Capítulo 3

Jonathan Zinzan Salisbury Vega
Universitat de les Illes Balears
Palma, España
jonathan.salisbury1@estudiant.uib.cat

Joan Sansó Pericás
Universitat de les Illes Balears
Palma, España
joan.sanso4@estudiant.uib.cat

Joan Vilella Candia
Universitat de les Illes Balears
Palma, España
joan.vilella1@estudiant.uib.cat

La multiplicación de números con muchos dígitos es un problema común en la informática, especialmente con la búsqueda de nuevos números primos. Se hablará de los distintos algoritmos conocidos para este problema y sus complejidades asintóticas. Además, se tratarán temas comunes de la programación como la Recursividad y la Concurrencia.

Keywords—Karatsuba, Divide y Vencerás, Coste asintótico, Recursividad, Concurrencia.

I. INTRODUCCIÓN

En este documento se pretende explicar uno de los paradigmas más importantes dentro de la algoritmia, los algoritmos "Divide y vencerás". Los cuales están basados en la resolución recursiva de problemas, en la que un problema mayor se va dividiendo en una serie de subproblemas que sean suficientemente sencillos o de solución trivial. Por esta naturaleza de los problemas, es muy común las soluciones basadas en recursividad. En la cual, por cada una de las llamadas, se va reduciendo la complejidad del problema.

Este paradigma se utiliza para resolver uno de los problemas más comunes en programación, que es la multiplicación de grandes números. Esto cobra una vital importancia para cálculos de gran precisión, por ejemplo. El problema surge cuando al multiplicar números con una cantidad considerable de números, la operación se vuelve inviable (de la manera común). Ya que el coste asintótico de dicho algoritmo es muy elevado (n^2). Con el método Karatsuba se reduce esa complejidad haciéndolo nuevamente viable. Esto se consigue tratando la multiplicación con subsecuencias de estos. Mediante recursividad se irán reduciendo estas subsecuencias hasta que su cálculo sea trivial (multiplicaciones de un dígito).

En esta práctica se implementarán tres posibles soluciones para la multiplicación de grandes números. Una de estas soluciones es inviable, la tradicional. También se implementará "a mano" el método de Karatsuba y finalmente se implementará una solución mixta, en la que se combinarán ambas técnicas.

Todo esto con motivo de comprobar empíricamente la viabilidad de las distintas opciones. Además, se ha implementado una funcionalidad adicional en la que cuando el programa se ejecuta, calcula a partir de qué número de dígitos es más recomendable utilizar un algoritmo u otro (algoritmo mixto).

Finalmente, toda esta práctica se ha de desarrollar siguiendo el patrón de desarrollo de software MVC (Modelo Vista Controlador). Para ello se ha tenido que desarrollar la práctica teniendo en cuenta estas tres clases que determinan este comportamiento

II. CONTEXTO Y ENTRONO DE ESTUDIO

Uno de los requisitos a la hora de realizar la práctica es el uso del lenguaje de programación Java. Además, se ha dado la opción de elegir entre varios IDE's (*Integrated Development Environment*).

- NetBeans
- IntelliJ
- VS Code

En este caso se ha escogido el IDE de NetBeans por familiaridad de uso. Además, se utiliza una herramienta de control de versiones (Git). Más específicamente su versión de escritorio Github Desktop por su facilidad de uso mediante interfaz gráfica.

III. DESPCRIPCIÓN DEL PROBLEMA

El problema que se plantea en esta práctica es el desarrollo y comparación de tres algoritmos de multiplicación de dos enteros de gran tamaño. Este problema es uno de los ejemplos más típicos dentro del paradigma de los algoritmos "Divide y vencerás". Estos han de ser de N y M cifras, por lo que hay que tener en cuenta la longitud de estos.
Los algoritmos que se han implementado en esta práctica son:

- Tradicional
- Karatsuba
- Mixto

A. *Multiplicacion Tradicional*

Es importante entender la connotación de "tradicional" a la hora de analizar el funcionamiento de este algoritmo.
"Los algoritmos convencionales tratan a las cifras en forma aislada como si fuesen números y no se tiene noción de la totalidad que implican las cifras, es decir el valor que tienen por su posicionalidad en el numeral. Además, ocultan cálculos y propiedades que se aplican."

El funcionamiento del algoritmo consiste en ir de derecha a izquierda multiplicando teniendo en cuenta: ley de signos y orden de unidades. Para ello se colocarán unidades debajo de mismo orden de unidades, finalmente se suman los productos de cada cifra del segundo factor por todas las del primero. La suma se realiza también de manera tradicional, de derecha a izquierda.

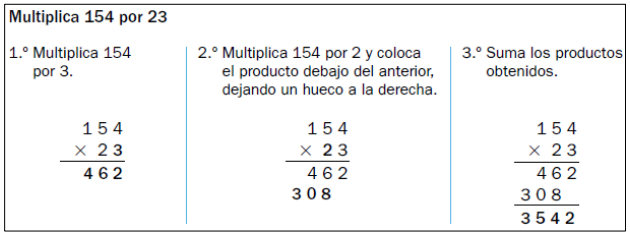

Ilustración 1

B. *Algoritmo de Karatsuba*

Este algoritmo es un algoritmo de multiplicación rápida. Esto se debe a que logra reducir la multiplicación de n dígitos a como máximon^2. Esto se consideraba imposible hasta que Anatolii Alexeevitch Karatsuba lo descubrió en el año 1960. Permitiendo así reducir el algoritmo clásico que necesita $n^{\log_2 3}$ multiplicaciones por dígito.

A continuación, se explicará el funcionamiento del algoritmo, la demostración matemática queda fuera del alcance de este documento.

