All exercise numbers from the textbook refer to the second edition. Problems 8, 9, and 12 refer to material we will not discuss until Monday (§2.4 of the text).

**Written problems**

1. Textbook exercise 1.35.

   **Remark.** A prime $p$ such that $q = \frac{1}{2}(p - 1)$ is also prime is called a safe prime, because the discrete logarithm problem modulo $p$ is particularly difficult, for reasons we'll see soon. The corresponding prime $q$ is called a Sophie Germain prime. Germaine originally studied these primes in the 19th century in connection with work on Fermat’s last theorem.

   **Solution.**

   Since $g \not\equiv 0 \pmod{p}$, it is a unit; it is only necessary to prove that its order is equal to $p - 1$. By Proposition 1.29, the order of $g$ must be a factor of $p - 1$. By assumption, $p - 1 = 2q$, where $q$ is prime, so the only factors of $p - 1$ are $1, 2, q$, and $p - 1$. It suffices to show that $a^d \not\equiv 1 \pmod{p}$ for $d \in \{1, 2, q\}$.

   We are given that $g^1 \not\equiv 1 \pmod{p}$ and $g^q \not\equiv 1 \pmod{p}$, so it remains to verify that $g^2 \not\equiv 1 \pmod{p}$. In other words, we must show that $g^2 - 1 = (g + 1)(g - 1) \not\equiv 0 \pmod{p}$. This follows since $p$ divides neither $g + 1$ nor $g - 1$ (since $g \not\equiv \pm 1 \pmod{p}$), so it cannot divide their product.

   Therefore the only possible order of $g$ is $p - 1$ itself, i.e. $g$ is a primitive root.

2. Textbook exercise 2.3.

   **Solution.**

   (a) We know that $g^a \equiv g^b \pmod{p}$. Without loss of generality, $a > b$. Multiplying by the inverse of $g$ times on both sides, it follows that $g^{a-b} \equiv 1 \pmod{p}$. By Proposition 1.29, the order of $g$ must divide $a - b$. Since $g$ is a primitive root, the order is $p - 1$, so $(p - 1) | (a - b)$, i.e. $a \equiv b \pmod{p - 1}$.

   For the function on page 65 to be well-defined, any solution $a$ to $g^a \equiv h \pmod{p}$ must give rise to the same equivalence class $[a]_p \in \mathbb{Z}/(p - 1)$. But this amounts to saying that any two such solutions lie in the same equivalence class modulo $p - 1$, which is exactly what we just proved.

   (b) Let $[a_1]_{p-1} = \log_g(h_1)$ and $[a_2]_{p-1} = \log_g(h_2)$. Then $g^{a_1} \cdot g^{a_2} = g^{a_1 + a_2} \equiv h_1 h_2 \pmod{p}$, so $[a_1 + a_2]_{p-1} = \log_g(h_1 h_2)$. In other words, $\log_g(h_1) + \log_g(h_2) = \log_g(h_1 h_2)$.

   (c) Let $a = \log_g(h)$. Then $g^a \equiv (g^n)^a \equiv h^n \pmod{p}$, so $an = \log_g(h^n)$, i.e. $n \log_g(h) = \log_g(h^n)$. Note that I am using implicitly that $n$th powers are well-defined for $n$ negative (in which case it denotes the $(-n)$th power of the inverse modulo $p$), and that the rules $(g^a)^{-n} \equiv g^{-an}$ remains valid regardless of the sign of $n$.

3. Textbook exercise 2.11.

   **Solution.**

   (a) $\tau \sigma^2 = (\tau \sigma) \sigma = (\sigma^2 \tau) \sigma = \sigma^2 (\tau \sigma) = \sigma^2 (\sigma^2 \tau) = (\sigma^3) \sigma \tau = \sigma \tau$.

   (b) $\tau (\sigma \tau) = (\tau \sigma) \tau = (\sigma^2 \tau) \tau = \sigma^2 (\tau^2) = \sigma^2$.

