UNIVERSIDAD DIEGO PORTALES

Laboratorio 1: Algoritmos Aritméticos

ESTRUCTURA DE DATOS Y ALGORITMOS: 2024-01

EZEQUIEL MORALES
PROFESORES: KAROL SUCHAN, YERKO ORTIZ, RODRIGO ALVAREZ.
AYUDANTE: DIEGO BANDA.

1. INTRODUCCIÓN:

En el presente informe se detalla el desarrollo e implementación de la clase BigNum, diseñada para manejar números enteros grandes en Java. La clase incluye métodos fundamentales para la adición y multiplicación de números grandes, además de la conversión a una representación en cadena. Los métodos implementados son: add para sumar dos números BigNum, multiply para multiplicar dos números BigNum, toString para obtener la representación en cadena del número, y métodos internos como addConCarreo, Resta, y compararAbsoluto para realizar operaciones aritméticas precisas. Se llevaron a cabo pruebas de correctitud para validar la implementación y se realizaron mediciones del uso de CPU para evaluar el rendimiento de las operaciones de suma y multiplicación en función del tamaño de los números. Los resultados y análisis se presentan para verificar la eficiencia y precisión de la implementación.

Los métodos requeridos para el programa son:

- ☐ BigNum(int x)
- ☐ BigNum(String x, boolean sign)
- ☐ add(BigNum x, BigNum y) : BigNum
- ☐ multiply(BigNum x, BigNum y) : BigNum
- ☐ toString() : String

udp

2. DESARROLLO:

Primeramente, se crea la base del código. La clase BigNum con sus atributos: num, size y sign.

```java
class BigNum {
    public int[] num; // numero representado en sistema decimal, cada casilla digito de 0 a 9.
    public int size; // largo del numero.
    public boolean sign; // positivo = true, negativo = false.

    public BigNum(int num[], int size, boolean sign){
        this.num = num;
        this.size = size;
        this.sign = sign;

    }
```

Listing 1: Clase BigNum.

Después se empiezan a crear los constructores el primero de ellos es tipo BigNum que recibe un número entero, determina si su signo es positivo o negativo, su valor absoluto y agrega sus dígitos a un arreglo.

```java
public BigNum(int x){

if (x >= 0){
    this.sign = true;
} else {
   this.sign = false;
} //asignar signo.

x = Math.abs(x);    // Valor absoluto de x
String NumToString = Integer.toString(x); // Convertir int a string
this.size = NumToString.length(); // largo del nro
this.num = new int[this.size]; // crea array del largo del nro

for (int i = 0; i < this.size; i++){
    this.num[i] = Character.getNumericValue((NumToString.charAt(i)));
}
}
```

Listing 2: Constructor 1 BigNum.

El segundo constructor es tipo BigNum recibe el valor absoluto del número entero y un booleano que representa el signo del número luego lo almacena en el arreglo.

```java
public BigNum(String x, boolean sign){

    this.sign = sign;  // asignar signo.
    this.size = x.length();  // Tamaño del numero
    this.num = new int[this.size];  // Arreglo para los digitos

    // Convertir cada digito de la cadena a un digito y almacenarlo en el arreglo num.
    for (int i = 0; i < this.size; i++) {
        this.num[i] = Character.getNumericValue(x.charAt(i));
    }
}
```

Listing 3: Constructor 2 BigNum.

2.1. Métodos de adición y multiplicación

El siguiente método add suma dos números BigNum. Si ambos números tienen el mismo signo, realiza una suma directa. Si tienen signos opuestos, compara sus valores absolutos y realiza una resta ajustando el signo según corresponda. Si ambos números son negativos, suma los valores absolutos y establece el signo negativo en el resultado.

