Multiplicación Rusa para números grandes

Sergio Salinas, Danilo Abellá

Resumen—Dado que en la codificación de algoritmos de multiplicación convencional se utilizan variables numéricas para guardar los factores y el producto, esto termina limitando el tamaño máximo de dígitos que se pueden utilizar para representar un número debido al uso limitado de memoria. Para poder llevar a cabo una multiplicación de números gigantes se propone utilizar el algoritmo de la multiplicación rusa, en el cual guardando los factores y el producto en cadenas de caracteres y utilizando memoria dinámica, se va operando carácter por carácter hasta finalizar la operación, haciendo uso de los clásicos métodos de suma, división y multiplicación en el proceso.

Index Terms—Multiplicación Rusa, Números grandes, C.

1. Introduction

El sistema de multiplicación aprendido en la educación básica es el más habitual a nivel mundial desde que se extendió la numeración arábiga (el sistema decimal que es utiliza actualmente), sin embargo hay otros muchos métodos o estrategias para obtener el resultado de una multiplicación, ya sea de forma mas eficiente o accesible a la hora de programar.

Desde hace muchos siglos las matemáticas han sido un punto fuerte en países como Rusia y del Norte de Europa, logrando así un éxito muy considerable al lograr aportar métodos muy interesantes para la multiplicación [MORENO et al. [2006]]. Uno de los más conocidos es el llamado método de los campesinos rusos (o simplemente, multiplicación rusa), un sistema que podemos definir como "lento pero seguro". De las características mas importantes a destacar se tiene que los únicos conocimientos requeridos son saber sumar, así como dividir y multiplicar por dos, sin saber ninguna otra tabla de multiplicación, por lo que brinda una perspectiva muy diferente a la hora de operar.

El tema a analizar en este informe es la efectividad del algoritmo del método de la multiplicación rusa para multiplicar números gigantes que no pueden ser utilizados en algoritmos de multiplicación mas convencionales, por falta de memoria.

2. ¿Qué es la Multiplicación?

La multiplicación consiste en una operación de composición que requiere sumar reiteradamente un número de acuerdo a la cantidad de veces indicada por otro, o sea, una suma abreviada de sumandos iguales, que pueden repetirse muchas veces [Rees [1986]].

$$2\ x\ 4\ =\ 2\ +\ 2\ +\ 2\ +\ 2\ =\ 8$$

El significado de su palabra lo dice todo, la cual es originada del latín "multus" que corresponde a mucho, y "plico", que es doblar.

Los números que forman parte del proceso de la multiplicación reciben el nombre de factores, mientras que el resultado se denomina producto. Por lo tanto, el objetivo de la operación es hallar el producto de dos factores. Cada factor, por otra parte, tiene su propia denominación. La cifra a sumar repetidamente es el multiplicando, mientras que el número que indica la cantidad de veces que hay que sumar el multiplicando es el multiplicador. La multiplicación, en definitiva, consiste en tomar el multiplicando y sumarlo tantas veces como unidades contiene el multiplicador para obtener un producto [Rees [1986]].

$$\begin{array}{lcl} multiplicando & & producto \\ \downarrow & & \downarrow \\ 2\ x\ 4 & = & 2\ +\ 2\ +\ 2\ +\ 2\ =\ 8 \\ \uparrow & & \\ multiplicador & & \end{array}$$

Como se puede ver en el ejemplo la multiplicación, 2 x 4 es lo mismo que sumar 2 cuatro veces, dando en ambos casos el 8 como producto.

La multiplicación cumple cuatro propiedades que harán más fácil la resolución de problemas. Estas son las propiedades conmutativa, asociativa, elemento neutro y distributiva [Smith [2000]].

La propiedad conmutativa dice que cuando se multiplican dos números, el producto es el mismo sin importar el orden de los multiplicandos.

$$\begin{array}{ccc} multiplicando & & producto \\ \downarrow & & \downarrow \\ 2\ x\ 4 & = 4\ x\ 2 & = \ 8 \\ \uparrow & & \\ multiplicador & & \end{array}$$

Como se muestra en el ejemplo, el producto de la multiplicación siempre será 8 independiente de si el 2 o 4 son multiplicando o multiplicador respectivamente.

Para la propiedad asociativa se afirma que cuando se multiplican tres o más números, el producto es el mismo sin importar como se agrupan los factores.

$$2\ x\ (\ 4\ x\ 3\ )\ =\ (\ 2\ x\ 4\ )\ x\ 3\ =\ 24$$

En este ejemplo se muestra que el producto de multiplicar el resultado de 4 x 3 por 2, es exactamente el mismo que el de multiplicar el resultado de 2 x 4 por 3, en ambos casos da 24.

Luego se tiene la propiedad del elemento neutro que dice que el producto de cualquier número por uno es el mismo número.

$$\begin{array}{c} elemento \\ neutro \\ \downarrow \\ 4\ x\ 1\ =\ 4 \end{array}$$

En el ejemplo se muestra el 4 como multiplicando y el 1 como multiplicador, el cual al ser este último el elemento neutro, da 4 como producto.

