Федеральное государственное бюджетное образовательное учреждение

высшего образования «Петрозаводский государственный университет»

Институт математики и информационных технологий

Кафедра информатики и математического обеспечения

Отчёт по лабораторной работе №1

по дисциплине: «Криптографические средства»

Вариант 1.2

Выполнил студент группы 22307:

Гордеев Никита Владиславович

Проверил преподаватель:

Кафедры прикладной математики и кибернетики

Института математики и информационных технологий

Ларионов Дмитрий Дмитриевич

Петрозаводск

2024

Оглавление

[1 Формулировка задания 2](#_Toc165397491)

[2 Описание метода решения 3](#_Toc165397492)

[3 Примеры кода программ: 3](#_Toc165397493)

[3.1 Реализован класс Random для генерации случайных чисел: 3](#_Toc165397494)

[3.1.1 Метод getrandbits генерирует k случайных битов. 3](#_Toc165397495)

[3.1.2 Метод randrange генерирует случайное число из заданного диапазона. 3](#_Toc165397496)

[3.1.3 Метод randint генерирует случайное целое число из заданного диапазона. 4](#_Toc165397497)

[3.1.4 Метод generate\_large\_number генерирует большое случайное число заданной длины. 4](#_Toc165397498)

[3.1.5 Метод generate\_prime генерирует случайное простое число заданной длины. 4](#_Toc165397499)

[3.2 Реализован класс Math: 4](#_Toc165397500)

[3.2.1 Метод pow возводит число в степень по модулю. 4](#_Toc165397501)

[3.2.2 Метод gcd находит наибольший общий делитель двух чисел. 5](#_Toc165397502)

[3.2.3 Метод is\_prime проверяет, является ли число простым. 5](#_Toc165397503)

[3.2.4 Метод mod\_inverse находит мультипликативно обратное число по модулю. 6](#_Toc165397504)

[3.3 Реализован класс RSA для работы с алгоритмом шифрования RSA: 6](#_Toc165397505)

[3.3.1 Метод generate\_keys генерирует открытый и закрытый ключи. 6](#_Toc165397506)

[3.3.2 Метод encrypt выполняет шифрование данных. 7](#_Toc165397507)

[3.3.3 Метод decrypt выполняет расшифрование данных. 7](#_Toc165397508)

[3.4 Возможность работы пользователю 7](#_Toc165397509)

[4 Тестовые данные 8](#_Toc165397510)

# Формулировка задания

Напишите программу шифрования и расшифрования алгоритмом RSA. Рекомендуется использовать библиотеку для работы с длинными числами. В случае применения этой библиотеки разрешается использовать функции сложения, вычитания, умножения, целочисленного деления, вычисления остатка от деления. Функции возведения числа в степень, нахождения наибольшего общего делителя, обратного элемента в мультипликативной группе вычетов, генерации простого числа реализовать самостоятельно. Выполняемые функции программы:

1) генерация пары открытый/закрытый ключ, при этом число e задается пользователем;

2) шифрование данных (целого числа);

3) расшифрование шифртекста (целого числа).

# Описание метода решения

Для шифрования и расшифровки информации используются различные ключи. Пользователь, желающий отправить зашифрованное сообщение, использует открытый ключ получателя для шифрования сообщения, а получатель использует свой закрытый ключ для расшифровки сообщения.

1. Первый пользователь вводит число e (показатель степени) вводится в консоли. // 3
2. Выбираются два случайных простых числа p и q заданного размера // 53, 59
3. Вычисляется модуль, n = p \* q // 3127
4. Вычисляется значение функции Эйлера phi(n) = (p-1) \* (q-1) // 3016
5. Если phi\_n не взаимно простое с e то выбирается новое phi\_n
6. Вычисляется секретная экспонента d, мультипликативно обратная числу e → d\*e(mod phi\_n) = 1 // 2011
7. Отрытые ключи n, e передаются второму пользователю
8. Второй пользователь вводит в консоли сообщение data // 3127
9. Сообщение шифруется формулой b = data^e (mod n) // 89^3 mod 3127 = 1394
10. Второй пользователь отправляет зашифрованное сообщение b первому пользователю
11. Первый пользователь расшифровывает data = b^d(mod n) // 89 =1394^2011 mod 3127

# Примеры кода программ:

## Реализован класс Random для генерации случайных чисел:

Реализованы методы getrandbits, randrange, randint, generate\_large\_number и generate\_prime.

