Министерство образования и науки Российской Федерации
Новосибирский государственный технический университет
Кафедра прикладной математики

Основы криптографии

Лабораторная работа №2

Факультет: прикладной математики и информатики
Группа: ПМ-63
Студенты: Шепрут И.М.
Кожекин М.В.
Утюганов Д.С.
Преподаватель: Ступаков И.М.

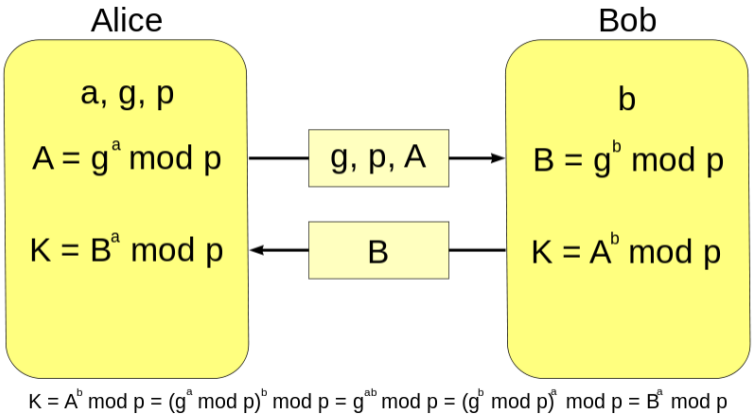
Новосибирск
2018

1. Цель работы

При помощи алгоритма Диффи-Хелмана произвести обмен публичными ключами и получить общий секретный ключ.

2. Анализ задачи

Криптографическая стойкость алгоритма Диффи — Хеллмана (то есть сложность вычисления $K = g^{ab} \bmod p$ по известным p, g, $A = g^b \bmod p$ и $B = g^a \bmod p$), основана на **предполагаемой** сложности задачи дискретного логарифмирования.



$K = A^b \bmod p = (g^a \bmod p)^b \bmod p = g^{ab} \bmod p = (g^b \bmod p)^a \bmod p = B^a \bmod p$

3. Текст программы

Для удобства программа была разбита на 3 файла:
DH.h – описание класса алгоритма Диффи-Хеллмана
DH.cpp – реализация методов класса
main.cpp – основная программа

Часть 1. Обычная 64-битная арифметика (uint64_t)

```
#include <iostream>
#include <random>
#include <cmath>

using std::cin;
using std::cout;
using std::endl;


// Алгоритм быстрого возведения в степень в поле вычета mod
uint64_t fastPow(uint64_t num, uint64_t deg, uint64_t mod) {

	uint64_t result = 1;
	uint64_t bit = num % mod;

	while (deg > 0) {

		if ((deg & 1) == 1) {
			result *= bit;
			result %= mod;
		}

		bit *= bit;
		bit %= mod;
		deg >>= 1;
```

```
	}
	return result;
}


// Проверка числа на простоту при помощи решета Эрастофена
bool isPrime(uint64_t num) {

	if (num <= 3) {                    // 2 и 3 простые
		return num > 1;            // а 1 - нет
	}
	else if (num % 2 == 0 || num % 3 == 0) {
		return false;
	}
	else {
		for (int i = 5; i * i <= num; i += 6) {
			if (num % i == 0 || num % (i + 2) == 0) {
				return false;
			}
		}
		return true;
	}
}


// Генерация случайного числа
uint64_t getRandomNumber() {

	static std::mt19937 generator(917401);
	static std::uniform_int_distribution<> distribution(0, INT32_MAX);
	return distribution(generator);
}


// Генерация простого числа длинной LENGTH бит
uint64_t getPrimeNumber() {

	uint64_t n;
	do {
		n = getRandomNumber();
	} while (!isPrime(n));
	return n;
}


// Поиск p и k
void calc_p_and_k(uint64_t &p, uint64_t &k) {

	int i = 0;
	k = 1, p = 4;
	while (!isPrime(p)) {
		k = getPrimeNumber();
		p = 2 * k + 1;
	}
}


// Поиск примитивного корня g
uint64_t calcPrimitiveRoot(uint64_t p, uint64_t k) {

	uint64_t g;
	while (true) {

		g = getPrimeNumber();
		if (fastPow(g, 2, p) == 1 || fastPow(g, k, p) == 1)
```

```
                        continue;
                return g;
        }
}


void main() {

        data p, g, B;
        cout << "Enter p: ";
        cin >> p;
        cout << "Enter g: ";
        cin >> g;

        data a = getRandomNumber();
        cout << "a = " << a << endl;
        data A = fastPow(g, a, p);
        cout << "A = " << A << endl;

        cout << "Enter B: ";
        cin >> B;

        data K = fastPow(B, a, p);
        cout << "K = " << K << endl;
}
```

