Правительство Российской Федерации
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ
«ВЫСШАЯ ШКОЛА ЭКОНОМИКИ»
(НИУ ВШЭ)

Московский институт электроники и математики им. А. Н. Тихонова

ОТЧЕТ
О ПРАКТИЧЕСКОЙ РАБОТЕ № 2

по дисциплине «Криптографические методы защиты информации»
Построение криптографических операций на эллиптических кривых.

Студент гр. БИБ232
Николаев Михаил
«09» февраля 2025 г.

Руководитель
Заведующий кафедрой информационной
безопасности киберфизических систем
канд. техн. наук, доцент
О. О. Евсютин
«___» _____ 2025 г.

Москва 2025

СОДЕРЖАНИЕ

1 Задание на практическую часть

Целью данной работы является приобретение навыков программной реализации операций над точками эллиптических кривых для построения криптографических преобразований.

В рамках практической работы необходимо выполнить следующее:

1) написать программную реализацию инструмента, позволяющего строить и исследовать группы точек эллиптических кривых $E_{a,b}(F_p)$;
2) построить и исследовать группу точек эллиптической кривой $E_{a,b}(F_p)$, $2^{10} \le \left|E_{a,b}(F_p)\right| \le 2^{512}$;
3) подготовить отчет о выполнении работы.

2 Краткая теоретическая часть

2.1 Эллиптические кривые

Эллиптической кривой над конечным полем вычетов по модулю простого числа $p>3$ называется множество точек $(x, y) \in F_p \times F_p$, удовлетворяющих уравнению $y^2 = x^3 + ax + b$, где $a, b \in F_p$ и $-4a^3 - 27b^2 \neq 0 (mod\ p)$, дополненное бесконечно удаленной точкой 0, не имеющей численного выражения. Данное множество точек, обозначаемое $E_{a,b}(F_p)$, представляет собой абелеву группу относительно операции сложения точек.

В общем случае эллиптическая кривая может быть задана над полем Галуа (полем многочленных вычетов) $F_q = F_{p^n}$, однако в данной работе общий случай не рассматривается, поскольку наиболее распространенные криптографические алгоритмы основываются на эллиптических кривых вида $E_{a,b}(F_p)$.

Операция сложения точек эллиптической кривой задается следующим образом. Чтобы сложить точки P и Q, необходимо провести через них прямую, которая в общем случае будет проходить еще через одну точку эллиптической кривой. Эту третью точку необходимо симметрично отразить относительно оси абсцисс, полученный результат и будет представлять собой сумму $P+Q$.

Зная координаты двух исходных точек $P=(x_1, y_1)$ и $Q=(x_2, y_2)$, достаточно легко вывести формулы для нахождения координат третьей точки $C=(x_3, y_3)=P+Q$. При этом необходимо учесть три случая.

Первый случай. Складываются две одинаковые точки $P=(x_1, y_1)$ и $P=(x_1, y_1)$. При выводе координат результирующей точки необходимо воспользоваться уравнением касательной к эллиптической кривой. Формулы для нахождения координат точки $C=(x_3, y_3)=P+P$ имеют вид

$$\begin{cases} ¿\, x_3 = \left(\dfrac{3x_1^2 + a}{2y_1}\right)^2 - 2x_1, \\ ¿\, y_3 = \left(\dfrac{3x_1^2 + a}{2y_1}\right)(x_1 - x_3) - y_1. \end{cases} \quad (1)$$

Второй случай. Складываются две разные точки $P=(x_1, y_1)$ и $Q=(x_2, y_2)$, причем $x_1 \neq x_2$. При выводе координат результирующей точки необходимо воспользоваться уравнением секущей к эллиптической кривой. Формулы для нахождения координат точки $C=(x_3, y_3)=P+Q$ имеют вид

$$\begin{cases} ¿\, x_3 = \left(\dfrac{y_2 - y_1}{x_2 - x_1}\right)^2 - x_1 - x_2, \\ ¿\, y_3 = \left(\dfrac{y_2 - y_1}{x_2 - x_1}\right)(x_1 - x_3) - y_1. \end{cases} \quad (2)$$

Третий случай. Складываются две разные точки, $P=(x_1, y_1)$ и $Q=(x_2, y_2)$, причем $x_1=x_2$, $y_1=-y_2$. Такие точки являются взаимно обратными элементами группы $E_{a,b}(F_p)$, то есть $Q=-P$, поэтому их сумма дает нейтральный элемент группы – бесконечно удаленную точку 0.

