RSA解题方法整理

适用于已知 n, e, c 需要解出m 的情况。

n可分解的情况

大合数分解方法

FactorDB + libnum + gmpy2

overview

Factor DB is the database to store known factorizations for any number.

Libnum is a python library for some numbers functions:

- working with primes (generating, primality tests)
- common maths (gcd, lcm, n'th root)
- modular arithmetics (inverse, Jacobi symbol, square root, solve CRT)
- converting strings to numbers or binary strings

gmpy2 is a C-coded Python extension module that supports multiple-precision arithmetic.

install

```
# install FactorDB
sudo pip install factordb-pycli
# install libnum
git clone https://github.com/hellman/libnum
cd libnum
python setup.py install
# install gmpy2
brew install mpfr # for OSX
brew install libmpc # for OSX
pip install gmpy2
```

usage

```
# factordb
>>> from factordb.factordb import FactorDB
>>> f=FactorDB(1424)
>>> f.connect()
<Response [200]>
>>> f.get_factor_list()
[2, 2, 2, 2, 89]
# libnum
>>> libnum.gcd(20,16)
```

```
4
>>> libnum.s2n('abc')
6382179
>>> >>> libnum.s2b('a')
'01100001'
>>> libnum.s2b('b')
'01100010'
>>> libnum.s2b('ab')
'0110000101100010' # 所以说s2b其实是二进制的拼接
# gmpy2
>>> gmpy2.invert(3,20) # returns the inverse of x modulo m
mpz(7) # The gmpy2 mpz type supports arbitrary precision integers. It
should be a drop-in replacement for Python’s long type.
```

完整代码：

```
def RSA_normal_getM(n, e, c, output='number'):  # output=number/string
    import warnings
    from factordb.factordb import FactorDB
    import libnum
    import gmpy2

    f=FactorDB(n)
    f.connect()
    f_fl=f.get_factor_list()
    if not len(f_fl)==2:
        warnings.warn("division of n fails")
    p=f_fl[0]
    q=f_fl[1]
    d=gmpy2.invert(e,(p-1)*(q-1))
    m=pow(c,d,n)
    if output=='string':
        #print libnum.n2s(m)
        return libnum.n2s(m)
    #print m
    return int(m)
```

e很大

可以尝试 wiener 攻击。

原理参考这篇和维基百科。

完整代码如下：

```
def bitlength(x):
  assert x >= 0
  n = 0
```

```
  while x > 0:
    n = n+1
    x = x>>1
  return n

# Squareroots an integer
def isqrt(n):
  if n < 0:
    raise ValueError('square root not defined for negative numbers')
  if n == 0:
    return 0
  a, b = divmod(bitlength(n), 2)
  x = 2**(a+b)
  while True:
    y = (x + n//x)//2
    if y >= x:
      return x
    x = y

# Checks if an integer has a perfect square
def is_perfect_square(n):
  h = n & 0xF; #last hexadecimal "digit"
  if h > 9:
    return -1 # return immediately in 6 cases out of 16.
  # Take advantage of Boolean short-circuit evaluation
  if ( h != 2 and h != 3 and h != 5 and h != 6 and h != 7 and h != 8 ):
    # take square root if you must
    t = isqrt(n)
    if t*t == n:
      return t
    else:
      return -1
  return -1

# Calculate a sequence of continued fractions
def partial_quotiens(x, y):
  partials = []
  while x != 1:
    partials.append(x // y)
    a = y
    b = x % y
    x = a
    y = b
  #print partials
  return partials

# Helper function for convergents
def indexed_convergent(sequence):
  i = len(sequence) - 1
```

```
  num = sequence[i]
  denom = 1
  while i > 0:
    i -= 1
    a = (sequence[i] * num) + denom
    b = num
    num = a
    denom = b
  #print (num, denom)
  return (num, denom)

# Calculate convergents of a  sequence of continued fractions
def convergents(sequence):
  c = []
  for i in range(1, len(sequence)):
    c.append(indexed_convergent(sequence[0:i]))
  #print c
  return c

# Calculate `phi(N)` from `e`, `d` and `k`
def phiN(e, d, k):
  return ((e * d) - 1) / k

# Wiener's attack, see http://en.wikipedia.org/wiki/Wiener%27s_attack for
more information
def RSA_wiener_attack(N,e,c,output='number'):
  (p,q,d) = (0,0,0)
  conv=convergents(partial_quotiens(e,N))
  for frac in conv:
    (k,d)=frac
    if k == 0:
      continue
    y = -(N - phiN(e, d, k) + 1)
    discr = y*y - 4*N
    if(discr>=0):
      # since we need an integer for our roots we need a perfect squared
discriminant
      sqr_discr = is_perfect_square(discr)
      # test if discr is positive and the roots are integers
      if sqr_discr!=-1 and (-y+sqr_discr)%2==0:
        p = ((-y+sqr_discr)/2)
        q = ((-y-sqr_discr)/2)
        m=pow(c,d,N)
        if output=='string':
            #print libnum.n2s(m)
            return libnum.n2s(m)
        #print m
        return int(m)
        #return p, q, d
```

```
    return 0
```

