Федеральное государственное автономное

образовательное учреждение

высшего профессионального образования

«СИБИРСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

институт

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

кафедра

**ОТЧЕТ ПО ЛАБОРАТОРНОЙ РАБОТЕ**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

тема

\_\_\_\_\_\_\_\_\_\_\_\_Алгоритм шифрования RSA\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Преподаватель \_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_

подпись, дата инициалы, фамилия

Студент \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_

номер группы, зачетной книжки подпись, дата инициалы, фамилия

Оглавление

[Задание 3](#_Toc417026947)

[Описание алгоритма 4](#_Toc417026948)

[Получение открытого и закрытого ключей 4](#_Toc417026949)

[Шифрование 4](#_Toc417026950)

[Расшифровка 4](#_Toc417026951)

[Блок-схема алгоритма 5](#_Toc417026952)

[Листинг программы, реализующей алгоритмы 6](#_Toc417026953)

[Класс PartitionManager 6](#_Toc417026954)

[Класс RsaCryptography 7](#_Toc417026955)

[Контрольные примеры 10](#_Toc417026956)

[Пример №1 10](#_Toc417026957)

[Пример №2 11](#_Toc417026958)

[Пример №3 12](#_Toc417026959)

[Пример №4 13](#_Toc417026960)

[Пример №5 14](#_Toc417026961)

[Тестирование производительности 15](#_Toc417026962)

[Заключение 20](#_Toc417026963)

# Задание

1. Составить в виде блок-схемы алгоритм шифрования/дешифрования RSA, со следующими особенностями:

* объём исходного текста – любой (в разумных пределах);
* исходный текст может состоять из русских и английских букв, цифр, а также знаков препинания;
* исходный текст находится в кодировке ASCII;
* N состоит из 31 десятичного знака.
* Числа P и Q выбираются случайным образом, так, что , где P и Q – простые числа.
* исходный текст разбивается на K блоков, где K выбирается исходя из значения модуля N

1. Убедиться в правильности составления алгоритмов, а затем на языке C# составить программу, которая реализует данный алгоритм.
2. На ряде контрольных примеров (не менее 10) открытого текста проверить правильность работы алгоритмов шифрования и дешифрования (в качестве контрольного примера понимается текстовый файл в кодировке ASCII).
3. Оценить криптостойкость моего варианта алгоритма RSA, а также сделать оценку производительности, разработанной программы.

# Описание алгоритма

Получение открытого и закрытого ключей

* 1. Мы задаем два различных простых числа .
  2. Положим
  3. Вычисляем
  4. Случайным образом выбираем число и взаимно простое с
  5. Вычисляем ***e***, такое что

Шифрование

* 1. Текст ***M*** разбивается на на блоки, так что размер каждого блока **.**
  2. Зашифровываем каждый блок:
  3. Зашифрованный передается.

Расшифровка

* 1. Расшифровываем каждый полученный блок зашифрованного текста

**:**

* 1. Блоки объединяются в сообщение.
  2. - исходный текст

# Блок-схема алгоритма



# Листинг программы, реализующей алгоритмы

## Класс PartitionManager

using System;

using System.Collections.Generic;

using System.Text;

namespace RSA\_Project

{

//Класс работающий с кодировками и разбиением текста

public static class PartitionManager

{

public static String PartitionToString(List<byte[]> p)

{

//Переобразует разбиение в строку

String result = "";

foreach (var b in p)

{

result += GetString(b);

}

return result;

}

public static List<byte[]> StringToPartition(String s, long length)

{

/\*Разбиваем строку на блоки

\* s - входная строка, length - длина блока в байтах

\*/

var result = new List<byte[]>();

byte[] bytes = GetBytes(s);

var block = new List<byte>();

int count = 0;

foreach (byte x in bytes)

{

count++;

if (count < length)

{

block.Add(x);

}

else

{

result.Add(block.ToArray());

block.Clear();

block.Add(x);

count = 1;

}

}

if (block.Count > 0) result.Add(block.ToArray());

return result;

}

public static byte[] GetBytes(string str)

{

/\*

\* Препобразует строку кодировки windows-1251 в байт-массив

\*

\*/

return Encoding.Convert(Encoding.Unicode,

Encoding.GetEncoding("windows-1251"),

Encoding.Unicode.GetBytes(str));

}

public static string GetString(byte[] bytes)