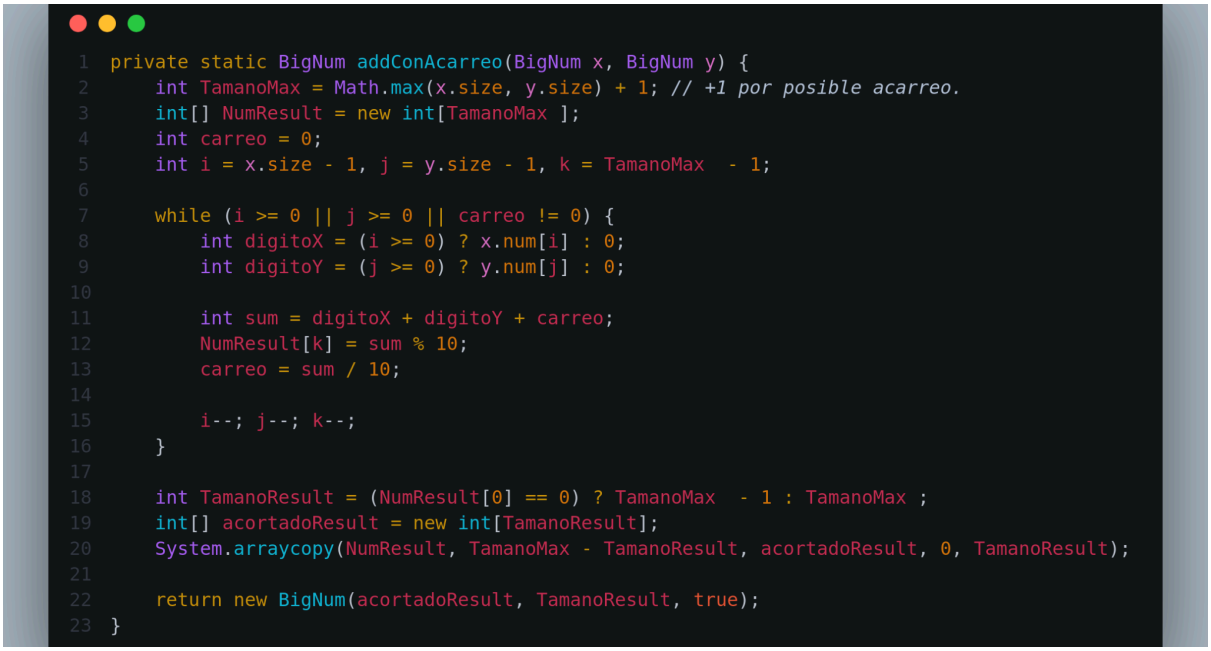
```java
public static BigNum add(BigNum x, BigNum y) {
    // Caso 1
    if (x.sign && y.sign) {
        return addConCarreo(x, y);  // Suma directa.
    }
    // Caso 2
    if (x.sign && !y.sign) {
        int comparacion = compararAbsoluto(x, y);
        if (comparacion  >= 0) {
            // abs(x) >= abs(y) -> x - y.
            return Resta(x, y);
        } else {
            // abs(x) < abs(y) -> -(y - x).
            BigNum result = Resta(y, x);
            result.sign = false;  // Aplicar signo negativo.
            return result;
        }
    }
```

Listing 4: Método add 1.1 BigNum.

```
// Caso 3
    if (!x.sign && y.sign) {
        int comparacion  = compararAbsoluto(x, y);
        if (comparacion >= 0) {
            // abs(x) >= abs(y) -> -(x - y).
            BigNum result = Resta(x, y);
            result.sign = false;  // Aplicar signo negativo.
            return result;
        } else {
            // abs(x) < abs(y) -> y - x.
            return Resta(y, x);
        }
    }
    // Caso 4
    if (!x.sign && !y.sign) {
        BigNum result = addConCarreo(x, y);  // Suma directa.
        result.sign = false;  // Aplicar signo negativo.
        return result;
    }
    return new BigNum(0);  // Caso por defecto (no debería alcanzarse).
}
```

Listing 5: Método add 1.2 BigNum.

