Министерство образования и науки Российской Федерации

Федеральное государственное автономное образовательное учреждение высшего образования

«Пермский национальный исследовательский политехнический университет»

Электротехнический факультет

Кафедра «Информационные технологии и автоматизированные системы»

Направление 09.03.01 – «Информатика и вычислительная техника»

Дисциплина: «Безопасность и защита информации»

Профиль: «Автоматизированные системы интеллектуальной обработки данных и управления»

Семестр 6

ОТЧЕТ

по лабораторной работе №2

Тема: «RSA»

Выполнил: студент группы АСУ-22-1б

Мельников Г. В. \_\_\_\_\_\_\_\_\_\_\_

Проверил: ст. преподаватель кафедры ИТАС

Шереметьев В. Г. \_\_\_\_\_\_\_\_\_\_\_

Дата \_\_\_\_\_\_

Пермь, 2025

**Теоретическая часть**

**Криптосистема RSA**

Алгоритм RSA (R.Rivest, A.Shamir, L.Adleman) был предложен еще в 1977 году. С тех пор он весьма упорно противостоит различным атакам, и сейчас является самым распространенным криптоалгоритмом в мире. Он входит во многие криптографические стандарты, используется во многих приложениях и секретных протоколах (включая PEM, S-HTTP и SSL).

**Основные принципы работы RSA**

Сначала пара математических определений. Целое число называют простым, если оно делится нацело только на единицу и на само себя, иначе его называют составным. Два целых числа называют взаимно простым, если их наибольший общий делитель (НОД) равен 1.

Алгоритм работы RSA таков. Сначала надо получить открытый и секретный ключи:

1. Выбираются два простых числа p и q
2. Вычисляется их произведение n(=p\*q)
3. Выбирается произвольное число e (e<n), такое, что НОД(e,(p-1)(q-1))=1, то есть e должно быть взаимно простым с числом (p-1)(q-1).
4. Методом Евклида решается в целых числах уравнение e\*d+(p-1)(q-1)\*y=1. Здесь неизвестными являются переменные d и y – метод Евклида как раз и находит множество пар (d,y), каждая из которых является решением уравнения в целых числах.
5. Два числа (e,n) – публикуются как открытый ключ.
6. Число d хранится в строжайшем секрете – это и есть закрытый ключ, который позволит читать все послания, зашифрованные с помощью пары чисел (e,n).

Как же производится собственно шифрование с помощью этих чисел:

1. Отправитель разбивает свое сообщение на блоки, равные k=[log2(n)] бит, где квадратные скобки обозначают взятие целой части от дробного числа.
2. Подобный блок, как Вы знаете, может быть интерпретирован как число из диапазона (0;2k-1). Для каждого такого числа (назовем его mi) вычисляется выражение ci=((mi)e)mod n. Блоки ci и есть зашифрованное сообщение Их можно спокойно передавать по открытому каналу, поскольку операция возведения в степень по модулю простого числа, является необратимой математической задачей. Обратная ей задача носит название «логарифмирование в конечном поле» и является на несколько порядков более сложной задачей. То есть даже если злоумышленник знает числа e и n, то по ci прочесть исходные сообщения mi он не может никак, кроме как полным перебором mi.

А вот на приемной стороне процесс дешифрования все же возможен, и поможет нам в этом хранимое в секрете число d. Достаточно давно была доказана теорема Эйлера, частный случай которой утверждает, что если число n представимо в виде двух простых чисел p и q, то для любого x имеет место равенство (x(p-1)(q-1))mod n = 1. Для дешифрования RSA-сообщений воспользуемся этой формулой. Возведем обе ее части в степень (-y) : (x(-y)(p-1)(q-1))mod n = 1(-y) = 1. Теперь умножим обе ее части на x : (x(-y)(p-1)(q-1)+1)mod n = 1\*x = x.