{

/\*

\* Преобразует байт-массив в строку в кодировке windows-1251

\* \*/

byte[] unicodeBytes = Encoding.Convert(

Encoding.GetEncoding("windows-1251"), Encoding.Unicode, bytes);

return Encoding.Unicode.GetString(unicodeBytes);

}

}

}

## Класс RsaCryptography

using System;

using System.Collections.Generic;

using System.Numerics;

using System.Security.Cryptography;

namespace RSA\_Project

{

public class RsaCryptography

{

private static readonly Random Rnd = new Random((int) DateTime.Now.Ticks);

private static readonly RandomNumberGenerator Rng = RandomNumberGenerator.Create();

private readonly int[] \_fermatNumbers = {17, 257, 65537}; //Числа ферма

private BigInteger \_d; //открытый и закрытый ключи

private BigInteger \_e; //открытый и закрытый ключи

private BigInteger \_n; //P,Q -простые числа. N - их произведение

private BigInteger \_p; //P,Q -простые числа. N - их произведение

private BigInteger \_q; //P,Q -простые числа. N - их произведение

public RsaCryptography(BigInteger p, BigInteger q)

{

//Инициализация класса

\_p = p;

\_q = q;

\_n = \_p\*\_q;

BigInteger eulierFunction = (\_p - 1)\*(\_q - 1);

\_e = \_fermatNumbers[Rnd.Next(0, \_fermatNumbers.Length)];

\_d = BuildPrivateKey(eulierFunction);

}

public RsaCryptography(int log10N)

{

GenerateKeys(log10N);

}

public long ModuleLength

{

//Длина модуля в байтах

get

{

long l = \_n.ToByteArray().Length;

return l;

}

}

public List<byte[]> EncryptMessage(List<byte[]> p)

{

//Зашифровать строку

var result = new List<byte[]>();

foreach (var block in p)

{

result.Add(Encrypt(block));

}

return result;

}

public List<byte[]> DecryptMessage(List<byte[]> p)

{

//Разшифровать строку

var result = new List<byte[]>();

foreach (var block in p)

{

result.Add(Decrypt(block));

}

return result;

}

public byte[] Encrypt(byte[] message)

{

//Шифруем сообщение открытым ключом

var val = new BigInteger(message);

val = BigInteger.ModPow(val, \_e, \_n);

return val.ToByteArray();

}

public byte[] Decrypt(byte[] message)

{

//Расшифруем cообщение открытм ключом

var val = new BigInteger(message);

val = BigInteger.ModPow(val, \_d, \_n);

return val.ToByteArray();

}

private BigInteger BuildPrivateKey(BigInteger E)

{

//Вычисляем закрытый ключ, находя обратный по модулю элемент кольца

BigInteger x, y;

BigInteger g = GCD(\_e, E, out x, out y);

return (x%E + E)%E;

}

private static BigInteger GCD(BigInteger a, BigInteger b, out BigInteger x, out BigInteger y)

{

//Расширеный Алгоритм Евклида

if (a.IsZero) //Находит НОД чисел A и B, и коэфициенты x,y уравнения Ax +By = НОД(A,B)

{

x = 0;

y = 1;

return b;

}

BigInteger x1, y1;

BigInteger d = GCD(b%a, a, out x1, out y1);

x = y1 - (b/a)\*x1;

y = x1;

return d;

}

#region Свойства

public BigInteger Module

{

//Возвращает значения модуля(N)

get { return \_n; }

}

public BigInteger D

{

//значение закрытого ключа

get { return \_d; }

}

public BigInteger E

{

//значение откртыго ключа

get { return \_e; }

}

#endregion

#region Методы для генерации случайного ключа

private static BigInteger Random(int bytes)

{

var buffer = new byte[bytes];

Rng.GetBytes(buffer);

buffer[bytes - 1] = (byte) ((buffer[bytes - 1] & 127) | 64); // старший байт

return new BigInteger(buffer);

}

private static bool IsPrimary(BigInteger x)

{

if (x < 2) return false; // отбрасываем отрицательные и единицу

int len = x.ToByteArray().Length;

int tests = 2\*len + 100; // Чем больше тестов тем меньше вероятность ошибиться

BigInteger y = x - 1;

for (int i = 0; i < tests; i++)