### Метод getrandbits генерирует k случайных битов.

def getrandbits(self, k):  
 *"""  
 Генерация k случайных битов.  
 :param k: Количество битов.  
 :return: Случайное число с k битами.  
 """* if self.seed is None:  
 raise ValueError("Seed is not set")  
 self.index += 1  
 return (self.seed + self.index) % (2 \*\* k)

### Метод randrange генерирует случайное число из заданного диапазона.

def randrange(self, start, stop=None, step=1):  
 *"""  
 Генерация случайного числа из диапазона.  
 :param start: Начало диапазона.  
 :param stop: Конец диапазона.  
 :param step: Шаг.  
 :return: Случайное число из диапазона.  
 """* if stop is None:  
 start, stop = 0, start  
 if step == 1:  
 return start + self.getrandbits(self.\_bit\_length(stop - start))  
 else:  
 return start + step \* self.getrandbits(self.\_bit\_length((stop - start) // step))

### Метод randint генерирует случайное целое число из заданного диапазона.

def randint(self, a, b):  
 *"""  
 Генерация случайного целого числа из диапазона [a, b].  
 :param a: Начало диапазона.  
 :param b: Конец диапазона.  
 :return: Случайное целое число.  
 """* return self.randrange(a, b + 1)

### Метод generate\_large\_number генерирует большое случайное число заданной длины.

def generate\_large\_number(self, length):  
 *"""  
 Генерация большого случайного числа длины length.  
 :param length: Длина числа.  
 :return: Случайное число.  
 """* return self.randint(2\*\*(length-1), 2\*\*length)

### Метод generate\_prime генерирует случайное простое число заданной длины.

def generate\_prime(self, length):  
 *"""  
 Генерация случайного простого числа длины length.  
 :param length: Длина числа.  
 :return: Простое число.  
 """* while True:  
 p = self.generate\_large\_number(length)  
 if Math().is\_prime(p):  
 return p

## Реализован класс Math:

Реализованы методы **pow**, **gcd**, **is\_prime** и **mod\_inverse**.

### Метод pow возводит число в степень по модулю.

def pow(self, x, y, z=None):  
 *"""  
 Возведение числа x в степень y по модулю z.  
 :param x: Основание.  
 :param y: Показатель степени.  
 :param z: Модуль.  
 :return: Результат возведения в степень по модулю.  
 """* if z is None:  
 return x \*\* y  
 result = 1  
 while y:  
 if y & 1:  
 result = result \* x % z  
 x = x \* x % z  
 y >>= 1  
 return result

### Метод gcd находит наибольший общий делитель двух чисел.

def gcd(self, a, b):  
 *"""  
 Нахождение наибольшего общего делителя чисел a и b.  
 :param a: Первое число.  
 :param b: Второе число.  
 :return: НОД(a, b).  
 """* while b != 0:  
 a, b = b, a % b  
 return a

### Метод is\_prime проверяет, является ли число простым.

def is\_prime(self, n, k=5):  
 *"""  
 Проверка, является ли число простым.  
 :param n: Число для проверки.  
 :param k: Количество итераций теста Миллера-Рабина.  
 :return: True, если число простое, иначе False.  
 """* if n <= 1:  
 return False  
 if n <= 3:  
 return True  
 def miller\_rabin(n, d):  
 a = self.random.randint(2, n - 2)  
 while self.gcd(a, n) != 1:  
 a = self.random.randint(2, n - 2)  
 x = self.pow(a, d, n)  
 if x == 1 or x == n - 1:  
 return True  
 while d != n - 1:  
 x = self.pow(x, 2, n)  
 d \*= 2  
 if x == 1:  
 return False  
 if x == n - 1:  
 return True  
 return False  
  
 d = n - 1  
 while d % 2 == 0:  
 d //= 2  
  
 for \_ in range(k):  
 if not miller\_rabin(n, d):  
 return False  
 return True

### Метод mod\_inverse находит мультипликативно обратное число по модулю.

def mod\_inverse(self, a, m):  
 *"""  
 Нахождение мультипликативно обратного числа a по модулю m.  
 :param a: Число.  
 :param m: Модуль.  
 :return: Мультипликативно обратное к a по модулю m.  
 """* m0, x0, x1 = m, 0, 1  
 while a > 1:  
 q = a // m  
 m, a = a % m, m  
 x0, x1 = x1 - q \* x0, x0  
 return x1 + m0 if x1 < 0 else x1

## Реализован класс RSA для работы с алгоритмом шифрования RSA:

Реализованы методы generate\_keys, encrypt и decrypt.

