Universidad Autónoma Metropolitana Unidad Azcapotzalco
Dvisión de Ciencias Básicas e Ingeniería
Licenciatura en Ingeniería en Computación
Proyecto tecnológico

Implementación y comparación de algoritmos para la multiplicación de enteros grandes

Juan Manuel García Reyes
209301390

Asesor: Dr. Víctor Cuauhtémoc García Hernández

Trimestre 2014 Invierno
10 de abril de 2014

Yo, Víctor Cuauhtemoc García Hernández, declaro que aprobé el contenido del presente Reporte de Proyecto de Integración y doy mi autorización para su publicación en la Biblioteca Digital, así como en el Repositorio Institucional de UAM Azcapotzalco.

Yo, Juan Manuel García Reyes, doy mi autorización a la Coordinación de Servicios de Información de la Universidad Autónoma Metropolitana, Unidad Azcapotzalco, para publicar el presente documento en la Biblioteca Digital, así como en el Repositorio Institucional de UAM Azcapotzalco.

Resumen

En el presente trabajo se aborda el problema computacional de la multiplicación de enteros grandes. Se presenta un estudio teórico y la implementación del algoritmo estándar de multiplicación, el método de Karatsuba y el uso del la transformada rápida de Fourier.

Como primer punto se hace una revisión teórica del algoritmo de Karatsuba. También se presenta el método de la transformada rápida de Fourier y una de sus aplicaciones en el problema de la multiplicación de enteros.

Se presentan los pseudocódigos y sus respectivos análisis de complejidad. Posteriormente se implementan los algoritmos y sus resultados de tiempo de ejecución presentados en gráficas.

Se finaliza con la descripción de los resultados remarcando la consistencia con lo que teóricamente se esperaba.

Índice general

1. Introducción, antecedentes y justificación **3**

2. Objetivos **5**

3. Marco teórico **6**
3.1. La multiplicación estandar 6
3.2. El método de Karatsuba, divide y vencerás 7
3.3. La transformada rápida de Fourier (FFT) 8
3.4. La transformada discreta de Fourier DFT 10
3.5. FFT en la multiplicación de entros grandes 11

4. Desarrollo del proyecto **13**
4.1. Resultados 16

5. Análisis y discusión de resultados **18**

6. Conclusiones **20**

Bibliografía **21**

7. Apéndice **22**

Capítulo 1

Introducción, antecedentes y justificación

Dados dos enteros a primera vista multiplicarlos no representa mayor problema. Si proponemos números cada vez más grandes es válido argumentar que el producto se obtiene con la misma idea sin importar el número de sus cifras. Sin embargo, esta respuesta refleja que el concepto de multiplicación de enteros (sin importar su tamaño) lo tenemos asociado al *algoritmo natural* que se aprende en la infancia. Aunque el método es teóricamente cierto, desde el punto de vista de la computación el algoritmo puede convertirse en un problema complejo cuando se habla de números suficientemente grandes. Por ejemplo, si tenemos que multiplicar dos enteros de 1000 cifras, el algoritmo común requiere más de 1000000 operaciones. En general, multiplicar en la forma común dos enteros de n cifras requiere el costo de más de n^2 operaciones [6]. Si asumimos que el número de las cifras n puede crecer de manera indefinida, el costo computacional del cálculo de los productos es alto.

El problema de la multiplicación de enteros consiste en el diseño de algoritmos que sean capaces de reducir tanto como sea posible el orden del número de operaciones necesarias para obtener el producto de dos enteros de n cifras, pensando que n es un parámetro que tiende a crecer de manera indefinida. La vertiginosa evolución de las computadoras del siglo XX trajo consigo la necesidad de construir algoritmos para multiplicar enteros tan rápido como fuese posible. El tiempo de ejecución de cualquier proceso de cómputo depende de que tan rápida sea la computadora para sumar y multiplicar.

El reto consiste en reducir el orden n^2 que demanda la multiplicación común. Aunque se trata de un problema de caracter elemental, el diseño de algoritmos que reducen el número de operaciones ha requerido de elementos teóricos profundos. El método de Karatsuba *Divide y Vencerás* conjunto con la *Transformada Rápida de Fourier* (FFT, por sus siglas en inglés) [7] son esencialmente las técnicas que permiten reducir el orden n^2. Se puede decir que, a grandes razgos y sin hacer énfasis en sus variantes, ambos métodos y las ideas que subyacen en ellos, son las herramientas base que dieron lugar al estudio moderno del problema de la multiplicación de enteros.

Se debe remarcar que la Transformada Rápida de Fourier ha encontrado numerosas aplicaciones en la ingeniería. Principalmente en el procesamiento de señales y sus múltiples aplicaciones. G. Strang describió a la transformada rápida como el algoritmo númerico más importante de todos los tiempos [9].

Capítulo 2

Objetivos

El objetivo de este trabajo es presentar un estudio esencial y autocontenido del problema de la multiplicación de enteros. De manera más precisa, sobre el márco teórico y la implementación de los método de Karatsuba y la Transfomada Rápida de Fourier en la versión presentada por . Para complementar, además se implementará el algoritmo de multiplicación estandar. Además se realizará una comparación de los tres métodos se obtendran concluisones sobre las ventajas y desventajas en base al costo computacional.

Capítulo 3

Marco teórico

Esta sección está dedicada a la justificación teórica de los métodos de Karatsuba y el de la transformada rápida de Fourier. También presentamos el análisis de los costos de cálculo antes de implementar los tres algoritmos.

Entenderemos como *una operación básica* en base $K \geq 2$ a la suma o producto de dos enteros a, b tales que $0 \leq a, b \leq K - 1$.

En adelante asumiremos que el parámetro n, que representa al número de dígitos de un entero x, tiende a tomar valores indefinidamente grandes.

Introducimos la siguiente notación. Sean f y g dos funciones con dominio en los reales. La notación $f \ll g$, o de manera equivalente $f(t) = \mathcal{O}(g(t))$, significa que existe una constante $C > 0$, que no depende de t, tal que $|f(x)| \leq Cg(t)$, para todo t suficientemente grande. Se dice que f es menor que el orden de g si $f \ll g$. Además, si se tiene que $g \ll f \ll g$, se dice que f y g tienen el mismo orden.

Dados dos enteros x y y. Podemos suponer que x y y tienen el mismo número de dígitos, el caso contrario se aborda exactamente con las mismas ideas.

3.1. La multiplicación estandar

Como se comentó en la introducción, entenderemos al método estandar como el algoritmo con que se enseña a multiplicar en la educación básica.