2.2 Подсчет числа точек эллиптической кривой

Для построения криптографического алгоритма на базе эллиптической кривой $E_{a,b}(F_p)$ необходимо знать порядок группы точек данной эллиптической кривой $\left|E_{a,b}(F_p)\right|$. Границы, в которых находится данное значение, определяются теоремой Хассе.

Теорема Хассе. $\left|\left|E_{a,b}(F_p)\right| - (p+1)\right| \le 2\sqrt{p}$.

Однако при больших значениях p множество возможных значений $\left|E_{a,b}(F_p)\right|$ также может быть достаточно велико.

Наивный алгоритм подсчета числа точек эллиптической кривой достаточно очевиден. Чтобы найти все точки эллиптической кривой, можно использовать полный перебор, подставляя все возможные значения элементов из F_p в качестве x в уравнение эллиптической кривой и определяя, можно ли извлечь квадрат из полученного значения. При положительном исходе в группу $E_{a,b}(F_p)$ необходимо добавить две точки вида $\left(x; \pm\sqrt{y^2}(mod\ p)\right)$ или одну точку, если $y^2(mod\ p)=0$. Кроме того, в группу $E_{a,b}(F_p)$ необходимо включить бесконечно удаленную точку 0.

Очевидно также и то, что такой подсчет целесообразно использовать лишь при малых значениях p.

Более эффективным является алгоритм больших и малых шагов, также называемый алгоритмом «шаг младенца, шаг великана». Данный алгоритм позволяет вычислить порядок любой точки $P \in E_{a,b}(F_p)$ и обладает сложностью $4\sqrt[4]{p}$. Если точка P является образующим группы $E_{a,b}(F_p)$, то алгоритм больших и малых шагов вычисляет порядок данной группы. В противном случае он вычисляет порядок некоторой подгруппы группы $E_{a,b}(F_p)$. Поэтому для вычисления порядка группы $E_{a,b}(F_p)$ может понадобиться несколько запусков алгоритма больших и малых шагов для случайных точек эллиптической кривой.

В основе алгоритма больших и малых шагов лежат две теоремы. Пусть $\left|E_{a,b}(F_p)\right|=N$ и $P \in E_{a,b}(F_p)$ – произвольная точка эллиптической кривой. Тогда по теореме

Лагранжа $NP=0$. В свою очередь, теорема Хассе указывает, что множество допустимых значений N удовлетворяет неравенству $p+1-2\sqrt{p} \le N \le p+1+2\sqrt{p}$.

Алгоритм больших и малых шагов.

Вход: эллиптическая кривая $E_{a,b}(F_p)$, точка $P \in E_{a,b}(F_p)$.

Выход: $M=O(P)$ или $N=\left|E_{a,b}(F_p)\right|$.

1. Вычисляем $Q \leftarrow (p+1)P$.

2. Выбираем целое $m>\sqrt[4]{p}$. Вычисляем и запоминаем точки jP для $j=\overline{0,m}$.

3. Вычисляем точки $Q+k(2mP)$ для $k=-m,-(m-1),\dots,m$ до тех пор, пока очередное значение $Q+k(2mP)$ не совпадет с некоторой точкой jP (или $-jP$), после чего присваиваем $l \leftarrow -j$ (или $l \leftarrow j$).

4. Полагаем $M \leftarrow p+1+2mk+l$.

5. Выполняем факторизацию M, полагая, что $p_1, p_2, \dots, p_r$ – это различные простые множители числа M.

6. Вычисляем $\left(\frac{M}{p_i}\right)P$ для $i=\overline{1,r}$. Если $\left(\frac{M}{p_i}\right)P=0$ для некоторого значения i, заменяем значение M значением $\frac{M}{p_i}$ и переходим в шагу 5. Если $\left(\frac{M}{p_i}\right)P \neq 0$ для всех $i=\overline{1,r}$, то $O(P)=M$.

7. Для нахождения $\left|E_{a,b}(F_p)\right|$ повторяем шаги 1–6 для случайно выбранных в $E_{a,b}(F_p)$ точек, до тех пор пока наименьшее общее кратное порядков этих точек не будет делить только одно целое N, удовлетворяющее неравенству $p+1-2\sqrt{p} \le N \le p+1+2\sqrt{p}$. Тогда $\left|E_{a,b}(F_p)\right|=N$.

