Práctica 1

Asignatura:

Programación y Estructuras de Datos Avanzadas

Iker Domingo Pérez
71126454N
UNED Madrid - Las Tablas
Curso 2016/2017

Multiplicación de grandes números

1.- Describa el esquema algorítmico utilizado y cómo se aplica al problema.

El esquema algorítmico elegido es el recomendado en el enunciado de la práctica: Divide y vencerás.

El problema al que nos enfrentamos es el del límite que tienen los tipos de datos numéricos en los lenguajes de programación para representar y operar con números.

El lenguaje de programación Java dispone de los siguientes tipos de datos numéricos con sus respectivos límites:

Tipo	**Tamaño**	**Rango**
byte	1 byte	-128 a 127
short	2 bytes	-32768 a 32767
integer	4 bytes	-2147483648 a 2147483647
long	8 bytes	-9223372036854775808 a 9223372036854775807
float	4 bytes	$\pm$ 3.4x10^{-38} a $\pm$ -3.4x10^{38}
double	8 bytes	$\pm$ 1.8x10^{-308} a $\pm$ 1.8x10^{308}

El objetivo de la práctica es escribir una aplicación en lenguaje Java que permita multiplicar dos números enteros de cualquier tamaño. Para ello vamos a implementar el algoritmo de karatsuba: procedimiento para multiplicar números basado en el esquema *divide y vencerás*.

El esquema *divide y vencerás* se basa en la idea de que un problema de gran tamaño se puede resolver descomponiéndolo en subproblemas del mismo tipo pero de menor tamaño. Concretamente el algoritmo de karatsuba parte de que un número cualquiera de *n* cifras puede representarse como la suma de dos números de *n/2* cifras. Tal y como aparece en el ejemplo del enunciado: 3221 = 32*10^2 + 21.

Desarrollando esta fórmula en el producto de dos números se llega a la expresión general:

Se tienen dos números *X* e *Y* de tamaño *n*.

Partimos *X* por la mitad obteniendo: A (parte izquierda) y B (parte derecha).

De la misma forma partimos *Y* obteniendo: C (parte izquierda) y D (parte derecha).

$$X * Y = AB * CD = A*C*10^{n} + (A*D + B*C)*10^{n/2} + B*D$$

Desarrollando el paréntesis obtenemos la versión mejorada, donde una multiplicación de dos números de tamaño n se puede resolver con tres multiplicaciones de dos números tamaño n/2.

$$X * Y = AB * CD = A*C*10^{n} + [(A+B)*(C+D) - AC - BD)*10^{n/2} + B*D$$

Se puede ver claramente que se trata de un algoritmo muy simple: incluye sumas, restas y multiplicaciones. Ahora toca encontrar una solución para implementarlo en Java.

Uno de los tipos de datos primitivos de Java que nos va a ayudar en este ejercicio es el tipo String. Al igual que los tipos de datos numéricos sí que tiene un límite: la longitud de la cadena no puede superar los $(2^{31} - 1)$ caracteres = 2.147.483.647. Esto es, el límite está por encima de los 2 billones de cifras. Comparado con los límites que tienen los tipos Integer y Long de 10 y 19 cifras respectivamente, se podría considerar que String "no tiene límite".

Así pues, vamos a trabajar con números almacenados como cadenas de caracteres. Antes de empezar a programar vamos a tener varias cosas en consideración:

- Números enteros: No se admiten puntos ni comas.
- Números positivos o negativos: Se deberá comprobar si tiene signo '+' o '-'.
- Operaciones a realizar: Suma, resta, producto.
- La suma y la resta se pueden implementar fácilmente con números de cualquier longitud guardados como String. Dígito a dígito contando el acarreo.
- Para realizar el producto tendremos que evaluar los dos números y asegurarnos que el resultado no vaya a superar el límite.
    - Hablando en 'cifras', si el límite está en 10 cifras para los integer, la forma más fácil de cumplirlo es asegurándose que el producto no supere las 9 cifras como máximo. Para ello, tirando de calculadora y de 'la cuenta de la vieja', si tenemos dos números cualquiera cuyo número de cifras combinado sea igual o inferior a 9, el producto nunca será superior a las 9 cifras.

- Se debe implementar la traza o log de operaciones. Necesitaremos herramientas para tratar con archivos y fechas (timestamp).

Ya podemos empezar a escribir algunos de los métodos que vamos a utilizar. Pero antes, vamos a definir las variables principales:

Variables:

String x, String y

Van a guardar los números que se desea multiplicar.

String A, B, C, D, AC, K, BD, resultado

Diferentes strings para llevar control de las operaciones.

boolean negX, boolean negY

Van a indicar el signo de cada número. True para negativo, False para positivo.

int m, n

Para dividir los números en subcadenas. Para números de longitud par m=n, para longitud impar m>n. 'm' va a indicar dónde se parte el número. 'n' para posteriormente multiplicar la primera parte por 10^n o $10^{n/2}$. Al ser Strings será suficiente con concatenar tantos ceros como sean necesarios a la derecha.

int index

Contador que lleva el número de iteraciones del algoritmo.

boolean traza

Al pasar '-t' como argumento se activa para guardar un log de operaciones.