Si x y y son enteros con n dígitos, se puede verificar sin dificultad que el número de operaciones necesarias para obtener el producto no excede a $6n^2$.

Si denotamos por $T(n)$ al número de operaciones necesarias para obtener el producto $x \cdot y$, el algoritmo estandar establece que $T(n) \ll n^2$.

3.2. El método de Karatsuba, divide y vencerás

El celebre matemático Andrey Kolmogorov conjeturó que $T(n)$ era exactamente del orden n^2. Dentro de un seminario liderado por Kolmogorov, el estudiante A. Karatsuba desmintió tal conjetura. Construyó un algoritmo capaz de reducir drásticamente el orden de n^2, estableció la nueva estimación

$$T(n) \ll n^{\log_2 3}, \qquad \text{donde} \qquad \log_2 3 = 1{,}5849\ldots . \tag{3.1}$$

Veanse [2] y [3].

Efectivamente, verifiquemos que la estimación (3.1) tiene lugar. Sean x, y enteros dados. Para dar agilidad al análisis de complejidad podemos asumir que x, y estan escritos en notación binaria. Notemos que el costo de cambiar la base decimal a binaria no es mayor que el orden $\mathcal{O}(n)$. Por lo tanto, para cierto entero m tenemos la siguiente representación de $2m$-bit:

$$x = x_0 + x_1 2 + \ldots + x_{2m-1}2^{2m-1}, \qquad y = y_0 + y_1 2 + \ldots + y_{2m-1}2^{2m-1},$$

donde

$$x_i, y_i \in \{0, 1\}, \qquad 0 \leq i \leq 2m-1.$$

Podemos escribir

$$x = 2^m X_1 + X_0, \qquad y = 2^m Y_1 + Y_0,$$

con

$$\begin{aligned} X_1 &= x_m + x_{m+1}2 + \ldots + x_{2m-1}2^{m-1}, \quad X_0 = x_0 + x_1 2 + \ldots + x_{m-1}2^{m-1}, \\ Y_1 &= y_m + y_{m+1}2 + \ldots + y_{2m-1}2^{m-1}, \quad Y_0 = y_0 + y_1 2 + \ldots + y_{m-1}2^{m-1}. \end{aligned}$$

De esta forma

$$x \cdot y = 2^m \left(2^m + 1\right) X_1 Y_1 + 2^m \left(X_1 - X_0\right)\left(Y_0 - Y_1\right) + (2^m + 1) X_0 Y_0.$$

Observe que esta forma de escribir el producto demanda tres multiplicaciones de enteros de m-bits, a decir: X_1Y_1, $(X_1 - X_0)(Y_0 - Y_1)$ y X_0Y_0. Posteriormente se requiere de un número fijo de sumas de números de a lo más $4m$-bits. Por lo tanto, para cierta constante $c > 0$ se tiene

$$T(2m) \leq 3T(m) + cm.$$

Ahora, para facilitar el cálculo se hace un proceso recursivo cuando el argumento es una potencia de 2 y se obtiene

$$T(2^k) \leq c(3^k - 2^k).$$

Finalmente, para cualquier m suficientemente grande se tiene

$$T(m) \leq T(2^{[\log_2 m]+1}) \leq c(3^{[\log_2 m]+1} - 2^{[\log_2 m]+1}) \leq 3c3^{\log_2 m} \leq 3cm^{\log_2 3}.$$

De esta forma hemos comprobado que el tiempo de computo del producto de dos enteros de n cifras satisface la ecuación (3.1).

3.3. La transformada rápida de Fourier (FFT)

Presentaremos a la FFT en una versión que conviene a nuestros objetivos.

Antes de retomar el problema de la multiplicación de enteros consideremos el problema de la multiplicación de polinomios con coeficientes enteros. Sean f y g dos polinomios de grado $n \geq 1$ con coeficientes enteros, digamos

$$f(x) = \sum_{i=0}^{n-1} a_i x^i, \qquad g(x) = \sum_{i=0}^{n-1} b_i x^i, \quad a_i, b_i \in \mathbb{Z}.$$

El producto de los polinomios se calcula de manera natural como

$$f(x)g(x) = \sum_{i=0}^{2n-2} c_i x^i, \quad c_i = \sum_{j=0}^{i} a_j b_{i-j}, \quad \text{con } \ 0 \leq i \leq 2n-2.$$

Observe que el cálculo de los coeficientes c_i del polinomio producto requiere de un número de operaciones del orden $\mathcal{O}(n^2)$.

El algoritmo de la Transformada Rápida de Fourier nos permite encontrar el polinomio producto hasta el sorprendente tiempo de ejecución del orden $n \log n$. Para describir de una manera comoda al algoritmo es necesario establecer las siguientes convenciones:

- Podemos entender a los polinomos como vectores. Al polinomio producto le corresponde un vector de $2n-1$ entradas pero por cuestiones de simetría a $p \cdot q$ se le asociará un vector de dimensión $2n$

$$p \cdot q \sim \mathbf{c} = (c_0, c_1, \ldots c_{2n-2}, c_{2n-1}), \quad \text{con} \quad c_{2n-1} = 0.$$

- De manera análoga, por cuestiones de simetría, a los vectores p y q les asociaremos vectores de dimensión $2n$, con al menos la mitad de sus entradas iguales a cero:

$$\begin{aligned} p \sim \mathbf{a} &= (a_0, a_1, \ldots, a_{2n-2}, a_{2n-1}), \quad \text{con} \quad a_n = a_{n+1} = \ldots = a_{2n-1} = 0, \\ q \sim \mathbf{b} &= (b_0, b_1, \ldots, b_{2n-2}, b_{2n-1}), \quad \text{con} \quad b_n = b_{n+1} = \ldots = b_{2n-1} = 0. \end{aligned}$$

La idea que ahora se plantea es obtener al vector $\mathbf{c}$ en una manera alternativa. Como punto de partida observe que dado un número ξ, posiblemente un número complejo, la evaluación del polinomio producto en ξ es de hecho el producto de las respectivas evaluaciones, es decir

$$(p \cdot q)(\omega) = p(\omega) \cdot q(\omega).$$

Evaluemos a los polinomios p y q en las raices 2-enésimas de la unidad $\xi^0, \xi^1, \xi^2, \ldots, \xi^{2n-1}$ (véase el Apéndice)

$$u_i = p(\xi^i), \qquad v_i = q(\xi^i), \qquad 0 \leq i \leq 2n-1.$$

De esta manera podemos construir otro par de vectores que corresponden a las imagenes

$$\mathbf{u} = (u_0, \ldots, u_{2n-1}), \qquad \mathbf{v} = (v_0, \ldots, v_{2n-1}).$$

Por otra parte, con esta información es posible conocer las imagenes $w_i = (pq)(\xi^i)$ sin necesidad de evaluar al polinomio producto. Es claro que $w_i = u_i \cdot v_i$, para cada $0 \leq i \leq 2n-1$, y de esta forma tenemos

$$\mathbf{w} = (w_0, \ldots, w_{2n-1}).$$

El siguiente paso es devolver al vector desconocido $\mathbf{c}$ a partir del vector $\mathbf{w}$ que se obtiene de una manera elemental. Para tal fin es necesario introducir el concepto de la *Transformada discreta de Fourier* (DFT por sus siglas en inglés).

