All exercise numbers from the textbook refer to the second edition.

1. Suppose that Alice and Bob are using NTRU with $N = 251$, $q = 131$, $p = 3$, and $d = 6$. How many bits are needed to represent the plaintext? How many bits are needed to represent the ciphertext? What is the message expansion ratio?

**Solution.** As discussed in class, an NTRU plaintext $m = \sum_{i=0}^{N-1} m_i x^i$ corresponds to a choice of $N$ coefficients, each from 0 to $p - 1$ inclusive. This in turn can be naturally encoded as an integer from 0 to $p^N - 1$, namely $\sum_{i=0}^{N-1} m_i p^i$ (where the integers $m_i$ serve as the digits in base $p$). Hence, if the plaintext is stored and processed as an integer from 0 to $p^N - 1$, it requires $\log_2 p^N = N \log_2 p$ bits (technically, this should be rounded up).

Similarly, the ciphertext requires $N \log_2 q$ bits to store and transmit. Thus the message expansion is $\log_2 q / \log_2 p = \log_p q$ (up to a small correction due to these logarithms not being integers).

In this case, the plaintext requires $251 \log_2 3 = 397.8$ bits (so it can be stored with 398 bits, but can only really encode 397 bits of information, since not all 398-bit numbers are less than $p^N$), while the ciphertext requires $251 \log_2 131 = 1765.3$ bits (so it can be stored with 1766 bits). The message expansion is roughly $\log_3 131 \approx 4.4$.

Note that both answers will change if a different encoding system is employed. For example, if the ciphertext is transmitted more literally as a sequence of 251 integers (the coefficients), then each coefficient will require at least 8 bits (since $2^7 \leq 131 < 2^8$), hence $8 \cdot 251 = 2008$ bits would be needed in this case. So the answer depends on the assumptions made about how the data is encoded for transmission, but 1766 is the theoretical lower bound on the number of bits that must be sent to specify the ciphertext uniquely.

2. Exercise 7.23, parts (a),(b),(c).

**Solution.**

(a)

\[
(1 + x) \star (-5 + 4x + 2x^3) = -5 + 4x + 2x^2
\]

\[
-5x + 4x^2 + 2
\]

\[
= -3 + x + 6x^2
\]

\[
\equiv 4 + 6x + 6x^2 \pmod{7}
\]

(b)

\[
(2 + 2x - 2x^2 + x^3 - 2x^4) \star (-1 + 3x - 3x^2 - 3x^3 - 3x^4)
\]

\[
\equiv (2 + 2x + 2x^2 + x^3 + 2x^4)
\]

\[
\star (-1 + x + x^2 + x^3 + x^4) \pmod{4}
\]

\[
\equiv -2 - 2x + 2x^2 + 2x^3 + 2x^4
\]

\[
-2x - 2x^2 + 2x^3 + 2x^4 + 2
\]

\[
-2x^2 - 2x^3 + 2x^4 + 2 + 2x
\]

\[
-x^3 - x^4 + 1 + x + x^2
\]

\[
-2x^4 - 2 + 2x + 2x^2 + 2x^3 \pmod{4}
\]

\[
\equiv 1 + x + x^2 + 3x^3 + 3x^4 \pmod{4}
\]
Problem Set 11
Math 158, Fall 2016

3. Exercise 7.24 (note that this exercise explains why \( f \) is chosen from \( T(d+1,d) \) in NTRU, rather than from \( T(d,d) \) like \( r \) and \( g \)).

Solution.

(a) Let \( a(x) \in \mathbb{F}_q[x] \) be any polynomial. By Proposition 2.44 (division with remainder in polynomial rings over a field), there exists a quotient \( k(x) \) and remainder \( r(x) \) (both elements of \( \mathbb{F}_q[x] \)) such that 
\[ a(x) \equiv k(x)(x-1) + r(x) \pmod{q} \]
and \( \deg r(x) < \deg(x-1) \). So this remainder must be degree \( \leq 0 \), i.e. \( r(x) \) is just a constant (possibly 0); call this constant \( r_0 \).

Now, note that this means that 
\[ a(1) \equiv k(1)(1-1) + r_0 \equiv r_0 \pmod{q} \]
So \( a(1) \equiv 0 \pmod{q} \) if and only if \( r_0 \equiv 0 \pmod{q} \). Now, if \( r_0 \equiv 0 \), then in fact \( a(x) \equiv k(x)(x-1) \pmod{q} \), so \( x-1 \) divides \( a(x) \) in \( R_q \).