已知部分明文攻击

这种类型的题目都是Stereotyped messages，在强网杯的next-RSA中，就是属于知道明文高位的类型。

其实强网杯的这道题目e只有3，我估计直接用三次方程的求根公式就可以算出来，我会用sage写一个尝试一下，现在我来研究一下其他形式的stereotyped message类型的题目应该怎么处理比较好。

这里就要提到一个叫做`coppersmith`的方法，当然也是一个人...总是最早这个方法提出来，是为了解决"find all small integer roots of some polynomial equaitions"的。为了让RSA可以被这种方法攻击，我们需要构造多项式，想要构造多项式，就需要除了公开的`n`,`e`和`c`之外的其他的信息，在这里我们讨论的是”部分明文“的情况。

于是，我们得到的就是如下问题：

The Problem (Univariate Modular Case):

- **Input:**
  - A polynomial $f(x) = x^\delta + a_{d-1}x^{\delta-1} + \cdots + a_1 x + a_0$.
  - N an integer of unknown factorization.
- **Find:**
  - All integers x_0 such that $f(x_0) \equiv 0 \mod N$.

Coppersmith定理提出的是只要根`x0`满足某个条件（其实就是特别小）的话，就可以在多项式时间内找到所有的满足如上条件的根。

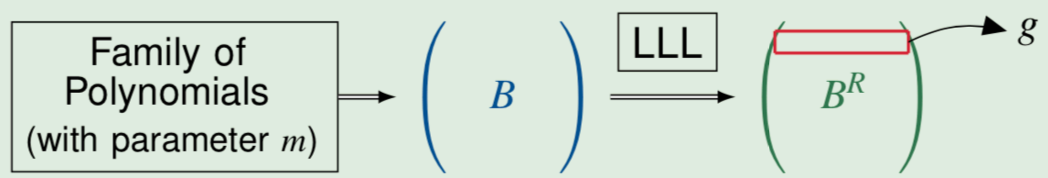
Coppersmith's Theorem for the Univariate Modular case

- The solutions x_0 can be found in polynomial time in $\log(N)$ if

$$|x_0| < N^{1/\delta}\ .$$

然后呢，现在又要简化问题，于是就有人（也许就是coppersmith，我不知道）发明了一个方法，叫做“LLL Reduction"，大概意思是用`Euclidean Lattices`生成一个新的多项式，比原来的多项式更好解一点。

How to find the polynomial g:



首先需要生成“一族”多项式，然后要构造一个`Coppersmith matrix`，并不知道这个东西是干什么用的。。

Consider the Family of Polynomials:

