1. Министерство науки и высшего образования Российской Федерации
2. Санкт-Петербургский Политехнический Университет Петра Великого
3. —
4. **Институт Кибербезопасности и Защиты Информации**

**ЛАБОРАТОРНАЯ РАБОТА № 8**

1. **«****Разложение числа на множители на эллиптической кривой»**
2. по дисциплине «Криптографические методы защиты информации»
3. Выполнил
4. студент гр. 4851003/80802 Сошнев М.Д.

<*подпись*>

1. Проверил
2. преподаватель Ярмак А.В.

<*подпись*>

1. Санкт-Петербург
2. 2022

# Цель работы

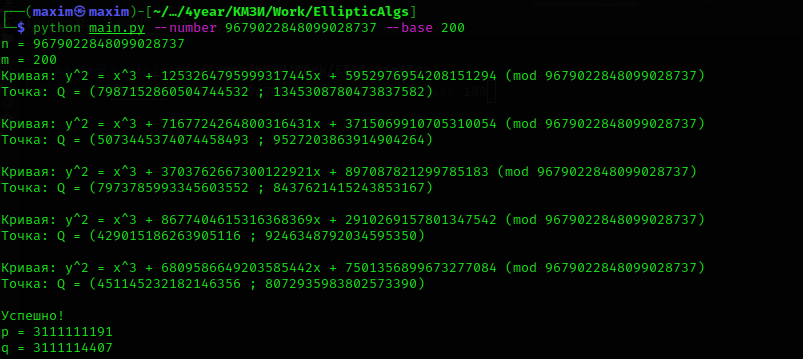
Реализация метода разложения числа на множители с использованием эллиптических кривых.

# Задача

Получить у преподавателя вариант задания и разработать программу **П-1**, которая находит разложение составного числа на эллиптической кривой. В качестве входных данных программа **П-1** должна принимать число , размер базы ; на выходе – возвращать нетривиальный делитель числа

# Ход работы

В результате выполнения работы была разработана программа, которая с помощью алгоритма эллиптических кривых выполняет факторизацию числа. При разработке использовался язык программирования python. Для начала был разработан класс эллиптической кривой и точки н аэллиптической кривой — реализована процедура сложения точек на эллиптической кривой. С помощью быстрого алгоритма возведения в степень была реализована процедура умножения точки эллиптической кривой на число. Далее, с использованием данных классов был реализован сам алгоритм факторизации числа. Его входными данными являются составное число и размер базы. Алгоритм, генерируя различные эллиптические кривые пытается решить задачу факторизации на них (в данной программе различные эллиптические кривые проверяются последовательно, но в перспективе можно разделять данный алгоритм на несколько потоков). Продемонстрируем работу программы на 64-битном числе n=9679022848099028737:

Рисунок 1 — успешный результат работы программы

Алгоритм успешно нашел множитель p = 3111111191

# Контрольные вопросы

1. Как зависит сложность разложения составного числа заданной длины методом эллиптических кривых от числа различных простых делителей числа ?

Алгоритм Ленстры зависит от длины минимального простого делителя — следовательно при увеличении количества делителей их длина будет уменьшаться, а вместе с ней и скорость работы алгоритма.

1. Сравните сложность разложения на эллиптической кривой составного числа вида , где − различные простые числа, и составного числа такой же длины, состоящего из двух различных простых делителей.

Пусть есть два числа следующего вида и одинаковой длины:

Пусть длины множителей и равны, тогда длины множителей и будут также равными.

Следовательно, будет справедливо следующее неравенство относительно длин множителей: . Так как при использовании алгоритма Ленстры быстрее раскладываются те числа, которые имеют простой делитель меньшего размера, то в данном случае сложность разложения будет меньше сложности разложения.

# Вывод

В результате выполнения работы была изучена математическая модель эллиптических кривых и применена на практике — реализован аналог р-1 метода полларда, который с использованием эллиптических кривых находит нетривиальный делитель составного числа.

# Приложение

import random  
import math  
import rsa  
  
import matplotlib.pyplot as plt  
import numpy as np  
  
  
def bits(n):  
 *"""*  
 *Генерирует двоичные разряды n, начиная*  
 *с наименее значимого бита.*  
  
 *bits(151) -> 1, 1, 1, 0, 1, 0, 0, 1*  
 *"""*  
while n:  
 yield n & 1  
 n >>= 1  
  
  
class EllipticCurvePoint:  
 def \_\_init\_\_(self, x, y, curve):  
 self.x = x  
 self.y = y  
 self.\_\_curve = curve  
  
 def \_\_str\_\_(self):  
 return '({} ; {})'.format(self.x, self.y)  
  
 def clone(self):  
 return EllipticCurvePoint(self.x, self.y, self.\_\_curve)  
  
 def \_\_add\_\_(self, other):  
 if self.x == other.x:  
 m = (3 \* self.x \*\* 2 + self.\_\_curve.a) \* rsa.common.inverse(2 \* self.y, self.\_\_curve.n)  
 m %= self.\_\_curve.n  
 else:  
 dx = (self.x - other.x) % self.\_\_curve.n  
 dy = (self.y - other.y) % self.\_\_curve.n  
 m = dy \* rsa.common.inverse(dx, self.\_\_curve.n)  
 m %= self.\_\_curve.n  
  