FileWriter log

Para escribir en el archivo del log

Scanner s

Para leer datos de entrada desde el terminal o desde archivo (con FileReader)

Clases:

```
java.util.Scanner;                  // Lectura de datos desde archivo o standard input
java.io.File;                       // Tratamiento de archivos
java.io.FileWriter;                 // Escritura en archivos
java.io.FileReader;                 // Lectura de archivos
java.io.FileNotFoundException;      // Tratamiento de excepciones
java.io.IOException;                        // Tratamiento de excepciones
java.text.SimpleDateFormat;         // Formato de fecha
java.util.Date;                     // Obtener timestamp
java.io.InputStream;                // Obtener recurso del archivo jar
```

Métodos:

- public static void main (String args[])

Método principal. Prepara el archivo de ayuda para imprimirlo por pantalla si se pasa '-h' como argumento o si se pasan más argumentos de los esperados.

Analiza los argumentos pasados. Inicializa el log creando una carpeta '*multiplica_logs*' y un archivo con timestamp al milisegundo.

Llama a uno de los 3 posibles métodos con los que se ejecuta el programa:

- **Input**() para leer y escribir por el terminal

- **InputFile**(args[m]) para leer los números desde un archivo (args[m]) y obtener el producto por el terminal,

- **InOutFile**(args[m], args[n]) donde la entrada y la salida es por archivo.

```
public static void main(String args[]) throws FileNotFoundException, IOException {
    InputStream is = multiplica.class.getResourceAsStream("Readme");
    Scanner s = new Scanner (is, "UTF-8");
    String h = s.useDelimiter("\\A").next();;
    if (args.length == 0) Input();
    for (String element:args) {
        if (element.equals("-h")||args.length>3) {
            System.out.println(h);
            System.exit(0);
        }
    }
    for (String element:args) {
        if (element.equals("-t")) {
            traza = true;
            new File("Multiplica_logs").mkdir();
            String d = new SimpleDateFormat("yyyyMMddhhmmssSSS'.log'").format(new Date());
            File f = new File("Multiplica_logs", d);
            if (f.exists()) {
                System.out.println("Se ha producido un error...");
                System.exit(0);
            }
            log = new FileWriter(f);
            if (args.length == 1) Input();
            if (args.length == 2) InputFile(args[1]);
            if (args.length == 3) InOutFile(args[1], args[2]);
            System.out.println(h);
            System.exit(0);
        }
        if (args.length == 1) InputFile(args[0]);
        if (args.length == 2) InOutFile(args[0], args[1]);
        System.out.println(h);
        System.exit(0);
    }
}
```

Nota: *Para mejorar la visualización del código presentado a lo largo de este documento se han eliminado los comentarios y los mensajes de error se han acortado. El código completo se encuentra en la carpeta /Fuente*

- public static boolean Comprobar (String x)
  ejemplo: negX = Comprobar(x); negY = Comprobar(y);

Este método cumple dos funciones:

- Revisa si el número tiene signo, es decir, si empieza con el carácter '+' o '-'. Si el número es negativo devuelve true, y ese true quedará almacenado en negX o negY. Devuelve false si el número es positivo.
- Antes de devolver true o false, elimina el signo del número si lo hubiese y comprueba dígito a dígito que la cadena restante es un número válido. Si se encuentra un carácter no esperado se lanza un mensaje de error y finaliza el programa.

```
public static boolean Comprobar(String x) {
    boolean numero_valido, signo;
    if (x.startsWith("-")) {
        x = x.substring(1);
        signo = true;
    } else if (x.startsWith("+")) {
        x = x.substring(1);
        signo = false;
    } else signo = false;
    for (int i=0; i<x.length(); i++) {
        numero_valido = false;
        if (Character.isDigit(x.charAt(i))) {
            numero_valido = true;
            continue;
        }
        if (numero_valido==false) {
            System.out.println("El n\u00famero introducido no es v\u00e1lido. Contiene el caracter " + x.charAt(i));
            System.exit(0);
        }
    }
    return signo;
}
```

- public static String Signo (String x)
  ejemplo: x = Signo(x); y = Signo(y);

Como en Java no es posible pasar parámetros por referencia hace falta crear otro método para eliminar el signo de las cadenas. Consiste sólo en una cláusula if-else, pero aun así se ahorran líneas de código repetido.

```
public static String Signo(String x) {
    if (x.startsWith("-")) {
        x = x.replace("-", "");
    } else if (x.startsWith("+")) {
        x = x.replace("+", "");
    }
    return x;
}
```

- public static String Igualar(String x, String y)

Método para igualar en longitud dos cadenas de caracteres añadiendo ceros a la izquierda. Antes de la llamada a este método se sabrá cuál de las dos es la cadena más corta y será la que se modifique.

Ejemplo: ***if*** *(x.length() < y.length()) x = Igualar (x, y);* ***else*** *y = Igualar (y, x);*

```
public static String Igualar(String x, String y) {
    int aux = y.length() - x.length();
    for (int i=1; i<=aux; i++) {
    x = "0".concat(x);
    }
    return x;
}
```

- public static boolean NumeroCero (String x)

Método que revisa si una cadena de caracteres está compuesta únicamente por ceros. A lo largo de la escritura del programa se ha visto muchas situaciones donde la multiplicación de grandes números puede implicar multiplicación de dos números de muy diferente tamaño. Este algoritmo iguala en longitud estos números añadiendo ceros, por lo que se dan muchos casos en que tras dividir un número en dos cadenas, una de ellas es todo ceros. Al verificar esto se pueden ahorrar muchas llamadas recursivas al algoritmo.