{

BigInteger a = (Random((int) (1+len\*Rnd.NextDouble()))%y) + 1; // берём ненулевое

// проверяем выполнение малой теоремы Ферма

// если простое то

if (!(BigInteger.ModPow(a, y, x) - 1).IsZero) return false;

}

// признаём число простым

// хотя можем продолжать ошибаться

return true;

}

private static BigInteger GeneratePrimary(int bytes)

{

BigInteger x = Random(bytes) | 1; // Простые являются нечётными

while (!IsPrimary(x)) x += 2; // Движемся вперёд пока не встретим простое

return x;

}

public void GenerateKeys(int log10N)

{

// 10^x==2^y

// x=y\*log10(2)

var bits = (int) Math.Ceiling(log10N/Math.Log10(2));

int bits1 = bits/2;

int bits2 = bits - bits1;

int bytes1 = (bits1 + 7)/8;

int bytes2 = (bits2 + 7)/8;

\_p = GeneratePrimary(bytes1);

\_q = GeneratePrimary(bytes2);

\_n = \_p\*\_q;

BigInteger eulierFunction = (\_p - 1)\*(\_q - 1);

\_e = \_fermatNumbers[Rnd.Next(0, \_fermatNumbers.Length)];

\_d = BuildPrivateKey(eulierFunction);