Часть 2. Длинная арифметика (cpp_int)

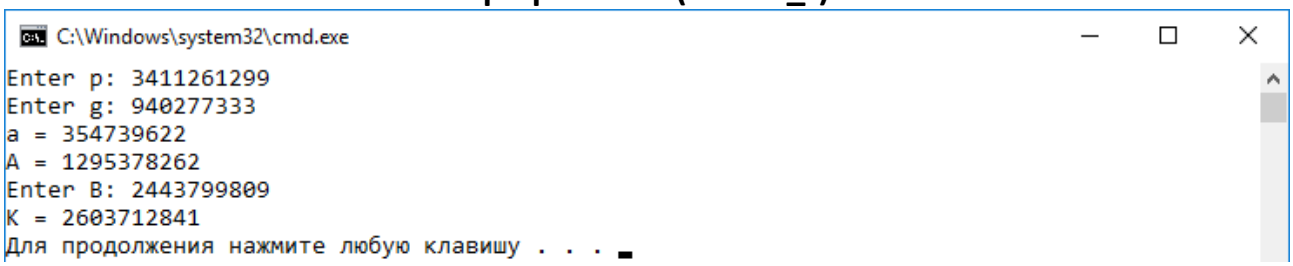
```
#include <iostream>
#include <random>
#include <boost/multiprecision/cpp_int.hpp>
#include <boost/multiprecision/miller_rabin.hpp>
#include <boost/random/mersenne_twister.hpp>
#include <boost/random.hpp>

using namespace boost::multiprecision;
using std::cin;
using std::cout;
using std::endl;

const int ITER = 25;
const int LENGTH = 80;


// Алгоритм быстрого возведения в степень в поле вычета mod
cpp_int fastPow(cpp_int &num, const cpp_int &deg_param, const cpp_int &mod) {

        cpp_int result = 1;
        cpp_int bit = num % mod;
        cpp_int deg = deg_param;

        while (deg > 0) {

                if ((deg & 1) == 1) {
                        result *= bit;
                        result %= mod;
                }

                bit *= bit;
                bit %= mod;
                deg >>= 1;
        }
        return result;
```

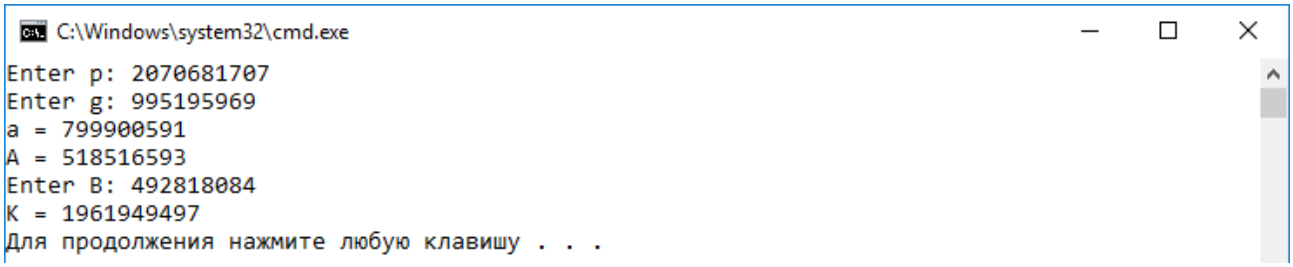
```
}


// Генерация случайного числа
cpp_int getRandomNumber() {

	static boost::random::mt19937 generator(917401);
	static boost::random::uniform_int_distribution<cpp_int> distribution(0, cpp_int(1)
<< LENGTH);
	return distribution(generator);
}


// Генерация простого числа длинной LENGTH бит
cpp_int getPrimeNumber() {

	cpp_int n;
	do {
		n = getRandomNumber();
	} while (!miller_rabin_test(n, ITER));
	return n;
}


// Поиск p и k
void calc_p_and_k(cpp_int &p, cpp_int &k) {

	int i = 0;
	k = 1, p = 4;
	while (!miller_rabin_test(p, ITER)) {
		k = getPrimeNumber();
		p = 2 * k + 1;
		cout << ++i << "\r";
	}
	cout << "Iterations count:" << i << endl;
}


// Поиск примитивного корня g
cpp_int calcPrimitiveRoot(cpp_int p, cpp_int k) {

	cpp_int g;
	while (true) {

		g = getPrimeNumber();
		if (fastPow(g, 2, p) == 1 || fastPow(g, k, p) == 1)
			continue;
		return g;
	}
}


void main() {

	data p, g, B;
	cout << "Enter p:" << endl;
	cin >> p;
	cout << "Enter g:" << endl;
	cin >> g;

	data a = getRandomNumber();
	cout << "a:" << endl << a << endl;
	data A = fastPow(g, a, p);
	cout << "A:" << endl << A << endl;
	cout << "Enter B:" << endl;
```

