Лабораторная работа №5

Математические основы защиты информации и информационной безопасности

Минов Кирилл Вячеславович | НПМмд-02-23

Содержание

1 Цель работы

Реализовать на языке программирования вероятностные алгоритмы проверки чисел на простоту.

2 Теоретическое введение

Детерминированный алгоритм всегда действует по одной и той же схеме и гарантированно решает поставленную задачу (или не дает никакого ответа). **Вероятностный** алгоритм использует генератор случайных чисел и дает не гарантированно точный ответ. Вероятностные алгоритмы в общем случае не менее эффективны, чем детерминированные (если используемый генератор случайных чисел всегда дает набор одних и тех же чисел, зависящих от входных данных, то вероятностный алгоритм становится детерминированным).

3 Выполнение лабораторной работы

Тест Ферма реализуем по следующей схеме:

а вход подается нечетное целое число n>= 5;

1)Выбрать случайное целое число a 2<= a <= n - 2;

2)Вычислить r <- a^(n-1) (mod n)

3)При r = 1 результат: "Число, вероятно, простое". В противном случае результат: "Число составное"

```java
import java.util.Random;

public class FermatTest {

    public static String fermat(int n) {
        Random random = new Random();
        int a = random.nextInt((n - 2) + 1) + 2; // Генерация случайного числа в диапазоне [2, n-2]
        int r = modPow(a, n - 1, n); // Вычисление a^(n-1) mod n

        if (r == 1) {
            return "Число " + n + " вероятно простое";
        } else {
            return "Число " + n + " составное";
        }
    }

    // Вспомогательная функция для вычисления a^b mod m
    private static int modPow(int a, int b, int m) {
        int result = 1;
        a = a % m;
        while (b > 0) {
            if (b % 2 == 1) {
                result = (result * a) % m;
            }
            b /= 2;
            a = (a * a) % m;
        }
        return result;
    }

    public static void main(String[] args) {
        int numberToTest = 15; // Замените на желаемое число для теста
        System.out.println(fermat(numberToTest));
    }
}
```

Рис. 1: Тест Ферма

Вычисление символа Якоби реализуем по следующей схеме:

Инициализация: Создание класса JacobiSymbol. Определение статической функции jacobiSymbol, которая принимает два целых числа a и n и возвращает символ Якоби (a/n). В функции main происходит тестирование алгоритма для заданных значений a и n.

Базовые случаи: Если n не положительное нечетное число или n четное, выбрасывается исключение IllegalArgumentException. Если a равно 0, возвращается 1, если n равно 1, и 0 в противном случае. Если a равно 1, возвращается 1.

Свойства символа Якоби: Если a отрицательно, устанавливается знак в зависимости от значения n % 4. Если a четное, вычисляется знак в зависимости от значения n % 8. Если a нечетное и неотрицательное, применяется критерий взаимной простоты.

Критерий взаимной простоты: Если a % 4 == 3 и n % 4 == 3, меняется знак. Рекурсивный вызов функции jacobiSymbol с аргументами n % a и a.

Тестирование: Задание значений a и n. Вызов функции jacobiSymbol с заданными значениями. Вывод результата. Если возникает исключение IllegalArgumentException, выводится сообщение об ошибке

```java
public class JacobiSymbol {

    // Функция для вычисления символа Якоби (a/n)
    private static int jacobiSymbol(int a, int n) {
        // Базовые случаи
        if (n <= 0 || n % 2 == 0) {
            throw new IllegalArgumentException("Второй аргумент должен быть положительным нечетным числом.");
        }

        if (a == 0) {
            return (n == 1) ? 1 : 0;
        } else if (a == 1) {
            return 1;
        }

        // Свойства символа Якоби
        if (a < 0) {
            int sign = (n % 4 == 1) ? 1 : -1;
            return sign * jacobiSymbol(-a, n);
        } else if (a % 2 == 0) {
            int sign = ((n % 8 == 1) || (n % 8 == 7)) ? 1 : -1;
            return sign * jacobiSymbol( a: a / 2, n);
        } else {
            // Критерий взаимной простоты
            int sign = 1;
            if (a % 4 == 3 && n % 4 == 3) {
                sign = -sign;
            }

            return sign * jacobiSymbol( a: n % a, a);
        }
```

Рис. 2: Вычисление символа Якоби

Тест Соловэя-Штрассена реализуем по следующей схеме:

Инициализация: Создание класса SolovayStrassenTest. Определение статических функций jacobiSymbol и modularExponentiation для вычисления символа Якоби и модульного возведения в степень соответственно.