Finalmente tenemos la propiedad distributiva que dice que la suma de dos números por un tercero es igual a la suma de cada sumando por el tercer número.

$$2\ x\ (6\ +\ 4\ )\ =\ 2\ x\ 6\ +\ 2\ x\ 4\ =\ 20$$

El multiplicar la suma de 6 + 4 (o sea 10) por 2 da exactamente el mismo resultado que sumar la multiplicación de 2 x 6 mas 2 x 4, siempre dará 20 como resultado.

3. La multiplicación rusa

Es un método de multiplicación basado en la forma en que multiplicaban los campesinos rusos en el siglo XIX [Dasgupta and Vazirani [2006]], el método esta basado en duplicar y reducir números a la mitad, también se le llama multiplicación binaria debido a que su lógica esta basada la forma binaria de los números

Sean n y m dos números enteros positivos de los que se quiere conocer su producto, se puede computar el resultado usando el método de la multiplicación rusa o multiplicación binaria, este método divide constantemente el primer número por 2 y multiplica constantemente el segundo número por 2 [Levitin [2014]].

El método se puede expresar mediante las dos siguientes formulas.

Si n es par

$$n \cdot m = \frac{n}{2} \cdot 2m$$

Si n es impar

$$n \cdot m = \frac{n-1}{2} \cdot 2m + m$$

Estas formulas tienen como caso trivial cuando $1 \cdot m = m$ para parar. Con ellas se puede calcular el producto de $n \cdot m$ recursivamente o iterativamente. En la tabla 3 se muestra un ejemplo de multiplicar 121*35 usando este método.

n	m	Sumar
121	35	35
60	70	
30	140	
15	280	280
7	560	560
3	1120	1120
1	2240	2240
		4235

Tabla 1
Multiplicación de $121 \cdot 35$

El pseudocódigo de la versión iterativa se muestra en el algoritmo 1, en la linea 6 se aprecia que se puede mezclar la división de las dos formulas en una usando la función piso que aproxima el resultado de la división por dos al menor entero, aun así se sigue teniendo el caso de cuando a es impar ya que este caso dice cuando hay que sumar b al resultado.

Algorithm 1: Multiplicación rusa

```
Data: Dos enteros positivos a y b
Result: El resultado del producto de a y b.
1 begin
2     res ← 0
3     while a > 0 do
4         if a is impar then
5             res ← res + b
6         a ← ⌊a/2⌋
7         b ← 2 · b
8     return res
```

La multiplicación rusa funciona debido a que la multiplicación es distributiva [Dasgupta and Vazirani [2006]], por ejemplo.

n	m	Sumar	n	m	Sumar
121	35	35	1111001	100011	100011
60	70		111100	1000110	
30	140		11110	10001100	
15	280	280	1111	100011000	100011000
7	560	560	111	1000110000	1000110000
3	1120	1120	11	10001100000	10001100000
1	2240	2240	1	100011000000	100011000000
		4235			1000010001011

Tabla 2
Multiplicación de $121 \cdot 35$ con su paralelo en forma binaria

$$\begin{aligned} 12 \cdot 9 &= 12 \cdot (1 \cdot 2^0 + 0 \cdot 2^1 + 0 \cdot 2^2 + 1 \cdot 2^3) && (1)\\ &= (12 \cdot 2^0 + 0 \cdot 2^1 + 0 \cdot 2^2 + 12 \cdot 2^3) && (2)\\ &= 12 + 96 && (3)\\ &= 108. && (4) \end{aligned}$$

Al proceso de multiplicar repetidamente un número por dos se le llama dubling y al de dividir continuamente por dos halving [Dasgupta and Vazirani [2006]].

La multiplicación rusa usa el proceso de halving para convertir el primer número a multiplicar en binario y la técnica de dubling que es la que simula el 2^n en el ejemplo, de esta forma multiplica dos números convirtiendo el primer número en sumas de factores de dos y multiplicando el segundo número por esos factores y sumando todos los factores de dos que no estén siendo multiplicados por cero, que es cuando hay un 1 en su expresión binaria.

La tabla 8 hace un paralelo entre la multiplicación rusa con los números en su forma decimal y binaria, con esta tabla se puede ver la relación que tiene multiplicación rusa con los números binarios ya que al dividir por dos quita un bit y al multiplicar por dos se agrega un bit.

La complejidad del algoritmo se calcula en base a la cantidad de iteraciones que tiene que hacer el algoritmo para dividir un número n hasta que este sea cero, asumiendo que el costo de la suma, multiplicación y división es constante ($\Theta(1)$). Como el primer número se divide constantemente por dos entonces la cantidad de iteraciones es igual a la cantidad de dígitos que tiene el número en su representación binaria.