Conversely, if \( x-1 \) divides \( a(x) \) in \( R_q \), then it is possible to write 
\[ a(x) \equiv (x-1)k(x) \pmod{q} \]
for some \( k(x) \), so \( a(1) \equiv (1-1)k(1) \equiv 0 \pmod{q} \). So in fact \( x-1 \) divides \( a(x) \) in \( R_q \) if and only if \( a(1) \equiv 0 \pmod{q} \).

(b) Note that \( 1^N - 1 = 0 \), so \( x-1 \) divides \( X^N - 1 \), by part (a). Therefore, if \( a(1) \equiv 0 \pmod{q} \), then \( x-1 \) is a factor of both \( a(x) \) and \( X^N - 1 \). Therefore \( \gcd(a(x),X^N-1) \neq 1 \), since at least \( x-1 \) is a common factor. So \( a(x) \) (when reduced modulo \( X^N - 1 \) to obtain an element of \( R_q \)) is not invertible.

4. Exercise 7.25.

Solution.

Using the Euclidean algorithm, we can compute the gcd of each polynomial with \( X^5 - 1 \) as follows (note that we are regarding these polynomials as elements of \( \mathbb{F}_3[x] \) now, rather than elements of \( R_3 \)).

First, we apply the Euclidean algorithm to \( 1 + x^2 + x^3 \) and \( x^5 - 1 \), as elements of \( \mathbb{F}_3[x] \). Note that since we are working with coefficients in \( \mathbb{F}_3 \), I will regard \(-1\) as being the same as \( 2 \) (since \(-1 \equiv 2 \pmod{3}\)). This makes some of the arithmetic a bit easier.

All congruences written below are \( \pmod{3} \). I omit the \( \pmod{3} \) to reduce clutter.

\[
\begin{align*}
r_1(x) &\equiv (x^5 - 1) - (x^2 - x + 1)(x^3 + x^2 + 1) \\
&\equiv x^2 + x + 1 \\
r_2(x) &\equiv (x^3 + x^2 + 1) - (x)(x^2 + x + 1) \\
&\equiv -x + 1 \\
r_3(x) &\equiv (x^2 + x + 1) - (-x + 1)(-x + 1) \\
&\equiv 0
\end{align*}
\]
5. Exercise 7.30.

Solution.

(a) The ciphertext can be computed as follows. I have done these computations using code from the programming portion (for arithmetic in $R_{29}$). All congruences in this part are modulo 29, which I omit for brevity. The operation $\star$ is convolution of polynomials (multiplication modulo $x^7 - 1$).

$$m(x) + ph(x) \star r(x) \equiv (1 + x - x^2 - x^3 - x^6) + 3(1 + x - x^2 - 3x^3 + 6x^4 + 2x^5 + 7x^6) \star (-1 + x^2 - x^5 + x^6)$$

$$\equiv (1 + x - x^2 - x^3 - x^6) + 3(17 + 5x + 26x^3 + 22x^4 + 15x^5 + 5x^6)$$

$$\equiv (1 + x - x^2 - x^3 - x^6) + (22 + 15x + 20x^2 + 8x^3 + 20x^4 + 16x^5 + 15x^6)$$

$$\equiv 23 + 16x + 19x^2 + 7x^3 + 20x^4 + 16x^5 + 14x^6$$

Here, the answer is reported with coefficients reduced modulo 29. Alternatively, one could report the answer in centerlifted form, i.e. $-6 - 13x - 10x^2 + 7x^3 - 9x^4 - 13x^5 + 14x^6$.

