Московский авиационный институт
(национальный исследовательский университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

Лабораторная работа №1 по курсу «Криптография»

Студент:	М. А. Бронников
Преподаватель:	А. В. Борисов
Группа:	М8О-307Б
Дата:	
Оценка:	
Подпись:	

Москва, 2020

Вариант №4

Задача:

Разложить каждое из чисел представленных ниже на нетривиальные сомножители.

Первое число:

160769357899975610828199539114109518167531134514190990785144666932076614717841

Второе число:

125017149737222798202655599967517010894791895137836734347092348310415859721663
206658630092156681126577646542739502645815124004236606127151210775258668169992
391490206188621302254449678307072706108376399663081627986916919462316925571113
542252192544413593901487827751529987053687596294826797389954562172854772654519
238259393698557497888130594948752323314867710633065081822344395580062277418993
6635106363035784698216185461573761714766211607812695281252356674432444279

Выходные данные:

`Для каждого числа необходимо найти и вывести все его множетели - простые числа.`

1 Описание

Процесс разложения числа на его простые множетели называется факторизацией. Для решения этой задачи существует множество алгоритмов, позволяющих находить множетели, используя свойства простых чисел.

Для решения задачи я выбрал алгоритм ρ - *Полларда*[2], как один из наиболее простых и эффективных. Однако эффективен он, как я узнал позже, для чисел меньшего порядка, чем в моем задании. Моя реализация этого алгоритма работала у меня более 12 часов, после чего я решил прервать вычисления и обратиться к другим способам.

Для чисел, десятичнных знаков меньше 100 эффективен *Метод квадратичного решета*[4], реализация которого довольно трудоемка, поэтому, так как преподавательне ограничил нас в выборе средств для решения задачи, я решил прибегнуть к готовому решению, реализованному профессионалами - программному продукту **msieve**[5], который реализует в себе целый ряд алгоритмов факторизации с общим названием *Общий метод решета числового поля*[3]. Этот метод считается одним из самых эффективных современных алгоритмов факторизации. Он справился с поставленной задачей для 1-го числа менее чем за 1 минуту, что является впечатляющим результатом.

Однако 2 число имеет более 400 квадратичных знаков, факторизация которого на обычном компьютере за разумное время практически невозможна ни одним из ныне существующих алгоритмов. Однако я узнал что один из множетелей этого числа определяется к наибольший общий делитель с одним из чисел другого варанта. Поэтому я быстро написал программу, пребирающую все числа других варантов, определяющую их *НОД* с числом моего варанта и выводящий его, если он > 1. Второе же число определяется как результат деления числа моего варанта на *НОД* (по свойству делителя). Для работы с большими числами и отыскания делителя в этой программе я использовал библиотеку **gmp**[1].

2 Исходный код

Моя реализация алгоритма ρ - *Полларда* на языке C++, описание которого я взял из [2].

```
#include <iostream>
#include <string>
#include <gmpxx.h>
#include <vector>
#include <ctime>


class po_Polard{
public:
    po_Polard(const mpz_class& num);
    po_Polard(const std::string& num);
    mpz_class get_factor();
private:
    mpz_class factor_of_num(mpz_class& num);
    void Polard_GCD(mpz_class& ans, mpz_class& x, mpz_class& y);
    void Polard_MOD(mpz_class& ans, mpz_class& x, mpz_class& y);
    void Polard_ABSOLUTE(mpz_class& ans, mpz_class& x, mpz_class& y);
    mpz_class number;
};

mpz_class po_Polard::factor_of_num(mpz_class& num){
    mpz_class x, y, ans, absolute;
    unsigned long long i = 0, stage = 2;
    x = (rand() % (number - 1)) + 1;
    y = 1;
    Polard_ABSOLUTE(absolute, x, y);
    Polard_GCD(ans, num, absolute);
    while(ans == 1){
        if(i == stage){
            y = x;
            stage <<= 1;
        }
        absolute = x * x + 1;
        Polard_MOD(x, absolute, num);
        ++i;
        Polard_ABSOLUTE(absolute, x, y);
        Polard_GCD(ans, num, absolute);
    }
    return ans;
}

mpz_class po_Polard::get_factor(){
    return factor_of_num(number);
}
```

```cpp


po_Polard::po_Polard(const mpz_class& num){
    srand(time(0));
    number = num;
}

po_Polard::po_Polard(const std::string& str){
    srand(time(0));
    number = str;
}

void po_Polard::Polard_ABSOLUTE(mpz_class& ans, mpz_class& x, mpz_class& y){
    x -= y;
    mpz_abs(ans.get_mpz_t(), x.get_mpz_t());
    x += y;
}


void po_Polard::Polard_GCD(mpz_class& ans, mpz_class& x, mpz_class& y){
    mpz_gcd(ans.get_mpz_t(), x.get_mpz_t(), y.get_mpz_t());
}

void po_Polard::Polard_MOD(mpz_class& ans, mpz_class& x, mpz_class& y){
    mpz_mod(ans.get_mpz_t(), x.get_mpz_t(), y.get_mpz_t());
}

using namespace std;

int main(){
    std::string number = "1607693578999756108281995391141095181675311345141909907851446669320766147178 41";
    po_Polard polard(number);
    std::cout << "Factor: " << polard.get_factor() << endl;
    return 0;
}
```