А теперь вспомним как мы создавали открытый и закрытый ключи. Мы подбирали с помощью алгоритма Евклида d такое, что e\*d+(p-1)(q-1)\*y=1, то есть e\*d=(-y)(p-1)(q-1)+1. А следовательно в последнем выражении предыдущего абзаца мы можем заменить показатель степени на число (e\*d). Получаем (xe\*d)mod n = x. То есть для того чтобы прочесть сообщение ci=((mi)e)mod n достаточно возвести его в степень d по модулю m : ((ci)d)mod n = ((mi)e\*d)mod n = mi.

На самом деле операции возведения в степень больших чисел достаточно трудоемки для современных процессоров, даже если они производятся по оптимизированным по времени алгоритмам. Поэтому обычно весь текст сообщения кодируется обычным блочным шифром (намного более быстрым), но с использованием ключа сеанса, а вот сам ключ сеанса шифруется как раз асимметричным алгоритмом с помощью открытого ключа получателя и помещается в начало файла.

В 1990-х годах с помощью распределенных вычислений через Internet предпринимались удачные попытки факторизовать некоторые произвольные большие числа максимальное факторизованное число имело 140 десятичных разрядов (1999 год), на что ушло около 2000 MY (Mips/Year – годовая работа компьютера мощностью в миллион целочисленных операций в секунду). Учитывая это, сейчас специалисты (Лаборатория RSA, www. rsa.com / rsalabs ) рекомендуют использовать минимальную длину ключа n, не менее чем 768 бит (~230 десятичных рзрядов) для малосекретной информации, 1024 бит для обычной и 2048 для особо секретной. Использующаяся в старых продуктах длина ключа в 512 бит (~160 разрядов) уже под угрозой взлома. Известно, что RSA имеет низкуюкриптостойкость при шифровании коротких блоков. В таких случаях злоумышленник может взять от блока шифротекста корень степени e по модулю n, что в данном случае будет намного быстрее факторизации. Поэтому короткие блоки обязательно надо «набивать» дополнительными битами. Еще один интересный вопрос касается простых чисел. Для построения ключей алгоритму RSA необходимо найти два простых числа. Благо, среди чисел простые попадаются довольно часто: на отрезке от 1 до n примерно n/ln(n) чисел являются простыми. Поэтому можно просто брать псевдослучайные числа нужной длины и проверять их на простоту. Проверку числа на простоту можно делать двумя способами: «в лоб», перебором всех его делителей (от 2 до округленного корня из n), или с помощью более «хитрых» тестов на делимость. При переборе всех делителей мы гарантированно можем утверждать, что прошедшее такую проверку число является простым. Однако, время работы такой процедуры будет очень велико. Среди «хитрых» тестов надо выделить тест Миллера–Рабина, так как он на сегодняшний день является наиболее лучшим по всем параметрам. Так вот, проверка ллера–Рабина работает намного быстрее перебора делителей, но в отличие от него, число, выдаваемое им как простое, с некоторой очень маленькой вероятностью может оказаться составным. На практике обычно применяют последовательно несколько разных «хитрых» тестов, минимизируя тем самым вероятность ошибки. Программная реализация RSA работает медленнее примерно на два порядка по сравнению с симметричными алгоритмами. RSA часто используют вместе с каким-нибудь симметричным шифром. При таком способе все сообщения шифруются с помощью более быстрого симметричного алгоритма, а для пересылки сессионного секретного ключа этого симметричного алгоритма используется RSA. Получается вычислительно дешево и сердито.

Проясним использование алгоритма RSA на конкретном примере. Выбираем два простые числа p=7; q=17 (на практике эти числа во много раз длиннее). В этом случае n = p\*q будет равно 119. Теперь необходимо выбрать e, выбираем e=5. Следующий шаг связан с формированием числа d так, чтобы (d\*e) mod [(p-1)(q-1)]=1. d=77 (использован расширенный алгоритм Эвклида). d – секретный ключ, а e и n характеризуют открытый ключ. Пусть текст, который нам нужно зашифровать представляется M=19. С = Memod n. Получаем зашифрованный текст C=66. Этот «текст» может быть послан соответствующему адресату. Получатель дешифрует полученное сообщение, используя М= Cdmod n и C=66. В результате получается M=19.