}

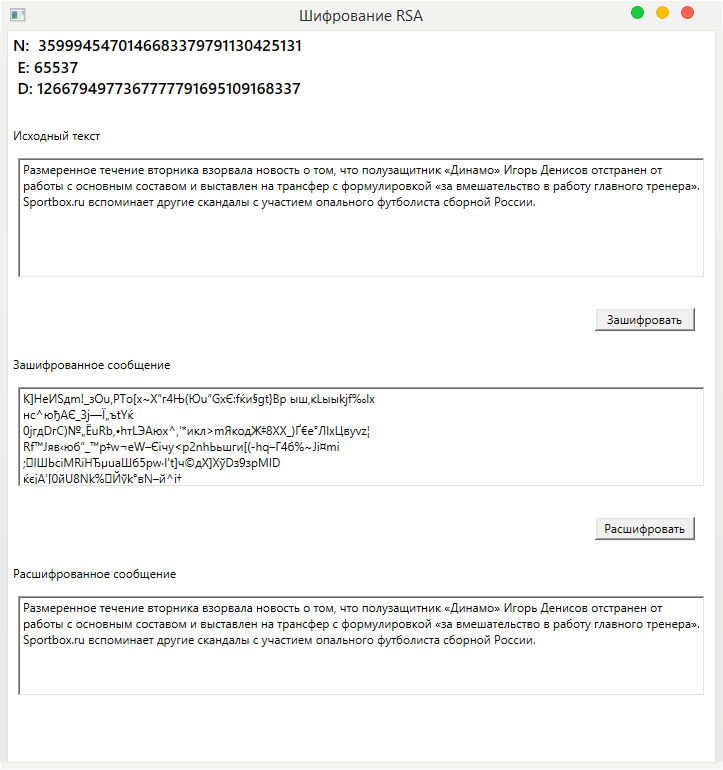
#endregion

}

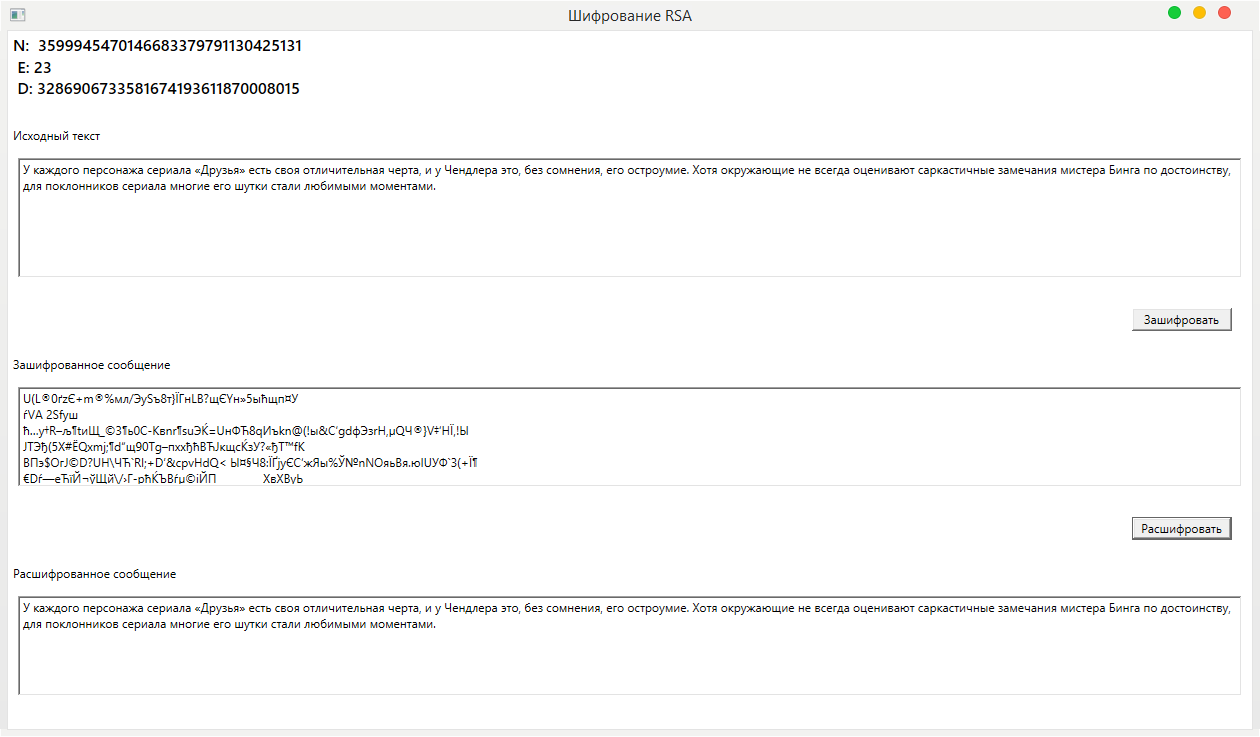
}

# Контрольные примеры

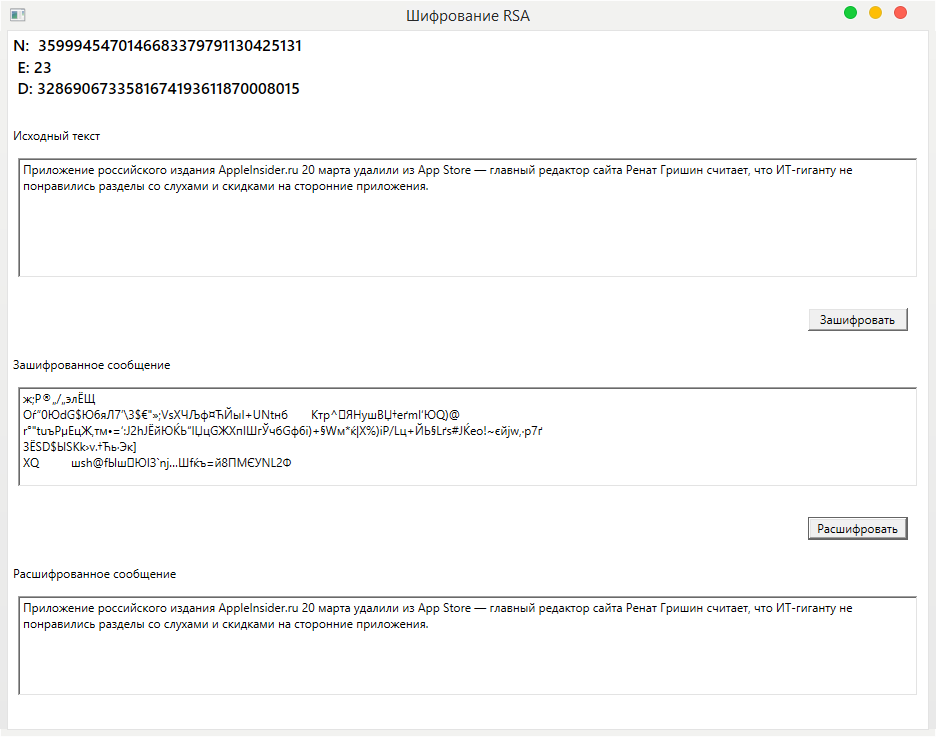
Пример №1



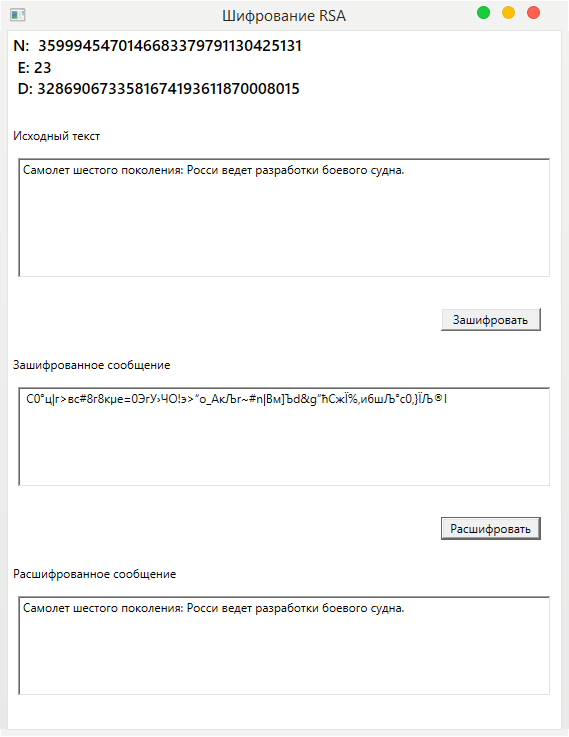
Пример №2



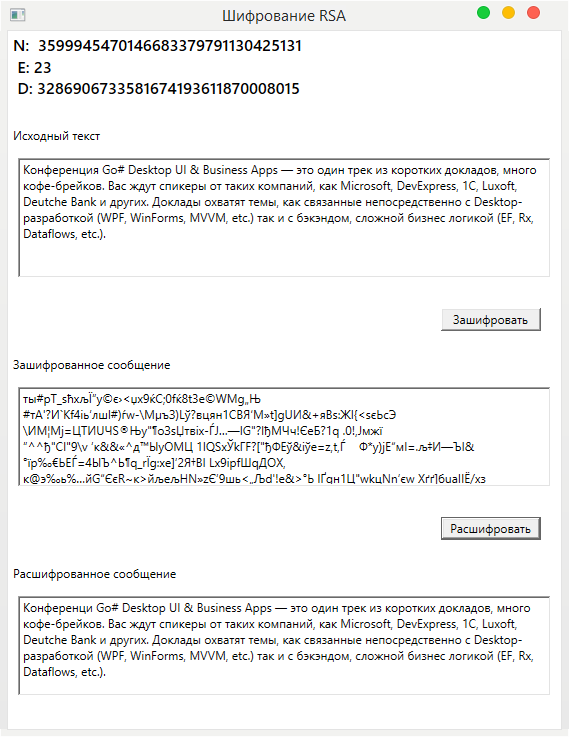
Пример №3



Пример №4



Пример №5



# Тестирование производительности

Windows 8.1 Профессиональная

Процессор: Intel Core Duo E8600 @ 3.33GHz 2.67GHz

Установленная память: 8,00 ГБ

Тип системы: 64-разрядная операционная система, процессор x64

Время измеряется в миллисекундах (1s == 1000ms)

N – количество десятичных разрядов

TestMethod1 – измерение среднего времени генерации ключа

TestMethod2 – измерение среднего времени генерации одного простого числа

TestMethod3 – измерение среднего времени одной итерации проверки числа на простоту

using System;

using System.Numerics;

using Microsoft.VisualStudio.TestTools.UnitTesting;

using RSA\_Project;

namespace RSARsaCryptographyProject.UnitTest

{

[TestClass]

public class UnitTest1

{

[TestMethod]

public void TestMethod1()

{

const int count = 20;

Console.WriteLine(@"# N GenerateKeys");

for (int decimals = 20; decimals < 50; decimals++)

{

var rsa = new RsaCryptography(decimals);

DateTime t = DateTime.Now;

for (int i = 0; i < count; i++)

rsa.GenerateKeys(decimals);

var ts = new TimeSpan(DateTime.Now.Ticks - t.Ticks);

Console.WriteLine(@"{0} {1}", decimals, ts.TotalMilliseconds/count);

}

}

[TestMethod]

public void TestMethod2()

{

const int count = 20;

Console.WriteLine(@"# N GeneratePrimary");

for (int decimals = 20; decimals < 50; decimals++)