El método addConAcarreo suma dos números BigNum dígito a dígito, manejando el acarreo. Calcula el resultado en un arreglo de tamaño máximo posible, ajusta el acarreo y ajusta el tamaño del resultado final eliminando ceros a la izquierda.



```
private static BigNum addConAcarreo(BigNum x, BigNum y) {
    int TamanoMax = Math.max(x.size, y.size) + 1; // +1 por posible acarreo.
    int[] NumResult = new int[TamanoMax ];
    int carreo = 0;
    int i = x.size - 1, j = y.size - 1, k = TamanoMax  - 1;

    while (i >= 0 || j >= 0 || carreo != 0) {
        int digitoX = (i >= 0) ? x.num[i] : 0;
        int digitoY = (j >= 0) ? y.num[j] : 0;

        int sum = digitoX + digitoY + carreo;
        NumResult[k] = sum % 10;
        carreo = sum / 10;

        i--; j--; k--;
    }

    int TamanoResult = (NumResult[0] == 0) ? TamanoMax  - 1 : TamanoMax ;
    int[] acortadoResult = new int[TamanoResult];
    System.arraycopy(NumResult, TamanoMax - TamanoResult, acortadoResult, 0, TamanoResult);

    return new BigNum(acortadoResult, TamanoResult, true);
}
```

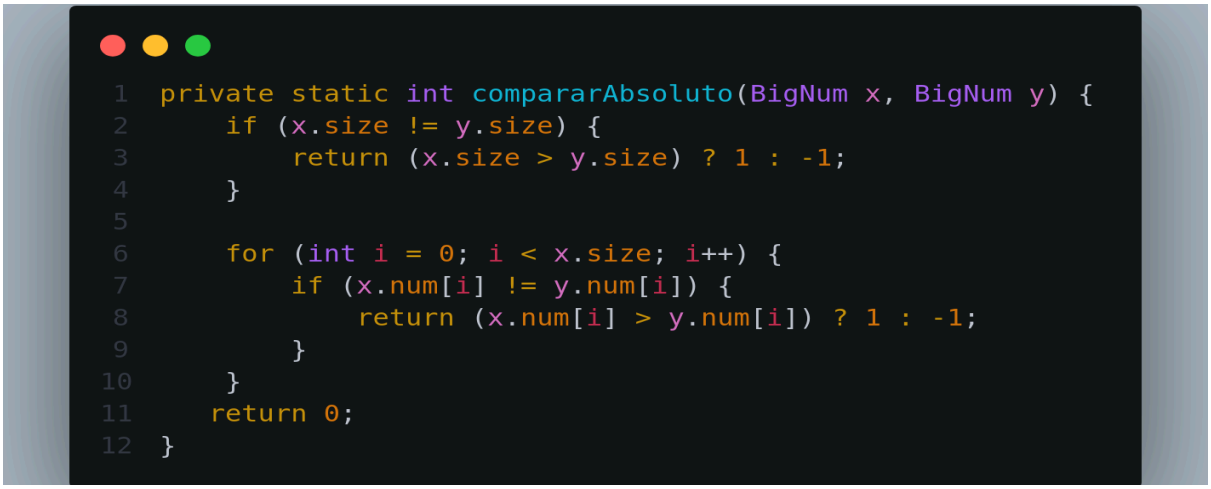
Listing 6: Método addConAcarreo BigNum x, BigNum y.

El método Resta realiza la resta de dos números BigNum dígito a dígito, manejando el préstamo cuando el dígito del minuendo es menor que el del sustraendo. Calcula el resultado en un arreglo de tamaño máximo del minuendo, ajusta el tamaño del resultado eliminando ceros a la izquierda y devuelve un nuevo objeto BigNum con el resultado, manteniendo el signo positivo por defecto.

```java
private static BigNum Resta(BigNum x, BigNum y) {
    int TamanoMax  = x.size;
    int[] NumResult = new int[TamanoMax];

    int aux = 0;
    int i = x.size - 1, j = y.size - 1, k = TamanoMax - 1;

    while (i >= 0 || j >= 0) {
        int digitoX = (i >= 0) ? x.num[i] : 0;
        int digitoY = (j >= 0) ? y.num[j] : 0;

        int diff = digitoX - digitoY - aux;
        if (diff < 0) {
            diff += 10;
            aux = 1;
        } else {
            aux = 0;
        }

        NumResult[k] = diff;
        i--; j--; k--;
    }

    int resultSize = TamanoMax ;
    while (resultSize > 1 && NumResult[TamanoMax - resultSize] == 0) {
        resultSize--;  // Reducir el tamaño si hay ceros a la izquierda.
    }

    int[] acortadoResult = new int[resultSize];
    System.arraycopy(NumResult, TamanoMax - resultSize, acortadoResult, 0, resultSize);

    return new BigNum(acortadoResult, resultSize, true);  // Mantener signo positivo.
}
```

Listing 7: Método Resta BigNum x, BigNum y.