$$g_{i,j}(x) = x^j N^{m-i} f^i(x)$$

- For $0 \leq i < m$ and $0 \leq j < \delta$
- For $i = m$ and $j = 0$

Crucial Property of the Polynomials

☞ $g_{i,j}(x_0) \equiv x_0^j N^{m-i} N^i \equiv x_0^j N^m \equiv 0 \mod N^m$

Construction of the Matrix with $|x_0| < X$

- **For i from 0 to $m-1$**

  For j from 0 to $\delta - 1$

  $$M[\delta i + j] = (xX)^j N^{m-i} f^i(xX)$$
- $M[\delta m] = f^m(xX)$

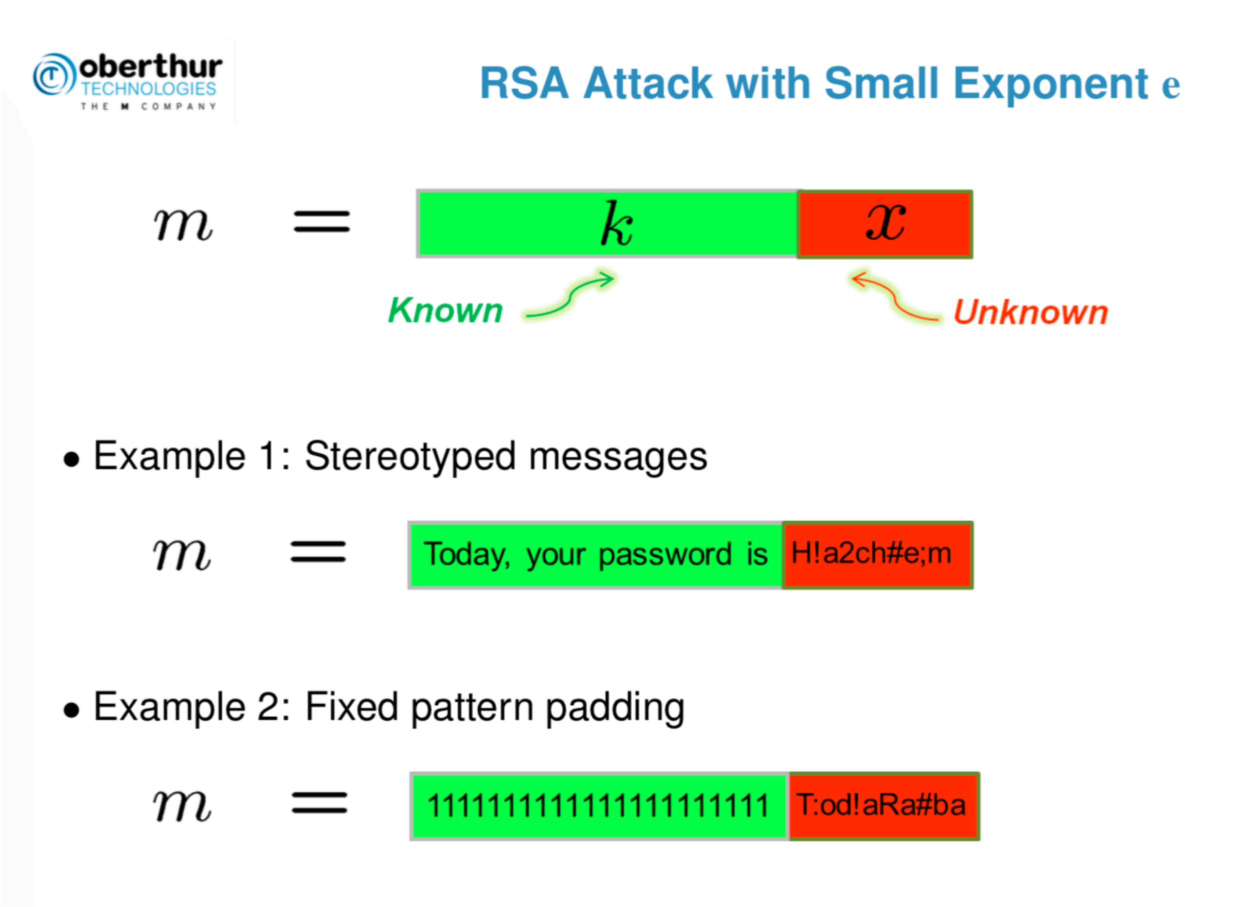
然后就可以找到一个good-basis，于是就做到了Lattice Reduction。而且有很多方法来找Lattice Reduction，比如HKZ、LLL之类的。然后就进入了“find small solution” 的阶段。

有了上面一步的`LLL-reduced basis`就可以直接得到`g(x)`了，也就是一开始提到的“更加容易解的多项式”。

Obtain Polynomial g such that $g(x_0) \equiv 0 \mod N^m$

- First vector of the LLL-reduced basis: $v = (v_0, v_1, \ldots, v_{d-1})$
- Get new polynomial $g(x) = v_0 + \frac{v_1}{X}x + \cdots + \frac{v_{d-1}}{X^{d-1}}x^{d-1}$

现在终于可以看看coppersmith在RSA中的应用了。。上面的一堆东西我真的是很懵逼。



这个就是next-RSA这道题目里的情况，一个是高位的message已知，还有就是e比较小。用简单的变量替换就可以得到如下方程：

Eve knows the two polynomials:

$$\begin{cases} p_1(k, r_1) & = & (k + r_1)^e - C_1 & \equiv & 0 \mod N \\ p_2(k, r_2) & = & (k + r_2)^e - C_2 & \equiv & 0 \mod N \end{cases}$$

Perform a change of variables:

$$\begin{cases} x & = & k + r_1 \\ y & = & r_2 - r_1 \end{cases} \Rightarrow \begin{cases} p_1(x) & = & (x)^e - c_1 & \equiv & 0 \mod N \\ p_2(x, y) & = & (x + y)^e - c_2 & \equiv & 0 \mod N \end{cases}$$