### Метод generate\_keys генерирует открытый и закрытый ключи.

def generate\_keys(self, e):  
 *"""  
 Генерация открытого и закрытого ключей.  
 :param e: Показатель степени e.  
 :return: Открытый и закрытый ключи.  
 """  
  
 # Выбираются два простых числа длины key\_length* p = self.random.generate\_prime(self.key\_length)  
 q = self.random.generate\_prime(self.key\_length)  
  
 *# Вычисляется модуль, n = p \* q* n = p \* q  
  
 *# Вычисляется значение функции Эйлера phi\_n = (p - 1) \* (q - 1)* phi\_n = (p - 1) \* (q - 1)  
  
 *# Если phi\_n не взаимно простое с e то выбирается новое phi\_n* while self.math.gcd(e, phi\_n) != 1:  
 p = self.random.generate\_prime(self.key\_length)  
 q = self.random.generate\_prime(self.key\_length)  
 n = p \* q  
 phi\_n = (p - 1) \* (q - 1)  
  
 *# Вычисляется число d, мультипликативно обратное к числу e* d = self.math.mod\_inverse(e, phi\_n)  
  
 return ((n, e), (n, d))

### Метод encrypt выполняет шифрование данных.

def encrypt(self, data, public\_key):

*"""*

*Шифрование данных.*

*:param data: Данные для шифрования.*

*:param public\_key: Открытый ключ.*

*:return: Зашифрованные данные.*

*"""*

n, e = public\_key

if not isinstance(data, int):

raise TypeError("Данные для шифрования должны быть целым числом")

if data <= 0:

raise ValueError("Данные для шифрования должны быть положительным целым числом")

return self.math.pow(data, e, n)

### Метод decrypt выполняет расшифрование данных.

def decrypt(self, encrypted\_data, private\_key):

"""

*Расшифрование данных.*

*:param encrypted\_data: Зашифрованные данные.*

*:param private\_key: Закрытый ключ.*

*:return: Расшифрованные данные*.

"""

n, d = private\_key

return self.math.pow(encrypted\_data, d, n)

## Возможность работы пользователю

В программе предусмотрена возможность ввода пользователем показателя степени **e** (открытого ключа) и данных для шифрования.

e = int(input("Введите показатель степени e (открытый ключ): "))

while not rsa.math.is\_prime(e):

print("e должно быть простым числом")

e = int(input("Введите e: "))

Программа принимает ввод пользователя, генерирует открытый и закрытый ключи, шифрует и расшифровывает введенные данные, используя алгоритм RSA.

public\_key, private\_key = rsa.generate\_keys(e)  
print("Открытый ключ P={n, e}:", public\_key)  
  
data = int(input("Введите данные для шифрования, целое число: "))  
encrypted\_data = rsa.encrypt(data, public\_key)  
  
print("Зашифрованные данные:", encrypted\_data)  
  
print("Закрытый ключ S={n, d}:", private\_key)  
decrypted\_data = rsa.decrypt(encrypted\_data, private\_key)  
  
print("Расшифрованные данные:", decrypted\_data)

# Тестовые данные

import unittest

from rsa import RSA

class TestRSA(unittest.TestCase):

def setUp(self):

self.rsa = RSA()

self.rsa = RSA()

def test\_is\_prime(self):

# Проверяем, что функция is\_prime корректно определяет простые числа

primes = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47]

for prime in primes:

self.assertTrue(self.rsa.math.is\_prime(prime))

def test\_generate\_prime(self):