El algoritmo se fundamenta en el uso de la siguiente ecuación:

$$N = N_1 B^m + N_0$$

Ecuación 1

Donde N es el número original. N_1 y N_0 son las subcadenas en las que se descompone el número original y m es la base del número.
Teniendo en cuenta esta fórmula, se desarrollan las siguientes teniendo en cuenta un producto de dos números x e y.

$$xy = (x_1 B^m + x_0)(y_1 B^m + y_0)$$

Ecuación 2

Si se aplican las siguientes igualdades:

- $z_2 = x_1 * y_1$
- $z_1 = x_1 * y_0 + x_0 * y_1$
- $z_0 = x_0 * y_0$

Ahora bien, para esto se necesitan cuatro multiplicaciones. Karatsuba se dio cuenta que se puede calcular z_1 en función de z_2 y z_0. Reduciendo así el número de multiplicaciones a costa de unas restas. Por lo que z_1 quedaría:

$$z_1 = (x_1 + x_0)(y_1 + y_0) - z_2 - z_0$$

Ecuación 3

Finalmente, la fórmula sería:

$$xy = z_2 B^{2m} + z_1 B^m + z_0$$

Ecuación 4

Toda esta mejora de eficiencia se hace a costa de un coste en almacenamiento. Ya que una vez se calculen los valores de z_0 y z_2, se almacenarán en memoria para poder volver a ser utilizados. Es un error muy común implementar este algoritmo sin tener en cuenta esto, por lo que al final acaba siendo igual en cuanto coste asintótico que el algoritmo tradicional.

C. *Algoritmo Mixto*

Este algoritmo surge con la intención de juntar lo mejor de los dos algoritmos antes mencionados. A priori, puede parecer que el algoritmo de Karatsuba no tiene rival, ya que, al ser mejor su coste asintótico siempre debería de superar al algoritmo tradicional. Esto no es así debido a la constante multiplicativa.

El conjunto de operaciones computacionalmente "menores" que se obvian a la hora de calcular el coste asintótico de los algoritmos, finalmente tiene un gran peso sobre el rendimiento real de los algoritmos.
Por esta razón, en esta práctica es objeto de estudio esta frontera donde el algoritmo tradicional deja de ser la mejor opción (aun siendo asintóticamente peor) y lo pasa a ser el algoritmo de Karatsuba.

A raíz de esto, el algoritmo mixto tiene un comportamiento dual. El algoritmo tradicional es más eficiente para números más pequeños, por lo que el algoritmo mixto comenzará como tal y cuando se supere ese número de cifras donde el Karatsuba es mejor, hará el cambio y se comportará como tal. Por esta razón, en cuanto a eficiencia de los algoritmos, el mixto es la mejor opción.

IV. SOLUCIÓN PROPUESTA

Para la implementación de los distintos algoritmos se han llevado a cabo a nivel de String. No se ha utilizado ninguna estructura de creación propia.

A. *Algoritmo Tradicional*

Una de las primeras cosas que se han tenido en cuenta a la hora de implementar este algoritmo es el signo de los números, es por eso por lo que es la primera comprobación que se realiza. Se tiene en cuenta si los signos de los números

introducidos difieren, en caso de ser así se sabe que el resultado será negativo y se guarda dicha información. A la hora de realizar el cálculo como tal, siempre se realiza con los números positivos. Otra comprobación que se tiene en cuenta es la de siempre multiplicar el número de mayor longitud (en dígitos) por el de menor longitud, en caso de que el usuario los introduzca en orden inverso se ordenan.

A partir de ahí se puede decir que empieza la multiplicación de los números como tal, no sin antes invertir el orden para facilitar a la hora de multiplicar (ya que se multiplica de derecha a izquierda).

El procedimiento intenta imitar al máximo la multiplicación "a mano". Para ello se multiplica dígito a dígito del número más pequeño por cada uno de los dígitos del número más grande. En este proceso se ha de tener en cuenta el acarreo de la operación, tanto como para sumarlo a ese nivel de dígito como para tenerlo en cuenta en la multiplicación siguiente.

Finalmente se realizará la suma de ese nivel (posición de dígito tratado) añadiendo tantos ceros como en la multiplicación clásica.

B. Karatsuba Mediante Recursividad

1) Recursividad y uso

Una definición simple de recursividad es: dada una función, la recursividad ocurre cuando esa misma función se vuelve a llamar. La recursividad tiene una especial utilidad en los algoritmos del tipo "Dividir y Vencer" ya que su aproximación a la resolución de los problemas es similar.

Definir un problema en términos de versiones más simples de él mismo. En el caso de los algoritmos de Karatsuba y mixto lo que se realiza a niveles generales es dividir la secuencia de dígitos a la mitad para más adelante realizar 3 llamadas recursivas. Una con las partes bajas de los dos números, otra con las altas y finalmente, una con la suma de las partes altas con las bajas de ambos números. Pero esto se explicará en profundidad en el siguiente apartado.

2) Solución Karatsuba

El algoritmo de Karatsuba implementado en la práctica se podría decir que tiene dos partes. La función *karatsuba()* y por la función *karatsubaRec()*. El método Karatsuba propiamente dicho es el *karatsubaRec()* pero el otro algoritmo es el encargado de realizar una serie de formatos a los números de entrada para asegurar el correcto funcionamiento del algoritmo. Puesto que el algoritmo *karatsuba()* se comenzará a comentar por ahí.

En primer lugar, se "limpian" los posibles ceros que haya podido introducir el usuario. Por ejemplo, si introduce un 0000001, después de este paso, se obtendrá un 1. A continuación y al igual que en el algoritmo tradicional, se analizan los signos de los números. Porque el algoritmo siempre realizará la multiplicación de manera positiva y añadirá el signo negativo en los casos pertinentes.

