Written problems

1. Read example 8.10 in the textbook, about a simple *blind signature scheme* based on RSA.

   (a) Assume that the document D to be signed is a unit modulo N (we saw in PSet 8, #6 that this is extremely likely to hold). Prove that the element D' is uniformly distributed in $\mathbf{Z}/N$, i.e. that any value is just as probable as any other. This shows that Samantha cannot possibly learn any information about D from D'.

   (b) Suppose that Samantha uses the same public key (N, e) to receive encrypted messages as she uses for the blind signatures, and that Alice has intercepted a ciphertext C meant for Samantha. Show that, by requesting a blind signature, Alice can learn the plaintext corresponding to C. Explain why Samantha has no way to detect that Alice is doing this.

   Remark. This does not necessarily mean that the RSA blind signature scheme should never be used in practice (as part of a carefully designed protocol), but it does mean that Samantha should not use the same public key for both encryption and blind signing.

   Solution.

   (a) Let a be any element of $\mathbf{Z}/N$. We must show that $\Pr(D' \equiv a \pmod N) = \frac{1}{N}$. Since this probability does not depend on D', this will show that the value of D' is uniformly distributed. Observe that $D' \equiv a$ if and only if $R^e D \equiv a$, which is true if and only if $R^e \equiv D^{-1}a \pmod N$ (this is where we must assume that D is a unit).

   At this stage, we must use the fact that e is part of an RSA public key. In particular, this means that there is a decrypting exponent d, with the property that $x^e \equiv y \pmod N$ if and only if $x \equiv y^d \pmod N$ (that is, exponentiating by e and exponentiating by d are inverse functions). Therefore $R^e \equiv D^{-1}a \pmod N$ if and only if $R \equiv \left(D^{-1}a\right)^d \pmod N$. Since R is chosen uniformly at random, the probability that it is exactly $\left(D^{-1}a\right)^d$ is $\frac{1}{N}$. Thus $\frac{1}{N}$ is also equal to the probability that $D' \equiv a \pmod N$, as desired.

   (b) An RSA signature on C is the same thing as the decryption of C: they are both $C^d \pmod N$. Therefore Alice can send Samantha $R^e C$ for a blind signature, and obtain the plaintext $C^d \pmod N$ by the process in the book. Samantha cannot detect this because, by part (a), the number she received from Alice is just a uniformly distributed random number from 0 to $N-1$; she cannot obtain any information about at all about what Alice is having her sign.

2. Let P be a point of order N on an elliptic curve over a finite field. Prove that, if m, n are any two integers, the two points $m \cdot P$ and $n \cdot P$ are equal if and only if $m \equiv n \pmod N$.

   Solution.

   Suppose that $mP = nP$. Assume without loss of generality that $m \geq n$. Then adding the inverse of P n times shows that $(m-n) \cdot P = \mathcal{O}$. We know that $N \cdot P = \mathcal{O}$, therefore

$[(m-n)\%N]\cdot P = \mathcal{O}$ (since we have merely added a multiple of $N\cdot P$, the identity, to both sides). The remainder $(m-n)\%N$ is strictly less than N. By definition of order, N is the *smallest* postive integer giving $\mathcal{O}$ when multiplied by P; it follows that $(m-n)\%N$ cannot be positive; it must be 0. Therefore $m \equiv n \pmod{N}$.

Conversely, if $m \equiv n \pmod{N}$, then mP and nP differ by addition of $N\cdot P$ some number of times. Since $N\cdot P$ is the identity, this means that $m\cdot P = n\cdot P$.

3. (a) Consider the elliptic curve $Y^2 = X^3 + 7$. Using your code (or the solution, or someone else's code) for computing the trace of Frobenius (PSet 8, # 15), compute the number $t_p/\sqrt{p}$ (as a floating-point number) for this curve, for each prime p between 1000 and 2000, and store the results in a list. Draw a histogram of the resulting numbers (you may choose the number of bins as you see fit), and describe briefly the distibution of these values.

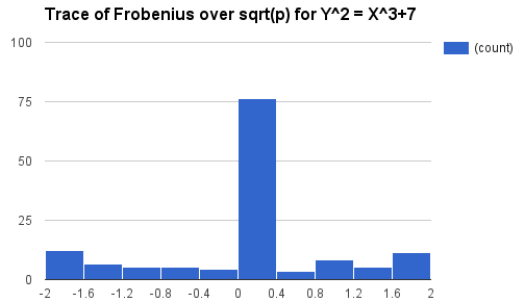
   (b) Do the same thing, but now for the curve $Y^2 = X^4 + 7$ (you will need to modify the trace of Frobenius code slightly since this doesn't have the form $Y^2 = X^3 + AX + B$).

