\bar{Y}$ representa la media de la variable dependiente de entrada
- $\hat{Y_n}$ representa $f(X_n)$ o el resultado de evaluar el polinomio de regresión en X_n
- Y_n representa el n-ésimo dato de la variable dependiente de entrada

R^2 varía en el rango $(0, 1)$ y a mayores medidas de este se espera una mayor confianza en que los resultados se asemejan a un polinomio cuadrático.

5.1. Potencias de dos

Para el experimento con potencias de dos, se sigue el protocolo definido en la sección 4.1.1, dando como entrada $k = 20$. Obtenidos los resultados, se genera el gráfico comparativo de los algoritmos presentado en la figura 4.

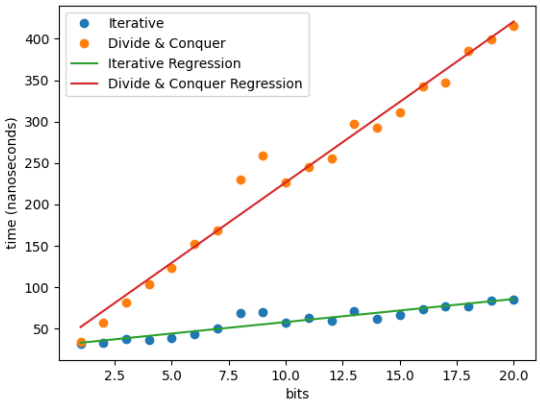

Figura 4: Gráfica comparativa de potencias de dos

La inspección empírica de la gráfica parece revelar que ambos algoritmos siguen una línea recta, cumpliendo con el análisis de complejidad. Así mismo, se resalta que el algoritmo iterativo se desempeña mucho mejor que el algoritmo dividir y vencer.

Posteriormente, se realiza la regresión lineal y se obtienen los siguientes resultados:

Algoritmo	a	b	R^2
Iterativo	2.782406	30.204736	0.877880
Dividir y vencer	19.421654	32.502632	0.976356

Como se puede ver, ambos algoritmos obtienen un buen R^2. Sin embargo, no tienen un ajuste bien alto, debido a que el tamaño de la muestra es muy pequeño, así como que el sistema operativo pudo haber interrumpido los procesos. Con este experimento, se puede probar que los algoritmos tienen un orden de complejidad $O(n)$.

5.2. Números aleatorios

Para el experimento con números aleatorios, se sigue el protocolo definido en la sección 4.2.1, dando como entrada $n \in N \mid N = \langle 100, 1000, 10000, 100000, 1000000, 10000000 \rangle$. Los resultados de los tiempos se presentan en la siguiente tabla.

n	Iterativo (ms)	Dividir y vencer (ms)
100	0.00699	0.03291
1000	0.06849	0.32310
10000	0.67827	3.21113
100000	6.68253	31.7693
1000000	65.8936	313.527
10000000	677.631	3196.51

Como se puede evidenciar, en todos los casos de n cantidad de números aleatorios, el algoritmo iterativo va más rápido que el algoritmo dividir y vencer. Aproxidamente, el algoritmo dividir y vencer es 5 veces más lento que el iterativo. Sin embargo, cabe resaltar que ambos algoritmos tienen un orden de complejidad similar $O(n)$ y que son sus implementaciones (por ejemplo, cantidad de instrucciones) las que hacen que uno se desempeñe mejor que el otro.