第九届（2024 年）
全国高校密码数学挑战赛
参赛论文

RSA 密码系统的特定密钥泄露攻击

战队名称 就是能解队

学　　校 南开大学

学　　院 网络空间安全学院

专　　业 信息安全、密码科学与技术

参赛选手 刘陈思含 苏长昊 陆皓喆

指导教师 汪定

二零二四年六月

摘　要

RSA 加密算法是一种非对称加密算法，被广泛应用于网络通信、数据传输以及信息安全领域。其本质上基于数论中的大数分解难题。其中，部分私钥泄露攻击是一项很重要的攻击，私钥一般较难获取，但是也往往可以由通讯时间以及测信道攻击等等的攻击方式，得到有关的信息。因此，我们需要用到部分私钥攻击这种攻击方式进行相关的攻击尝试与分析。

对于赛题而言，所给的十二道题均是已知模数 n、公钥 e、密文 c 以及私钥 d 的部分信息泄露，于私钥 d 的信息而言，或是取值范围，或是高位泄露，或是低位泄露，或是汉明重量，或是与$\varphi(N)$的关系已知。针对前三题，我们主要利用 wiener 攻击以及在其基础上扩展的 Boneh-Durfee 攻击。针对后九题，我们主要使用在 LLL 算法寻找最小根后，再用基于格的 coppersmith 方法进行攻击。最后对所求结果分析，得到关于随机数发生器的信息。

最终得知，在私钥 d 泄露部分比特位或其具体取值与部分已知信息有关时，可以通过数学推导对已知信息进行处理，采取构造多变元模方程或整系数方程的方式对私钥 d 进行攻击，求取方程小值解后利用获得的 d 或分解的 p、q 恢复明文。

关键词：RSA 部分私钥泄露，Boneh-Durfee 攻击，基于格的 coppersmith 攻击

目 录

第一章　引言

RSA 加密算法是一种非对称加密算法，被广泛应用于网络通信、数据传输以及信息安全领域。其本质上基于数论中的大数分解难题，利用了大质数的乘积很容易计算，但从乘积中找出原始质因数却非常困难的特性，使得 RSA 算法在实际应用中具有很高的安全性，有效地保护数据的机密性和完整性。其中，部分私钥泄露攻击是一项很重要的攻击，我们往往可以通过网络抓包，获取明文，密文，以及公钥等等的信息，私钥一般较难获取，但是也往往可以由通讯时间以及测信道攻击等等的攻击方式，得到有关的信息。因此，我们往往需要用到部分私钥攻击这种攻击方式进行相关的攻击尝试与分析。

对于赛题而言，所给的十二道题均是已知模数 n、公钥 e、密文 c 以及私钥 d 的部分信息泄露，于私钥 d 的信息而言，或是取值范围，或是高位泄露，或是低位泄露，或是汉明重量，或是与$\varphi(N)$的关系已知，针对这些信息，我们主要使用的方法有：

- Coppersmith 攻击：令 B 表示利用 Coppersmith 方法求解模方程

  $$f(x_1,\dots,x_l)\equiv 0 mod b$$

  小根时构造的格基矩阵。满足

  $$\frac{\det(\mathbf{B}')}{\det(\mathbf{B})}<b^m$$

  称$\hat{g}(x_1,\dots,x_l)$为有益多项式。Coppersmith 方法的基本思路是将模多项式方程转化为整数上的多项式方程。

- LLL 算法：格可以视作 n 维欧式空间R^n上的一个离散加法子群，定义

  $$det(\mathcal{A})=\sqrt{det(\mathrm{B}(\mathcal{A})^{\top}\mathrm{B}(\mathcal{A}))}$$

  为格$\mathcal{A}$的行列式。Lenstra, Lenstra, Lovász 三人提出的 LLL 算法可以在多项式时间内找到格的短向量，对于 n 维格$\mathcal{A}$，利用 LLL 算法可以得到约化基向量，遵循以下不等式：

  $$\| \mathrm{v}_t \| \leq 2^{\frac{n(n-1)}{4(n+1-t)}} det(\mathcal{L})^{\frac{t}{(n+1-t)}}$$

以下是每题的解题结果：

第一题所求得明文 m 为
721156075061547613364110141099856850563950957473214919673494232477495653879136119681837293285641323913019261856399366962867926733241410500465446031610582623751314859721369608219089460972326894980564507282258352507478729034946935770 37

对应字符串为：Mathematics is the queen of sciences, and arithmetic [number theory] is the queen of mathematics.

第二题所求得明文 m 为：

467953840882317181499103662991214298422064486570574390714467505707261187314252509296933072417732732809103029658160034371

对应字符串为: Cryptography is typically bypassed,not penetrated.

第三题未能求得正确的明文 m，但寻找到了私钥 d 的近似值为 193914681430486465632183530546598936683720682945402197337772960150041799264105917715496687，后续可利用这一近似值进一步尝试攻击

第四题未能求得正确的明文 m，但寻找到了私钥 d 的近似值为 133827283030570403645842687100884536417541672291610202010477861050088020623058760688581352048252729967461294866786600925 3，后续可利用这一近似值进一步尝试攻击

第五题未能求得正确的明文 m，但寻找到了私钥 d 的近似值为 905968769074465236458895157906627664794778079189447975955579653645922818075385404622930251458073034407017853892415022301795027296716592687429818653745655 3，后续可利用这一近似值进一步尝试攻击

第六题所求得明文 m 为 411708110548976412778536399221192829597879130418779704340440446197846122866994974250241237132115958563099708213990983544262962354754778176004782101155215298442496870041733756567944507725673757021127337172975026812709705824528341829542021957086677416458885678605362243282371896905，转换为字符串为 If people do not believe that mathematics is simple, it is only because they do not realize how complicated life is.

第七题所求得明文 m 为 230913323188643024057354978541995677532283168674841160289183905564513987191160091845749079，具体字符串为 Wir müssen wissen, Wir werden wissen.

第八题所求得明文 m 为 192570830640220274585063503419629687903707052610041097928280510574598878104480270754462613796569918908818368893186928242512483444388677719466238488340104039379084134332969994290293840 转换为字符串为 Plato is my friend, Aristotle is my friend, but my greatest friend is truth.

第九题所求得明文 m 为 296267822835550277958334573237828196460580610559986269272317446980320467354458194682415976970656098509606755024351835977686637261523083927977866582051597064139413447088244638993075223273917789633213926216056075825104748462415991091464079215197153377585439659703407527700696923535573051407358620 98，其字符串形式为：Resolving the composite numbers into their prime factors is known to be one of the most important and useful in arithmetic.

第十题所求得明文 m 为

820307570753893249794408475962305073085942139245561017669 其字符串形式为：

Eureka, I have found it!

第十一题所求得明文 m 为

871459330528493685900356342244447392885109122774019102948898640050654591861594334271960704663752901134803717028296430848471791111128678516670233579393144431497657245644265754833188314998240383835607271327755688562060388816458780444461498953491390155933696 73 对应字符串：I have discovered a truly marvelous proof of this, which however the margin is not large enough to contain.

第十二题未能求得正确的明文 m，但找到私钥 d 的近似值为
910092353521253284372505374155384584387022788971302134985765940679051744612213073906174156820918624766614266796734129771220437211599443395753058923116656738136028231369028485794933987047092183958068113565350070069639238339143660566536172783294925460908849242737708306614612297322160738195514331321626774961 32，后续可利用这一近似值进一步尝试攻击

第二章　解题过程

2.1 第一题：wiener 攻击

1990 年，Wiener 提出如果解密指数$d < N^{0.25}$,那么就能够在多项式时间内分解模数 N,从而恢复完整的私钥并攻破 RSA 密码系统,整个攻击依赖连分式有理逼近的性质。

2.1.1 代码实现

2.1.1.1 根据参数特征确定攻击类型

我们观察到 d 小于 n 的 1/4 次，于是查阅资料，我们使用 wiener attack 来对其发起攻击。

```
def factor_rsa_wiener(N, e):
    N = Integer(N)
    e = Integer(e)
    cf = (e / N).continued_fraction().convergents()
    for f in cf:
        k = f.numer()
        d = f.denom()
        if k == 0:
            continue
        phi_N = ((e * d) - 1) / k
        b = -(N - phi_N + 1)
        dis = b ^ 2 - 4 * N
        if dis.sign() == 1:
            dis_sqrt = sqrt(dis)
            p = (-b + dis_sqrt) / 2
            q = (-b - dis_sqrt) / 2
            if p.is_integer() and q.is_integer() and (p * q) % N == 0:
                p = p % N
                q = q % N
                if p > q:
                    return (p, q)
                else:
                    return (q, p)

    factor_rsa_wiener(n,e)
```

2.1.1.2 获得 d 回代

求得 n 的欧拉函数，即可获得 d，我们往回代入，验证 d 的 bit 位是否在规定的范围内，发现 d 的 bit 位刚好是 250 位，符合题目条件。

```
from gmpy2 import *
n=p*q
phi=(p-1)*(q-1)
d=invert(e,phi)
m=pow(c,d,n)
    print(m)
```

2.1.1.3 还原明文

根据求得的明文

m=721156075061547613364110141099856850563950957473214919673494232477495653879136119681837293285641323913019261856399366962867926733241410500465446031610582623751314859721369608219089460972326894980564507282258352507478729034946935770 37

可以将其转换为字符串形式：Mathematics is the queen of sciences, and arithmetic [number theory] is the queen of mathematics.

2.2 第二题：Boneh-Durfee 攻击

将相关的条件输入后，若在$d < N^{0.25}$内，通常情况下，wiener attack 可以保证攻击有效。但如果在$d < N^{0.25}$到$d < N^{0.292}$之间，那么使用 wiener attack 的时间加长，也可能无法求解。此时，往往需要转向考虑 Boneh-Durfee 的方法。

2.2.1 数学推导

由$ed \equiv 1 mod \varphi(N)$可得$ed = 1 + k\varphi(N)$，即$ed = 1 + k(N - p - q + 1)$，由此可以构造模方程$f(x,y) = 1 + x(A + y) mod\ e$，选定多项式集合:

$$g_{i,k}(x,y) = x^i f^k e^{m-k}$$
$$k = 0,\cdots,m \quad i = 0,\cdots,m-k$$
$$h_{j,k}(x,y) = y^j f^k e^{m-k}$$
$$k = 0,\cdots,m \quad j = 1,\cdots,t$$

可以构造格基矩阵进行求解，除此之外，还可以进行线性化变换，令$1 + xy = u$，方程可以转换为$f(u,x) = u + Ax$，可以构造多项式集合：

$$g_{i,k}(u,x) = xif^k e^{m-k}$$
$$k = 0,\cdots,m, i = 0,\cdots,m-k$$
$$h_{j,k}(x,x,y) = y^j + ke^{m-k}$$
$$j = 0,\cdots,tk = \lfloor\frac{m}{0}\rfloor j,\cdots,m$$

且将其中xy项替换为$u-1$，经过上述变换可以获得更好的格基矩阵优化求解。

2.2.2 代码实现

2.2.2.1 根据参数特征确定攻击类型

我们根据 d 和 n 的大小关系，确定出该题我们应该使用 Boneh -Durfee 的方法来进行攻击。

2.2.2.2 区分向量是否有用

```
def helpful_vectors(BB, modulus):
    nothelpful = 0
    for ii in range(BB.dimensions()[0]):
        if BB[ii,ii] >= modulus:
            nothelpful += 1

    print(nothelpful, "/", BB.dimensions()[0], " vectors are not helpful")
```

```
def remove_unhelpful(BB, monomials, bound, current):
    # end of our recursive function
    if current == -1 or BB.dimensions()[0] <= dimension_min:
        return BB

    # we start by checking from the end
    for ii in range(current, -1, -1):
        # if it is unhelpful:
        if BB[ii, ii] >= bound:
            affected_vectors = 0
            affected_vector_index = 0
            # let's check if it affects other vectors
            for jj in range(ii + 1, BB.dimensions()[0]):
                # if another vector is affected:
                # we increase the count
                if BB[jj, ii] != 0:
                    affected_vectors += 1
                    affected_vector_index = jj
```

```
            # level:0
            # if no other vectors end up affected
            # we remove it
            if affected_vectors == 0:
                print("* removing unhelpful vector", ii)
                BB = BB.delete_columns([ii])
                BB = BB.delete_rows([ii])
                monomials.pop(ii)
                BB = remove_unhelpful(BB, monomials, bound, ii-1)
                return BB

            # level:1
            # if just one was affected we check
            # if it is affecting someone else
            elif affected_vectors == 1:
                affected_deeper = True
                for kk in range(affected_vector_index + 1, BB.dimensions()[0]):
                    # if it is affecting even one vector
                    # we give up on this one
                    if BB[kk, affected_vector_index] != 0:
                        affected_deeper = False
                # remove both it if no other vector was affected and
                # this helpful vector is not helpful enough
                # compared to our unhelpful one
                if affected_deeper and abs(bound - BB[affected_vector_index,
affected_vector_index]) < abs(bound - BB[ii, ii]):
                    print("* removing unhelpful vectors", ii, "and",
affected_vector_index)
                    BB = BB.delete_columns([affected_vector_index, ii])
                    BB = BB.delete_rows([affected_vector_index, ii])
                    monomials.pop(affected_vector_index)
                    monomials.pop(ii)
                    BB = remove_unhelpful(BB, monomials, bound, ii-1)
                    return BB
    # nothing happened
    return BB
```

2.2.2.3 求解多项式方程

```
def boneh_durfee(pol, modulus, mm, tt, XX, YY):
    # 替换 (Herrman 和 May)
    PR.<u, x, y> = PolynomialRing(ZZ)
    Q = PR.quotient(x*y + 1 - u)
    polZ = Q(pol).lift()
    UU = XX*YY + 1

    # x-shifts
    gg = []
    for kk in range(mm + 1):
```

```
        for ii in range(mm - kk + 1):
            xshift = x^ii * modulus^(mm - kk) * polZ(u, x, y)^kk
            gg.append(xshift)
    gg.sort()

    # 收集单项式
    monomials = sorted({monomial for polynomial in gg for monomial in
polynomial.monomials()})

    # y-shifts (选自 Herrman 和 May)
    for jj in range(1, tt + 1):
        for kk in range(floor(mm/tt) * jj, mm + 1):
            yshift = y^jj * polZ(u, x, y)^kk * modulus^(mm - kk)
            gg.append(Q(yshift).lift())

    for jj in range(1, tt + 1):
        for kk in range(floor(mm/tt) * jj, mm + 1):
            monomials.append(u^kk * y^jj)

    # 构建矩阵 B
    nn = len(monomials)
    BB = Matrix(ZZ, nn)
    for ii in range(nn):
        BB[ii, 0] = gg[ii](0, 0, 0)
        for jj in range(1, ii + 1):
            if monomials[jj] in gg[ii].monomials():
                BB[ii, jj] = gg[ii].monomial_coefficient(monomials[jj]) *
monomials[jj](UU, XX, YY)

    # 可选步骤：减少矩阵
    if helpful_only:
        BB = remove_unhelpful(BB, monomials, modulus^mm, nn - 1)
        nn = BB.dimensions()[0]
        if nn == 0:
            print("failure")
            return 0, 0

    # 检查行列式是否正确
    det = BB.det()
    bound = modulus^(mm * nn)
    if det >= bound:
        print("We do not have det < bound. Solutions might not be found.")
        print("Try with higher m and t.")
        if strict:
            return -1, -1

    # 显示矩阵基
    if debug:
        matrix_overview(BB, modulus^mm)

    # LLL 算法优化基
    if debug:
```

```
        print("optimizing basis of the lattice via LLL, this can take a long
time")
    BB = BB.LLL()
    if debug:
        print("LLL is done!")

    # 寻找独立向量
    found_polynomials = False
    for pol1_idx in range(nn - 1):
        for pol2_idx in range(pol1_idx + 1, nn):
            PR.<w, z> = PolynomialRing(ZZ)
            pol1 = sum(monomials[jj](w*z + 1, w, z) * BB[pol1_idx, jj] /
monomials[jj](UU, XX, YY) for jj in range(nn))
            pol2 = sum(monomials[jj](w*z + 1, w, z) * BB[pol2_idx, jj] /
monomials[jj](UU, XX, YY) for jj in range(nn))
            rr = pol1.resultant(pol2)

            if not rr.is_zero() and rr.monomials() != [1]:
                print("found them, using vectors", pol1_idx, "and", pol2_idx)
                found_polynomials = True
                break
        if found_polynomials:
            break

    if not found_polynomials:
        print("no independent vectors could be found. This should very rarely
happen...")
        return 0, 0

    rr = rr(q, q)
    soly = rr.roots()
    if len(soly) == 0:
        print("Your prediction (delta) is too small")
        return 0, 0
    soly = soly[0][0]
    ss = pol1(q, soly)
    solx = ss.roots()[0][0]
    return solx, soly
```

2.2.2.4 利用小值解求解明文

针对上述步骤中获得的小值解，

```
if solx > 0:
    print("=== solution found ===")
    if False:
        print("x:", solx)
        print("y:", soly)

    d = int(pol(solx, soly) / e)
```

```
    print("private key found:", d)
else:
    print("=== no solution was found ===")
```

可以获得公钥 d 的值，进而可以获取明文

m=467953840882317181499103662991214298422064486570574390714467505707261187314252509296933072417732732809103029658160034371

对应字符串为: Cryptography is typically bypassed,not penetrated.