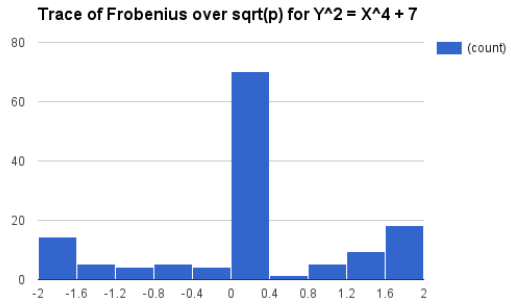
   (c) Do the same thing, but now for the curve $Y^2 = X^5 + 7$. What is the main difference you notice between this distribution and the distributions you saw in parts (a) and (b)?

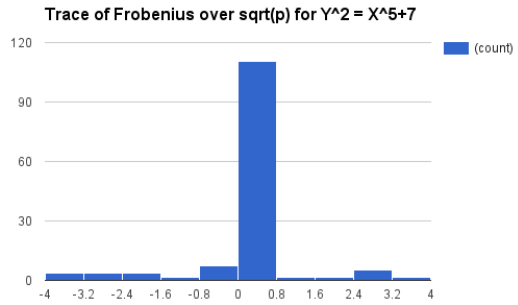
Remark. The curve in part (b) is also an elliptic curve, although it's equation hasn't been expressed in our usual form. The curve in part (c) is not; it is an example of a *hyperelliptic curve*; some basics about these curves are discussed in section 8.10.

Solution.

Three histograms for $t_p/\sqrt{p}$ are shown below; each breaks the data into 10 buckets.







In all three cases, there is a large spike at $t_p = 0$ (which appears just off-center in these pictures since 0 is included in the bucket beginning at 0, not the bucket ending at 0), and a distribution that spreads over an interval, slightly flaring up at the ends.

The main qualitative shift between parts (a),(b) and (c) is the width of the distribution: in parts (a) and (b) the distribution ends abruptly at ± 2, while in part (c) it ends abruptly at ± 4.

In general, if we do a similar point count for any curve defined by a polynomial in two variables, we will find a distibution abruptly ending at $\pm 2g$, where g is an integer called the *genus* of the curve. Elliptic curves (e.g. in part (a), and also, it turns out, in (b)) have genus 1. In contrast, a straight line has genus 0, which explains why a straight line always has exactly $p + 1$ points on it $\pmod p$ (counting the point at infinity). The genus is related in a fascinating way to the topology of the curve when viewed as a set of pairs of *complex* numbers.

4. Textbook exercise 6.14.

Solution.

(a) Bob should send $n_B \cdot P$ to Alice. Using a program to compute elliptic curve arithmetic, this value is $1943 \cdot (1980, 431) = (1432, 667)$.

(b) The shared secret is $n_B \cdot Q_A = 1943 \cdot (2110, 543) = (2424, 911)$.

(c) When p is only 2671, it is trivial to find n_A with a computer: trial and error is fast enough. For example, using the elliptic curve arithmetic code displayed before the coding problem solutions, I can find it as follows. Note that I've chosen to run the `for` loop up to 3000 since this is definitely larger than $p + 2\sqrt{p}$, hence definitely larger than the order of the point P.

```
>>> C = curve(171,853,2671)
>>> P = ec_pt(C,(1980,431))
>>> QA = ec_pt(C,(2110,543))
>>> for i in range(3000):
...     if (i*P).coords == QA.coords: print i
...
726
2045
>>>
```

In fact, we found two "logarithms:" 726 and 2045. This shows that the order of P must be $2045 - 726 = 1319$. In any case, we can conclude that $n_A = 726$ (or is congruent to it modulo the order of P, which is good enough).

(d) Bob computes $n_B \cdot P = 875 \cdot (1980, 431) = (161, 2040)$ and sends Alice only the x-coordinate 161.

Bob can infer that a possible y-coordinate of Alice's point Q_A is a square root of $2^3 + 171 \cdot 2 + 853 \equiv 1203 \pmod p$. Since $p \equiv 3 \pmod 4$, we can apply proposition 2.26 to quickly find a square root, as $y_A \equiv 1203^{(p+1)/4} \equiv 1203^{2672/4} \equiv 2575 \pmod p$. So Alice's point is $(2, \pm 2575)$, but Bob cannot say which for sure.