(b) First, we multiply $f$ by $e$ (working in $R_{29}$, i.e. reducing coefficients modulo 29).

$$(-1 + x - x^2 + x^4 + x^6) \star (23 + 16x + 19x^2 + 7x^3 + 20x^4 + 16x^5 + 14x^6)$$
\[ \equiv 27 + 3x + 26x^2 + x^3 + 7x^4 + 27x^5 + 24x^6 \pmod{29} \]

This should now be centerlifted modulo \( q = 29 \). In the notation of the table 7.4, the resulting polynomial is called \( a \).

\[ a = -2 + 3x - 3x^3 + x^4 + 7x^5 - 2x^5 - 5x^6 \]

We know this polynomial should be equal on the nose to \( f \ast \text{cl}_p(m) + pg \ast r \) (where \( \text{cl}_p \) denotes the centerlift at \( p \)). So we can reduce it modulo \( p \) to strip off the part involving \( r \), obtaining

\[ f \ast \text{cl}_p(m) \equiv 1 + x^3 + x^4 + x^5 + x^6 \pmod{3} \]

Now multiply by \( F_3 = 1 + x + x^2 + x^4 + x^5 - x^6 \) to obtain

\[ \text{cl}_p(m) \equiv (1 + x + x^2 + x^4 + x^5 - x^6) \ast (1 + x^3 + x^4 + x^5 + x^6) \pmod{3} \]

\[ \equiv 1 + x + 2x^2 + 2x^3 + 2x^6 \pmod{3} \]

Therefore, in centerlifted form, the plaintext \( m \) is \( 1 + x - x^2 - x^3 - x^6 \), which is indeed the plaintext that we started with.

6. Exercise 7.35 (the second part should be labeled part (b)).

(a) Multiplying by \( f \) gives \( f \ast e \equiv f \ast m + g \ast r \pmod{q} \), and as long as \( q \) is sufficiently large, the centerlift at \( q \) of either side will be exactly equal to \( f \ast \text{cl}_p(m) + g \ast r \), because the coefficients of the latter polynomial will all be less than \( q/2 \) in absolute value. So if \( a \in R \) is defined to be the centerlift of \( f \ast e \), then \( a = f \ast \text{cl}_p(m) + g \ast r \), as an element of \( R \).

Furthermore, since \( f \equiv 1 \pmod{p} \) and \( g \equiv 0 \pmod{p} \), it follows that in fact \( a \equiv 1 \ast m + 0 \ast r \equiv m \pmod{p} \). So reducing \( a \) modulo \( p \) gives the plaintext \( m \) immediately.

(b) What is needed is that \( f \ast \text{cl}_p(m) + g \ast r \) is equal to its own centerlift modulo \( q \), i.e. that all of its coefficients are less than \( q/2 \) in absolute value. In other words, we must show that

\[ |f \ast \text{cl}_p(m) + g \ast r|_\infty < q/2. \]

Expanding the definitions of \( f \) and \( g \) and applying the triangle inequality, we have

\[ |f \ast \text{cl}_p(m) + g \ast r|_\infty = |\text{cl}_p(m) + p f_0 \ast \text{cl}_p(m) + p g_0 \ast r|_\infty \]

\[ \leq |\text{cl}_p(m)|_\infty + p|f_0 \ast \text{cl}_p(m)|_\infty + p|g_0 \ast r|_\infty \]

And then using the lemma from class (stating that if \( a \in R \) has only \( \pm 1 \) and 0 as coefficients, with at most \( d \) of them nonzero, and \( b \) is any element of \( R \), then \( |a \ast b|_\infty \leq d \cdot |b|_\infty \) we know that \( f_0, g_0 \in T(d, d) \) implies that \( |f_0 \ast \text{cl}_p(m)|_\infty \leq 2d \cdot |\text{cl}_p(m)|_\infty \) and \( |g_0 \ast r|_\infty \leq 2d |r|_\infty \), since both \( f_0 \) and \( g_0 \) have only \( \pm 1 \) and 0 as coefficients, with at most \( 2d \) of them nonzero in each case. Since \( m \) is assumed to be ternary, and \( r \) is also assumed to be ternary, it follows that \( |f_0 \ast \text{cl}_p(m)|_\infty \leq 2d \) and \( |g_0 \ast r|_\infty \leq 2d \). Combining these bounds gives the inequality

\[ |f \ast \text{cl}_p(m) + g \ast r|_\infty \leq 1 + 2dp + 2dp = 1 + 4dp. \]
Therefore, we are guaranteed that $|f \ast c_p(m) + g \ast r|_\infty < q/2$ as long as the parameters are chosen so that

$$1 + 4dp < q/2$$

which is equivalent to $q > 8dp + 2$. As long as this bound holds, decryption is guaranteed to succeed, as desired.

7. Exercise 7.45.

Solution.