При больших значениях p подсчет числа точек эллиптической кривой может быть произведен с помощью алгоритма Шуфа.

2.3 Теоретико-числовые алгоритмы для реализации криптографических преобразований

2.3.1 Нахождение обратного элемента по модулю простого числа

Формулы (1) и (2) используют операцию деления, под которой в арифметике остатков подразумевается умножение на обратное по модулю значение. Для нахождения обратного значения по модулю натурального числа применяется расширенный алгоритм Евклида.

Вход: целые числа $a \ge b>0$.

Выход: $d=НОД(a,b)$ и целые x, y, такие, что $ax+by=d$.

1. Полагаем $x_2 \leftarrow 1$, $x_1 \leftarrow 0$, $y_2 \leftarrow 0$, $y_1 \leftarrow 1$.

2. Пока $b>0$, выполнять следующее:

2.1. $q \leftarrow \left[\frac{a}{b}\right]$, $r \leftarrow a - qb$, $x \leftarrow x_2 - q\,x_1$, $y \leftarrow y_2 - q\,y_1$;

2.2. $a \leftarrow b$, $b \leftarrow r$, $x_2 \leftarrow x_1$, $x_1 \leftarrow x$, $y_2 \leftarrow y_1$, $y_1 \leftarrow y$.

3. $d \leftarrow a$, $x \leftarrow x_2$, $y \leftarrow y_2$ и возврат (d, x, y).

Чтобы найти $a^{-1} \bmod n$, необходимо подать на вход алгоритма Евклида пару n a, и если $НОД(a, n)=1$, вернуть в качестве a^{-1} значение y_2.

Альтернативный способ вычисления $a^{-1} \bmod n$ основан на теореме Эйлера, которая может быть сформулирована следующим образом.

Теорема Эйлера. Пусть натуральное число $a \in Z_n$. Если $НОД(a, n)=1$, то верно следующее сравнение $a^{\varphi(n)} \equiv 1 (mod\, n)$.

Тогда $a^{-1} \equiv a^{\varphi(n)-1} (mod\, n)$.

2.3.2 Возведение в степень по модулю

Криптографические алгоритмы, основывающиеся на математическом аппарате эллиптических кривых, при их использовании на практике оперируют числами большой битовой длины (или просто большими числами), когда речь идет о сотнях и тысячах бит. Для некоторых операцией над такими числами созданы специальные алгоритмы. В первую очередь, необходимо иметь алгоритм, который позволит осуществлять быстрое возведение в степень по модулю. Данный алгоритм представлен ниже.

Алгоритм возведения в степень по модулю.

Вход: $a, k \in Z_n$, $k=\sum_{i=0}^{t} k_i \cdot 2^i$.

Выход: $a^k \bmod n$.

1. $b \leftarrow 1$. Если $k=0$, то переход к шагу 5.

2. $A \leftarrow a$.

3. Если $k_0=1$, то $b \leftarrow a$.

4. Для $i=\overline{1,t}$ выполняем следующее:

4.1. $A \leftarrow A^2 \bmod n$.

4.2. Если $k_i=1$, то $b \leftarrow (A \cdot b) \bmod n$.

5. Возврат b.

Сложение точек эллиптической кривой осуществляется по аналогичному алгоритму. Если для данной точки $P \in E_{a,b}(F_p)$ необходимо вычислить точку $Q=kP$, то искомая точка представляется в виде $\frac{k}{2}(2P)$ или $P+\frac{k-1}{2}(2P)$ в зависимости от четности числа k. Далее

происходит удвоение точки P по формулам (1), после чего процесс повторяется, пока не будет вычислена искомая точка.

Известны и более быстрые алгоритмы вычисления кратной точки, однако и приведенный алгоритм является достаточно эффективным для его применения на практике.

2.3.3 Тесты целых чисел на простоту

Еще одним важным аспектом эллиптической криптографии является использование простых чисел.

Наиболее развитые вероятностные алгоритмы проверки чисел на простоту основаны на малой теореме Ферма, которая представляет собой следствие из теоремы Эйлера.

Малая теорема Ферма. Пусть p — простое число, $a \neq 0$ и $a \in Z_p$. Тогда верно сравнение $a^{p-1} \equiv 1 (mod\ p)$.

