Implementación de multiplicación de números grandes sin pérdida de precisión

Fernanda Téllez Girón, Guillermo Javier Quiroz Martínez, Lizbeth Contreras Figueroa

30 de mayo de 2017

Contenido

Introducción y motivación

En este trabajo se tiene como objetivo implementar el algoritmo de Toom-Cook para realizar multiplicaciones de números grandes sin perder precisión en los cálculos. Este algoritmo de multiplicación es un método para multiplicar dos números enteros que son muy grandes. El nombre del algoritmo se debe a sus autores Andrei Toom[1], que introdujo el nuevo algoritmo con su baja complejidad y Stephen Cook[2], quién limpió la descripción del mismo.

[1] Matemático ruso famoso por sus trabajos en el análisis de algoritmos, los autómatas celulares, y la teoría de probabilidades.

[2] Reconocido científico de la computación que formalizó el concepto de NP-completitud por el cual recibió el Premio Turing en 1982.

Dados 2 números enteros, a y b, Toom-Cook divide los números a y b en k partes más pequeñas, todas de longitud l, y realiza las operaciones sobre las partes. Conforme la k va creciendo va combinando muchas de las suboperaciones de multiplicación, reduciendo así poco a poco la complejidad total del algoritmo. Las suboperaciones de multiplicación ahora sí se pueden realizar recursivamente usando de nuevo el método de multiplicación Toom-Cook, y así sucesivamente. Aunque los términos "Toom-3" y "Toom-Cook" se utilizan en algunas ocasiones incorrectamente como sinónimos, Toom-3 es una única instancia del algoritmo de Toom-Cook, donde $k = 3$.

Multiplicación de alta precisión de números binarios

Dado un entero positivo n y dos enteros $n - bit$ no negativos u y v, el algoritmo forma un producto $2n - bit$, w. Para contener los números largos que se manipulan durante el procedimiento se utilizan cuatro *stacks* auxiliares.

- *Stacks* U, V: Almacenamiento temporal de $U(j)$ y $V(j)$ en el paso $C4$.
- *Stack* C: Número que deben ser multiplicados y códigos de control.
- *Stack* W: Almacenamiento de W.

Estos *stacks* deben contener número binarios o símbolos de control llamados $código - 1$, $código - 2$ y $código - 3$. El algoritmo también construye una tabla auxiliar de números q_k, r_k; esta tabla se mantiene de una manera en la que debe ser almacenada como una lista lineal, donde un único puntero atraviesa la lista (moviéndose hacia atrás y hacia adelante) debe ser utilizado para acceder a la entrada de interés de la tabla actual.

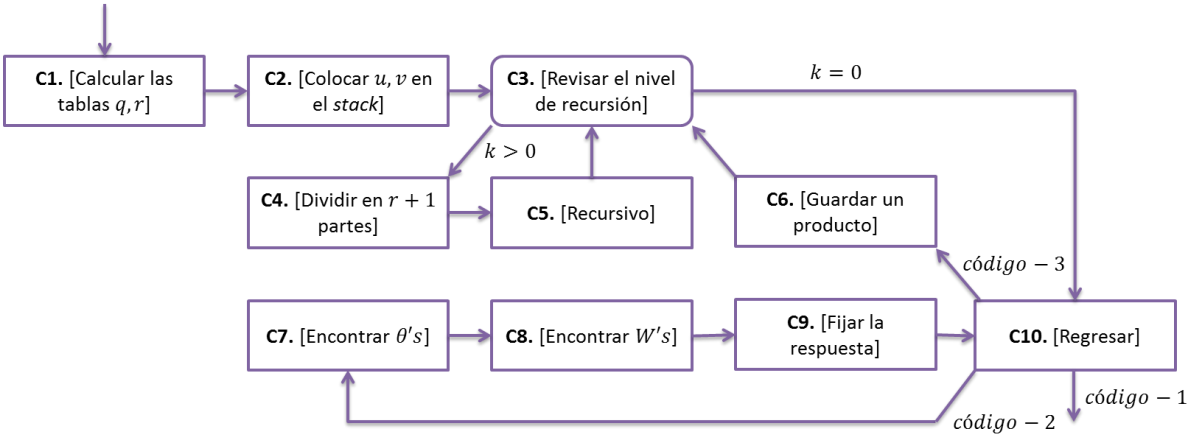

Diagrama 1. Algoritmo Toom-Cook para multiplicación de alta precisión

C1. [Calcular las tablas q, r]. Se establecen vacíos los stacks U, V, C, y W y se fijan:

$$k \leftarrow 1, \quad q_0 \leftarrow q_1 \leftarrow 16, \quad r_0 \leftarrow r_1 \leftarrow 4, \quad Q \leftarrow 4, \quad R \leftarrow 2$$