Вычисление символа Якоби: Функция jacobiSymbol(a, n) вычисляет символ Якоби для целых чисел a и n. Рекурсивные вызовы и базовые случаи рассматриваются в соответствии с определением символа Якоби.

Модульное возведение в степень: Функция modularExponentiation(a, exponent, n) использует алгоритм быстрого возведения в степень.

Тест Соловея-Штрассена: Функция solovayStrassenTest(n, iterations) проверяет простоту числа n. Проверяется, что n больше 2, и в случае, если n равно 2, возвращается true. Выполняется цикл с заданным количеством итераций: Генерируется случайное целое число a в интервале [2, n-2]. Вычисляется символ Якоби jacobi Проверяется условие теста Соловея-Штрассена. Если не выполняется, возвращается false. Если условие прошло для всех итераций, возвращается true, что означает, что n вероятно простое число.

Тестирование: В функции main выбирается число n, которое нужно проверить на простоту, и указывается количество итераций. Вызывается функция solovayStrassenTest для проверки простоты n. Выводится результат проверки. Если число вероятно простое, выводится "вероятно простое число", в противном случае - "составное число".

```java
public class SolovayStrassenTest {

    private static final int MAX_ITERATIONS = 50;

    // Вспомогательная функция для вычисления символа Якоби (a/n)
    private static int jacobiSymbol(BigInteger a, BigInteger n) {
        if (n.compareTo(BigInteger.ZERO) <= 0 || n.mod(BigInteger.valueOf(2)).equals(BigInteger.ZERO)) {
            throw new IllegalArgumentException("Второй аргумент должен быть положительным нечетным числом.");
        }

        if (a.equals(BigInteger.ZERO)) {
            return (n.equals(BigInteger.ONE)) ? 1 : 0;
        } else if (a.equals(BigInteger.ONE)) {
            return 1;
        }

        if (a.compareTo(BigInteger.ZERO) < 0) {
            int sign = (n.mod(BigInteger.valueOf(4)).equals(BigInteger.ONE)) ? 1 : -1;
            return sign * jacobiSymbol(a.negate(), n);
        } else if (a.mod(BigInteger.valueOf(2)).equals(BigInteger.ZERO)) {
            int sign = (n.mod(BigInteger.valueOf(8)).equals(BigInteger.ONE) || n.mod(BigInteger.valueOf(8)).equals(BigInteger.valueOf(7))) ? 1 : -1;
            return sign * jacobiSymbol(a.divide(BigInteger.valueOf(2)), n);
        } else {
            if (a.equals(BigInteger.ONE)) {
                return 1;
            } else {
                int sign = 1;
                if (a.mod(BigInteger.valueOf(4)).equals(BigInteger.valueOf(3)) && n.mod(BigInteger.valueOf(4)).equals(BigInteger.valueOf(3))) {
                    sign = -sign;
                }

                return sign * jacobiSymbol(n.mod(a), a);
            }
        }
    }
```

Рис. 3: Тест Соловэя-Штрассена

```java
// Вспомогательная функция для вычисления a^((n-1)/2) mod n
private static BigInteger modularExponentiation(BigInteger a, BigInteger exponent, BigInteger n) {
    if (exponent.equals(BigInteger.ZERO)) {
        return BigInteger.ONE;
    }

    BigInteger result = BigInteger.ONE;
    a = a.mod(n);

    while (exponent.compareTo(BigInteger.ZERO) > 0) {
        if (exponent.mod(BigInteger.valueOf(2)).equals(BigInteger.ONE)) {
            result = result.multiply(a).mod(n);
        }

        exponent = exponent.divide(BigInteger.valueOf(2));
        a = a.multiply(a).mod(n);
    }

    return result;
}
```