Соотношение, приведенное в теореме, используется в тесте, проверяющем, является ли заданное число составным. Этот тест называют тестом Ферма.

Тест Ферма.

Вход: нечетное число n.

Выход: ответ на вопрос «является ли n простым».

1. Для $i = \overline{1,t}$ выполняем следующее:

    1.1. Выбираем случайное целое число $a \in [2; n-1]$.

    1.2. Вычисляем $r = a^{n-1} mod\ n$ с помощью алгоритма возведения в степень по модулю.

    1.3. Если $r \neq 1$, то возврат «n — составное».

Тест Ферма по основанию a определяет простоту n с вероятностью $\frac{1}{2}$, после t итераций вероятность ошибки составляет $\frac{1}{2^t}$.

3 Программная реализация

Программа реализована на языке Golang и предназначена для исследования эллиптических кривых в конечных полях. Она предоставляет удобный интерфейс для работы с точками эллиптической кривой, их сложения, нахождения порядков и исследования подгрупп.

Основной функционал программы включает:

1) Генерацию эллиптической кривой – вычисление и отображение всех точек на кривой.
2) Операции сложения точек – выполнение групповой операции сложения точек по заданным математическим правилам.
3) Вычисление кратных точек – возможность умножать точку на число (многократное сложение).
4) Определение подгрупп кривой – поиск подгрупп точек, порядок которых является простым числом.

При запуске программы пользователь вводит параметры кривой: a, b и простое число p, определяющее конечное поле F_p. После этого программа строит кривую $E_{a,b}(F_p)$ и выводит список всех найденных точек.

Далее программа выводит по очереди:

1) Сложение двух случайных точек – функция curve.Add(P, Q), вычисляющая новую точку с учетом геометрических свойств эллиптической кривой.
2) Нахождение подгрупп простого порядка – функция curve.FindPointsOfPrimeOrder, определяющая циклические подгруппы с простым числом элементов.
3) Вычисление кратной точки kP – функция curve.DiscreteLog.

3.1 Пример работы программы

В качестве примера работы будет использована кривая с параметрами a=3, b=5 над конечным полем с параметром p=17.