Ahora, si $q_{k-1} + q_k < n$, se fija

$$k \leftarrow k + 1, \quad Q \leftarrow Q + R, \quad R \leftarrow \lfloor \sqrt{Q} \rfloor, \quad q_k \leftarrow 2^Q, \quad r_k \leftarrow 2^R,$$

Y se repite esta operación hasta que $q_{k-1} + q_k \geq n$

C2. [Colocar u, v en el *stack*]. Se coloca el código-1 en el *stack* C, entonces se pone u y v en el *stack* C como números de exactamente $q_{k-1} + q_k$ bits cada uno.

C3. [Revisar el nivel de recursión]. Se decrece k en 1. Si $k = 0$, el top del *stack* C contiene dos números de 32-bits, u y v; se quitan, y se fija $w \leftarrow uv$ utilizando una rutina para multiplicar número de 32-bits y se va al paso **C10**. Si $k > 0$, se fija $r \leftarrow r_k$, $q \leftarrow q_k$, $p \leftarrow q_{k-1} + q_k$, y se va al paso **C4**.

C4. [Dividir en $r + 1$ partes]. Sea el número en el top del *stack* C considerado como una lista de $r + 1$ números con $q\ bits$ cada uno, $(U_r \dots U_1 U_0)_{2q}$. Para $j = 0, 1, \dots, 2r$, los números $p - bit$.

$$(\dots(U_r j + U_{r-1})j + \cdots + U_1)j + U_0 = U(j)$$

Y sucesivamente se ponen estos valores en el *stack* de U. Después se quitan $U_r \dots U_1 U_0$ del *stack* C. Ahora el top del *stack* C contiene otra lista de $r + 1$ números de $q - bit$, $V_r \dots V_1 V_0$ del *stack* C.

C5. [Recursivo]. Se ponen los siguientes ítems en el *stack* C, al mismo tiempo se vacían los *stacks* U y V:

$$code - 2, V(2r), U(2r), code - 3, V(2r-1), U(2r-1), \dots,$$
$$code - 3, V(1), U(1), code - 3, V(0), U(0)$$

Se regresa al paso **C3**.

C6. [Guardar un producto]. Se pone $w$en el *stack* W. (Este número w contiene $2(q_k + q_{k-1})bits$. Se regresa al paso **C3**.

C7. [Encontrar $\theta's$]. Se fija $r \leftarrow r_k$, $q \leftarrow q_k$, $p \leftarrow q_{k-1} + q_k$. Ahora para $j = 1, 2, 3, \dots, 2r$, se realiza el siguiente loop: para $t = 2r, 2r-1, 2r-2, \dots, j$, se fija $W(t) \leftarrow (W(t) - W(t-1))/j$.

C8. [Encontrar $W's$]. Para $j = 2r-1, 2r-2, \dots, 1$, se realiza el siguiente loop: para $t = j, j+1, \dots, 2r-1$, se fija $W(t) \leftarrow W(t) - jW(t+1)$.

C9. [Fijar la respuesta]. Se fija w al entero de $2(q_k + q_{k+1})$ bits

$$\Big(\dots\big(W(2r)2^q + W(2r-1)\big)2^q + \cdots + W(1)\Big)2^q + W(0)$$

Se quita $W(2r), \dots, W(0)$ del *stack* W.

C10. [Regresar]. Se fija $k \leftarrow k+1$. Se quita el top del *stack* C. Si es $código - 3$, se va al paso **C6.** Si es $código - 2$, se pone w en el *stack* W y se va al paso **C7**. Y si es $código - 1$, se termina el algoritmo (w es la respuesta) ■

El Método Toom-3

El algoritmo Toom Cook es el método avanzado para dividir los números en partes. De esta manera, el producto se reduce a $2 * (n) - 1$ multiplicaciones. Donde, en este caso n es igual a 3.