A continuación, y por una cuestión meramente funcional, se añaden 0's a la derecha del número más pequeño para equiparar los tamaños. Esto claramente añade un problema bastante grande, ya que se está modificando la magnitud de uno de los números. Pero aprovechando las propiedades de la multiplicación, es tan sencillo como borrar el mismo número de 0's en el resultado final.

Finalmente, dentro de este primer algoritmo de *karatsuba()* se almacena dentro de un String el resultado de la operación. Este resultado es fruto de la primera llamada al algoritmo *karatsubaRec().*

Es importante hacer una pequeña salvedad a la hora de comentar la solución implementada para el algoritmo de Karatsuba y es que en este mismo algoritmo se ha incluido la parte de código capaz de realizar el algoritmo mixto. Para comentar este apartado solo se hablará de la parte del código propiamente de Karatsuba, en el siguiente apartado se comentarán los trozos propios del algoritmo mixto.

En primer lugar, el caso base del algoritmo. Este es cuando la longitud de uno de los dígitos es menor que dos, es decir, uno. Como ya no se pueden dividir los números en secuencias de menor tamaño, simplemente se llama a la función de multiplicar de manera tradicional.

Una vez cubierto el caso base, el algoritmo continúa guardando los tamaños (longitud en dígitos) de ambos números. Ya que, aunque en un primer momento el algoritmo ajusta para que ambos números tengan la misma cantidad de dígitos, durante la ejecución recursiva no siempre tiene que ser así. Este tamaño se utiliza para calcular el punto de división de los números. El cual será la mitad de la longitud máxima de ambos números. Después nuevamente se rellenan con ceros para asegurar que tengan el mismo tamaño.

A continuación, se hacen las divisiones pertinentes de los números, teniendo un total de cuatro. Las partes corresponden con "la parte alta" y "baja" de cada uno de los números. Con estas partes se realizan las tres llamadas recursivas. Una con la parte alta del primer número con la parte alta del segundo número. Otra con la parte baja del primer número con la parte baja del segundo número y finalmente, con la suma de las partes altas con las partes bajas.

En esta última llamada recursiva es cuando se construye la solución. Se juntan las soluciones parciales y se añaden los ceros pertinentes a la parte alta del número. Una vez se ha construido bien la solución, se quitan todos los ceros auxiliares que se habían añadido. Finalmente se devuelve la solución.

3) Solución Mixto

La manera en la cual se ha implementado el algoritmo de Karatsuba incluye a su vez la implementación del algoritmo mixto. Para ello lo que se hace es utilizar un booleano que se pasa por parámetro el cual indica si se está ante un algoritmo mixto o uno de Karatsuba único.

Esto permite que el único añadido que se le tenga que hacer al algoritmo "original" sea la inclusión de un nuevo caso base. Este consiste en primero comprobar si se trata de un algoritmo mixto, en caso de que sea así, comprobará si al menos uno de los dos números es menor que el umbral que

se ha calculado previamente. Este umbral se almacena en la variable *nMix* la cual se le asignará el valor correspondiente desde el controlador.

V. PATRÓN MVC

El Modelo-Vista-Controlador (MVC) [1] es un patrón de diseño de software[1] en el que se divide la lógica del programa de su representación gráfica, además se hace uso de un controlador para los eventos y comunicaciones entre las distintas partes.

Este patrón de arquitectura se basa en las ideas de reutilización de código y la separación de conceptos distintos, estas características pretenden facilitar la tarea de desarrollo de aplicaciones y su posterior mantenimiento. Para ello se proponen la construcción de tres componentes:

- *Modelo*
  Es donde se almacena la información necesaria para la ejecución de la aplicación. Gestiona el acceso a dicha información mediante peticiones a través del controlador. También contiene los procedimientos lógicos que hagan uso de esa información.
- *Vista*
  Presenta la información en un formato adecuado para la interacción del usuario con los datos.

- *Controlador*
  Responde a eventos usualmente generados por el usuario y los convierte en comandos para la vista o modelo. Se podría decir que hace la función de intermediario entre la vista y el modelo encargándose de la lógica del programa.

Como se ha mencionado anteriormente se ha decidido utilizar este patrón debido a la facilidad que aporta al programar la separación de conceptos. Además, se obtiene una capa más de abstracción ya que es posible utilizar diferentes vistas con un mismo modelo.

Hoy en día, es muy común el uso de la programación orientada a objetos (POO) como paradigma principal, por ello cada componente del patrón MVC suele implementarse como una clase independiente. A continuación, se explica que proceso se ha seguido y como se ha implementado cada parte.

A. Implementación Modelo

Se ha diseñado la clase *Model* que se encarga de almacenar todos los algoritmos y funciones auxiliares necesarias para resolver el problema.

Los algoritmos han sido diseñados para realizar todos los cálculos con el uso de *Strings* para poder almacenar números grandes sin ningún problema. Únicamente se utilizan valores numéricos para almacenar dígitos sueltos, nunca resultados compuestos. También cabe destacar que todos los algoritmos se han diseñado para que sean compatibles con el conjunto de números enteros *Z*, tanto con signo positivo como con negativo.

