МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

**НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «ЛЬВІВСЬКА ПОЛІТЕХНІКА»**

Інститут комп'ютерних наук та інформаційних технологій

Кафедра інформаційних систем та мереж



ЗВІТ

про виконання лабораторної роботи № 6

«Ключовий обмін Діффі-Хеллмана»

з дисципліни

«Технології захисту інформації»

Виконала:

студентка групи ІТ-31

Щербак Л. В. Прийняв викладач:

Досин Д.Г.

Львів 2022

**Мета роботи:** освоїти методи генерації великих простих чисел і методи

перевірки великих чисел на простоту. Ознайомитися з теоремою Ейлера,

навчитися будувати початкові корені по модулю n. Вивчити схему обміну

ключами Діффі-Хеллмана.

**Завдання роботи:**

індивідуальне завдання з детальним формулюванням розв’язуваної

задачі, використовувані (власні) теоретичні відомості;

1. Написати програму на мові java, яка надає засоби:

А) генерації великих простих чисел, які перевищують 264 .Програма за

заданими t (кількість перевірок за тестом Рабіна-Міллера) та n (кількість біт) повинна генерувати просте n-бітне число, відображаючи при цьому кількість ітерацій алгоритму генерації простого числа, які необхідно виконати для його генерації та час який для цього був необхідний;

Б) програма за заданими межами діапазону повинна виводити всі прості

числа з цього діапазону, відображаючи час, витрачений на генерацію всіх

чисел;

В) визначити для заданого числа перші 100 початкових коренів,

відображаючи при цьому сумарний час витрачений програмою на їх пошук;

**Відповіді на контрольні питання:**

1. Поняття великих простих чисел та процес їх генерації.

Алгоритм генерації простого числа

1. Згенерувати випадкове n-бітове число p.

2. Встановити його старший і молодший біти рівними 1. Старший біт гарантуватиме необхідну довжину шуканого числа, а молодший біт – забезпечувати його непарність.

3. Переконатися, що p не ділиться на невеликі прості числа: 3, 5, 7, 11 і т.д.

Найбільш ефективною є перевірка на можливість ділення на всі прості числа, менші 2000.

4. Виконати тест Рабіна-Міллера мінімум 5 разів.

Якщо p не пройшло хоча б одну перевірку в пунктах 3 або 4, то воно не є

простим. Перевірка того, що випадкове непарне число р не ділиться на 3, 5 і 7, відсікає 54% непарних чисел. Перевірка можливості ділення на всі прості числа, які менші 256, відсікає 80% складених непарних чисел.

Навіть якщо складене число «пройшло» через цей алгоритм, це буде відразу ж помічено, тому що шифрування і дешифрування не будуть працювати.

2. Алгоритм ефективної реалізації піднесення цілого числа в степінь за модулем

BigInteger z = a.modPow(m, p);

m – степінь

p – модуль

a - число

3. На чому базується безпека ключового обміну по схемі Діффі-Хеллмана

Безпека обміну ключами в алгоритмі Діффі-Хеллмана витікає з того факту, що, хоча відносно легко обчислити експоненти по модулю простого числа, проте важко обчислити дискретні логарифми. Для великих простих чисел задача вважається нерозв'язною

**Текст програми реалізації:**

package lab6;

import java.math.BigDecimal;

import java.math.BigInteger;

import java.math.RoundingMode;

import java.util.ArrayList;

import java.util.List;

import java.util.Random;

public class SimpleNumbers {

public static BigInteger \_start;

public static BigInteger \_end;

public static int numOfTests;

//Generate simple number

public static List<String> generateSimpleNumber() {

final long start = System.currentTimeMillis();

final List<String> list = new ArrayList<>();

BigInteger p = BigInteger.valueOf(4);

int iterations = 0;

while (!notDivideSimple(p)) {

p = getRandomBigInteger();

iterations++;

}

while (!isSimple(p)) {

p = getRandomBigInteger();

iterations++;

}

final long end = System.currentTimeMillis();

\_start = p;

\_end = p.add(BigInteger.valueOf(2000));

list.add("New simple number : " + p.toString());

list.add("Iterations : " + String.valueOf(iterations));

list.add("Execution time : " + String.valueOf(end - start) + " ms");

return list;

}

//Not divide simple numbers

private static boolean notDivideSimple(final BigInteger p) {

for (int i = 2; i < 2000; i++) {

if (isPrime(i)) {

if (p.mod(BigInteger.valueOf(i)).equals(BigInteger.ZERO) && !p.equals(BigInteger.valueOf(i))) {

return false;

}

}

}

return true;

}

//Is simple (Test Rabin-Miller)

private static boolean isSimple(final BigInteger p) {

if (p.equals(BigInteger.TWO) || p.equals(BigInteger.valueOf(3))) { return true;

}

if (p.compareTo(BigInteger.ONE) <= 0 || p.mod(BigInteger.TWO).equals(BigInteger.ZERO)) { return false;

}

int b = 0;

BigInteger d = p.subtract(BigInteger.ONE);

while (d.mod(BigInteger.TWO).equals(BigInteger.ZERO)) {

d = d.divide(BigInteger.TWO);

b++;

}

for (int i = 0; i < numOfTests; i++) {

final BigInteger a = randomA(String.valueOf(2),

String.valueOf(p.subtract(BigInteger.TWO)));

final BigInteger m = (p.subtract(BigInteger.ONE)).divide(BigInteger.TWO.pow(b)); BigInteger z = a.modPow(m, p);

if (!z.equals(BigInteger.ONE) && !z.equals(p.subtract(BigInteger.ONE))) { int j = 1;

while (j < b && !z.equals(p.subtract(BigInteger.ONE))) { z = z.modPow(BigInteger.TWO, p);

if (z.equals(BigInteger.ONE)) {

return false;