2.3 第三题：Boneh-Durfee 攻击

此题思路同第二题，在其基础上进行拓展。目前已知在私钥 d 完全未泄露的情况下能对其成功攻击的界限为$d < N^{0.292}$,本题$d \approx N^{0.292}$,仅通过 Boneh-Durfee 攻击难以破解 d，理论可行性与计算资源的可行性均面临较大挑战。为此，可通过猜测 d 的部分比特位的方法，尝试在可行的穷举范围内，穷举较少比特位的各种可能进而实现攻击。除此之外，也可以沿用第二题的方法调整参数进行试错，尝试进一步获得更有效的信息。

2.3.1 数学推导

根据 Yoshinori Aono 的成果，当β约为 0.290，δ约为 0.246 时，需要已知约 40 比特位，选取参数 m=10，时长约为 3 小时 15 分，当β约为 0.295，δ约为 0.246 时，需要已知约 50 比特位，选取参数 m=10，时长约为 4 小时 2 分，综合考虑穷举的次数以及每次求解双变元模方程所需的时间，通过这样的攻击方式本题难以在有限时间内给出解答。在参数 m 取值较小时，我们无法得到符合攻击条件的格基矩阵，但能够在较短时间内获得 d 的近似值，可以进一步利用该近似值尝试进行攻击。

2.3.2 代码实现

本题使用的代码与第二题相同，当参数 m 取值较小时，获得近似值d'后将方程中的 d 用$d' + d_0$替代，尝试求解与d_0有关的方程即可。如果采用穷举的方法，可以尝试穷举 40 或 50 比特位的最低有效位，令$d = d_1 \cdot 2^r + d_0$，修改原本的方程进行求解，并将获得的 d 的近似值作为新的信息进一步尝试攻击。

2.4 第四题：汉明重量

一个码字中非零分量的个数即为汉明重量，简单来说在我们的 0-1 码中即 1 的个数。而赛题中提到 d 的汉明重量较轻，可以理解为 0 的个数多而 1 的个数少。同时最高 310 位

比特 1 的个数只可能是 0,1,2,3,4,5，代表着此段消息中具有较长段存在连续的 0。因此可将此题转换为已知私钥中间部分比特位泄露，泄露部分值为 0，求解私钥 d 的问题。对于私钥中间部分比特位泄露的情况，记 $d = d_1 \cdot 2^{r_1} + d_2 \cdot 2^{r_2} + d_3, 0 < r_2 < r_1 < l_d, 0 \leqslant d_1 < 2^{l_d - r_1}, 0 \leqslant d_2 < 2^{r_1 - r_2}, 0 \leqslant d_3 < 2^{r_2}$，$\delta_1 = \log_N 2^{r_1}, \delta_2 = \log_N 2^{r_2}$，定义$A = e \cdot d_2 \cdot 2^{r_2} - 1, W = e \cdot 2^{r_1} = N^{\alpha+\delta_1}$，可以构造模方程$f_{MBs1}(x, y, z) = A + ex - Ny + yz \equiv 0 (\mathrm{mod} W)$进行求解，可行攻击范围为$\beta \leqslant (\delta_1 - \delta_2) + \frac{7}{6} - \frac{1}{3}\sqrt{6[\alpha + (\delta_1 - \delta_2)] + 1} - \varepsilon'$

2.4.1 数学推导

就本题而言，对于 RSA 密码的私钥 d(比特位数为l_d),，定义

$$l_{max} = \max\{ l \in \mathbb{Z}^+ \mid d \text{ 的 } l_d \text{ 个比特位中,存在连续 } l \text{ 个比特位均为 } 0 \}$$

对任意小的 ε'>0,存在大整数N_0 ,使得当$N > N_0$时,只要

$$\beta \leqslant \frac{l_{max}}{\log_2 N} + \frac{7}{6} - \frac{1}{3}\sqrt{6(\alpha + \frac{l_{max}}{\log_2 N}) + 1} - \varepsilon'$$

由于 $d_2 = 0$，$\delta_1 - \delta_2 = \frac{r_1 - r_2}{\log_2 N}$，对$(r_1, r_2)$进行穷搜，可以找到一组$(r_1, r_2)$满足$r_1 - r_2 = l_{max}$，且因为$C_{l_d - 1}^2 < (l_d - 1)^2 < (\lceil \log_2 N \rceil)^2$，可以在多项式时间内实现对 d 的攻击。据此，模方程可以修改为$f_{MBs1}(x, y, z) = ex - Ny + yz - 1 \equiv 0 (\mathrm{mod} W)$，并有相应的多项式集合如下：

$$g = x^{t_1} y^{t_2} (f_{MBS1})^{t_3} W^{m - t_3}$$
$$t \in \{0,1,\cdots,m\} \quad t_1, t_2, t_3 \in z^+ \cup \{0\} \quad t_1 + t_2 + t_3 = t$$
$$h = z^i \times^{t_1} (fMBs_1)^{t_3} W^{m - t_3}$$
$$i \in \{1,\cdots,t\} \quad t_1, t_3 \in z^+ \cup \{0\}, \quad t_1 + t_3 = t$$

利用上述多项式集合可以构造格基矩阵进行求解。

2.4.2 代码实现

2.4.2.1 依据参数范围决定攻击方式

本题可行攻击范围为$\beta \leqslant (\delta_1 - \delta_2) + \frac{7}{6} - \frac{1}{3}\sqrt{6[\alpha + (\delta_1 - \delta_2)] + 1} - \varepsilon'$，$\delta_1 - \delta_2 = \frac{r_1 - r_2}{\log_2 N}$，根据题目已知信息可以确定$\beta$约为 0.390625，$\alpha$约为 1，进而可以确定$\delta_1 - \delta_2$至少为 0.109 方可实现攻击，因此我们对$r_1 - r_2$为 112 位的所有可能情况进行穷搜。首先确定$r_1$的取值，$r_2$为$r_1 - 112$，利用每次穷搜得到信息作为已知信息，重复进行攻击直至破解成功

2.4.2.2 构造模方程，确定小值解范围

在整数环 ZZ 上定义三个多项式变量 x,y,z,利用这些变量构造双变元模方程$f(x,y,z)$，小值解的上界分别为$X = N^{\delta_2}, Y = 2N^{\alpha+\beta-1}, Z = 3N^{0.5}$,根据上一步确定的参数范围可以进一步确定其具体数值。

2.4.2.3 构造多项式集合

```
    shifts = []
    for t1 in range(m + 1):
        for t2 in range(m + 1):
            for t3 in range(m + 1):
                g = x ** t1 * y ** t2 * f ** t3 * W ** (m - t3)
                if(0<=t1+t2+t3<=m):
                   shifts.append(g)

    for i in range(1, tau+1):
        for t1 in range(m+1):
            for t3 in range(m+1):
                h = z ** i * x ** t1 * f ** t3 * W ** (m - t3)
                if(0<=t1+t3<=m):
                      shifts.append(h)
```

2.4.2.4 构造格基矩阵并约减

初始化一个集合，存储所有多项式的单项式，遍历每个多项式将其中的单项式添加到集合中，初始化一个整数矩阵 L，遍历每一个单项式和多项式计算格矩阵中的对应项，将单项式集合中的每个元素转换成原始的多项式环。具体代码如下：

```
if pr.ngens() > 1:
        pr_ = pr.change_ring(ZZ, order=order)
        shifts = [pr_(shift) for shift in shifts]

    monomials = set()
    for shift in shifts:
        monomials.update(shift.monomials())

    shifts.sort(reverse=sort_shifts_reverse)
    monomials = sorted(monomials, reverse=sort_monomials_reverse)
    L = matrix(ZZ, len(shifts), len(monomials))
    for row, shift in enumerate(shifts):
        for col, monomial in enumerate(monomials):
            L[row, col] = shift.monomial_coefficient(monomial) *
monomial(*bounds)

    monomials = [pr(monomial) for monomial in monomials]

       return L, monomials
```

并使用 LLL 算法对格进行约简，以便更有效地寻找小根。

2.4.2.5 重构多项式方程

遍历格基矩阵的每一列，如果当前格基元素不为零，将其加入到多项式构建中，如果当前多项式可以被原始多项式整除，则进行除法，并尝试通过最大公约数简化当前多项式与已经重构的多项式，跳过那些是常数的多项式，将重构后的多项式添加到列表中。

2.4.2.6 使用 Groebner 基方法求根

计算 Groebner 基，如果 Groebner 基的大小与生成元数量相等，说明每个变量可能都有对应的多项式，可以进一步寻找根，遍历 Groebner 基中的每个多项式，如果是一元多项式，使用单变量根查找方法寻找并合并根，如果找到的根数量与多项式环的生成元数量相等，说明已经为每个变量找到了解，生成这些根并结束函数。

2.4.2.7 恢复明文 m

对于使用 Groebner 基方法求解的小值解$(x_0, y_0, z_0) = (d_3, k, p + q - 1)$，利用$p + q - 1$可以确定$\varphi(N)$的取值，从而确定 d 的取值，继而使用函数执行模幂运算，根据公式$m = c^d mod\ n$, 求解 m，再依据题目提供的编码方式将数值形式的明文 m 转换为字符串形式。

2.5 第五题：高位&低位同时泄露

此问题也即私钥 d 的 n 个未知块泄露中 n=2 的情况，对于部分高位和部分低位已知的私钥 d，STK 格攻击是目前最佳的攻击方法。解决这类问题的核心是构造多变元模方程，并利用线性代换等改进格基构造的技巧，优化格基矩阵的构造。定义$d = d_3 M_3 + d_2 M_2 + d_1$，$d_3 > N^{\delta_1}, d_2 < N^{\beta - \delta_1 - \delta_2}, d_1 > N^{\delta_2}$分别表示私钥 d 的已知高位，未知中间比特位，已知低位，可以构造模方程，$f_{eM_2}(x, y) = 1 - ed_1 + (\ell_1 + x)(N + y)(\mathrm{mod}\ \ eM_2), f_e(x, y) = 1 + (\ell_1 + x)(N + y)(\mathrm{mod}\ \ e)$

2.5.1 数学推导

对于上面的两个方程，它们拥有共同的小根 $(x_0, y_0) = (\ell - \ell_1, -p - q + 1)$, 并且小根的上界为$X = N^{max(\beta - \delta_1, \beta - 1/2)}, Y = N^{1/2}$，针对本题而言，我们引入定义$\ell_{k,\tau}^{LSBs}(x) := max\left\{0, \left\lceil \frac{x-k}{\tau} \right\rceil\right\}$，构造如下的多项式集合：

$$g_{[u,i]}^{(x)}(x,y) = x^{u-i} f_{eM_2}(x,y)^i (eM_2)^{m-i}$$

$$u = 0,1,\dots,m; i = 0,1,\dots,u$$

$$g_{[u,i]}^{(y)}(x,y) = y^j f_{eM_2}(x,y)^{u-\ell_{k,\tau}^{LSBs}(j)} f_e(x,y)^{\ell_{k,\tau}^{LSBs}(j)} e^{m-u} M_2^{m-(u-\ell_{k,\tau}^{LSBs}(j))}$$

$$u = 0,1,\dots,m; j = 1,2,\dots,k + \lfloor \tau u \rfloor$$

其中$k = 2\delta_1 m$，$\tau = 1 - 2(\beta + \delta_1 - \delta_2)$，引入变量$z = 1 + (\ell_1 + x)y$进行线性化处理，可以得到更好的格基，上述攻击在

$$\delta_1 > \frac{\beta - \delta_2 - 1 + \sqrt{-3(\beta-\delta_2)^2 + 4(\beta-\delta_2) + 2\beta - 1}}{2}, \frac{1}{2} - \delta_1 + \delta_2 \le \beta \le \frac{1}{2} + \delta_2$$

的范围内是有效的。

2.5.2 代码实现

2.5.2.1 依据参数范围决定攻击方式

STK 格攻击中提到了三种有效的攻击范围，根据本题提供的参数信息，我们可以确定β约为 0.5，δ_1约为 0.25，δ_2约为 0.171，满足

$$\delta_1 > \frac{\beta - \delta_2 - 1 + \sqrt{-3(\beta-\delta_2)^2 + 4(\beta-\delta_2) + 2\beta - 1}}{2}, \frac{1}{2} - \delta_1 + \delta_2 \le \beta \le \frac{1}{2} + \delta_2$$

故而采用 STK 格攻击尝试对 RSA 的私钥进行攻击，并选定参数$k = 2\delta_1 m$，$\tau = 1 - 2(\beta + \delta_1 - \delta_2)$

2.5.2.2 构造模方程，确定小值解范围

在整数环 ZZ 上定义三个多项式变量 x,y,利用这些变量构造双变元模方程$f(x,y)$，小值解的上界分别为$X = N^{max(\beta-\delta_1,\beta-1/2)}, Y = N^{1/2}$,根据上一步确定的参数范围可以进一步确定其具体数值。

2.5.2.3 构造多项式集合求解

```
def LSB(i,k,tau):
    temp=math.ceil((i-k)/tau)
    largenumber=max(0,temp)
```

```
return largenumber
shifts = []
    for u in range(m + 1):
        for i in range(u + 1):
                g = x ** (u-i) * fem2 ** i * (e*M2) ** (m-i)
                g = g.subs({L1+x: w})
                shifts.append(g)
    for u in range(m + 1):
        for j in range(1,(math.floor(k+tau*u))):
                p=LSB(j,k,tau)
                g = y**j * fem2 ** (u-p) * fe ** p * e ** (m-u) *
M2 ** (m-u+p)
                g = g.subs({L1+x: w})
                    shifts.append(g)
```

在多项式集合的构造过程中，我们引入了新的变量w,替换$L_1 + x$,对多项式集合进行线性化处理，以期获得更好的格基矩阵。针对上述步骤获得的多项式集合，我们使用与第四题中相同的方法进行格基矩阵的构造，在格基约减后进行多项式的重构并求解小值解$(x_0, y_0) = (\ell - \ell_1, -p - q + 1)$，进而恢复明文 m。

2.6 第六题：低位泄露

Boneh, Durfee, and Frankel 等人首次提出了暴露部分最低有效位的攻击，Blömer 等人对他们的攻击提出了改进，将已知部分最低有效位攻击扩展到了$N^{0.875}$,Takayasu 等人进一步将已知部分最低有效位攻击扩展到了$N^{0.368}$。

除此之外，针对私钥 d 部分低位已知且较小的情况，可以引入新的变量，利用线性化技巧能够实现对 RSA 的攻击。

2.6.1 数学推导

通过对模方程

$$f_{LSB1}(y, z) = Ny - yz - ed_0 + 1 \equiv 0 \bmod (e \cdot 2^r)$$

引入新变元

$$u = yz + ed_0 - 1$$

定义

$$f_{LSB2}(y, z, u) = Ny - u$$

构建新的模方程

$$f_{LSB2}(y, z, u) \equiv 0 (\bmod e \cdot 2^r)$$

求解小值解

$$(y_0, z_0, u_0) := (k, p+q-1, k(p+q-1)+ed_0-1)$$

利用多项式集合:

$$g_{i,j} = y^{t-j}(W)^{m-j} f_{LSB2}^{j} t = 0, \dots, m; j = 0, \dots, t$$

$$h_{i,j} = z^{i}(W)^{m-j} f_{LSB2}^{j} i = 0, \dots, \tau; j = \theta_i, \theta_{i+1}, \dots, m$$

进行格基矩阵的构建进而求解得到小值解。

记$\delta = \log_N 2^r$，$d \le N^{\beta}$，$e = N^{\alpha}$，上述攻击对于$d_0 < N^{\beta-0.5}$，$\beta \le \delta + 1 - \frac{1}{2}\sqrt{2(\alpha+\delta)}$是有效的。

2.6.2 代码实现

2.6.2.1 依据参数范围决定攻击方式

分析题目所给参数可以发现，私钥 d 泄露的低位d_0为 120542853124939，β约为 0.55，δ约为 0.42，α约为 1.01，满足$d_0 < N^{\beta-0.5}$，$\beta \le \delta + 1 - \frac{1}{2}\sqrt{2(\alpha+\delta)}$，由此可以进一步构造格基矩阵进行攻击。

2.6.2.2 构造模方程，确定小值解范围

在整数环 ZZ 上定义三个多项式变量 y,z,u 利用这些变量构造双变元模方程$f(y,z,u)$，小值解的上界分别为$Y = 2N^{\alpha+\beta-1}, Z = 3N^{\frac{1}{2}}, U = 7N^{\alpha+\beta-0.5}$,根据上一步确定的参数范围可以进一步确定其具体数值。

2.6.2.3 构造多项式集合

```
Q = pr.quotient(y*z + A - u)
    polZ = Q(f).lift()

    shifts = []
    for t in range(m + 1):
        for j in range(t + 1):
            g = y ** (t - j) * f ** j * W ** (m-j)
            g = Q(g).lift()
            shifts.append(g)

    for i in range(1, tau + 1):
        for j in range(theati, m+1):
            h = z ** i * f ** j * W ** (m-j)
            h = Q(h).lift()
```

```
                shifts.append(h)
```

与第五题相似，第六题也使用了引入新变量进行线性化操作，在多项式构建时将$y*z$替换为$u-A$,可以增加有益多项式，减少非有益多项式。

2.6.2.4 求解明文 m

根据所求得的小值解$(y_0,z_0,u_0):=(k,p+q-1,k(p+q-1)+ed_0-1)$，利用$z_0$可以获得$\varphi(N)$的具体数字，进而求出 d 的值，根据公式$m=c^d mod\, n$, 求解 m 为
411708110548976412778536399221192829597879130418779704340440446197846122866994974250241237132115958563099708213990983544262962354754778176004782101155215298442496870041733756567944507725673757021127337172975026812709705824528341829542021957086677416458885678605362243282371896905，转换为字符串为 If people do not believe that mathematics is simple, it is only because they do not realize how complicated life is.

2.7 第七题：d 与$\varphi(N)$接近

当我们已知 d 的部分低位信息时，可以利用已知信息构造模方程，而针对低位信息未知，但 d 与$\varphi(N)$接近的情况，可以尝试使用包含$\varphi(N)$的表达式替换 d，进而构造模方程。针对此题，对于本题，我们注意到$\varphi(N)$与 d 的差值t取值范围在2^{267}与2^{268}之间，我们可以将模方程转换为与 Boneh-Durfee 攻击所使用的模方程类似的形式,进而可以使用与 Boneh-Durfee 中相同的多项式集合构建格基矩阵进行求解。

2.7.1 数学推导

$$ed\equiv 1\ mod\varphi(N)$$
$$e(\varphi(N)-t)=k\varphi(N)+1$$
$$e\varphi(N)-et=k\varphi(N)+1$$
$$et=(e-k)\varphi(N)+1$$
$$(e-k)(N-p-q+1)+1=et$$
$$y(N-z)+1=0\ mod(e)$$
$$z=p+q-1\approx 2N^{\frac{1}{2}}\quad y=e-k=\frac{et-1}{\varphi(N)}\approx t$$

2.7.2 代码实现

2.7.2.1 参数范围确定

根据题目所给条件特征，可以确定α约为 1，根据上述数学推导可知，本题可以使用与 Boneh-Durfee 相同的模方程以及多项式集合构造格基矩阵进行求解，区别在于未知变量由 d 改变为 t，但 t 的范围与 d 可攻击的范围相近，可以使用 Boneh-Durfee 进行攻击。

2.7.2.2 构造多项式集合

```
shifts = []
    for k in range(m + 1):
        for i in range(m - k + 1):
            g = x ** i * f ** k * e ** (m - k)
            g = qr(g).lift()
            shifts.append(g)

    for j in range(1, t + 1):
        for k in range(m // t * j, m + 1):
            h = y ** j * f ** k * e ** (m - k)
            h = qr(h).lift()
                shifts.append(h)
```

2.7.2.3 求解 m

利用上述多项式集合我们可以进行格基矩阵的构造，并且重构多项式方程，求得小值解$(y_0，z_0) = (e - k，p + q - 1)$,利用$z_0$的值可以进一步确定$\varphi(N)$以及$d$，遵循与前几题相同的方式，我们可以求得明文 m 的具体值并将其转换为字符串。

m 为 230913323188643024057354978541995677532283168674841160289183905564513987191600918457490 79，具体字符串为 Wir müssen wissen, Wir werden wissen.