然后就到了最关键的步骤！！！之前我好像看懂了现在又有点忘记了。。

好的，又想起来了。我们把

$$p_1(x) = (x)^e - c_1 \equiv 0 \mod N$$

记作1️⃣式，把

$$p_2(x, y) = (x + y)^e - c_2 \equiv 0 \mod N$$

记作2️⃣式。由于`y=r1-r2`，而`r1`和`r2`都只是`m`中的一部分（甚至是很小的一部分），所以说`r1-r2`也就是说`y`，是可以用coppersmith方法解出来，一旦这个`y`被解出来，事情就变的很简单了。式1️⃣和式2️⃣中的`x`都是`m`，也就是说如果把式1️⃣和式2️⃣的左半边展开，都应该会有`(x-m)`这个因子，而且共同的因子也只有这一个（否则如果RSA不就有歧义了吗。。）。也就是说，式1️⃣和式2️⃣的左半边的最大公因子是`x-m`，也就有了如下的解法：

Use resultants and Coppersmith's method

- Get a new polynomial $p(y) \equiv 0 \mod N$ of degree e^2
- Solution $y_0 = r_2 - r_1$ found if $|y_0| < N^{1/e^2}$
- Compute $gcd(p_1(x), p_2(x)) = x - m$

其实！！！如果原本的“明文已知部分”已经很多了，那就十分的开心，因为是可以直接用coppersmith方法解出剩余部分的`m`的。。比如next-rsa这道题就可以直接用如下代码解出：

```
from random import randrange

n = 
0x79982a272b9f50b2c2bc8b862ccc617bb39720a6dc1a22dc909bbfd1243cc0a03dd406ec0
b1a78fa75ce5234e8c57e0aab492050906364353b06ccd45f90b7818b04be4734eeb8e859ef
92a306be105d32108a3165f96664ac1e00bba770f04627da05c3d7513f5882b2807746090ce
bbf74cd50c0128559a2cc9fa7d88f7b2d
e = 3
mbar = 
0xfedcba9876543210000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000
c = 
0x381db081852c92d268b49a1b9486d724e4ecf49fc97dc5f20d1fad902b5cdfb49c8cc1e96
8e36f65ae9af7e8186f15ccdca798786669a3d2c9fe8767a7ae938a4f9115ae8fed4928d95a
d550fddd3a9c1497785c9e2279edf43f04601980aa28b3b52afb55e2b34e5b175af25d5b3bd
71db88b3b31e48a177a469116d957592c

beta = 1
epsilon = beta^2/7

nbits = n.nbits()
kbits = floor(nbits*(beta^2/e-epsilon))

print "upper %d bits (of %d bits) is given" % (nbits-kbits, nbits)

PR.<x> = PolynomialRing(Zmod(n))
f = (mbar + x)^e - c

x0 = f.small_roots(X=2^kbits, beta=1)[0]  # find root < 2^kbits with factor
= n
print mbar + x0
print x0
```

两对相近的p、q

来看两对两个大素数的积：

```
n1=0x78e2e04bdc50ea0b297fe9228f825543f2ee0ed4c0ad94b6198b672c3b005408fd8330
c36f55d36fb129d308c23e5cb8f4d61aa7b058c23607cef83d63c4ed0f066fc0b3c0062a2ac
68c75ca8035b3bd7a320bdf29cfcf6cc30377743d2a8cc29f7c588b8043412366ab69ec8243
09cb1ef3851d4fb14a1f0a58e4a1193f5518fa1d0c159621e1f832b474182593db2352ef051
01bf367865ad26efe14fce977e9e48d3310a18b67991958d1a01bd0f3276a669866f4deaef2
a68bfaefd35fe2ba5023a22c32ae8b2979c26923ee3f855363f18d8d58bb1bc3b7f585c9d9f
6618c727f0f7b9e6f32af2864a77402803011874ed2c65545ced72b183f5c55d4d1

n2=0x78e2e04bdc50ea0b297fe9228f825543f2ee0ed4c0ad94b6198b672c3b005408fd8330
c36f55d36fb129d308c23e5cb8f4d61aa7b058c23607cef83d63c4ed0f066fc0b3c0062a2ac
68c75ca8035b3bd7a320bdf29cfcf6cc30377743d2a8cc29f7c588b8043412366ab69ec8243
09cb1ef3851d4fb14a1f0a58e4a1193f5a58ee70a59ac06b64dbe04b876ff69436b78cf0337
1f2062707897bf4e580870e42b5e62709b69f6d4939ac5641ea0f29de44aaee8f2fcd0f66aa
a720b584f7c801e52ce7cd41db45ceb99ebd7b51bef8d0cd2deb5c50b59f168276c9c98d46a
1c37bd3d6ef81f2c6e89028680a172e00d92dd8b392135112dd16efab57d00b26b9
```