На практике общедоступные ключи могут помещаться в специальную базу данных. При необходимости послать партнеру зашифрованное сообщение можно сделать сначала запрос его открытого ключа. Получив его, можно запустить программу шифрации, а результат ее работы послать адресату. На использовании общедоступных ключей базируется и так называемая электронная подпись, которая позволяет однозначно идентифицировать отправителя. Сходные средства могут применяться для предотвращения внесения каких-либо корректив в сообщение на пути от отправителя к получателю. Быстродействующие аппаратные 512-битовые модули могут обеспечить скорость шифрования на уровне 64 кбит в сек. Готовятся ИС, способные выполнять такие операции со скоростью 1 Мбайт/сек. Разумный выбор параметра *e* позволяет заметно ускорить реализацию алгоритма.

**Ход работы**

Система RSA использует ассиметричный ключ, что позволяет надёжно передавать сообщения. Алгоритм работает на больших числах, что создаёт сложность получения закрытого ключа. Чтобы использовать большие числа был выбран тип данных BigInteger. Для начала вычисляются вспомогательные числа p и q, дополнительно добавлена проверка, на условие не равенства p и q, иначе происходит регенерация числа q, их перемножение даёт открытый ключ n, перемножить большие числа легко, но обратная задача нахождение делителей числа гораздо сложнее, занимает больше времени. Значение f = (p – 1) \* (q – 1) служит для нахождения открытого ключа e, его значение должно быть взаимно простым с f. Ключ d является секретным, он помогает быстро расшифровать сообщение, без использования декомпозиции, его значение находится с помощью частного решения расширенного уравнения Эйлера e \* d + f \* y = 1. Для шифрования сообщение разбивается на блоки байт, размер байт блока не должен превышать n. Каждый блок конвертируется в BigInteger, полученное число возводится в степень e по модулю n, полученная закодированная последовательность добавляется в результат шифрования, с добавлением пробела (кроме первой последовательности), для однозначного разделения. Для декодирования нужно пройтись по последовательностям и провести операцию возведения в степень d по модулю n. BigInteger по умолчанию перетасовывает байты по свойству Little-Endian, но большинство кодировок, в том числе используемая ASCII работают в Big-Endian, поэтому чтобы такое не происходило нужно переворачивать массив байт при шифровании и расшифровании.