Esto quiere decir que los operandos considerados sean divididos en 3 piezas de igual longitud. Las partes se escriben en términos de polinomios

$$X(t) = (X_2)t^2 + X_1(t) + X_0$$

$$Y(t) = (Y_2)t^2 + Y_1(t) + Y_0$$

Se elige la base $B = b$, de tal manera que el número de dígitos de ambos x y m en la base B sea como máximo k (por ejemplo, 3 en Toom-3). Una opción para i está dada por:

$$i = max\lfloor \lfloor logbm \rfloor / k \rfloor, l\lfloor \lfloor logbn \rfloor / k \rfloor + 1$$

En este ejemplo se toma el valor:

$$B = b^2 = 10^8$$

Entonces se separa x e y en los dígitos de la base B x, y: Estas 2 ecuaciones se multiplican para formar $w(t) = x(t) * y(t)$

$$W(t) = w_4 * t^4 + w_3 * t^3 + w_2 * t^2 + w_1 * t + w_0$$

Es importante señalar que la $w(t)$ final se calcula a través del valor de t, aunque la etapa final va a ser la adición. $X(t)$ e $Y(t)$ se calculan y multiplican eligiendo un conjunto de puntos, formando $w(t)$.

Ahora, sean los siguientes puntos $(0, 1, 2, -1, inf)$, tenemos:

$t = 0 \rightarrow x_0 * y_0$,

$t = 1 \rightarrow (x_2 + x_1 + x_0) * (y_2 + y_1 + y_0)$,

$t = -1 \rightarrow (x_2 - x_1 + x_0) * (y_2 - y_1 + y_0)$,

$t = \rightarrow 2(4 * x_2 + 2 * x_1 + x_0) * (4 * y_2 + 2 * y_1 + y_0)$,

$t = inf \rightarrow x_2 * y_2$,

Entonces el valor de esas combinaciones se calcula:

$W(0) = w_0$

$W(1) = w_4 + w_3 + w_2 + w_1 + w_0$

$W(-1) = w_4 - w_3 + w_2 - w_1 + w_0$

$W(2) = 16 * w_4 + 8 * w_3 + 4 * w_2 + 2 * w_1 + w_0$

$W(inf) = w4$

La complejidad es esencial para medir cuánto tiempo un algoritmo está tomando para funcionar. La complejidad de un problema se define usando la notación grande de O. Esto ayuda a elegir el mejor algoritmo que se adapte a nuestro problema.

La tarea principal es ejecutar el algoritmo lo más rápido posible para obtener resultados eficientes; el método clásico de adición y multiplicación toma $O(n2)$ para la multiplicación. Toom-3 divide el operando en $n/2$ partes, estas dos ecuaciones se multiplican para formar $w(t) = x(t) * y(t)$

$$W(t) = w_4 * t^4 + w_3 * t^3 + w_2 * t^2 + w_1 * \mathrm{t} + w_0$$

El algoritmo implementado toma 5 multiplicaciones para 3 divisiones.

$$Z_0 = x_0 * y_0$$

$$Z_1 = (x_0 + x_1 + x_2)\,(y_0 + y_1 + y_2)$$

$$Z_2 = (x_0 - x_1 + x_2)\,(y_0 - y_1 + y_2)$$

$$Z_3 = (x_0 + 2 * x_1 + 4 * x_2)\,(y_0 + 2 * y_1 + 4 * y_2)$$

$$Z_4 = (x_0 - 2 * x_1 + 4 * x_2)\,(y_0 - 2 * y_1 + 4 * y_2)$$

Por lo tanto, vemos que la división del método k requiere $2\ (k) - 1$ multiplicaciones.

Toom-3 reduce 9 multiplicaciones a 5, y corre en $O(n^{log(5)/log(3)})$, aproximadamente $O(n^{1.465})$

Ejemplo de implementación

Para aplicar el método Toom-3 se consideran 2 números para la operación:

$$\mathbf{831275469}\ por\ \mathbf{897512436}$$

a) Dividimos cada uno de los números en 3 miembros de longitud de tres dígitos:

