*Лабораторная работа №5*

*студент группы ИТз-41*

*Котик Дмитрия Степановича*

*Выполнение:\_\_\_\_\_\_\_\_\_ Защита:\_\_\_\_\_\_\_\_\_\_\_*

**Алгоритм обмена ключами Диффи-хеллмана**

Цель работы: изучить и реализовать на практике алгоритм обмена

ключами Диффи-Хеллмана.

## Содержание работы

## Реализовать приложение, позволяющее выполнять следующие действия:

## Изучить схему обмена ключами Диффи-Хелмана.

## Реализовать подпрограмму, определяющую для заданного числа первые 100 первообразных корней, отображая при этом суммарное время, затраченное на их поиск.

## Реализовать подпрограмму, моделирующую обмен ключами между абонентами по схеме Диффи-Хеллмана. Программа должна получать большие простые числа XA, XB и n случайным образом с помощью алгоритма генерации простого числа, а также предоставлять пользователю возможность задавать их.

## Основные понятия

**Криптографический протокол** – это набор формализованных правил, описывающий последовательность действий, исполняемые двумя и более лицами, для решения задачи защиты информации с использованием криптографии. Одна из основных проблем криптографии – передача сообщения по незащищенному каналу связи. При симметричном шифровании ключи шифрования ценны также как и само передаваемое сообщение. Отсюда следует проблема распределения ключей. Надежность любой симметричной криптографической системы во многом зависит от выбранной схемы распределения ключей.

## Ход работы

Разработаем программы:

* + Получение первообразны корней.py
  + Алгоритм обмена ключами Диффи-Хеллмана.py

Получение первообразных ключей.py:

# -- Программа реализации алгоритма генерации первообразных ключей --

def algoritm\_Euclid(a, b):

'''An implementation of extended Euclidean algorithm.

Returns integer x, y and gcd(a, b) for Bezout equation:

ax + by = gcd(a, b).

'''

x0, x1, y0, y1 = 1, 0, 0, 1

step = 0

while b:

# 12741

q = a // b

a, b = b, a % b

x0, x1 = x1, x0 - x1\*q

y, y1 = y1, y0 - y1\*q

step += 1

# return (a, x0, y0)

return a

# Получение первообразных корней

def primitive\_root(m): # функция принимает модуль m

result = []

counter = 0

required\_set = set(num for num in range (1, m) if algoritm\_Euclid(num, m) == 1)

# создаем множество из num для которых НОД (num и m) равны 1

for g in range(1, m): # g от 1 до m

actual\_set = set(pow(g, powers) % m for powers in range (1, m))

if required\_set == actual\_set:

result.append(g)

counter += 1

if counter == 100:

return result

# print(g)

return result

# Точка входа для основной программы

def main():

m = int(input("Введите m = "))

result = primitive\_root(m)

k = 0

# 317

for i in result:

print("{0}: {1}".format(k+1, i))

k += 1

# Вызов программы

main()

Алгоритм Диффи-Хеллмана.py:

# -- Программа реализации алгоритма обмена ключами Диффи-Хеллмана --

from gen\_simple\_numbers import gen\_sp, test\_rabin\_miller

from gen\_primitive\_root import pollard, root

# Генерируем общеизвестные получаем n, Xa, Xb

def get\_start\_data():

choice = int(input("x, Xa, Xb - вводим с клавиатуры, либо генерируем случайным образом? (1/2): "))

if choice == 1:

while True:

n = int(input("Введите x: "))

if test\_rabin\_miller.test\_Rabin\_Miller(n,5) == True:

break

else:

print("х должен быть простым.")

Xa = int(input("Введите Xa: "))

Xb = int(input("Введите Xb: "))

else:

n = gen\_sp.gen\_sp(64)

Xa = gen\_sp.gen\_sp(40)

Xb = gen\_sp.gen\_sp(40)

return n, Xa, Xb

def get\_root(n):

list\_dividers = list(set(pollard.get\_factor\_n(n-1)))

print(list\_dividers)

for g in range(2, n):

g = root.is\_root(g, list\_dividers, n)

if g:

return g

# Точка входа

def main():

print("---"\*20)

n, Xa, Xb = get\_start\_data()

print("---"\*20)

print("n = {0}".format(n))

print("Xa = {0}".format(Xa))

print("Xb = {0}".format(Xb))

print("---"\*20)

g = get\_root(n)

print("g = {0}".format(g))

print("---"\*20)

Ya = root.fast\_pow(g, Xa, n)

Yb = root.fast\_pow(g, Xb, n)

print("Ya = {0}".format(Ya))

print("Yb = {0}".format(Yb))

print("---"\*20)

# Вычисление секретных ключей

Ka = root.fast\_pow(Yb, Xa, n)

Kb = root.fast\_pow(Ya, Xb, n)

print("Ka = {0}".format(Ka))

print("Kb = {0}".format(Kb))

print("Равны ключи?: {0}".format(Ka==Kb))

print("---"\*20)

main()

root.py:

def fast\_pow(base, degree, module):

degree = bin(degree)[2:]

r = 1

for i in range(len(degree) - 1, -1, -1):

r = (r \* base \*\* int(degree[i])) % module

base = (base \*\* 2) % module

return r

def is\_root(g, list\_dividers, n):

for num in list\_dividers:

power = int((n-1) / num)

res = fast\_pow(g, power, n)

if res == 1:

return False

return g

pollard.py:

from random import randint

from time import perf\_counter

import random

def gcd(a, b):

'''An implementation of extended Euclidean algorithm.