Por ejemplo, el número 15 en binario es 1111, por lo que se deben hacer 4 iteraciones para que el algoritmo termine.

$$\begin{aligned} 106 &= (1101010)_2 && \div 2\\ 53 &= (110101)_2 && \div 2\\ 26 &= (11010)_2 && \div 2\\ 13 &= (1101)_2 && \div 2\\ 6 &= (110)_2 && \div 2\\ 3 &= (11)_2 && \div 2\\ 1 &= (1)_2 && \div 2 \end{aligned}$$

Como la cantidad de dígitos que tiene un número en su presentación binaria se puede obtener calculando el logaritmo en base 2 del número, entonces que la complejidad asintótica del algoritmo es $\mathcal{O}(\log n)$.

Aun así la complejidad asintótica de la implementación para que pueda trabajar con números grandes no es exactamente logarítmica, esto es debido a que para implementar el método se uso la multiplicación, división y suma clásica, por lo se requiere entender como funcionan estos métodos para saber el costo total de la multiplicación rusa.

La implementación de la multiplicación de la multiplicación rusa en C se muestra en el apéndice A.

4. Suma clásica

El algoritmo que se utiliza para sumar dos números es el de la suma clásica [Knuth [2014]] de dos números a y b con a $\geq$ b, la ventaja de usar este algoritmo a la hora sumar es que hace adiciones dígito a dígito permitiendo trabajar con números grandes sin tener que almacenar el número como un todo sino como un arreglo de varios números.

Solo se ve la condición de sumar dos números en que uno es siempre mayor o igual al otro debido a que en la multiplicación rusa cuando se suma el segundo número más el resultado, el segundo número siempre es mayor al resultado debido a que esta constantemente siendo multiplicado por dos.

Algorithm 2: Suma clásica

Input: Dos números A[0..la] y B[0..la]
Output: La suma C de A más B

```
1  begin
2      carry ← 0;
3      C[0..la + 1] ← 0;
4      Mov(B, la-lb);
5      B[0..(la − lb)] ← 0;
6      while la > 0 do
7          sum ← A[la − 1] + B[la − 1] + carry;
8          carry ← sum/10;
9          C[la] = sum mod 10;
10         la = la − 1;
11     if carry > 0 then
12         C[0] ← 1
13     return C;
```

En el algoritmo 2 se muestra el pseudocódigo en el que se baso al implementación para sumar. El algoritmo define un carry en la linea 2 que es lo que guarda el sobrante de sumar dos dígitos, cuando la suma es mayor a 9 la

unidad se guarda en el resultado y la decena en el carry. La linea 4 y 5 rellena por la derecha de ceros el número B para que así tenga el mismo largo que A. El ciclo while de la linea 6 recorre el número de mayor largo desde el final de este hasta el inicio, debido a que los números se escriben de derecha a izquierda, cuando el largo es 0 es porque se recorrió todo el número. De la linea 7 a 9 se hace la operación de suma dígito a dígito, si A tiene más dígitos que B entonces va a llegar un momento en que solo se sumen los dígitos de A más ceros, si ambos tienen igual cantidad de dígitos entonces puede que el carry sea 1 cuando se termine de recorrer el número mayor, por lo que en la linea 12 se agrega el carry al final del número, como el carry siempre es 1 solo se agrega un uno.

La complejidad del algoritmo es $\mathcal{O}(n)$, donde n es el la cantidad de dígitos que tiene el número mayor número, esto es debido a que en la iteración se hacen tantas operaciones de coste constante como dígitos tenga el número n.

$$
\begin{array}{r}
\downarrow \\
420 \\
+ \\
642 \\
\hline
2
\end{array}
\qquad
\begin{array}{r}
\downarrow \\
420 \\
+ \\
642 \\
\hline
62
\end{array}
\qquad
\begin{array}{r}
\downarrow \\
420 \\
+ \\
642 \\
\hline
062
\end{array}
$$

$$
\begin{array}{r}
\downarrow \\
1 \longrightarrow carry \\
+ \\
420 \\
+ \\
642 \\
\hline
1062
\end{array}
\qquad
\begin{array}{r}
420 \\
+ \\
642 \\
\hline
1062
\end{array}
$$

La implementación de la suma clásica en C se muestra en el apéndice D.

5. División larga

Para la división por dos se modifico el algoritmo de la división larga [Knuth [2014]] para que solo divida por dos. Este algoritmo es ventajoso debido a que nos permite trabajar con el número dígito a dígito, optimizando enormemente el uso en memoria en comparación a usar el número como un solo entero.