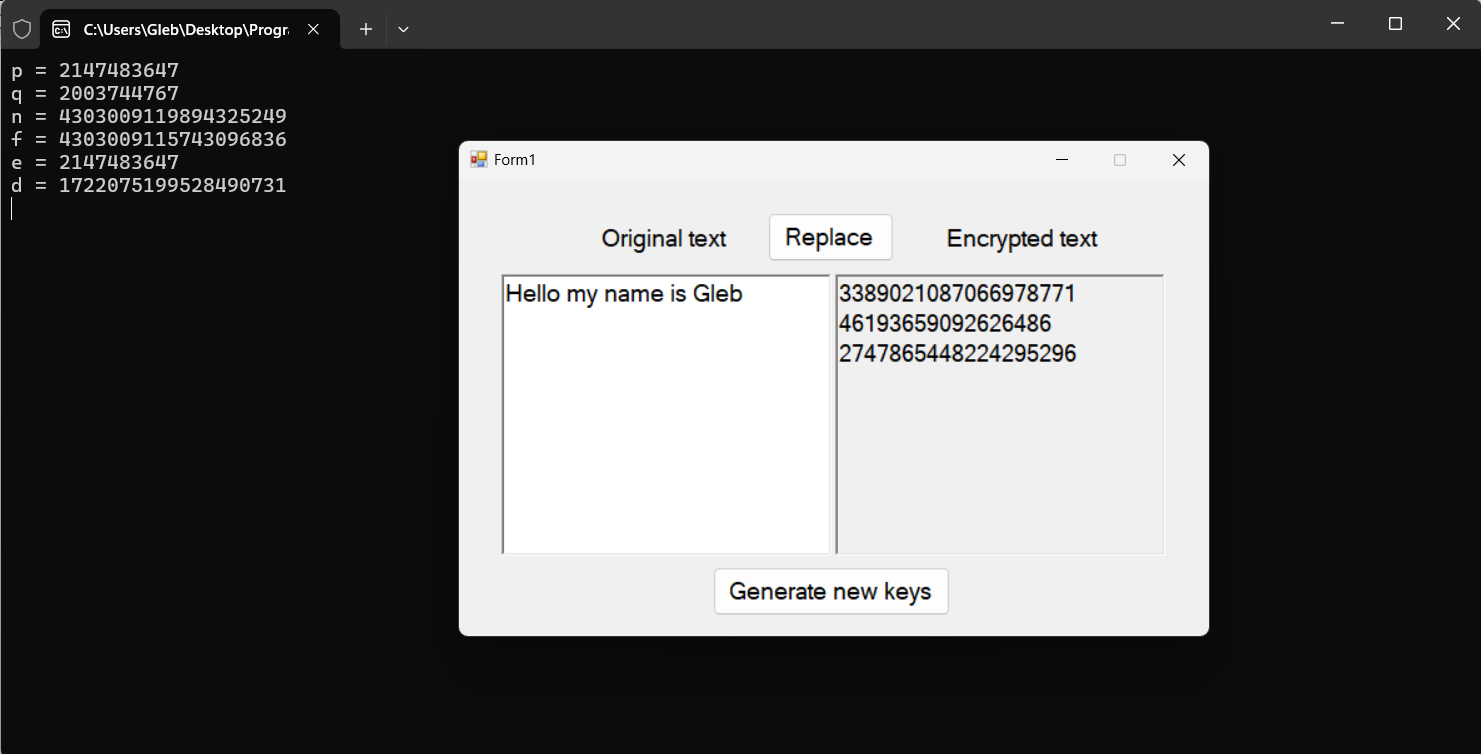
****

Рисунок 1 – пример работы 1