   (c) $(\sigma \tau)(\sigma \tau) = \sigma (\tau \sigma) \tau = \sigma (\sigma^2 \tau) \tau = \sigma^3 \tau^2 = e$. 

Due the night of Thursday 9/29 (hard deadline 4am on 9/30).
(d) \((\sigma \tau)(\sigma^2 \tau) = \sigma(\tau \sigma)(\sigma \tau) = (\sigma \sigma^2)(\tau \sigma) = e(\sigma^2 \tau)\tau = \sigma^2 \tau^2 = \sigma^2.\)

\(S_3\) is not a commutative group. Observe for example that \(\tau \sigma = \sigma^2 \tau\), which is not the same element as \(\sigma \tau\).


**Solution.** Below, I will write \(g^d\) rather than \(g^*d\) to denote the operation \(*\) applied to \(d\) copies of \(g\) in a row (as is done in the problem statement in the textbook).

(a) If \(g \in G[d]\), then \(g^d = e\) (by definition). Therefore, multiplying by \(g^{-d}\), \(e = g^{-d}\), i.e. \(e = (g^{-1})^d\). So by definition, \(g^{-1} \in G[d]\) as well. (Note that I did not need commutativity in this argument).

(b) If \(G\) is commutative and \(g_1, g_2\) are both in \(G[d]\), then \((g_1 \ast g_2)^d = g_1^d \ast g_2^d\) (by rearranging the \(2d\) terms in the product to put all of the \(g_1s\) first), which in turn is \(e \ast e = e\). So \(g_1 \ast g_2 \in G[d]\).

(c) If \(G\) is commutative, then by part (b), the operation \(*\) (restricted to \(G[d] \times G[d]\)) gives a function to \(G[d] \times G[d] \rightarrow G[d]\). This operation of \(G[d]\) is associative automatically, since it is already associative as an operation on \(G\). The identity of \(G\) lies in \(G[d]\) and is also an identity there. Finally, by (a) the inverse of every element in \(G[d]\) is also in \(G[d]\), and since the product \(g \ast g^{-1}\) is the identity \(e\) in \(G\), which is the same as the identity in \(G[d]\), the element \(g^{-1}\) also works as an inverse in \(G[d]\).

(d) Let \(G = S_3\), and consider the subset \(G[2]\). Then \(\tau\) and \(\sigma \tau\) both lie in this set, since \(\tau^2 = e\) (given) and \((\sigma \tau)^2 = e\) by part (c) of the previous problem. However, the product \((\sigma \tau)\tau = \sigma\) does not lie in \(G[2]\), since \(\sigma^2 \neq e\). So \(G[2]\) is not closed under the operation \(*\), so it is not a group with the operation \(*\).

5. Consider the set \(\mathbb{N}\) of positive integers, equipped with the following operation.

\[
x \ast y = \max(x, y)
\]

Show that \((\mathbb{N}, \ast)\) satisfies all of the conditions in the definition of a group (as on page 74 of the textbook) except one. Which condition does not hold?

**Solution.** Note that if \(x, y \in \mathbb{N}\), then \(\max(x, y)\) is again a positive integer, so \(*\) does induce a well-defined operation \(\mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}\).

The identity law holds. To see this, observe that for any \(x \in \mathbb{N}\), \(x \ast 1 = 1 \ast x = \max(x, 1) = x\), since all natural numbers are greater than or equal to 1 by definition. So 1 is the identity element.

The associative law holds, because both of the expressions \(a \ast (b \ast c)\) and \((a \ast b) \ast c\) amount to the same thing: the maximum of the set \(\{a, b, c\}\).

However, the inverse law does not hold. For a specific counterexample, \(2 \ast n \geq 2\) for all \(n \in \mathbb{N}\) (since the maximum of 2 and \(n\) is certainly at least as large as 2). In particular, \(2 \ast n\) is never equal to 1. Since 1 is the identity element, this shows that 2 has no inverse. In fact, 1 itself is the only number that has a \(*\)-inverse.

