Реализация протокола RSA.

Алгоритм RSA и обоснование его работы изложено в Заметках об алгоритме RSA. Решение некоторых задач опирается на теоремы, сформулированные и доказанные в этой статье. Ссылки на эти теоремы даны в условии задач.

A. *Шифрование с публичным ключом*

По данному публичному ключу (m, e) зашифровать и вывести сообщение x.

Все числа могут содержать до 500 десятичных цифр, $x < m$.

Даны числа m, e, x. Вывести $E(x)$.

Input	Output
21 5 3	12
12874165001 2345764347 872345692	6555996597

B. *Дешифрование с секретным ключом*

По данному секретному ключу (p, q, d) и зашифрованному с модулем $m = pq$ сообщению $E(x)$ вывести дешифрованное сообщение x.

Все числа могут содержать до 500 десятичных цифр, $x < pq$.

Даны числа $p, q, d, E(x)$. Вывести x.

Input	Output
5 7 5 32	2
839 761 424587 286370	283950

C. *Генерирование публичного ключа и шифрование с его помощью*

Дано сообщение x и битовая длина простых чисел p и q (модуль $m = pq$) в алгоритме RSA.

Гарантируется, что заданное сообщение (целое число) меньше сгенерированного модуля при условии, что битовая длина обоих его простых множителей равна b.

Пусть $m = pq$. Выбрать простое e такое, что НОД$(e, \varphi(m)) = 1$. Зашифровать сообщение x публичным ключом (m, e).

Вывести 5 строк: публичный ключ (m, e), выбранные p и q и зашифрованное сообщение $E(x)$.

Input	Output
42 4	143 7 13 11 81
314159 16	1842845239 5 36973 49843 598205699

D. *Дешифрование со слабым модулем*

Даны публичный ключ (m, e) и зашифрованное им сообщение $E(x)$. Автор ключа пренебрёг указаниями по его созданию и хотя модуль и является произведением двух различных простых чисел, тем не менее $m < 10^{12}$.

Покажите, что так делать нельзя и расшифруйте сообщение.

Даны числа m, e и $E(x)$. Выведите исходное сообщение x и заодно секретный ключ (p, q, d).

Input	Output
143 7 62	101 11 13 103
770681 13 455104	314159 773 997 709765

E. *Дешифрование по известному значению $\varphi(m)$*

Автор публичного ключа снова позабыл про то, чем можно делиться, а чем нет. В результате вместе со значением публичного ключа (m, e) стало известно значение $\varphi(m)$. Расшифруйте по этим данным зашифрованное сообщение, не используя вероятностных алгоритмов.

Дан публичный ключ (m, e), значение $\varphi(m)$ и зашифрованное сообщение $E(x)$.

Вывести секретный ключ (p, q, d) и расшифрованное сообщение x. Числа p и q укажите в порядке возрастания.

Указание: используйте результат Теоремы 5.4 (Заметки об алгоритме RSA, стр. 11).

Input	Output
55 13 40 16	5 11 37 36
43337531 17 43324272 17942930	5843 7417 12742433 12041961

F. *Дешифрование по известному кратному значению $\varphi(m)$*

Предыдущую задачу можно обобщить (правда, способ решения будет совсем иной). Оказывается, можно дешифровать сообщение, найдя разложение модуля на простые p и q, по известному публичному ключу и какому-то *кратному* функции Эйлера $\varphi(m)$.

Дан публичный ключ (m, e), кратное функции Эйлера $\varphi(m)$ и зашифрованное сообщение $E(x)$.

Расшифруйте по этим данным сообщение и выведите секретный ключ (p, q, d) и расшифрованное сообщение x. Числа p и q укажите в порядке возрастания.

Указание: используйте результат Теоремы 5.6 (Заметки об алгоритме RSA, стр. 12).

Input	Output
55 13 1680 16	5 11 37 36
43337531 17 67022648784 17942930	5843 7417 12742433 12041961

Если вам непонятно, как работать с бинарными файлами и/или вы не понимаете, как применить шифрование целых чисел к шифрованию файла (текстового или бинарного), почитайте соответствующие разделы после задачи J.