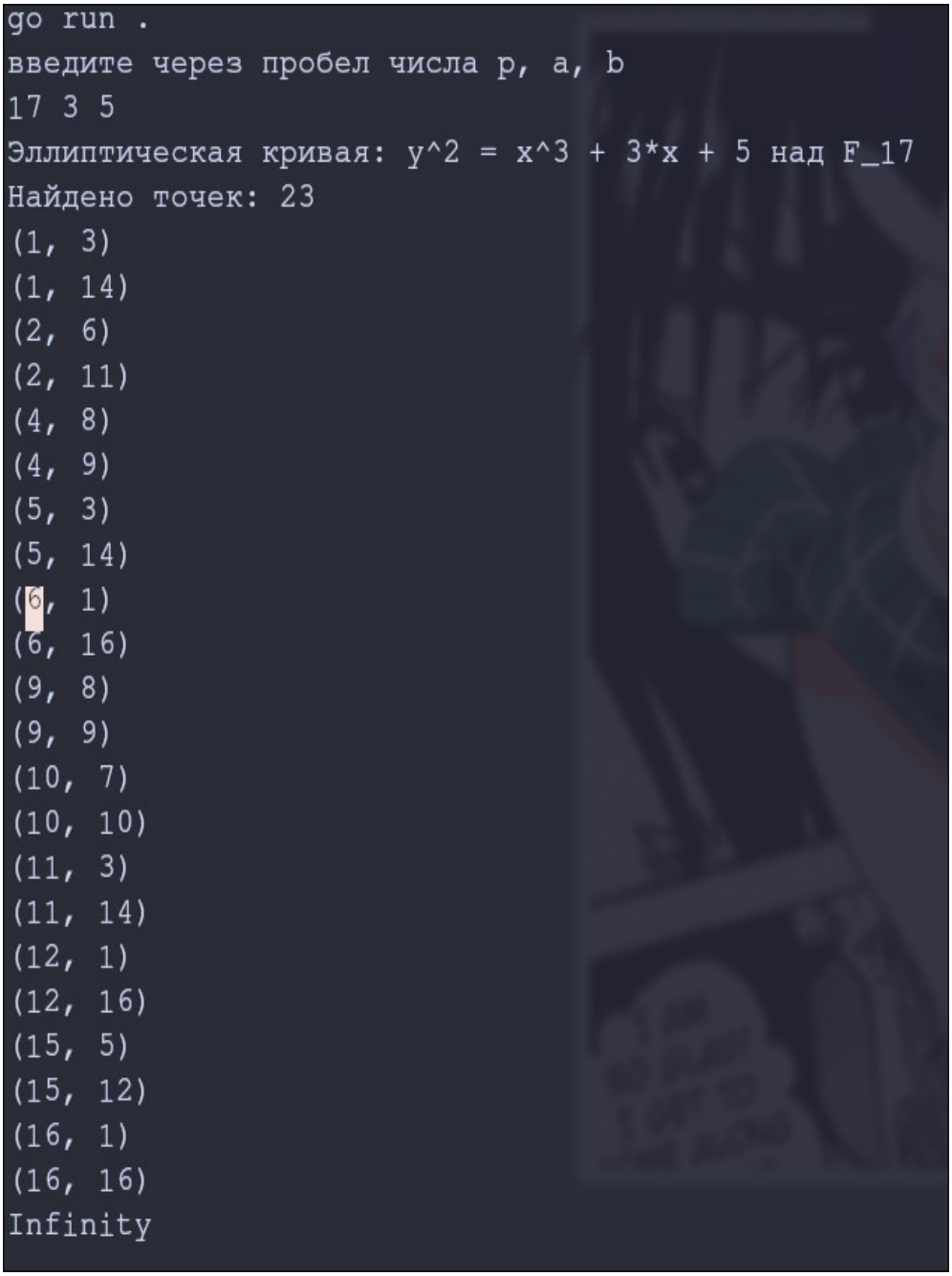

Рисунок 1 – Построение эллиптической кривой и вывод точек

Программа нашла 23 точки на этой кривой (Включая точку “O”).

Далее выберем точку (1149, 487) и произведем ее скалярное умножение, для чего выберем опцию “ Вычислить кратную точку kP”, передав ей координаты точки P (1149, 487) и значение n=15:

```
удалось провесит скалярное умножение: k = 3, P={1 14}, Q={4 9}
```

Рисунок 2 – Нахождение кратной точки kP

Теперь определим все подгруппы простого порядка в группе точек эллиптической кривой. Для этого выбираем опцию под номером “3”, которая автоматически выполняет поиск без необходимости ввода дополнительных аргументов. В результате программа отображает найденные подгруппы. 1

```
Точки простого порядка 23:
1,3
1,14
2,6
2,11
4,8
4,9
5,3
5,14
6,1
6,16
9,8
9,9
10,7
10,10
11,3
11,14
12,1
12,16
15,5
15,12
16,1
16,16
```

Рисунок 3 – Нахождение подгрупп простого порядка группы точек эллиптической кривой

4 Выводы о проделанной работе

В ходе выполненной работы была исследована группа точек эллиптической кривой $E_{a,b}(F_p)$ в конечном поле F_p. Для этого была реализована программа, позволяющая находить все точки кривой, выполнять операции сложения и умножения точек, вычислять их порядок и исследовать подгруппы простого порядка.

Был рассмотрен конкретный пример эллиптической кривой с параметрами a=349, b=673, p=1381 с порядком 1402.

Этот пример подтвердил корректность работы алгоритмов поиска точек, сложения, умножения и исследования структуры группы. Также была проверена возможность нахождения подгрупп с простым порядком, что является важным аспектом в криптографическом применении эллиптических кривых.

В результате работы было подтверждено, что исследуемая математическая модель верно реализована, а её функционал может быть применён в задачах, связанных с криптографией и теорией чисел.