Los métodos para los principales algoritmos utilizan únicamente funciones para el tratamiento de Strings, y las funciones auxiliares *add()* y *sub()*. Estas funciones realizan las operaciones básicas de sumas y restas, pero utilizando *Strings* para que sea compatible con números grandes. Para que estas sean compatibles con números con signos se ha seguido el siguiente proceso:

- *Add()*:

Num1	Num2	Res	
+A	+B	+A+(+B) = A + B	add(A, B)
+A	-B	+A+(-B) = A - B	sub(A, B)
-A	+B	-A+(+B) = -A + B = B - A	sub(B, A)
-A	-B	-A+(-B) = -(A + B)	-add(A, B)

Tabla 1

De esta manera conseguimos transformar todas las operaciones a operaciones normales sin signo. De esta forma dentro de la suma quedaría una suma normal, una resta normal, una resta girada y una suma añadiendo el menos.

- *Sub()*:

Num1	Num2	Res	
+A	+B	+A-(+B) = A - B	sub(A, B)
+A	-B	+A-(-B) = A + B	add(A, B)
-A	+B	-A-(+B) = -A - B = -(A+B)	-add(A, B)
-A	-B	-A-(-B) = B-A	sub(B,A)

En este caso quedan las mismas operaciones, pero cambiando el orden de estas.

Además, hay definidas algunas funciones auxiliares para poder comparar los distintos algoritmos.

B. Implementacion Vista

Para la vista se han definido dos clases diferentes para cada tipo de ventana:

- *View*:
  Es la ventana principal del problema, contiene dos campos para que el usuario pueda introducir los números y una serie de botones para obtener el resultado con los distintos algoritmos. Finalmente hay definido un campo donde se muestra el resultado de la operación juntamente con el tiempo y el algoritmo. Además, hay definido un botón "*Comparison*" que permite crear otra ventana para comparar los distintos algoritmos.

[1] Un patrón de diseño es un conjunto de técnicas utilizadas para resolver problemas comunes en el desarrollo de software.

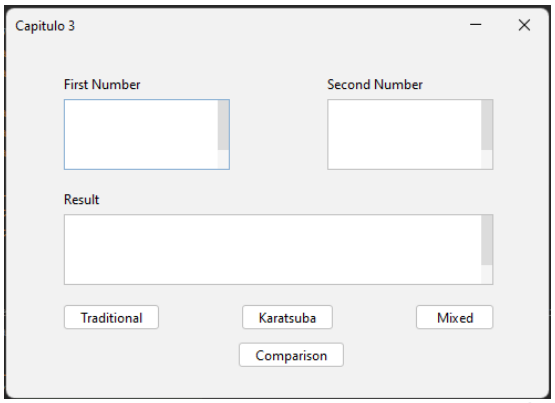

Ilustración 2

- *CompareFrame:*
  Esta ventana nos permite graficar los distintos algoritmos según el tiempo de duración, para poder compararlos de una manera visual.

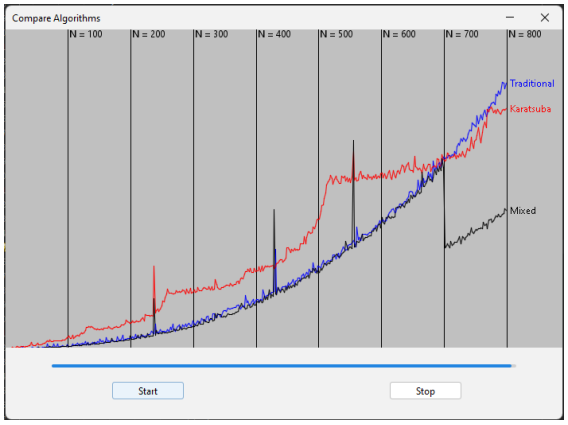

Ilustración 3

Esta ventana muestra una gráfica de cada algoritmo de un color distinto para poder diferenciarlos, además escribe el nombre de dicho algoritmo al final. Como se puede observar también muestra unos ejes para tener puntos de comparación a medida que se va dibujando.

C. *Implementacion Controlador*

Se ha diseñado la clase Controller que se encarga de gestionar eventos y manejar la interacción entre el resto de las partes. Para ello se han implementado algunos métodos que servirán de *listeners* [2]para los botones de la vista.

Cuando el usuario utilice algún botón de la interfaz este lo notifica al método correspondiente que realiza la acción necesaria si es posible y devuelve los resultados a la vista para que lo muestre. Para que el usuario pueda disfrutar de una interfaz gráfica fluida se hace uso del concepto de Concurrencia que se explica a continuación.

1) *Concurrencia*

La programación concurrente es una forma de cómputo en la que el trabajo se divide en varios hilos de ejecución distintos que trabajan simultáneamente. Esto suele mejorar el rendimiento de una aplicación al poder realizar cálculos largos en un hilo aparte. Para este proyecto se ha decidido utilizar este concepto en dos apartados distintos:

- Multiplicación:
  Para poder computar los distintos algoritmos de multiplicación sin dejar a la vista bloqueada, este cálculo se ejecuta en hilo diferente. Para ello se define el hilo usando la clase *Thread* de java con un elemento que implemente la interfaz *Runnable* el cual indica al hilo las acciones que debe realizar. En el método *run()* se utilizan los datos introducidos por el usuario en la vista y se computa la solución utilizando los métodos del modelo y finalmente se muestra el resultado de nuevo en la vista.

- Animación:
  Para poder animar la solución en la vista sin dejarla bloqueada se ejecuta en un hilo distinto. Esto se realiza dentro del método *compareActionPerformed()* que se ejecuta cuando el usuario presiona el botón de comparar, después de introducir el tamaño a probar, se realizan los cálculos en el modelo y se va graficando los puntos mediante el método *anímate()* de la vista.

VI. GUIA DE USUARIO

A continuación, se explicará cómo utilizar el programa.

Cuando se ejecuta por primera vez el programa se podrá observar la siguiente pantalla.