Algorithm 3: División por dos

Input: Un número A[0..la]
Output: La división de A por 2.

```
1 begin
2     carry ← 0
3     C[0..la + 1] ← 0
4     for i ← 0..la do
5         n ← (carry · 10 + a[i])/2
6         carry ← (carry · 10 + a[i]) − 2 · n
7         c[i] ← n
8     return C
```

En el algoritmo 3 se muestra el pseudocódigo en el que se baso al implementación dividir por dos, esta es la misma división que se enseña en los colegios actualmente solo que modificada para dividir por dos. En la linea 3 se rellena el arreglo que tendrá el resultados con 0, en la linea 4 empieza el ciclo que se hace la división, el ciclo se inicia apuntando al dígito en la posición más grande del número, en la linea 5 obtiene un dígito del resultado, este se obtiene al dividir por dos un número compuesto con el carry como décima y el dígito actual del ciclo como unidad, el carry es lo que le falta al resultado para llegar al dividendo y se obtiene al tener el número divido en dos anteriormente y restarle dos veces el resultado de la división y por último en la linea 7 se guarda el dígito del resultado.

La complejidad del algoritmo es $\mathcal{O}(n)$, donde n es el la cantidad de dígitos que tiene el número a dividir por dos, esto es debido a que en la iteración se hacen tantas operaciones de coste constante como dígitos tenga el número n.

La implementación de la división larga adaptada para solo dividir por dos en C se muestra en el apéndice B.

6. Multiplicación larga

El algoritmo 4 se presenta como una modificación del algoritmo de multiplicación clásica [Knuth [2014]] para que solo realice multiplicaciones donde unos de los factores siempre es 2, lo cual permite optimizar el código ya que tiene que considerar muchos menos casos.

Como primer paso se define el tamaño del resultado (línea 2). Luego la estrategia consiste en recorrer dígito por dígito el primer factor e ir multiplicándolo por 2 mientras se va guardando el resultado, en caso de que el resultado sea de dos dígitos, se guarda el primer dígito para

Algorithm 4: Multiplicación por dos

```
Input: Some input
Output: Some output
begin
    c ← [0, 0, ..,0]
    k ←i + 1
    for i ← (largo − 1).,0 do
        n ← a[i] ∗ 2 + c[k]
        carry ← n div 10
        c[k] ← n mod 10
        k ←k - 1
        c[k] ← c[k] sum carry
    if c[0] = 0 then
        Se elimina el primer digito de 'c'
    return C
```

luego sumarse en la siguiente multiplicación. Se continúa realizando este ciclo hasta recorrer todos los dígitos del primer factor. Una vez finalizado se obtiene el producto de la multiplicación por 2, y en caso de empezar con 0, se elimina el primer dígito.

La complejidad del algoritmo es $\mathcal{O}(n)$, donde n es el la cantidad de dígitos que tiene el número a multiplicar por dos, esto es debido a que en la iteración se hacen tantas operaciones de coste constante como dígitos tenga el número n.

Sea el ejemplo:

$$
\begin{array}{rl}
 & \downarrow \\
 & 150 \\
x & 2 \\
\hline
 & 0
\end{array}
\qquad
\begin{array}{rl}
 & \downarrow \\
 & 150 \\
x & 2 \\
\hline
 & 00
\end{array}
$$

$$
\begin{array}{rl}
 & \downarrow \\
 & 1 \longrightarrow carry \\
+ & \\
 & 150 \\
x & 2 \\
\hline
 & 00
\end{array}
\qquad
\begin{array}{rl}
 & 150 \\
x & 2 \\
\hline
 & 300
\end{array}
$$

La implementación de la multiplicación larga adaptada para solo multiplicar por dos en C se muestra en el apéndice C.

7. COMPLEJIDAD ASINTÓTICA

Anteriormente se dijo que la complejidad de la multiplicación rusa es de $\mathcal{O}(\log n)$ asumiendo que la operación multiplicación por dos, división por dos y suma de dos números enteros es de $\Theta(1)$, pero como ya se demostró en las secciones anteriores esto no era así, la complejidad del algoritmo requiere un nuevo análisis.

Sean dos números a multiplicar n y m, la multiplicación rusa divide constantemente n por dos hasta que llegue a 0, esto hace se hagan $\log_2(n)$ iteraciones cada vez que se multiplican dos números, en cada iteración se hace siempre una operación de división por dos y una operación de multiplicación por dos, ambas tienen coste $\mathcal{O}(n)$, además dependiendo si n es par o impar puede se haga una suma en cada ciclo. Con lo anterior la complexidad asintótica de la multiplicación rusa es:

$$
\begin{aligned}
\log_2(n) \cdot (\mathcal{O}(n) + \mathcal{O}(n) + \mathcal{O}(n)) &= \log_2(n) \cdot (\mathcal{O}(3n)) \\
&= \log_2(n) \cdot (\mathcal{O}(n)) \\
&= \log_2(n) \cdot \mathcal{O}(n) \\
&= \mathcal{O}(\log_2(n) \cdot n) \\
&= \mathcal{O}(n \cdot \log(n))
\end{aligned}
$$

Por lo que la complejidad total del algoritmo es $\mathcal{O}(n \log(n))$.

8. LA ESTRUCTURA DE DATOS

Los números son almacenados en una cadena de caracteres (string), esto debido a de esta manera se puede trabajar con el número dígito a dígito y porque en C un carácter es la unidad más pequeña almacenamiento que se tiene [ISO/IEC 9899:1999], gastando tan solo 8 bits, la siguiente unidad de almacenamiento que le sigue es el short int que usa 16 bits.