5. Код реализованной программы

```
package main

import (
	"bufio"
	"errors"
	"fmt"
	"math/big"
	"os"
)

type EllipticCurve struct {
	P *big.Int
	A *big.Int
	B *big.Int
}
```

```
type Point struct {
	X	*big.Int
	Y	*big.Int
	Infinity bool
}



func extendedEuclid(a, b *big.Int) (g, x, y *big.Int) {
	zero := big.NewInt(0)
	if b.Cmp(zero) == 0 {
		return new(big.Int).Set(a), big.NewInt(1), big.NewInt(0)
	}

	mod := new(big.Int).Mod(a, b)
	g, x1, y1 := extendedEuclid(b, mod)

	q := new(big.Int).Div(a, b)
	x = new(big.Int).Set(y1)
	y = new(big.Int).Sub(x1, new(big.Int).Mul(q, y1))
	return g, x, y
}



func modInverse(a, mod *big.Int) *big.Int {
	g, x, _ := extendedEuclid(a, mod)
	if g.Cmp(big.NewInt(1)) != 0 {
		return nil
	}
	return new(big.Int).Mod(x, mod)
}
```

```
func modExp(base, exponent, mod *big.Int) *big.Int {
	result := big.NewInt(1)
	baseMod := new(big.Int).Mod(base, mod)
	exp := new(big.Int).Set(exponent)
	zero := big.NewInt(0)
	two := big.NewInt(2)
	for exp.Cmp(zero) > 0 {

		if new(big.Int).And(exp, big.NewInt(1)).Cmp(big.NewInt(1)) == 0 {
			result.Mul(result, baseMod)
			result.Mod(result, mod)
		}
		exp.Div(exp, two)
		baseMod.Mul(baseMod, baseMod)
		baseMod.Mod(baseMod, mod)
	}
	return result
}




func isPrimeFermat(n *big.Int, iterations int) bool {
	one := big.NewInt(1)
	two := big.NewInt(2)
	if n.Cmp(two) < 0 {
		return false
	}
	if n.Cmp(two) == 0 {
		return true
	}
```

```
        if new(big.Int).Mod(n, two).Cmp(big.NewInt(0)) == 0 {
                return false
        }
        nMinusOne := new(big.Int).Sub(n, one)

        for i := 0; i < iterations; i++ {


                a := big.NewInt(int64(2 + i))

                if a.Cmp(new(big.Int).Sub(n, two)) > 0 {
                        a = big.NewInt(2)
                }
                res := modExp(a, nMinusOne, n)
                if res.Cmp(one) != 0 {
                        return false
                }
        }
        return true
}


func (curve *EllipticCurve) IsOnCurve(P Point) bool {

        if P.Infinity {
                return true
        }

        y2 := new(big.Int).Mul(P.Y, P.Y)
        y2.Mod(y2, curve.P)

        x3 := new(big.Int).Exp(P.X, big.NewInt(3), curve.P)
```

```
	ax := new(big.Int).Mul(curve.A, P.X)
	rhs := new(big.Int).Add(x3, ax)
	rhs.Add(rhs, curve.B)
	rhs.Mod(rhs, curve.P)

	return y2.Cmp(rhs) == 0
}


func (curve *EllipticCurve) Add(P, Q Point) Point {

	if P.Infinity {
		return Q
	}
	if Q.Infinity {
		return P
	}
	p := curve.P
	zero := big.NewInt(0)


	if P.X.Cmp(Q.X) == 0 {

		sumY := new(big.Int).Add(P.Y, Q.Y)
		sumY.Mod(sumY, p)
		if sumY.Cmp(zero) == 0 {
			return Point{Infinity: true}
		} else {

			return curve.Double(P)
		}
	}
```

```
        numerator := new(big.Int).Sub(Q.Y, P.Y)
        denominator := new(big.Int).Sub(Q.X, P.X)
        denomInv := modInverse(denominator, p)
        if denomInv == nil {
                return Point{Infinity: true}
        }
        lambda := new(big.Int).Mul(numerator, denomInv)
        lambda.Mod(lambda, p)


        x3 := new(big.Int).Mul(lambda, lambda)
        x3.Sub(x3, P.X)
        x3.Sub(x3, Q.X)
        x3.Mod(x3, p)


        y3 := new(big.Int).Sub(P.X, x3)
        y3.Mul(lambda, y3)
        y3.Sub(y3, P.Y)
        y3.Mod(y3, p)

        return Point{
                X:        x3,
                Y:        y3,
                Infinity: false,
        }
}


func (curve *EllipticCurve) Double(P Point) Point {
        if P.Infinity {
                return P
        }
```

```
p := curve.P
zero := big.NewInt(0)
if P.Y.Cmp(zero) == 0 {
        return Point{Infinity: true}
}

three := big.NewInt(3)
numerator := new(big.Int).Mul(three, new(big.Int).Mul(P.X, P.X))
numerator.Add(numerator, curve.A)
denom := new(big.Int).Mul(big.NewInt(2), P.Y)