3.4. La transformada discreta de Fourier DFT

De manera formal, la transformada discreta de Fourier del polinomio p, representado mediante el vector $\mathbf{a}$, está definida por los valores del vector $\mathbf{u}$ definidos por la evaluación de p en las raices 2-enésimas de la unidad

$$u_j = p(\xi^j) = \sum_{i=0}^{2n-1} a_i \xi^{ji}.$$

En la notación matricial también se puede escribir como

$$\mathbf{u} = \mathbf{F} \cdot \mathbf{a},$$

Donde $\mathbf{F}$ es la matriz $\{F_{i,j}\}$ simétrica de dimensión $(2n)^2$ y cuyas entradas son

$$F_{i,j} = \xi^{ij}.$$

La matriz $\mathbf{F}$ es conocida como la *matriz de la transformada discreta* y es quien establece la correspondencia el vector de los coeficientes $\mathbf{a}$ y el vector de las imagenes $\mathbf{u}$. De esta forma tenemos

$$\mathbf{u} = \mathbf{F} \cdot \mathbf{a}, \qquad \mathbf{v} = \mathbf{F} \cdot \mathbf{b}.$$

La construcción de $\mathbf{F}$, mediante potencias ordenadas de ξ, hace que la matriz de transformada discreta tenga propiedades interesantes y en particular tenga inversa $\mathbf{F^{-1}}$ que puede calcularse de manera sencilla (véase [7]):

$${F_{i,j}}^{-1} = \frac{1}{2n} \xi^{-ij}.$$

De esta forma, dado $\mathbf{u}$, tenemos $\mathbf{a} = \mathbf{F^{-1}} \cdot \mathbf{u}$. De manera más explícita:

$$a_i = \sum_{j=0}^{2n-1} \frac{1}{2n} u_j \xi^{-ji}.$$

Finalmente, si partimos de polinomios p, q expresados como vectores $\mathbf{a}, \mathbf{b}$, el vector $\mathbf{c}$ correspondiente a los coeficientes del producto $p \cdot q$ es

$$\mathbf{c} = \mathbf{F^{-1}}(\mathbf{w}),$$

donde $\mathbf{w} = (u_0 v_0, \ldots, u_{2n-1} v_{2n-1})$.

3.5. FFT en la multiplicación de entros grandes

En esta sección se extiende la idea de Karatsuba para expander al entero en una forma particular. Más adelante se utilizará la FFT para multiplicar de manera óptima el las expansiones elegidas.

El objetivo es extender el resultado de Karatsuba a la siguiente forma. El tiempo de ejecución de la multiplicación de enteros satisface

$$T((r+1)n) \leq (2r+1)T(n) + cn, \tag{3.2}$$

para cualquier entero $r \geq 1$.

Para tener mayor control sobre esta descripción teórica, supongamos que x y y son enteros grandes con el mismo número de cifras y además tienen la siguiente expresión binaria de $(r+1)n$-bits

$$x = x_0 + x_1 2 + \ldots + x_{(r+1)n-1} 2^{(r+1)n-1},$$
$$y = y_0 + y_1 2 + \ldots + y_{(r+1)n-1} 2^{(r+1)n-1}.$$

Hagamos la siguiente expansión para x e y en $r+1$ "partes"

$$x = X_0 + X_1 2^n + \ldots + X_r 2^{rn}, \qquad y = Y_0 + Y_1 2^n + \ldots + Y_r 2^{rn},$$

donde cada X_i y Y_i son enteros de n-bits. Nos interesa manejar a la estructura de los números que depende unicamente de los X_i y Y_i. Con esa idea considere a los siguientes polinomios

$$p(t) = X_0 + X_1 t + \ldots + X_r t^r, \qquad q(t) = Y_0 + Y_1 t + \ldots + Y_r t^r. \tag{3.3}$$

El producto de ellos es

$$w(t) = p(t)q(t) = W_0 + W_1 t + \cdots + W_{2r} t^{2r}.$$

Con la construcción de (3.3) se tiene $x = p(2^n)$ y $y = q(2^n)$, y de esta forma el producto deseado xy se puede conocer mediante la evaluación del polinomio $w(t)$. De hecho $xy = w(2^n)$. El problema ahora es encontrar una manera conveniente para conocer los coeficientes W_i para usar unicamente

$2r+1$ multiplicaciones entre enteros de n-bits más otras tantas operaciones de menor orden. El cálculo de los coeficientes se hace utilizando la FFT y para ello es preciso hacer productos $W(t) = X(t)Y(t)$ que demanda la multiplicación de enteros de n-bits (T(n)) en un número de $2r+1$ evaliuaciones, se sabe que el costo de aplicar la FFT es del orden $r \log r$ y por otra parte las sumas no exceden el orden n o r. De esta forma tenemos

$$T((r+1)n) \leq (2r+1)T(n) + cn,$$

que conicide precisamente con (3.2). Finalmente, dicha relación nos lleva a establecer la siguiente cota superior.

$$T(n) \ll n^{1+\log_{r+1} 2}.$$

Otra forma de escribir este resultado es en la manera siguiente. Dado cualquier $\varepsilon > 0$ existe un algoritmo tal que el número de operaciones elementales $T(n)$ necesarias para multiplicar enteros de n-bits satisface

$$T(n) < c(\varepsilon)n^{1+\varepsilon}.$$

Capítulo 4

Desarrollo del proyecto

Se implementaron los tres algoritmos en el lenguaje C++. Para representar un entero grande se utilizó la clase `vector`, puesto que C++ sólo permite utilizar a lo mÃ¡s nÃºmeros de 12 cifras (declarÃ¡ndolos como `long long int`).

