Escuela Superior de Cómputo

Matematicas avanzadas para la ingenieria

Obtencion de la raiz cuadrada de dos mediante series

por:
López Manríquez Ángel

profesor
Olvera Andana Miguel

17 de junio de 2019

1. Introduccion

Para hacer la aplicacion que calcule la raiz cuadrada de dos mediante series, se hara uso de los siguientes temas.

Teorema 1.1. *El binomio de newton*
El teorema del binomio establece que cualquier potencia entera positiva n de un binomio puede ser expresada como

$$(a+b)^n = \sum_{k=0}^{n} \binom{n}{k} a^k b^{n-k}$$

Teorema 1.2. *Serie de taylor*
Cualquier funcion f infinitamente derivable puede ser representada como una serie de la forma

$$f(z) = \sum_{k=0}^{\infty} \frac{f^{(k)}(z_0)}{k!}(z-z_0)^k$$

Aqui, se dice que la serie esta centrada en z_0

Definición 1.1. El doble factorial
El doble factorial de un número natural n, que se denota como $n!!$, se define como el producto de n por todos los naturales que le preceden que tienen la misma paridad que el propio n, esto es:

$$n!! = \prod_{k=0}^{\lceil \frac{n}{2} \rceil - 1} (n-2k)$$

2. Planteamiento del problema

Mediante series, hacer un programa en C o JAVA para aproximar la suma parcial numero 100 de la raiz de dos, expresado como una fraccion.

3. Diseño y funcionamiento de la solución

Se procede a obtener los coeficiente de la serie de Taylor para $f(x) = \sqrt{x}$ centrada en 1. Por tanto, tenemos

0. $f^{(0)}(x) = \sqrt{x}$
1. $f^{(1)}(x) = \frac{1}{2^1\sqrt{x}}$
2. $f^{(2)}(x) = -\frac{1}{2^2\sqrt{x^3}}$
3. $f^{(3)}(x) = \frac{1 \cdot 3}{2^3\sqrt{x^5}}$
4. $f^{(4)}(x) = -\frac{1 \cdot 3 \cdot 5}{2^4\sqrt{x^7}}$
5. $f^{(5)}(x) = \frac{1 \cdot 3 \cdot 5 \cdot 7}{2^5\sqrt{x^7}}$

vemos que f puede ser escrita como

$$f(x) = 1 - \sum_{k=1}^{\infty} \frac{(1-x)^k}{2^k k!}(2k-3)!!$$

Lo que se va a codificar es una aproximacion, S_{100} para $f(2)$.

4. Implementacion de la solucion

Para la implementacion definimos una clase *BigRational* para guardar el entero para el numerador y denominador, como se aprecia en la serie, vemos que hay un gran problema con respecto a la capacidad de un tipo de dato primitivo, por ejemplo, para obtener el sumando no. 100 para el denominador vemos que $100! \approx 9{,}332622 \times 10^{157}$, $2^{64} \approx 1{,}267651 \times 10^{30}$. Sin contar que aun no los hemos multiplicado ni sumado con los terminos anteriores... aqui es donde se agradece el poder computacional.

```
1
2  import java.util.*;
3  import java.math.*;
4
5  public class BigRational
6  {
7     // Declaramos el numerador y numerador de la fraccion
8     private BigInteger num, den;
9
10    // Definimos un constructor tal que la fraccion sera 0 / 1
11    public BigRational () {
12       num = BigInteger.ZERO;
13       den = BigInteger.ONE;
14    }
15
16    // Definimos un constructor donde si el usuario estableciese como 0 el
17    // el denominador terminaremos la ejecucion del programa
18    public BigRational (BigInteger num, BigInteger den) {
19       if (den.compareTo(BigInteger.ZERO) == 0)
20          throw new ArithmeticException ("Indeterminacion del tipo x / 0");
21       this.num = num;
22       this.den = den;
23       simplify ();
24    }
25
26    // Simplifica las fraccion dividiendo tanto numerador como denominador por
27    // su maximo comun divisor
28    public void simplify () {
29       BigInteger mxd = num.gcd (den);
30       num = num.divide(mxd);
31       den = den.divide(mxd);
32    }
33
34    // Se define un metodo que retorna la suma de una fraccion dada con esta
35    // fraccion
36    public BigRational add (BigRational fraction) {
37       BigRational z = new BigRational ();
38       z.num = num.multiply (fraction.den).add (fraction.num.multiply (den));
39       z.den = den.multiply (fraction.den);
40       z.simplify ();
41       return z;
```

```
42    }
43
44    // Metodo estatico en el que sumamos las dos fracciones dadas
45    public static BigRational add (BigRational x, BigRational y) {
46       return new BigRational (x.num.multiply (y.den).add (x.den.multiply (y.num)),
47                       x.den.multiply (y.den));
48    }
49
50    // Getters
51    public BigInteger getNumerator () {
52       return num;
53    }
54
55    public BigInteger getDenominator () {
56       return den;
57    }
58
59    // Setters
60    public void setNumerator (BigInteger _num) {
61       num = _num;
62       simplify ();
63    }
64
65    // Saldremos de la aplicacion en caso de que se quiera establecer por 0 al
66    // denominador
67    public void setDenominator (BigInteger _den) {
68       if (_den.compareTo (BigInteger.ZERO) == 0)
69          throw new ArithmeticException ("Se esta tratando de establecer al " +
70             "denominador en 0");
71       den = _den;
72       simplify ();
73    }
74 }
```