Returns integer x, y and gcd(a, b) for Bezout equation:

ax + by = gcd(a, b).

'''

x0, x1, y0, y1 = 1, 0, 0, 1

step = 0

while b:

# 12741

q = a // b

a, b = b, a % b

x0, x1 = x1, x0 - x1\*q

y, y1 = y1, y0 - y1\*q

step += 1

# return (a, x0, y0)

return a

def miller\_rabin\_test(a, n, s, d):

"""Один тест Миллера-Рабина.

Является ли число 2 <= a <= n-2 свидетелем простоты для числа n (n-1 = d \* 2\*\*s)

Если вдруг среди чисел a\*\*(d\*2\*\*0), ..., a\*\*(d\*2\*\*s) перед какой-то 1 идёт не -1,

то число n — составное, а a — не является свидетелем простоты."""

x = pow(a, d, n) # x = a\*\*(d\*2\*\*0)

if x in (1, n - 1):

return True

for r in range(s - 1):

x = (x \* x) % n # Из числа a\*\*(d\*2\*\*r) вычисляем a\*\*(d\*2\*\*(r+1))

if x == 1: # Ну всё, число составное

return False

elif x == n - 1: # Нашлась -1. Число a — свидетель простоты для n

return True

return False # Даже тест Ферма не пройден: a\*\*(n-1) != 1

def miller\_rabin(n):

"""Проверяет простоту числа n>3, выполняя log2(n) тестов Миллера-Рабина"""

# Ищем разложение n-1 = n-1 = d \* 2\*\*s

s, d = 0, n - 1

while d % 2 == 0:

s, d = s + 1, d // 2

for \_ in range(n.bit\_length()): # Повторяем тест log2(n) раз.

a = randint(2, n - 2) # берём случайное число

if not miller\_rabin\_test(a, n, s, d): # Если тест не пройден, то число составное

return False

return True # С вероятностью 1/n\*\*2 число простое

def pollards\_rho\_iter(n):

"""Поиск делителя у нечётного составного числа"""

# Начинаем с функции F(x) = x\*\*2 + 1 из точки x=2. Обычно это срабатывает

x = y = 2

a = 1

while True:

d = 1

while d == 1: # Если 1 < d < n, то мы нашли делитель d

x = (x \*\* 2 - a) % n # x = F(F(..(F(x)..))

y = (y \*\* 2 - a) % n

y = (y \*\* 2 - a) % n # y = F(F(F(F(..(F(F(x))..)))) (в два раз больше раз F)

d = gcd(n, abs(x - y))

if d < n: # Если 1 < d < n, то мы нашли делитель d

return d

# Редко бывает так, что для функции x\*\*2 + 1 при старте с 2 делитель не находится

# В этом случае используем F(x) = x\*\*2 + a, и стартуем с другого случайного числа

x = y = randint(1, n - 1)

a = randint(-100, 100) % n

def factor(n):

"""Факторизация числа при помощи ро-метода Полларда и тестов Миллера-Рабина"""

# Избавляемся от всех двоек, троек и пятёрок

ans = []

for x in (2, 3, 5):

while n % x == 0:

ans.append(x)

n //= x

if n == 1:

pass # n = 2\*\*a \* 3\*\*b \* 5\*\*c

elif miller\_rabin(n): # Остаток простой

ans.append(n)

else: # Ищем делители

d = pollards\_rho\_iter(n)

ans.extend(factor(d))

ans.extend(factor(n // d))

return sorted(ans)

# nums = [10, 437, 3127, 23707, 1752967, 6682189, 12659363, 494370889, 1435186847, 11843161246077928296]

# for num in nums:

# st = perf\_counter()

# fct = factor(num)

# en = perf\_counter()

# print('Число {} разложено на множители {} за {:02f}c'.format(num, fct, en-st))

def get\_factor\_n(n):

fct = factor(n)

return fct

## Вывод:

В данной лабораторной работе был реализован алгоритм обмена ключами Диффи-Хеллмана. Однако стоит отметить, что данный алгоритм актуален только в том случаем, когда канал лишь прослушивается злоумышленником, а не когда злоумышленник является посредником. Если злоумышленных будет являться посредником – будет возможность подменить данные, и получиться секретный ключ.