3 Консоль

```
(base) max@max-X550CC:~/CryptoGraphia/lab1$ msieve -m -q


next number: 16076935789997561082819953911410951816753113451419099078514
4666932076614717841

160769357899975610828199539114109518167531134514190990785144666932076614
717841
p39: 311085479666681393523461051238251988263
p39: 516801227984781147501487032815011600007



next number: (base) max@max-X550CC:~/CryptoGraphia/lab1$
(base) max@max-X550CC:~/CryptoGraphia/lab1$ ls
factoring.cpp  Laboratornaya_1.odt  main.cpp  msieve.dat
(base) max@max-X550CC:~/CryptoGraphia/lab1$ g++ -std=c++17 -Wall -pedantic
main.cpp -o run -lgmpxx -lgmp
(base) max@max-X550CC:~/CryptoGraphia/lab1$ ./run
number #1: 1311861868023388708436569880100863752594599738140838016786763259
030178045833387835767275495926714633261320847000554336229411367110638183361
259555995952592066358028224811325614156674303320145810874262337999285108153
095489581002384985301528820724913562492347754784565013969160301478301896735
42345190817454789651
number #2: 9529749494555314941026898230964286976390558536530597076052746857
783935161470996178925561964178284903449922704254418495039018172365984023913
697310447084429
(base) max@max-X550CC:~/CryptoGraphia/lab1$ python3
Python 3.7.4 (default,Aug 13 2019,20:35:49)
[GCC 7.3.0] :: Anaconda,Inc. on linux
Type "help","copyright","credits" or "license" for more information.
>>>x1 = 131186186802338870843656988010086375259459973814083801678676325903
017804583338783576727549592671463326132084700055433622941136711063818336125
955599595259206635802822481132561415667430332014581087426233799928510815309
548958100238498530152882072491356249234775478456501396916030147830189673542
345190817454789651
>>>x2 = 952974949455531494102689823096428697639055853653059707605274685778
393516147099617892556196417828490344992270425441849503901817236598402391369
7310447084429
```

```
>>>x1 * x2
125017149737222798202655599967517010894791895137836734347092348310415859721
663206658630092156681126577646542739502645815124004236606127151210775258668
169992391490206188621302254449678307072706108376399663081627986916919462316
925571113542252192544413593901487827751529987053687596294826797389954562172
854772654519238259393698557497888130594948752323314867710633065081822344395
580062277418993663510636303578469821618546157376171476621160781269528125235
6674432444279
>>>x1 = 311085479666681393523461051238251988263
>>>x2 = 516801227984781147501487032815011600007
>>>x1 * x2
160769357899975610828199539114109518167531134514190990785144666932076614717
841
>>>exit()
(base) max@max-X550CC:~/CryptoGraphia/lab1$ ls
factoring.cpp  Laboratornaya_1.odt  main.cpp  msieve.dat  run
(base) max@max-X550CC:~/CryptoGraphia/lab1$ rm run msieve.dat
(base) max@max-X550CC:~/CryptoGraphia/lab1$ exit
```

4 Ответ

Разложение первого числа:

- 311085479666681393523461051238251988263
- 516801227984781147501487032815011600007

Разложение второго числа:

- 13118618680233887084365698801008637525945997381408380167867632590301780458
33387835767275495926714633261320847000554336229411367110638183361259555995
95259206635802822481132561415667430332014581087426233799928510815309548958
10023849853015288207249135624923477547845650139691603014783018967354234519
0817454789651
- 95297494945553149410268982309642869763905585365305970760527468577839
35161470996178925561964178284903449922704254418495039018172365984023916973
10447084429

5 Выводы

Благодаря превой лабораторной работе по курсу «Криптография», я напрямую познакомился с новой для себя темой - факторизацией больших чисел.

Эта работа сама по себе не является очень трудной для выполнения, однако я столкнулся со сложностью в выборе адекватного метода для поиска множетелей, ведь метод простого перебора, который превым мне пришел на ум, будет выполнять данную работу на моем компьютере дольше сотни лет, метод решета Эратофена, который я использовал ранее для схожих задач, но с меньшими числами, также требует больше ресурсов, чем я могу представить для этой задачи, а метод Полларда, который я реализовал для этой задачи, не смог справиться с этой задачей в разумные сроки. Это стало мне уроком и показало насколько важно выбрать оптимальный алгоритм для решения задачи.

Кроме того, оценив наколько сложно факторизовать число, количество знаков в котором превышает хотя бы 400, я оценил надежность такого метода шифрования, как RSA, впервые с которым я познакомился на практических занятиях этим летом.

Я рад, что мне пришлось позаниматься этой задачей в курсе «Криптография», поскольку она по-настоящему заинтересовала меня на дальнейшее развитие в области компьютерной безопасности и шифрования.

Список литературы

[1] *Библиотека GMP*
URL: https://gmplib.org/ (дата обращения: 20.02.2020).

[2] *Алгоритм ρ - Полларда*
URL: https://ru.wikipedia.org/wiki/Ро-алгоритм_Полларда (дата обращения: 21.02.2020).

[3] *Общий метод решета числового поля*
URL: https://ru.wikipedia.org/wiki/Общий_метод_решета_числового_поля (дата обращения: 21.02.2020).

[4] *Метод квадратичного решета*
URL: https://ru.wikipedia.org/wiki/Метод_квадратичного_решета (дата обращения: 21.02.2020).

[5] *Библиотека Msieve*
URL: https://github.com/radii/msieve (дата обращения: 23.02.2020).