En una estructura llamada `gran_entero` se tiene el atributo I, que es un vector, y el método `inicializar(n)` donde se generan números aleatorios e incertándolos en el vector para crear el entero grande de tamaño n. Esta estructura se utilizó para la implementación de los tres algoritmos.

Para la multiplicación estándar se implementó la siguiente forma recursiva del algoritmo, siendo u y v dos entero grandes de tamaño n.

producto_estandar(u,v)
$n \leftarrow \max(|u|,|v|)$
si $u = 0$ o $v = 0$ **entonces**
  devolver 0
si no
  si $n = 1$ **entonces**
    devolver $u \times v$
  fin si
si no
  $m \leftarrow \lfloor \frac{n}{2} \rfloor$
  $x = u \div 10^m$
  $y = u \bmod 10^m$

$w = v \div 10^m$
$z = v \bmod 10^m$
devolver producto$(x, w) \times 10^{2m}$+ (producto(x, z)+ producto$(w, y)) \times$
10^m+ producto(y, z)
fin si

Las operaciones de división, módulo y suma fueron implementadas para poder manipularse sobre vectores. El código se puede observar en el apéndice A.1.

Para el método de Karatsuba fue similar, ya que hace una factorización y una llamada recursiva menos. El algoritmo implementado fue el siguiente.

producto_karatsuba(u,v)
$n \leftarrow \max(|u|,|v|)$
si $u = 0$ o $v = 0$ **entonces**
devolver 0
si no
si $n = 1$ **entonces**
devolver $u \times v$
fin si
si no
$m \leftarrow \lfloor \frac{n}{2} \rfloor$
$x = u \div 10^m \quad y = u \bmod 10^m$
$w = v \div 10^m \quad z = v \bmod 10^m$
$r =$producto2$(x + y, w + z)$
$p =$producto2(x, w)
$q =$producto2(y, z)
devolver $p \times 10^{2m} + (r - p - q) \times 10^m + q$
fin si

De igual manera, la operación resta fue implementada con el mismo propósito de manipularse sobre vectores. En el apéndice A.2 se observa el código de este método.

Por ultimo para multiplicar dos enteros utilizando la transformada rápida de Fourier se implementó el algoritmo de SchÃ¶nhage-Strassen, y asÃ para la implementación de la FFT se utilizó una forma recursiva. Sea a un entero

grande de tamaño n (suponer que n es una potencia de dos), ω una raíz enésima de la unidad y y con los valores de la transformada.

FFT(a,ω)
si $n = 1$ **entonces**
  devolver $y = a$
si no
  $x \leftarrow \omega$
  $a^{pares} \leftarrow [a_0, a_2, \ldots, a_{n-2}]$
  $a^{impares} \leftarrow [a_1, a_3, \ldots, a_{n-1}]$
  $y^{pares} \leftarrow \text{FFT}(a^{pares}, \omega^2)$
  $y^{impares} \leftarrow \text{FFT}(a^{impares}, \omega^2)$
  para $i \leftarrow 0$ hasta $n/2$ **hacer**
    $y_i \leftarrow y_i^{pares} + x * y_i^{impares}$
    $y_{i+n/2} \leftarrow y_i^{pares} - x * y_i^{impares}$
    $x \leftarrow x * \omega$
  fin para
  devolver y
fin si

Esta implementación divide el vector en posiciones pares e impares para hcaer la combinación de la transformada, multiplicando por las raíces de la unidad. En el algoritmo de SchÃ¶nhage-Strassen se hace uso de la FFT inversa, asÃ que se implementó la siguiente forma de la inversa.

IFFT(a,ω)
$a \leftarrow complejo\_conjugado$
FFT(a,ω)
Escalar(a)
devolver a

Una vez desarrollado la transformada junto con su inversa fue el implementado el algoritmo de multiplicación de enteros grandes de Schonhage-Strassen.

producto_Schonhage-Strassen(u,v)
$m \leftarrow |u| + |v| - 1$
$k \leftarrow 0$

mientras $m > 2^k$ **hacer**
  $k \leftarrow k + 1$
fin mientras
$u \leftarrow$ aumentar_ceros(2^k)
$v \leftarrow$ aumentar_ceros(2^k)
$uf \leftarrow$FFT(u,k)
$vf \leftarrow$FFT(v,k)
para $i \leftarrow 0$ hasta 2^k **hacer**
  $r_i \leftarrow uf_i * vf_i$
fin para
$res \leftarrow$IFFT(r,k)
devolver suma_acareos(res)

En la implementación se utilizó la clase `complex` y la clase `valarray`. Esto por el manejo de números complejos en el algoritmo y por la versatilidad de utilizar operaciones entre vectores con `valarray`. En el código del apéndice A.3 se observa la implementación como es descrita.

4.1. Resultados

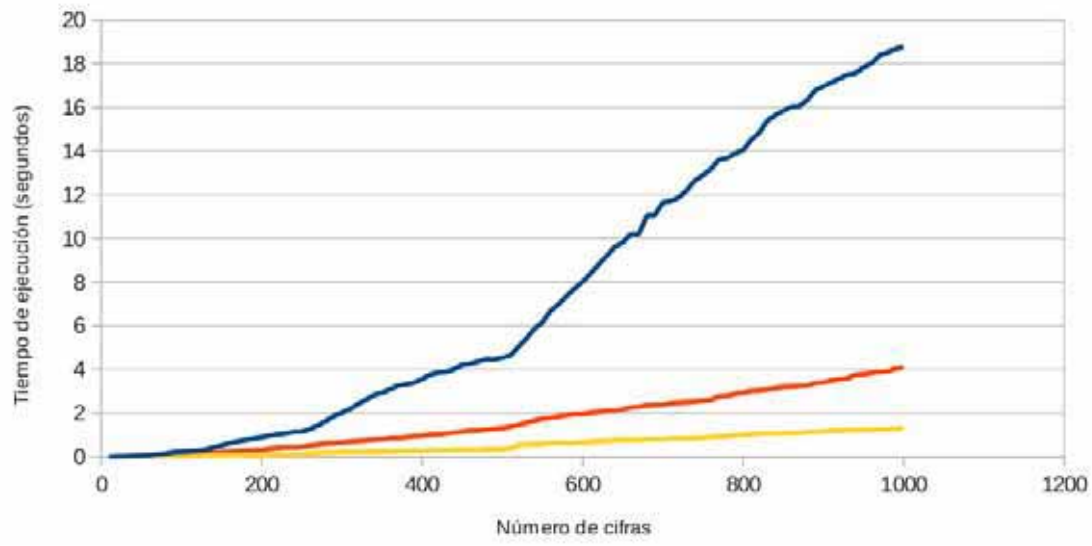
En la tabla 1 se muestra el resultado de las pruebas realizadas. Nada fuera de lo comentado teóricamente, los tiempos de ejecución del algoritmo estándar son muy grandes a comparación del método empleando la FFT.