Para la asignación de memoria se utiliza memoria dinámica, debido a que en la multiplicación rusa el número a ser multiplicado es divido constantemente por dos, lo que asegura que la cantidad de memoria va desde n hasta 1, la memoria dinámica permite asignar una cantidad de memoria n que sea acorde al largo del número que se ingresa [ISO/IEC 9899:1999], permitiendo que se pueda modificar constantemente dependiendo del largo de los números que se ingresan.

9. COMPARACIÓN CON OTROS MÉTODOS DE MULTIPLICAR

9.1. Multiplicación egipcia

Este algoritmo de multiplicar es muy similar a la multiplicación rusa y se viene usando desde el antiguo Egipto [Dasgupta and Vazirani [2006]], cuando se quiere multiplicar dos números n y m se escriben dos columnas,

en una columna se escribe un uno y en la otra uno de los factores a multiplicar, en este caso b, luego se van duplicando los valores hasta que la columna que tenía el 1 no rebase el otro factor a multiplicar (a), luego se buscan que factores de la primera columna sumados hacen al primer factor (a) y la suma total es el resultado de la multiplicación.

En la tabla 9.1 se muestra un ejemplo de como funciona la multiplicación.

2^i	m	Sumar
1	35	35
2	70	
4	140	
8	280	280
16	560	560
32	1120	1120
64	2240	2240
		4235

Tabla 3
Multiplicación de $121 \cdot 35$ usando la multiplicación egipcia

La lógica es la misma que la multiplicación rusa, esta basada en el sistema binario, lo que hace es crear uno de los factores en binario al multiplicar constantemente 1 por 2, por ejemplo, el número 121 mostrado en binario es $(1111001)_2$, si uno mira el ejemplo de la tabla 9.1 se puede observar que solo se suman las posiciones en la primera columna que son equivalente a un uno en el 121 binario.

Como la multiplicación rusa y egipcia tienen la misma lógica también tienen la misma complejidad, la cantidad de iteraciones que tiene la multiplicación egipcia es $\log_2(n)$, donde n es el primer número a multiplicar, esto se debe a que se tienen tanto valores en la primera columna como dígitos en su valor binario tenga n, esto determina la cantidad de iteraciones que hará el algoritmo y la cantidad de operaciones que se hacen dentro de esta, asumiendo que la suma y la multiplicación por dos son de coste constante $\Theta(1)$ entonces la complejidad total de la multiplicación egipcia es $\mathcal{O}(n\log(n))$.

9.2. La multiplicación larga

Anteriormente se hablo de la multiplicación larga, una versión modificada de esta se uso en la implementación debido a que permite el trabajo dígito a dígito de los números a multiplicar aunque esto también es una desventaja. Si se tienen dos números de largo n y m y se desean multiplicar, la multiplicación larga debe operar cada dígito de n y cada dígito de m haciendo que la cantidad total de operaciones sea $n \cdot m$ haciendo que la complejidad total sea $\mathcal{O}(nm)$, o en caso de que ambos tengan la misma cantidad de dígitos n la complejidad es $\mathcal{O}(n^2)$, haciendo que sea menos eficiente que la multiplicación rusa en tiempo.

Como ventaja por sobre la multiplicación rusa se tiene que la multiplicación larga ocupa menos memoria, si se quieren multiplicar dos números de largo n y m, la cantidad de memoria total ocupada es $2n + 2m$, esto es debido a que el resultado de la multiplicación de dos números n y m es como máximo la suma del largo de ambos números. En cambio la multiplicación rusa usa más memoria debido a que esta multiplica constantemente un número por dos haciendo que el uso de memoria crezca exponencialmente.

9.3. Multiplicación de Karatsuba

La multiplicación de karatsuba es un algoritmo divide y conquistar descubierto en 1960 por el ruso Anatoly Karatsuba a la edad de 23 años [Levitin [2014]], este algoritmo es usado para multiplicar números grandes debido a que reemplaza una multiplicación grande en varias pequeñas y sumas de números pequeños. Se basa en que dado dos números a y b se puede representar de la forma

$$a = a_1 a_2 = a_1 10^{n/2} + a_0 \tag{5}$$
$$b = b_1 b_2 = b_1 10^{n/2} + b_0 \tag{6}$$
$$\tag{7}$$

Por ejemplo los números 10 y 23 se pueden escribir como

$$10 = 1 \cdot 10^{n/2} + 0 \tag{8}$$
$$23 = 2 \cdot 10^{n/2} + 3 \tag{9}$$
$$\tag{10}$$

En base a esto se puede crear la siguiente formula

$$a \cdot b = (a_1 10^{n/2} + a_0) \cdot (b_1 10^{n/2} + b_0) \tag{11}$$
$$= c_2 10^n + c_1 10^{n/2} + c_0 \tag{12}$$