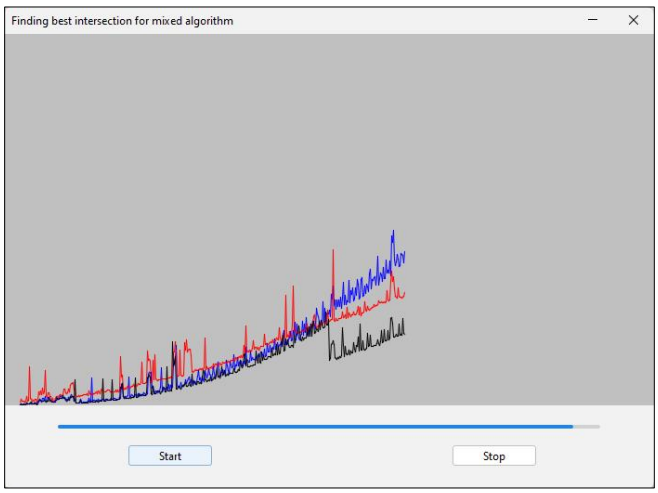

Ilustración 4

Al inicio de la ejecución, el programa calcula la N de intersección entre ambos algoritmos. Este número de corte se almacenará para posteriormente implementar el algoritmo mixto. Además, incluye tanto una barra de progreso de la

[2] Un *Event Listener* es un procedimiento o función en un programa que espera a que ocurra un evento. Ejemplos de un evento son el usuario haciendo clic o moviendo el ratón, presionando una tecla en el teclado.

ejecución como botones de Start y Stop para iniciar o parar el programa.
Una vez acaba la ejecución, el programa arroja una ventana flotante donde se muestra el punto de corte calculado.

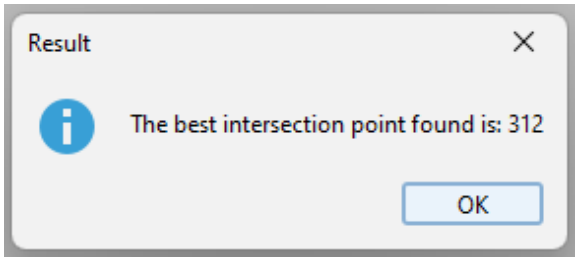

Ilustración 5

A continuación, se tiene el programa de cómputo como tal en el cual se pueden distinguir distintas zonas.

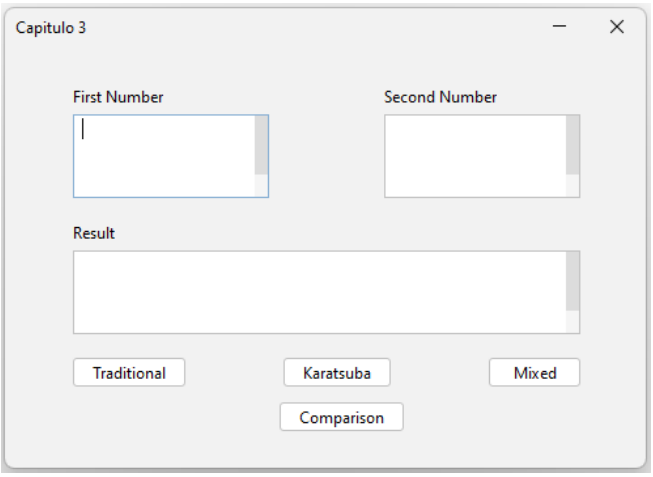

Ilustración 6

Empezando por la parte superior y de izquierda a derecha, dos zonas donde introducir los dos números a multiplicar. En la parte inmediatamente inferior un recuadro donde se mostrará el resultado al igual que el tiempo de ejecución. Y finalmente, los botones con los cuales el usuario decidirá que algoritmo de cálculo utilizar: Tradicional, Karatsuba o mixto.

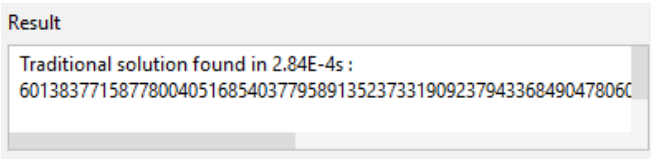

Ilustración 7

Una vez se haya computado el resultado mediante uno de los algoritmos se podrá mostrar una comparación de los distintos tiempos de cálculo mediante una gráfica. Esto se consigue al hacer click sobre el botón “Comparison”.

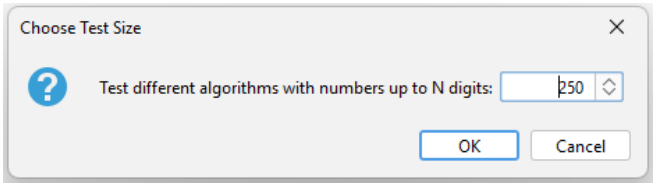

Ilustración 8

Surge este desplegable donde se puede elegir el número de dígitos con el cual se quiere comparar. Lo ideal es elegir un número superior de dígitos superior al punto de corte, así se puede observar la diferencia. Con las flechas se puede aumentar el número de 50 en 50 pero también se puede introducir de manera manual el número. Para mostrar se hace click en el botón “OK”.