2.8 第八题：低位泄露

对于私钥部分低位泄露的情况，可以考虑构造双变元模方程或三变元整系数方程进行求解，通过求解方程的小值解进而求解$\varphi(N)$的值，从而得到私钥 d 的值。令$d = d_1 \cdot 2^r +$

d_0，通过$ed \equiv 1\ mod\varphi(N)$，可以构造二变元模方程$f_{LSB_1}(y,z) = Ny - yz - ed_0 - 1 \equiv (mode \cdot 2^r)$

2.8.1 数学推导

对于上述二变元模方程，其小值解$(y_0, z_0) = (k, p+q-1)$，且上界为

$$Y = 2N^{\alpha+\beta-1}, Z := 3N^{\frac{1}{2}}$$

根据模方程以及根的上界，选定参数 m、τ，可以构造多项式集合：

$$g = y^{t-j} f_{LSB1}{}^{j} W^{m-j} \quad t \in (0, \dots, m) \quad j = 0,1,2,\cdots,t$$

$$h = z^{i}\left(f_{LSB_1}\right)^{j} W^{m-j}\ i \in (1,\cdots,\tau) \quad j = 0,1,2,\cdots m$$

其中$W = e \cdot 2^r$，记$\delta = \log_N 2^r$，$d \le N^B$，$e = N^\alpha$，当满足$\beta \le \delta + \frac{7}{6} - \frac{1}{3}\sqrt{6(\alpha+\delta)+1}$时，能够在多项式时间内找到解。由于能够实现攻击所需要的条件为 $\det(\Lambda)\, T_1(s) < W^{m(s-1)}\left(T_1(s) = T(s,s) = 2^{\frac{s(s-1)}{4}} s^{\frac{s-1}{2}}\right)$,其中

$$s = \frac{1}{2}(m+1)(m+2) + \tau(m+1),$$

$$\det(\Lambda) = W^{\frac{1}{3}m(m+1)(m+2)+\frac{1}{2}\tau m(m+1)} Y^{\frac{1}{3}m(m+1)(m+2)+\frac{1}{2}\tau m(m+1)} Z^{\frac{1}{6}m(m+1)(m+2)+\frac{1}{2}\tau m(m+1)+\frac{1}{2}\tau(\tau+1)(m+1)}.$$

由此可得$Y^{\frac{1}{3}m(m+2)+\frac{1}{2}\tau m} Z^{\frac{1}{6}m(m+2)+\frac{1}{2}\tau m+\frac{1}{2}\tau(\tau+1)} T_1(s)^{\frac{1}{m+1}} < W^{\frac{1}{6}m(m+2)+\frac{1}{2}\tau m-\frac{m}{m+1}}$，选取参数 m=9，$\tau = 3$即可得到合适的格基矩阵进行求解。

2.8.2 代码实现

2.8.2.1 依据参数范围决定攻击方式

首先观察所给条件特征，比如 n 与 e 的关系，n 与d的关系，可以确定α约为 1，β约为 1，δ约为 0.87，满足$\beta \le \delta + \frac{7}{6} - \frac{1}{3}\sqrt{6(\alpha+\delta)+1}$，可以实现攻击

2.8.2.2 构造模方程，确定小值解范围

在整数环 ZZ 上定义两个多项式变量 y,z 利用这些变量构造双变元模方程$f(y,z)$，小值解的上界分别为$Y = 2N^{\alpha+\beta-1}, Z = 3N^{0.5}$,模数大小为$2^{900} * e$，根据上一步确定的参数范围可以进一步确定其具体数值。

2.8.2.3 构造多项式集合

```
    shifts = []
     for t in range(m + 1):
         for j in range(t + 1):
             g = y ** (t-j) * f ** j * W ** (m-j)
             shifts.append(g)

     for i in range(1, tau + 1):
         for j in range(m + 1):
             h = z ** i * f ** j * W ** (m-j)
             shifts.append(h)
```

针对已经构造的模方程，可以构造对应的多项式集合

2.8.2.4 构造格基矩阵并约减

create_lattice 函数用于从多项式的移位列表 shifts 中创建一个格基矩阵，并按照指定的顺序和排序规则进行排序和创建。

create_lattice

```
def create_lattice(pr, shifts, bounds, order="invlex",
sort_shifts_reverse=False, sort_monomials_reverse=False):

    logging.debug(f"Creating a lattice with {len(shifts)} shifts
({order = }, {sort_shifts_reverse = }, {sort_monomials_reverse
= })...")
    if pr.ngens() > 1:
        pr_ = pr.change_ring(ZZ, order=order)
        shifts = [pr_(shift) for shift in shifts]

    monomials = set()
    for shift in shifts:
        monomials.update(shift.monomials())

    shifts.sort(reverse=sort_shifts_reverse)
    monomials = sorted(monomials, reverse=sort_monomials_reverse)
    L = matrix(ZZ, len(shifts), len(monomials))
    for row, shift in enumerate(shifts):
        for col, monomial in enumerate(monomials):
            L[row, col] = shift.monomial_coefficient(monomial) *
monomial(*bounds)

    monomials = [pr(monomial) for monomial in monomials]

       return L, monomials
```

reduce_lattice

```
def reduce_lattice(L, delta=0.8):

       return L.LLL(delta)
```

使用格基约减算法（当前为 LLL 算法）来减小格基。L：格基，delta：LLL 算法的 delta 参数（默认为 0.8），调用完成后返回减小后的格基。

2.8.2.5 重构多项式方程

reconstruct_polynomials 函数从约减后的格基矩阵 B 中重构出多项式。它遍历每一行，根据约减后的基向量重构多项式，并可选地进行多项式的预处理、除以原始多项式 f、以及多项式之间的最大公因数除法。

```
def reconstruct_polynomials(B, f, modulus, monomials, bounds,
preprocess_polynomial=lambda x: x, divide_gcd=True):
    divide_original = f is not None
    modulus_bound = modulus is not None
    logging.debug(f"Reconstructing polynomials ({divide_original
= }, {modulus_bound = }, {divide_gcd = })...")
    polynomials = []
    for row in range(B.nrows()):
        norm_squared = 0
        w = 0
        polynomial = 0
        for col, monomial in enumerate(monomials):
            if B[row, col] == 0:
                continue
            norm_squared += B[row, col] ** 2
            w += 1
            assert B[row, col] % monomial(*bounds) == 0
            polynomial += B[row, col] * monomial //
monomial(*bounds)

        polynomial = preprocess_polynomial(polynomial)

        if divide_original and polynomial % f == 0:
            logging.debug(f"Original polynomial divides
reconstructed polynomial at row {row}, dividing...")
            polynomial //= f

        if divide_gcd:
            for i in range(len(polynomials)):
                g = gcd(polynomial, polynomials[i])
                if g != 1 and g.is_constant():
                    logging.debug(f"Reconstructed polynomial has
gcd {g} with polynomial at {i}, dividing...")
                    polynomial //= g
                    polynomials[i] //= g

        if polynomial.is_constant():
            logging.debug(f"Polynomial at row {row} is constant,
ignoring...")
            continue

        polynomials.append(polynomial)
    return polynomials
```

2.8.2.6 使用 Grobner 基方法求根

该函数利用 Groebner 基的概念，对于多项式环中的多项式集合尝试找到其根。通过逐步求解方程组的方式，利用 Groebner 基的性质进行根的计算。如果成功找到符合条件的根，则以生成器的形式将其返回。

```
def find_roots_groebner(pr, polynomials):
    gens = pr.gens()
    s = Sequence(polynomials, pr.change_ring(QQ, order="lex"))
    while len(s) > 0:
        G = s.groebner_basis()
        logging.debug(f"Sequence length: {len(s)}, Groebner basis
length: {len(G)}")
        if len(G) == len(gens):
            logging.debug(f"Found Groebner basis with length
{len(gens)}, trying to find roots...")
            roots = {}
            for polynomial in G:
                vars = polynomial.variables()
                if len(vars) == 1:
                    for root in find_roots_univariate(vars[0],
polynomial.univariate_polynomial()):
                        roots = {**roots, **root}

            if len(roots) == pr.ngens():
                yield roots
                print(roots)
                return

            logging.debug(f"System is underdetermined, trying to
find constant root...")
            G = Sequence(s, pr.change_ring(ZZ,
order="lex")).groebner_basis()
            vars = tuple(map(lambda x: var(x), gens))
            for solution_dict in solve([polynomial(*vars) for
polynomial in G], vars, solution_dict=True):
                logging.debug(solution_dict)
                found = False
                roots = {}
                for i, v in enumerate(vars):
                    s = solution_dict[v]
                    if s.is_constant():
                        if not s.is_zero():
                            found = True
                        roots[gens[i]] = int(s) if s.is_integer()
else int(s) + 1
                    else:
                        roots[gens[i]] = 0
                if found:
                    yield roots
                    return

            return
        else:
```

```
        s.pop()
```

2.8.2.7 恢复明文 m

在使用 Groebner 方法求得小值解$(y_0, z_0) = (k, p + q - 1)$之后，沿用前几题的求解思路，可以得到 m 的具体值及其字符串表达。对于本题，我们求出的明文 m 为 19257083064022027458506350341962968790370705261004109792828051057459887810448027075446261379656991890881836889318692824251248344438867771946623848834010403937908413433296999429029384O，转换为字符串为 Plato is my friend, Aristotle is my friend, but my greatest friend is truth.

2.9 第九题：低位泄露

与第八题相同，对于私钥部分低位泄露的情况，考虑构造双变元模方程进行求解，通过求解方程的小值解进而求解$\varphi(N)$的值，从而得到私钥 d 的值。令$d = d_1 \cdot 2^r + d_0$，通过$ed \equiv 1\ mod\varphi(N)$，可以构造二变元模方程$f_{LSB_1}(y, z) = Ny - yz - ed_0 - 1 \equiv (mode \cdot 2^r)$

2.9.1 数学推导

对于上述二变元模方程，其小值解$(y_0, z_0) = (k, p + q - 1)$，且上界为

$$Y = 2N^{\alpha+\beta-1}, Z := 3N^{\frac{1}{2}}$$

根据模方程以及根的上界，选定参数 m、τ，可以构造多项式集合：

$$g = y^{t-j} {f_{LSB1}}^{j} W^{m-j} \quad t \in (0, \dots, m) \quad j = 0,1,2, \cdots, t$$

$$h = z^{i} \left(f_{LSB_1}\right)^{j} W^{m-j}\ i \in (1, \cdots, \tau) \quad j = 0,1,2, \cdots m$$

对于本题，可以选取参数 m=4，τ=1，利用多项式集合构造格基矩阵即可进一步求解双变元模方程的解。

2.9.2 代码实现

2.9.2.1 依据参数范围决定攻击方式

针对本题提供的题设信息，可以确定α约为 0.017，β约为 1，δ约为 0.517，满足$\beta \leq \delta + \frac{7}{6} - \frac{1}{3}\sqrt{6(\alpha+\delta)+1}$，因此可以在多项式时间内实现攻击

。

2.9.2.2 构造模方程与多项式集合

在本题中使用了与第八题相同的思路，使用了相同的模方程，区别在于参数 m 和 t 的值选取不同，多项式集合构建的格基矩阵也有所不同，本题使用 m=4，t=1 即可实现攻击。

2.9.2.3 构造格基矩阵并约减

在多项式的集合中按照指定的顺序和排序规则进行排序和创建格基矩阵，并且使用 LLL 算法对创建的格基矩阵进行约减。

2.9.2.4 重构多项式方程

从约减后的格基矩阵中重构出多项式，它遍历每一行，根据约减后的基向量重构多项式，并可选地进行多项式的预处理、除以原始多项式 f、以及多项式之间的最大公因数除法。

2.9.2.5 使用 Grobner 基方法求根

该函数利用 Groebner 基的概念，通过多项式环中的多项式集合，尝试找到其根。通过逐步求解方程组的方式，利用 Groebner 基的性质进行根的计算。如果成功找到符合条件的根，则以生成器的形式返回。

2.9.2.6 恢复明文 m

与第八题相似，在使用 Groebner 方法求得小值解$(y_0, z_0) = (k, p+q-1)$之后，沿用前几题的求解思路，可以得到 m 的具体值及其字符串表达。

M=29626782283555027795833457323782819646058061055998626927231744698032046735445819468241597697065609850960675502435183597768663726152308392797786658205159706413941344708824463899307522327391778963321392621605607582510474846241599109146407921519715337758543965970340752770069692353557305140735862098，其字符串形式为：

Resolving the composite numbers into their prime factors is known to be one of the most important and useful in arithmetic.

2.10第十题：高位泄露

对于部分私钥泄露的 RSA 攻击，Boneh, Durfee, and Frankel 等人首次提出了暴露部分最高有效位的攻击，Ernst 等人将攻击扩展到了全范围的加密指数,Takayasu 等人进一步改善了上述攻击的范围,将已知部分最高有效位攻击扩展到了$N^{0.5625}$。

2.10.1数学推导

记$d = d_1 \cdot 2^r + d_2, 0 < r < l_d, 0 \leqslant d_1 < 2^{l_d - r}, 0 \leqslant d_2 < 2^r$,由$1 = ed - k\varphi(N) = e(d_1 \cdot 2^r + d_2) - k[N - (p + q - 1)]$可以构造模方程$f_{MSBs1}(x, y, z) = A + ex - Ny + yz$，其小值解的上界为$X = N^{\delta}$，$Y = 2N^{\alpha+\beta-1}, Z = 3N^{\frac{1}{2}}$，上述攻击在$\delta \leqslant \frac{5}{6} - \frac{1}{3}\sqrt{6(\alpha + \beta) - 5} - \varepsilon$时是有效的，根据本题所给已知信息，可以判断本题符合攻击界限。除此之外还可以根据已知信息对k的值进行估计，将估计值作为已知信息修改原本的模方程。

2.10.2代码实现

2.10.2.1 寻找小根，进行 coppersmith 攻击

根据题目所给的已知信息，仅最低有效位的 10 位未泄露，可确定参数 bounds = $(2^{20}, 2^{513})$

```
def small_roots(f, bounds, m=1, d=None):
    if not d:
        d = f.degree()

    R = f.base_ring()
    N = R.cardinality()

    k = ZZ(f.coefficients().pop(0))
    g = gcd(k, N)
    k = R(k/g)

    f *= 1/k
```

```
    f = f.change_ring(ZZ)

    vars = f.variables()
    G = Sequence([], f.parent())
    for k in range(m):
        for i in range(m-k+1):
            for subvars in itertools.combinations_with_replacement(vars[1:], i):
                g = f**k * prod(subvars) * N**(max(d-k, 0))
                G.append(g)

    B, monomials = G.coefficient_matrix()
    monomials = vector(monomials)

    factors = [monomial(*bounds) for monomial in monomials]
    for i, factor in enumerate(factors):
        B.rescale_col(i, factor)

    B = B.dense_matrix().LLL()
    B = B.change_ring(QQ)
    for i, factor in enumerate(factors):
        B.rescale_col(i, Integer(1)/factor)

    H = Sequence([], f.parent().change_ring(QQ))
    for h in filter(None, B*monomials):
        H.append(h)
        I = H.ideal()
        if I.dimension() == -1:
            H.pop()
        elif I.dimension() == 0:
            roots = []
            for root in I.variety(ring=ZZ):
                root = tuple(R(root[var]) for var in f.variables())
                roots.append(root)
            return roots

    return []

k1 = e1*leak1 // n1 + 1
PR.<x,y> = PolynomialRing(Zmod(e1*leak1))
f = 1 + k1*((n1+1)-y) - e1*x
bounds = (2^10,2^513)

res = small_roots(f,bounds,m=2,d=2)
leak = int(res[0][1])

PR.<x> = PolynomialRing(RealField(1000))
f = x*(leak-x) - n1
ph = int(f.roots()[0][0])


PR.<x> = PolynomialRing(Zmod(n1))
f = ph + x
res = f.small_roots(X=2^(160+6), beta=0.499,epsilon=0.02)[0]
```

```
p1 = int(ph + res)
q1 = n1 // p1
print(p1)
    print(q1)
```

2.10.2.2 恢复明文

```
from gmpy2 import *
phi=(p-1)*(q-1)
d=invert(e,phi)
m=pow(c,d,n)
    print(m)
```

得到 m=820307570753893249794408475962305073085942139245561017669 转换为字符串即：Eureka, I have found it!

2.11第十一题：高位泄露

此题与第十题相似，在其基础上进行拓展。本题高位泄露的信息发生的变化，相较于第十题有所减少，但仍满足可行的攻击范围，可以在多项式时间内求解 d。本题仍然采用与第十题相同的方程和格基矩阵，区别在于方程的小值解的上界发生了变化。

2.11.1数学推导

具体推导过程同第十题

2.11.2代码实现

脚本同第十题，根据题目所给信息最低有效位的 20 位未泄露，其余部分均已知，修改参数 bounds = $(2^{20}, 2^{513})$，得到结果
m=871459330528493685900356342244447392885109122774019102948898640050654591861594334271960704663752901134803717028296430848471791111128678516670233579393144431497657245644265754833188314998240383835607271327755688562060388816458780444461498953491390155933696 73 对应字符串：I have discovered a truly marvelous proof of this, which however the margin is not large enough to contain.