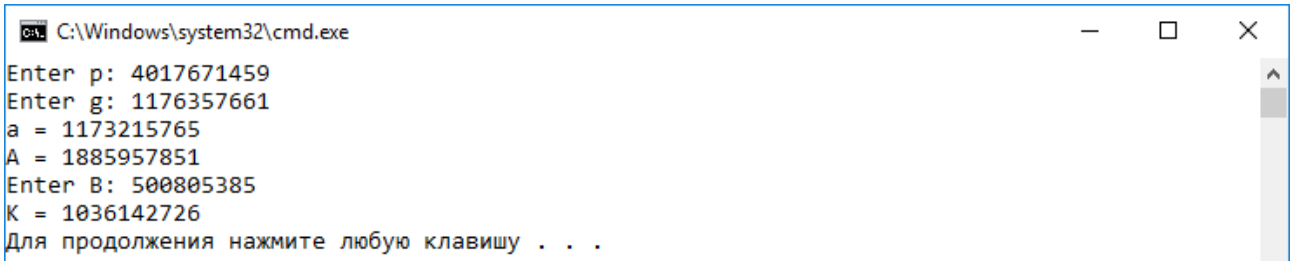
```
        cin >> B;

        data K = fastPow(B, a, p);
        cout << "K:" << endl << K << endl;
}
```

4. Результат работы программы

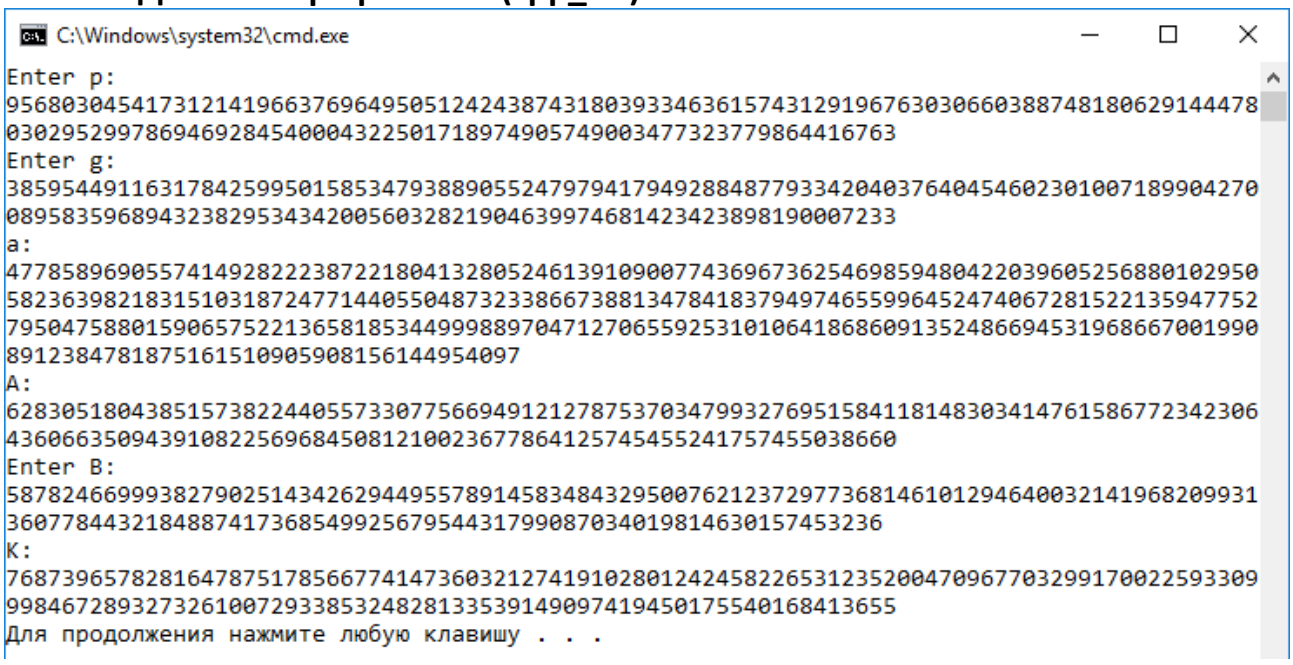
Часть 1. Обычная 64-битная арифметика (uint64_t)