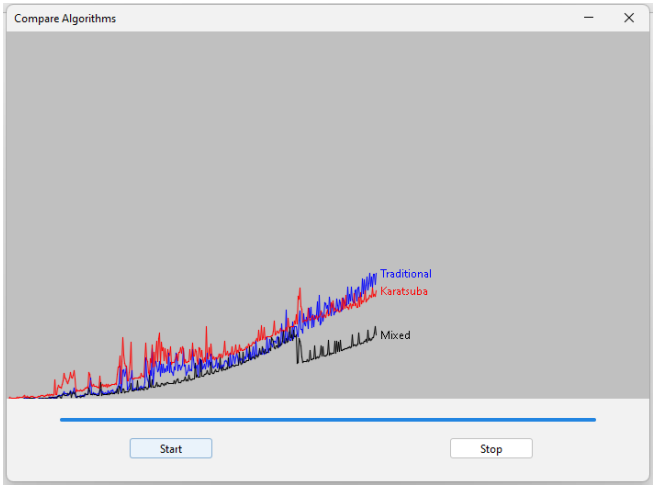

Ilustración 9

Al igual que en la primera ejecución del programa, se mostrará un gráfico con los distintos tiempos. EN azul el tradicional, en rojo el Karatsuba y en negro el mixto. Además, sigue habiendo la barra de progreso y lo botones de Start y Stop para controlar el comportamiento del programa.

VII. Estudio de rendimiento y comparacion de los algoritmos

A. Estudio de rendimiento

En este apartado se explicará el proceso que se ha llevado a cabo para hacer un estudio de rendimiento del ordenador de uno de los integrantes del equipo.
Para obtener los datos de rendimiento, se ha implementado un apartado en el programa que calcula el tiempo de cada algoritmo con un numero de entrada de tamaño N determinado.
Una vez hechos los cálculos, se guardan los resultados en un archivo externo (se hace después para no interrumpir el proceso de multiplicación).

Antes de pasar a los datos, se pasará a analizar cómo podemos obtener una medida que indique de alguna forma el rendimiento del ordenador.

Los datos provienen de dos algoritmos distintos: El algoritmo de multiplicación clásica, y el algoritmo de Karatsuba. Éstos tienen las siguientes fórmulas de complejidad asintótica:

- **Tradicional**:

$$O(CM_c * N^2) = O(N^2)$$

Ecuación 5

Se puede ver fácilmente porque el algoritmo tradicional es $O(N^2)$: se tienen dos números de longitud N, y para cada digito de uno, se hacen N multiplicaciones, así que al final surgen N*N multiplicaciones.

- **Karatsuba**:
  Vamos a ver cuál sería la complejidad asintótica del algoritmo de Karatsuba usando la notación asintótica:

$$T(n) = \begin{cases} cn^k, si\ 1 \leq n < b \\ aT\left(\frac{n}{b}\right) + cn^k, si\ n \geq b \end{cases}$$

Ecuación 6

  En nuestro caso, el caso base es O(1) (multiplicación de 1 dígito), así que *k*=0.
  En el algoritmo se hacen tres llamadas recursivas, así que *a*=3. Finalmente, dichas llamadas recursivas se hacen con una entrada de mitad de tamaño, así que *b*=2.
  Con esto, y teniendo en cuenta que

$$T(n) \in \begin{cases} \theta(n^k), si\ a < b^k \\ \theta(n^k \log n), si\ a = b^k \\ \theta\left(n^{\log_b a}\right), si\ a > b^k \end{cases}$$

Ecuación 7

  Nos encontramos en el caso $a>b^k$, ya que a=3, b=2 y k=1. Por lo tanto, la complejidad asintótica del algoritmo de Karatsuba és:

$$O(CM_k * 3 * N^{log_2 3}) = O(N^{log_2 3})$$

Ecuación 8

- **Mixto**: el algoritmo mixto hace uso del Karatsuba hasta que las partes de la llamada recursiva son menores que un umbral determinado, momento en que se llama al algoritmo tradicional. Este umbral se puede encontrar de distintas maneras, que se explicará más adelante.

Para encontrarlo, hemos estado discutiendo con unos compañeros y hemos llegado al siguiente método:

Supongamos que tenemos una entrada con N=1000 (donde N es el número de dígitos de la entrada) y umbral *u*=300. Se sabe cuántas veces se va a llamar recursivamente el Karatsuba, tantas veces como podamos dividir 1000 entre 2 hasta llegar a un número menor o igual al umbral:

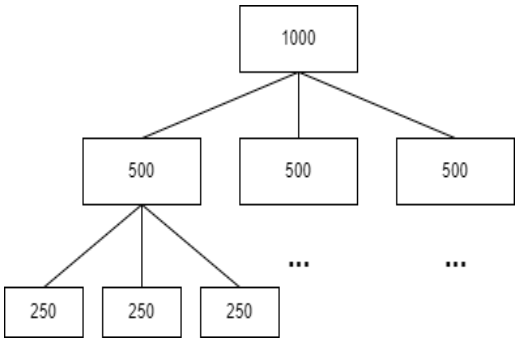

Ilustración 10

En este caso, se puede dividir dos veces.
Una vez se haya dividido dos veces, tendremos X multiplicaciones clásicas de $O(s^2)$, donde X será la cantidad de subproblemas y *s* el tamaño de los trozos. Para encontrar estas dos variables, sabemos que tendremos tantos trozos como veces hayamos llamado al Karatsuba:

$$X = 3^T$$

Ecuación 9

Y, como cada vez que se llama al Karatsuba se divide por 2 el tamaño del elemento,

$$s = \frac{N}{2^T}$$

Ecuación 10

donde N es el tamaño original de la entrada.
Por lo tanto, nos quedaría una complejidad de este estilo:

$$t = 3^T * \left(\frac{N}{2^T}\right)^2$$

Ecuación 11

Ahora nos queda resolver la *T*, que es la cantidad de veces que se llama recursivamente a Karatsuba (teniendo en cuenta que se para de llamar recursivamente cuando el tamaño de la entrada parcial es menor que el umbral).

En este caso, era igual a 2, pero se necesita la fórmula general.