Donde
$c_2 = a_1 \cdot b_1$
$c_0 = a_0 \cdot b_0$
$c_1 = (a_1 + a_0) \cdot (b_1 + b_0) - (c_2 + c_0)$

Por ejemplo si quiere saber el producto de $41 \cdot 42$ se tiene:

$$41 = 4 \cdot 10^{n/2} + 1 \tag{13}$$
$$42 = 4 \cdot 10^{n/2} + 2 \tag{14}$$
$$\tag{15}$$

$$c_2 = 4 \cdot 4 \tag{16}$$
$$c_0 = 1 \cdot 2 \tag{17}$$
$$c_1 = (4+1) \cdot (4+2) - (16+2) \tag{18}$$
$$\tag{19}$$

Por lo que $41 \cdot 42$ es

$$41 \cdot 42 = 16 \cdot 10^2 + 12 \cdot 10^1 + 2 = 1722$$

Este algoritmo es notablemente más eficiente que la multiplicación rusa tanto en memoria como en tiempo, su complejidad esta dada por su función de recurrencia

$$M(n) = 3M(n/2), \quad M(1) = 1$$

Usando el teorema maestro Levitin [2014] se obtiene que su complejidad es $\mathcal{O}(n^{1,585})$, donde n es la cantidad de dígitos que tiene el mayor de los números a multiplicar, haciendo que sea más eficiente que la multiplicación rusa.

10. Las pruebas

Las pruebas se realizaron en un pc Con las siguientes características.

Software

- SO: Xubuntu 16.04.1 LTS x86
- Editor: Gedit
- GCC version: 5.4.0
- Mousepad 0.4.0

Hardware

- AMD Turion(tm) X2 Dual-Core Mobile RM-72 2.10GHz
- Memoria (RAM): 4,00 GB(3,75 GB utilizable)
- Adaptador de pantalla: ATI Raedon HD 3200 Graphics

Para las pruebas se tomo en cuenta la cantidad de bits de los números a multiplicar en vez del largo del número debido a que la multiplicación rusa trabaja a nivel de bit, agregando y quitando bits a los números a multiplicar [Levitin [2014]].

Se hicieron dos experimentos, el primero es la multiplicación de dos números con la misma cantidad de bits.

El segundo experimento se probo la propiedad conmutativa del producto, analizando si el algoritmo es más eficiente a la hora de multiplicar de dos números $a \cdot b$ con $a > b$ multiplicando $a \cdot b$ ó $b \cdot a$, siendo el primer número el que se divide constamente por dos y el segundo el que se multiplica constamente por dos.

Como la complejidad asintótica de la multiplicación rusa esta dada solo por el primer número que es divido por dos, mientras menos bits tenga este más eficiente es el algoritmo ya que menos iteraciones va a hacer algoritmo, haciendo que haya menos operaciones de multiplicación y divisón, pero contra parte mayor coste memoria de se tiene debido a que el segundo número, que desde un inicio tiene una mayor cantidad de bits, va a tener un crecimiento exponencial en base a los multiplos de 2.

11. Resultados

En la imagen 1 se muestra el tiempo vs el número de bits de dos números a multiplicar cuando ambos números tienen el mismo largo de bits, cada uno de los números tiene la cantidad de bits que aparece en el eje x. Los números a multiplicar fueron generados aleatoriamente.

En la imagen 2 se muestra el tiempo vs el número de bits de dos números a y b a multiplicar con a $>$ b cuando a número tiene un largo de bits cambiante y b tiene un largo de bits fijo de $2^100 - 1$, el número cambiante tiene el largo de bits que aparece en el eje x. Los números a multiplicar fueron generados aleatoriamente según su número de bits.

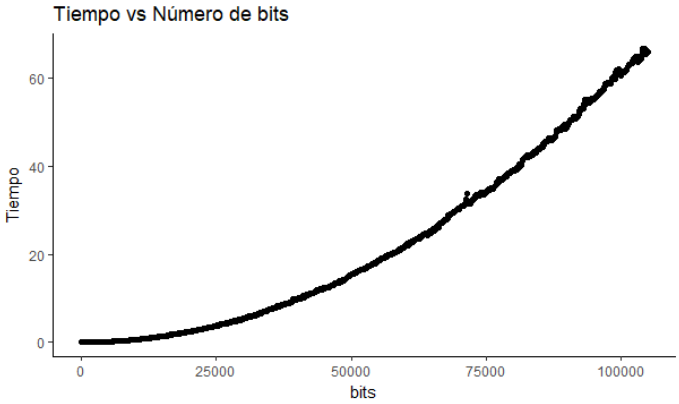

Figura 1. Multiplicación de dos números a y b con la misma cantidad de bits

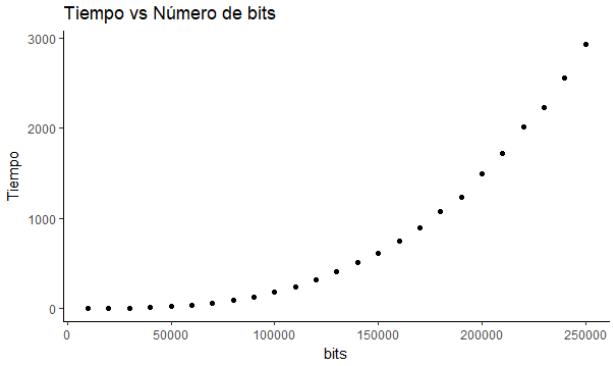

Figura 2. Multiplicación de dos números a·b con a $>$ b, a es fijo y b variante

Los datos con lo que se creo el gráfico 2 se muestran en la tabla 4.