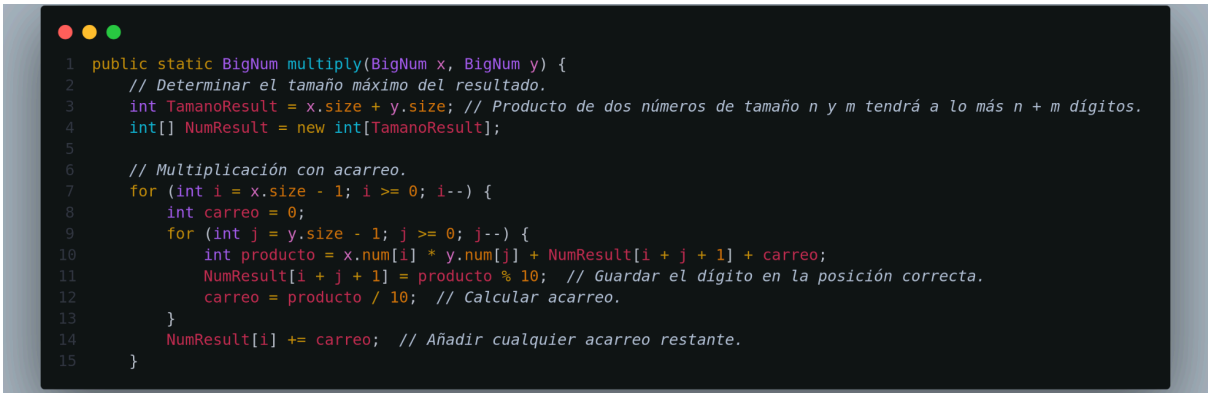
El método compararAbsoluto compara dos números BigNum sin considerar el signo. Primero compara los tamaños y luego, si son iguales, compara los dígitos uno a uno. Devuelve 1 si el primer número es mayor, -1 si es menor, o 0 si son iguales.



```java
private static int compararAbsoluto(BigNum x, BigNum y) {
    if (x.size != y.size) {
        return (x.size > y.size) ? 1 : -1;
    }

    for (int i = 0; i < x.size; i++) {
        if (x.num[i] != y.num[i]) {
            return (x.num[i] > y.num[i]) ? 1 : -1;
        }
    }
    return 0;
}
```

Listing 8: Método Resta BigNum x, BigNum y.

Una vez completados los métodos de adición, se puede proceder con el método multiply. Este método realiza la multiplicación de dos números BigNum utilizando el acarreo. Determina el tamaño máximo del resultado, que es la suma de los tamaños de los dos números multiplicados. Luego, multiplica cada dígito de un número por cada dígito del otro, acumulando el acarreo y almacenando el resultado en un arreglo. Después de calcular el producto, elimina los ceros a la izquierda y ajusta el tamaño del resultado. Finalmente, determina el signo del resultado basado en los signos de los números multiplicados y devuelve un nuevo objeto BigNum con el resultado.



```java
public static BigNum multiply(BigNum x, BigNum y) {
    // Determinar el tamaño máximo del resultado.
    int TamanoResult = x.size + y.size; // Producto de dos números de tamaño n y m tendrá a lo más n + m dígitos.
    int[] NumResult = new int[TamanoResult];

    // Multiplicación con acarreo.
    for (int i = x.size - 1; i >= 0; i--) {
        int carreo = 0;
        for (int j = y.size - 1; j >= 0; j--) {
            int producto = x.num[i] * y.num[j] + NumResult[i + j + 1] + carreo;
            NumResult[i + j + 1] = producto % 10;  // Guardar el dígito en la posición correcta.
            carreo = producto / 10;  // Calcular acarreo.
        }
        NumResult[i] += carreo;  // Añadir cualquier acarreo restante.
    }
```