To obtain the shared value, Bob can compute $n_B \cdot (2, 2575) = (1708, 1419)$ and extract the x-coordinate 1708, which is the shared secret. If we had used $(2, -2575)$ instead, he would have obtained the inverse point (since $n_B(\ominus Q_A) = \ominus n_B \cdot Q_A$), which has the same x-coordinate. So the shared secret is definitiely 1708.

5. Textbook exercise 6.16.

Solution.

(a) If (x, y) is one point on an elliptic curve, then the only other point with the same x-coordinate is $(x, -y)$, since any square has only two square roots modulo p (or one square root of the square is 0). Assuming that $x_R^3 + Ax_R + B \not\equiv 0 \pmod{p}$ (in which case, there is no ambiguity about y_R), the two possible points are (x_R, y_R) and $(x_R, p - y_R)$. One of the numbers $y_R, p - y_R$ is less than $\frac{1}{2}p$ and one is greater than or equal to it (we are assuming $p \neq 2$ implicitly here), so β_R uniquely specifies which one is which.

(b) Bob can compute that $x_R^3 + Ax_r + B \equiv 216 \pmod{p}$, and then obtain a square root by computing $216^{(p+1)/4} \equiv 487 \pmod{p}$ (by proposition 2.26). This is less than $\frac{1}{2}p$, and $\beta_R = 0$, so Alice concludes that $(278, 487)$ is the point that Bob has in mind.

If Alice had instead said $\beta_R = 1$, then Bob would know that her point is $(278, 636)$, since $636 \equiv -487 \pmod{p}$.

6. Textbook exercise 6.17.

(a) The point T is $n_A \cdot R$, and in turn it is $(n_A k) \cdot P = k \cdot (n_A \cdot P) = k \cdot Q_A = S$. Therefore $x_T = x_S$ and $y_T = y_S$. Therefore $c_1 \equiv x_T m_1$ and $c_2 \equiv y_T m_2 \pmod{p}$. Multiplying by x_T^{-1} and $y_T - 1 \pmod{p}$ this shows that $m_1' \equiv m_1$ and $m_2' \equiv m_2 \pmod{p}$.

(b) The plaintext is two integers modulo p, and the ciphertext is one point on the elliptic curve plus two integers modulo p. Since the elliptic curve point has two coordinates, both integers modulo p, the ciphertext amounts to four integers modulo p. So the message expansion is two-to-one.

Of course, Alice and Bob could agree to use the technique from the previous problem to compress the size of the point R be about half. So if this modification is performed, the message expansion becomes only three-to-two, or 1.5-to-one.

(c) Alice's public key is $595 \cdot (278, 285) = (1104, 492)$. To decrypt $((1147, 640), 289, 1189)$, Alice computes $T = n_A \cdot R = 595 \cdot (1147, 640) = (942, 476)$, and then finds the plaintext as

$$\begin{aligned} m_1 &\equiv 942^{-1} 279 \pmod{1201} \\ &\equiv 509 \\ m_2 &\equiv 476^{-1} 1189 \pmod{1201} \\ &\equiv 767 \end{aligned}$$

So the plaintext is $(509, 767)$.

7. Textbook exercise 6.18. *Revision:* in part (b), a short list of possible values is an acceptable answer. In practice this is just as good as getting the plaintext on the nose, since Eve can examine each possibility and see which one has the right format (e.g. is in English).

Solution.

(a) Since (x_S, y_S) is a point on the elliptic curve, it follows that

$$(m_2^{-1}c_2)^2 \equiv (m_1^{-1}c_1)^3 + A(m_1^{-1}c_1) + B \pmod{p}.$$

Both c_1 and c_2 are known. As long as one of m_1, m_2 is known (e.g. because it is a common header to all messages that Bob sends to Alice), then the congruence above is a polynomial congruence for the other (technically, for its inverse, but finding the inverse is good enough since inversion modulo p is efficient).