Número de cifras	Estándar	Karatsuba	FFT
10	0	0	0
20	0	0	0
30	0.02	0.01	0
40	0.05	0.03	0.01
50	0.06	0.05	0.01
100	0.23	0.11	0.02
150	0.52	0.22	0.06
200	0.9	0.32	0.08
250	1.19	0.45	0.1
300	2.1	0.66	0.21
350	2.95	0.81	0.23
400	3.76	0.98	0.29
450	4.28	1.15	0.31
500	4.74	1.41	0.36
600	8.76	1.97	0.73
700	11.78	2.37	0.80
800	14.57	2.98	1.04
900	17.51	3.45	1.15
1000	18.87	3.86	1.30
1200	33.39	6.0	2.6

Cuadro 4.1: Tabla de resusltados los tiempos de ejecución en segundos al multiplicar 20 parejas de enteros grandes con cada uno de los algoritmos

Capítulo 5

Análisis y discusión de resultados



En la gráfica se muestra el número de cifras contra el tiempo de ejecución en segundos. la línea azul representa la gráfica de la multiplicación estándar, la línea roja al algoritmo de Karatsuba y la línea naranja al algoritmo de Strassen.

Es notable que a partir de números de 100 cifras comienzan a despegarse las gráficas. El algoritmo común comienza a crecer muy rápido, el algoritmo de Karatsuba tiene a crecer mucho menos pero el algoritmo de Strassen, bastante rápido, crece muy lento. Para números de más de 400 cifras se observa que el algoritmo común ya está muy por arriba de éstos dos, y el algoritmo de Karatsuba comienza apenas a despegarse del de Strassen, esto quiere decir que es aceptable multiplicar con el algoritmo de Karatsuba hasta números de 400 cifras, a partir de ahí lo apropiado y óptimo será utilizar el algoritmo de Strassen.

Capítulo 6

Conclusiones

Los resultados son consitentes con el marco teórico. El algoritmo estandar se volvió cada vez más lento como se muestra en las gráficas y en las tablas de resultados. Además, el uso de la FFT mejoró considerablemente los tiempos de ejecución tal y como se esperaba.

Existen muchas variantes en el uso de la FFT, una de esas variantes es usando aritmética modular y mejora el tiempo de ejecución. Sin embargo, lo que hemos presentamos representa las ideas pioneras que hicieron posible la aceleración del proceso de multiplicación.

Bibliografía

[1] D. Dozen, *The Design and Analysis of Algortihms*, Ed. Springer, 1991, pp. 186–190.

[2] A. Karatsuba, *The complexity of computations,* Proceedings of the Steklov Institute of Mathematics, **211**, 1995, 169–183.

[3] A. Karatsuba and Yu. Ofman, *Multiplication of multiplace numbers on automata,* Dokl. Akad. Nauk SSSR, **145** No. 2, (1962), 293–294.

[4] D. Knuth, *The art of computer programming Vol. 2,* (second edition), Ed. Addison–Wesley, 1980.

[5] K. Lo and A. Parvis, *Reconstructing randomly sampled signals by the FFT*, Circuits and Systems, 1996. ISCAS '96, Connecting the World, 1996 IEEE International Symposium, Vol 2, pp. 124–127.

[6] I. Parberry, *"Lecture Notes on Algorithm Analysis and Computational Complexity"*, 4ta. Ed. University of North Texas, 2001, pp. 3-5.

[7] R. Sedgewirck, *"Algorithms In C++"*, Ed. Pearson Education, 2009, pp. 637–647.

[8] I. Selesnick and C. Burrus, *Automatic Generation of Prime Length FFT Program*, [En línea]: `http://dsp.rice.edu/publications/automatic-generation-prime-length-fft-programs`

[9] G. Strang, *Wavelets,* American Scientist, **82** No. 3, 1994, 250–255.