Listing 9: Método multiply 1.1 BigNum x, BigNum y.

```java
// Encontrar el tamaño real del resultado, eliminando ceros a la izquierda.
    int aux = 0;
    while (aux < TamanoResult - 1 && NumResult[aux] == 0) {
        aux++;
    }

    int[] acortadoResult = new int[TamanoResult - aux];
    System.arraycopy(NumResult, aux, acortadoResult, 0, acortadoResult.length);

    // Determinar el signo del resultado.
    boolean resultSign = (x.sign == y.sign);  // true si ambos son positivos o negativos, false si son diferentes.

    return new BigNum(acortadoResult, acortadoResult.length, resultSign);
}
```

Listing 9: Método multiply 1.2 BigNum x, BigNum y.

El método toString convierte un objeto BigNum en una representación en cadena. Si el número es negativo, agrega un signo negativo al principio. Luego, itera sobre cada dígito del número y lo añade a un StringBuilder. Finalmente, devuelve la cadena resultante que representa el número.

```java
public String toString() {
    StringBuilder sb = new StringBuilder();
    if (!this.sign) {
        sb.append('-'); // Añadir el signo negativo si corresponde
    }
    for (int digito : this.num) {
        sb.append(digito);
    }
    return sb.toString();
}
```

Listing 10: Método ToString.

2.2. Experimentación del programa.

Una vez implementado los métodos se puede proceder a realizar un prueba de la velocidad de ejecución del programa donde se tomará el tiempo en nanosegundos.

```java
public static void main(String[] args) {
    int digitos = (int) Math.pow(10, 6); // 10^6
    Random random = new Random();

    StringBuilder num1Builder = new StringBuilder(digitos);
    StringBuilder num2Builder = new StringBuilder(digitos);

    // Generar dos números de 10^9 dígitos.
    for (int i = 0; i < digitos; i++) {
        num1Builder.append(random.nextInt(10)); // Dígitos aleatorios entre 0 y 9.
        num2Builder.append(random.nextInt(10));
    }

    BigNum num1 = new BigNum(num1Builder.toString(), true);
    BigNum num2 = new BigNum(num2Builder.toString(), true);

    // Medir el tiempo de ejecución en nanosegundos.
    long Tinicial = System.nanoTime();
    BigNum adicion = add(num1, num2);
    BigNum product = multiply(num1, num2);
    long Tfinal = System.nanoTime();
    long duracion = Tfinal - Tinicial; // Tiempo en nanosegundos.

    System.out.println("Tiempo de ejecución: " + duracion + " nanosegundos.");
}
}
```

Listing 11: Main benchmark tiempo de ejecución.

Todos estas pruebas se realizaron en un pc con:

CPU: Ryzen 5 5600x
RAM: 32GB 3200MHZ

Donde la pruebas unitarias para verificar la funcionalidad de la calculadora se hicieron con el siguiente main