Por ello, al inicio de cada iteración la primera comprobación es

if *(NumeroCero(x)==****true*** *|| NumeroCero(y)==****true****)* ***return*** *"0";*

```
public static boolean NumeroCero (String x) {
    boolean numero_cero = false;
    for (int i=0; i<x.length(); i++) {
        if (x.charAt(i)=='0') {
            numero_cero = true;
        } else {
            numero_cero = false;
            break;
        }
    }
        return numero_cero;
}
```

- public static String Suma (String x, String y)
  Ejemplo: String suma1 = Suma(A, B);

Suma dos números guardados como String. El resultado lo devuelve también en forma de String.

En primer lugar iguala ambos números en número de cifras. Acto seguido entra en un bucle ***for*** donde, de derecha a izquierda, evalúa como Integer cada carácter y los suma, contando el acarreo.

```
public static String Suma (String x, String y) {
    String resultado = "";
    boolean acarreo = false;
    if (x.length() < y.length()) {
        x = Igualar(x, y);
    } else if (x.length() > y.length()) {
        y = Igualar(y, x);
    }
    for (int i = x.length() - 1; i >= 0; i--) {
        int j = Integer.parseInt(String.valueOf(x.charAt(i)));
        int k = Integer.parseInt(String.valueOf(y.charAt(i)));
        int m = j+k;
        if (acarreo == true) m++;
        if (m>=10) {
            m = m % 10;
            acarreo = true;
        } else {
            acarreo = false;
        }
        resultado = Integer.toString(m).concat(resultado);
    }
        if (acarreo == true) {
        return "1".concat(resultado);
    } else {
        return resultado;
    }
}
```

- public static String Resta (String x, String y)

Similar al método para Sumar. Iguala en longitud dos números, y luego de derecha a izquierda evalúa cada carácter como integer y los resta.

La diferencia es que antes de llamar a este método sabemos de antemano que el número *x* va a ser mucho mayor (del orden de $10^{n/2}$) que el número *y*, por lo que no hace falta tener en cuenta el acarreo en el último dígito ni los posibles números negativos.

```
public static String Resta (String x, String y) {
    String resultado = "";
    int j, k, m;
    boolean acarreo = false;
    if (y.length() < x.length()) y = Igualar(y, x);
    for (int i = x.length() - 1; i >= 0; i--) {
        j = Integer.parseInt(String.valueOf(x.charAt(i)));
        k = Integer.parseInt(String.valueOf(y.charAt(i)));
        if (acarreo == true) {
            m = j-k-1;
         } else {
            m = j-k;
         }
        if (m < 0) {
            acarreo = true;
            m += 10;
        } else {
            acarreo = false;
        }
        resultado = Integer.toString(m).concat(resultado);
    }
    return resultado;
}
```

- public static void Input()

Lee dos números introducidos por el terminal, verifica que son válidos, apunta el signo e inicia el algoritmo. Devuelve el resultado por pantalla. Código trivial.

- public static void InputFile (String file)

Lee dos números desde un archivo, verifica que son válidos, apunta el signo e inicia el algoritmo. Devuelve el resultado por pantalla. Código compartido con el siguiente método.

- public static void InOutFile (String FileIn, String FileOut)

Lee dos números desde un archivo pasado como argumento. Para ello:

- Comprueba que el archivo existe. Si no, finaliza el programa.
- Comprueba que el archivo no está protegido contra lectura. Si no, finaliza.
- Inicializa un Scanner tipo FileReader. Para leer cada número entra en un bucle do-while donde estará leyendo con el método next() mientras la cadena siga vacía.
- Utilizamos replaceFirst("^ *", "") para ir eliminando posibles espacios en blanco a la izquierda y para ignorar líneas vacías.
- Comprueba que el número leído es válido y apunta el signo.
- Comprueba que no haya más contenido dentro del archivo tras leer los dos números.

El resultado lo escribe en un archivo pasado como argumento. Para ello:

- Comprueba que el archivo de destino no exista previamente.
- Crea el archivo y escribe en él únicamente el resultado de la llamada al método central del programa: karatsuba (x, y)

```
public static void InOutFile(String FileIn, String FileOut) throws FileNotFoundException, IOException {
    String x, y;
    File in = new File(FileIn);
    if (!in.isFile()) {
        System.out.println("Se ha producido un error. El archivo de entrada '" + FileIn + "' no existe.");
        System.exit(0);
    } else if (!in.canRead()) {
        System.out.println("Se ha producido un error. El archivo de entrada '" + FileIn + "' est\u00e1 protegido y no se puede leer.");
        System.exit(0);
    } else if (FileIn.length()==0) {
        System.out.println("Se ha producido un error. El archivo de entrada '" + FileIn + "' no contiene datos.");
        System.exit(0);
    }
    File out = new File(FileOut);
    if (out.exists()) {
        System.out.println("Se ha producido un error. El archivo de salida '" + FileOut + "' ya existe.");
        System.exit(0);
    }
    FileWriter f = new FileWriter(out);
    Scanner s = new Scanner(new FileReader(in));
    do {
        x = s.next().replaceFirst("^ *", "");
    } while (x.isEmpty());
    negX = Comprobar(x);
    if (!s.hasNext()) {
        System.out.println("Se ha producido un error. El archivo de entrada '" + FileIn + "' s\u00f3lo contiene un n\u00famero");
        System.exit(0);
    }
    do {
        y = s.next().replaceFirst("^ *", "");
    } while (y.isEmpty());
    negY = Comprobar(y);
    if (s.hasNext()) {
        System.out.println("Se ha producido un error. El archivo de entrada '"+FileIn+"' contiene m\u00e1s de dos n\u00fameros.");
        System.exit(0);
    }
    x = Signo(x);
    y = Signo(y);
    if (negX==negY) {
        f.write(karatsuba(x, y));
        f.close();
    } else {
        f.write("-"+karatsuba(x, y));
        f.close();
    }
    if (traza==true) log.close();
    System.exit(0);
}
```

- public static String karatsuba(String x, String y)

Método central del algoritmo.

