1. **Цель работы:** реализовать схему разделения секрета Шамира.

2. Описание алгоритма

2.1. Разделение секрета

Данный алгоритм позволяет разделить секрет между n участниками, причем только k любых из них смогут его восстановить. Также смогут восстановить секрет больше k человек, но не меньше.

Чтобы «спрятать» секрет M, необходимо задать простое число $p > M$. Это число можно сообщить всем n участникам. Число p задает конечное поле, над которым строится многочлен степени $k-1$

$$F(x) = a_{k-1}x^{k-1} + a_{k-2}x^{k-2} + \ldots + a_1x + M mod p$$

где коэффициенты a_iзадаются случайно, и их нужно «забыть» после процедуры разделения секрета.

После того, как был задан многочлен *F(x)*, необходимо вычислить «тени» — значения многочлена в n различны точках, причем $x \neq 0$.

$$k_i = F(i) = a_{k-1}i^{k-1} + a_{k-2}i^{k-2} + \ldots + a_1i + M mod p$$

Аргументы не обязательно должны идти по порядку, главное — чтобы они были все различные по модулю p.

Далее участникам сообщается $k_i, i, p, k-1$.

2.2. Восстановление секрета

Восстановить секрет могут любые k участников, т. к. они знают значение многочлена в k различных точках. Чтобы восстановить исходный многочлен *F(x)*, где и находится секрет в качестве свободного члена, воспользуемся многочленом Лагранжа

$$F(x) = \sum_i l_i(x) y_i mod p$$

$$l_i(x) = \prod_{i \neq j} \frac{x - x_j}{x_i - x_j} mod p$$

где (x_i, y_i)— координаты точек многочлена.

2.3. Особенности системы

- Отсутствие избыточности — тень является секретом.
- Число владельцев части секрета n может быть увеличено до p, при этом количество теней k, необходимых для восстановления секрета, останется прежним.
- Можно пересчитывать тени, оставляя секрет неизменным.
- Можно каждому участнику выдавать по одной тени, либо одному участнику дать несколько теней, если стороны не являются равными между собой.
- В схеме предполагается, что тот, кто генерирует и раздает тени, является надежным, что не всегда так.

3. Описание реализации алгоритма

3.1. Класс Shamir

Основной класс, производящий разделение и восстановление секрета.

```
public BigInteger[] encode (BigInteger secret);
```

Метод принимает на вход секрет и возвращает тени. Также, все необходимые данные записывает в файл. Ниже представлен код данного метода с элементами псевдокода.

```
public BigInteger[] encode (BigInteger secret) throws
                                                FileNotFoundException {
     Random rnd = new Random ();
     int bitLength = secret.bitLength ();
     this.p = сгенерировать простое число длиной  (bitLength + 1);
     BigInteger[] coef = сгенерировать случайные коэффициенты многочлена;
     BigInteger[] projections = new BigInteger[this.n];
     BigInteger[] x = new BigInteger[this.n];
     for (int i = 0; i < this.n; i++) {
         x[i] = BigInteger.valueOf ((long) i + 1);
     }
     for (int i = 0; i < this.n; i++) {
         projections[i] = вычислить значение функции в точке x[i];
         projections[i] = projections[i].add (secret).mod (this.p);
     }
     вывести результаты в файл;
     return projections;
}
```

`public BigInteger decode (int[] nums, String filename);`

Метод принимает на вход номера участников, файл созданный предыдущим методом. На выходе получается секрет.

```
public BigInteger decode (int[] nums, String filename) throws
                                                FileNotFoundException {
     /*
     *   params[..][0] - x
     *   params[..][1] - y
     */
     BigInteger[][] params = загрузить параметры из файла;
     BigInteger[][] lagranzhPolynom = new BigInteger[nums.length][k-1];
     BigInteger[] x = new BigInteger[nums.length];
     for (int i = 0; i < nums.length; i++) {
         x[i] = params[i][0];
     }
     for (int i = 0; i < nums.length; i++) {
         int indexPolynom = i;
         lagranzhPolynom[i] = вычислить многочлен Лагранжа;
     }
     BigInteger[] res = new BigInteger[nums.length];
     for (int i = 0; i < nums.length; i++) {
         res[i] = BigInteger.ZERO;
     }
     res = сложить полученные многочлены Лагранжа вместе;
     BigInteger secret = res[0];
     return secret;
}
```

3.2. Класс HelpFunctions

Класс содержит вспомогательные статические методы.

public static boolean rabinMillerTest (BigInteger n, int checkCount);

Метод выполняет проверку числа на простоту с помощью вероятностного теста Рабина-Миллера. На вход подается число и количество проверок.

public static BigInteger[] getPolynom (int degree, int maxBitLength);

Метод возвращает случайные коэффициенты полинома степени degree.

