Raízes Primitivas

Aluno: Lucas Pereira da Silva (10100754)

Raíz primitiva modulo n é um número m o qual para cada co-primo de n, existe uma potência de m onde m módulo n seja esse co-primo.

Em outras palavars:

- Sendo números co-primos, dois inteiros tal que ambos possuam apenas o 1 como divisor comum.
- Sendo $\varphi(n)$ a função que retorna a quantidade de co-primos de um inteiro n que são menores que n.
- Considere um inteiro p e um inteiro r.
- Considere todas as potências de r, indo de r^1 até $r^{\varphi(p)}$.
- Se essas potências módulo p gerarem os co-primos de p, então r é uma raíz primitiva módulo p.

Exemplos:

2 é raíz primitiva módulo 5:

- $2^1 = 2 \rightarrow 2$ módulo $5 = 2$
- $2^2 = 4 \rightarrow 4$ módulo $5 = 4$
- $2^3 = 8 \rightarrow 8$ módulo $5 = 3$
- $2^4 = 16 \rightarrow 16$ módulo $5 = 1$

Isso mostra que 2 é raíz primitiva módulo 5, pois suas potências (2^1, …, $2^{\varphi(n)}$) módulo 5 geram os co-primos de 5.

Outras raízes primitivas:

- As raízes primitivas módulo 11 são: 2, 6, 7, 8.
- As raízes primitivas módulo 17 são: 3, 5, 6, 7, 10, 11, 12, 14.
- As raízes primitivas módulo 29 são: 2, 3, 8, 10, 11, 14, 15, 18, 19, 21, 26, 27.

Código Fonte

```
package br.ufsc.inf.ine5429.raizPrimitiva;


import java.math.BigInteger;
import java.util.HashSet;
```

```
import java.util.InputMismatchException;
import java.util.LinkedList;
import java.util.List;
import java.util.Scanner;
import java.util.Set;

public class RaizPrimitiva {
     private BigInteger numero;
     private Set<BigInteger> raizes;

     public RaizPrimitiva(BigInteger numero) {
          this.numero = numero;
     }

     /* Fornece uma lista com todas as raízes primitivas módulo
algum número. */
     public List<BigInteger> encontrarRaizesPrimitivas() {
          BigInteger phi = calcularPhi(numero);
          Set<BigInteger> fatoresPrimosDePhi =
encontrarFatoresPrimos(phi);
          List<BigInteger> raizesPrimitivas = new
LinkedList<BigInteger>();
          BigInteger candidataRaizPrimitiva = BigInteger.ONE;
          BigInteger quantidadeDeRaizesPrimitivas =
calcularPhi(phi);
          BigInteger contadorDeRaizesPrimitivas = BigInteger.ZERO;
          while
(contadorDeRaizesPrimitivas.compareTo(quantidadeDeRaizesPrimitivas
) == -1) {
               if (testarRaizPrimitiva(candidataRaizPrimitiva,
this.numero, phi, fatoresPrimosDePhi)) {
                    raizesPrimitivas.add(candidataRaizPrimitiva);
                    contadorDeRaizesPrimitivas =
contadorDeRaizesPrimitivas.add(BigInteger.ONE);
               }
               candidataRaizPrimitiva =
candidataRaizPrimitiva.add(BigInteger.ONE);
          }
```

```
            return raizesPrimitivas;
      }

      /* Dada uma candidata a raíz primitiva módulo um número, dado
o phi desse número e dado os fatores primos desse phi, então testa
se a candidata a raiz primitiva é realmente uma raiz primitiva. */
      private Boolean testarRaizPrimitiva(BigInteger raizPrimitiva,
BigInteger numero, BigInteger phi, Set<BigInteger> fatoresPrimos)
{
            for (BigInteger fatorPrimo : fatoresPrimos) {
                  if (raizPrimitiva.modPow(phi.divide(fatorPrimo),
numero).equals(BigInteger.ONE)) {
                        return false;
                  }
            }
            return true;
      }

      /* Fornece um conjunto com os fatores primos de um dado
número. Utiliza o método nextProbablePrime da classe BigInteger
para realizar esta tarefa de forma mais rápida. */
      private Set<BigInteger> encontrarFatoresPrimos(BigInteger
numero) {
            Set<BigInteger> fatoresPrimos = new
HashSet<BigInteger>();
            BigInteger fatorPrimoAtual =
BigInteger.ONE.nextProbablePrime();
            while (!numero.equals(BigInteger.ONE)) {
                  BigInteger[] divisaoSobra =
numero.divideAndRemainder(fatorPrimoAtual);
                  BigInteger divisao = divisaoSobra[0];
                  BigInteger sobra = divisaoSobra[1];
                  if (sobra.equals(BigInteger.ZERO)) {
                        numero = divisao;
                        fatoresPrimos.add(fatorPrimoAtual);
                  } else {
                        fatorPrimoAtual =
fatorPrimoAtual.nextProbablePrime();
                  }
```

```
            }
            return fatoresPrimos;
      }

      /* Encontra o phi de algum número. Ele primeiro verifica se o
número é provavelmente primo e, caso seja, retorna o número menos
um. É feito isso, pois o phi de um número primo é o próprio número
menos um. Caso o número não seja primo, então cálcula o phi de uma
forma mais cara. */
      private BigInteger calcularPhi(BigInteger numero) {
            if (numero.isProbablePrime(30)) {
                  return numero.subtract(BigInteger.ONE);
            }
            BigInteger phi = BigInteger.ZERO;
            BigInteger candidatoCoPrimo = BigInteger.ONE;
            while (candidatoCoPrimo.compareTo(numero) == -1) {
                  if
(candidatoCoPrimo.gcd(numero).equals(BigInteger.ONE)) {
                        phi = phi.add(BigInteger.ONE);
                  }
                  candidatoCoPrimo =
candidatoCoPrimo.add(BigInteger.ONE);
            }
            return phi;
      }


      public static void main(String[] argumentos) {
            Scanner leitor = new Scanner(System.in);
            Boolean encerrar = false;
            while (!encerrar) {
                  System.out.print("Digite o número para o qual deseja
encontrar as raízes primitivas: ");
                  try {
                        BigInteger numero = leitor.nextBigInteger();
                        List<BigInteger> raizesPrimitivas = new
RaizPrimitiva(numero).encontrarRaizesPrimitivas();
                        System.out.printf("O número possui %d raízes
primitivas. Elas são: %s.\n", raizesPrimitivas.size(),
raizesPrimitivas.toString());
```

```
                } catch (InputMismatchException excecao) {
                    System.out.println("Programa encerrado.");
                    encerrar = true;
                }
            }
        }
}
```