2.12第十二题：已知 q 信息

本题与第七题相似，并未直接给出 d 的泄露信息，但给出了与 RSA 密钥中N_{12}取值相近的数N的已知信息，并且N与N_{12}含有部分近似公因子，即$N = kq_{12} + r$

2.12.1数学推导

基于第七题根据已知信息对无任何比特位泄露的私钥 d 进行变换的思路，以及本题提供的已知信息和 RSA 密钥体制的特性，我们可以得到：

$$dq = d(mod\ (q-1))$$
$$N' = kq + r$$
$$r = N' - k(mod\ (q-1))$$
$$2^{511} < k < 2^{512}, 2^{255} < r < 2^{256}$$
$$d = N' - k\ \ e(N'-k) \equiv 1(mod\varphi(N))$$
$$e(k(q-1)+r) \equiv 1(mod\varphi(N))$$
$$er = 1 + t(\frac{N+1}{2} - \frac{P+q}{2}) - 2e(\frac{N'}{2} - \frac{r+k}{2})$$
$$1 + x(A-y) - 2e(B-z) = 0(mod\ \ e)$$

对于上述方程，由于$2^{255} < r < 2^{256}$，与 Boneh-Durfee 中的 d 有着相近的数量级，理论上仍可通过求解三变元整系数方程进行求解。

除此之外，也可也考虑将其转换为部分高位泄露的形式，由于：

$$e(N'-k') = k\varphi(N) + 1$$
$$eN' - ek' - k\varphi(N) - 1 = 0$$
$$eN' - 1 - ek' - k(N - (p+q-1)) = 0$$
$$eN' - 1 - ex - yN + yz = 0$$

其中$2^{511} < k < 2^{512}$，故$X = N^{\delta} = 2^{512}$，$Y = 2N^{\alpha+\beta-1}, Z = 3N^{\frac{1}{2}}$，根据题目所给信息可以确定$\delta$、$\alpha$、$\beta$的具体取值，发现其满足$\delta \leqslant \frac{5}{6} - \frac{1}{3}\sqrt{6(\alpha+\beta)-5} - \varepsilon$，因此可以采用与已知高位泄露相似的方法进行求解

2.12.2代码实现

若采用将其转换为与 Boneh-Durfee 攻击相似的形式，可以使用与第七题或第二题的脚本，需将模方程进行修改，并修改具体参数进行格基矩阵构建和多项式重构，进而求解方程的解，若采用将其转换为高位泄露类似的情况多项式集合构造代码如下：

```
pr = f.parent()
x, y, z = pr.gens()
A = int(f.constant_coefficient())
assert A != 0
while gcd(A, X) != 1:
    X += 1
while gcd(A, Y) != 1:
    Y += 1
while gcd(A, Z) != 1:
    Z += 1
while gcd(A, M) != 1:
    M += 1
```

```
    R =M* X ** (m-1)*Y**(m-1) * Z ** (m + tau-1)
    f_ = (pow(A, -1, R) * f % R).change_ring(ZZ)
    shifts = []
    for j in range(m):
        for i2 in range(j + 1):
            for i3 in range(i2 + tau + 1):
                g = x ** (j-i2) * y ** i2 * z ** i3 * f_ * X ** (m - 1-j+i2) * Y
** (m - 1-i2) * Z ** (m-1+tau-i3)
                shifts.append(g)
    for i2 in range(m+1):
        for i3 in range(i2 + tau+1):
            g_ = x ** (m-i2) * y ** i2 * z ** i3 * R

            shifts.append(g_)
```

格基矩阵构造与多项式重构的代码与低位泄露相同，此处不再给出。通过以上步骤可以求得 d 的值，进而求取 m。

2.13 随机数发生器

2.13.1寻找未翻转的比特串

根据题目所给信息，构成大整数 N 的两个大素数 p、q 中的 p 由随机数发生器生成，q 是随机产生的素数。对求解得到的 p、q 进行观察可以发现其中一个数具有明显的周期特征。随机数生成器通过对输入的种子密钥s_i进行变换扩展使之成为具有周期的 512 位比特串，比特串部分位发生翻转后寻找最近的素数即为 p，且根据素数密度公式及实验检验可知，搜寻素数仅对比特串的末尾几个比特位产生影响。进而对攻击得到的 p，针对其种子密钥s_i长度不超过 2048 的特性，对所有可能的周期进行遍历，并按照周期长度将 p 截断，依次比对得到翻转次数尽可能少的周期序列。即首先观察周期长度 11，生成所有长度为 11 位的比特串，逐一将其作为周期序列，与截断后的 p 进行比较，得到每一个比特串作为周期序列时的翻转次数，然后观察周期长度 10，依次类推，遍历所有可能的情况后选择翻转次数尽可能少的比特串作为周期序列。

2.13.2种子密钥及随机数生成器

对于获得的周期序列，利用其推测随机数发生器的规律以及种子密钥。对于第一题中获得的 p，可能的周期序列为 00，对于第二题中获得的 p，可能的周期序列为 0001010，对于第六题中获得的 p，可能的周期序列为 00110101100，对于第七题中获得的 p，可能的周期序列为 001111000，对于第八题中获得的 p，可能的周期序列为 110011000，对于第九题

中获得的 p，可能的周期序列为 000111001，对于第十题中获得的 p，可能的周期序列为 110000001，对于第十一题中获得的 p，可能的周期序列为 11010000000，通过上述周期序列我们对种子密钥以及随机数生成器存在的规律进行猜测。通过查阅资料发现，目前较为主流的随机数生成器主要有线性同余生成器 LCG、线性反馈移位寄存器 LFSR、BBS 产生器、Rabin 生成器等。基于多种随机数发生器的特点以及对得到的周期序列进行观察，第一题的周期序列其种子密钥s_1大概率为 0，第二题的周期序列其可能的种子密钥s_2为 90，第六题的种子密钥s_6可能为 864，第七题的种子密钥s_7可能为 216，第八题的种子密钥s_8可能为 148，第九题的种子密钥s_9可能为 101，第十题的种子密钥s_{10}可能为 195，第十一题的种子密钥s_{11}可能为 320。随机数发生器可能的规律为，对输入的种子密钥s_i，基于线性同余算法对其进行处理，对生成的随机数数列，将其中的每个数作为线性反馈移位的输入，利用设定的反馈多项式，计算出一个新的反馈位，对数列中的第 i 个数，其第 i 次迭代时产生的新的最低位输出作为比特序列的一部分。

第三章　结论

根据以上十二题的解题过程以及随机数发生器做出的猜想，我们可以得知 RSA 密码体制在私钥泄露方面的安全性仍不够牢靠，如若通过侧信道等手段得到解密信息，就可以利用上述攻击手段进行破解。经过以上解题过程，我们对加深了 RSA 体制的加解密过程，对于理论上界的确定以及部分参数的选取仍显生疏。而最后对于随机数生成器的分析，更能帮助我们深入地从产生两个大素数这一密钥生成过程理解 RSA 的奥义。

对于 RSA 私钥部分信息的泄露而言，目前为止，攻击手段较为初级，更多的是对于其间过程的理解，正如 wiener attack 拓展为 Boneh-Durfee 以及 coppersmith 进阶为基于格的 coppersmith 攻击一样，再加上双变元与三变元（或更多变元）间的辨析，基于部分私钥信息泄露的 RSA 攻击才显得如此莫测。

第四章　参考文献

[1] 王世雄.基于格的 Coppersmith 方法与针对 RSA 密码的部分私钥泄露攻击[D].国防科学技术大学,2015.

[2] 王世雄,屈龙江,李超,等.私钥低比特特定泄露下的 RSA 密码分析[J].密码学报,2015,2(05):390-403.DOI:10.13868/j.cnki.jcr.000088.

[3] 周永彬,姜子铭,王天宇,等.RSA 及其变体算法的格分析方法研究进展[J].软件学报,2023,34(09):4310-4335.DOI:10.13328/j.cnki.jos.006657.

[4] Boneh, Dan, Glenn Durfee, and Yair Frankel. "Exposing an RSA private key given a small fraction of its bits." Full version of the work from Asiacrypt 98 (1998).

[5] Blomer, Johannes. "New partial key exposure attacks on RSA." Crypto 2003. 2003.

[6] Boneh, Dan, and Glenn Durfee. "Cryptanalysis of RSA with private key d less than N 0.292." Advances in Cryptology—EUROCRYPT'99: International Conference on the Theory and Application of Cryptographic Techniques Prague, Czech Republic, May 2–6, 1999 Proceedings 18. Springer Berlin Heidelberg, 1999.

[7] Boneh, Dan. "The decision diffie-hellman problem." International algorithmic number theory symposium. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998.

[8] Boneh, Dan, and Glenn Durfee. "Cryptanalysis of RSA with private key d less than N/sup 0.292." IEEE transactions on Information Theory 46.4 (2000): 1339-1349.

[9] Boneh, Dan, Glenn Durfee, and Yair Frankel. "An attack on RSA given a small fraction of the private key bits." Advances in Cryptology—ASIACRYPT'98: International Conference on the Theory and Application of Cryptology and Information Security Beijing, China, October 18–22, 1998 Proceedings. Springer Berlin Heidelberg, 1998.

[10] Coppersmith, Don. "Finding a small root of a univariate modular equation." International Conference on the Theory and Applications of Cryptographic Techniques. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996.

[11] Durfee, Glenn, and Phong Q. Nguyen. "Cryptanalysis of the RSA schemes with short secret exponent from Asiacrypt'99." International Conference on the Theory and Application of Cryptology and Information Security. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000.

[12] Ernst, Matthias, et al. "Partial key exposure attacks on RSA up to full size exponents." Advances in Cryptology–EUROCRYPT 2005: 24th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Aarhus, Denmark, May 22-26, 2005. Proceedings 24. Springer Berlin Heidelberg, 2005.

[13] Herrmann, Mathias, and Alexander May. "Attacking power generators using unravelled linearization: When do we output too much?." International Conference on the Theory and Application of Cryptology and Information Security. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009.

[14]Howgrave-Graham, Nicholas. "Finding small roots of univariate modular equations revisited." IMA International Conference on Cryptography and Coding. Berlin, Heidelberg: Springer Berlin Heidelberg, 1997.

[15]Jochemsz, Ellen, and Alexander May. "A strategy for finding roots of multivariate polynomials with new applications in attacking RSA variants." Advances in Cryptology–ASIACRYPT 2006: 12th International Conference on the Theory and Application of Cryptology and Information Security, Shanghai, China, December 3-7, 2006. Proceedings 12. Springer Berlin Heidelberg, 2006.

[16]Lu, Yao, et al. "Solving linear equations modulo unknown divisors: revisited." International Conference on the Theory and Application of Cryptology and Information Security. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015.

[17]Sarkar, Santanu. "Partial key exposure: generalized framework to attack RSA." Progress in Cryptology–INDOCRYPT 2011: 12th International Conference on Cryptology in India, Chennai, India, December 11-14, 2011. Proceedings 12. Springer Berlin Heidelberg, 2011.

[18]Suzuki, Kaichi, Atsushi Takayasu, and Noboru Kunihiro. "Extended partial key exposure attacks on RSA: improvement up to full size decryption exponents." Theoretical Computer Science 841 (2020): 62-83.

[19]Takayasu, Atsushi, and Noboru Kunihiro. "Better lattice constructions for solving multivariate linear equations modulo unknown divisors." IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences 97.6 (2014): 1259-1272.

[20]Takayasu, Atsushi, and Noboru Kunihiro. "Partial key exposure attacks on RSA: achieving the Boneh-Durfee bound." International Conference on Selected Areas in Cryptography. Cham: Springer International Publishing, 2014.

第五章　谢辞

从四月下旬开始，本小组开始着手进行题目求解，几乎从零开始接触真正的解密，一点点摸索，一点点探究，查阅了大量文献资料，尝试了各种可能的攻击脚本，最终得出上述解答，但这是我们在繁忙的学习生活中挤压时间所得出的成果，在漫天的作业与大山般的期末考试中匀得一些喘息的时间，探索自己喜欢的事情。

首先感谢竞赛官方精心拟题让热爱密码的学子能够切实感受到 RSA 体制的精妙与保护信息安全的重要性。然后感谢指导老师汪定的指导。接下来感谢我们自己，尽管这学期学业尤为繁重，但还是用不怕苦的精神，往赛题里钻，又用不怕难的精神反复进行“失败-再来”的循环。或许这就是解密人的一点点影子吧，只有从每次的“not found”里总结经验，才能知道看到解答不是乱码的惊喜。

第六章　附录

- 数学工具：sage

Sage 是免费的开源数学软件，支持代数、几何、数论、密码学、数值计算和相关领域的研究和教学。

Sage 的主要作用包括，求逆元，设置有限域上的一元多项式，扩展欧几里得算法，中国剩余定理，求离散对数，取模求根，求欧拉函数，输出表达式近似值，素数分布$（P_i(x)）$，创建整数域中的椭圆曲线，求常微分方程和方程组，矩阵特征值求解，描述随机过程等。这些密码学当中的常用方法，如果单独使用 Python 进行，将会构造一个较为复杂的函数并且需要包含更多的库，尤其是在大整数方面，构造相应的数据结构十分复杂，使用其他的库，诸如 gmm 等也有许多问题。我们尝试了一些可能的方法，均不如直接配置使用 sage 容易，并且在计算的过程当中还存在着诸多困难。因此我们还是使用了 sage 编程环境，进行相关数学问题的处理。

- 程序源代码

第一题：

```
n=115757306476480823890101488188630776729264818625956711197209182923316919450004566705136037138124707757241364238362202848334326230965866182884372905599812001417295792076008382155051507854558917751540997152031158153627516638609891662992021455525208876152512219426022730388430168986634812925997477433074009315297

e=367591197601075143424515220145047237567819327788545360078448869575666747868142196161845696066248292540754114660115311774621928346740423145826321699988699376300478224028491416362044056244652836185591483880232396180100239835153028290962332335463556391207823768151660708588698077410375905280297922784966144084 03

c=102306866548166297283990570728719380746274356547349456313911134525511009884527595077619658312746154540918040609904792766267540939554230661255790442398718139161414133218935455395440009626652377478022047762911840522170980051387391261774896710421983974318857176556467366545165306790641364917195790444235849931561


def factor_rsa_wiener(N, e):
```

```
        N = Integer(N)
        e = Integer(e)
        cf = (e / N).continued_fraction().convergents()
        for f in cf:
            k = f.numer()
            d = f.denom()
            if k == 0:
                continue
            phi_N = ((e * d) - 1) / k
            b = -(N - phi_N + 1)
            dis = b ^ 2 - 4 * N
            if dis.sign() == 1:
                dis_sqrt = sqrt(dis)
                p = (-b + dis_sqrt) / 2
                q = (-b - dis_sqrt) / 2
                if p.is_integer() and q.is_integer() and (p * q) % N == 0:
                    p = p % N
                    q = q % N
                    if p > q:
                        return (p, q)
                    else:
                        return (q, p)

factor_rsa_wiener(n,e)
```