Метод возвращает значение функции в точке *x* по модулю *p*. Функция представляется в виде коэффициентов *coef*.

```
public static BigInteger getFunctionResult (BigInteger[] coef, BigInteger
                                            x, BigInteger p) {
     BigInteger res = BigInteger.ZERO;
     for (int i = 0; i < coef.length; i++) {
         res = res.add (x.modPow (BigInteger.valueOf ((long) i+1),
                                          p).multiply (coef[i]));
     }
     return res;
}
```

Метод производит умножение двух полиномов по модулю *modul*.

```
public static BigInteger[] polynomMultiply (BigInteger[] a, BigInteger[]
                                            b, BigInteger modul) {
     int size = a.length + b.length - 1;
     BigInteger[] res = new BigInteger[size];
     for (int i = 0; i < size; i++) {
         res[i] = BigInteger.ZERO;
     }
     for (int i = 0; i < a.length; i++) {
         for (int j = 0; j < b.length; j++) {
             res[i+j] = res[i+j].add (a[i].multiply (b[j]));
         }
     }
     for (int i = 0; i < size; i++) {
         res[i] = res[i].mod (modul);
         if (res[i].compareTo (BigInteger.ZERO) < 0) {
             res[i] = res[i].add (modul);
         }
     }
     return res;
 }
```

Метод вычисляет Многочлен Лагранжа.

```
public static BigInteger[] lagranzhPolynom (int polynomIndex,
                                    BigInteger[] x, BigInteger modul) {
     BigInteger[] res = new BigInteger[1];
     res[0] = BigInteger.ONE;
     BigInteger coef = BigInteger.ONE;
     for (int i = 0; i < x.length; i++) {
         if (i != polynomIndex) {
             BigInteger[] tmp = new BigInteger[2];
             tmp[0] = x[i].multiply (BigInteger.valueOf (-1));
```

```
                tmp[1] = BigInteger.ONE;
                res = polynomMultiply (res, tmp, modul);
                coef = coef.multiply (x[polynomIndex].subtract (x[i]));
            }
        }
        coef = coef.modInverse (modul);
        for (int i = 0; i < x.length; i++) {
            res[i] = res[i].multiply (coef).mod (modul);
        }
        return res;
    }
```

Метод складывает два полинома a и b, умножая на скаляр. Операции выполняются по модулю *modul*.

```
    public static BigInteger[] polynomAdder (BigInteger[] a, BigInteger[] b,
                                    BigInteger scalar, BigInteger modul) {
        for (int i = 0; i < b.length; i++) {
            b[i] = b[i].multiply (scalar).mod (modul);
        }
        BigInteger[] x = a;
        BigInteger[] y = b;
        if (a.length < b.length) {
            x = b;
            y = a;
        }
        BigInteger[] res = x;
        for (int i = 0; i < y.length; i++) {
            res[i] = res[i].add (y[i]).mod (modul);
        }
        return res;
    }
```

4. Пример

Пусть мы хотим разделить секрет $M = 12345$между $n = 10$участниками, и любые $k = 5$из них могли его восстановить, число $p = 20947$. Результат вычисления теней приведен на рисунке 1, где указаны общие параметры, в три столбца записаны номер участника (первый столбец), точка x (второй столбец), в которой получен результат y (третий столбец).

Любое количество участников не меньше k восстанавливают секрет. Если взять, например, первых четырех участников и попробовать восстановить секрет, получим $M = 1569$. Так получилось потому, что был восстановлен многочлен 3-ей степени, а при разделении секрета был использован многочлен 4-ой степени.

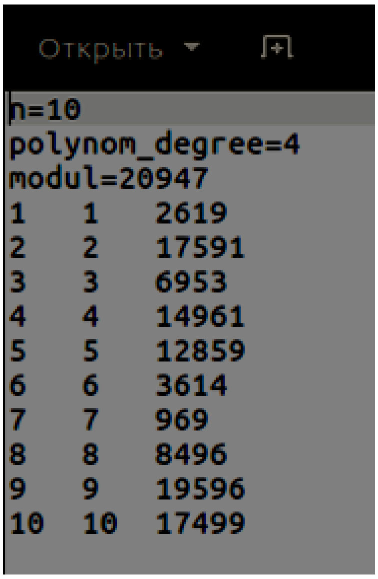

Рис. 1 Результат разделения секрета