(a) Reduction of this basis takes 7 iterations (including the final one, in which we compute $m = 0$ and discover that we are finished) to find the shortest vector $(14, -47)$. The steps are shown in the following automatically generated output.

```python
>>> reduce(((120670, 110521),(323572, 296358))

Reducing the basis
v = (120670,110521)
w = (323572,296358).

Iteration 1:
Computed m = 3 by rounding 71799215758/26776140341 to the nearest integer.
Replacing w by w - 3 v = (-38438,-35205)

Iteration 2:
w is shorter than v; swapping.
Computed m = -3 by rounding -8529205265/2716871869 to the nearest integer.
Replacing w by w + 3 v = (5356,4906)

Iteration 3:
w is shorter than v; swapping.
Computed m = -7 by rounding -378589658/52755572 to the nearest integer.
Replacing w by w + 7 v = (-946,-863)

Iteration 4:
w is shorter than v; swapping.
Computed m = -6 by rounding -9300654/1639685 to the nearest integer.
Replacing w by w + 6 v = (-320,-272)

Iteration 5:
w is shorter than v; swapping.
Computed m = 3 by rounding 537456/176384 to the nearest integer.
Replacing w by w - 3 v = (14,-47)

Iteration 6:
w is shorter than v; swapping.
Computed m = 3 by rounding 8304/2405 to the nearest integer.
Replacing w by w - 3 v = (-362,-131)

Iteration 7:
Computed m = 0 by rounding 1089/2405 to the nearest integer.
The basis is now reduced, and v=(14,-47) is the short vector.
((14, -47), (-362, -131))

(b) Reducing this basis takes 8 iterations (counting the last one, in which no replacement occurs), to find the short vector $(147, 330)$.

Reducing the basis
(174748650,45604569)
(35462559,9254748).

Iteration 1:
  w is shorter than v; swapping.
  Computed m = 5 by rounding \( \frac{6619093104538962}{1343243451371985} \) to the nearest integer.
  Replacing w by \( w - 5v = (-2564145, -669171) \)

Iteration 2:
  w is shorter than v; swapping.
  Computed m = -14 by rounding \( \frac{-97124152320963}{70226294308266} \) to the nearest integer.
  Replacing w by \( w + 14v = (-435471, -113646) \)

Iteration 3:
  w is shorter than v; swapping.
  Computed m = 6 by rounding \( \frac{1192659394761}{202550405157} \) to the nearest integer.
  Replacing w by \( w - 6v = (48681, 12705) \)

Iteration 4:
  w is shorter than v; swapping.
  Computed m = -9 by rounding \( \frac{-22643036181}{2531256786} \) to the nearest integer.
  Replacing w by \( w + 9v = (2658, 699) \)

Iteration 5:
  w is shorter than v; swapping.
  Computed m = 18 by rounding \( \frac{138274893}{7553565} \) to the nearest integer.
  Replacing w by \( w - 18v = (837, 123) \)

Iteration 6:
  w is shorter than v; swapping.
  Computed m = 3 by rounding \( \frac{2310723}{715698} \) to the nearest integer.
  Replacing w by \( w - 3v = (147, 330) \)

Iteration 7:
  w is shorter than v; swapping.
  Computed m = 1 by rounding \( \frac{163629}{130509} \) to the nearest integer.
  Replacing w by \( w - 1v = (690, -207) \)

Iteration 8:
  Computed m = 0 by rounding \( \frac{33120}{130509} \) to the nearest integer.
  The basis is now reduced, and v=(147,330) is the short vector.

(c) Reducing this basis takes 12 iterations (including the one where \( m = 0 \) is found) to reach the short vector (4690,126).

Reducing the basis
  (725734520,613807887)
  (3433061338,2903596381).

Iteration 1:
  Computed m = 5 by rounding \( \frac{4273741481586444707}{903450715663035169} \) to the nearest integer.
  Replacing w by \( w - 5v = (-195611262, -165443054) \)

Iteration 2:
  w is shorter than v; swapping.
  Computed m = -4 by rounding \( \frac{-243512096728731138}{65635169938079560} \) to the nearest integer.
  Replacing w by \( w + 4v = (-56710528, -47964329) \)

Iteration 3:
  w is shorter than v; swapping.

Computed $m = 3$ by rounding $19028583023587102/5516660842459025$ to the nearest integer. 
Replacing $w$ by $w - 3 \, v = (-25479678, -21550067)$

Iteration 4: 
$w$ is shorter than $v$; swapping. 
Computed $m = 2$ by rounding $2478600496210027/1113619378688173$ to the nearest integer. 
Replacing $w$ by $w - 2 \, v = (-5751172, -4864195)$

Iteration 5: 
$w$ is shorter than $v$; swapping. 
Computed $m = 4$ by rounding $251361738833681/56736372371609$ to the nearest integer. 
Replacing $w$ by $w - 4 \, v = (-2474990, -2093287)$

Iteration 6: 
$w$ is shorter than $v$; swapping. 
Computed $m = 2$ by rounding $24416249347245/10507425964469$ to the nearest integer. 
Replacing $w$ by $w - 2 \, v = (-801192, -677621)$

Iteration 7: 
$w$ is shorter than $v$; swapping. 
Computed $m = 3$ by rounding $3401397418307/1101078840505$ to the nearest integer. 