第二题：

```
from __future__ import print_function
import time


"""
Setting debug to true will display more informations
about the lattice, the bounds, the vectors...
"""
debug = True

"""
Setting strict to true will stop the algorithm (and
return (-1, -1)) if we don't have a correct
upperbound on the determinant. Note that this
doesn't necesseraly mean that no solutions
will be found since the theoretical upperbound is
usualy far away from actual results. That is why
you should probably use `strict = False`
"""
strict = False

"""
This is experimental, but has provided remarkable results
so far. It tries to reduce the lattice as much as it can
while keeping its efficiency. I see no reason not to use
this option, but if things don't work, you should try
disabling it
"""
helpful_only = True
dimension_min = 7 # stop removing if lattice reaches that dimension

# display stats on helpful vectors
def helpful_vectors(BB, modulus):
    nothelpful = 0
    for ii in range(BB.dimensions()[0]):
        if BB[ii,ii] >= modulus:
            nothelpful += 1

    print(nothelpful, "/", BB.dimensions()[0], " vectors are not helpful")

# display matrix picture with 0 and X
def matrix_overview(BB, bound):
    for ii in range(BB.dimensions()[0]):
        a = ('%02d ' % ii)
        for jj in range(BB.dimensions()[1]):
            a += '0' if BB[ii,jj] == 0 else 'X'
            if BB.dimensions()[0] < 60:
                a += ' '
        if BB[ii, ii] >= bound:
            a += '~'
        print(a)

# tries to remove unhelpful vectors
# we start at current = n-1 (last vector)
def remove_unhelpful(BB, monomials, bound, current):
    # end of our recursive function
    if current == -1 or BB.dimensions()[0] <= dimension_min:
        return BB

    # we start by checking from the end
    for ii in range(current, -1, -1):
        # if it is unhelpful:
        if BB[ii, ii] >= bound:
            affected_vectors = 0
            affected_vector_index = 0
            # let's check if it affects other vectors
            for jj in range(ii + 1, BB.dimensions()[0]):
```

```
            # if another vector is affected:
            # we increase the count
            if BB[jj, ii] != 0:
                affected_vectors += 1
                affected_vector_index = jj

        # level:0
        # if no other vectors end up affected
        # we remove it
        if affected_vectors == 0:
            print("* removing unhelpful vector", ii)
            BB = BB.delete_columns([ii])
            BB = BB.delete_rows([ii])
            monomials.pop(ii)
            BB = remove_unhelpful(BB, monomials, bound, ii-1)
            return BB

        # level:1
        # if just one was affected we check
        # if it is affecting someone else
        elif affected_vectors == 1:
            affected_deeper = True
            for kk in range(affected_vector_index + 1, BB.dimensions()[0]):
                # if it is affecting even one vector
                # we give up on this one
                if BB[kk, affected_vector_index] != 0:
                    affected_deeper = False
            # remove both it if no other vector was affected and
            # this helpful vector is not helpful enough
            # compared to our unhelpful one
            if affected_deeper and abs(bound - BB[affected_vector_index, affected_vector_index]) < abs(bound -
BB[ii, ii]):
                print("* removing unhelpful vectors", ii, "and", affected_vector_index)
                BB = BB.delete_columns([affected_vector_index, ii])
                BB = BB.delete_rows([affected_vector_index, ii])
                monomials.pop(affected_vector_index)
                monomials.pop(ii)
                BB = remove_unhelpful(BB, monomials, bound, ii-1)
                return BB
    # nothing happened
    return BB

"""
Returns:
* 0,0   if it fails
* -1,-1 if `strict=true`, and determinant doesn't bound
* x0,y0 the solutions of `pol`
"""
def boneh_durfee(pol, modulus, mm, tt, XX, YY):
    """
    Boneh and Durfee revisited by Herrmann and May

    finds a solution if:
    * d < N^delta
    * |x| < e^delta
    * |y| < e^0.5
    whenever delta < 1 - sqrt(2)/2 ~ 0.292
    """

    # substitution (Herrman and May)
    PR.<u, x, y> = PolynomialRing(ZZ)
    Q = PR.quotient(x*y + 1 - u) # u = xy + 1
    polZ = Q(pol).lift()

    UU = XX*YY + 1

    # x-shifts
```

```
gg = []
for kk in range(mm + 1):
    for ii in range(mm - kk + 1):
        xshift = x^ii * modulus^(mm - kk) * polZ(u, x, y)^kk
        gg.append(xshift)
gg.sort()

# x-shifts list of monomials
monomials = []
for polynomial in gg:
    for monomial in polynomial.monomials():
        if monomial not in monomials:
            monomials.append(monomial)
monomials.sort()

# y-shifts (selected by Herrman and May)
for jj in range(1, tt + 1):
    for kk in range(floor(mm/tt) * jj, mm + 1):
        yshift = y^jj * polZ(u, x, y)^kk * modulus^(mm - kk)
        yshift = Q(yshift).lift()
        gg.append(yshift) # substitution

# y-shifts list of monomials
for jj in range(1, tt + 1):
    for kk in range(floor(mm/tt) * jj, mm + 1):
        monomials.append(u^kk * y^jj)

# construct lattice B
nn = len(monomials)
BB = Matrix(ZZ, nn)
for ii in range(nn):
    BB[ii, 0] = gg[ii](0, 0, 0)
    for jj in range(1, ii + 1):
        if monomials[jj] in gg[ii].monomials():
            BB[ii, jj] = gg[ii].monomial_coefficient(monomials[jj]) * monomials[jj](UU,XX,YY)

# Prototype to reduce the lattice
if helpful_only:
    # automatically remove
    BB = remove_unhelpful(BB, monomials, modulus^mm, nn-1)
    # reset dimension
    nn = BB.dimensions()[0]
    if nn == 0:
        print("failure")
        return 0,0

# check if vectors are helpful
if debug:
    helpful_vectors(BB, modulus^mm)

# check if determinant is correctly bounded
det = BB.det()
bound = modulus^(mm*nn)
if det >= bound:
    print("We do not have det < bound. Solutions might not be found.")
    print("Try with highers m and t.")
    if debug:
        diff = (log(det) - log(bound)) / log(2)
        print("size det(L) - size e^(m*n) = ", floor(diff))
    if strict:
        return -1, -1
else:
    print("det(L) < e^(m*n) (good! If a solution exists < N^delta, it will be found)")

# display the lattice basis
if debug:
    matrix_overview(BB, modulus^mm)
```

```
    # LLL
    if debug:
        print("optimizing basis of the lattice via LLL, this can take a long time")

    BB = BB.LLL()

    if debug:
        print("LLL is done!")

    # transform vector i & j -> polynomials 1 & 2
    if debug:
        print("looking for independent vectors in the lattice")
    found_polynomials = False

    for pol1_idx in range(nn - 1):
        for pol2_idx in range(pol1_idx + 1, nn):
            # for i and j, create the two polynomials
            PR.<w,z> = PolynomialRing(ZZ)
            pol1 = pol2 = 0
            for jj in range(nn):
                pol1 += monomials[jj](w*z+1,w,z) * BB[pol1_idx, jj] / monomials[jj](UU,XX,YY)
                pol2 += monomials[jj](w*z+1,w,z) * BB[pol2_idx, jj] / monomials[jj](UU,XX,YY)

            # resultant
            PR.<q> = PolynomialRing(ZZ)
            rr = pol1.resultant(pol2)

            # are these good polynomials?
            if rr.is_zero() or rr.monomials() == [1]:
                continue
            else:
                print("found them, using vectors", pol1_idx, "and", pol2_idx)
                found_polynomials = True
                break
        if found_polynomials:
            break

    if not found_polynomials:
        print("no independant vectors could be found. This should very rarely happen...")
        return 0, 0

    rr = rr(q, q)

    # solutions
    soly = rr.roots()
    if len(soly) == 0:
        print("Your prediction (delta) is too small")
        return 0, 0
    soly = soly[0][0]
    ss = pol1(q, soly)
    solx = ss.roots()[0][0]
    return solx, soly

def example():
    # the modulus
    N =
0xd231f2c194d3971821984dec9cf1ef58d538975f189045ef8a706f6165aab4929096f61a3eb7dd8021bf3fdc41fe
3b3b0e4ecc579b4b5e7e035ffcc383436c9656533949881dca67c26d0e770e4bf62a09718dbabc2b40f2938f16327
e347f187485aa48b044432e82f5371c08f6e0bbde46c713859aec715e2a2ca66574f3eb
    # the public exponent
    e =
0x5b5961921a49e3089262761e89629ab6dff2da1504a0e5eba1bb7b20d63c785a013fd6d9e021c01baf1b238309
54d488041b92bca2fe2c92e3373dedd7e625da11275f6f18ee4aef336d0637505545f70f805902ddbacb21bb8276d
34a0f6dfe37ede87dd95bb1494dbb5763639ba3984240f1178e32aa36ee3c5fcc8115dde5

    # the hypothesis on the private exponent (the theoretical maximum is 0.292)
```

```
    delta = .28 # this means that d < N^delta

    m = 13 # size of the lattice (bigger the better/slower)

    # you need to be a lattice master to tweak these
    t = int((1-2*delta) * m)  # optimization from Herrmann and May
    X = 2*floor(N^delta)  # this _might_ be too much
    Y = floor(N^(1/2))    # correct if p, q are ~ same size
    # Problem put in equation
    P.<x,y> = PolynomialRing(ZZ)
    A = int((N+1)/2)
    pol = 1 + x * (A + y)

    #
    # Find the solutions!
    #

    # Checking bounds
    if debug:
        print("=== checking values ===")
        print("* delta:", delta)
        print("* delta < 0.292", delta < 0.292)
        print("* size of e:", int(log(e)/log(2)))
        print("* size of N:", int(log(N)/log(2)))
        print("* m:", m, ", t:", t)

    # boneh_durfee
    if debug:
        print("=== running algorithm ===")
        start_time = time.time()

    solx, soly = boneh_durfee(pol, e, m, t, X, Y)

    # found a solution?
    if solx > 0:
        print("=== solution found ===")
        if False:
            print("x:", solx)
            print("y:", soly)

        d = int(pol(solx, soly) / e)
        print("private key found:", d)
    else:
        print("=== no solution was found ===")

    if debug:
        print(("=== %s seconds ===" % (time.time() - start_time)))

if __name__ == "__main__":
    example()
```

第三题：

```
from __future__ import print_function
import time

##########################################
# Config
##########################################

"""
Setting debug to true will display more informations
about the lattice, the bounds, the vectors...
"""
debug = True

"""
Setting strict to true will stop the algorithm (and
```

```
return (-1, -1)) if we don't have a correct
upperbound on the determinant. Note that this
doesn't necesseraly mean that no solutions
will be found since the theoretical upperbound is
usualy far away from actual results. That is why
you should probably use `strict = False`
"""
strict = False

"""
This is experimental, but has provided remarkable results
so far. It tries to reduce the lattice as much as it can
while keeping its efficiency. I see no reason not to use
this option, but if things don't work, you should try
disabling it
"""
helpful_only = True
dimension_min = 7 # stop removing if lattice reaches that dimension

# display stats on helpful vectors
def helpful_vectors(BB, modulus):
    nothelpful = 0
    for ii in range(BB.dimensions()[0]):
        if BB[ii,ii] >= modulus:
            nothelpful += 1

    print(nothelpful, "/", BB.dimensions()[0], " vectors are not helpful")

# display matrix picture with 0 and X
def matrix_overview(BB, bound):
    for ii in range(BB.dimensions()[0]):
        a = ('%02d ' % ii)
        for jj in range(BB.dimensions()[1]):
            a += '0' if BB[ii,jj] == 0 else 'X'
            if BB.dimensions()[0] < 60:
                a += ' '
        if BB[ii, ii] >= bound:
            a += '~'
        print(a)

# tries to remove unhelpful vectors
# we start at current = n-1 (last vector)
def remove_unhelpful(BB, monomials, bound, current):
    # end of our recursive function
    if current == -1 or BB.dimensions()[0] <= dimension_min:
        return BB

    # we start by checking from the end
    for ii in range(current, -1, -1):
        # if it is unhelpful:
        if BB[ii, ii] >= bound:
            affected_vectors = 0
            affected_vector_index = 0
            # let's check if it affects other vectors
            for jj in range(ii + 1, BB.dimensions()[0]):
                # if another vector is affected:
                # we increase the count
                if BB[jj, ii] != 0:
                    affected_vectors += 1
                    affected_vector_index = jj

            # level:0
            # if no other vectors end up affected
            # we remove it
            if affected_vectors == 0:
                print("* removing unhelpful vector", ii)
                BB = BB.delete_columns([ii])
```

```
            BB = BB.delete_rows([ii])
            monomials.pop(ii)
            BB = remove_unhelpful(BB, monomials, bound, ii-1)
            return BB

        # level:1
        # if just one was affected we check
        # if it is affecting someone else
        elif affected_vectors == 1:
            affected_deeper = True
            for kk in range(affected_vector_index + 1, BB.dimensions()[0]):
                # if it is affecting even one vector
                # we give up on this one
                if BB[kk, affected_vector_index] != 0:
                    affected_deeper = False
            # remove both it if no other vector was affected and
            # this helpful vector is not helpful enough
            # compared to our unhelpful one
            if affected_deeper and abs(bound - BB[affected_vector_index, affected_vector_index]) < abs(bound -
BB[ii, ii]):
                print("* removing unhelpful vectors", ii, "and", affected_vector_index)
                BB = BB.delete_columns([affected_vector_index, ii])
                BB = BB.delete_rows([affected_vector_index, ii])
                monomials.pop(affected_vector_index)
                monomials.pop(ii)
                BB = remove_unhelpful(BB, monomials, bound, ii-1)
                return BB
    # nothing happened
    return BB

"""
Returns:
* 0,0   if it fails
* -1,-1 if `strict=true`, and determinant doesn't bound
* x0,y0 the solutions of `pol`
"""
def boneh_durfee(pol, modulus, mm, tt, XX, YY):
    """
    Boneh and Durfee revisited by Herrmann and May

    finds a solution if:
    * d < N^delta
    * |x| < e^delta
    * |y| < e^0.5
    whenever delta < 1 - sqrt(2)/2 ~ 0.292
    """

    # substitution (Herrman and May)
    PR.<u, x, y> = PolynomialRing(ZZ)
    Q = PR.quotient(x*y + 1 - u) # u = xy + 1
    polZ = Q(pol).lift()

    UU = XX*YY + 1

    # x-shifts
    gg = []
    for kk in range(mm + 1):
        for ii in range(mm - kk + 1):
            xshift = x^ii * modulus^(mm - kk) * polZ(u, x, y)^kk
            gg.append(xshift)
    gg.sort()

    # x-shifts list of monomials
    monomials = []
    for polynomial in gg:
        for monomial in polynomial.monomials():
            if monomial not in monomials:
```

```
            monomials.append(monomial)
    monomials.sort()

    # y-shifts (selected by Herrman and May)
    for jj in range(1, tt + 1):
        for kk in range(floor(mm/tt) * jj, mm + 1):
            yshift = y^jj * polZ(u, x, y)^kk * modulus^(mm - kk)
            yshift = Q(yshift).lift()
            gg.append(yshift) # substitution

    # y-shifts list of monomials
    for jj in range(1, tt + 1):
        for kk in range(floor(mm/tt) * jj, mm + 1):
            monomials.append(u^kk * y^jj)

    # construct lattice B
    nn = len(monomials)
    BB = Matrix(ZZ, nn)
    for ii in range(nn):
        BB[ii, 0] = gg[ii](0, 0, 0)
        for jj in range(1, ii + 1):
            if monomials[jj] in gg[ii].monomials():
                BB[ii, jj] = gg[ii].monomial_coefficient(monomials[jj]) * monomials[jj](UU,XX,YY)

    # Prototype to reduce the lattice
    if helpful_only:
        # automatically remove
        BB = remove_unhelpful(BB, monomials, modulus^mm, nn-1)
        # reset dimension
        nn = BB.dimensions()[0]
        if nn == 0:
            print("failure")
            return 0,0

    # check if vectors are helpful
    if debug:
        helpful_vectors(BB, modulus^mm)

    # check if determinant is correctly bounded
    det = BB.det()
    bound = modulus^(mm*nn)
    if det >= bound:
        print("We do not have det < bound. Solutions might not be found.")
        print("Try with highers m and t.")
        if debug:
            diff = (log(det) - log(bound)) / log(2)
            print("size det(L) - size e^(m*n) = ", floor(diff))
        if strict:
            return -1, -1
    else:
        print("det(L) < e^(m*n) (good! If a solution exists < N^delta, it will be found)")

    # display the lattice basis
    if debug:
        matrix_overview(BB, modulus^mm)

    # LLL
    if debug:
        print("optimizing basis of the lattice via LLL, this can take a long time")

    BB = BB.LLL()

    if debug:
        print("LLL is done!")

    # transform vector i & j -> polynomials 1 & 2
    if debug:
```

```
        print("looking for independent vectors in the lattice")
    found_polynomials = False

    for pol1_idx in range(nn - 1):
        for pol2_idx in range(pol1_idx + 1, nn):
            # for i and j, create the two polynomials
            PR.<w,z> = PolynomialRing(ZZ)
            pol1 = pol2 = 0
            for jj in range(nn):
                pol1 += monomials[jj](w*z+1,w,z) * BB[pol1_idx, jj] / monomials[jj](UU,XX,YY)
                pol2 += monomials[jj](w*z+1,w,z) * BB[pol2_idx, jj] / monomials[jj](UU,XX,YY)

            # resultant
            PR.<q> = PolynomialRing(ZZ)
            rr = pol1.resultant(pol2)

            # are these good polynomials?
            if rr.is_zero() or rr.monomials() == [1]:
                continue
            else:
                print("found them, using vectors", pol1_idx, "and", pol2_idx)
                found_polynomials = True
                break
        if found_polynomials:
            break

    if not found_polynomials:
        print("no independant vectors could be found. This should very rarely happen...")
        return 0, 0



    rr = rr(q, q)

    # solutions
    soly = rr.roots()
    if len(soly) == 0:
        print("Your prediction (delta) is too small")
        return 0, 0
    soly = soly[0][0]
    ss = pol1(q, soly)
    solx = ss.roots()[0][0]
    return solx, soly

def example():
    # the modulus
    N =
0xf4c548636db62ffcc7ac4a0797952bea9a65bd426175af2435f72657e67ec8194667bfa94ce23c6f1e5baf320186
7ab41701f6b8768e71009c41a3d5e9e7c109455341d549c7611f9f52851a2f017906aa9ccbedb95d238468e2c8577
d30ecc4f158e3811fd5e2a6051443d468e3506bbc39bba710e34a604ac9e85d0feef8b3
    # the public exponent
    e =
0x16f4b438ba14e05afa944f7da9904f8c78ea52e4ca0be7fa2b5f84e22ddd7b0578a3477b19b7bb4a7f825acc45da
2dd10e62dbd94a3386b97d92ee817b0c66c1507514a7860b9139bc2ac3a4e0fe304199214da00a4ca82bfcb7b182
53e7e6144828e584dac2dfb9a03fabaf2376ce7c269923fbb60fc68325b9f6443e1f896f

    # the hypothesis on the private exponent (the theoretical maximum is 0.292)
    delta = .29 # this means that d < N^delta

    m = 8 # size of the lattice (bigger the better/slower)

    # you need to be a lattice master to tweak these
    t = int((1-2*delta) * m)  # optimization from Herrmann and May
    X = 2*floor(e^delta)  # this _might_ be too much
    Y = 2*floor(e^(1/2))    # correct if p, q are ~ same size
    # Problem put in equation
    P.<x,y> = PolynomialRing(ZZ)
```

```
    A = N
    pol =  x * (A - y)-1

    #
    # Find the solutions!
    #



    # boneh_durfee
    if debug:
        print("=== running algorithm ===")
        start_time = time.time()

    solx, soly = boneh_durfee(pol, e, m, t, X, Y)

    # found a solution?
    if solx > 0:
        print("=== solution found ===")
        if False:
            print("x:", solx)
            print("y:", soly)
        print(solx)
        print(soly)
        d = int(pol(solx, soly) / e)
        print("private key found:", d)
    else:
        print("=== no solution was found ===")

    if debug:
        print(("=== %s seconds ===" % (time.time() - start_time)))

if __name__ == "__main__":
    example()
```