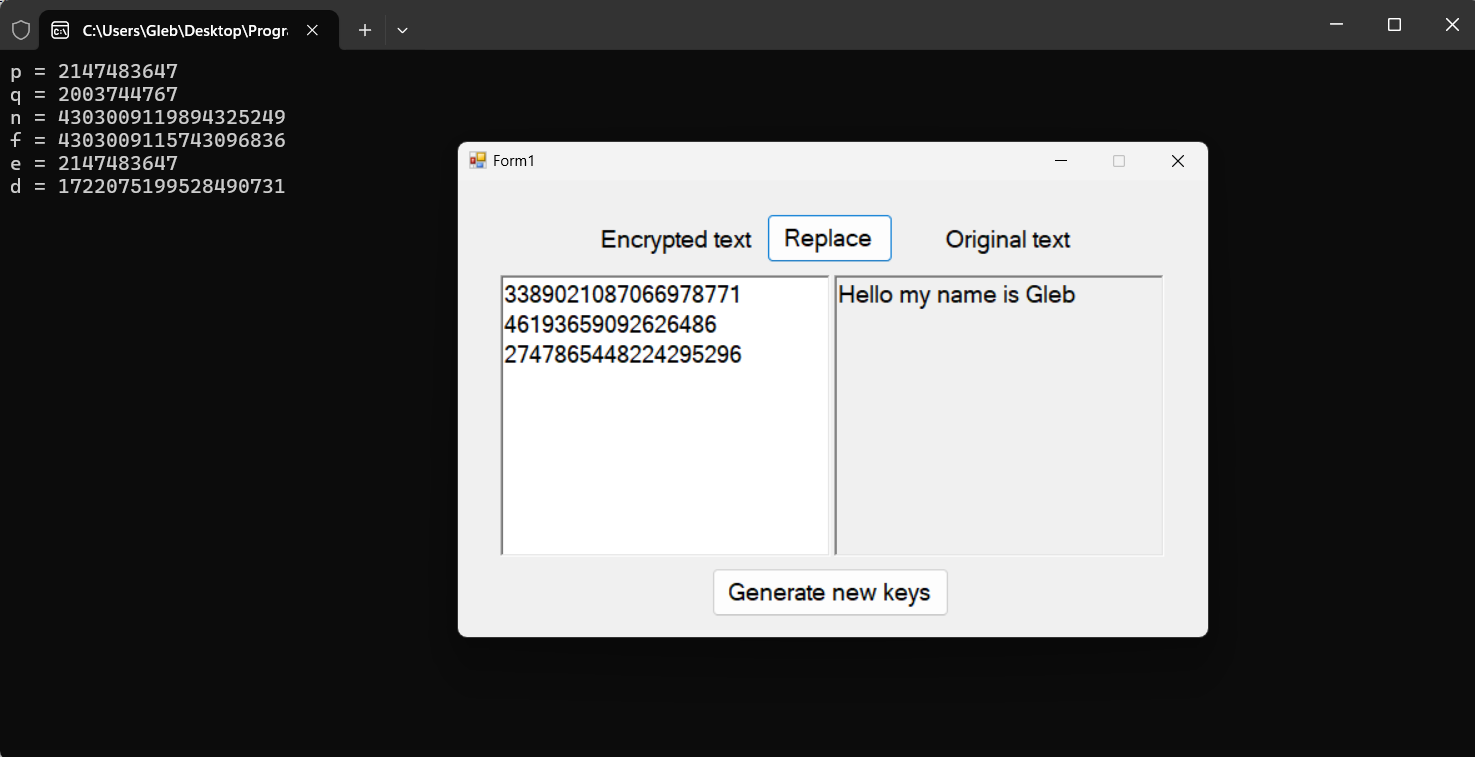


Рисунок 2 – пример использования функции “Replace”