- Inicia las variables necesarias para trabajar con los números.
- Utiliza SimpleDateFormat.format(new Date()) en cada línea que escribe en el log para obtener un timestamp actualizado. En un primer momento se guardaba el timestamp en un String al inicio de cada iteración, pero se ha comprobado que la primera llamada al algoritmo es también la última en terminar, por lo que el timestamp de la primera línea del log era el mismo que el de la última línea.
- Solución simple '0': Comprueba si alguno de los números es una cadena de ceros
- Solución simple x*y: Comprueba si los dos números juntos suman 9 cifras o menos.
- Compara la longitud de los dos números y los iguala si es necesario.
- Divide cada número en dos subcadenas. x = A, B y = C, D
- AB * CD = [AC * 10^n] + [K * 10^(n/2)] + [BD] donde K = [(A+B)(C+D) - AC - BD]
- llamada recursiva #1: String AC = karatsuba (A, C);
- llamada recursiva #2: String BD = karatsuba (B, D);
- llamada recursiva #3:
  - suma1 = Suma(A, B); suma2 = Suma(C, D);
  - mult = karatsuba(suma1, suma2);
  - String K = Resta(Resta(mult, AC), BD);
- La parte AC se multiplica por 10^{2n} (en este programa 'n' es la mitad redondeada a la baja del tamaño de los números) y la parte central K se multiplica por 10^{n}.
- Se suman las 3 partes, se eliminan los ceros a la izquierda y se obtiene el resultado=Suma(Suma(AC, K), BD).replaceFirst("^0*", "");

```
public static String karatsuba(String x, String y) throws IOException {
    String A, B, C, D, resultado;
    int m, n;
    SimpleDateFormat sdfLog = new SimpleDateFormat("'Multiplica_['yyyy-MM-dd_hh:mm:ss.SSS']'");
    index++;
    if (NumeroCero(x)==true || NumeroCero(y)==true) return "0";
    if (x.length() + y.length() <= 9) return Integer.toString(Integer.parseInt(x)*Integer.parseInt(y));
    if (x.length() < y.length()) x = Igualar(x, y);
    if (x.length() > y.length()) y = Igualar(y, x);
    m = x.length()/2 + x.length()%2;
    n = x.length()/2;
    A = x.substring(0, m);
    B = x.substring(m);
    C = y.substring(0, m);
    D = y.substring(m);
    String AC = karatsuba(A, C);
    String BD = karatsuba(B, D);
    String suma1 = Suma(A, B);
    String suma2 = Suma(C, D);
    String mult = karatsuba(suma1, suma2);
    String K = Resta(Resta(mult, AC), BD);
    for (int i=1;i<=n*2;i++) {  AC = AC.concat("0");  }
    for (int i=1;i<=n;i++) {    K = K.concat("0");    }
    resultado = Suma(Suma(AC, K), BD).replaceFirst("^0*", "");
    return resultado;
}
```

Tratamiento de excepciones

Por imposición del lenguaje Java se han importado las excepciones IOException y FileNotFoundException. Su ausencia provoca errores a la hora de compilación en los métodos donde es posible que aparezcan estas excepciones.

Sin embargo en este programa no se van a utilizar. Todos los puntos susceptibles de capturar una excepción están bajo control y finalizan el programa con un mensaje de error describiendo el problema encontrado.

De esta forma, el programa finaliza si se cumple alguno de los siguientes casos:

- Se ha pasado '-h' como argumento.
    - Imprime el mensaje de ayuda y finaliza.
- Se han pasado más argumentos de los esperados.
    - Imprime el mensaje de ayuda y finaliza.
- Ya existe un archivo de <date>.log con el mismo nombre.
    - Informa del error y finaliza.
- Uno de los números contiene un carácter que no isDigit().
    - Informa del error y finaliza.
- El archivo de entrada no existe o está protegido contra lectura.
    - Informa del error y finaliza.
- El archivo de entrada contiene más de dos números.
    - Informa del error y finaliza.
- El archivo de salida ya existe.
    - Informa del error y finaliza.
- El programa no ha encontrado errores.
    - Devuelve el resultado y finaliza.

2.- Analice el coste computacional y espacial del algoritmo teniendo en cuenta el coste de realizar la optimización indicada en el enunciado (pasar de 4 a 3 multiplicaciones)

El algoritmo de multiplicación clásico tiene un coste de n^2 multiplicaciones de un dígito.

El algoritmo de karatsuba, de acuerdo con diversas fuentes, tiene un coste de $n^{\log_2 3}$ o lo que es lo mismo $n^{1'585}$ donde *n* determina la longitud de los números.

Poniendo como ejemplo n=10, una multiplicación clásica requiere 100 multiplicaciones, mientras que con el método de karatsuba sólo se necesitan 38'45.