# Проверяем, что generate\_prime возвращает простое число

prime = self.rsa.random.generate\_prime(512)

self.assertTrue(self.rsa.math.is\_prime(prime))

def test\_gcd(self):

# Проверяем, что функция gcd возвращает наибольший общий делитель

self.assertEqual(self.rsa.math.gcd(10, 15), 5)

self.assertEqual(self.rsa.math.gcd(14, 63), 7)

self.assertEqual(self.rsa.math.gcd(24, 36), 12)

def test\_mod\_inverse(self):

# Проверяем, что функция mod\_inverse возвращает модульное обратное число

self.assertEqual(self.rsa.math.mod\_inverse(3, 11), 4)

self.assertEqual(self.rsa.math.mod\_inverse(17, 3120), 2753)

def test\_generate\_keys(self):

# Проверяем, что generate\_keys возвращает корректные ключи

e = 65537

public\_key, private\_key = self.rsa.generate\_keys(e)

self.assertEqual(len(public\_key), 2)

self.assertEqual(len(private\_key), 2)

self.assertEqual(public\_key[1], e)

def test\_encrypt\_decrypt\_1(self):

# Проверяем, что данные можно зашифровать и расшифровать

e = 65537

public\_key, private\_key = self.rsa.generate\_keys(e)

data = 123456789

encrypted\_data = self.rsa.encrypt(data, public\_key)

decrypted\_data = self.rsa.decrypt(encrypted\_data, private\_key)

self.assertEqual(decrypted\_data, data)

def test\_encrypt\_decrypt\_2(self):

# Проверяем, что данные можно зашифровать и расшифровать

# 12 ^ 3 mod 55 = 23

# 23 ^ 27 mod55 = 12

e = 3

public\_key, private\_key = self.rsa.generate\_keys(e)

data = 12

encrypted\_data = self.rsa.encrypt(data, public\_key)

decrypted\_data = self.rsa.decrypt(encrypted\_data, private\_key)

self.assertEqual(decrypted\_data, data)

def test\_encrypt\_decrypt\_2(self):

# Проверяем, что данные можно зашифровать и расшифровать

# 12 ^ 3 mod 55 = 23

# 23 ^ 27 mod55 = 12

e = 5

public\_key, private\_key = self.rsa.generate\_keys(e)

data = 24

encrypted\_data = self.rsa.encrypt(data, public\_key)

decrypted\_data = self.rsa.decrypt(encrypted\_data, private\_key)

self.assertEqual(decrypted\_data, data)

def test\_encrypt\_decrypt\_3(self):

# Проверяем, что данные можно зашифровать и расшифровать

e = 5

public\_key, private\_key = self.rsa.generate\_keys(e)

data = 31

encrypted\_data = self.rsa.encrypt(data, public\_key)

decrypted\_data = self.rsa.decrypt(encrypted\_data, private\_key)

self.assertEqual(decrypted\_data, data)

def test\_encrypt\_decrypt\_2(self):

# Проверяем, что данные можно зашифровать и расшифровать

e = 3

public\_key, private\_key = self.rsa.generate\_keys(e)

data = 12

encrypted\_data = self.rsa.encrypt(data, public\_key)

decrypted\_data = self.rsa.decrypt(encrypted\_data, private\_key)

self.assertEqual(decrypted\_data, data)

def test\_encrypt\_decrypt\_2(self):

# Проверяем, что данные можно зашифровать и расшифровать

e = 3

public\_key, private\_key = self.rsa.generate\_keys(e)

data = 12

encrypted\_data = self.rsa.encrypt(data, public\_key)

decrypted\_data = self.rsa.decrypt(encrypted\_data, private\_key)

self.assertEqual(decrypted\_data, data)

def test\_encrypt\_decrypt\_large\_data(self):

# Проверяем, что большие данные можно зашифровать и расшифровать

e = 65537

public\_key, private\_key = self.rsa.generate\_keys(e)

data = 12345678901234567890

encrypted\_data = self.rsa.encrypt(data, public\_key)

decrypted\_data = self.rsa.decrypt(encrypted\_data, private\_key)

self.assertEqual(decrypted\_data, data)

def test\_encrypt\_decrypt\_negative\_data(self):