(b) In this case, $x_S \equiv m_1^{-1}c_1 \equiv 1050^{-1}814 \equiv 957 \pmod{p}$. Therefore:

$$(m_2^{-1}1050)^2 \equiv 957^3 + 19 * 957 + 17 \equiv 697 \pmod{p}.$$

We could turn this into an equation for m_2 as follows.

$$\begin{aligned} 697 &\equiv m_2^{-2}1050^2 \pmod{p} \\ m_2^2 &\equiv 1050^2 697^{-1} \pmod{p} \\ &\equiv 815. \end{aligned}$$

There are efficient ways to compute square roots modulo p (even in cases where $p \not\equiv 3 \pmod 4$, so that proposition 2.26 does not apply), which Eve would use in practice. For the purposes of this assignment, we can just find the square root by trial and error:

$$m_2 \equiv 179, 1022 \pmod{p}.$$

So the second part of the plaintext is either 179 or 1022. It is not possible to determine for sure which of these two it is using the information given. In practice, Eve could examine them both to see which matches the expected format for a secret message (e.g. written in English), and keep the one that looks real.

8. Suppose that Samantha has published an ECDSA verification key as described on page 322 of the textbook. Suppose that d, d' are two different documents (i.e. $d \not\equiv d' \pmod q$), and that Samantha signs both of them using the same random element e, resulting in signatures (s_1, s_2) and (s_1', s_2'). Prove that Eve can efficiently extract Alice's secret signing key s from these two signed documents.

Solution.

We know from the first signing equiation that $s_1 = s_1'$ (since the same value of e was used). We know from the second signing equation that the following two modulo-q congruences hold.

$$\begin{aligned} e \cdot s_2 - s \cdot s_1 &\equiv d \pmod{q} \\ e \cdot s_2' - s \cdot s_1 &\equiv d' \pmod{q} \end{aligned}$$

Eve knows every term in these congruences, except e and s. The congruences form a linear system, which she can solve by any of the conventional methods. For example, she could write this as a congruence of matrices

$$\begin{pmatrix} s_2 & -s_1 \\ s_2' & -s_1 \end{pmatrix} \begin{pmatrix} e \\ s \end{pmatrix} = \begin{pmatrix} d \\ d' \end{pmatrix}$$

and observe that the determinant of this matrix is $s_1(s_2' - s_2) \pmod{q}$. Therefore the matrix is invertible $\pmod{q}$ if and only if $s_1 \not\equiv 0 \pmod{q}$ and $s_2 \not\equiv s_2' \pmod{q}$. The first condition is essentially certain since s_1 behaves like a random number, and the second condition follows from the assumption that $d \not\equiv d' \pmod{q}$. Therefore both e and s can now be recoevered by Eve by inverting this matrix $\pmod{q}$ and multiplying by the vector on the right.

Programming problems

Note. Problems 9 and 11 below will make use of public parameters specified in this document.

`http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.pdf`

You certainly do not need to read and understand the whole document, just find the information you need for the algorithms. You'll need to look up how to convert hexadecimal strings to integers.

Note for the following solutions.

In the following solution, I will make use of the following code for elliptic curve arithmetic. It uses a bit of Python's object-oriented capabilities. I've avoided doing this until now to keep the syntax to a minimum, but I think it was be interesting to you to see how to do this sort of thing.

```
# Class for an elliptic curve over a finite field (in Weierstrass form).
class curve:
        def __init__(self,A,B,p):
                self.A = A
                self.B = B
                self.p = p
        def add(self,P,Q):
                if P==0: return Q
                if Q==0: return P
                if P==Q:
                        x = P[0]
                        y = P[1]
                        rise = (3*x*x + self.A) % self.p
```

```
                        run = (2*y) % self.p
                else:
                        rise = (P[1]-Q[1])%self.p
                        run = (P[0]-Q[0])%self.p
                if run == 0:
                        return 0
                m = (rise*inv_mod(run,self.p))%self.p
                b = (P[1] - P[0]*m)%self.p
                x = (m*m - P[0]-Q[0])%self.p
                y = (-(m*x+b))%self.p
                return (x,y)
        def inv(self,P):
                if P == 0: return 0
                return (P[0],(-P[1])%self.p)
        def mult(self,P,n):
                if n<0:
                        P = self.inv(P)
                        n = -n
                res = 0
                while n > 0:
                        if n%2 == 1:
                                res = self.add(res,P)
                        n /= 2
                        P = self.add(P,P)
                return res
        # Check if the curve contains a given point
        def contains(self,P):
                if P==0: return True
                lhs = P[1]**2
                rhs = P[0]**3 + self.A * P[0] + self.B
                return (lhs-rhs)%self.p == 0


# Class for a point on an elliptic curve.
class ec_pt:
        def __init__(self,C,coords):
                assert C.contains(coords)
                self.C = C
                self.coords = coords
        def __add__(self,Q):
                assert self.C.contains(Q.coords)
                if Q==0: return self
                return ec_pt(self.C, self.C.add(self.coords,Q.coords))
        def __radd__(self,Q):
                assert self.C.contains(Q.coords)
```

```
                if Q==0: return self
                else: return self+Q
        def __sub__(self,Q):
                assert self.C.contains(Q.coords)
                return self + (-Q)
        def __rmul__(self,n):
                return ec_pt(self.C, self.C.mult(self.coords,n))
        def __neg__(self):
                return ec_pt(self.C, self.C.inv(self.coords))
        def __str__(self):
                return str(self.coords)

# Inverses modulo m.
def inv_mod(a,m):
        pre = (a,1)
        cur = (m,0)
        while cur[0] != 0:
                k = pre[0] / cur[0]
                nex = (pre[0]-k*cur[0],pre[1]-k*cur[1])
                pre = cur
                cur = nex
        return pre[1]%m
```