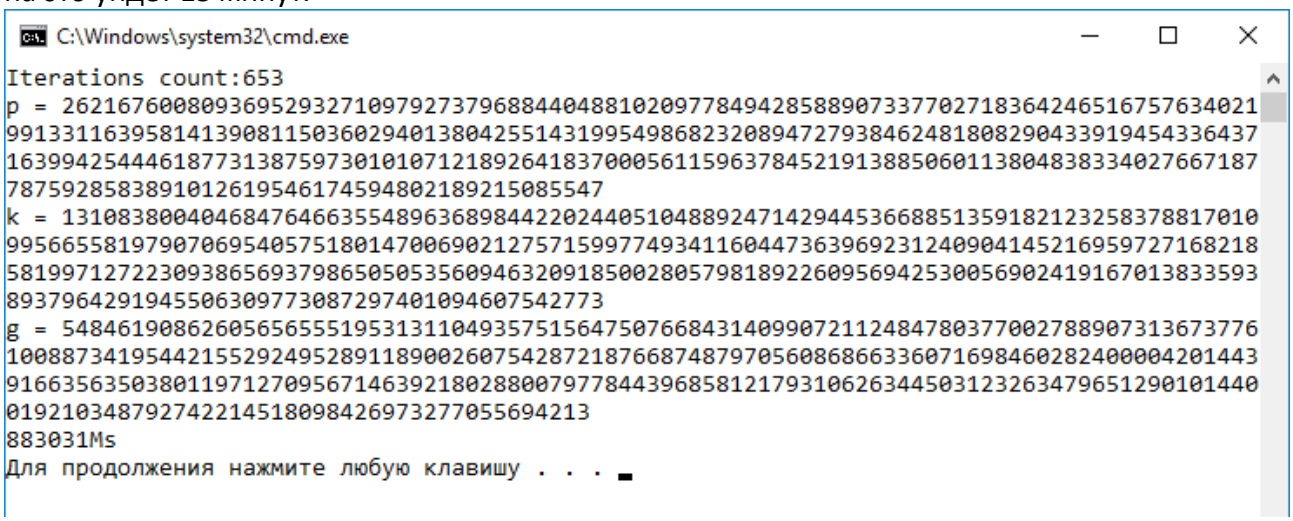




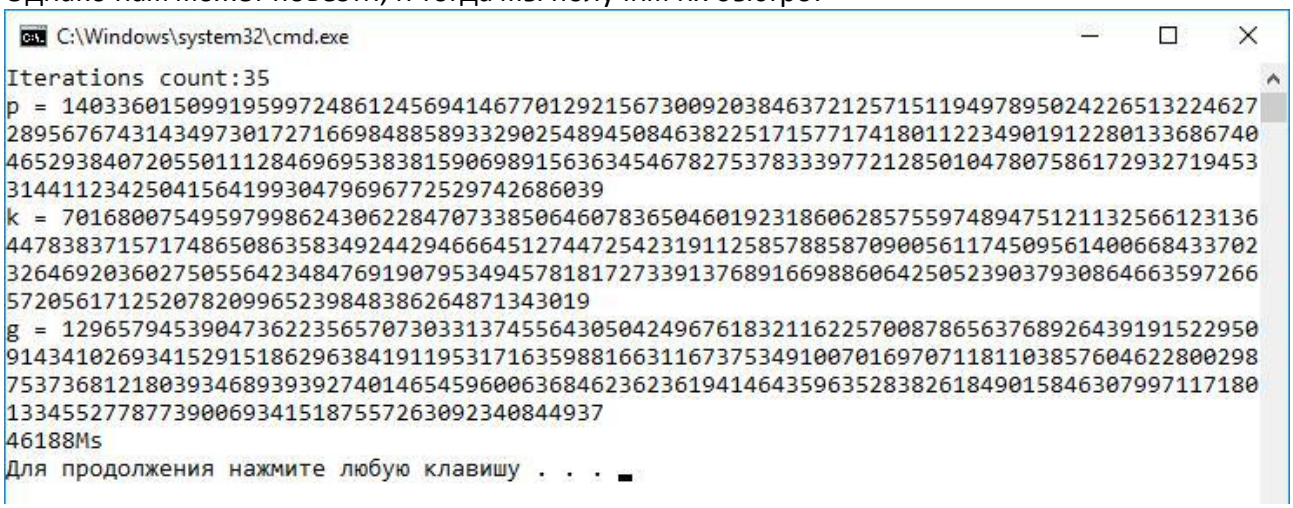
Часть 2. Длинная арифметика (cpp_int)



Единственным недостатком оказалось то, что если искать примитивный корень самому, то на это уйдёт 15 минут.



Однако нам может повезти, и тогда мы получим их быстро:



5. Выводы

В ходе выполнения 2 лабораторной работы был реализован алгоритм Диффи-Хелмана для генерации общего секретного ключа при помощи библиотеки boost версии 1.68, а точнее модулей длинной арифметики multiprecision и работы со случайными числами random:

https://www.boost.org/doc/libs/1_68_0/libs/multiprecision/doc/html/index.html

https://www.boost.org/doc/libs/1_68_0/doc/html/boost_random.html