ЛАБОРАТОРНАЯ РАБОТА №5

ВАРИАНТ 8

КРАЖЕВСКИЙ АЛЕКСЕЙ ИГОРЕВИЧ, 15 ГРУППА

Условие варианта:

8	1102914252601991	571301412050021	624840313709071966800768010501	267222621555915275276288463243	291064433434228628162063527294

Для выполнения всех заданий необходимо использовать "длинную" арифметику. Разрешается использовать любую готовую библиотеку или написать свою.

Шаг 1. В условиях своего варианта (см. в таблице 1) для заданных чисел p, q и e, необходимо вычислить личный ключ d. Можно использовать либо расширенный алгоритм Евклида или малую теорему Ферма (частный случай теоремы Эйлера).

Шаг 2. Для заданного сообщения X1, вычислить зашифрованное сообщение Y1, используя открытый ключ e.

Шаг 3. Расшифровать сообщение Y1, используя найденный личный ключ d, сравнить результат с исходным сообщением X1.

Шаг 4. Для заданного шифртекста Y2, вычислить исходный открытый текст X2, используя личный ключ d.

Описание RSA.

RSA— криптографический алгоритм с открытым ключом, основывающийся на вычислительной сложности задачи факторизации больших целых чисел.

Генерация ключей:

Публичный ключ e нам дан по условию. Для вычисления секретного ключа d нужно использовать формулу $d = e^{-1} \ (mod\ \phi(n))$. Т.е., нужно найти обратный элемент для e.

Шифрование происходит по формуле
$E_e(X) = X^e \ (mod\ n)$, где n –модуль, e – открытый ключ.

Расшифровка же по формуле $D_d(Y) = Y^d \ (mod\ n)$, где n –модуль, d –личный ключ.

Алгоритм возведения в степень.

Я использовал алгоритм бинарного возведения в степень, работающий за O(log n) и сразу считающий по модулю. Вот сам алгоритм:

```python
res = 1
while n:
    if n & 1:
        res *= a % mod
    a *= a % mod
    n >>= 1
return res % mod
```

Сам алгоритм основан на тождестве:

$$a^n = (a^{n/2})^2 = a^{n/2} \cdot a^{n/2}$$

- это для четной степени

В случае нечетной степени просто переходим к четной $a^n = a^{n-1} \cdot a$

Нахождение обратного элемента по модулю.

Сам алгоритм использует расширенный алгоритм Евклида.

```python
def eea(a, b):
    if b == 0:
        return 1, 0
    (q, r) = (a//b, a % b)
    (s, t) = eea(b, r)
    return t, s - (q * t)
```

И после считает обратный элемент:

```python
inv = eea(e, phi)[0]
if inv < 1:
    inv += phi
return inv
```

Значения.

Значения, полученные из варианта и в ходе выполнения кода:

```
p = 1102914252601991
q = 571301412050021
Public key (key, n): (624840313709071966800768010501, 630096469881611006140696191811)
Private key (key, n): (559958735455296994218786003301, 630096469881611006140696191811)
Encrypted y1 = 438463217645674859061541778951
Decrypted x1 = 267222621555915275276288463243
Check x1: True
Decrypted x2 = 86698905067844839532213957140
y2 = 291064433434228628162063527294
```

Дополнительное задание – реализация метода факторизации числа

Я реализовал p – 1 метод Полланда.

Код метода:

```
b = 2

for i in range(2, B):
    # b is updated as a mod power
    b = fast_pow(b, i, n)
    # d is gcd of b-1 and n
    d = gcd(b-1, n)

    # if d is not 1 or n then a factor has been found
    if 1 < d < n:
        return d, n // d
```

Верхняя граница вводится пользователем. На каждой итерации (от 2 до введенной границы) мы считаем b и d и проверяем на принадлежность промежутку от 1 до числа. Если условие выполняется, значит мы нашли простой делитель.

Результат выполнения для теста с числом 10001 и границей 10.

```
Factorization of 10001 with bound 10: (73, 137)
```