Capítulo 7

Apéndice

A.1 Código de la multiplicación estándar

```
#include <iostream>
#include <cstdlib>
#include <time.h>
#include <vector>
#include <algorithm>
using namespace std;

struct gran_entero
{
    vector <int> I;

    void inicializar(int n)
    {
        for(int i=0;i<n;i++)
        {
            int temp;
            temp=rand()%10;
            if(i==0 && temp==0)
            {
                while(temp==0)
                {
                    temp=rand()%10;
```

```
                }
                I.push_back(temp);
            }
            else
            {
                I.push_back(temp);
            }
        }
    }

};

gran_entero multiplica (gran_entero n, int m)
{
    for(int j=1;j<=m;j++)
    {
        n.I.push_back(0);
    }
    return n;
}

gran_entero divide (gran_entero n, int m)
{
    if(n.I.size( ) == 1)
    {
        n.I.pop_back( );
        n.I.push_back(0);
        return n;
    }
    else {
    for(int i=1;i<=m;i++)
    {
        n.I.pop_back();
    }
    return n;
    }
}
```

```
gran_entero rem (gran_entero n, int m)
{
    if(n.I.size( ) == 1)
    {
        return n;
    }
    else {
    int b = n.I.size() - m;
    n.I.erase(n.I.begin(), n.I.begin()+b);
    return n;
    }
}

gran_entero redim(int m, gran_entero n)
{
    gran_entero nw;
    vector <int> aux;
    int a; int t = n.I.size( );

    for(int i = 1;i<=m;++i)
    {
        nw.I.push_back(0);
    }

    for(int j=1;j<=t;j++)
    {
        a = n.I.back();
        n.I.pop_back();
        aux.push_back(a);
    }

    for(int j=1;j<=t;j++)
    {
        a = aux.back();
        aux.pop_back();
        nw.I.push_back(a);
    }
    return nw;
```

```
}

gran_entero sumar(gran_entero num1,gran_entero num2)
{
    gran_entero res;
    int t,d,n;
    int max; int s;

    if(num1.I[0] == 0 || num2.I[0] == 0)
    {
        if(num1.I[0] == 0)
        {
            return num2;
        }
        else
        {
            return num1;
        }
    }
    if (num1.I.size () != num2.I.size()) {
        if(num1.I.size( ) > num2.I.size( ))
        {
            s = num1.I.size( ) - num2.I.size( );
            num2 = redim(s,num2);
            max = num1.I.size( );
        }
        else
        {
            s = num2.I.size ( ) - num1.I.size ( );
            num1 = redim(s,num1);
            max = num2.I.size( );
        }
    }
    else {
      max = num1.I.size( );
    }
```

```
 int v[max];
for(unsigned i=0;i<max;i++)
{
    v[i] = 0;
}

t= num1.I[0] + num2.I[0];
d = t%10;
n = t/10;
if( n!=0)
{
    res.I.push_back(n);
    res.I.push_back(d);
}
else
{
    res.I.push_back(d);
}
for(unsigned i=1;i<max;i++)
{
        t = num1.I[i] + num2.I[i];
        d = t%10;
        n = t/10;
        if(n !=0) {
            v[i-1] = 1;
        }
        res.I.push_back(d);
}


for(unsigned i=0;i<max;i++)
{
    if(v[i] == 1)
    {
        res.I[i] += 1;
    }
}
return res;
```

```
}

gran_entero producto(gran_entero n1, gran_entero n2)
{
    gran_entero x; gran_entero y; gran_entero w; gran_entero z;
    gran_entero a; gran_entero b; gran_entero c;
    gran_entero resultado;
    int n,m; int t,d,p;


    n = max(n1.I.size( ), n2.I.size( ));
    if(n <= 1)
    {
        t = n1.I[0] * n2.I[0];
        d = t%10;
        p = t/10;
        if(p!=0)
        {
            resultado.I.push_back(p);
            resultado.I.push_back(d);
            return resultado;
        }
        else
        {
            resultado.I.push_back(d);
            return resultado;
        }
    }
    if(n > 1)
    {
        m = n/2;
        x = divide(n1,m); y = rem(n1,m);
        w = divide(n2,m); z = rem(n2,m);

        a = multiplica(producto(x,w), 2*m);
        b = multiplica(sumar(producto(x,z),producto(w,y)),m);
        c = sumar(a,b);
        resultado = sumar(c,producto(y,z));
```

```
        return resultado;
    }


}


int main()
{
    clock_t clo;
    srand(time(NULL));
    int n;
    int d; int t; int m=0; int a;
    vector <int> ::iterator it;
    cin >> n;

    gran_entero num1; gran_entero num2;
    gran_entero res;

    clo = clock();
    num1.inicializar(n);
    num2.inicializar(n);
    res = producto(num1,num2);
     clo = clock() - clo;

     cout << "Entero 1" << endl;
    for(it=num1.I.begin();it!=num1.I.end();++it)
    {
        cout  << *it;
    }

    cout << "\n";

    cout << "Entero 2" << endl;
    for(it=num2.I.begin();it!=num2.I.end();++it)
    {
        cout << *it;
    }
```

```
    cout << "\n";

    cout << "Producto" << endl;
   for(it=res.I.begin();it!=res.I.end();++it)
    {
        cout << *it;
    }
    cout << "\n";

    cout << "tiempo de ejecuci\'on " << ((float)clo)/CLOCKS_PER_SEC << endl;

    return 0;

}
```