这两个数的高位是不是特别相似！没有错，这是因为`n1=p1*q1, n2=p2*q2`，而`p1`和`p2`，`q1`和`q2`之间特别的接近，也就是说`p2=p1+x`，`q2=q1+y`的`x`和`y`特别小。

我们可以得到如下的方程：

```
pq = n
(p + x)(q + y) = n'
xy + py + qx = t     (t = n' - n)
xq^2 + (xy - t)q + ny = 0     (1)
```

只要方程(1)有素数解就可以了！

于是直接爆破`x`和`y`。

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

from gmpy2 import is_prime as prime
from gmpy2 import iroot
```

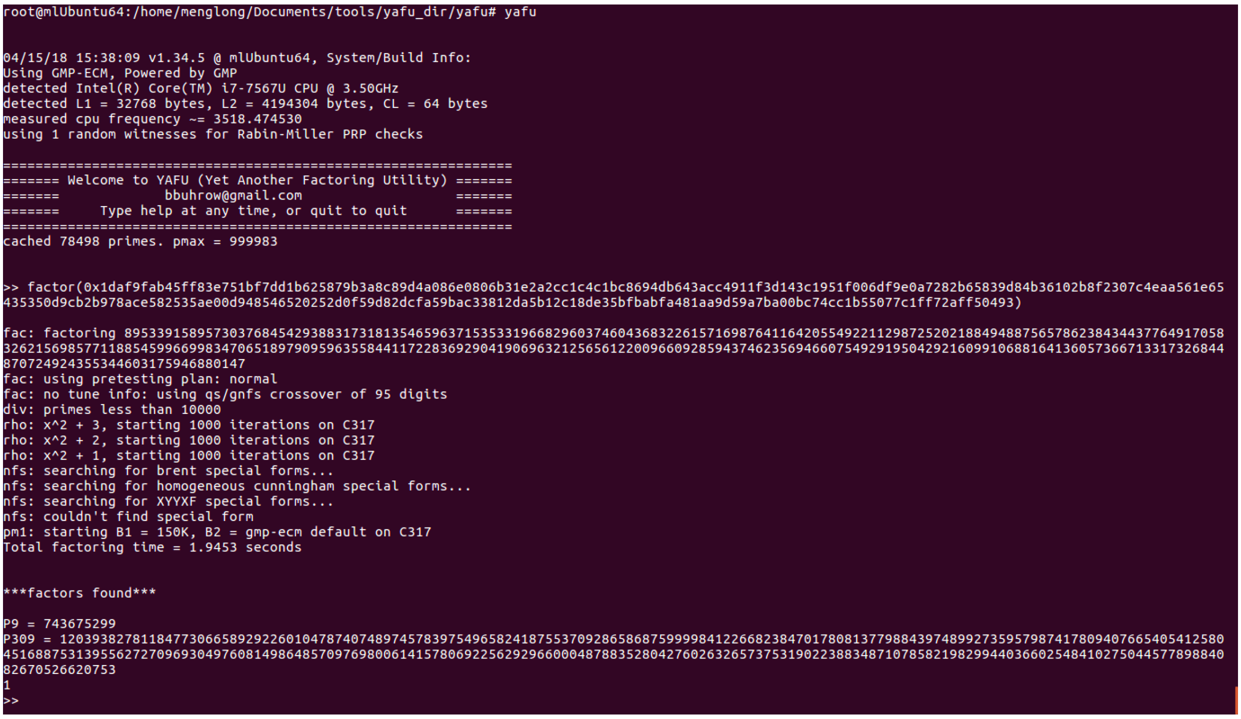
```
n =
152604733989160716862877523402491582793430130771456677945147176923529418734
666906862884422047145858548692268727052930088375540129495980442975960155070
313150061952306447225817446437565739827293444994522001163663278691786946921
620144465787119566633482629327031603941016067084757257428900493119336918497
477076732163227584185304201185025567193235430777718211634398248767518746758
620754019867962720147469250157720455784033573000381928301979858710776212136
698173805771766115604670518489510449635271859167810949818108046413879753983
616949350934737170395503616442523819239948411388171527482794778688540930332
34031994635408593
nn =
152604733989160716862877523402491582793430130771456677945147176923529418734
666906862884422047145858548692268727052930088375540129495980442975960155070
313150061952306447225817446437565739827293444994522001163663278691786946921
620144465787119566633482629327031603941016067084757257428900493119336918497
477079148180827164748812243364727090746790471456677961068952980861290572884
495082182376799743219522794298327072293035944744945199458994813202310919836
656549470818397843753736573450238903326913714435713700166237890418838216103
534737264650375904140081982289569209131815265507755240280594263121734938772
28564452496582329