G. *Шифрование бинарного файла*

Вам известны следующие параметры шифрования:

`p = 41863, q = 61261`

`e = 94397, d = 175850813`

Зашифруйте входной файл публичным ключом. Результат запишите в выходной файл `output.txt` в бинарном виде. В зашифрованный файл первым блоком надо записать (зашифрованную) информацию о размере входного файла в байтах.

Простые числа p и q подобраны таким образом, чтобы битовая длина их произведения $m = pq$ была равна 32. При шифровании надо выбрать размер блока, равный 3.

Тестовый файл для шифрования: ссылка.

Тестовый зашифрованный файл: ссылка.

H. *Дешифрование бинарного файла*
Вам известны следующие параметры шифрования:
`p = 41863, q = 61261`
`e = 94397, d = 175850813`
Вам дан файл, зашифрованный публичным ключом. Расшифруйте его и запишите в выходной файл `output.txt` в бинарном виде. Считайте, что при шифровании файла был выбран размер блока, равный 3.

В этой задаче некоторые файлы-ответы являются не бинарными, а текстовыми. Так что в случае прохождения тестов можно посмотреть отчёт о тестировании. Там может встретиться любопытное.

I. *Цифровая подпись бинарного файла*
Цифровая подпись нужна для того, чтобы получатель сообщения мог удостовериться в личности отправителя.
Пусть у отправителя есть сообщение `M` и пара ключей — публичный `P` и секретный `S`, соответствующие модулю с битовой длиной, равной `B`. У получателя есть только публичный ключ отправителя `P`. Вместе с сообщением `M` отправитель посылает сертификат, который получатель затем анализирует и убеждается в том, что именно отправитель составил это сообщение (или убеждается в обратном). Собственно, эта задача о том, как по бинарному файлу сделать его сертификат.
Создание сертификата состоит из трёх шагов:

- вычисление по сообщению `M` его hash-значения (используйте `sha3_512()` из модуля `hashlib`, её аргументом должна быть последовательность байтов);
- дополнение этого hash-значения до длины модуля в RSA-ключе (у нас она будет равна 3072 бита);
  Схема дополнения следующая:

0x00	0x01	0xFF…FF	0x00	*data*

  Здесь `data` это hash-значение длиной 512 бит. `0x00` и `0x01` это запись байта с нулём и единицей двумя шестнадцатеричными цифрами. Длину блока из чисел $255 = FF_{16}$ надо подобрать, чтобы вся последовательность была 3072 бита длиной.
- шифрование полученной последовательности секретным ключом отправителя (размер блока выберите равным 32 бита).

Полученное число (мѐньшее, чем модуль шифрования `M`) и надо в виде $\frac{3072}{8} = 384$ байтов записать в качестве сертификата.

Для этой задачи был заранее сгенерирован ключ длины 3072 бита, он хранится в файле RSA_keys.txt.
Для создания ключа шифрования удобно написать конструктор RSA-ключа таким образом, чтобы он мог генерировать ключи не только по заданной длине модуля, а также ключи с заданными параметрами p, q, e и d.

Создайте сертификат для входного бинарного файла и запишите в бинарный файл с ответом сначала сертификат, а затем сам исходный файл (подряд без пропусков и разделителей).
Тестовый файл для подписи: ссылка.
Подисанный файл с сертификатом: ссылка.

J. *Проверка цифровой подписи*
В этой задаче надо проверить файл, содержащий сообщение и его цифровую подпись. Алгоритм подписи и формат правильно подписанного файла такой же, как и в прошлой задаче.
Можно считать, что файл содержит больше 384 байтов: ровно 384 байтов подписи и непустое сообщение, сертификатом которого предположительно является эта подпись.

Если сообщение действительно подписано обладателем имеющегося у вас публичного ключа, выведите `YES`, иначе выведите `NO`.
Для самостоятельного тестирования можно взять правильную программу для предыдущей задачи, сгенерировать *правильный* подписанный файл и убедиться, что на нём ваша программа выдаёт `YES`, а если в нём поменять что угодно (в подписанных данных или в сертификате) — то `NO`.