Рис. 4: Тест Соловэя-Штрассена

```java
// Функция для проверки простоты числа с использованием теста Соловея-Штрассена
private static boolean solovayStrassenTest(BigInteger n, int iterations) {
    if (n.compareTo(BigInteger.valueOf(2)) < 0) {
        return false; // Число должно быть больше или равно 2
    }

    if (n.equals(BigInteger.valueOf(2))) {
        return true; // 2 - простое число
    }

    Random rand = new Random();

    for (int i = 0; i < iterations; i++) {
        // Выбираем случайное число a в интервале [2, n-2]
        BigInteger a = BigInteger.valueOf(2 + rand.nextInt( bound: n.intValue() - 3));

        // Вычисляем символ Якоби и a^((n-1)/2) mod n
        int jacobi = jacobiSymbol(a, n);
        BigInteger exponent = n.subtract(BigInteger.ONE).divide(BigInteger.valueOf(2));
        BigInteger result = modularExponentiation(a, exponent, n);

        // Проверяем условие теста Соловея-Штрассена
        if (jacobi == 0 || !result.equals(BigInteger.ONE) && !result.equals(n.subtract(BigInteger.ONE))) {
            return false; // n - составное число
        }
    }

    return true; // Вероятно, n простое
}
```

Рис. 5: Тест Соловэя-Штрассена

Тест Миллера-Рабина реализуем по следующей схеме:

Инициализация:

Создание класса MillerRabinTest. Определение статической функции power
Определение статической функции millerRabinTest для проверки простоты числа.

Вычисление степени по модулю: Функция power(a, b, m) вычисляет с использованием алгоритма быстрого возведения в степень.

Функция millerRabinTest(n, iterations) проверяет простоту числа n. Проверяется, что n больше 2. Если n равно 2, возвращается true (2 - простое число). Если n четное (кроме 2), возвращается false, так как четные числа (кроме 2) не являются простыми.

Выполняется цикл с заданным количеством итераций: Выбирается случайное число a в интервале [2, n-2]. Проверяется условие Миллера-Рабина. Если выполняется, переход к следующей итерации. Если условия не выполняются, возвращается false (n - составное число). Если все итерации прошли успешно, возвращается true (n вероятно простое).

Тестирование: В функции main выбирается число n, которое нужно проверить на простоту, и указывается количество итераций. Вызывается функция millerRabinTest для проверки простоты n. Выводится результат проверки. Если число вероятно простое, выводится "вероятно простое число", в противном случае - "составное число".

```java
    // Вспомогательная функция для вычисления a^b mod m
    private static BigInteger power(BigInteger a, BigInteger b, BigInteger m) {...}

    // Функция для проверки простоты числа с использованием теста Миллера-Рабина
    private static boolean millerRabinTest(BigInteger n, int iterations) {
        if (n.compareTo(BigInteger.valueOf(2)) < 0) {
            return false; // Число должно быть больше или равно 2
        }

        if (n.equals(BigInteger.valueOf(2))) {
            return true; // 2 - простое число
        }

        if (n.and(BigInteger.ONE).equals(BigInteger.ZERO)) {
            return false; // Четные числа (кроме 2) не являются простыми
        }

        // Представление n - 1 в виде 2^s * d
        BigInteger d = n.subtract(BigInteger.ONE);
        int s = 0;
        while (d.and(BigInteger.ONE).equals(BigInteger.ZERO)) {
            s++;
            d = d.shiftRight(1);
        }

        Random rand = new Random();

        for (int i = 0; i < iterations; i++) {
            // Выбор случайного числа a в интервале [2, n-2]
            BigInteger a = BigInteger.valueOf(2 + rand.nextInt( bound: n.intValue() - 3));

            // Вычисление a^d mod n
            BigInteger x = power(a, d, n);
```

Рис. 6: Тест Миллера-Рабина

```java
            // Проверка условия Миллера-Рабина
            if (x.equals(BigInteger.ONE) || x.equals(n.subtract(BigInteger.ONE))) {
                continue; // Вероятность, что n - простое, высока
            }

            // Повторное возведение в квадрат s раз
            for (int r = 1; r < s; r++) {
                x = x.multiply(x).mod(n);
                if (x.equals(BigInteger.ONE)) {
                    return false; // n - составное число
                }
                if (x.equals(n.subtract(BigInteger.ONE))) {
                    break; // Вероятность, что n - простое, высока
                }
            }

            if (!x.equals(n.subtract(BigInteger.ONE))) {
                return false; // n - составное число
            }
        }

        return true; // Вероятно, n простое
    }

    public static void main(String[] args) {...}
}
```

Рис. 7: Тест Миллера-Рабина

4 Выводы

В ходе выполнения данной лабораторной работы были реализованы вероятностные алгоритмы проверки чисел на простоту.