831	275	469

897	512	436

b) Se escriben en forma de polinomio

$a_2 = 831$	$a_1 = 275$	$a_0 = 469$

$b_2 = 897$	$b_1 = 512$	$b_0 = 436$

$$P(x) = a_2 * x^2 + a_1 * x + a_0$$

$$Q(x) = b_2 * x^2 + b_1 * x + b_0$$

$$p(x) = 831x^2 + 275x + 469$$

$$q(x) = 897x^2 + 512x + 436$$

c) Se escribe: $p(x)q(x) = r(x)$

$$(831x^2 + 275x + 469) * (897x^2 + 512x + 436) =$$

$$ax^4 + bx^3 + cx^3 + dx + e = r(x)$$

d) Se sustituyen los valores de x por el conjunto de ecuaciones. Esto permite que el conjunto de puntos a ser sustituidos sea $-2, -1, 0, 1, 2$

- **Para el caso de $\mathbf{0} \rightarrow p\,(0)\,q\,(0)\;=\;r\,(0)$**

$$831(0)^2 + 275(0) + 469) * (897(0)^2 + 512(0) + 436) \;=$$

$$a(0)^4 + b(0)^3 + c(0)^2 + dx + e \;=\; r(x)$$

Por lo tanto, $e = 204484$.

- **Para el caso de $\mathbf{1} \rightarrow p\,(1)\,q\,(1)\;=\;r\,(1)$**

$$(831(1)^2 + 275(1) + 469) * (897(1)^2 + 512(1) + 436)$$
$$= \; a(1)^4 + b(1)^3 + c(1)^2 + d(1) + e \;=\; r(x)$$

$$a + b + c + d + \; e \;=\; 2905875$$

- **Para el caso de $-\mathbf{1} \rightarrow p(-1)q(-1)\;=\;r(-1)$**

$$(831(-1)^2 + 275(-1) + 469) * (897(-1)^2 + 512(-1) + 436)$$
$$= \; a(-1)^4 + b(-1)^3 + c(-1)^2 + d(-1) + e \;=\; r(x)$$

$$a - b + c - d + e = \; 841525$$

- **Para el caso de $\mathbf{2} \rightarrow p(2)q(2)\;=\;r(2)$**

$$(831(2)^2 + 275(2) + 469) * (897(2)^2 + 512(2) + 436)$$
$$= \; a(2)^4 + b(2)^3 + c(2)^2 + d(2) + e \;=\; r(x)$$

$$16a + 8b + 4c + 2d + e = \; 21923464$$

- **Para el caso de $-\mathbf{2} \rightarrow p(-2)q(-2)\;=\;r(-2)$**

$$(831(-2)^2 + 275(-2) + 469) * (897(-2)^2 + 512(-2) + 436)$$
$$= \; a(-2)^4 + b(-2)^3 + c(-2)^2 + d(-2) + e \;=\; r(x)$$

$$16a - 8b + 4c - 2d + e = 9729000$$

e) Se forma el siguiente sistema:

$$e \;=\; 204484$$

$$a + \; b + \; c + \; d + e = \; 2905875$$

$$a - b + \; c - d + e = \; 841525$$

$$16a + 8b + 4c + 2d + e \;=\; 21923464$$

$$16a - 8b + 4c - 2d + e = \; 972900$$

Matricialmente:

$$\begin{pmatrix} 0 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 & 1 \\ 16 & 8 & 4 & 2 & 1 \\ 16 & -8 & 4 & -2 & 1 \end{pmatrix} \begin{pmatrix} a \\ b \\ c \\ d \\ e \end{pmatrix} = \begin{pmatrix} 204488 \\ 2905875 \\ 841525 \\ 21923464 \\ 9729000 \end{pmatrix}$$

f) Se resuelve el Sistema y tenemos:

$$a = 745407$$

$$b = 672147$$

$$c = 923809$$

$$d = 360028$$

$$e = 244484$$

Se obtiene el producto final mediante la suma de los números del inciso f.

7	4	5	46	07	72	19	42	73	83	06	90	02	20	84	4	8	4
7	**4**	**6**	**0**	**8**	**0**	**0**	**7**	**1**	**1**	**6**	**9**	**2**	**3**	**2**	**4**	**8**	**4**

Programación del algoritmo

Con la idea de implementar este algoritmo se tomaron las siguientes condiciones de diseño:

a) Construir un *framework* para la aplicación inicial de la versión Toom-Cook 3 que facilite la implementación de otras versiones del algoritmo y sus variantes.
b) Extender el algoritmo a potenciales aplicaciones intensivas de la multiplicación, por ejemplo, en operaciones matriciales, con la ayuda de tecnologías multiproceso.