Replacing $w$ by $w - 3 \, v = (-71414, -60424)$

Iteration 8: 
$w$ is shorter than $v$; swapping. 
Computed $m = 11$ by rounding $98160896792/8751019172$ to the nearest integer. 
Replacing $w$ by $w - 11 \, v = (-15638, -12957)$

Iteration 9: 
$w$ is shorter than $v$; swapping. 
Computed $m = 5$ by rounding $1899685900/412430893$ to the nearest integer. 
Replacing $w$ by $w - 5 \, v = (6776, 4361)$

Iteration 10: 
$w$ is shorter than $v$; swapping. 
Computed $m = -3$ by rounding $-162468565/64932497$ to the nearest integer. 
Replacing $w$ by $w + 3 \, v = (4690, 126)$

Iteration 11: 
$w$ is shorter than $v$; swapping. 
Computed $m = 1$ by rounding $32328926/22011976$ to the nearest integer. 
Replacing $w$ by $w - 1 \, v = (2086, 4235)$

Iteration 12: 
Computed $m = 0$ by rounding $10316950/22011976$ to the nearest integer. 
The basis is now reduced, and $v = (4690, 126)$ is the short vector. 

8. In some implementations of NTRU, rather than fixing one public parameter $d$, one chooses three different parameters $d_1, d_2, d_3$, and stipulates that Alice chooses $f$ from $T(d_1 + 1, d_1)$ and $g$ from $T(d_2, d_2)$, and Bob chooses $r$ from $T(d_3, d_3)$. Assuming that $d_1 \geq d_2 \geq d_3$, determine an inequality of the form $q > \cdots$ to replace the inequality $q > (6d + 1)p$ in table 7.3, serving the same purpose in this more general formulation (your inequality should specialize to $q > (6d + 1)p$ in the case $d_1 = d_2 = d_3 = d$).

Solution. Decryption will work as long as the polynomial $f \ast \text{cl}_p(m) + pg \ast r$ equals its own centerlift modulo $q$. In other words, we require that $|f \ast \text{cl}_p(m) + pg \ast r|_\infty < q/2$. 

Due the night of Sunday 12/11 (hard deadline 4am on 12/12).
A lemma proved in class showed that if \( a \in \mathbb{R} \) is ternary (all coefficients ±1 or 0) and at most \( d \) of its coefficients are nonzero, then for any \( b \in \mathbb{R} \) it follows that \( |a \ast b|_\infty \leq d \cdot |b|_\infty \). It follows from this that

\[
|f \ast c_{p}(m)|_\infty \leq (2d_1 + 1)|c_{p}(m)|_\infty \leq (2d_1 + 1) \cdot \frac{p}{2} = \frac{1}{2}(2d_1 + 1)p
\]

and

\[
|g \ast r|_\infty \leq 2d_3|g|_\infty = 2d_3.
\]

Note that in the second inequality I am taking \( r \) as the polynomial \( a \) from the statement of the lemma above, since we have assumed that \( d_3 \leq d_2 \), hence fewer of \( r \)'s coefficients are nonzero. One can also obtain \( |g \ast r|_\infty \leq 2d_2 \) by taking \( a = g \) in the statement of the lemma, but this may be a weaker bound.

Combining these two bounds with the triangle inequality gives

\[
|f \ast c_{p}(m) + pg \ast r|_\infty \leq |f \ast c_{p}(m)|_\infty + p|g \ast r|_\infty
\]

\[
\leq \frac{1}{2}(2d_1 + 1)p + 2d_3p
\]

\[
\leq \frac{1}{2}(2d_1 + 4d_3 + 1)p
\]

Therefore we can guarantee that decryption will succeed by stipulating that

\[
\frac{1}{2}(2d_1 + 4d_3 + 1)p < \frac{1}{2}q
\]

or equivalently \( q > (2d_1 + 4d_3 + 1)p \). Note that this specializes to \( q > (6d + 1)p \) in the case \( d_1 = d_2 = d_3 \). Note also that no role at all is played by \( d_2 \) in this bound, so it could be fact be chosen as large as desired without impacting the guarantee of successful decryption.

Programming problems

Full formulation and submission: [https://www.hackerrank.com/m158-2016-pset-11](https://www.hackerrank.com/m158-2016-pset-11)

9. Analyze a sequence of proposed “transactions” in a basic cryptocurrency, based on ECDSA signatures with the elliptic curve secp256k1. See the online statement for a complete description of the rules underlying the system and the format of the transactions.

Solution.

The main ingredient in the code is code for ECDSA verification (see problem set 9, problem 7). Using this, together with the function to generate a hash value from two public keys (provided in the starter code), we can process the transactions one by one as follows:

- Maintain a list of \( N \) items, telling the public key owning each particular coin (in order).
- For each transaction, first check that the proposed sender is an owner (i.e. it appears in the owner list somewhere) and that the proposed recipient is not an owner.
- If these first two conditions hold, then compute a hash value from the two public keys, and verify that the provided signature is valid, using the sender’s public key as a verification key.
- If the signature is valid, then find the place in the owner list where the sender appears, and replace that list element with the recipient.
An implementation is below.