En la imagen 2 se muestra el tiempo vs el número de bits de dos números a y b a multiplicar con a $>$ b cuando b número tiene un largo de bits cambiante y a tiene un largo de bits fijo de $2^{100} - 1$, el número cambiante tiene el largo de bits que aparece en el eje x. Los números a multiplicar fueron generados aleatoriamente según su número de bits.

La tabla con la que se creo el gráfico 3 se muestran en la tabla 5.

12. Conclusiones

Los resultados demuestran que efectivamente la complejidad de la multiplicación rusa es $\mathcal{O}(n \log(n))$, esto se puede ver claramente en la forma de la curva de la figura 1.

Además de que la rapidez de multiplicación depende únicamente de la cantidad de bits del número que va a ser divido en dos, esto se comprueba en la figura 2 versus la figura 3, en la primera el número a dividir con un largo variante de que va desde $2^{10000} - 1$ bits hasta $2^{250000} - 1$, en cambio en la segunda figura se dejo el

Tabla 4
Resultados de multiplicar dos números a·b con a > b, a fijo y b variante

N° bits	Tiempo (seg)
10000	0.266831
20000	1.749454
30000	5.307608
40000	12.194655
50000	23.345639
60000	39.947685
70000	62.885600
80000	92.826035
90000	131.680444
100000	180.657729
110000	244.533478
120000	319.893299
130000	408.241408
140000	515.545736
150000	619.613212
160000	751.622437
170000	899.317050
180000	1080.881010
190000	1238.962152
200000	1493.745582
210000	1717.450705
220000	2014.780954
230000	2233.099756
240000	2563.618059
250000	2937.357255

Tabla 5
Resultados de multiplicar dos números aleatorios de dos números a·b con a > b, a fijo y b variante

N° bits	Tiempo (seg)
10000	0.003576
20000	0.007298
30000	0.010669
40000	0.013326
50000	0.017059
60000	0.020606
70000	0.020369
80000	0.028653
90000	0.031274
100000	0.034317
110000	0.039210
120000	0.041115
130000	0.045586
140000	0.052769
150000	0.055029
160000	0.057380
170000	0.060642
180000	0.061165
190000	0.065194
200000	0.074877
210000	0.081134
220000	0.076351
230000	0.084789
240000	0.081468
250000	0.093469

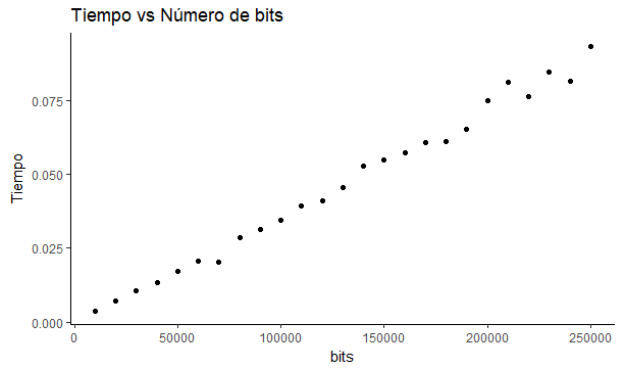

Figura 3. Multiplicación de dos números dos números a·b con a > b, a variante y b fijo

número a dividir como un número $2^{100} - 1$ bits, logrando que en el segundo experimento tuviera un notablemente menor tiempo de ejecución al tener siempre una cantidad fija de 100 iteraciones, por lo que es conveniente a la hora de multiplicar dos números $a \cdot b$ siempre dividir por dos el menor número y multiplicar por dos el mayor, aunque esto implique quedarse sin memoria más rápido debido a que el segundo número puede alcanzar un límite de memoria más temprano.

Por último a pesar de que existen mejores algoritmos para multiplicar números grandes como Karatsuba, la multiplicación rusa se presenta como un método interesante a la hora de multiplicar dos números y sirve mucho cuando se debe considerar un número en su forma binaria, también sirve para trabajar números grandes usando lápiz y papel de manera más rápida y simple ya que reemplaza las típicas tablas de multiplicar con dos operaciones simples como son el multiplicar por dos y el dividir por dos.

En definitiva, lo que en principio puede parecer una operación tan fácil como lo es multiplicar, puede convertirse en algo tan difícil y largo como un trabajo de investigación, y a pesar de que algunas formas de multiplicar pretendan facilitar la multiplicación, la hacen más compleja.

