Московский авиационный институт
(государственный технический университет)
Факультет прикладной математики и физики
Кафедра вычислительной математики и программирования

Лабораторная работа №2

по спецкурсу «Криптография»:
Дискретное логарифмирование

Выполнил:	Баскаков О.А.
Группа:	08-306
№ по списку:	2
Преподаватель:	Рисенберг Д.В.
Оценка:	
Дата:	

Москва
2011 г.

Задание

Необходимо написать программу на языке C++, C# или Python, реализующую алгоритм дискретного логарифмирования. Должны поддерживаться числа длиннее 64 бит.

Вариант №3.

ρ-метод Полларда.

Исходные коды на языке Python:

```
#! /usr/bin/python3.1
# -*- coding: utf-8 -*-

def ExtendedGCD(a, b):
	'''
		Из книги Кормана <<Алгоритмы. Построение и Анализ>> стр 966
	'''
	if b == 0:
		return a , 1, 0
	d1, x1, y1 = ExtendedGCD(b, mod(a, b) )
	d, x, y = d1, y1, x1 - (a//b)*y1
	return d, x, y

def generate2(a, b, p, lim = 900500):
	""" a**x == b (mod p) """
	u = [0]
	v = [0]
	z = [1]
	z1 = [1]

	phi_n = EulerPhi(p)

	for i in range(lim):
	#	g = (z[-1]*3)//p
		if  (z[-1] <= 1*p/3): # z[-1]
			u.append( mod(u[-1]+1, phi_n) )
			v.append( mod(v[-1],   phi_n) )
		elif(z[-1] <= 2*p/3):
			u.append( mod(2*u[-1], phi_n) )
			v.append( mod(2*v[-1], phi_n) )
		elif(z[-1] <= 3*p/3):
			u.append( mod(u[-1],   phi_n) )
			v.append( mod(v[-1]+1, phi_n) )
		else:
			print("FAIL")
		z.append( (pow(b, u[-1], p) * pow(a, v[-1], p)) % p )

		if(i%2==0)and(z[i] == z[i//2])and(i>2):
			print( "\nz[",i,"] =", z[i//2],",", z[i])
			delta = u[i]-u[i//2]

			vx = mod(v[i] - v[i//2], phi_n)
			ux = mod(u[i//2] - u[i], phi_n)
			print("vx =", pow(a, vx, p), "; ux =", pow(b ,ux, p) )

			x = solve_diofant(ux, vx, p, a, b)
			return x
	print("Over time")
```

```
def solve_diofant(ux, vx, p, a, b):

      phi_n = EulerPhi(p)

      if( mod(vx , phi_n) == 0 ): return False
      d, nu, mu = ExtendedGCD(vx , phi_n)
      # nu = v^(-1) mod n
      '''
            a^ux == b^vx   mod p
            a^(ux*nu) = b^(v*nu)
            b ^(d - phi_n*nu) = b^(d) = g(x*d) mod p
            x*d =  ux*nu + w*phi_n
      '''

      for w in range(d+1):
            x = mod( (ux * nu + w * phi_n)//d, phi_n)
            if( pow(a, x, p) == b):
                  print("solved by extend_GCD")
                  return x

      return False


def premutive_log(a, b, n, lim = 100500):
      z = []
      for x in range(min(n, lim)):
            if(pow(a,x,n) == b%n):
                  z.append(x)
      return z
```

Тестирование:

```
oleg@debian:~/crypto$ ./2.py -help
This programm use Pollard's pho algorithm
for solving discrete logarithm problem.

-help for this text
a b n - solve a^x == b (mod n)
none - standart test

oleg@debian:~/crypto$ ./2.py
a = 13
b = 925174080
n = 2148568033

z[ 45679 ] = 113347378 , 113347378
b^vx = 1044654178 ; a^ux = 1044654178
solved by extend_GCD

z[ 106090 ] = 654455104 , 654455104
b^vx = 417976672 ; a^ux = 417976672
solved by extend_GCD

x  = []
x1 = 917771
x2 = 917771
```

Для GF 2148568033 обе реализации дают правильный ответ.

```
oleg@debian:~/crypto$ ./2.py 7 1111 4096

z[ 46 ] = 2913 , 2913
b^vx = 1 ; a^ux = 1

z[ 52 ] = 3383 , 3383
b^vx = 2071 ; a^ux = 2071
solved by extend_GCD

x  = [483, 995, 1507, 2019, 2531, 3043, 3555]
x1 = False
x2 = 2019
```

На малых тестах наивный алгоритм оказывается даже более эффективным, выдавая все возможные решения. При этом последовательность метода Полларда может зациклиться.

```
oleg@debian:~/crypto$ ./2.py 3 13 625

z[ 26 ] = 144 , 144
b^vx = 621 ; a^ux = 621
solved by extend_GCD

z[ 92 ] = 4 , 4
b^vx = 211 ; a^ux = 211
solved by extend_GCD

x  = [277]
x1 = 277
x2 = 277
```

Получили единственное решение.

```
oleg@debian:~/crypto$ ./2.py 41 13666 2148568033

z[ 6447 ] = 1377749731 , 1377749731
b^vx = 480262955 ; a^ux = 480262955
solved by extend_GCD

z[ 69772 ] = 1338952159 , 1338952159
b^vx = 340537948 ; a^ux = 340537948
solved by extend_GCD

x  = []
x1 = 879017484
x2 = 879017484
```

На сложных примерах метод Полларда значительно эффективнее.
В наивном методе стоит отсечение по времени, хотя он всегда находит ответ.

Выводы

Дискретное логарифмирование (обращения функции a^x в конечной мультипликативной группе GF(p^n)) имеет важные приложения в криптографии. На ее сложности базируется криптография с открытым ключом. На данный момент существуют только суб- и экспоненциальные алгоритмы решения, одним из которых является ρ-метод Полларда. При выполнении лаб. работы я убедился, что такой скорости недостаточно для работы с большими числами порядка 2^256.