```python
### Omitted: source code for inverse(a,m) (inverse of a modulo m),
### and methods add(P,Q,A,B,p) and mult(P,n,A,B,p) for elliptic curve arithmetic.

# Code for ECDSA verification, including the curve parameters.
def parseHex(hexString):
    return int(hexString.replace(' ',''),16)

def verify(d,s1,s2,V):
    # Parameters for Secp256k1, from Bitcoin wiki
    p = parseHex('FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF
    FFFFFFFF FFFFFFFF FFFFFFFE FFFFFC2F')
    q = parseHex('FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFE
    BAAEDCE6 AF48A03B BFD25E8C D0364141')
    A = 0
    B = 7
    Gx = parseHex('79BE667E F9DCBBAC 55A06295 CE870B07
    029BFCD 2DCE28D9 59F2815B 16F81798')
    Gy = parseHex('483ADA77 26A3C465 5DA4FBFC 0E1108A8
    FD17B448 A6855419 9C47D08F FB10D488')
    G = (Gx,Gy)
    w1 = inverse(s2,q)*d % q
    w2 = inverse(s2,q)*s1 % q
    Q = add(mult(G,w1,A,B,p),mult(V,w2,A,B,p),A,B,p)
    return Q[0]%q == s1

def hashval(A,B,C,D):
    m = hashlib.sha256()
    m.update('%d %d %d %d'%(A,B,C,D))
    return int(m.hexdigest(),16)

def process(transaction, owners):
    x1,y1,x2,y2,s1,s2 = transaction
    if (x1,y1) not in owners:
        return # Sender isn't an owner
    if (x2,y2) in owners:
        return # Recipient is already an owner
    d = hashval(x1,y1,x2,y2)
    if not(verify(d,s1,s2,(x1,y1))):
        return # Signature is invalid
    # If we've reached this point, it's a valid transaction
    i = owners.index((x1,y1))
    owners[i] = (x2,y2)
```

Due the night of Sunday 12/11 (hard deadline 4am on 12/12).
### I/O

N, T = map(int, raw_input().split())
owners = []
for n in range(N):
    x, y = map(int, raw_input().split())
    owners += [(x, y)]
for t in range(T):
    transaction = map(int, raw_input().split())
    process(transaction, owners)

for x, y in owners:
    print x, y

10. Implement functions to add or multiply two polynomials, or to centerlift or compute the inverse of a single element, in the ring $R_q$, where element are represented as lists of length $N$ consisting of integers from 0 to $q - 1$ inclusive.

**Solution.**

Here is an implementation of these four functions, making use of the functions for arithmetic in $F_p[x]$ implemented last time.