第四题：

```
def _bm_6(N, e, r1,r2, m, t,tau):
    x,y, z = ZZ["x","y", "z"].gens()
    f = e*x-N*y+y*z-1
    X = 2**r2
    Y = int(2*RR(N)**(0.39022))
    Z = int(3 * RR(N) ** (1/2))
    W = e*2**r1
    for x0,y0, z0 in modular_trivariate(f, W, m, t, tau, X, Y , Z):
        phi = N - z0
        d = pow(e, -1, phi)
        print("d:")
        print(d)
        if pow(pow(2, e, N), d, N) == 2:
            print("d:")
            print(d)
            factors =factorize(N, phi)
            if factors:
                return *factors, d

    return None

def modular_trivariate(f, W, m, t, tau,X, Y, Z, roots_method="groebner"):
    f = f.change_ring(ZZ)
    pr = f.parent()
    x, y, z = pr.gens()

    shifts = []
    for t1 in range(m + 1):
        for t2 in range(m + 1):
```

```
            for t3 in range(m + 1):
                g = x ** t1 * y ** t2 * f ** t3 * W ** (m - t3)
                if(0<=t1+t2+t3<=m):
                  shifts.append(g)

    for i in range(1, tau+1):
        for t1 in range(m+1):
            for t3 in range(m+1):
                h = z ** i * x ** t1 * f ** t3 * W ** (m - t3)
                if(0<=t1+t3<=m):
                    shifts.append(h)

    L, monomials = create_lattice(pr, shifts, [X, Y, Z])
    L = reduce_lattice(L)
    polynomials = reconstruct_polynomials(L, f, W ** m, monomials, [X, Y, Z])
    for roots in find_roots_groebner(pr, polynomials)
        yield roots[x], roots[y], roots[z]

def factorize(N, phi):
    s = N + 1 - phi
    d = s ** 2 - 4 * N
    p = int(s - isqrt(d)) // 2
    q = int(s + isqrt(d)) // 2
    return p, q if p * q == N else None
def find_roots_groebner(pr, polynomials):
    gens = pr.gens()
    s = Sequence(polynomials, pr.change_ring(QQ, order="lex"))
    while len(s) > 0:
        G = s.groebner_basis()
        logging.debug(f"Sequence length: {len(s)}, Groebner basis length: {len(G)}")
        if len(G) == len(gens):
            logging.debug(f"Found Groebner basis with length {len(gens)}, trying to find roots...")
            roots = {}
            for polynomial in G:
                vars = polynomial.variables()
                if len(vars) == 1:
                    for root in find_roots_univariate(vars[0], polynomial.univariate_polynomial()):
                        roots = {**roots, **root}

            if len(roots) == pr.ngens():
                yield roots
                print(roots)
                return

            logging.debug(f"System is underdetermined, trying to find constant root...")
            G = Sequence(s, pr.change_ring(ZZ, order="lex")).groebner_basis()
            vars = tuple(map(lambda x: var(x), gens))
            for solution_dict in solve([polynomial(*vars) for polynomial in G], vars, solution_dict=True):
                logging.debug(solution_dict)
                found = False
                roots = {}
                for i, v in enumerate(vars):
                    s = solution_dict[v]
                    if s.is_constant():
                        if not s.is_zero():
                            found = True
                        roots[gens[i]] = int(s) if s.is_integer() else int(s) + 1
                    else:
                        roots[gens[i]] = 0
                if found:
                    yield roots
                    return

            return
        else:
            s.pop()
```

```
def create_lattice(pr, shifts, bounds, order="invlex", sort_shifts_reverse=False,
sort_monomials_reverse=False):

    if pr.ngens() > 1:
        pr_ = pr.change_ring(ZZ, order=order)
        shifts = [pr_(shift) for shift in shifts]

    monomials = set()
    for shift in shifts:
        monomials.update(shift.monomials())

    shifts.sort(reverse=sort_shifts_reverse)
    monomials = sorted(monomials, reverse=sort_monomials_reverse)
    L = matrix(ZZ, len(shifts), len(monomials))
    for row, shift in enumerate(shifts):
        for col, monomial in enumerate(monomials):
            L[row, col] = shift.monomial_coefficient(monomial) * monomial(*bounds)

    monomials = [pr(monomial) for monomial in monomials]
    return L, monomials


def reduce_lattice(L, delta=0.8):
    logging.debug(f"Reducing a {L.nrows()} x {L.ncols()} lattice...")
    return L.LLL(delta)


def reconstruct_polynomials(B, f, modulus, monomials, bounds, preprocess_polynomial=lambda x: x,
divide_gcd=True):

    divide_original = f is not None
    modulus_bound = modulus is not None
    logging.debug(f"Reconstructing polynomials ({divide_original = }, {modulus_bound = }, {divide_gcd
= })...")
    polynomials = []
    for row in range(B.nrows()):
        norm_squared = 0
        w = 0
        polynomial = 0
        for col, monomial in enumerate(monomials):
            if B[row, col] == 0:
                continue
            norm_squared += B[row, col] ** 2
            w += 1
            assert B[row, col] % monomial(*bounds) == 0
            polynomial += B[row, col] * monomial // monomial(*bounds)



        polynomial = preprocess_polynomial(polynomial)

        if divide_original and polynomial % f == 0:
            logging.debug(f"Original polynomial divides reconstructed polynomial at row {row}, dividing...")
            polynomial //= f

        if divide_gcd:
            for i in range(len(polynomials)):
                g = gcd(polynomial, polynomials[i])
                if g != 1 and g.is_constant():
                    logging.debug(f"Reconstructed polynomial has gcd {g} with polynomial at {i}, dividing...")
                    polynomial //= g
                    polynomials[i] //= g

        if polynomial.is_constant():
            logging.debug(f"Polynomial at row {row} is constant, ignoring...")
            continue
```

```
        if DEBUG_ROOTS is not None:
            logging.debug(f"Polynomial at row {row} roots check: {polynomial(*DEBUG_ROOTS)}")

        polynomials.append(polynomial)

    logging.debug(f"Reconstructed {len(polynomials)} polynomials")
    return polynomials

    N =
14917269868724734330748477442746394704043538593953831799557780293370835665974478130884965
81491994632704029460549590262470114966436097223810368834629936062084054544487937482828562
17226973570288117498818638210423816294135228225752144034736417495450129714250843040389723
69669132601706257568298912467717021277 4709
    e =
10321575536967147974494601992533919217013066046349360011274070671319877681362923424739141
13685086701949994408770625180341349926935851209082729569122250796564747437189111312489434
24182704537251516793612743605344558538258231516990650738302350887664785241741452204750976
04793630070181666080981938766830135570 6749

    m=4
    t=2
    tau=2
    for r2 in range(1,280):
        r1=r2+120
        result=_bm_6(N, e, r1,r2, m, t,tau)
        print(result)
```

第五题：

```
N =
10457299607549794131909181519260953703025474723566238558365855104066129316674627875566503
53453028412903210555510949988013491543417067943845693672205422111252084002419254298175372
33572712806366915693639454747762776346860624353939431050176100147487275981958992226231116
26433688406700726567792693618581993024 9121
e =
12849414952895198423852531624597162624746264880311312338450707412343893846568649235153233
31158662685101204664010425902511625764173338853300845440335923952476107039621725340354262
91450677560471955241983215901211916819840833873038760023523455723633175352792200061518956
68476230800333225447771855406768594758121
d1=78240989953785170095539221006587788208684073909320290992544625739340203214273
d3=16144942937992801419219495484979117548755795338679209
d3_length=176
d2_length=80
d1_length=256
d=2**512
beata = math.log(d, N)
m=3
M1=2**(d3_length+d2_length)
M2=2**d3_length
theat1=math.log(M1,N)
theat2=math.log(M2, N)
result=_bm_6(N, e, d1, d3, m, M1,M2,beata,theat1,theat2)
print(result)
def _bm_6(N, e, d1, d3, m, M1,M2,beata,theat1,theat2):
    k=2* (theat1) * m
    tau=1-2*(beata+theat1-theat2)
    x,y,z = ZZ["x", "y","z"].gens()
    L1 = math.floor(e * d1 * M1 / N)
    fem2 = 1-e*d3+(L1+x)*(N+y)
    fe = 1 + (L1+x) * (N+y)
    X = int(RR(N) ** (0.5-0.23))
    Y = int(RR(N)**(1/2))
    Z = int(RR(N)**(0.5+1/2))
    W=e*M2
    for x0, y0 ,z0 in modular_trivariate(fem2, fe, W, m, k,tau, X,Y,Z,e,M2,L1):
        phi = N + y0
```

```
        d = pow(e, -1, phi)
        print("d:")
        print(d)
        if pow(pow(2, e, N), d, N) == 2:
            print("d:")
            print(d)
            factors ==factorize(N, phi)
            if factors:
                return *factors, d

    return None

def LSB(i,k,tau):
    temp=math.ceil((i-k)/tau)
    largenumber=max(0,temp)
    return largenumber

def modular_trivariate(fem2,fe, W, m,k, tau, X, Y, Z ,e,M2,L1):
    fem2 = fem2.change_ring(ZZ)
    fe = fe.change_ring(ZZ)
    pr = fem2.parent()
    x, y,z= pr.gens()

shifts = []
    for u in range(m + 1):
        for i in range(u + 1):
                g = x ** (u-i) * fem2 ** i * (e*M2) ** (m-i)
                g = g.subs({(1+(L1+x)*y): z})
                shifts.append(g)
    for u in range(m + 1):
        for j in range(1,(math.floor(k+tau*u))):
                p=LSB(j,k,tau)
                g = y**j * fem2 ** (u-p) * fe ** p * e ** (m-u) * M2 ** (m-u+p)
                g = g.subs({(1+(L1+x)*y): z})
                shifts.append(g)
    L, monomials = create_lattice(pr, shifts, [X, Y,Z])
    L = reduce_lattice(L)
    polynomials = reconstruct_polynomials(L, fem2, W ** m, monomials, [X, Y,Z])
    for roots in find_roots_groebner (pr, polynomials):
        yield roots[x], roots[y],roots[z]
def factorize(N, phi):
    s = N + 1 - phi
    d = s ** 2 - 4 * N
    p = int(s - isqrt(d)) // 2
    q = int(s + isqrt(d)) // 2
    return p, q if p * q == N else None
def find_roots_groebner(pr, polynomials):
    gens = pr.gens()
    s = Sequence(polynomials, pr.change_ring(QQ, order="lex"))
    while len(s) > 0:
        G = s.groebner_basis()
        logging.debug(f"Sequence length: {len(s)}, Groebner basis length: {len(G)}")
        if len(G) == len(gens):
            logging.debug(f"Found Groebner basis with length {len(gens)}, trying to find roots...")
            roots = {}
            for polynomial in G:
                vars = polynomial.variables()
                if len(vars) == 1:
                    for root in find_roots_univariate(vars[0], polynomial.univariate_polynomial()):
                        roots = {**roots, **root}

            if len(roots) == pr.ngens():
                yield roots
                print(roots)
                return

            logging.debug(f"System is underdetermined, trying to find constant root...")
```

```
            G = Sequence(s, pr.change_ring(ZZ, order="lex")).groebner_basis()
            vars = tuple(map(lambda x: var(x), gens))
            for solution_dict in solve([polynomial(*vars) for polynomial in G], vars, solution_dict=True):
                logging.debug(solution_dict)
                found = False
                roots = {}
                for i, v in enumerate(vars):
                    s = solution_dict[v]
                    if s.is_constant():
                        if not s.is_zero():
                            found = True
                        roots[gens[i]] = int(s) if s.is_integer() else int(s) + 1
                    else:
                        roots[gens[i]] = 0
                if found:
                    yield roots
                    return

            return
        else:
            s.pop()

def create_lattice(pr, shifts, bounds, order="invlex", sort_shifts_reverse=False,
sort_monomials_reverse=False):

    if pr.ngens() > 1:
        pr_ = pr.change_ring(ZZ, order=order)
        shifts = [pr_(shift) for shift in shifts]

    monomials = set()
    for shift in shifts:
        monomials.update(shift.monomials())

    shifts.sort(reverse=sort_shifts_reverse)
    monomials = sorted(monomials, reverse=sort_monomials_reverse)
    L = matrix(ZZ, len(shifts), len(monomials))
    for row, shift in enumerate(shifts):
        for col, monomial in enumerate(monomials):
            L[row, col] = shift.monomial_coefficient(monomial) * monomial(*bounds)

    monomials = [pr(monomial) for monomial in monomials]
    return L, monomials


def reduce_lattice(L, delta=0.8):
    logging.debug(f"Reducing a {L.nrows()} x {L.ncols()} lattice...")
    return L.LLL(delta)


def reconstruct_polynomials(B, f, modulus, monomials, bounds, preprocess_polynomial=lambda x: x,
divide_gcd=True):

    divide_original = f is not None
    modulus_bound = modulus is not None
    logging.debug(f"Reconstructing polynomials ({divide_original = }, {modulus_bound = }, {divide_gcd
= })...")
    polynomials = []
    for row in range(B.nrows()):
        norm_squared = 0
        w = 0
        polynomial = 0
        for col, monomial in enumerate(monomials):
            if B[row, col] == 0:
                continue
            norm_squared += B[row, col] ** 2
            w += 1
            assert B[row, col] % monomial(*bounds) == 0
```

```
            polynomial += B[row, col] * monomial // monomial(*bounds)



        polynomial = preprocess_polynomial(polynomial)

        if divide_original and polynomial % f == 0:
            logging.debug(f"Original polynomial divides reconstructed polynomial at row {row}, dividing...")
            polynomial //= f

        if divide_gcd:
            for i in range(len(polynomials)):
                g = gcd(polynomial, polynomials[i])
                if g != 1 and g.is_constant():
                    logging.debug(f"Reconstructed polynomial has gcd {g} with polynomial at {i}, dividing...")
                    polynomial //= g
                    polynomials[i] //= g

        if polynomial.is_constant():
            logging.debug(f"Polynomial at row {row} is constant, ignoring...")
            continue

        if DEBUG_ROOTS is not None:
            logging.debug(f"Polynomial at row {row} roots check: {polynomial(*DEBUG_ROOTS)}")

        polynomials.append(polynomial)

    logging.debug(f"Reconstructed {len(polynomials)} polynomials")
    return polynomials
```

第六题

```
N =
104556698048600203626950165924925366165334476316915816649397417985813496266987449960811577093664367473788888518016834497934860317491855097332706561873673818512261986876973716463864153427030912390166859238341638205923415533401450135625243037527240632143380066610130842021437571861432446137749410332589024170613

e =
104354806659442032615995278408125030047663054103923038456358962497329608952981242065133260467880065624314483842629750226961268260007379542842119784436159064739136470818550766408787646655833181458621455153603177265630462735000992749122301250684285945010609426381819923048668179614515564136977857232552022228387
r = 437
d = 2**560
# Logarithm calculations
log_N = math.log(N)
log_e = math.log(e)
log_2 = math.log(2)

# Calculate 'a'
a = log_e / log_N
b = math.log(d, N)
# Calculate log_N(2^r)
theat = (r * log_2) / log_N

#result,d=attack(N, e, partial_d)

d_bit_length=560
d0_bit_length = 437
d0= 120542853124939
m=8
tau=3
```

```
theati=3
result=_bm_6(N, e, d_bit_length, d0, d0_bit_length, m, tau,theati,a,b,theat)
print(result)

def _bm_6(N, e, d_bit_length, d0, d0_bit_length,  m, tau,theati,a,b,theat):
    y, z, u = ZZ["y", "z","u"].gens()
    A = e*d0-1
    u = y*z+A
    f = (-1)*N*y + u

    W = int(RR(N)**(a+theat))
    #W = 2**d0_bit_length*e
    Y = int(2*RR(N)**(a+b-1))
    Z = int(2 * RR(N) ** (1/2))
    U = int(7*RR(N)**(a+b-0.5))
    logging.info(f"Trying {m = }, {tau = }...")
    for y0, z0, u0 in modular_trivariate(f, N, m, W ,A,theati, tau, Y, Z, U):
        phi = N - z0
        d = pow(e, -1, phi)
        print("d:")
        print(d)
        if pow(pow(2, e, N), d, N) == 2:
            print("d:")
            print(d)
            factors = factorize(N, phi)
            if factors:
                return *factors, d

    return None
def modular_trivariate(f, N, m, W ,A,theati, tau, Y, Z, U, roots_method="groebner"):
    f = f.change_ring(ZZ)
    pr = f.parent()
    y, z, u = pr.gens()

    logging.debug("Generating shifts...")
    Q = pr.quotient(y*z + A - u) # u = xy + 1
    polZ = Q(f).lift()

    shifts = []
    for t in range(m + 1):
        for j in range(t + 1):
            g = y ** (t - j) * f ** j * W ** (m-j)
            g = Q(g).lift()
            shifts.append(g)

    for i in range(1, tau + 1):
        for j in range(theati, m+1):
            h = z ** i * f ** j * W ** (m-j)
            h = Q(h).lift()
            shifts.append(h)

    L, monomials = create_lattice(pr, shifts, [Y, Z, U])
    L = reduce_lattice(L)
    polynomials =reconstruct_polynomials(L, f, W **m, monomials, [Y, Z, U], preprocess_polynomial=lambda
p: p(y, z, 1 + y * z))
    for roots in find_roots_groebner(pr, polynomials):
        yield roots[y], roots[z], roots[u]
def factorize(N, phi):
    s = N + 1 - phi
    d = s ** 2 - 4 * N
    p = int(s - isqrt(d)) // 2
    q = int(s + isqrt(d)) // 2
    return p, q if p * q == N else None
def find_roots_groebner(pr, polynomials):
    gens = pr.gens()
    s = Sequence(polynomials, pr.change_ring(QQ, order="lex"))
    while len(s) > 0:
```

```
        G = s.groebner_basis()
        logging.debug(f"Sequence length: {len(s)}, Groebner basis length: {len(G)}")
        if len(G) == len(gens):
            logging.debug(f"Found Groebner basis with length {len(gens)}, trying to find roots...")
            roots = {}
            for polynomial in G:
                vars = polynomial.variables()
                if len(vars) == 1:
                    for root in find_roots_univariate(vars[0], polynomial.univariate_polynomial()):
                        roots = {**roots, **root}

            if len(roots) == pr.ngens():
                yield roots
                print(roots)
                return

            logging.debug(f"System is underdetermined, trying to find constant root...")
            G = Sequence(s, pr.change_ring(ZZ, order="lex")).groebner_basis()
            vars = tuple(map(lambda x: var(x), gens))
            for solution_dict in solve([polynomial(*vars) for polynomial in G], vars, solution_dict=True):
                logging.debug(solution_dict)
                found = False
                roots = {}
                for i, v in enumerate(vars):
                    s = solution_dict[v]
                    if s.is_constant():
                        if not s.is_zero():
                            found = True
                        roots[gens[i]] = int(s) if s.is_integer() else int(s) + 1
                    else:
                        roots[gens[i]] = 0
                if found:
                    yield roots
                    return

            return
        else:
            s.pop()

def create_lattice(pr, shifts, bounds, order="invlex", sort_shifts_reverse=False,
sort_monomials_reverse=False):

    if pr.ngens() > 1:
        pr_ = pr.change_ring(ZZ, order=order)
        shifts = [pr_(shift) for shift in shifts]

    monomials = set()
    for shift in shifts:
        monomials.update(shift.monomials())

    shifts.sort(reverse=sort_shifts_reverse)
    monomials = sorted(monomials, reverse=sort_monomials_reverse)
    L = matrix(ZZ, len(shifts), len(monomials))
    for row, shift in enumerate(shifts):
        for col, monomial in enumerate(monomials):
            L[row, col] = shift.monomial_coefficient(monomial) * monomial(*bounds)

    monomials = [pr(monomial) for monomial in monomials]
    return L, monomials


def reduce_lattice(L, delta=0.8):
    logging.debug(f"Reducing a {L.nrows()} x {L.ncols()} lattice...")
    return L.LLL(delta)
```

```
def reconstruct_polynomials(B, f, modulus, monomials, bounds, preprocess_polynomial=lambda x: x,
divide_gcd=True):

    divide_original = f is not None
    modulus_bound = modulus is not None
    logging.debug(f"Reconstructing polynomials ({divide_original = }, {modulus_bound = }, {divide_gcd
= })...")
    polynomials = []
    for row in range(B.nrows()):
        norm_squared = 0
        w = 0
        polynomial = 0
        for col, monomial in enumerate(monomials):
            if B[row, col] == 0:
                continue
            norm_squared += B[row, col] ** 2
            w += 1
            assert B[row, col] % monomial(*bounds) == 0
            polynomial += B[row, col] * monomial // monomial(*bounds)



        polynomial = preprocess_polynomial(polynomial)

        if divide_original and polynomial % f == 0:
            logging.debug(f"Original polynomial divides reconstructed polynomial at row {row}, dividing...")
            polynomial //= f

        if divide_gcd:
            for i in range(len(polynomials)):
                g = gcd(polynomial, polynomials[i])
                if g != 1 and g.is_constant():
                    logging.debug(f"Reconstructed polynomial has gcd {g} with polynomial at {i}, dividing...")
                    polynomial //= g
                    polynomials[i] //= g

        if polynomial.is_constant():
            logging.debug(f"Polynomial at row {row} is constant, ignoring...")
            continue

        if DEBUG_ROOTS is not None:
            logging.debug(f"Polynomial at row {row} roots check: {polynomial(*DEBUG_ROOTS)}")

        polynomials.append(polynomial)

    logging.debug(f"Reconstructed {len(polynomials)} polynomials")
    return polynomials
```