Para obtener la suma parcial y los factoriales y dobles factoriales.

```
1
2  import java.util.*;
3  import java.math.*;
4
5  public class SquareRoot
6  {
7     /* Se declaran dos mapas para consultar los valores para n! y n!! de forma
8      * instantanea, ademas de que serviran para obtener sus valores de forma
9      * recursiva mas rapidamente mediante la tecnica de memorizacion */
10    private static Map <Integer, BigInteger> doubleFact =
11       new HashMap <Integer, BigInteger> ();
12    private static Map <Integer, BigInteger> fact =
13       new HashMap <Integer, BigInteger> ();
14
```

```
15    // Obtenemos los valores de forma recursiva para n!
16    static BigInteger factorial (final int n) {
17       if (fact.containsKey (n))
18          return fact.get(n);
19       BigInteger value = new BigInteger ("1");
20       if (n == 0)
21          value = BigInteger.ONE;
22       else
23          value = factorial (n - 1).multiply (BigInteger.valueOf (n));
24       fact.put (n, value);
25       return value;
26    }
27
28    // Obtenemos los valores de forma recursiva para n!!
29    static BigInteger doubleFactorial (final int n) {
30       if (doubleFact.containsKey (n))
31          return doubleFact.get (n);
32       BigInteger value = new BigInteger ("1");
33       if (n <= 1)
34          value = BigInteger.ONE;
35       else
36          value = doubleFactorial (n - 2).multiply (BigInteger.valueOf (n));
37       doubleFact.put (n, value);
38       return value;
39    }
40
41    // Obtenemos la aproximacion de la raiz de dos mediante la suma parcial numero 100
42    static BigRational squareRootOfTwo () {
43       BigRational sum = new BigRational ();
44       BigRational frac2 = new BigRational();
45       for (int k = 1; k <= 100; k++) {
46          frac2.setNumerator (doubleFactorial (2 * k - 3).multiply (
47                  BigInteger.valueOf ((long) Math.pow (-1, k + 1))));
48          frac2.setDenominator (factorial (k).multiply (new BigInteger ("2").pow (k)));
49          sum = sum.add (frac2);
50       }
51       frac2.setNumerator (BigInteger.ONE);
52       frac2.setDenominator (BigInteger.ONE);
53       sum = sum.add (frac2);
54       return sum;
55    }
56
57    /* Declaramos una fraccion que soporta valores enormes tanto para numerador
58     * como denominador, para mostrar por pantalla la fraccion aproximada para
59     * la raiz de dos */
60    public static void main(String[] args) {
61       BigRational fraction = new BigRational ();
62       fraction = squareRootOfTwo();
63       System.out.println(fraction.getNumerator());
64       System.out.println(fraction.getDenominator());
```

```
65    }
66 }
```

5. Funcionamiento

Obteniendo S_{50}, tenemos:

```
angel@lambda: square » make run                                    [11:20:16]
javac -g SquareRoot.java BigRational.java && java SquareRoot
224028349837775086224261198949
158456325028528675187087900672
```

vemos que $\sqrt{2} \approx 1{,}413817655$

Obteniendo S_{100}, tenemos:

```
angel@lambda: square » make run                                    [17:07:20]
javac -g SquareRoot.java BigRational.java && java SquareRoot
284040972217357055485739049123142155961436133985593359044221
200867255532373784442745261542645325315275374222849104412672
```

vemos que $\sqrt{2} \approx 1{,}414073048$
Obteniendo S_{1000}, tenemos:

```
angel@lambda: square » make run                                    [17:10:05]
javac -g SquareRoot.java BigRational.java && java SquareRoot
253702637740051247574917307432179602197496451430469306178067241036302708774054383202
543735464325750223322501655507041209299678052531494420309196831032349451266892628762
930550709502143027979545170951536922570769839995138108250030964369584790314150620536
004491871390610658851612086512731188764065993668099416876261361526173007296441529126
695257241610408654735277816186104045245713882907743599117917570631240480138769868149
800151821382555262061836643458552632798061613832873707869480822598649006238150677629
965117431597229341369459443320192180632097475556129479665052064950598815408660509488
0077400851849
179395421136602269411380187684012810003487140951358625074631677629025978342557861540
103044736954104674757181974841791058351112337634852395535301774401039560217390608039
550437501076217419125070111607698421974197257471274161947481818667682853188228678079
539057122128748138975983758786424452400256596828644814600263920288216415003717945012
365717032710588281920316744854102860190637706619189518376981067683135310930306903323
471531028756315874770598830532639740472018625867121536858862561187628058150985285555
281914974571899263044978780362585170180118412316601836618013751285691829403071021503
4138299203584
```

vemos que $\sqrt{2} \approx 1{,}414209104$