```python
def RQ_add(a, b, q):
    assert(len(a) == len(b))
    N = len(a)
    return [(ai+bi)%q for ai, bi in zip(a, b)]

def RQ_mult(a, b, q):
    assert(len(a) == len(b))
    N = len(a)
    return [sum([a[j]*b[(i-j)%N] for j in xrange(N)]) % q for i in xrange(N)]

def RQ_inv(a, q):
    N = len(a)
    u, v, g = FPX_euclid(trim(a), [-1] + [0]*(N-1) + [1], q)
    if len(g) != 1:
        return None
    res = u + [0]*(N-len(u))  # Extend to the correct length
    # Check that this is indeed the inverse
    assert tuple(RQ_mult(a, res, q)) == tuple([1] + [0]*(N-1))
    return res

def RQ_centerlift(a, q):
    cl = a[:]  # Duplicates the array
    for i in xrange(len(a)):
cl[i] %= q
if cl[i] > q/2:
    cl[i] -= q
return cl

11. Decipher an NTRU ciphertext $e$, given the parameters $N, q, p, d$, and the private elements $f, g$ (notation as in table 7.4).

We can follow the steps of table 7.4 to implement decryption as follows.

### Omitted: source code for all functions in the previous problem.

def decipher(e,f,g,q,p):
    N = len(e)
    Fp = RQ_inv(f,p)
    a = RQ_centerlift( RQ_mult(e,f,q), q )
    return RQ_mult(a,Fp,p)

### Omitted: input/output processing.

12. Cryptanalyze the congruential cryptosystem (prototype for NTRU) from section 7.1 (described in table 7.1). You will be given the public parameter $q$, a public key $h$ (but not the private key), and a ciphertext $e$; you must determine the plaintext $m$. The recommended method is to implement Gauss’s lattice basis reduction algorithm from section 7.13.1 of the textbook, which we will discuss in class on Friday. Half the test cases will also have small enough $q$ to be susceptible to a more brute-force attack.

Solution.

The solution divides into two parts: we must implement a function `decipher(e,f,g,q)` that deciphers the ciphertext $e$ given the private key $(f, g)$ and the parameter $q$; and we must implement a function to extract the private key from the public key $h$. The first part amounts to following the instructions in table 7.1. The second part requires implementing Gauss’s lattice basis reduction algorithm. For this, we need a couple helper functions: to compute dot products of vectors (represented as either tuples or lists of two numbers), and a function to round a quotient of two integers to the nearest integer.

One technical detail that may arise at the end of this process is that Gauss’s algorithm does not prefer vectors with positive coordinates to vectors with negative coordinates (if they have the same length). So it’s possible that, when we reduce the basis $(h, 1), (q, 0)$ using Gauss’s algorithm, we’ll end up not with $(g, f)$, but with $(-g, -f)$ (both are tied for “shortest nonzero vector”). So we need to check after running Gauss’s algorithm whether the coordinates are positive or negative, and flip them if necessary.

An implementation is below.

```python
# Dot product of 2-dimensional vectors
def dot(a,b):
    return a[0]*b[0] + a[1]*b[1]

# Rounds m/n to the nearest integer
```
def round(m,n):
    return (m+n/2)/n

# Gauss's algorithm for basis reduction
def reduce(v,w):
    while True:
        if dot(w,w) < dot(v,v): v,w = w,v
        m = round(dot(v,w),dot(v,v))
        if m==0: break
        w = (w[0]-m*v[0],w[1]-m*v[1])
    return v,w

# Steals the private key from the public key
def extractKey(h,q):
    v,w = reduce( (h,1), (q,0) )
    g,f = v
    # Coords might be negative; flip them if so
    if f < 0: f,g = -f,-g
    return f,g

# Deciphers a message, given the public key
def decipher(e,f,g,q):
    a = f*e % q
    return a * inverse(f,g) % g

### I/O
q,h = map(int,raw_input().split())
e = int(raw_input())
f,g = extractKey(h,q)
print (f*e % q) * inverse(f,g) % g