6. (a) Consider the set \(M\) consisting of all \(2 \times 2\) matrices with integer entries, with the operation \(\cdot\) being ordinary matrix multiplication. Show that \((M, \cdot)\) is not a group.
(b) Let $G$ denote the subset of $M$ consisting of those matrices with determinant $\pm 1$. Show that $(G, \cdot)$ is a group. (This group is usually denoted $\text{GL}_2(\mathbb{Z})$ and is called the general linear group of degree 2 over $\mathbb{Z}$).

(c) Show that $(G, \cdot)$ is not a commutative group.

Solution.

(a) The identity and associativity laws hold; the identity is the identity matrix $\left( \begin{array}{cc} 1 & 0 \\ 0 & 1 \end{array} \right)$. However, the inverse law fails. For example, the zero matrix $\left( \begin{array}{cc} 0 & 0 \\ 0 & 0 \end{array} \right)$ has no inverse.

(b) The operation of matrix multiplication gives a well-defined operation on $G$ due to the fact that $\det(AB) = \det A \cdot \det B$; this means that the product of two elements in $S$ again has determinant $\pm 1$ (and integer entries), hence it again lies in $G$. So $\cdot$ is a well-defined operation on $G$.

The identity law holds with identity element $I = \left( \begin{array}{cc} 1 & 0 \\ 0 & 1 \end{array} \right)$, as is verified by checking that $\left( \begin{array}{cc} a & b \\ c & d \end{array} \right) I = I \left( \begin{array}{cc} a & b \\ c & d \end{array} \right) = \left( \begin{array}{cc} a & b \\ c & d \end{array} \right)$. The associative law holds since matrix multiplication is associative (which amounts to the fact that matrix multiplication is defined to be the composition of the two linear transformations described by the matrices). It remains to verify that inverses exist.

The following formula can be used to compute the inverse of a $2 \times 2$ matrix. There are numerous ways to obtain this formula, such as row-reduction or an easy application of Cramer’s rule.

\[
(a \ b) \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 & -b \\ d & -c \end{pmatrix}
\]

If $M = \left( \begin{array}{cc} a & b \\ c & d \end{array} \right) \in G$, then by definition $ad - bc = \pm 1$, so $\frac{1}{ad - bc} \in \mathbb{Z}$, and the inverse matrix again has integer entries. A straightforward calculation shows that the determinant of this inverse is $\frac{1}{ad - bc}$, which is $\pm 1$. So the inverse matrix lies in $G$ as well.

Thus $G$, with matrix multiplication, is closed under an operation that is associative, has an identity, and has inverses of all elements. Therefore it is a group.

(c) It suffices to find a single example of two matrices that do not commute. There are many options; here is one.

\[
A := \left( \begin{array}{cc} 1 & 1 \\ 0 & 1 \end{array} \right) \\
B := \left( \begin{array}{cc} 1 & 0 \\ 1 & 1 \end{array} \right)
\]

Notice that both these matrices lie in $S$, since they have determinant 1 and have integer entries. Now observe what happens if we multiple them the two possible ways.

\[
A \cdot B = \left( \begin{array}{cc} 2 & 1 \\ 1 & 1 \end{array} \right) \\
B \cdot A = \left( \begin{array}{cc} 1 & 1 \\ 1 & 1 \end{array} \right)
\]

Since $AB \neq BA$, it follows that the commutative law does not hold in $S$. 
7. Let \((G, \star)\) be a group, and assume that \(|G|\) is a prime number. For \(e \geq 0\), denote by \(g^e\) the element \(g \star g \star \cdots \star g\), where \(g\) appears \(e\) times. Prove that if \(e\) is a positive integer not divisible by \(p\), then the function \(f : G \to G\) given by \(f(g) = g^e\) is a one-to-one function.

**Solution.** We shall prove that if \(g^e = h^e\), then \(g = h\). This will show that no two distinct elements are sent to the element by this function.