 x = (m \*\* 2 - self.x - other.x) % self.\_\_curve.n  
 y = (self.y + m \* (x - self.x)) % self.\_\_curve.n  
 y = (-y) % self.\_\_curve.n  
  
 return EllipticCurvePoint(x, y, self.\_\_curve).check\_curve\_exist()  
  
 def \_\_mul\_\_(self, n: int):  
 *"""*  
 *Возвращает результат n \* self, вычисленный*  
 *алгоритмом удвоения-сложения.*  
 *"""*  
result = None  
 addend = self  
  
 for bit in bits(n):  
 if bit == 1:  
 if result is None:  
 result = addend.clone()  
 else:  
 result += addend  
 addend = addend + addend  
  
 return result.check\_curve\_exist()  
  
 def check\_curve\_exist(self):  
 # if pow(self.y, 2, self.\_\_curve.n) != \  
 # (pow(self.x, 3, self.\_\_curve.n) + self.\_\_curve.a \* self.x + self.\_\_curve.b) % self.\_\_curve.n:  
 # print('[WARNING] {} не лежит на кривой {}'.format(self, self.\_\_curve))  
 # pass  
 return self  
  
 def show(self):  
 self.\_\_curve.show(self)  
  
  
class EllipticCurve:  
 def \_\_init\_\_(self, a, b, n):  
 self.a = a  
 self.b = b  
 self.n = n  
  
 def \_\_str\_\_(self):  
 return 'y^2 = x^3 {} {}x {} {} (mod {})'.format(  
 '+' if self.a >= 0 else '-',  
 self.a,  
 '+' if self.b >= 0 else '-',  
 self.b,  
 self.n)  
  
 def \_\_call\_\_(self, x):  
 y\_2 = (x \*\* 3 + self.a \* x + self.b) % self.n  
 Y = []  
 for y in range(self.n):  
 if pow(y, 2, self.n) == y\_2:  
 Y.append(y)  
  
 return Y  
  
 @staticmethod  
 def generate\_curve\_and\_point(n):  
 while True:  
 a = random.randint(0, n - 1)  
 x = random.randint(0, n - 1)  
 y = random.randint(0, n - 1)  
 b = (y \*\* 2 - x \*\* 3 - a \* x) % n  
  
 if (4 \* a \*\* 3 + 27 \* b \*\* 2) % n != 0:  
 break  
  
 ec = EllipticCurve(a, b, n)  
 point = EllipticCurvePoint(x, y, ec)  
  
 return ec, point  
  
 def show(self, point=None):  
 x\_space = np.linspace(0, self.n, self.n+1)  
 x\_show, y\_show = [], []  
 for x in x\_space:  
 Y = self(x)  
 for y in Y:  
 x\_show.append(x)  
 y\_show.append(y)  
  
 plt.scatter(x\_show, y\_show)  
 if point is not None:  
 plt.scatter(point.x, point.y, c='black')  
  
 plt.show()

import argparse  
import rsa.prime  
import threading  
from elliptic\_cryptography import \*  
  
  
def prime\_generator(n):  
 yield 2  
 yield 3  
  
 p, i = 5, n - 2  
 while True:  
 if rsa.prime.is\_prime(p):  
 i -= 1  
 yield p  
 p += 2  
 if i == 0:  
 break  
  
  
class Attack(threading.Thread):  
 def \_\_init\_\_(self, n, m, on\_done, on\_iter):  
 super().\_\_init\_\_()  
 self.\_\_n = n  
 self.\_\_m = m  
 self.\_\_on\_done = on\_done  
 self.\_\_on\_iter = on\_iter  
  
 def run(self):  
 if rsa.prime.is\_prime(self.\_\_n):  
 return self.\_\_on\_done(self.\_\_n, None)  
  
 print('n = {}'.format(self.\_\_n))  
 print('m = {}'.format(self.\_\_m))  
  
 while True:  
 # Генерируем эллиптическую кривую и точку на ней  
  
 ec, Q = EllipticCurve.generate\_curve\_and\_point(self.\_\_n)  
 # ec = EllipticCurve(-1, 3231, n)  
 # Q = EllipticCurvePoint(87, 2, ec)  
  
 self.\_\_on\_iter(ec, Q)  
  
 for i, p in enumerate(prime\_generator(self.\_\_m)):  
 a = int(math.log2(self.\_\_n) / math.log2(p) / 2)  
 try:  
 for j in range(a):  
 Q \*= p  
 except rsa.common.NotRelativePrimeError as ex:  
 self.\_\_on\_done(self.\_\_n, ex.d)  
 return  
  
  
def report(n, p):  
 if (p is not None) and (n % p == 0) and p != n:  
 q = n // p  
  
 print('Успешно!')  
 print('p = {}'.format(p))  
 print('q = {}'.format(q))  
 else:  
 print('Безуспешно')  
  
  
def iter(ec, Q):  
 print('Кривая: {}'.format(ec))  
 print('Точка: Q = {}\n'.format(Q))  
  
  
def main(context):  
 n = int(context.number)  
 m = int(context.base)  
  
 attack = Attack(n, m, on\_done=report, on\_iter=iter)  
 attack.start()  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 parser = argparse.ArgumentParser()  
 parser.add\_argument('--number', required=True)  
 parser.add\_argument('--base', required=True)  
  
 args = parser.parse\_args()  
  
 main(args)