Hasta aquí la teoría. Ahora veamos la práctica. Como se ha comentado anteriormente, he incorporado una variable *index* para llevar la cuenta de las veces que se llama al algoritmo. Los resultados son los siguientes:

Gráfico 1: Solución simple habilitada: multiplicar sólo si suman menos de 9 cifras o devolver cero si uno de los números es igual a cero. 30 muestras, de n=1 a n=31.
El algoritmo con 3 multiplicaciones presenta una evolución creciente uniforme. El algoritmo con 4 multiplicaciones tiene forma escalonada con saltos muy bruscos. Los resultados no se ajustan a la teoría, principalmente por la incorporación de la solución simple.
Teoría, para n=10, 38'45 iteraciones. Práctica, para n=10, 12 iteraciones (x3) ó 20 (x4)

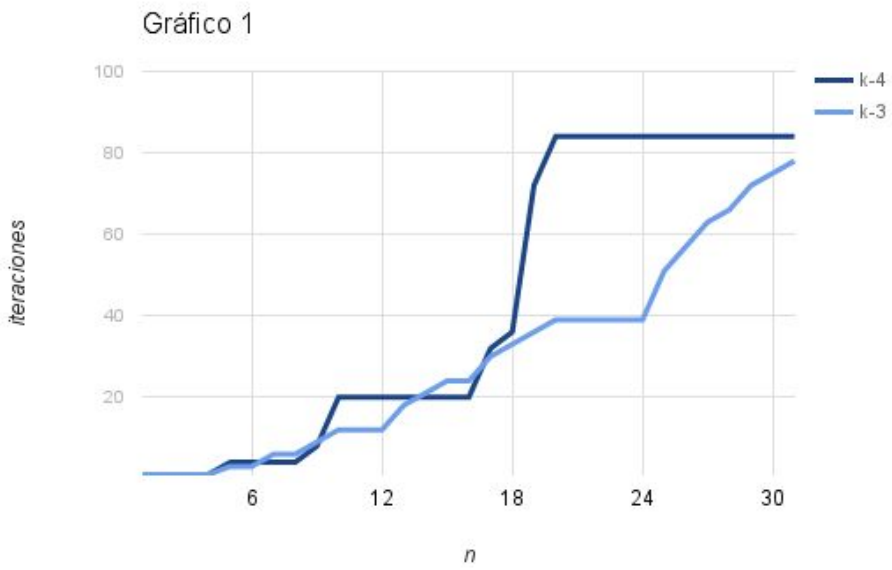


Gráfico 2: 35 muestras desde n=1 hasta n=35. Solución simple reducida a: multiplicar sólo si son números de 2 dígitos. Al igual que antes el algoritmo de 3 multiplicaciones presenta una evolución uniforme mientras que el de 4 multiplicaciones lo hace de forma escalonada. El resultado se ajusta más a la teoría, para n=10 se obtienen 51 y 68 iteraciones respectivamente.

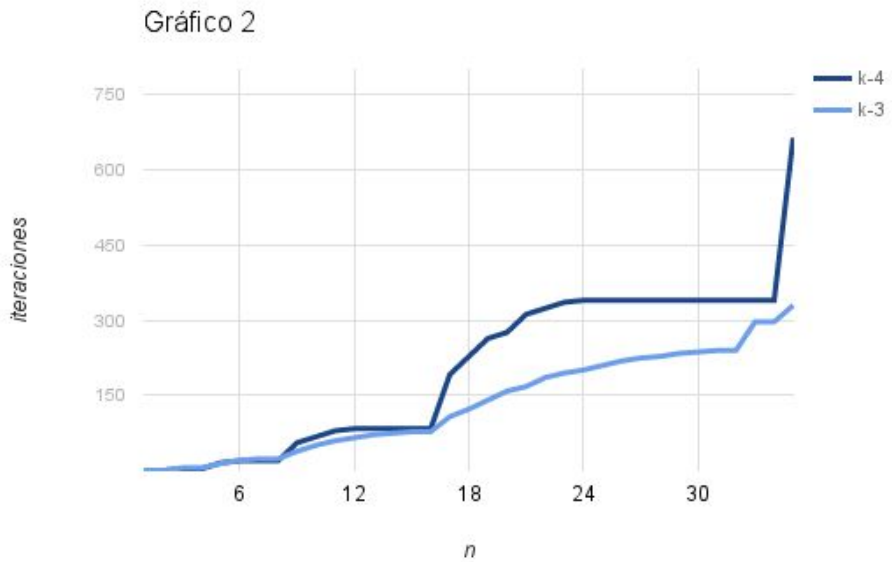
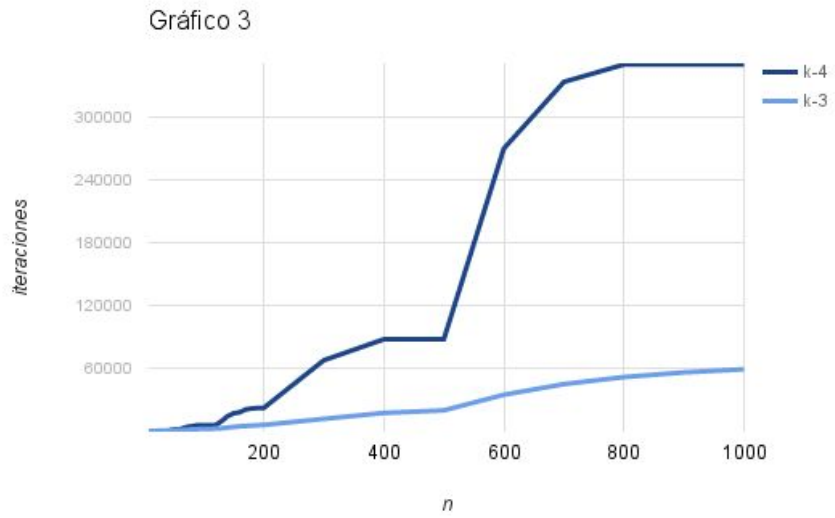