By Lagrange’s theorem (Proposition 2.13), for all \(g \in G\), \(g^p\) is the identity (since \(p\) is the order of the group). From this it follows that if \(i, j\) are any two positive integers that are congruent modulo \(p\), then \(g^i = g^j\): indeed, we can write \(i = j + kp\) for some integer \(k\), and the \(g^i = g^j \cdot (g^p)^k = g^j\).

Let \(u\) be an inverse modulo \(p\) of \(e\). This inverse must exist since \(\gcd(e, p) = 1\). Then \(ue \equiv 1 \pmod{p}\). Therefore using the observation in the previous paragraph:

\[
\begin{align*}
g^e = h^e & \Rightarrow (g^e)^u = (h^e)^u \\
& \Rightarrow g^e u = h^e u \\
& \Rightarrow g = h,
\end{align*}
\]

where in the last line we use that \(g^e u = g^1 = g\), and similarly with \(h\). So no two group elements have the same \(e\)th power, and the function is one-to-one, as desired.

8. Textbook exercise 2.8.

**Solution.** All bracketed numbers refer to congruence classes modulo \(p = 1373\).

(a) \([A] = [2]^{947} = [177]\) (computing using Python’s built-in modular power function).

(b) \([B] = [2]^{716} = [469]\). For the message \([m] = [583]\) and ephemeral key \(k = 877\), Alice computes

\[
\begin{align*}
[c_2] &= [469]^{877} \cdot [583] = [644] \cdot [583] = [623]
\end{align*}
\]

(c) Alice decipher the original message as \([m] = [c_1]^{-a} \cdot [c_2] = [661]^{-299} \cdot [1325]\). The inverse of \([661]\) modulo \(p\) is \([673]\) (this can be found, for example, with the extended Euclidean algorithm, or as \([661]^{p-2}\) using Fermat’s little theorem). So the message is \([673]^{-299} \cdot [1325]\). Using a fast-powering algorithm, this evaluates to \([322]\).

(d) Here is some Python code to steal Bob’s private key. The prime \(p\) is small enough that this can be done by a rather naive trial-and-error.

```python
>>> p = 1373
>>> g = 2
>>> B = 893
>>> for b in xrange(p-1):
...     if pow(g,b,p) == B:
...         print b
...     ...
>>> 219
```

So \(b = 219\). Therefore the plaintext is \([m] = [c_1]^{-b}[c_2] = [693]^{-219}[793]\). We can compute that \([693]^{-1} = [317]\), so the plaintext is \([m] = [317]^{219} \cdot [793] = [532] \cdot [793] = [365]\). So the plaintext send to Bob was 365.

Solution.

(a) The key fact is the following, which we have discussed in class but bears emphasis. We give a proof here for completeness, but you may use this fact without proof in this and all future homework and exams.

Lemma. If \( a \not\equiv 0 \pmod{p} \), and \( i, j \) are two integers such that \( i \equiv j \pmod{p-1} \), then \( a^i \equiv a^j \pmod{p} \).

Proof. Without loss of generality \( i > j \), so write \( i = j + (p-1)k \) for some nonnegative integer \( k \). Then \( a^i = a^j \cdot (a^{p-1})^k \equiv a^j \cdot 1^k \equiv a^j \pmod{p} \), by Fermat’s little theorem.

We now use this fact to analyze this cryptosystem.

Alice’s number 3589 and 15619 are related by the fact that they are inverses modulo \( p-1 = 32610 \): \( 3589 \cdot 15619 \equiv 1 \pmod{p-1} \). Similarly, Bob’s two numbers are inverses modulo \( p-1 \) as well: \( 4037 \cdot 31883 \equiv 1 \pmod{p-1} \).

Bob’s final number can be re-expressed modulo \( p \) as \( w^{31883} \equiv v^{15619} \cdot 31883 \equiv u^{4037} \cdot 15619 \cdot 31883 \equiv m^{3589} \cdot 15619 \cdot 31883 \pmod{p} \). In turn, this is \( (m^{3589} \cdot 15619)^{4037} \cdot 31883 \pmod{p} \) (note that I have groups Alice’s two number together and Bob’s two numbers together).