denomInv := modInverse(denom, p)
if denomInv == nil {
        return Point{Infinity: true}
}
lambda := new(big.Int).Mul(numerator, denomInv)
lambda.Mod(lambda, p)


x3 := new(big.Int).Mul(lambda, lambda)
twoX := new(big.Int).Mul(big.NewInt(2), P.X)
x3.Sub(x3, twoX)
x3.Mod(x3, p)


y3 := new(big.Int).Sub(P.X, x3)
y3.Mul(lambda, y3)
y3.Sub(y3, P.Y)
y3.Mod(y3, p)

return Point{
        X:        x3,
        Y:        y3,
        Infinity: false,
```

```
        }
}


func (curve *EllipticCurve) ScalarMult(P Point, k *big.Int) Point {
        result := Point{Infinity: true}
        addend := P

        kCopy := new(big.Int).Set(k)
        zero := big.NewInt(0)
        two := big.NewInt(2)
        for kCopy.Cmp(zero) > 0 {
                if new(big.Int).And(kCopy, big.NewInt(1)).Cmp(big.NewInt(1)) == 0 {
                        result = curve.Add(result, addend)
                }
                addend = curve.Double(addend)
                kCopy.Div(kCopy, two)
        }
        return result
}


func (curve *EllipticCurve) Neg(P Point) Point {
        if P.Infinity {
                return P
        }
        negY := new(big.Int).Neg(P.Y)
        negY.Mod(negY, curve.P)
        return Point{
                X:        new(big.Int).Set(P.X),
                Y:        negY,
                Infinity: false,
        }
```

```
}


func (curve *EllipticCurve) Sub(P, Q Point) Point {
	return curve.Add(P, curve.Neg(Q))
}


func pointToString(P Point) string {
	if P.Infinity {
		return "inf"
	}
	return P.X.String() + "," + P.Y.String()
}


func (curve *EllipticCurve) Points() []Point {
	points := []Point{}
	zero := big.NewInt(0)
	one := big.NewInt(1)


	pInt64 := curve.P.Int64()
	for i := int64(0); i < pInt64; i++ {
		x := big.NewInt(i)


		x3 := new(big.Int).Exp(x, big.NewInt(3), curve.P)
		ax := new(big.Int).Mul(curve.A, x)
		fx := new(big.Int).Add(x3, ax)
		fx.Add(fx, curve.B)
		fx.Mod(fx, curve.P)
```

```
                        if fx.Cmp(zero) == 0 {
                                points = append(points, Point{X: new(big.Int).Set(x), Y:
big.NewInt(0), Infinity: false})
                        } else {
                                exp := new(big.Int).Sub(curve.P, one)
                                exp.Div(exp, big.NewInt(2))
                                legendre := modExp(fx, exp, curve.P)
                                if legendre.Cmp(one) == 0 {

                                        for j := int64(0); j < pInt64; j++ {
                                                y := big.NewInt(j)
                                                y2 := new(big.Int).Mul(y, y)
                                                y2.Mod(y2, curve.P)
                                                if y2.Cmp(fx) == 0 {
                                                        points = append(points, Point{X:
new(big.Int).Set(x), Y: new(big.Int).Set(y), Infinity: false})
                                                }
                                        }
                                }
                        }
                }

                points = append(points, Point{Infinity: true})
                return points
        }



        func (curve *EllipticCurve) DiscreteLog(P, Q Point, groupOrder *big.Int) (*big.Int,
error) {

                m := new(big.Int).Sqrt(groupOrder)
```

```
	m.Add(m, big.NewInt(1))


	babySteps := make(map[string]*big.Int)
	current := Point{Infinity: true}
	j := big.NewInt(0)
	one := big.NewInt(1)
	for j.Cmp(m) < 0 {
		babySteps[pointToString(current)] = new(big.Int).Set(j)
		current = curve.Add(current, P)
		j.Add(j, one)
	}

	mP := curve.ScalarMult(P, m)

	current = Q
	i := big.NewInt(0)
	for i.Cmp(m) < 0 {
		if jVal, ok := babySteps[pointToString(current)]; ok {