The main advantage of defining these classes is that I've overloaded the operators `+`,`-` and `*`. This makes it possible to do elliptic curve computations more succinctly and intuitively. For example:

```
>>> from ec_utils import *
>>> C = curve(171,853,2671)
>>> P = ec_pt(C,(1980,431))
>>> print P+P
(1950, 1697)
>>> print P-P
0
>>> print 123456*P
(2148, 1151)
>>> print -P
(1980, 2240)
>>>
```

Of course, this isn't necessary to solve the problems. In the solutions below, you can just replace all the uses of `+`,`-`,`*` with calls to the appropriate elliptic curve arithmetic functions, supplying the coefficients A, B as arguments.

9. Write a program that determines whether a given ECDSA signature (s_1, s_2) for a document d is valid or not, given paramters and an ECDSA public key (notation as on page 322 of the

textbook). The signatures will all use curve P-384 from the document above.

Solution.

We do the computation as outlined in the textbook's table. Here is one implementation (using the `curve` and `ec_pt` classes written above).

Note that I have chosen to enter the P-384 parameters by pasting the hexadecimal representation into the code, and using some Python operations to remove the spaces and convert to an integer.

```
### Omitted: code of curve, ec_pt, and inv_mod (see above).

# ECDSA verification
# Arguments:
#        G is the point from the public parameters (an ec_pt)
#        p is the modulus
#        q is the order of G
#        V is the verification key (an ec_pt on the same curve as G)
#        (s1,s2) is the purported signature
#        d is the document
def valid(C,p,q,G,V,(s1,s2),d):
        s2i = inv_mod(s2,q)
        v1 = s2i*d % q
        v2 = s2i*s1 % q
        return (v1*G + v2*V).coords[0] % q  == s1

# Set up the curve and point for P-384; also gives p and q as return values
def p384():
        p = 39402006196394479212279040100143613805079739270465446666794829
340424572177149687032904726608825893800186160697311231 9
        q = 39402006196394479212279040100143613805079739270465446666794690
527962765939911326356939895630815229491355443365394264 3
        Bs = 'b3312fa7 e23ee7e4 988e056b e3f82d19 181d9c6e fe814112 0314088f
5013875a c656398d 8a2ed19d 2a85c8ed d3ec2aef'.replace(' ','')
        b = int(Bs,16)
        Gxs = 'aa87ca22 be8b0537 8eb1c71e f320ad74 6e1d3b62 8ba79b98
59f741e0 82542a38 5502f25d bf55296c 3a545e38 72760ab7'.replace(' ','')
        Gx = int(Gxs,16)
        Gys = '3617de4a 96262c6f 5d9e98bf 9292dc29 f8f41dbd 289a147c
e9da3113 b5f0b8c0 0a60b1ce 1d7e819d 7a431d7c 90ea0e5f'.replace(' ','')
        Gy = int(Gys,16)
        C = curve(-3,b,p)
        G = ec_pt(C,(Gx,Gy))
        return C,p,q,G

# I/O
```

```
xv,yv = map(int,raw_input().split())
d,s1,s2 = map(int,raw_input().split())
C,p,q,G = p384()
V = ec_pt(C,(xv,yv))
if valid(C,p,q,G,V,(s1,s2),d): print 'valid'
else: print 'invalid'
```

10. Suppose that Alice and Bob have performed elliptic curve Diffie-Hellman key exchange (as described as on page 317 of the textbook), but they have chosen the prime p to be only 24 bits in length. Write a program that is given the public parameters and the exchanged points Q_A and Q_B, and determines the shared secret.