Since \( 3589 \cdot 15619 \equiv 1 \pmod{p-1} \), it follow that \( m^{3589} \cdot 15619 \equiv m^1 \pmod{p} \). In turn, \( (m^{3589} \cdot 15619)^{4037} \cdot 31883 \equiv m^{4037} \cdot 31883 \equiv m^1 \pmod{p} \), since 4037 \cdot 31883 \equiv 1 \pmod{p-1} . Therefore Bob has indeed computed Alice’s original plaintext.

(b) In general, Alice chooses two numbers \( a, c \) that are inverses modulo \( p-1 \), and Bob chooses two numbers \( b, d \) that are inverses modulo \( p-1 \). Alice chooses her plaintext \( m \), then she sends \( u \equiv m^a \pmod{p} \) to Bob. Bob computes \( v \equiv u^b \pmod{p} \) and sends it back to Alice. Alice computes \( w \equiv v^c \pmod{p} \) and sends it back to Bob. Finally, Bob computes \( w^d \pmod{p} \). This will be the original plaintext, because it is congruent to \( m^{abcd} \equiv (m^ac)^bd \pmod{p} \). The fact that \( ac \equiv 1 \pmod{p-1} \) and \( bd \equiv 1 \pmod{p-1} \) guarantees that this is congruent to \( m^{bd} \pmod{p} \), and in turn to \( m \pmod{p} \), the original plaintext.

(c) This encryption system requires three transmissions of data: two from Alice to Bob and one from Bob to Alice. In contrast, Elgamal requires only two (if Alice publishing her key is counted as a transmission), or just one transmission if Alice’s public key is distributed beforehand. It is worth observing, however, that the transmission in Elgamal is twice the size of the transmissions taking place in this problem, since it consists of two elements of \( \mathbb{Z}/p^* \) rather than one.

(d) Eve will know the three numbers \( u, v, w \) that have passed over the channel. If Eve can solve the discrete logarithm problem, then she can determine the number \( b \) from the first two transmissions, as \( \log_u(v) \). She can then invert \( b \pmod{p-1} \) to determine \( d \), and then compute \( w^d \), which is the plaintext when reduced modulo \( p \).

In fact, Eve can break this encryption as long as she can break Diffie-Hellman key exchange (whether this involves taking a discrete logarithm or not). Observe that the three numbers Eve knows satisfy these congruences:

\[
\begin{align*}
  u &\equiv m^a \pmod{p} \\
  v &\equiv m^{ab} \pmod{p} \\
  w &\equiv m^{abc} \pmod{p}
\end{align*}
\]
The number Eve wishes to know is \( m \), which is the congruent to \( m^{abcd} \pmod{p} \). Notice also that, since \( x^{bd} \equiv x \pmod{p} \) for all units \( x \), we can also write \( u \equiv m^{abcd} \pmod{p} \). Therefore, letting \( g \equiv m^{ob} \pmod{p} \) (which Eve knows; it is \( v \)), Eve knows \( g^c \pmod{p} \) (it is \( w \)) and \( g^d \pmod{p} \) (it is \( u \)), and the plaintext that she wishes to extract is \( g^{cd} \pmod{p} \). This is precisely the Diffie-Hellman problem: she knows a base and two powers of it, and she wants to be able to “multiply the powers” together.

**Programming problems**

Full formulation and submission: [https://www.hackerrank.com/m158-2016-pset-3](https://www.hackerrank.com/m158-2016-pset-3)

10. Write a program that breaks Diffie-Hellman for 16-bit primes. That is, you will receive as input the parameters \( g, p \) (where \( p < 2^{16} \)) and elements \( A, B \), and you must determine the shared secret \( S \). (16-bit primes are small enough that a fairly naive approach will work).