Tenemos que:

$$s \leq u$$

Es decir, el tamaño de la entrada parcial en el nivel que se para la recursión tiene que ser menor o igual al umbral. Sabemos el valor de s,

$$s = \frac{N}{2^T}$$

Ecuación 12

Sustituimos s:

$$\frac{N}{2^T} \leq u$$

Ecuación 13

Pasamos a aislar la T:

$$\frac{N}{u} < 2^T$$
$$T < \log_2 \frac{N}{u}$$

Ecuación 14

Como no podemos tener umbrales fraccionarios, vamos a redondear hacia arriba el resultado del logaritmo:

$$T = \left\lceil \log_2 \frac{N}{u} \right\rceil$$

Ecuación 15

Esto es cierto cuando $N \geq u$, pero cuando N es menor que el umbral, se hacen 0 llamadas recursivas, así que podemos reescribirlo como:

$$T = \begin{cases} 0 & , si\ N < u \\ \left\lceil \log_2 \frac{N}{u} \right\rceil & , si\ N \geq u \end{cases}$$

Ecuación 16

Al tener la T, podemos pasar sustituir:

$$O(N) = \begin{cases} CM * N^2 & , si\ N < u \\ CM * 3^{\lceil \log_2 \frac{N}{u} \rceil} * \left(\frac{N}{2^{\lceil \log_2 \frac{N}{u} \rceil}} \right)^2 & , si\ N \geq u \end{cases}$$

Ecuación 17

Una vez se tienen las tres complejidades asintóticas, la medida que se busca es esa CM (Constante multiplicativa), que variará dependiendo del dispositivo y servirá como medida de rendimiento.
Al saber la fórmula de complejidad asintótica, se puede afirmar que:

$$CM_c * N^2 = t_c$$
$$\mathrm{CM_k} * \mathrm{N}^{log_2 3} = t_k$$
(Incluimos el 3 dentro de CM_k)

Ecuación 18

$$\begin{cases} CM * N^2 = t_m & , si\ N < u \\ CM * 3^{\lceil \log_2 \frac{N}{u} \rceil} * \left(\frac{N}{2^{\lceil \log_2 \frac{N}{u} \rceil}} \right)^2 = t_m & , si\ N \geq u \end{cases}$$

Ecuación 19

Por lo tanto, si se sabe el tiempo que ha tardado el algoritmo para una entrada N, se puede saber la Constante Multiplicativa para un algoritmo y ordenador en concreto. Se

Una vez se tiene esto en mente, procedemos a analizar los datos obtenidos por los distintos algoritmos.
El dispositivo usado es un portátil con un Intel i5 de séptima generación y 16GB de RAM.

El umbral para el algoritmo mixto que ha encontrado nuestro método en este dispositivo ha sido de 745.
Se han ejecutado una serie de multiplicaciones con los tres algoritmos para un rango de N desde 1 hasta 900.

Con esto, se espera tener una cantidad de datos significativa con la que obtener la medida deseada.

Se ha guardado en un archivo de texto los tiempos de cada N para cada algoritmo, y luego se han importado en una hoja de cálculo para el análisis.

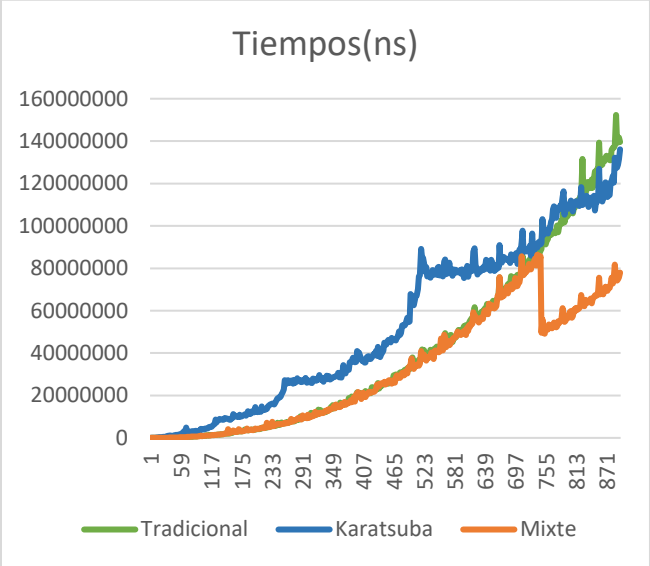

Ilustración 11

Aquí podemos observar que el tiempo aumenta con respecto a N, y llega un momento en el que Karatsuba pasa a ser más rápido que el algoritmo tradicional (el punto de corte 745 anteriormente mencionado).

También se puede observar como el algoritmo mixto imita al tradicional hasta que llega al punto de corte, momento en que tiene una mejora significativa, ya que pasa a utilizar el Karatsuba para disminuir el tamaño del problema.

Con las fórmulas descritas anteriormente, hemos calculado la constante multiplicativa para cada N, y hemos obtenido los siguientes resultados:

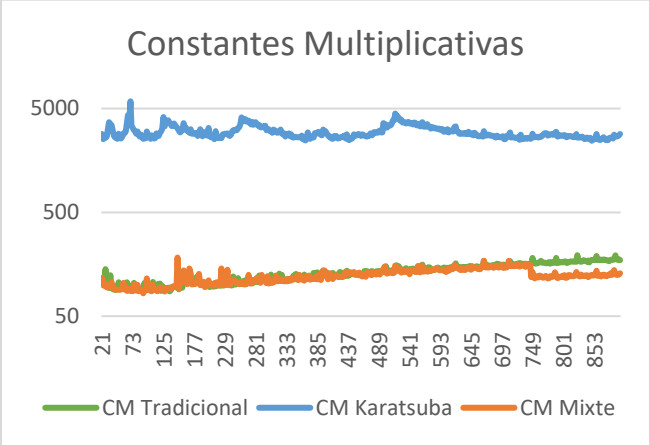

Ilustración 12

Se ha graficado sobre un eje logarítmico para que así se puedan ver el Karatsuba y los otros dos conjuntos de datos en el mismo gráfico, ya que tienen una diferencia bastante grande.