    Revision: I have dropped the length of the prime from 32 bits to 24 bits, to allow a broader range of algorithms to work.

    Solution.

    We an implement the babystep-giantstep algorithm in almost example the same way as we did for the classical discrete logarithm problem. This can be used to extract either Alice's or Bob's secret exponent from their transmitted value, from which the shared secret can be computed.

    The only issue is that we no longer have access to the order of the group directly. However, we know for certain (by Hasse's theorem) that this order is at most $p + 1 + 2\sqrt{p}$. So we can use this as our N, and take the giantstep size to be $\sqrt{N}$, rounded up.

```
### Omitted: code for curve, ec_pt, and inv_mod (see above).

import math
def ec_bsgs(P,Q):
        p = P.C.p # Get the prime modulus from P's curve
        # Maximum possible order of the curve, by Hasse's theorem:
        N = p+int(2*math.sqrt(p))+1
        B = int(math.sqrt(N))+1 # Size of giant-step
        dpn = dict()
        R = 0*P
        for i in range(B):
                dpn[R.coords] = i
                R += P
        S = Q
        PP = B*P # Giant stepsize
        for j in range(B):
                if S.coords in dpn: return dpn[S.coords]+B*j
                S -= PP

def dha(P,QA,QB):
        na = ec_bsgs(P,QA)
        return na*QB
```

```
# I/O
A,B,p = map(int,raw_input().split())
px,py = map(int,raw_input().split())
qax,qay = map(int,raw_input().split())
qbx,qby = map(int,raw_input().split())

C = curve(A,B,p)
P = ec_pt(C,(px,py))
QA = ec_pt(C,(qax,qay))
QB = ec_pt(C,(qbx,qby))
S = dha(P,QA,QB)
print S.coords[0],S.coords[1]
```

11. Write a program to decipher messages enciphered with the Menezes-Vanstone cryptosystem described in problems 6 and 7 (textbook exercises 6.17 and 6.18), given a private key and a ciphertext. The public parameters will be those of curve P-192 from the above document.

Solution.

The following code essentially follows the notation and procedure form the book's table.

```
### Omitted: code for curve, ec_pt, and inv_mod (see above)

# Returns C,p,q,G for the curve P-192.
def p192():
        p = 6277101735386680763835789423207666416083908700390324961279
        q = 6277101735386680763835789423176059013767194773182842284081
        bs = '64210519 e59c80e7 0fa7e9ab 72243049 feb8deec
c146b9b1'.replace(' ','')
        Gxs = '188da80e b03090f6 7cbf20eb 43a18800 f4ff0afd
82ff1012'.replace(' ','')
        Gys = '07192b95 ffc8da78 631011ed 6b24cdd5 73f977a1
1e794811'.replace(' ','')

        b = int(bs,16)
        Gx = int(Gxs,16)
        Gy = int(Gys,16)

        C = curve(-3,b,p)
        G = ec_pt(C,(Gx,Gy))
        return C,p,q,G

# Notation as in the textbook's table
def mv_decipher(p,P,nA,R,c1,c2):
        T = nA * R
```

```
        xt,yt = T.coords
        return (inv_mod(xt,p)*c1 % p, inv_mod(yt,p)*c2 % p)

# I/O
na = int(raw_input())
xr,yr = map(int,raw_input().split())
c1,c2 = map(int,raw_input().split())

C,p,q,P = p192()
R = ec_pt(C,(xr,yr))
m1,m2 = mv_decipher(p,P,na,R,c1,c2)
print m1,m2
```

12. (*Extra credit*) Write a program to factor RSA moduli (i.e. products of two prime numbers) at least 63 bits in length. The test cases for this problem range in length from 63 bits to 128 bits, and your score is based on how many your program is able to solve, as usual.

    Solving half the test cases will be worth as many points as one ordinary programming problem, and should be possible using Lenstra's algorithm (from section 6.6). The remaining cases leave you plenty of room to try to improve the algorithm's performance, or to try any other ideas you may come up with.

    Solution.

    There are many factoring algorithms, and many ways to try to optimize each one. See the submitted solutions online for some examples (I've made them invisible for now, since you are still allowed to continue to try this challenge until the end of the week).