Бинарные файлы

Прежде чем приступать к реализации алгоритма RSA, надо разобраться с некоторыми техническими особенностями. Во-первых, все манипуляции до сих пор сводились к действиям с числами, а хочется шифровать файлы. Кроме того, надо научиться шифровать (и потом дешифровать) файлы произвольной природы, поэтому чтение и запись файлов надо делать в т.н. `binary mode`.

```
open(file_input, "rb") # read
open(file_output, "wb") # write
```

Для работы с бинарными данными в Python есть два типа: `bytes` и `bytearray`. Их можно рассматривать, как аналоги строк и массивов соответственно (`bytes` — неизменяемый тип, `bytearray` — изменяемый), но хранить они могут только байты — числа в диапазоне $0 \dots 255$. Результат чтения всего файла `f.read()` имеет тип `bytes`.

Оба типа поддерживают индексацию и срезы. Документацию по типам `bytes` и `bytearray` можно посмотреть здесь.
Например, такой код перебирает куски по 3 байта из прочитанного файла:

```
CHUNK_SIZE = 3

with open("test.dat", "rb") as f:
    bb = f.read()
    for k in range(0, len(bb), CHUNK_SIZE):
        print(f"{bb[k:k + CHUNK_SIZE]}", end = "")
        print([bb[i] for i in range(k, min(len(bb), k + CHUNK_SIZE))])
```

В выводе нетрудно заметить числовые значения ASCII-кодов. Попробуйте восстановить содержимое файла (`\r` — возврат каретки, `\n` — перевод на следующую строку).

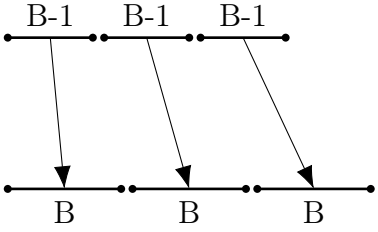
```
b"Thi"[84, 104, 105]
b"s i"[115, 32, 105]
b"s\r\n"[115, 13, 10]
b"an\r"[97, 110, 13]
b"\nex"[10, 101, 120]
b"amp"[97, 109, 112]
b"le\r"[108, 101, 13]
b"\n"[10]
```

Как шифровать

Пусть у вас уже есть пара ключей — публичный (m, e) и секретный (p, q, d), причём модуль m имеет определённую и известную длину, кратную 8 битам. То каким ключом вы шифруете, зависит от того, какой именно протокол вы реализуете, на процесс это никак не влияет.
Кратко процесс шифрования файла выглядит так:

- любой файл, вне зависимости от содержимого рассматривается как последовательность байтов;
- надо разбить файл на куски равной длины (зависящей от m) + возможно, останется один кусок мѐньшего размера;
- каждый такой кусок представить в виде *одного* числа, для чего можно использовать встроенный метод `from_bytes()`;
- зашифровать это число;
- результат шифрования представить в виде набора байтов (для этого пригодится метод `to_bytes()`) и записать в файл.

Чтобы понять, где тут могут случиться проблемы, и как их поправить, рассмотрим пример.
Зафиксируем битовую длину модуля в RSA-ключе, пусть она равна $B = 32$ бит. Это значит, что брать из файла куски больше, чем по 4 байта ($4 \cdot 8 = 32$ бита) нельзя — они гарантированно больше m.
Более того, даже брать куски ровно по 4 байта тоже нельзя, потому что получающееся 32-битное число также может оказаться больше m.[1]
Сгруппируем байты кусками по 3. После возведения такого 24-битного числа x в степень по модулю m может получиться число *длиннее 3 байт*, ведь результат этой операции ограничен сверху 32-битным (4-байтовым) модулем m.
Тем не менее, этот способ годится, только надо правильно его реализовать. Возьмём для шифрования куски на 1 байт меньше, чем размер модуля (в нашем случае 3 байта). Записывать в зашифрованный файл будем куски по 4 байта (даже если число фактически занимает меньше 4 байт).



