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

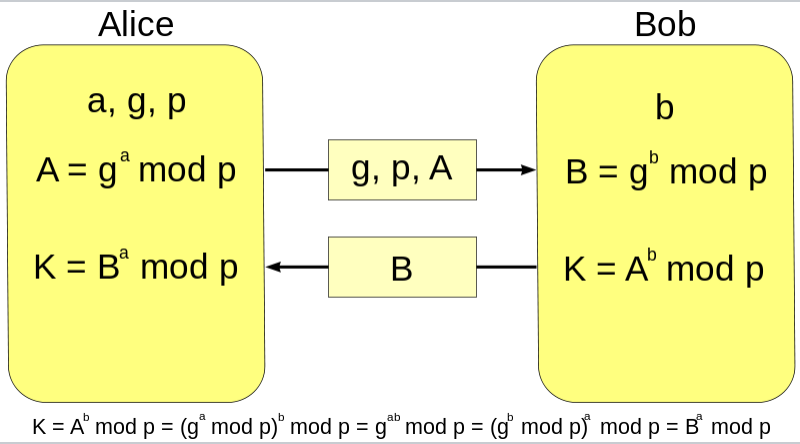
2018

1. Цель работы

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

1. Анализ задачи

Криптографическая стойкость алгоритма Диффи — Хеллмана (то есть сложность вычисления K = gab mod p K = g a b mod p {\displaystyle K=g^{ab}{\bmod {p}}} Kпо известным p, g, A = gb mod p A = g a mod p {\displaystyle A=g^{a}{\bmod {p}}} , Aи B = g b mod p {\displaystyle B=g^{b}{\bmod {p}}} B = ga mod p), основана на **предполагаемой** сложности задачи дискретного логарифмирования.



1. 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)**

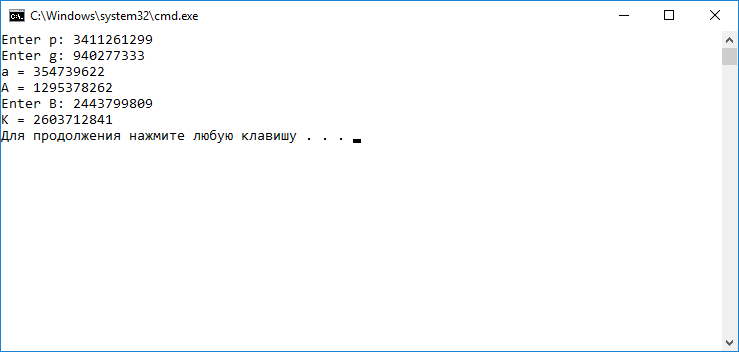
|  |
| --- |
| #include <iostream>  #include <random>  #include <cmath>  using std::cin;  using std::cout;  using std::endl;  // Алгоритм быстрого возведения в степень в поле вычета mod  uint64\_t fastPow(uint64\_t num, uint64\_t deg, uint64\_t mod) {  uint64\_t result = 1;  uint64\_t bit = num % mod;  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;  } |

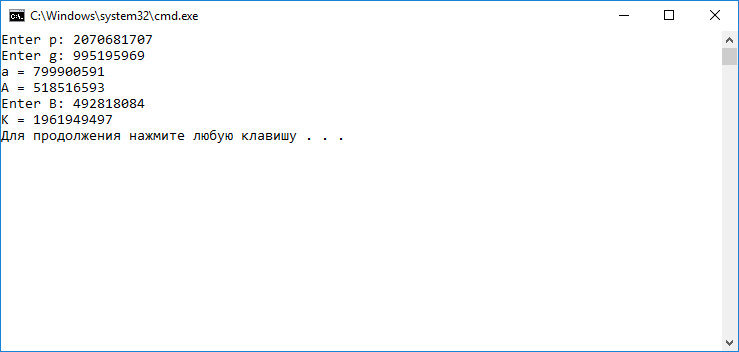
**Часть 2. Длинная арифметика (cpp\_int)**