**Solution.**

Eve can break Diffie-Hellman by extracting Bob’s secret number \( b \) by taking a discrete logarithm. With 16-bit primes this is tractable to do by trial and error. Here is an implementation.

```python
p, g, A, B = map(int, raw_input().split())

# Discrete log base g of [x]_p
# Naive trial-and-error
def dlog(g, x, p):
    e = 0
    y = 1 # Will be g^e % p throughout
    while y != x:
        y = y * g % p
        e += 1
    if (e >= p-1): return None # Would have found it by now
    return e

def crackdh(p, g, A, B):
    b = dlog(g, B, p)
    return pow(A, b, p)

print crackdh(p, g, A, B)
```

11. Given an integer \( m \) and the four entries of a \( 2 \times 2 \) matrix \( A \), and a positive integer \( e \) (up to 1024 bits long), determine the result of reducing all entries of the matrix \( A^e \) modulo \( m \). (You will want to reduce the entries of matrices “along the way;” it will not be feasible to compute the entries of the matrix \( A^e \) in full).

**Solution.**

The key observation is the the same fast-powering algorithm that we discussed for powers in modular arithmetic works in any group whatsoever (and in fact for any associative operation, whether invertible or not). So we simply implement this algorithm, replacing multiplication with matrix multiplication. In order to keep the numbers involved reasonably sized, we should
reduce all matrix entries modulo $m$ at each step, rather than waiting until the end. Here is an implementation.

Note that I’ve written an ad hoc matrix multiplication function. In real applications, you would presumably make up of a linear algebra library to do this sort of thing, but for $2 \times 2$ matrices in makes very little difference.

```python
# Multiply two 2 x 2 matrices and reduce modulo m
# Assumes without checking that A,B represent matrices of matching dimensions
def matmult(A,B,m):
    arows,brows,bcols = len(A), len(B), len(B[0])
    return [[sum([A[i][k] * B[k][j] for k in xrange(bcols)]) % m
              for j in xrange(bcols)] for i in xrange(arows)]

# Computes a non-negative power of a matrix, modulo m
# Uses the square-and-multiply (fast-powering) algorithm
def matpow(A,e,m):
    assert(e >= 0)
    res = [[1,0],[0,1]]  # Init. to identity
    while e > 0:
        if e%2 == 1:
            res = matmult(res,A,m)
        e /= 2
        A = matmult(A,A,m)
    return res

# Read a matrix of the given number of rows from stdin
# Returns result as a list of lists
def readMatrix(rows):
    return [map(int,raw_input().split()) for i in xrange(rows)]

# I/O
m = int(raw_input())
A = readMatrix(2)
e = int(raw_input())
B = matpow(A,e,m)
print B[0][0], B[0][1]
print B[1][0], B[1][1]
```

12. Write a program which deciphers a message enciphered to you with Elgamal encryption, given the parameters $p,g$ and your private key. The length of the prime $p$ will be up to 1024 bits.

**Solution**

We essentially need only follow the steps shown in table 2.3. The necessary ingredients are fast exponentiation (which is built into Python as `pow`, but which we could have implemented...
easily enough with the fast-powering algorithm), and a function to compute inverses modulo $p$ (below, I use the extended Euclidean algorithm). Here is an implementation.

```python
p, g, a = map(int, raw_input().split())
c1, c2 = map(int, raw_input().split())

# A version of the extended Euclidean Algorithm.
# Returns [u,v,g], where g = gcd(a,m) and au + bv = g.
def ext_euclid(a, b):
    u0, v0, r0 = 1, 0, a
    u1, v1, r1 = 0, 1, b
    while r1 != 0:
        k = r0 // r1
        u2, v2, r2 = u0 - k * u1, v0 - k * v1, r0 - k * r1
        u0, v0, r0 = u1, v1, r1
        u1, v1, r1 = u2, v2, r2
    return [u0, v0, r0]

def decipher(c1, c2, p, g, a):
    A = pow(g, a, p)
    S = pow(c1, a, p)  # Analog of the DH shared secret
    Sinv = ext_euclid(S, p)[0] % p
    return c2 * Sinv % p

print decipher(c1, c2, p, g, a)
```

Due the night of Thursday 9/29 (hard deadline 4am on 9/30).