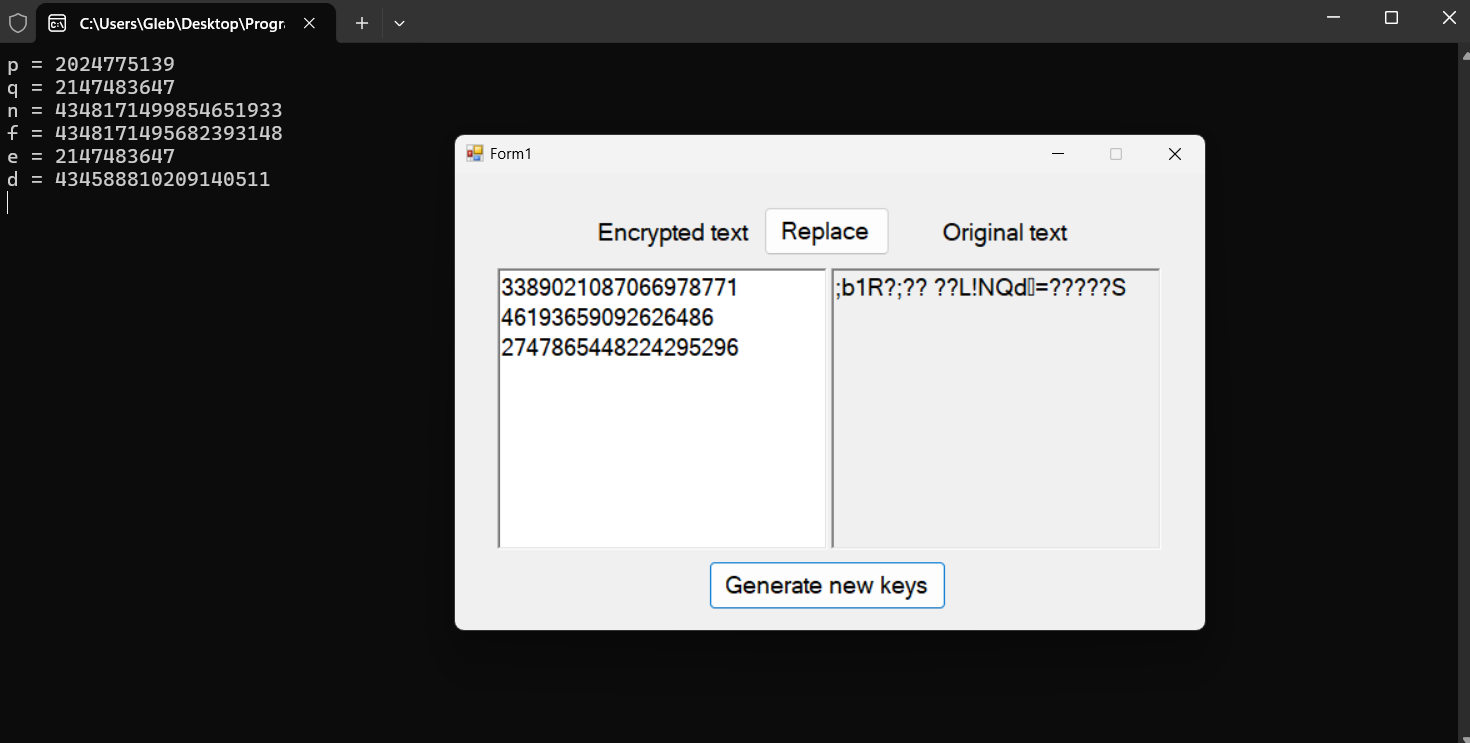


Рисунок 3 – пример использования функции “Generate new keys”

Листинг файла Form1.cs

using System;

using System.Drawing;

using System.Windows.Forms;

namespace WindowsFormsApp1

{

public partial class Form1 : Form

{

private bool isEncryptText = true;

private const int bitLenght = 16;

private RSA RSA = new RSA(bitLenght);

private void FromText\_Changed(object sender, EventArgs e)

{

ConvertText();

}

private void ConvertText()

{

ToText.ForeColor = Color.Black;

ToText.Font = new Font(ToText.Font.FontFamily, 14);

try

{

ToText.Text = isEncryptText ?

EncryptText(FromText.Text) :

DecryptText(FromText.Text);

}

catch (Exception ex)

{

ToText.Text = ex.Message;

ToText.Font = new Font(ToText.Font.FontFamily, 25);

ToText.ForeColor = Color.Red;

}

}

private void ReplaceButton\_Click(object sender, EventArgs e)

{

isEncryptText = !isEncryptText;

FromText.Text = ToText.Text;

string c = FromTextLabel.Text;

FromTextLabel.Text = ToTextLabel.Text;

ToTextLabel.Text = c;

}

private string EncryptText(string text)

{

return RSA.Encrypt(text);

}

private string DecryptText(string text)

{

return RSA.Decrypt(text);

}

public Form1()

{

InitializeComponent();

}

private void GenerateNewKeysButton\_Click(object sender, EventArgs e)

{

RSA = new RSA(bitLenght);

ConvertText();

}

}

}

Листинг файла RSA.cs

using System;

using System.Linq;

using System.Numerics;

using System.Text;

namespace WindowsFormsApp1

{

internal class RSA

{

public BigInteger e { get; private set; }

private BigInteger d { get; set; }

public BigInteger n { get; private set; }

public RSA(int bitLength)

{

BigIntegerHelper biH = new BigIntegerHelper();

BigInteger p = biH.GeneratePrime(bitLength);

BigInteger q;

do

{

q = biH.GeneratePrime(bitLength);

} while (q == p);

n = p \* q;

BigInteger f = (p - 1) \* (q - 1);

e = biH.CalculateCoprimeNumber(f, bitLength);

d = biH.CalculateModInverse(e, f);

#if DEBUG

Console.Clear();

Console.WriteLine($"p = {p}");

Console.WriteLine($"q = {q}");