#  print nn > n
t = nn - n
f1 = lambda x, y: pow(x * y - t, 2) - 4 * n * x * y
f2 = lambda x, y, s: (t - x * y - s) / (2 * x)

for x in xrange(1, 3000):
    for y in xrange(1, 3000):
        print x, y
        if f1(x, y) >= 0:
            s, b = iroot(f1(x, y), 2)
            if b:
                if prime(f2(x, y, int(s))):
                    print "Success"
                    print f2(x, y, int(s))
                    exit()
```

p与q相差过大

直接同yafu解。（说实话我也不知道为什么要用yafu解，可能是yafu特别擅长应对这种情况？）



总之yafu很好用，而且确实可以解出这道题。

低加密指数攻击

低加密指数攻击，显而易见，就是`e`特别小的意思。

于是就可以直接用gmpy2的iroot进行暴力破解。

```
from gmpy2 import iroot

n=0x7003581fa1b15b80dbe8da5dec35972e7fa42cd1b7ae50a8fc20719ee641d6080980125d18039e95e435d2a60a4d5b0aaa42d5c13b0265da4930a874ddadcd9ab0b02efcb4463a33361a84df0c02dfbd05c0fdc01e52821c683bd265e556412a3f55e49517778079cb1c1c1c22ef8a6e0bccd5e78888ff46167a471f6bff25664a34311c5cb8d6c1b1e7ac2ab0e6676d594734e8f7013b33806868c151316d0cf762a50066c596244fd70b4cb021369aae432e174da502a806e7a8ab13dad1f1b83ac73c0e9e39648630923cbd5726225f17cc0d15afadb7d2c2952b6e092ffc53dcff2914bfddedd043bbdf9c6f6b6b5a6269c5bd423294b9deac4f268eaadb
e=0x3
c=0xb2ab05c888ab53d16f8f7cd39706a15e51618866d03e603d67a270fa83b16072a35b5206da11423e4cd9975b4c03c9ee0d78a300df1b25f7b69708b19da1a5a570c824b2272b163de25b6c2f358337e44ba73741af708ad0b8d1d7fa41e24344ded8c6139644d84dc810b38450454af3e375f68298029b7ce7859f189cdae6cfaf166e58a22fe5a751414440bc6bce5ba580fd210c4d37b97d8f5052a69d31b275c53b7d61c87d8fc06dc713e1c1ce05d7d0aec710eba2c1de6151c84d7bc3131424344b90e3f8947322ef1a57dd3a459424dd31f65ff96f5b8130dfd33111c59f3fc3a754e6f98a836b4fc6d21aa74e676f556aaa5a703eabe097140ec9d98

i = 0
while True:
    if iroot(c + i * n, 3)[1] == True:
        print "Success!"
        print iroot(c + i * n, 3)
```

```
        break
    i += 1
    print i
```

公约数攻击，给出n1和n2

直接寻找公约数

```
from libnum import gcd
print "p -> {}".format(gcd(n1, n2))
print "q1 -> {}".format(n1 / gcd(n1, n2))
print "q2 -> {}".format(n2 / gcd(n1, n2))
```

共模攻击，同一个n，给出多组c,e

其实就是数论知识。

首先，我们需要`e1`和`e2`互质。用gcd判断一下就好。

```
from libnum import gcd
gcd(e1,e2)
```

如果说它们确实互质！那问题就好办了，就可以使用接下来要讨论的*共模攻击*了！

先说一条数论定理，如果`e1`和`e2`互质，那么存在`s1`，`s2`，使得下式成立：

```
e1*s1+e2*s2 = 1
```

`s1`和`s2`都是整数，并且一正一负。我们可以用拓展欧几里得算法的到一组解。算法代码如下：

```
# from libnum import gcd
q=gcd(a,b)
def ext_euclid(a, b):
    if b == 0:
        return 1, 0, a
    else:
        x, y, q = ext_euclid(b, a % b)
        x, y = y, (x - (a // b) * y)
        return x, y, q
```

根据RSA的定义，可以得到下式：

```
c1 = m^e1%n
c2 = m^e2%n
```

所以

```
(c1^s1*c2^s2)%n = ((m^e1%n)^s1*(m^e2%n)^s2)%n
```

模运算简化后得到：

```
(c1^s1*c2^s2)%n = ((m^e1)^s1*(m^e2)^s2)%n
                = (m^(e1^s1+e2^s2))%n
```

又因为`e1*s1+e2*s2 = 1`，所以：