Con estas ideas en mente, se decidió utilizar un enfoque de objetos en *lenguaje C* con la ayuda de *pthreads*. Cabe destacar que *C* no es un lenguaje orientado a objetos, sin embargo, tomando las ideas centrales del diseño orientado a objetos se pueden tomar

ventaja de ello mediante la abstracción un número entero muy grande e implementarlo con las siguientes ideas en mente:

- ***Encapsulado.*** Solo el mismo objeto puede manipular sus estructuras de datos y sus métodos públicos para manipularlas están bien definidos. Esto se logra en el código con la ayuda de funciones y la construcción *struct* de *C*. Aunque no se enforza pues el lenguaje no lo permite.
- ***Herencia.*** Se pueden construir nuevos objetos a partir de los ya existentes. Esto se logra con agregación pues se pueden construir en *C* funciones parte de nuevos objetos que utilicen objetos ya existentes como parte de sí mismos. Es importante señalar que no se obtiene resolución dinámica de métodos por que el lenguaje no lo permite. Sin embargo una buena aproximación.

Los objetos seleccionados son un número (con la *struct numerote*) y una matriz (con *numeroteMatrix*), con interface como sigue:

```
struct numerote
{
        char* buffer;
        int   status;
        int   num_boxes;
        int   index[NEBOXES];
        long  values[NEBOXES];
        int   inUse[NEBOXES];
};
/*****************************************************************************
            numerote operations
*****************************************************************************/
int   numeroteNew         (pNumerote pNumero, char newValue[50]);
int   numeroteDestroy     (pNumerote pNumero);
int   numeroteMultiply    (pNumerote pA, pNumerote pB, int resultados[NEDEFAULT_BUFER_SIZE]);
int   numeroteAdd         (pNumerote pA, pNumerote pB, pNumerote pC);
int   numerotePrint       (pNumerote pA);
```

Y para matriz:

```
struct numeroteMatrix
{
        char* buffer;
        int   m;
        int   n;
        int   status;
        numerote matrix[MAX_MATRIX_SIZE];
};

/*****************************************************************************
            numerote matrix operations
*****************************************************************************/
int    numeroteMatrixNew                (pNumeroteMatrix pMatrix, char** newValue, int m, int n);
int    numeroteMatrixDestroy            (pNumeroteMatrix pMatrix);
int    numeroteMatrixPrint              (pNumeroteMatrix pMatrix);
int    numeroteMatrixMultiply           (pNumeroteMatrix pA, pNumeroteMatrix pB,
pNumeroteMatrix pC);
int    numeroteMatrixMultiplyNoThreads  (pNumeroteMatrix pA, pNumeroteMatrix pB,
pNumeroteMatrix pC);
int    numeroteMatrixAdd                (pNumeroteMatrix pA, pNumeroteMatrix pB,
pNumeroteMatrix pC);
int    numeroteMatrixMultiplyPoint      (pNumeroteMatrix pA, pNumeroteMatrix pB, int row, int col,
int
                                          results[NEDEFAULT_BUFER_SIZE] );
void   *multiplyMatrizRow               ( void *commArea);
```

El código implementa siguiendo al pie de la letra el algoritmo descrito anteriormente. Se usa una matriz fija de 3 *boxes* y longitud de ítem 6. Esta estructuración tiene ventajas importantes pues el código admite, con algo de ingenio en la programación, expresiones del siguiente tipo para multiplicar dos números:

```
#include "toom_cook.h"
numerote xx;
numeroteNew(&xx, "123456789012345678");
numerote yy;
numeroteNew(&yy, "78978797789987890");
numerotePrint(&yy);
numeroteMultiply(&xx,&yy, resultados);
```

o expresiones como la siguiente para multiplicar dos matrices:

```
#include "toom_cook.h"
char *  z[] = {"1234567890123456", "1234567890123456", "8765432109876543",
                "8765432109876543", "8765432109876543", "1111111111111111",
                 "2222222222222222", "3333333333333333", "4444444444444444"};
numeroteMatrix x;
numeroteMatrixNew(&x,z,3,2);
numeroteMatrix y;
numeroteMatrixNew(&y,z,2,3);
numeroteMatrixMultiply(&x,&y,&w);
numeroteMatrixPrint(&w);
```

En lo que se refiere a concurrencia y utilización de *pthreads*, esta organización de código permite también algunas ventajas:

- Se sabe que la biblioteca de *C* presenta afinidades de concurrencia que la hacen inapropiada para la escritura de código multihilo. Es importante evitarlas en la medida de lo posible.
- Si se minimiza la utilización de la biblioteca con substitutos para algunas operaciones que se programen adecuadamente (sin la utilización de estructuras de datos compartidos y globales), se puede simplificar el uso de *locking* para el proceso.