第七题：

```
N =
122689918447277815594943686510739028484370044271504056551541462200471980751841624858210264536660464774499659750886314000284267567603472695526673695496895551351073804885392417528790967016144729996908982001503242120492551192253210865308554615583054581596696365790191179344735793019759201836850351629092137955949

e =
468466784735363134157793073508982702542147640218172043833853102422653308875352753190727952103970865386195202320748068870072174454077607340322584181108714630705278263770490644728515093392106685231694369917346854007151105339302953452581957846053904289490818009361213454266597983763511423546394892042567375580 29
m=8
t = int((1 - 2 * 0.27) * m)
result=_bm_6(N, e,  m,t)
print(result)
```

```
def _bm_6(N, e,  m,t):
    x, y = ZZ["x", "y"].gens()
    A = N
    f = x*(A-y)+1

    delta=0.27
    X = int(2*(N**delta))  # this _might_ be too much
    Y = int(2*(N**(1/2)))    # correct if p, q are ~ same size
    W = e
    #W = int(RR(N)**(a+theata))
    for x0, y0 in modular_bivariate(f, W, m, t, X, Y):
        phi = N-y0
        d = pow(e, -1, phi)
        if pow(pow(2, e, N), d, N) == 2:
            print("d:")
            print(d)
            factors =factorize(N, phi)
            if factors:
                return *factors, d

    return None
def modular_bivariate(f, e, m, t, X, Y):
    f = f.change_ring(ZZ)

    pr = ZZ["x", "y", "u"]
    x, y, u = pr.gens()
    qr = pr.quotient(1 + x * y - u)
    U = X * Y

    logging.debug("Generating shifts...")

    shifts = []
    for k in range(m + 1):
        for i in range(m - k + 1):
            g = x ** i * f ** k * e ** (m - k)
            g = qr(g).lift()
            shifts.append(g)

    for j in range(1, t + 1):
        for k in range(m // t * j, m + 1):
            h = y ** j * f ** k * e ** (m - k)
            h = qr(h).lift()
            shifts.append(h)

    L, monomials = create_lattice(pr, shifts, [X, Y, U])
    L = reduce_lattice(L)

    pr = f.parent()
    x, y = pr.gens()

    polynomials = reconstruct_polynomials(L, f, None, monomials, [X, Y, U], preprocess_polynomial=lambda
p: p(x, y, 1 + x * y))
    for roots in find_roots_groebner (pr, polynomials):
        yield roots[x], roots[y]
def factorize(N, phi):
    s = N + 1 - phi
    d = s ** 2 - 4 * N
    p = int(s - isqrt(d)) // 2
    q = int(s + isqrt(d)) // 2
    return p, q if p * q == N else None
def find_roots_groebner(pr, polynomials):
    gens = pr.gens()
    s = Sequence(polynomials, pr.change_ring(QQ, order="lex"))
    while len(s) > 0:
        G = s.groebner_basis()
        logging.debug(f"Sequence length: {len(s)}, Groebner basis length: {len(G)}")
        if len(G) == len(gens):
```

```
            logging.debug(f"Found Groebner basis with length {len(gens)}, trying to find roots...")
            roots = {}
            for polynomial in G:
                vars = polynomial.variables()
                if len(vars) == 1:
                    for root in find_roots_univariate(vars[0], polynomial.univariate_polynomial()):
                        roots = {**roots, **root}

            if len(roots) == pr.ngens():
                yield roots
                print(roots)
                return

            logging.debug(f"System is underdetermined, trying to find constant root...")
            G = Sequence(s, pr.change_ring(ZZ, order="lex")).groebner_basis()
            vars = tuple(map(lambda x: var(x), gens))
            for solution_dict in solve([polynomial(*vars) for polynomial in G], vars, solution_dict=True):
                logging.debug(solution_dict)
                found = False
                roots = {}
                for i, v in enumerate(vars):
                    s = solution_dict[v]
                    if s.is_constant():
                        if not s.is_zero():
                            found = True
                        roots[gens[i]] = int(s) if s.is_integer() else int(s) + 1
                    else:
                        roots[gens[i]] = 0
                if found:
                    yield roots
                    return

            return
        else:
            s.pop()

def create_lattice(pr, shifts, bounds, order="invlex", sort_shifts_reverse=False,
sort_monomials_reverse=False):

    if pr.ngens() > 1:
        pr_ = pr.change_ring(ZZ, order=order)
        shifts = [pr_(shift) for shift in shifts]

    monomials = set()
    for shift in shifts:
        monomials.update(shift.monomials())

    shifts.sort(reverse=sort_shifts_reverse)
    monomials = sorted(monomials, reverse=sort_monomials_reverse)
    L = matrix(ZZ, len(shifts), len(monomials))
    for row, shift in enumerate(shifts):
        for col, monomial in enumerate(monomials):
            L[row, col] = shift.monomial_coefficient(monomial) * monomial(*bounds)

    monomials = [pr(monomial) for monomial in monomials]
    return L, monomials


def reduce_lattice(L, delta=0.8):
    logging.debug(f"Reducing a {L.nrows()} x {L.ncols()} lattice...")
    return L.LLL(delta)


def reconstruct_polynomials(B, f, modulus, monomials, bounds, preprocess_polynomial=lambda x: x,
divide_gcd=True):

    divide_original = f is not None
```

```
    modulus_bound = modulus is not None
    logging.debug(f"Reconstructing polynomials ({divide_original = }, {modulus_bound = }, {divide_gcd
= })...")
    polynomials = []
    for row in range(B.nrows()):
        norm_squared = 0
        w = 0
        polynomial = 0
        for col, monomial in enumerate(monomials):
            if B[row, col] == 0:
                continue
            norm_squared += B[row, col] ** 2
            w += 1
            assert B[row, col] % monomial(*bounds) == 0
            polynomial += B[row, col] * monomial // monomial(*bounds)



        polynomial = preprocess_polynomial(polynomial)

        if divide_original and polynomial % f == 0:
            logging.debug(f"Original polynomial divides reconstructed polynomial at row {row}, dividing...")
            polynomial //= f

        if divide_gcd:
            for i in range(len(polynomials)):
                g = gcd(polynomial, polynomials[i])
                if g != 1 and g.is_constant():
                    logging.debug(f"Reconstructed polynomial has gcd {g} with polynomial at {i}, dividing...")
                    polynomial //= g
                    polynomials[i] //= g

        if polynomial.is_constant():
            logging.debug(f"Polynomial at row {row} is constant, ignoring...")
            continue

        if DEBUG_ROOTS is not None:
            logging.debug(f"Polynomial at row {row} roots check: {polynomial(*DEBUG_ROOTS)}")

        polynomials.append(polynomial)

    logging.debug(f"Reconstructed {len(polynomials)} polynomials")
    return polynomials
```

第八题：

```
N =
169363980487060707717369566653776389918374362966796797224050706518802635848129653437959801710237872741503170867835428492796911209009771528071095511503963814647068972370930967491940245744970118663777256526025997346495324320663818398526154636986701000928385328203032808969382507933130905332528902771966796618029

e =
966641142431017829954389146371316272684806572724224545040555741644783390509933612275773024031823172391218076033727326413586115156070878256683475230801862 3
r = 900
d = 2**1024
log_N = math.log(N)
log_e = math.log(e)
log_2 = math.log(2)
a = log_e / log_N
b = math.log(d, N)
theata = (r * log_2) / log_N
d_bit_length=1024
d0_bit_length = 900
```

```
d0=
25340028815017958137322910554549850092558257182497119145050726149537140091111638251015291
62499294515877582218716674980797301691097724080357476936079271514631246935707964795811604
05141451643152445999469346745308381903645145368566840248140559007348984551693442684109604
5703
m=9
tau=3
result=_bm_6(N, e, d_bit_length, d0, d0_bit_length,  m, tau,a,b,theata)
print(result)
def _bm_6(N, e, d_bit_length, d0, d0_bit_length,  m, tau,a,b,theata):
    y, z = ZZ["y", "z"].gens()
    A = e * d0-1
    f = A-N*y+y*z
    Y = int(2*RR(N)**(a+b-1))
        Z = int(3 * RR(N) ** (1/2))
        W = 2**(d0_bit_length)*e
        logging.info(f"Trying {m = }, {tau = }...")
        for y0, z0 in modular_bivariate(f, W, m, tau, Y, Z):
            phi = N - z0
            d = pow(e, -1, phi)
            print("d:")
            print(d)
            if pow(pow(2, e, N), d, N) == 2:
                print("d:")
                print(d)
                factors =factorize(N, phi)
                if factors:
                    return *factors, d

        return None
    def modular_bivariate(f, W, m, tau, Y, Z, roots_method="groebner"):
        f = f.change_ring(ZZ)
        pr = f.parent()
        y, z = pr.gens()
        #bounds=int(1/6*m*(m+2)+1/2*tau*m-m/(m+1))
        logging.debug("Generating shifts...")

        shifts = []
        for t in range(m + 1):
            for j in range(t + 1):
                g = y ** (t-j) *  f ** j * W ** (m-j)
                shifts.append(g)

        for i in range(1,tau + 1):
            for j in range(m + 1):
                h = z ** i *  f ** j * W ** (m-j)
                shifts.append(h)

        L, monomials = create_lattice(pr, shifts, [Y, Z])
        L = reduce_lattice(L)
        polynomials = reconstruct_polynomials(L, f, W**m, monomials, [Y, Z])
        for roots in find_roots_groebner (pr, polynomials):
            yield roots[y], roots[z]
def factorize(N, phi):
    s = N + 1 - phi
    d = s ** 2 - 4 * N
    p = int(s - isqrt(d)) // 2
    q = int(s + isqrt(d)) // 2
    return p, q if p * q == N else None
def find_roots_groebner(pr, polynomials):
    gens = pr.gens()
    s = Sequence(polynomials, pr.change_ring(QQ, order="lex"))
    while len(s) > 0:
        G = s.groebner_basis()
        logging.debug(f"Sequence length: {len(s)}, Groebner basis length: {len(G)}")
        if len(G) == len(gens):
            logging.debug(f"Found Groebner basis with length {len(gens)}, trying to find roots...")
```

```
            roots = {}
            for polynomial in G:
                vars = polynomial.variables()
                if len(vars) == 1:
                    for root in find_roots_univariate(vars[0], polynomial.univariate_polynomial()):
                        roots = {**roots, **root}

            if len(roots) == pr.ngens():
                yield roots
                print(roots)
                return

            logging.debug(f"System is underdetermined, trying to find constant root...")
            G = Sequence(s, pr.change_ring(ZZ, order="lex")).groebner_basis()
            vars = tuple(map(lambda x: var(x), gens))
            for solution_dict in solve([polynomial(*vars) for polynomial in G], vars, solution_dict=True):
                logging.debug(solution_dict)
                found = False
                roots = {}
                for i, v in enumerate(vars):
                    s = solution_dict[v]
                    if s.is_constant():
                        if not s.is_zero():
                            found = True
                        roots[gens[i]] = int(s) if s.is_integer() else int(s) + 1
                    else:
                        roots[gens[i]] = 0
                if found:
                    yield roots
                    return

            return
        else:
            s.pop()

def create_lattice(pr, shifts, bounds, order="invlex", sort_shifts_reverse=False, sort_monomials_reverse=False):

    if pr.ngens() > 1:
        pr_ = pr.change_ring(ZZ, order=order)
        shifts = [pr_(shift) for shift in shifts]

    monomials = set()
    for shift in shifts:
        monomials.update(shift.monomials())

    shifts.sort(reverse=sort_shifts_reverse)
    monomials = sorted(monomials, reverse=sort_monomials_reverse)
    L = matrix(ZZ, len(shifts), len(monomials))
    for row, shift in enumerate(shifts):
        for col, monomial in enumerate(monomials):
            L[row, col] = shift.monomial_coefficient(monomial) * monomial(*bounds)

    monomials = [pr(monomial) for monomial in monomials]
    return L, monomials


def reduce_lattice(L, delta=0.8):
    logging.debug(f"Reducing a {L.nrows()} x {L.ncols()} lattice...")
    return L.LLL(delta)


def reconstruct_polynomials(B, f, modulus, monomials, bounds, preprocess_polynomial=lambda x: x,
divide_gcd=True):

    divide_original = f is not None
    modulus_bound = modulus is not None
```

```
    logging.debug(f"Reconstructing polynomials ({divide_original = }, {modulus_bound = }, {divide_gcd
= })...")
    polynomials = []
    for row in range(B.nrows()):
        norm_squared = 0
        w = 0
        polynomial = 0
        for col, monomial in enumerate(monomials):
            if B[row, col] == 0:
                continue
            norm_squared += B[row, col] ** 2
            w += 1
            assert B[row, col] % monomial(*bounds) == 0
            polynomial += B[row, col] * monomial // monomial(*bounds)



        polynomial = preprocess_polynomial(polynomial)

        if divide_original and polynomial % f == 0:
            logging.debug(f"Original polynomial divides reconstructed polynomial at row {row}, dividing...")
            polynomial //= f

        if divide_gcd:
            for i in range(len(polynomials)):
                g = gcd(polynomial, polynomials[i])
                if g != 1 and g.is_constant():
                    logging.debug(f"Reconstructed polynomial has gcd {g} with polynomial at {i}, dividing...")
                    polynomial //= g
                    polynomials[i] //= g

        if polynomial.is_constant():
            logging.debug(f"Polynomial at row {row} is constant, ignoring...")
            continue

        if DEBUG_ROOTS is not None:
            logging.debug(f"Polynomial at row {row} roots check: {polynomial(*DEBUG_ROOTS)}")

        polynomials.append(polynomial)

    logging.debug(f"Reconstructed {len(polynomials)} polynomials")
    return polynomials
```

第九题：

```
N =
14350449507413511652347957251319325753845789197605229843865207992959665152343236493734193
09821730235521754361738856549309713769703229224983179764935620729261366598523449200098583
40197366796444840464302446464493305526983923226244799894266646253468068881999233902997176
32368444319764277312321391737257305060147 7
e = 65537
r = 530
d = 2**1024
log_N = math.log(N)
log_e = math.log(e)
log_2 = math.log(2)
a = log_e / log_N
b = math.log(d, N)
theata = (r * log_2) / log_N
d_bit_length=1024
d0_bit_length = 530
d0=
17617146364519807052255965154418246970340963048225666436979818980358870556588070204426629
24585355268098963915429014997296853529408546333631721472245329506038801
m=4
tau=1
result=_bm_6(N, e, d_bit_length, d0, d0_bit_length,  m, tau,a,b,theata)
```

```
print(result)
def _bm_6(N, e, d_bit_length, d0, d0_bit_length,  m, tau,a,b,theata):
    y, z = ZZ["y", "z"].gens()
    A = e * d0-1
    f = A-N*y+y*z
    Y = int(2*RR(N)**(a+b-1))
        Z = int(3 * RR(N) ** (1/2))
        W = 2**(d0_bit_length)*e
        logging.info(f"Trying {m = }, {tau = }...")
        for y0, z0 in modular_bivariate(f, W, m, tau, Y, Z):
            phi = N - z0
            d = pow(e, -1, phi)
            print("d:")
            print(d)
            if pow(pow(2, e, N), d, N) == 2:
                print("d:")
                print(d)
                factors =factorize(N, phi)
                if factors:
                    return *factors, d

        return None
    def modular_bivariate(f, W, m, tau, Y, Z, roots_method="groebner"):
        f = f.change_ring(ZZ)
        pr = f.parent()
        y, z = pr.gens()
        #bounds=int(1/6*m*(m+2)+1/2*tau*m-m/(m+1))
        logging.debug("Generating shifts...")

        shifts = []
        for t in range(m + 1):
            for j in range(t + 1):
                g = y ** (t-j) *  f ** j * W ** (m-j)
                shifts.append(g)

        for i in range(1,tau + 1):
            for j in range(m + 1):
                h = z ** i *  f ** j * W ** (m-j)
                shifts.append(h)

        L, monomials = create_lattice(pr, shifts, [Y, Z])
        L = reduce_lattice(L)
        polynomials = reconstruct_polynomials(L, f, W**m, monomials, [Y, Z])
        for roots in find_roots_groebner (pr, polynomials):
            yield roots[y], roots[z]
def factorize(N, phi):
    s = N + 1 - phi
    d = s ** 2 - 4 * N
    p = int(s - isqrt(d)) // 2
    q = int(s + isqrt(d)) // 2
    return p, q if p * q == N else None
def find_roots_groebner(pr, polynomials):
    gens = pr.gens()
    s = Sequence(polynomials, pr.change_ring(QQ, order="lex"))
    while len(s) > 0:
        G = s.groebner_basis()
        logging.debug(f"Sequence length: {len(s)}, Groebner basis length: {len(G)}")
        if len(G) == len(gens):
            logging.debug(f"Found Groebner basis with length {len(gens)}, trying to find roots...")
            roots = {}
            for polynomial in G:
                vars = polynomial.variables()
                if len(vars) == 1:
                    for root in find_roots_univariate(vars[0], polynomial.univariate_polynomial()):
                        roots = {**roots, **root}

            if len(roots) == pr.ngens():
```

```
            yield roots
            print(roots)
            return

        logging.debug(f"System is underdetermined, trying to find constant root...")
        G = Sequence(s, pr.change_ring(ZZ, order="lex")).groebner_basis()
        vars = tuple(map(lambda x: var(x), gens))
        for solution_dict in solve([polynomial(*vars) for polynomial in G], vars, solution_dict=True):
            logging.debug(solution_dict)
            found = False
            roots = {}
            for i, v in enumerate(vars):
                s = solution_dict[v]
                if s.is_constant():
                    if not s.is_zero():
                        found = True
                    roots[gens[i]] = int(s) if s.is_integer() else int(s) + 1
                else:
                    roots[gens[i]] = 0
            if found:
                yield roots
                return

        return
    else:
        s.pop()

def create_lattice(pr, shifts, bounds, order="invlex", sort_shifts_reverse=False, sort_monomials_reverse=False):

    if pr.ngens() > 1:
        pr_ = pr.change_ring(ZZ, order=order)
        shifts = [pr_(shift) for shift in shifts]

    monomials = set()
    for shift in shifts:
        monomials.update(shift.monomials())

    shifts.sort(reverse=sort_shifts_reverse)
    monomials = sorted(monomials, reverse=sort_monomials_reverse)
    L = matrix(ZZ, len(shifts), len(monomials))
    for row, shift in enumerate(shifts):
        for col, monomial in enumerate(monomials):
            L[row, col] = shift.monomial_coefficient(monomial) * monomial(*bounds)

    monomials = [pr(monomial) for monomial in monomials]
    return L, monomials


def reduce_lattice(L, delta=0.8):
    logging.debug(f"Reducing a {L.nrows()} x {L.ncols()} lattice...")
    return L.LLL(delta)


def reconstruct_polynomials(B, f, modulus, monomials, bounds, preprocess_polynomial=lambda x: x,
divide_gcd=True):

    divide_original = f is not None
    modulus_bound = modulus is not None
    logging.debug(f"Reconstructing polynomials ({divide_original = }, {modulus_bound = }, {divide_gcd
= })...")
    polynomials = []
    for row in range(B.nrows()):
        norm_squared = 0
        w = 0
        polynomial = 0
        for col, monomial in enumerate(monomials):
            if B[row, col] == 0:
```

```
            continue
        norm_squared += B[row, col] ** 2
        w += 1
        assert B[row, col] % monomial(*bounds) == 0
        polynomial += B[row, col] * monomial // monomial(*bounds)



    polynomial = preprocess_polynomial(polynomial)

    if divide_original and polynomial % f == 0:
        logging.debug(f"Original polynomial divides reconstructed polynomial at row {row}, dividing...")
        polynomial //= f

    if divide_gcd:
        for i in range(len(polynomials)):
            g = gcd(polynomial, polynomials[i])
            if g != 1 and g.is_constant():
                logging.debug(f"Reconstructed polynomial has gcd {g} with polynomial at {i}, dividing...")
                polynomial //= g
                polynomials[i] //= g

    if polynomial.is_constant():
        logging.debug(f"Polynomial at row {row} is constant, ignoring...")
        continue

    if DEBUG_ROOTS is not None:
        logging.debug(f"Polynomial at row {row} roots check: {polynomial(*DEBUG_ROOTS)}")

    polynomials.append(polynomial)

logging.debug(f"Reconstructed {len(polynomials)} polynomials")
return polynomials
```