Gráfico 3: 30 muestras. De 10 en 10 desde 0 hasta 200 y de 100 en 100 desde 200 a 1000. Aquí se aprecia la gran diferencia entre la primera versión con 4 multiplicaciones y la versión mejorada. Para n=1000, hacen falta 58846 iteraciones en la versión mejorada frente a 349525 (6 veces más). Al igual que con los anteriores gráficos, se aprecia la forma escalonada de uno frente a la forma lineal del otro.



3.- Exponga alternativas al esquema utilizado si las hay, y compare su coste con el de la solución realizada.

La principal alternativa al método utilizado la hemos encontrado navegando por el código del tipo BigInteger: El algoritmo 3-way Toom-Cook. El tipo BigInteger es diferente de los tipos de datos primitivos, pero con un simple comando es capaz de multiplicar dos números cualquiera y, como veremos más adelante, de una forma más eficiente. BigInteger trata los números como strings, al igual que hemos hecho a mano a lo largo de la práctica. El límite del tipo BigInteger es 2^{31}-1, es decir, el mismo límite que el tipo String.

El siguiente fragmento de código es capaz de multiplicar dos números de cualquier tamaño. El producto de números de 1000, 2000 y 4000 cifras coincide con el resultado obtenido con esos mismos números con el programa multiplica.jar.

```
import java.math.BigInteger;
import java.util.Scanner;
public class biginteger {
    public static void main (String args[]) {
        BigInteger x, y;
        Scanner input = new Scanner(System.in);
        System.out.print("Introduzca el primer n\u00famero: ");
        x = new BigInteger(input.next());
        System.out.print("Introduzca el segundo n\u00famero: ");
        y = new BigInteger(input.next());
        System.out.println("Resultado: " +x.multiply(y));
    }
}
```

El método **multiply** del tipo BigInteger no utiliza un sólo algoritmo, sino que elige de qué forma va a multiplicar los números tras evaluarlos:

```
public BigInteger multiply(BigInteger val) {
        if (val.signum == 0 || signum == 0) return ZERO;
        int xlen = mag.length;
        if (val == this && xlen > MULTIPLY_SQUARE_THRESHOLD)  return square();
        int ylen = val.mag.length;
        if ((xlen < KARATSUBA_THRESHOLD) || (ylen < KARATSUBA_TH  RESHOLD)) {
            int resultSign = signum == val.signum ? 1 : -1;
            if (val.mag.length == 1) {
                return multiplyByInt(mag,val.mag[0], resultSign);
            }
            if (mag.length == 1) {
                return multiplyByInt(val.mag,mag[0], resultSign);
            }
            int[] result =  multiplyToLen(mag, xlen, val.mag, ylen, null);
            result = trustedStripLeadingZeroInts(result);
            return new BigInteger(result, resultSign);
        } else {
            if ((xlen < TOOM_COOK_THRESHOLD) && (ylen < TOOM_COO  K_THRESHOLD)) {
                return multiplyKaratsuba(this, val);
            } else {
                return multiplyToomCook3(this, val);
            }
        }
    }
```

Al igual que en nuestro programa multiplica.jar, lo primero que hace es comprobar que alguno de los números sea igual a cero. Después compara la longitud de los números con las siguientes constantes globales (**Nota:** *a partir de este punto cuando hablemos de longitud de números estaremos hablando en unidades de int, donde 1 int = 9.65 números decimales*)

```
KARATSUBA_THRESHOLD = 80;
TOOM_COOK_THRESHOLD = 240;
KARATSUBA_SQUARE_THRESHOLD = 128;
```

Si los dos números son iguales (x*x = x^2) y su longitud es superior a `KARATSUBA_SQUARE_THRESHOLD = 128;` los envía a square(), que a su vez evalúa los números de nuevo y llama a otros métodos derivados de ToomCook y Karatsuba para realizar el cuadrado del número.

Si la longitud es inferior a `KARATSUBA_THRESHOLD = 80;` los multiplica con int o long.

Para números entre `KARATSUBA_THRESHOLD = 80;` y `TOOM_COOK_THRESHOLD = 240;` los multiplica con el algoritmo de karatsuba:

```
private static BigInteger multiplyKaratsuba(BigInteger x, BigInteger y) {
        int xlen = x.mag.length;
        int ylen = y.mag.length;
        int half = (Math.max(xlen, ylen)+1) / 2;
        BigInteger xl = x.getLower(half);
        BigInteger xh = x.getUpper(half);
        BigInteger yl = y.getLower(half);
        BigInteger yh = y.getUpper(half);
```

```
        BigInteger p1 = xh.multiply(yh);
        BigInteger p2 = xl.multiply(yl);
        BigInteger p3 = xh.add(xl).multiply(yh.add(yl));
       BigInteger result =
p1.shiftLeft(32*half).add(p3.subtract(p1).subtract(p2)).shiftLeft(32*half).add(p2);
        if (x.signum != y.signum) {
            return result.negate();
        } else {
            return result;
        }
    }
```

Se puede apreciar a simple vista que tiene exáctamente la misma estructura que nuestro algoritmo “mejorado”, el de las 3 multiplicaciones. La parte recursiva la envía de vuelta a multiply() donde se vuelven a evaluar los números.