После дешифрования у нас будут получаться не более, чем 3-байтовые числа и их в виде последовательности байтов мы и запишем в файл. Особенностью такого подхода является больший размер размер зашифрованного файла (в $\dfrac{B}{B-1}$ раз).
Остался ещё один нюанс — размер исходного файла может оказаться некратным $B - 1$. Можно перед шифрованием посчитать количество байт в «хвосте»[2] и записать это число, например, первым байтом в зашифрованный файл. [3]

[1] Подумайте, почему такой проблемы не будет (т.е. можно брать блоки по B/4 байт), если вы шифруете текстовый файл.
[2] Это остаток при делении длины файла в байтах на длину блока: `len(open(<filename>, "rb").read()) % (B - 1)`
[3] Размер зашифрованного файла без учёта этого «технического» байта всегда кратен B.

Как расшифровывать

В начале надо прочитать из зашифрованного файла размер некратной $B - 1$ части исходного файла (первый байт или блок, в зависимости от того, как это было сделано в алгоритме шифрования). Длина оставшейся части файла точно кратна B. Причём все куски по B байт кроме последнего точно займут ровно $B - 1$ байт после дешифрования. Последний кусок после дешифрования может занимать (в смысле занимал в исходном файле) 1, 2 или 3 байта.

Пример

Зафиксируем битовую длину модуля $B = 32$.
Сгенерируем пару простых чисел p, q, битовая длина произведения которых была бы равна 32: $p = 41863, q = 61261, m = pq = 2564569243$.

Также выберем простое $e = 94397$, такое что $(e, \varphi(m)) = 1$ (проверьте!)

Для выбранного e из соотношения $ed \equiv 1 \bmod \varphi(m)$ вычислим показатель дешифрующей экспоненты $d = 175850813$.

Пусть текстовый файл содержит 4 буквы: `bcde`. Их ASCII-коды от 98 до 101 [4], в двоичном виде это такая строка:

$$01100010\ 01100011\ 01100100\ 01100101$$

Разобьём исходный файл на два блока: в первом 3 байта, во втором 1. Будем записывать в зашифрованный файл длину всего файла (его тоже зашифруем для порядка). Тогда нам надо записать результат шифрования следующих трёх чисел:

- `4` — это длина файла;
- `6447972 = int('011000100110001101100100', 2)` — число, получившееся склеиванием первых 3 байтов в одно число;
- `101 = int('01100101', 2)` — число, оставшееся в «хвосте».

Применим ко всем трём числам функцию шифрования $x^e \bmod m$:

$4^{94397} \bmod 2564569243 = 1562314617$

$6447972^{94397} \bmod 2564569243 = 1313154801$

$101^{94397} \bmod 2564569243 = 609943375$

В зашифрованный файл каждое число надо записать в виде последовательности 4 байт. Получается такая последовательность чисел:

$1562314617 = 1011101\ 00011111\ 00000111\ 01111001_2$

$1562314617 \to 5d_{16} = 01011101_2; 1f_{16} = 00011111_2; 7_{16} = 00000111_2; 79_{16} = 01111001_2$

$1313154801 = 1001110\ 01000101\ 00100110\ 11110001_2$

$1313154801 \to 4e_{16} = 01001110_2; 45_{16} = 01000101_2; 26_{16} = 00100110_2; f1_{16} = 11110001_2$

$609943375 = 100100\ 01011010\ 11111111\ 01001111_2$

$609943375 \to 24_{16} = 100100_2; 5a_{16} = 01011010_2; ff_{16} = 11111111_2; 4f_{16} = 01001111_2$

Результат шифрования в шестнадцатеричном виде:

Address	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
00000000	5d	1f	07	79	4e	45	26	f1	24	5a	ff	4f				

Так выглядит файл в текстовом редакторе:

]USBELyNE&ñ$ZÿO

[4] Для просмотра бинарных файлов удобно поставить плагин HEX-Editor к Notepad++, он позволяет смотреть на побайтовое содержимое файла в виде последовательности шестнадцатеричных чисел.