第十题：

```
import itertools
def small_roots(f, bounds, m=1, d=None):
    if not d:
        d = f.degree()

    R = f.base_ring()
    N = R.cardinality()

    k = ZZ(f.coefficients().pop(0))
    g = gcd(k, N)
    k = R(k/g)

    f *= 1/k
    f = f.change_ring(ZZ)

    vars = f.variables()
    G = Sequence([], f.parent())
    for k in range(m):
        for i in range(m-k+1):
            for subvars in itertools.combinations_with_replacement(vars[1:], i):
                g = f**k * prod(subvars) * N**(max(d-k, 0))
                G.append(g)

    B, monomials = G.coefficient_matrix()
    monomials = vector(monomials)

    factors = [monomial(*bounds) for monomial in monomials]
    for i, factor in enumerate(factors):
        B.rescale_col(i, factor)

    B = B.dense_matrix().LLL()
```

```
    B = B.change_ring(QQ)
    for i, factor in enumerate(factors):
        B.rescale_col(i, Integer(1)/factor)

    H = Sequence([], f.parent().change_ring(QQ))
    for h in filter(None, B*monomials):
        H.append(h)
        I = H.ideal()
        if I.dimension() == -1:
            H.pop()
        elif I.dimension() == 0:
            roots = []
            for root in I.variety(ring=ZZ):
                root = tuple(R(root[var]) for var in f.variables())
                roots.append(root)
            return roots

    return []


c1=0x3b42fa3dc9089a21e9dabfe18297df47272f7e0ff59bf9bf16bc55e7fa70504c03fed56ca5ae93ac028f60ce5d
a3c145c6d181c5bd3c267288ec4765a19ca6b957b4535a1a185bd1b87d2e39b30e2430ed648175c29fdc1fde378
7c426783dd66ba17f98b42ba13a7b3532970d0aa31b5ffa5f3eae243337a1668bae456bfbfb
leak1=0x19ffe8024fcf0320b3107f380f2e7deff71d561c4266c0f439d1aca20cd43d2aa6aed8679a16b2e1d3ff4ba
3fc4da69cf34e35ead6f7eb79923960b9c83d9923e591b07b65275bf67f0b3d424cd7e6e6dd88ea39a5cfa27ecee6
1caaacc93e751dbb2a4c196f0ce0c36d44c35d6658d71b6c48b7b29400ab9161a0000000000
n1=0x8d0df1ce526c39f9b057de462778a61ceda2049c7e32ee99d40baa4b22b7fd438e9ca1dfd7467684625add2
52095ee97c698199f4c5991279f6d3e74d4c14d01d137d42722df0d4565ff2a5275f9cac66dc4dfdf3304f85cbdc3
d18eda1e32ac5d03675141a722ceefe0ea0533b53d7e50ed7eda1a1bbce47ed0ecb966f8678d
e1=65537


k1 = e1*leak1 // n1 + 1
PR.<x,y> = PolynomialRing(Zmod(e1*leak1))
f = 1 + k1*((n1+1)-y) - e1*x
bounds = (2^10,2^513)

res = small_roots(f,bounds,m=2,d=2)
leak = int(res[0][1])

PR.<x> = PolynomialRing(RealField(1000))
f = x*(leak-x) - n1
ph = int(f.roots()[0][0])


PR.<x> = PolynomialRing(Zmod(n1))
f = ph + x
res = f.small_roots(X=2^(160+6), beta=0.499,epsilon=0.02)[0]
p1 = int(ph + res)
q1 = n1 // p1
print(p1)
print(q1)
攻击，求出 p 和 q

p=84147670891338737694689450322242729025436170282367766217749984657082681139159059122107822
4624621783029655317730320816741301674210320660490381271483955120458 9
q=11771189496077447472415256585690278237732856915764247194903035840506904720003320569381951
6821289302271514693096409033531450133776467521063914733736528776 01
然后套用 RSA 解密脚本即可，求出 m 的值

n=0x8d0df1ce526c39f9b057de462778a61ceda2049c7e32ee99d40baa4b22b7fd438e9ca1dfd7467684625add25
2095ee97c698199f4c5991279f6d3e74d4c14d01d137d42722df0d4565ff2a5275f9cac66dc4dfdf3304f85cbdc3d
18eda1e32ac5d03675141a722ceefe0ea0533b53d7e50ed7eda1a1bbce47ed0ecb966f8678d
p=84147670891338737694689450322242729025436170282367766217749984657082681139159059122107822
4624621783029655317730320816741301674210320660490381271483955120458 9
q=11771189496077447472415256585690278237732856915764247194903035840506904720003320569381951
6821289302271514693096409033531450133776467521063914733736528776 01
```

```
c=0x3b42fa3dc9089a21e9dabfe18297df47272f7e0ff59bf9bf16bc55e7fa70504c03fed56ca5ae93ac028f60ce5da
3c145c6d181c5bd3c267288ec4765a19ca6b957b4535a1a185bd1b87d2e39b30e2430ed648175c29fdc1fde3787
c426783dd66ba17f98b42ba13a7b3532970d0aa31b5ffa5f3eae243337a1668bae456bfbfb
e=65537
from gmpy2 import *
phi=(p-1)*(q-1)
d=invert(e,phi)
m=pow(c,d,n)
print(m)
```

第十一题：

```
#from Crypto.Util.number import *
#from tqdm import *
import itertools

def small_roots(f, bounds, m=1, d=None):
    if not d:
        d = f.degree()

    R = f.base_ring()
    N = R.cardinality()

    k = ZZ(f.coefficients().pop(0))
    g = gcd(k, N)
    k = R(k/g)

    f *= 1/k
    f = f.change_ring(ZZ)

    vars = f.variables()
    G = Sequence([], f.parent())
    for k in range(m):
        for i in range(m-k+1):
            for subvars in itertools.combinations_with_replacement(vars[1:], i):
                g = f**k * prod(subvars) * N**(max(d-k, 0))
                G.append(g)

    B, monomials = G.coefficient_matrix()
    monomials = vector(monomials)

    factors = [monomial(*bounds) for monomial in monomials]
    for i, factor in enumerate(factors):
        B.rescale_col(i, factor)

    B = B.dense_matrix().LLL()
    B = B.change_ring(QQ)
    for i, factor in enumerate(factors):
        B.rescale_col(i, Integer(1)/factor)

    H = Sequence([], f.parent().change_ring(QQ))
    for h in filter(None, B*monomials):
        H.append(h)
        I = H.ideal()
        if I.dimension() == -1:
            H.pop()
        elif I.dimension() == 0:
            roots = []
            for root in I.variety(ring=ZZ):
                root = tuple(R(root[var]) for var in f.variables())
                roots.append(root)
            return roots

    return []
```

```
c1=0x84e4aa0be481e9c4bbd4c71dba5235cccd8312759de35c326c7e4cdda494196d1c0cae298240942af3082fac
215965999c908a79bf07e093ee0c402e727a09a1c1f13831875d66ebbc3f89507163de90339af055bcd7d7785747
75214accfbd8ae20001f27bc196b974cb3ac215fea3debb7b17a21a8ebb1a9880a671539ef21
leak1=0x4f77b72b04e6fb2d02e5a43edef4784a2e22df0d42bfc7c9093a58ec35eb21a11962103be960b0088d0cc
2e0dfb473bc2ba0a22cea1c73997442c8fab5e4bad22cd131055b0382eb9264ad40ec8257abaff11b33b173ffd016
8039bf40dc203eb325d884d2845fd2b5a37f41a0f64183db0c256c24450000000000000000000000
n1=0xcb5645c59c402b0edcf96cbd6a7308b64aac2f37a3c6f96be7c421c4b7f0a4adbdecd88cbea1128352fb21ba
ae583fe4ceb3fc93c4905803ad3e9214ada050d5c0ff785a13a5c9157c3154ad8d7015a2d239fe13ef836d3279c5cd
5dc96013ac40f372a9c9226d2f5fe73f312c56e11d9cdfbf9fb0db627ac1a752f5f0bd2b29
e1=65537


k1 = e1*leak1 // n1 + 1
PR.<x,y> = PolynomialRing(Zmod(e1*leak1))
f = 1 + k1*((n1+1)-y) - e1*x
bounds = (2^20,2^963)

res = small_roots(f,bounds,m=2,d=2)
leak = int(res[0][1])

PR.<x> = PolynomialRing(RealField(1000))
f = x*(leak-x) - n1
ph = int(f.roots()[0][0])


PR.<x> = PolynomialRing(Zmod(n1))
f = ph + x
res = f.small_roots(X=2^(220+6), beta=0.499,epsilon=0.02)[0]
p1 = int(ph + res)
q1 = n1 // p1
print(p1)
print(q1)
```

第十二题：

```
N =
112126112695822031597079043686641347096965614977031243692574328815291152925792114534327
818121529472111754548534713423735960454177872494310794091492785010018503347446737454324
090876853567778396904897722958311344630670170702319596083649159113766252985217344801903
445281846985013381344698564280453510050410298353
e = 65537
m=4
tau=2
a = 0.01
b = 0.99
theata = 0.49
result,d=_ernst_4_1_1(N, e, m, tau ,a,b,theata)
print(result)

def _ernst_4_1_1(N, e, m, tau,a,b,theata):

N_=910092353521253284372505374155384584387022788971302134985765940679051744612213073906
174156820918624766614266796734129771220437211599443395753058923116656756042904263631608
114821834896900204704646430063457220449522349591422740204307087004029783517981607229213
280792689840494237055376468729065847293529839778 72
    x, y, z = ZZ["x", "y", "z"].gens()
    f = e*N_ - e * x - N * y + y * z-1
    X = 2**513  # Equivalent to N^delta
    Y = e  # Equivalent to N^beta
    Z = int(3 * RR(N) ** (1 / 2))
    M = N * Y
    logging.info(f"Trying {m = }, {tau = }...")
    for x0, y0, z0 in integer_trivariate_1(f, m, tau, M, X, Y, Z):
        print(z0)
        phi = N - z0
        print(phi)
        d = pow(e, -1, phi)
        if pow(pow(2, e, N), d, N) == 2:
```

```
            print("hello")
            print(d)
            factors = factorize(N, phi)
            if factors:
                return *factors, d

    return None
def integer_trivariate_1(f, m, tau, M, X, Y, Z, check_bounds=True, roots_method="groebner"):
    pr = f.parent()
    x, y, z = pr.gens()

    #if check_bounds and RR(X) ** (1 + 3 * tau) * RR(Y) ** (2 + 3 * tau) * RR(Z) ** (1 + 3 * tau + 3 * tau
** 2) > RR(W) ** (1 + 3 * tau):
        #logging.debug(f"Bound check failed for {m = }, {t = }")
        #return


    A = int(f.constant_coefficient())
    assert A != 0
    while gcd(A, X) != 1:
        X += 1
    while gcd(A, Y) != 1:
        Y += 1
    while gcd(A, Z) != 1:
        Z += 1
    while gcd(A, M) != 1:
        M += 1

    R =M* X ** (m-1)*Y**(m-1) * Z ** (m + tau-1)
    f_ = (pow(A, -1, R) * f % R).change_ring(ZZ)

    logging.debug("Generating shifts...")

    shifts = []
    for j in range(m):
        for i2 in range(j + 1):
            for i3 in range(i2 + tau + 1):
                g = x ** (j-i2) * y ** i2 * z ** i3 * f_ * X ** (m - 1-j+i2) * Y ** (m - 1-i2) * Z ** (m-1+tau-i3)
                shifts.append(g)


    for i2 in range(m+1):
        for i3 in range(i2 + tau+1):
            g_ = x ** (m-i2) * y ** i2 * z ** i3 * R
            shifts.append(g_)


    L, monomials = create_lattice(pr, shifts, [X, Y, Z])
    L = reduce_lattice(L)
    polynomials = reconstruct_polynomials(L, f, R, monomials, [X, Y, Z])
    for roots in ind_roots_groebner (pr, [f] + polynomials):
        yield roots[x], roots[y], roots[z]
def factorize(N, phi):
    s = N + 1 - phi
    d = s ** 2 - 4 * N
    p = int(s - isqrt(d)) // 2
    q = int(s + isqrt(d)) // 2
    return p, q if p * q == N else None
def find_roots_groebner(pr, polynomials):
    gens = pr.gens()
    s = Sequence(polynomials, pr.change_ring(QQ, order="lex"))
    while len(s) > 0:
        G = s.groebner_basis()
        logging.debug(f"Sequence length: {len(s)}, Groebner basis length: {len(G)}")
        if len(G) == len(gens):
            logging.debug(f"Found Groebner basis with length {len(gens)}, trying to find roots...")
            roots = {}
```

```
            for polynomial in G:
                vars = polynomial.variables()
                if len(vars) == 1:
                    for root in find_roots_univariate(vars[0], polynomial.univariate_polynomial()):
                        roots = {**roots, **root}

            if len(roots) == pr.ngens():
                yield roots
                print(roots)
                return

            logging.debug(f"System is underdetermined, trying to find constant root...")
            G = Sequence(s, pr.change_ring(ZZ, order="lex")).groebner_basis()
            vars = tuple(map(lambda x: var(x), gens))
            for solution_dict in solve([polynomial(*vars) for polynomial in G], vars, solution_dict=True):
                logging.debug(solution_dict)
                found = False
                roots = {}
                for i, v in enumerate(vars):
                    s = solution_dict[v]
                    if s.is_constant():
                        if not s.is_zero():
                            found = True
                        roots[gens[i]] = int(s) if s.is_integer() else int(s) + 1
                    else:
                        roots[gens[i]] = 0
                if found:
                    yield roots
                    return

            return
        else:
            s.pop()

def create_lattice(pr, shifts, bounds, order="invlex", sort_shifts_reverse=False,
sort_monomials_reverse=False):

    if pr.ngens() > 1:
        pr_ = pr.change_ring(ZZ, order=order)
        shifts = [pr_(shift) for shift in shifts]

    monomials = set()
    for shift in shifts:
        monomials.update(shift.monomials())

    shifts.sort(reverse=sort_shifts_reverse)
    monomials = sorted(monomials, reverse=sort_monomials_reverse)
    L = matrix(ZZ, len(shifts), len(monomials))
    for row, shift in enumerate(shifts):
        for col, monomial in enumerate(monomials):
            L[row, col] = shift.monomial_coefficient(monomial) * monomial(*bounds)

    monomials = [pr(monomial) for monomial in monomials]
    return L, monomials


def reduce_lattice(L, delta=0.8):
    logging.debug(f"Reducing a {L.nrows()} x {L.ncols()} lattice...")
    return L.LLL(delta)


def reconstruct_polynomials(B, f, modulus, monomials, bounds, preprocess_polynomial=lambda x: x,
divide_gcd=True):

    divide_original = f is not None
    modulus_bound = modulus is not None
```

```
    logging.debug(f"Reconstructing polynomials ({divide_original = }, {modulus_bound = }, {divide_gcd
= })...")
    polynomials = []
    for row in range(B.nrows()):
        norm_squared = 0
        w = 0
        polynomial = 0
        for col, monomial in enumerate(monomials):
            if B[row, col] == 0:
                continue
            norm_squared += B[row, col] ** 2
            w += 1
            assert B[row, col] % monomial(*bounds) == 0
            polynomial += B[row, col] * monomial // monomial(*bounds)



        polynomial = preprocess_polynomial(polynomial)

        if divide_original and polynomial % f == 0:
            logging.debug(f"Original polynomial divides reconstructed polynomial at row {row}, dividing...")
            polynomial //= f

        if divide_gcd:
            for i in range(len(polynomials)):
                g = gcd(polynomial, polynomials[i])
                if g != 1 and g.is_constant():
                    logging.debug(f"Reconstructed polynomial has gcd {g} with polynomial at {i}, dividing...")
                    polynomial //= g
                    polynomials[i] //= g

        if polynomial.is_constant():
            logging.debug(f"Polynomial at row {row} is constant, ignoring...")
            continue

        if DEBUG_ROOTS is not None:
            logging.debug(f"Polynomial at row {row} roots check: {polynomial(*DEBUG_ROOTS)}")

        polynomials.append(polynomial)

    logging.debug(f"Reconstructed {len(polynomials)} polynomials")
    return polynomials
```