```
c1^s1*c2^s2)%n = (m^(1))%n
```

也就是：

```
c1^s1*c2^s2 = m
```

我们就得到了`m`。这里需要注意一点，就是模运算中的**负数次幂**，跟常规方法不同。需要先算出模反元素，再求幂（幂取正数即可）。

```
#coding=utf-8
from libnum import xgcd
def modinv(a, m):
  x, y, g = xgcd(a, m)
  if g != 1:
    raise Exception('modular inverse does not exist')
  else:
    return x % m
def main():
  n = eval(raw_input("input n:"))
  c1 = eval(raw_input("input c1:"))
  c2 = eval(raw_input("input c2:"))
  e1 = eval(raw_input("input e1:"))
  e2 = eval(raw_input("input e2:"))

  s = xgcd(e1, e2)
  s1 = s[1]
  s2 = s[2]
  if s1<0:
    s1 = - s1
    c1 = modinv(c1, n)
```

```
  elif s2<0:
    s2 = - s2
    c2 = modinv(c2, n)
  m = (pow(c1,s1)*pow(c2,s2))%n
  print m
if __name__ == '__main__':
  main()
```

广播攻击，e相同，给出多组n和c

next-rsa里给出的e特别小，我以为可以直接低指数攻击，然而我想多了。

广播攻击的背景是有人用不同的模`N`加密了同一条消息发送给好几个人，而这些消息全部被截获，广播攻击解题方法的重点在于*中国剩余定理*。

用现代数学的语言来说明的话，中国剩余定理给出了以下的一元线性同余方程组：

$$(S):\quad \begin{cases} x \equiv a_1 \pmod{m_1} \\ x \equiv a_2 \pmod{m_2} \\ \quad\vdots \\ x \equiv a_n \pmod{m_n} \end{cases}$$

有解的判定条件，并用构造法给出了在有解情况下解的具体形式。

中国剩余定理说明：假设整数$m_1, m_2, \ldots, m_n$其中任两数互质，则对任意的整数：$a_1, a_2, \ldots, a_n$，方程组(S)有解，并且通解可以用如下方式构造得到：

1. 设$M = m_1 \times m_2 \times \cdots \times m_n = \prod_{i=1}^{n} m_i$是整数$m_1, m_2, \ldots, m_n$的乘积，并设$M_i = M/m_i$，$\forall i \in \{1, 2, \cdots, n\}$，即$M_i$是除了$m_i$以外的$n-1$个整数的乘积。
2. 设$t_i = M_i^{-1}$为M_i模m_i的数论倒数：$t_i M_i \equiv 1 \pmod{m_i}$，$\forall i \in \{1, 2, \cdots, n\}$.
3. 方程组(S)的通解形式为：
   $x = a_1 t_1 M_1 + a_2 t_2 M_2 + \cdots + a_n t_n M_n + kM = kM + \sum_{i=1}^{n} a_i t_i M_i, \quad k \in \mathbb{Z}.$ 在模M的意义下，方程组(S)只有一个解：$x = \sum_{i=1}^{n} a_i t_i M_i$.

基本上看懂这一段Wiki介绍就知道该怎么解决广播攻击了。

```
import gmpy2
import binascii
#e
e=0x3
#n
n1=0x43d819a4caf16806e1c540fd7c0e51a96a6dfdbe68735a5fd99a468825e5ee55c4087106f7d1f91e10d50df1f2082f0f32bb82f398134b0b8758353bdabc5ba2817f4e6e0786e176686b2e75a7c47d073f346d6adb2684a9d28b658dddc75b3c5d10a22a3e85c6c12549d0ce7577e79a068405d3904f3f6b9cc408c4cd8595bf67fe672474e0b94dc99072caaa4f866fc6c3feddc74f10d6a0fb31864f52adef71649684f1a72c910ec5ca7909cc10aef85d43a57ec91f096a2d4794299e967fcd5add6e9cfb5baf7751387e24b93dbc1f37315ce573dc063ecddd4ae6fb9127307cfc80a037e7ff5c40a5f7590c8b2f5bd06dd392fbc51e5d059cffbcb85555
```