}

j += 1;

if (!z.equals(p.subtract(BigInteger.ONE))) { return false;

}

}

}

}

return true;

}

//Generate random simple number more then 2^64

private static BigInteger getRandomBigInteger() {

int power = 64;

BigInteger \_2pow64 = BigInteger.valueOf(2).pow(power);

final Random rand = new Random();

//final BigInteger \_2pow65 = BigInteger.valueOf(2).pow(power + 1);

BigInteger result;

do {

result = new BigInteger(\_2pow64.bitLength(), rand);

}

while (result.compareTo(\_2pow64) < 0);

return result;

}

private static BigInteger randomA(final String min, final String max) { final BigInteger maxLimit = new BigInteger(max);

final BigInteger minLimit = new BigInteger(min);

final BigInteger bigInteger = maxLimit.subtract(minLimit);

final Random randNum = new Random();

final int len = maxLimit.bitLength();

BigInteger res = new BigInteger(len, randNum);

if (res.compareTo(minLimit) < 0)

res = res.add(minLimit);

if (res.compareTo(bigInteger) >= 0)

res = res.mod(bigInteger).add(minLimit);

return res;

}

//Is prime number

public static boolean isPrime(final int n) {

for (int i = 2; i <= Math.sqrt(n); i++) {

if (n % i == 0) {

return false;

}

}

return true;

}

//Display simple numbers in range

public static List<String> displayPrimesInRange(BigInteger l, BigInteger r){

final long start = System.currentTimeMillis();

final List<String> resultList = new ArrayList<>();

while (l.compareTo(r) < 0) {

l = l.nextProbablePrime();

resultList.add(l.toString());

}

final long end = System.currentTimeMillis();

resultList.add("Execution time : " + BigInteger.valueOf(end - start) + " ms");

return resultList;

}

//Return first 100 sqrt of simple number

public static List<String> firstHundredSqrt(BigDecimal num){

final long start = System.currentTimeMillis();

final List<String> resultList = new ArrayList<>();

for (int i = 0; i < 100; i++) {

resultList.add(sqrt(num, 1).toString());

}

final long end = System.currentTimeMillis();

resultList.add("Execution time : " + BigInteger.valueOf(end - start) + " ms");

return resultList;

}

public static BigDecimal sqrt(BigDecimal A, final int SCALE) {

BigDecimal x0 = new BigDecimal("0");

BigDecimal x1 = BigDecimal.valueOf(Math.sqrt(A.doubleValue()));

while (!x0.equals(x1)) {

x0 = x1;

x1 = A.divide(x0, SCALE, RoundingMode.HALF\_UP);

x1 = x1.add(x0);

BigDecimal TWO = BigDecimal.valueOf(2);

x1 = x1.divide(TWO, SCALE, RoundingMode.HALF\_UP);

}

return x1;

}

public static void main(String[] args) {

int power = 64;

BigInteger \_2pow64 = BigInteger.valueOf(2).pow(power);

System.out.println("2^64 = " + \_2pow64);

numOfTests = 5;

System.out.println("Number of checks : " + numOfTests);

List<String> list = generateSimpleNumber();

System.out.println("Part 1. " + list);

System.out.println("Show all simple numbers between " + \_start + " and " + \_end);

System.out.println("Part 2. " + displayPrimesInRange(\_start, \_end));

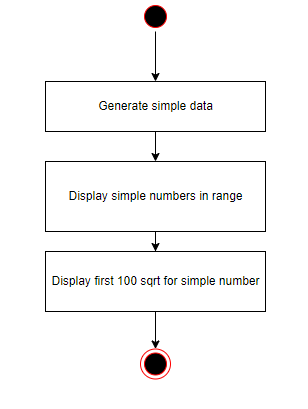
System.out.println("Show first 100 sqrt for the number : " + \_2pow64);

System.out.println("Part 3. " + firstHundredSqrt(new BigDecimal(\_2pow64)));

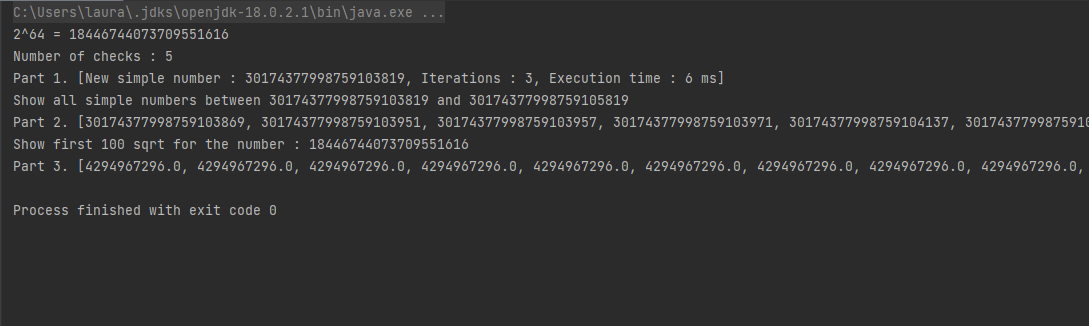
}

}

**Блок-схема:**



**Результати роботи програми:**



**Висновки:** освоїла методи генерації великих простих чисел і методи перевірки великих чисел на простоту. Ознайомилася з теоремою Ейлера, навчилася будувати початкові корені по модулю n. Вивчила схему обміну

ключами Діффі-Хеллмана.