El código utilizado en este trabajo, utiliza al mínimo funciones riesgosas (con la excepción de las funciones *sprintf* y *powl*). Durante el proceso de prueba no se presentaron problemas de concurrencia, pero si se llegaran a necesitar el código está marcado donde habría que activar y desactivar *mutexes*.

La función de multiplicación de matrices fue pensada para sacar ventaja de los hilos. Dado que el algoritmo seguido utiliza multiplicación renglón por columna y que las matrices multiplicándose solo requieren acceso de lectura y que un punto de la matriz resultado no es tocado más que por un hilo, no es necesario hacer control de acceso (con la excepción que se menciona en el párrafo anterior). Se inician tantos hilos como renglones tenga la primer matriz.

El proceso de prueba es todo un reto. En pocos lugares de la literatura se encontraron referencias a este tema con este objetivo. Comenzamos con multiplicaciones manuales, con el correspondiente consumo de tiempo. Afortunadamente, a la mitad del proyecto, se encontró una clase en *Java (BIgInteger)* que auxilió en el proceso. También se utilizó programación en *R* para validar los resultados de este código.

Resultados y conclusiones

Los resultados fueron muy interesantes, a continuación, se presenta la salida para un ejemplo, los resultados has sido verificados extensivamente:

```
Printing matrix x
  Dimensions: (3,2)
  1234567890123456 1234567890123456
  8765432109876543 8765432109876543
  8765432109876543 1111111111111111
----------------------------------------------------------------------------------------
Printing matrix y
  Dimensions: (2,3)
  1234567890123456 1234567890123456 8765432109876543
  8765432109876543 8765432109876543 1111111111111111
----------------------------------------------------------------------------------------
Voy a multiplicar xy= w
Printing matrix
  Dimensions: (3,3)
  12345678901234558765432109876544 12345678901234558765432109876544 12193263126047850234720311812224
  87654321098765421234567890123457 87654321098765421234567890123457 86572169083828679888736467200122
  20560890036884612731290954061881 20560890036884612731290954061881 78067367974089311850327689285170
```

Al algoritmo Toom-Cook presenta una alternativa interesante para multiplicar grandes números sin pérdida de precisión su complejidad es del orden $O(n^{1.435})$ contra $O(n^2)$ que es el la complejidad de la mutiplicacion directa. La implementacion con pthreads es viable y proporciona buenos resultados.

Referencias

---. **Singular Value Decomposition (SVD) tutorial**. Recuperado en mayo de 2017, de: http://web.mit.edu/be.400/www/SVD/Singular_Value_Decomposition.htm

---. **Toom 3-Way Multiplication**. Recuperado en mayo de 2017, de: https://gmplib.org/manual/Toom-3_002dWay-Multiplication.html

---. **Toom–Cook multiplication**. Recuperado en mayo de 2017, de: https://en.m.wikipedia.org/wiki/Toom%E2%80%93Cook_multiplication

Bodrato, M., & Zanoni, A. (2006). **What About Toom-Cook Matrices Optimality**. Recuperado en mayo de 2017, de: http://www.bodrato.it/papers/WhatAboutToomCookMatricesOptimality.pdf

Parhi, K. **Fast Convolution**. Recuperado en mayo de 2017, de: https://www.dropbox.com/s/8flmijftnfk6her/chap8.pdf?dl=0

Schieck, J. **Solving of linear Equations using SVD**. Recuperado en mayo de 2017, de: https://www.mpp.mpg.de/%7Eschieck/svd.pdf

Yedugani, S. **Toom Cook**. Recuperado en mayo de 2017, de: https://www.dropbox.com/s/bwziz0hnv78pgp6/ToomCook.pdf?dl=0

Zanoni, A. **Iterative Toom-Cook Methods For Very Unbalanced Long Integers Multiplication**. Recuperado en mayo de 2017, de: https://www.dropbox.com/s/catd6x49sdc451n/iterative_Toom_Cook.pdf?dl=0