Console.WriteLine($"n = {n}");

Console.WriteLine($"f = {f}");

Console.WriteLine($"e = {e}");

Console.WriteLine($"d = {d}");

#endif

}

public string Encrypt(string text)

{

if (text.Length == 0)

{

return string.Empty;

}

StringBuilder sb = new StringBuilder();

byte[] textData = Encoding.ASCII.GetBytes(text);

int maxBlockSize = n.ToByteArray().Length - 1;

int index = 0;

while (index < textData.Length)

{

int blockSize = Math.Min(maxBlockSize, textData.Length - index);

var block = textData.Skip(index).Take(blockSize).ToArray();

Array.Reverse(block);

BigInteger biFromBlock = new BigInteger(block);

sb.Append($" {BigInteger.ModPow(biFromBlock, e, n)}");

index += blockSize;

}

return sb.ToString(1, sb.Length - 1);

}

public string Decrypt(string text)

{

if (text.Length == 0)

{

return string.Empty;

}

try

{

StringBuilder sb = new StringBuilder();

foreach (var item in text.Split().Select(t => BigInteger.Parse(t)))

{

var bytes = BigInteger.ModPow(item, d, n).ToByteArray();

Array.Reverse(bytes);

sb.Append(Encoding.ASCII.GetString(bytes));

}

return sb.ToString();

}

catch

{

throw new Exception("Error. The text is not decipherable");

}

}

}

}

Листинг файла BigIntegerHelper.cs

using System;

using System.Collections.Generic;

using System.Numerics;

using System.Security.Authentication.ExtendedProtection;

namespace WindowsFormsApp1

{

internal class BigIntegerHelper

{

private Random rnd = new Random();

private BigInteger ExtendedGreatestCommonDivisor(

BigInteger x, BigInteger y, ref BigInteger a, ref BigInteger b

)

{

if (y == 0 || x == 0)

{

a = y == 0 ? 1 : 0;

b = 1 - a;

return y + x;

}

BigInteger gcd = 0;

if (y < x)

{

gcd = ExtendedGreatestCommonDivisor(x % y, y, ref a, ref b);

b -= x / y \* a;

}

else

{

gcd = ExtendedGreatestCommonDivisor(x, y % x, ref a, ref b);

a -= y / x \* b;

}

return gcd;

}

public BigInteger Sqrt(BigInteger number)

{

if (number < 0)

{

throw new ArgumentException("Number cannot be negative");

}

if (number == 0 || number == 1)

{

return number;

}

BigInteger x0 = number / 2;

BigInteger x1 = (x0 + number / x0) / 2;

while (x1 < x0)

{

x0 = x1;

x1 = (x0 + number / x0) / 2;

}

return x0;

}

public bool IsPrime(BigInteger number)

{

if (number < 2)

{

return false;

}

BigInteger sqrtNumber = Sqrt(number);

for (BigInteger i = 2; i <= sqrtNumber; i++)

{

if (number % i == 0)

{

return false;

}

}

return true;

}

public BigInteger GeneratePrime(int bitLength)

{

BigInteger number = BigInteger.One << (bitLength - 1);

do

{

number |= BigInteger.Abs(new BigInteger(rnd.Next())) | 1;

} while (!IsPrime(number));

return number;

}

public BigInteger CalculateModInverse(BigInteger number, BigInteger mod)

{

BigInteger a = 0, b = 0;

BigInteger gcd = ExtendedGreatestCommonDivisor(number, mod, ref a, ref b);

if (gcd != 1)

{

throw new ArgumentException("Error! Numbers are not coprime!");

}

return a < 0 ? a + mod : a;

}

public BigInteger CalculateCoprimeNumber(BigInteger f, int bitLenght)

{

BigInteger otherNumber = GeneratePrime(bitLenght);

while

(

!IsPrime(otherNumber) ||

BigInteger.GreatestCommonDivisor(otherNumber, f) != 1

)

{

otherNumber += 2;

}

return otherNumber;

}

}

}