Por último, si los números son superiores a `TOOM_COOK_THRESHOLD = 240;` se llama al método `multiplyToomCook3(BigInteger a, BigInteger b).`

El algoritmo 3-way Toom Cook también está basado en el esquema divide y vencerás. El coste es de $\Theta(n^{1.465})$ frente a $\Theta(n^{1.585})$ de karatsuba. La técnica consiste en dividir cada número en 3 partes y realizar 5 multiplicaciones. Se ha comprobado que el coste es inferior al algoritmo de Karatsuba cuando los números superan una determinada longitud. En BigInteger este límite está fijado como hemos visto antes en

```
TOOM_COOK_THRESHOLD = 240.
```

```
private static BigInteger multiplyToomCook3(BigInteger a, BigInteger b) {
        int alen = a.mag.length;
        int blen = b.mag.length;
        int largest = Math.max(alen, blen);
        int k = (largest+2)/3;
        int r = largest - 2*k;
        BigInteger a0, a1, a2, b0, b1, b2;
        a2 = a.getToomSlice(k, r, 0, largest);
        a1 = a.getToomSlice(k, r, 1, largest);
        a0 = a.getToomSlice(k, r, 2, largest);
        b2 = b.getToomSlice(k, r, 0, largest);
        b1 = b.getToomSlice(k, r, 1, largest);
        b0 = b.getToomSlice(k, r, 2, largest);
        BigInteger v0, v1, v2, vm1, vinf, t1, t2, tm1, da1, db1;
        v0 = a0.multiply(b0);
        da1 = a2.add(a0);
        db1 = b2.add(b0);
        vm1 = da1.subtract(a1).multiply(db1.subtract(b1));
        da1 = da1.add(a1);
        db1 = db1.add(b1);
        v1 = da1.multiply(db1);
        v2 = da1.add(a2).shiftLeft(1).subtract(a0).multiply(
             db1.add(b2).shiftLeft(1).subtract(b0));
        vinf = a2.multiply(b2);
        t2 = v2.subtract(vm1).exactDivideBy3();
        tm1 = v1.subtract(vm1).shiftRight(1);
```

```
            t1 = v1.subtract(v0);
            t2 = t2.subtract(t1).shiftRight(1);
            t1 = t1.subtract(tm1).subtract(vinf);
            t2 = t2.subtract(vinf.shiftLeft(1));
            tm1 = tm1.subtract(t2);
            int ss = k*32;
            BigInteger result =
vinf.shiftLeft(ss).add(t2).shiftLeft(ss).add(t1).shiftLeft(ss).add(tm1).shiftLeft(s
s).add(v0);
            if (a.signum != b.signum) {
                return result.negate();
            } else {
                return result;
            }
        }
```

En conclusión, al utilizar el tipo BigInteger se evalúan los números introducidos para realizar la operación que tenga el coste más eficiente. Sin embargo no utiliza un sólo algoritmo, sino que utiliza varios. Si sólo podemos utilizar un algoritmo en nuestro programa habría que estudiar la longitud media de los números que vamos a multiplicar, para así poder escoger la opción con el mejor coste.

Para números superiores a las 240 ints (2316 números decimales) el algoritmo Toom Cook es el ganador, para cantidades inferiores: Karatsuba.

El algoritmo de Karatsuba fue descubierto en 1960 y publicado en 1962. El algoritmo Toom-Cook fue descubierto en 1963 y publicado en 1966. En 1971 se descubrió un tercer método alternativo: el algoritmo de Schönhage–Strassen.

Este algoritmo consiste en colocar un número encima del otro, multiplicar dígito a dígito, como en el colegio, pero luego se colocan los números en filas, y se suman por columnas

```
          1   2   3
      ×   4   5   6
    ---------------
          6  12  18
      5  10  15
  4   8  12
-------------------
  4  13  28  27  18
```

En este punto se coge el número de la derecha del todo, si son dos dígitos nos quedamos con el último (el 8) y el otro lo sumamos al siguiente grupo: 27+1 = 28. Nos volvemos a quedar con el número de la derecha (el 8) y el otro lo sumamos al siguiente grupo (28+2=30) y así hasta el final (nos quedamos con el 0, el 3 lo sumamos a 13+3=16, nos quedamos con el 6 y el 1 lo sumamos al 4+1=5) 123*456=56088.

A mi modo de ver, el coste de este algoritmo en número de multiplicaciones es $\Theta(n \cdot m)$, siendo m la longitud del segundo número. Si consultamos la wikipedia, el coste real es de $\Theta(n \cdot (\log(n) \cdot \log(\log(n))))$.