A.2 Código del método de Karatsuba

```
#include <iostream>
#include <cstdlib>
#include <time.h>
#include <vector>
#include <algorithm>
using namespace std;

struct gran_entero
{
    vector <int> I;

    void inicializar(int n)
    {
        for(int i=0;i<n;i++)
        {
            int temp;
            temp=rand()%10;
            if(i==0 && temp==0)
            {
```

```
                while(temp==0)
                {
                    temp=rand()%10;
                }
                I.push_back(temp);
            }
            else
            {
                I.push_back(temp);
            }
        }
    }

};

gran_entero multiplica (gran_entero n, int m)
{
    for(int j=1;j<=m;j++)
    {
        n.I.push_back(0);
    }
    return n;
}

gran_entero divide (gran_entero n, int m)
{
    if(n.I.size() <= m)
    {
        n.I.clear();
        n.I.push_back(0);
        return n;
    }
    else {
    if(n.I.size( ) == 1)
    {
        n.I.pop_back( );
        n.I.push_back(0);
        return n;
```

```
    }
    else {
    for(int i=1;i<=m;i++)
    {
        n.I.pop_back();
    }
    return n;
    }

    }
}

gran_entero rem (gran_entero n, int m)
{
    if(n.I.size( ) == 1 || n.I.size() <= m)
    {
        return n;
    }
    else {
    int b = n.I.size() - m;
    n.I.erase(n.I.begin(), n.I.begin()+b);
    return n;
    }
}

gran_entero diez(gran_entero m)
{
    gran_entero r;
    vector <int> aux;
    int a; int t = m.I.size();

    r.I.push_back(1);
    r.I.push_back(0);

    for(int j=1;j<=t-1;j++)
    {
        a = m.I.back();
        m.I.pop_back();
```

```
        aux.push_back(a);
    }

    for(int j=1;j<=t-1;j++)
    {
        a = aux.back();
        aux.pop_back();
        r.I.push_back(a);
    }
    return r;

}

gran_entero redim(int m, gran_entero n)
{
    gran_entero nw;
    vector <int> aux;
    int a; int t = n.I.size( );

    for(int i = 1;i<=m;++i)
    {
        nw.I.push_back(0);
    }

    for(int j=1;j<=t;j++)
    {
        a = n.I.back();
        n.I.pop_back();
        aux.push_back(a);
    }

    for(int j=1;j<=t;j++)
    {
        a = aux.back();
        aux.pop_back();
        nw.I.push_back(a);
    }
    return nw;
```

```
}

gran_entero sumar(gran_entero num1,gran_entero num2)
{
    gran_entero res;
    int t,d,n;
    int max; int s, tam;

    if (num1.I.size () != num2.I.size()) {
        if(num1.I.size( ) > num2.I.size( ))
        {
            s = num1.I.size( ) - num2.I.size( );
            num2 = redim(s,num2);
            max = num1.I.size( );
        }
        else
        {
            s = num2.I.size ( ) - num1.I.size ( );
            num1 = redim(s,num1);
            max = num2.I.size( );
        }
    }
    else {
      max = num1.I.size( );
    }

    t= num1.I[0] + num2.I[0];
    d = t%10;
    n = t/10;
    if( n!=0)
    {
        res.I.push_back(n);
        res.I.push_back(d);
        tam = max+1;
    }
    else
    {
```

```
    tam = max;
    res.I.push_back(d);

}

if(num1.I.size() == 1 && num2.I.size() == 1)
{
    return res;
}
else {

int v[tam];
for(unsigned i=0;i<tam;i++)
{
    v[i] = 0;
}

for(unsigned i=1;i<max;i++)
{
        t = num1.I[i] + num2.I[i];
        d = t%10;

        n = t/10;
        if(n !=0) {
            if(tam == max)
            {
                v[i-1] = 1;
            }
            else
            {
                v[i] = 1;
            }
        }
        res.I.push_back(d);
}

for(unsigned i=0;i<tam;i++)
{
```

```
    if(v[i] == 1)
    {
        if( i == 0 && res.I[i] == 9)
        {
            res = diez(res);
        }
        else {
        if(res.I[i] == 9)
        {
            res.I[i] = 0;
            res.I[i-1] += 1;
        }
        else {
        res.I[i] += 1;
        }
        }
    }

    }
}

int contc = 0;
if(res.I[0] == 0)
{
  for(int i=0;i<res.I.size();i++)
  {
      if(res.I[i] == 0)
      {
          contc++;
      }
      else if(res.I[i] != 0)
      {
          break;
      }
  }
  res.I.erase(res.I.begin(), res.I.begin()+contc);
}
```

```
    return res;
    }

gran_entero restar (gran_entero num1, gran_entero num2)
{
    gran_entero res;
    int s, max;
    int t; int contc = 0;

        if(num1.I.size( ) != num2.I.size( ))
        {
            s = num1.I.size() - num2.I.size();
            num2 = redim(s,num2);
            max = num1.I.size();
        }
        else
        {
            max = num1.I.size();
        }
        int v[max];
        for(unsigned i=0;i<max;++i)
        {
            v[i] = 0;
        }
        t = num1.I[0] - num2.I[0];
        res.I.push_back(t);

        for(unsigned i=1;i<max;i++)
        {
            if(num1.I[i] == num2.I[i])
            {
                res.I.push_back(0);
            }
            else if(num1.I[i] < num2.I[i])
            {
                num1.I[i] = num1.I[i] + 10;
                t = num1.I[i] - num2.I[i];
                res.I.push_back(t);
```

```
            v[i-1] = 1;
        }
        else
        {
            t = num1.I[i] - num2.I[i];
            res.I.push_back(t);
        }
    }

    for(unsigned i=0;i<max;i++)
    {
        if(v[i] == 1 && res.I[i] == 0)
        {
            res.I[i] = 9;
            res.I[i-1] -= 1;
        }
        else if (v[i] == 1)
        {
            res.I[i] -= 1;
        }
    }

if(res.I[0] == 0)
{
  for(int i=0;i<res.I.size();i++)
  {
      if(res.I[i] == 0)
      {
          contc++;
      }
      else if(res.I[i] != 0)
      {
          break;
      }
  }
  res.I.erase(res.I.begin(), res.I.begin()+contc);
}
```

```
        return res;

}

gran_entero producto(gran_entero n1, gran_entero n2)
{
    gran_entero x; gran_entero y; gran_entero w; gran_entero z;
    gran_entero r; gran_entero p; gran_entero q;
    gran_entero a; gran_entero b; gran_entero v; gran_entero u;
    gran_entero resultado;
    int n,m; int t,d,c;

    n = max(n1.I.size( ), n2.I.size( ));
    if(n <= 1)
    {
        t = n1.I[0] * n2.I[0];
        d = t%10;
        c = t/10;
        if(c!=0)
        {
            resultado.I.push_back(c);
            resultado.I.push_back(d);
            return resultado;
        }
        else
        {
            resultado.I.push_back(d);
            return resultado;
        }
    }
    if(n > 1)
    {
        m = n/2;
        x = divide(n1,m); y = rem(n1,m);
        w = divide(n2,m); z = rem(n2,m);

        u = sumar(x,y);
        v = sumar(w,z);
```

```
        r = producto(u,v);
        p = producto(x,w);
        q = producto(y,z);

        a = multiplica(p,2*m);
        b = multiplica(restar(restar(r,p),q),m);


        resultado = sumar(sumar(a,b),q);
        return resultado;
    }


}


int main()
{
    clock_t clo;
    srand(time(NULL));
    int n;
    int t;
    vector <int> ::iterator it;
    cin >> n;

    gran_entero num1; gran_entero num2;
    gran_entero res;

    clo = clock();
    num1.inicializar(n);
    num2.inicializar(n);
    res = producto(num1,num2);
    //res = restar(num1,num2);
     clo = clock() - clo;

     cout << "Entero 1" << endl;
    for(it=num1.I.begin();it!=num1.I.end();++it)
```

```
    {
        cout  << *it;
    }

    cout << "\n";

    cout << "Entero 2" << endl;
    for(it=num2.I.begin();it!=num2.I.end();++it)
    {
        cout << *it;
    }

    cout << "\n";

    cout << "Producto" << endl;
   for(it=res.I.begin();it!=res.I.end();++it)
    {
        cout << *it;
    }
    cout << "\n";

    cout << "tiempo de ejecuci\'on " << ((float)clo)/CLOCKS_PER_SEC << endl;

    return 0;

}
```