			k := new(big.Int).Mul(i, m)
			k.Add(k, jVal)
			return k, nil
		}
		current = curve.Sub(current, mP)
		i.Add(i, one)
	}
	return nil, errors.New("discrete log not found")
}


func primeFactors(n int64) []int64 {
	factors := []int64{}
```

```
	for i := int64(2); i*i <= n; i++ {
		for n%i == 0 {
			factors = append(factors, i)
			n /= i
		}
	}
	if n > 1 {
		factors = append(factors, n)
	}
	return factors
}



func (curve *EllipticCurve) PointOrder(P Point, groupOrder *big.Int) *big.Int {
	order := new(big.Int).Set(groupOrder)

	groupOrderInt64 := order.Int64()
	factors := primeFactors(groupOrderInt64)

	factorCounts := make(map[int64]int)
	for _, factor := range factors {
		factorCounts[factor]++
	}
	for q, count := range factorCounts {
		qBig := big.NewInt(q)
		for i := 0; i < count; i++ {
			temp := new(big.Int).Div(order, qBig)

			if curve.ScalarMult(P, temp).Infinity {
				order = temp
			} else {
				break
```

```
                        }
                    }
                }
                return order
            }


            func (curve *EllipticCurve) FindPointsOfPrimeOrder(prime int64, groupOrder *big.Int)
[]Point {
                candidates := []Point{}
                pts := curve.Points()
                target := big.NewInt(prime)
                for _, P := range pts {
                    if P.Infinity {
                        continue
                    }
                    ord := curve.PointOrder(P, groupOrder)
                    if ord != nil && ord.Cmp(target) == 0 {
                        candidates = append(candidates, P)
                    }
                }
                return candidates
            }

            func main() {



                fmt.Println("введите через пробел числа p, a, b")
                in := bufio.NewReader(os.Stdin)
              var pR, aR, bR int64
              fmt.Fscan(in, &pR, &aR, &bR)
              p := big.NewInt(pR)
                a := big.NewInt(aR)
```

```
		b := big.NewInt(bR)
		curve := EllipticCurve{
			P: p,
			A: a,
			B: b,
		}
		fmt.Printf("Эллиптическая кривая: y^2 = x^3 + %s*x + %s над F_%s\n",
a.String(), b.String(), p.String())


		pts := curve.Points()
		fmt.Printf("Найдено точек: %d\n", len(pts))
		for _, pt := range pts {
			if pt.Infinity {
				fmt.Println("Infinity")
			} else {
				fmt.Printf("(%s, %s)\n", pt.X.String(), pt.Y.String())
			}
		}


		groupOrder := big.NewInt(int64(len(pts)))
		fmt.Printf("Порядок группы: %s\n", groupOrder.String())


		if len(pts) >= 2 {
			P := pts[2]
			Q := pts[1]
			R := curve.Add(P, Q)
			fmt.Printf("P = %s\nQ = %s\nP+Q = %s\n", pointToString(P),
pointToString(Q), pointToString(R))
		}
```

```
	if len(pts) > 0 {
		P := pts[0]
		k := big.NewInt(3)
		R := curve.ScalarMult(P, k)
		fmt.Printf("3 * P = %s\n", pointToString(R))
	}


	if isPrimeFermat(p, 5) {
		fmt.Printf("%s является простым числом (по тесту Ферма).\n",
p.String())
	} else {
		fmt.Printf("%s не является простым числом.\n", p.String())
	}

	if len(pts) > 2 {
		P := pts[1]
		kExpected := big.NewInt(3)
		Q := curve.ScalarMult(P, kExpected)
		kFound, err := curve.DiscreteLog(P, Q, groupOrder)
		if err != nil {
			fmt.Println("Не удалось провести скалярное умножение:", err)
		} else {
			fmt.Printf("удалось провесит скалярное умножение: k = %s,
P={%v %v}, Q={%v %v}\n", kFound.String(), P.X, P.Y, Q.X, Q.Y)
		}
	}


	groupOrderInt64 := groupOrder.Int64()
	pFactors := primeFactors(groupOrderInt64)
```

```
	uniquePrimes := make(map[int64]bool)
	for _, q := range pFactors {
		uniquePrimes[q] = true
	}
	for q := range uniquePrimes {
		ptsPrime := curve.FindPointsOfPrimeOrder(q, groupOrder)
		fmt.Printf("Точки простого порядка %d:\n", q)
		for _, pt := range ptsPrime {
			fmt.Println(pointToString(pt))
		}
	}
}
```