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

Лабораторная работа №1

по спецкурсу «Криптография»:
Генерация простых чисел

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 проверки на простоту и генерации простых чисел.
Вариант №3.

Проверка на простоту с использованием полного разложения $n-1$ на простые множители (тест Люка).

Исходные коды на языке Python:

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

def constr(L):
	return product( [ (p**k) for (p,k) in L ] )

#~ Тест Люка принимает на вход полное разложение
def Lukas_test(L, k = 33):
	n1 = constr(L)
	n = n1+1
	for (p,m) in L:
		find_ai = False
		for i in range(k):
			a_i = randrange(2, n)
			t1 = pow(a_i, n1  , n)
			t2 = pow(a_i, n1//p, n)
			if (t1 == 1)and(t2 != 1):
				find_ai = True
				break
		if (find_ai == False): return False
	return True

#~ Тест Миллера-Рабина используется для тестирования во всех лаб. работах
def Miller_Rabin(n, k = 256):
	"""Primality test"""
	s = 0
	d = n-1
	while d%2 == 0:
		s += 1
		d //= 2
	for i in range(k):
		a = randrange(2, n-2)
		x = pow(a, d, n)
		if (x==1)or(x==n-1): continue
		for r in range(1, s):
			x = x*x % n
			if x==1   : return False
			if x==n-1 : break
		if (x==n-1) : continue
		return False
	return True
```

```
#~ Генерация происходит последовательным добавлением простых чисел в список
#~ На каждой итерации перебираем подмножества простых чисел
#~ И проверяем на простоту их произведение + 1
def gener( p, lim, st = 1):
    """ st = speed of enlargement """
    st = 2 ** st
    z = 1
    itt = 1
    while (p[-1] < lim):
        #~ print("iter", itt)
        if (itt > 100): return 0
        itt+=1
        h = (1 << len(p))
        for i in range (h//2+st, h, st):
            l = get_bits(i)

            z = [(pp, 1) for (pp,bb) in zip(p,l) if (bb==1)]
            if ( z[0] != (2,1) ):
                z.insert( 0, (2,1) ) #palubas chetnoe
            else:
                z[0] = (2,2)

            n1 = constr(z) + 1
            if( Lukas_test(z) ):
                if not(Miller_Rabin(n1)): print("FAIL"); return 0
                p.append(n1)
                break

    return n1

#~ Подмножество с номером n
def get_bits( n):
    L = []
    while( n>0):
        L.append( n&1 )
        n = n >> 1
    return L

#~ Факторизация
def prima(n):
    L = []
    for i in range(2, n+1):
        k = 0
        while (n%i == 0):
            n //= i
            k+=1
        if(k>0):
            L.append( (i,k) )
        if (n == 1): break
    return L
```

Тестирование:

```
oleg@debian:~/crypto$ ./1.py -help
Lucas primality test
-help for this text
-test (num) for test number
-gen (lim) for generate number
none - standart test
oleg@debian:~/crypto$ ./1.py
n = 45300907
True
lim = 10050013452347853422436578923456234954278978
num = 17158628206615839761062631550244248971923790671
oleg@debian:~/crypto$ ./1.py gen
100500
784183
oleg@debian:~/crypto$ ./1.py test 784183
True
```

Внимание!! т.к. тест Люка использует полное разложение, он достаточно медлительный при тестировании больших чисел. При генерации это не сказывается.

Выводы

Простые числа в силу своих «магических» свойств получили широкое распространение. Поэтому так важно проверять и генерировать их. Большинство алгоритмов являются вероятностными, т.е. дают не 100% гарантию полученного результата. Но увеличивая количество итераций можно добиться, чтобы величина $1 - 2^{-n}$ была очень близка к 1. На мой взгляд, самым эффективным является тест Миллера-Рабина, поэтому он будет использоваться повсеместно там, где идет работа с простыми числами.