```
n2=0x60d175fdb0a96eca160fb0cbf8bad1a14dd680d353a7b3bc77e620437da70fd9153f76
09efde652b825c4ae7f25decf14a3c8240ea8c5892003f1430cc88b0ded9dae12ebffc6b236
32ac530ac4ae23fbffb7cfe431ff3d802f5a54ab76257a86aeec1cf47d482fec970fc27c5b3
76fbf2cf993270bba9b78174395de3346d4e221d1eafdb8eecc8edb953d1ccaa5fc250aed83
b3a458f9e9d947c4b01a6e72ce4fee37e77faaf5597d780ad5f0a7623edb08ce76264f72c3f
f17afc932f5812b10692bcc941a18b6f3904ca31d038baf3fc1968d1cc0588a656d0c53cd5c
89cedba8a5230956af2170554d27f524c2027adce84fd4d0e018dc88ca4d5d26867
n3=0x280f992dd63fcabdcb739f52c5ed1887e720cbfe73153adf5405819396b28cb54423d1
96600cce76c8554cd963281fc4b153e3b257e96d091e5d99567dd1fa9ace52511ace4da407f
5269e71b1b13822316d751e788dc935d63916075530d7fb89cbec9b02c01aef19c39b4ecaa1
f7fe2faf990aa938eb89730eda30558e669da5459ed96f1463a983443187359c07fba8e9702
4452087b410c9ac1e39ed1c74f380fd29ebdd28618d60c36e6973fc87c066cae05e9e270b5a
c25ea5ca0bac5948de0263d8cc89d91c4b574202e71811d0ddf1ed23c1bc35f3a042aac6a0b
df32d37dede3536f70c257aafb4cfbe3370cd7b4187c023c35671de3888a1ed1303
#c
c1=0x5517bdd6996b54aa72c2a9f1eec2d364fc71880ed1fa8630703a3c38035060b675a144
e78ccb1b88fa49bad2ed0c6d5ad0024d4bb18e7d87f3509b0dbf238a0d1ff33f48ffc99c1bd
f2f2547a193e7ab66eec562a7bc3f9521f70d453ff6d1fdb24de40b3f621ca6be6606440d09
d0f302d5806e7cebc9b612522f181baa43373d6827ffd794916ffcc205147c8d88a59d2fce4
bbcdfd6a4934fb72d5f74be79a1bd64b4305865c9d20eb96d8bd7976440a4bc326fdb5b9a04
bac3762a664346a175f1029f448bb421506f3dfeb75d6531f89f0b92a7e66e295ede5928ec8
301a202d5c9fd528cda84190c2b47f423af1a59c63ae6253d1903c83ae158f9b42
c2=0x3288e3ea8c74fd004e14b66a55acdcbcb2e9bd834b0f543514e06198045632b664dac3
cf8578cde236a16bef4a1246de692ec6a61ce507a220fa04e09044632787ba42b856cb13be6
e905c20b493004822888d3c44c6fc367c7af0287f1683f08baae5bb650902067908e93246af
3954d62437aa14248529fd07c8902b9403920b6550f12d1c398881cd7fc8b5f096f38c33df2
1887bfe989fb011a9deade2370d90347510b76f1f3e3dedf09c148675ea8919878c8ac18825
3b78886d906cd1f3aee5484d6d13fb4bbad233f670f825fa618adbf0705ed4e31b60957f5c2
8cfd1febd13370630a6c94990e341d38918a9c1faa614fd14cdd41b7bc8461f2f0c
c3=0xb0c5ee1ac47c671c918726287e70239147a0357a9638851244785d552f307ed6a04939
8d3e6f8ed373b3696cfbd0bce1ba88d152f48d4cea82cd5dafd50b9843e3fa2155ec7dd4c99
6edde630987806202e45821ad6622935393cd996968fc5e251aa3539ed593fe893b15d21ecb
e6893eba7fe77b9be935ca0aeaf2ec53df7c7086349eb12792aefb7d34c31c18f3cd7fb68e8
a432652ef76096096e1a5d7ace90a282facf2d2760e6b5d98f0c70b23a6db654d10085be9dc
c670625646a153b52c6c710efe8eb876289870bdd69cb7b45813e4fcfce815d191838926e9d
60dd58be73565cff0e10f4e80122e077a5ee720caedc1617bf6a0bb072bbd2dab0


M = n1*n2*n3
m1 = M/n1
m2 = M/n2
m3 = M/n3

t1 = c1*(m1)*gmpy2.invert(m1,n1)
t2 = c2*(m2)*gmpy2.invert(m2,n2)
t3 = c3*(m3)*gmpy2.invert(m3,n3)

x = (t1+t2+t3) % M # chinese reminder theorem
```

```
print "-------------------"
print  gmpy2.iroot(x,e)
m, exact = gmpy2.iroot(x,e) # recover m
if exact:
    #print binascii.unhexlify(gmpy2.digits(m,16))
    print m
```