BIBLIOGRAPHY

C. H. Papadimitriou Dasgupta and U. V. Vazirani. *Algorithms*. McGraw-Hill Education (India) Pvt Limited, 2006. ISBN 9780070636613.

ISO/IEC 9899:1999. Programming languages - c. Standard, WG14/N1256, September 2007.

D.E. Knuth. *Art of Computer Programming, Volume 2: Seminumerical Algorithms*. Pearson Education, 2014. ISBN 9780321635761.

A. Levitin. *Introduction to the Design and Analysis of Algorithms: International Edition*. Pearson Education Limited, 2014. ISBN 9781292014111.

MORENO, R. y VEGAS, and J.M. *Una historia de las matemáticas para jóvenes.* Nivola Libros y Ediciones., 2006.

P.K. Rees. *Algebra*. Reverte, Editorial S.A., 1986. ISBN 9789686708073.

S.A. Smith. *Algebra*. Serie AWLI. Alhambra Mexicana, Editorial, S.A. de C.V., 2000. ISBN 9789684443587.

APÉNDICE

APÉNDICE A: MULTIPLICACIÓN RUSA EN C

```c
#define I(a) (a - '0')
#define MAX 100000000

void russianmul(char * a, char * b, char * result) {
  //Declaración  de variables
  int la = strlen(a);
  int lb = strlen(b);
  char * aux = (char * ) malloc(la + 1);
  char * aux2 = (char * ) malloc(lb + MAX + 1);
  char * aux3 = (char * ) malloc((la + lb) + MAX);

  memset(result, '0', la + lb);
  result[0] = '\0';

  //Ciclo Multiplicación rusa
  while (a[0]) { //Mientras a no sea 0
    if (I(a[strlen(a) - 1]) % 2) { //si a es impar
      longsum(b, result, aux3); //Se suma el primer segundo número por el resultado
      strcpy(result, aux3);
    }
    longdiv(a, aux); //se divide el primer número por 2
    strcpy(a, aux);
    longmulti(b, aux2); //Se multiplica el segundo número por dos
    strcpy(b, aux2);
  }

  //Liberación memoria
  free(aux);
  free(aux2);
  free(aux3);
}
```

APÉNDICE B: DIVISIÓN POR 2 DÍGITO A DÍGITO EN C

```c
#define I(a) (a - '0')
#define MAX 100000000

void longdiv(char *a, char *c){
    int la = strlen(a);
    memset(c, '0', la);
    c[la] = '\0';
    int carry = 0;
    int n=0;
    int i;
    for(i=0; i<la; i=i+1){
        n = (carry*10 + I(a[i]))/2;
        carry = (carry*10 + I(a[i])) - 2*n;
        c[i] = n + '0';
    }
    if(c[0] == '0'){ //se elimina el cero que quede adelante
                        memmove(c, c + 1, la);
            c[la] = '\0';
    }

}
```

APÉNDICE C: MULTIPLICACIÓN POR 2 DÍGITO A DÍGITO EN C

```c
#define I(a) (a - '0')
#define MAX 100000000

void longmulti(const char *a, char *c){
        int i = 0, k = 0, carry, n, la;

        la = strlen(a);
        memset(c, '0', la + 1);
        c[la + 2] = '\0';
        for (i = la - 1, k = i + 1; i >= 0; i--){
                n = I(a[i]) * 2 + I(c[k]);
                carry = n / 10;
                c[k] = (n % 10) + '0';
                k--;
                c[k] += carry;
        }

        if(c[0] == '0'){
                memmove(c, c + 1, la);
                c[la] = '\0';
     }
}
```

APÉNDICE D: SUMA CLÁSICA DE DOS ENTEROS EN C

```c
#define I(a) (a - '0')
#define MAX 100000000


void longsum(const char * a,  const char * b, char * c){
  int la = strlen(a);
  int lb = strlen(b);
  int carry = 0;
  int sum;
  memset(c, '0', la+1); //llena c de 0
        //Crea un arreglo aux con el mismo largo que a
    //pero con los valores de b y lleno adelante con ceros
        char *aux= (char *)malloc(la+2);
        memset(aux, '0', la-lb);
    aux[la-lb] = '\0';
        strcat(aux,b);
  while( la >  0) {
      sum = I(a[la-1]) + I(aux[la-1]) + carry; // se suman dos digitos más el carry
      carry = sum/10; //si es mayor a 10 el carry es zero
      c[la] = sum%10 + '0'; // se guarda el primer digito de la suma como resultado
      la--;
  }
        if (carry >0) { //si la última suma es mayor a 10 entonces
                c[0] = carry + '0';
        }
        la = lb = strlen(a);
  if(c[0] == '0'){ //se eliminan los ceros de adelante
      memmove(c, c + 1, la);
       c[la] = '\0';
     }
    free(aux);
}
```