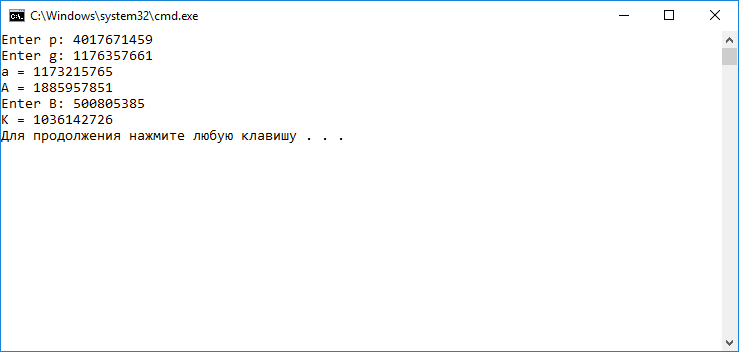
|  |
| --- |
| #include <iostream>  #include <random>  #include <boost/multiprecision/cpp\_int.hpp>  #include <boost/multiprecision/miller\_rabin.hpp>  #include <boost/random/mersenne\_twister.hpp>  #include <boost/random.hpp>  using namespace boost::multiprecision;  using std::cin;  using std::cout;  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;  } |

1. Результат работы программы

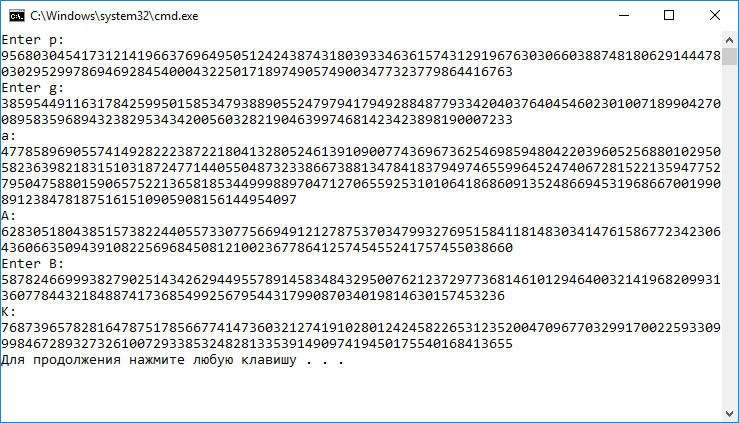
**Часть 1. Обычная 64-битная арифметика (uint64\_t)**



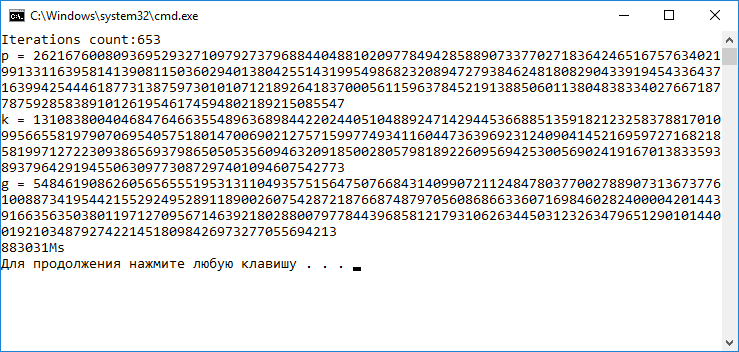




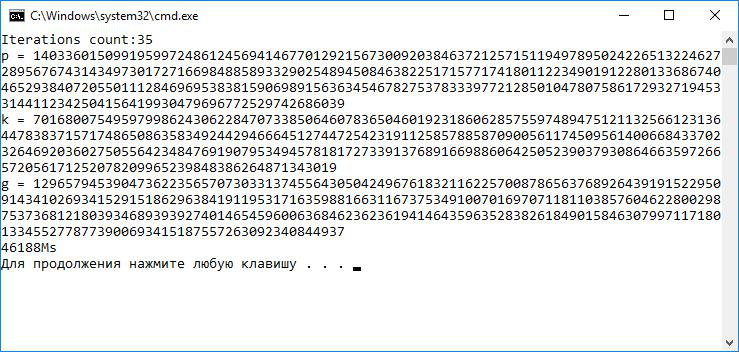
**Часть 2. Длинная арифметика (cpp\_int)**



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:



1. 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>