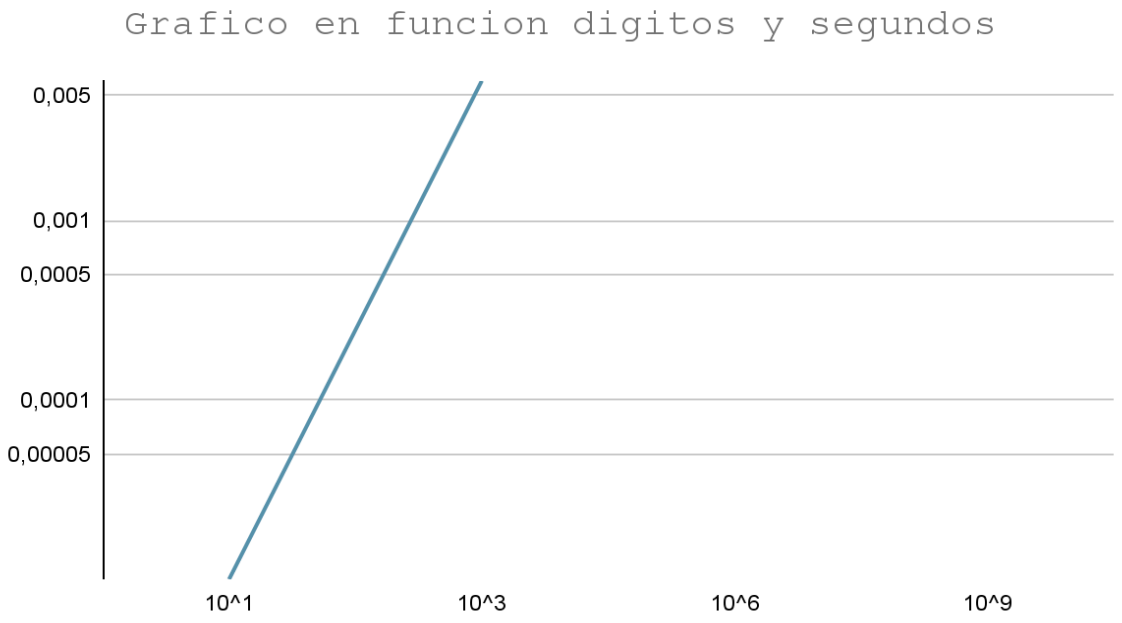
```java
public static void main(String[] args) {
    BigNum num1 = new BigNum(6565);
        BigNum num2 = new BigNum(-5454);
        BigNum sum = BigNum.add(num1, num2);
        System.out.println("Suma: " + sum.toString());

        BigNum product = multiply(num1, num2);

        System.out.println("El producto es: " + product.toString());
}
```

Listing 12: Main calculadora.

La cual dió los siguientes resultados:

INPUT	OUTPUT
NUM1 = 290 NUM2 = 909	SUMA: 1199 PRODUCTO: 263610
NUM1 = -132 NUM2 = -323	SUMA: -455 PRODUCTO: 42636
NUM1 = 6565 NUM2 = -5454	SUMA: 1111 PRODUCTO: -35805510
NUM1 = -89 NUM2 = 1054	SUMA: 965 PRODUCTO: -93806

Luego de verificar la funcionalidad correcta de los métodos se puede proceder a la eficiencia del programa con el main benchmark del Listing 11.



El programa pudo terminar de ejecutarse en 10^1 y en 10^3 más no en 10^6, 10^9.

3. ANÁLISIS:

a) ¿Cuál es el número más grande que la clase BigNum puede almacenar en Java? Documente el número más grande que fue posible crear con su implementación y cómo dicho número fue creado.

El número más grande que pude almacenar en java fue el de 10^5 este se pudo ejecutar en un tiempo de 17537082816 (nanosegundos) o 17.537082816 (segundos).

b) En términos de uso de memoria, ¿qué mejoras podrían hacerse a la clase BigNum para que ahorre memoria?

Para ahorrar memoria se puede hacer una mejor implementación de los métodos del programa, donde la estructura de los datos puede ser más eficiente así ahorrando espacio en la memoria.

c) ¿Cuál es la complejidad teórica de los métodos implementados? Para este análisis, debe utilizar la notación O(f(n)) y presentar una breve reflexión comparando la complejidad teórica de los métodos aritméticos respecto los resultados obtenidos en los benchmark del punto anterior.

En términos de O(f(n)) (BIG O) analizando el programa se puede notar el método O(n * m) más dominante, lo que puede explicar el porque al pasar de 10^5 a 10^6 Al realizar la multiplicación no se puede terminar de ejecutar el programa.

4. CONCLUSIÓN:

Se ha implementado la clase BigNum en Java para operar con números enteros grandes, incluyendo suma, resta, multiplicación y conversión a cadena. La complejidad de suma y resta es O(n), mientras que la multiplicación es O(n * m), siendo más lenta para números grandes. En las pruebas, la ejecución con números de tamaño 10^5 tomó 17 segundos, pero con 10^6 no se completó, evidenciando problemas de rendimiento.

Para mejorar el uso de memoria y la eficiencia, se podrían considerar técnicas como el uso de tipos de datos más compactos y algoritmos de multiplicación más avanzados. La implementación actual es adecuada para números de tamaño moderado, pero necesita optimización para manejar números más grandes de manera efectiva.