{

var bits = (int) Math.Ceiling(decimals/Math.Log10(2));

int bytes = (bits + 7)/8;

DateTime t = DateTime.Now;

for (int i = 0; i < count; i++)

RsaCryptography.GeneratePrimary(bytes);

var ts = new TimeSpan(DateTime.Now.Ticks - t.Ticks);

Console.WriteLine(@"{0} {1}", decimals, ts.TotalMilliseconds/count);

}

}

[TestMethod]

public void TestMethod3()

{

const int count = 20;

Console.WriteLine(@"# N IsPrimary");

for (int decimals = 20; decimals < 50; decimals++)

{

var bits = (int) Math.Ceiling(decimals/Math.Log10(2));

int bytes = (bits + 7)/8;

DateTime t = DateTime.Now;

int total = 0;

for (int i = 0; i < count; i++)

{

BigInteger x = RsaCryptography.Random(bytes) | 1; // Простые являются нечётными

total += RsaCryptography.NumberOfTests(x);

RsaCryptography.IsPrimary(x);

}

var ts = new TimeSpan(DateTime.Now.Ticks - t.Ticks);

Console.WriteLine(@"{0} {1}", decimals, ts.TotalMilliseconds / count);

}

}

}

}

**# N GenerateKeys**

**20 6.75034**

**21 8.00045**

**22 6.40038**

**23 14.85087**

**24 11.650665**

**25 8.00043**

**26 10.25056**

**27 7.85043**

**28 19.301155**

**29 10.200605**

**30 8.400485**

**31 9.400505**

**32 9.700535**

**33 10.700615**

**34 10.20062**

**35 14.400835**

**36 11.500655**

**37 9.95056**

**38 11.000615**

**39 19.901185**

**40 26.55151**

**41 23.251325**

**42 16.50097**

**43 21.95126**

**44 21.25121**

**45 30.201735**

**46 28.501635**

**47 22.45127**

**48 22.501335**

**49 26.75156**

**# N GeneratePrimary**

**20 8.050425**

**21 8.200505**

**22 9.500525**

**23 10.200605**

**24 9.150535**

**25 10.800625**

**26 10.550605**

**27 12.70073**

**28 11.550645**

**29 17.151005**

**30 18.80107**

**31 18.051045**

**32 20.651225**

**33 23.951345**

**34 21.151235**

**35 22.951325**

**36 24.301425**

**37 25.55147**

**38 24.4014**

**39 34.05202**

**40 33.751935**

**41 40.35234**

**42 41.2524**

**43 38.5022**

**44 41.85243**

**45 44.852585**

**46 52.153025**

**47 45.1526**

**48 50.352905**

**49 65.753805**

**# N IsPrimary**

**20 0.39999**

**21 0.050005**

**22 0.049985**

**23 0.10002**

**24 0.40002**

**25 0.55006**

**26 0.09999**

**27 0.050005**

**28 0.099985**

**29 0.100005**

**30 0.1001**

**31 0.149915**

**32 0.900075**

**33 0.95005**

**34 1.050035**

**35 1.850125**

**36 1.05007**

**37 1.050035**

**38 1.100085**

**39 0.200015**

**40 0.19999**

**41 0.250045**

**42 0.199995**

**43 1.6501**

**44 1.850075**

**45 0.300045**

**46 0.25**

**47 1.900125**

**48 0.350035**

**49 2.600115**

# Заключение

1. Полученные навыки:

* Изучены вспомогательные алгоритмы, такие как:
  + Обобщенный алгоритм Евклида
  + Вычисление функции эйлера
  + Нахождение обратного по модулю элемента
* Длинная арфиметика в .NET(BigInteger)
* Для создания графического интерфейса использовалась технология WPF
* Базовые знания теории чисел такие как:
  + Кольцо целых чисел
  + Функция Эйлера
  + Теорема Ферма
  + НОД
  + Сравнение по модулю

1. Несмотря на свою простоту и относительно солидный возраст алгоритм шифрования RSA не потерял своей актуальности. Задача факторизации больших чисел до сих пор остается тяжело решаемой, хотя и существуют некоторые способы взлома данной криптосистемы - все они очень сложны и трудно исполнимы. Естественно, что его криптостойкость напрямую зависит от величины простых чисел *P* и *Q, которые и требуется найти в процессе факторизации.*