A.3 Código de la multiplicación usando FFT

```
#include <complex>
#include <iostream>
#include <cstdlib>
#include <time.h>
#include <valarray>
#include <vector>
#include <math.h>
```

```
using namespace std;

const double PI = 3.141592653589793238460;
typedef std::complex<double> Complex;
typedef std::valarray<Complex> CArray;



struct gran_entero
{
    vector <int> I;

    void inicializar(int n)
    {
        for(int i=0;i<n;i++)
        {
            int temp;
            temp=rand()%10;
            if(i==0 && temp==0)
            {
                while(temp==0)
                {
                    temp=rand()%10;
                }
                I.push_back(temp);
            }
            else
            {
                I.push_back(temp);
            }
        }
    }

};


gran_entero conv_entero_vec (int n)
{
```

```
    gran_entero m; gran_entero aux;
    int a;

    while(n != 0)
    {
        a = n % 10;
        aux.I.push_back(a);
        n /= 10;
    }

    int t = aux.I.size();

    for(int i=0;i<t;++i)
    {
        a = aux.I.back();
        m.I.push_back(a);
        aux.I.pop_back();
    }
    return m;
}

gran_entero diez(gran_entero m)
{
    gran_entero r;
    vector <int> aux;
    int a; int t = m.I.size();

    r.I.push_back(1);
    r.I.push_back(0);

    for(int j=1;j<=t-1;j++)
    {
        a = m.I.back();
        m.I.pop_back();
        aux.push_back(a);
    }

    for(int j=1;j<=t-1;j++)
```

```
    {
        a = aux.back();
        aux.pop_back();
        r.I.push_back(a);
    }
    return r;

}

gran_entero redim(int m, gran_entero n)
{
    gran_entero nw;
    vector <int> aux;
    int a; int t = n.I.size( );

    for(int i = 1;i<=m;++i)
    {
        nw.I.push_back(0);
    }

    for(int j=1;j<=t;j++)
    {
        a = n.I.back();
        n.I.pop_back();
        aux.push_back(a);
    }

    for(int j=1;j<=t;j++)
    {
        a = aux.back();
        aux.pop_back();
        nw.I.push_back(a);
    }
    return nw;

}

gran_entero sumar(gran_entero num1,gran_entero num2)
```

```
{
    gran_entero res;
    int t,d,n;
    int max; int s, tam;

    if (num1.I.size () != num2.I.size()) {
        if(num1.I.size( ) > num2.I.size( ))
        {
            s = num1.I.size( ) - num2.I.size( );
            num2 = redim(s,num2);
            max = num1.I.size( );
        }
        else
        {
            s = num2.I.size ( ) - num1.I.size ( );
            num1 = redim(s,num1);
            max = num2.I.size( );
        }
    }
    else {
      max = num1.I.size( );
    }

    t= num1.I[0] + num2.I[0];
    d = t%10;
    n = t/10;
    if( n!=0)
    {
        res.I.push_back(n);
        res.I.push_back(d);
        tam = max+1;
    }
    else
    {
        tam = max;
        res.I.push_back(d);

    }
```

```
if(num1.I.size() == 1 && num2.I.size() == 1)
{
    return res;
}
else {

int v[tam];
for(unsigned i=0;i<tam;i++)
{
    v[i] = 0;
}

for(unsigned i=1;i<max;i++)
{
        t = num1.I[i] + num2.I[i];
        d = t%10;

        n = t/10;
        if(n !=0) {
            if(tam == max)
            {
                v[i-1] = 1;
            }
            else
            {
                v[i] = 1;
            }
        }
        res.I.push_back(d);
}

for(unsigned i=0;i<tam;i++)
{
    if(v[i] == 1)
    {
        if( i == 0 && res.I[i] == 9)
        {
```

```
                res = diez(res);
            }
            else {
            if(res.I[i] == 9)
            {
                res.I[i] = 0;
                res.I[i-1] += 1;
            }
            else {
            res.I[i] += 1;
            }
            }
        }

        }
    }

    int contc = 0;
    if(res.I[0] == 0)
    {
      for(int i=0;i<res.I.size();i++)
      {
          if(res.I[i] == 0)
          {
              contc++;
          }
          else if(res.I[i] != 0)
          {
              break;
          }
      }
      res.I.erase(res.I.begin(), res.I.begin()+contc);
    }

    return res;
}

gran_entero ceros(gran_entero n, int m, int k)
```

```
{
    int d = m - k;

    for(int i=1;i<=d;++i)
    {
        n.I.push_back(0);
    }
    return n;
}

gran_entero suma_total(vector <double> aca)
{
    int num = aca.size(); double a;
    gran_entero sum;
    gran_entero aux;
    sum.I.push_back(0);

    for(int i=1;i<=num;++i)
    {
        a = round(aca.back());
        aux = conv_entero_vec(a);
        aux = ceros(aux,num,i);
        sum = sumar(sum,aux);
        aca.pop_back();
    }

    return sum;
}


void fft(CArray& x)
{
    const size_t n = x.size();
    if (n <= 1)
    {
        return;
    }
    else {
```

```
    CArray even = x[std::slice(0, n/2, 2)];
    CArray odd = x[std::slice(1, n/2, 2)];

    fft(even);
    fft(odd);

    for (size_t k = 0; k < n/2; ++k)
    {
        Complex t = std::polar(1.0, -2 * PI * k / n) * odd[k];
        x[k] = even[k] + t;
        x[k+n/2] = even[k] - t;
    }

    }
}


void ifft(CArray &x)
{
    x = x.apply(std::conj);
    fft(x);
    x /= x.size();
}


int main()
{
    clock_t clo;
    int n; int k,p; int m,t;
    srand(time(NULL));
    gran_entero n1; gran_entero n2; gran_entero resultado;
    int aux;
    vector <double> acareos;
    vector <int> ::iterator it;

    cin >> n;
```

```
clo = clock();
n1.inicializar(n);
n2.inicializar(n);

cout << "Entero 1" << endl;
for(it=n1.I.begin();it!=n1.I.end();++it)
{
    cout  << *it;
}

cout << "\n";

cout << "Entero 2" << endl;
for(it=n2.I.begin();it!=n2.I.end();++it)
{
    cout << *it;
}

cout << "\n";

k=0; m=(n1.I.size()+n2.I.size())-1;


while(m > pow(2,k))
{
    k++;
}
int tam = pow(2,k);


Complex num1[tam];
int b = n1.I.size();

for(int i=0;i<b;++i)
{
    aux = n1.I.back();
    num1[i]=aux;
    num1[i+1]=0;
```

```
    n1.I.pop_back();
}

t = k- b;
p = (b*2)-1;

for(int j=1;j<=t;j++)
{
    num1[p+j] = 0;
}

CArray fft1(num1, tam);


Complex res[tam];
for(int i=0;i<tam;++i)
{
    res[i] = 0;
}

Complex num2[tam];
b = n2.I.size();

for(int i=0;i<b;++i)
{
    aux = n2.I.back();
    num2[i]=aux;
    num2[i+1]=0;
    n2.I.pop_back();
}

t = k- b;
p = (b*2)-1;

for(int j=1;j<=t;j++)
{
    num2[p+j] = 0;
}
```

```
    CArray fft2(num2, tam);


    CArray result(res,tam);


    fft(fft1);
    fft(fft2);

    result = fft1 * fft2;
    ifft(result);

    for(int i=0; i<tam;i++)
    {
        if( result[i].real() >= 0)
        {
            acareos.push_back(result[i].real());
        }
    }


   resultado = suma_total(acareos);
   clo = clock() - clo;


    cout << "Producto" << endl;
   for(it=resultado.I.begin();it!=resultado.I.end();++it)
    {
        cout << *it;
    }
    cout << "\n";

    cout << "tiempo de ejecuci\'on " << ((float)clo)/CLOCKS_PER_SEC << endl;

    return 0;
}
```