Como se puede observar, al inicio los valores son muy grandes y fluctúan mucho, pero a medida que la N aumenta, se va estabilizando esta CM.

Los valores resultantes los hemos obtenido con la media de los últimos 10 valores de cada conjunto de datos, y éstos han sido los resultados:

- Tradicional: la constante multiplicativa del algoritmo tradicional para este dispositivo en concreto es: 176,39. Estos cálculos se han hecho con el tiempo en nanosegundos. Si escalamos a segundos, se obtiene: 1,76E-08.
- Karatsuba: La CM obtenida ha sido: 2692,85, que cuando escalas el tiempo a segundos da 2,69E-07.
- Mixto: el resultado es 127,85, y al escalar el tiempo a segundos: 1,27E-08.

Por lo tanto, el algoritmo con la menor constante multiplicativa es el mixto, seguido del tradicional y finalmente el Karatsuba.

B. Estudio automático del programa

Al iniciar el programa, se hace un estudio para encontrar el punto de corte de los dos algoritmos en este dispositivo.

El estudio se basa en realizar las multiplicaciones con los dos algoritmos, e ir guardando los tiempos resultantes. Esto se hace para Ns desde 1 hasta 900.
Después, analizamos el array de resultados de la siguiente forma:

> Vamos recorriendo los dos arrays, y cuando el Karatsuba está por debajo del Tradicional 10 veces seguidas, usamos el índice actual menos diez como resultado, el punto de corte. Este resultado lo mostramos por pantalla y lo actualizamos como punto de corte en la variable del Modelo para que el algoritmo mixto la use como umbral.

C. Observaciones

Cuando se estaba haciendo el análisis de los tiempos, se ha encontrado extraño el salto súbito que da el algoritmo mixto al llegar al umbral, tal como se puede ver en la Ilustración 2, así que se probó de disminuir el umbral de forma manual y volver a obtener los datos para ver los resultados.

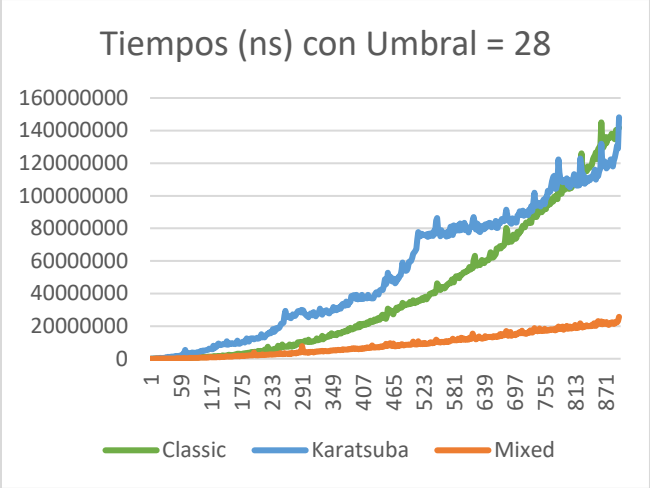

Ilustración 4

Como se puede observar, con un umbral realmente bajo se consigue que el algoritmo mixto sea muchísimo mejor. Esto es desconcertante, ya que el punto de corte de los dos algoritmos a probar sigue siendo del orden de ~700, pero cuando se trata del algoritmo mixto, es mejor un umbral realmente bajo.

Debido a esto, se ha ideado otra forma de encontrar este umbral. Básicamente, se va probando, con umbrales desde 1 hasta 300, haciendo multiplicaciones de 500 cifras.

Al final, se busca qué umbral ha tenido el menor tiempo, y ese es el resultado.
Con este algoritmo, los umbrales varían desde 20 hasta 50, y el gráfico resultante del algoritmo Mixto es siempre menor al Karatsuba y al tradicional, y sin saltos súbitos.

Al final, se dejará en el código el algoritmo para encontrar el umbral original, que miraba el punto de corte, pero se debe tener en cuenta que ese no es el óptimo al parecer.

VIII. CONCLUSIONES

Durante la realización del proyecto se han revisado distintos conceptos vistos en asignaturas anteriores, como por ejemplo la programación Swing de Java, o el uso de concurrencia y la interfaz Runnable.

Además, se han puesto en práctica conceptos teóricos como los costes computacionales y la complejidad asintótica. También se ha utilizado el concepto clave del tema actual que es el diseño de algoritmos mediante *Divide and Conquer*.

Ha sido interesante implementar todos los métodos para realizar operaciones aritméticas básicas, pero utilizando *strings* para poder tener números de tamaño ilimitado.

Además, durante los estudios de rendimiento a veces nos costaba o no entendíamos directamente algunos de los resultados obtenidos y hemos tenido que dedicar más tiempo para acabar de entender los resultados obtenidos y de donde venían.

IX. BIBLIOGRAFÍA

- MATEMÁTICA EN LA ESCUELA PRIMARIA: LOS ALGORITMOS DE LAS OPERACIONES (mariamatica.blogspot.com)
- Karatsuba Multiplication -- from Wolfram MathWorld
- Divide-and-conquer algorithm - Wikipedia
- Big O notation - Wikipedia
- Mastering recursive programming - IBM Developer