# Проверяем, что отрицательные данные можно зашифровать и расшифровать

e = 65537

public\_key, private\_key = self.rsa.generate\_keys(e)

data = -123456789

with self.assertRaises(ValueError):

self.rsa.encrypt(data, public\_key)

def test\_encrypt\_decrypt\_zero\_data(self):

# Проверяем, что данные равные нулю вызывают исключение ValueError

e = 65537

public\_key, private\_key = self.rsa.generate\_keys(e)

data = 0

with self.assertRaises(ValueError):

self.rsa.encrypt(data, public\_key)

def test\_encrypt\_decrypt\_string\_data(self):

# Проверяем, что строки можно зашифровать и расшифровать

e = 65537

public\_key, private\_key = self.rsa.generate\_keys(e)

data = "Hello, RSA!"

with self.assertRaises(TypeError):

self.rsa.encrypt(data, public\_key)

def test\_encrypt\_decrypt\_empty\_data(self):

# Проверяем, что пустые данные можно зашифровать и расшифровать

e = 65537

public\_key, private\_key = self.rsa.generate\_keys(e)

data = ""

with self.assertRaises(TypeError):

self.rsa.encrypt(data, public\_key)

def test\_encrypt\_decrypt\_large\_key(self):

# Проверяем, что данные можно зашифровать и расшифровать с очень большим ключом

e = 2\*\*16 + 1

public\_key, private\_key = self.rsa.generate\_keys(e)

data = 123456789

encrypted\_data = self.rsa.encrypt(data, public\_key)

decrypted\_data = self.rsa.decrypt(encrypted\_data, private\_key)

self.assertEqual(decrypted\_data, data)

def test\_encrypt\_decrypt\_different\_keys(self):

# Проверяем, что данные, зашифрованные с одним ключом, расшифровываются с другим ключом

e1 = 65537

public\_key1, private\_key1 = self.rsa.generate\_keys(e1)

e2 = 3

public\_key2, private\_key2 = self.rsa.generate\_keys(e2)

data = 123456789

encrypted\_data = self.rsa.encrypt(data, public\_key1)

decrypted\_data = self.rsa.decrypt(encrypted\_data, private\_key2)

self.assertNotEqual(decrypted\_data, data)

def test\_encrypt\_decrypt\_large\_key\_with\_large\_data(self):

# Проверяем, что очень большие данные можно зашифровать и расшифровать с очень большим ключом

e = 2\*\*16 + 1

public\_key, private\_key = self.rsa.generate\_keys(e)

data = 2\*\*1000

encrypted\_data = self.rsa.encrypt(data, public\_key)

decrypted\_data = self.rsa.decrypt(encrypted\_data, private\_key)

self.assertEqual(decrypted\_data, data)

def test\_encrypt\_decrypt\_random\_data(self):

# Проверяем, что случайные данные можно зашифровать и расшифровать

import random

e = 65537

public\_key, private\_key = self.rsa.generate\_keys(e)

data = random.randint(0, 2\*\*512)

encrypted\_data = self.rsa.encrypt(data, public\_key)

decrypted\_data = self.rsa.decrypt(encrypted\_data, private\_key)

self.assertEqual(decrypted\_data, data)

def test\_encrypt\_decrypt\_float\_data(self):

# Проверяем, что вещественные данные можно зашифровать и расшифровать

e = 65537

public\_key, private\_key = self.rsa.generate\_keys(e)

data = 1234.5678

with self.assertRaises(TypeError):

self.rsa.encrypt(data, public\_key)

def test\_encrypt\_decrypt\_negative\_float\_data(self):

# Проверяем, что отрицательные вещественные данные можно зашифровать и расшифровать

e = 65537

public\_key, private\_key = self.rsa.generate\_keys(e)

data = -1234.5678

with self.assertRaises(TypeError):

self.rsa.encrypt(data, public\_key)

def test\_encrypt\_decrypt\_fraction\_data(self):

# Проверяем, что дробные данные можно зашифровать и расшифровать

e = 65537

public\_key, private\_key = self.rsa.generate\_keys(e)

data = 1/3

with self.assertRaises(TypeError):

self.rsa.encrypt(data, public\_key)

if \_\_name\_\_ == '\_\_main\_\_':

unittest.main()