En el maravilloso github he encontrado un repositorio del BigInteger.java modificado que incluye un **multiplySchoenhageStrassen**() y puntos de evaluación en **multiply**() donde se implica este nuevo método. El código a continuación:

```
    private static int[] multiplySchoenhageStrassen(int[] a, int[] b, int numThreads)
{
        boolean square = a == b;
        int M = Math.max(a.length*32, b.length*32);
        int m = 32 - Integer.numberOfLeadingZeros(2*M-1-1);
        int n = m/2 + 1;
        boolean even = m%2 == 0;
        int numPieces = even ? 1<<n : 1<<(n+1);
        int pieceSize = 1 << (n-1-5);   // in ints
        int numPiecesA = (a.length+pieceSize) / pieceSize;
        int[] u = new int[(numPiecesA*(3*n+5)+31)/32];
        int uBitLength = 0;
        for (int i=0; i<numPiecesA && i*pieceSize<a.length; i++) {
            appendBits(u, uBitLength, a, i*pieceSize, n+2);
            uBitLength += 3*n+5;
        }
        int[] gamma;
        if (square)
            gamma = new BigInteger(1, u).square(numThreads).mag;   // gamma = u * u
        else {
            int numPiecesB = (b.length+pieceSize) / pieceSize;
            int[] v = new int[(numPiecesB*(3*n+5)+31)/32];
            int vBitLength = 0;
            for (int i=0; i<numPiecesB && i*pieceSize<b.length; i++) {
                appendBits(v, vBitLength, b, i*pieceSize, n+2);
                vBitLength += 3*n+5;
            }
          gamma = new BigInteger(1, u).multiply(new BigInteger(1, v), numThreads).mag;
        }
        int[][] gammai = splitBits(gamma, 3*n+5);
        int halfNumPcs = numPieces / 2;
        int[][] zi = new int[gammai.length][];
        for (int i=0; i<gammai.length; i++)
            zi[i] = gammai[i];
        for (int i=0; i<gammai.length-halfNumPcs; i++)
            subModPow2(zi[i], gammai[i+halfNumPcs], n+2);
        for (int i=0; i<gammai.length-2*halfNumPcs; i++)
            addModPow2(zi[i], gammai[i+2*halfNumPcs], n+2);
        for (int i=0; i<gammai.length-3*halfNumPcs; i++)
            subModPow2(zi[i], gammai[i+3*halfNumPcs], n+2);
        MutableModFn[] ai = split(a, halfNumPcs, pieceSize, (1<<(n-6))+1);
        MutableModFn[] bi = null;
        if (!square)
            bi = split(b, halfNumPcs, pieceSize, (1<<(n-6))+1);
        int omega = even ? 4 : 2;
        if (square) {
            dft(ai, omega, numThreads);
            squareElements(ai, numThreads);
        }
        else {
            dft(ai, omega, numThreads);
            dft(bi, omega, numThreads);
            multiplyElements(ai, bi, numThreads);
        }
        MutableModFn[] c = ai;
        idft(c, omega, numThreads);
        int[][] cInt = toIntArray(c);
        int[] z = new int[(1<<(m-5))+1];
        for (int i=0; i<halfNumPcs; i++) {
            int[] eta = i>=zi.length ? new int[(n+2+31)/32] : zi[i];
            subModPow2(eta, cInt[i], n+2);
```

```
            int shift = i*(1<<(n-1-5));   // assume n>=6
            addShifted(z, cInt[i], shift);
            addShifted(z, eta, shift);
            addShifted(z, eta, shift+(1<<(n-5)));
        }
        MutableModFn.reduce(z);   // assume m>=5
        return z;
    }
```

A simple vista es más complejo que las alternativas de Karatsuba o Toom-Cook, sin embargo tiene sus usos. En el método **shouldUseSchoenhageStrassen**(int length) se explican los casos en los que Schönhage–Strassen (SS para abreviar) es más eficiente:

Para máquinas de 64bits, el algoritmo SS es más lento que Toom-Cook para cantidades inferiores a 2000 ints, pero es más eficiente en los intervalos {2000 y 2048}, {3120 y 4096}, {5232 y 8192} y por encima de 8720 ints (~84,000 números decimales).

Para máquinas de 32 bits, SS es más lento que Toom-Cook por debajo de 3500 ints, más rápido en los intervalos {3504 y 4096}, {5968 y 8192} y por encima de los 9648 ints (~92,900 números).

4.- Conclusiones

Antes de empezar a realizar la práctica sabía de la existencia del tipo BigInteger, pero me negué a utilizarlo. Ni siquiera investigué en qué consistía. Me documenté sobre el algoritmo de karatsuba y lo escribí a mi manera utilizando Strings. Una vez tuve el programa escrito y funcionando fue cuando abrí BigInteger.java y miré en sus tripas. Siendo realistas, un programa sencillo que lea un archivo, lo guarde como tipo BigInteger y llame a x.multiply(y) va a suponer un programa mucho más eficiente que el actual.

La realización de esta práctica me ha servido de mucha ayuda para comprender el propio lenguaje Java, los algoritmos, la programación estructurada, y no me arrepiento de ninguna de las incontables horas que me he tirado frente a la pantalla intentando comprender algunos conceptos.

El punto más débil de este proyecto es el log. No he conseguido dar con la clave para sacar la traza de operaciones de una forma limpia y ordenada. Al menos no de la forma que a mí me gustaría encontrarme un log. El formato actual es lo mejor que he podido alcanzar. Me gustaría alguna recomendación para mejorarlo.

A modo de resumen, uno de los principios de la programación es “no inventes nada que ya esté inventado”. Si tenemos que multiplicar grandes números, utilicemos BigInteger. Pero tampoco hay que olvidar que esto es la universidad y aquí hemos venido a aprender.