Written problems

1. Solve each system of congruences. Your answer should take the form of a single congruence of the form $x \equiv c \pmod{m}$ describing all solutions to the system.

   (a) $x \equiv 1 \pmod{3}$
   $x \equiv 2 \pmod{5}$

   (b) $x \equiv 6 \pmod{11}$
   $x \equiv 2 \pmod{10}$

   (c) $x \equiv 2 \pmod{3}$
   $x \equiv 1 \pmod{10}$
   $x \equiv 3 \pmod{7}$

   (d) $x \equiv 6 \pmod{8}$
   $x \equiv 3 \pmod{9}$
   $x \equiv 17 \pmod{17}$

Solution.

(a) Since $2 \cdot 3 - 5 = 1$, the inverse of 3 modulo 5 is 2. We use this below.

$$\begin{aligned} x &= 1 + 3k \text{ for some integer } k \\ \Rightarrow 1 + 3k &\equiv 2 \pmod{5} \\ 3k &\equiv 1 \pmod{5} \\ 3^{-1} \cdot 3k &\equiv 3^{-1} \pmod{5} \\ k &\equiv 2 \pmod{5} \\ \Rightarrow k &= 2 + 5h \text{ for some integer } h \\ x &= 1 + 3(2 + 5h) \\ x &= 7 + 15h \\ \Rightarrow x &\equiv 7 \pmod{15}. \end{aligned}$$

(b) The inverse of 11 modulo 10 is just 1 (since $11 \equiv 1$ itself), which makes the computation a bit simpler.

$$\begin{aligned} x &\equiv 6 \pmod{11} \\ \Rightarrow x &= 6 + 11k \text{ for some integer } k \\ 6 + 11k &\equiv 2 \pmod{10} \\ 11k &\equiv -4 \pmod{10} \\ k &\equiv 6 \pmod{10} \\ \Rightarrow k &= 6 + 10h \text{ for some integer } h \\ x &= 6 + 11(6 + 10h) \\ x &= 72 + 110h \\ \Rightarrow x &\equiv 72 \pmod{110} \end{aligned}$$

(c) We can proceed in two steps, first merging the first two congruences, then merging the result with the third. To be more succinct, I will skip some of the more routine steps shown in the first two parts.

$$\begin{aligned} x &= 2+3k \\ 3k &\equiv 1-2 \equiv 9 \pmod{10} \\ k &\equiv 3 \pmod{10} \\ x &= 2+3(3+10h) = 11+30h \\ 30h &\equiv 3-11 \equiv 6 \pmod{7} \\ 2h &\equiv 6 \pmod{7} \\ h &\equiv 4\cdot 6 \equiv 3 \pmod{7} \\ x &= 11+30(3+7j) = 101+210j \\ x &\equiv 101 \pmod{210} \end{aligned}$$

(d) We proceed similarly to the previous part.

$$\begin{aligned} x &= 6+8k \\ 8k &\equiv 3-6 \equiv 6 \pmod{9} \\ k &\equiv -6 \equiv 3 \pmod{9} \\ x &= 6+8(3+9h) = 30+72h \\ 72h &\equiv 17-30 \equiv 4 \pmod{17} \\ 4h &\equiv 4 \pmod{17} \\ h &\equiv 1 \pmod{17} \\ x &= 30+72(1+17j) = 102+1224j \\ x &\equiv 102 \pmod{1224} \end{aligned}$$

2. The element $288 \in (\mathbf{Z}/919)^\times$ has order 17. Use the babystep-giantstep algorithm, making using of the fact that the order of 288 is known to be 17, to evaluate the discrete logarithm $\log_{288} 162$ for the prime $p = 919$. You may use a computer to do the arithmetic, but show explicitly the two lists from which you find the collision.

Solution.

Since $17 < 5^2$, we can choose $B = 5$. Then it suffices to compute two lists, containing $288^e \pmod{919}$ and $162(288^{-5})^e \pmod{919}$ for $e = 0, 1, 2, 3, 4$.

The first five powers of 288 modulo 919 are: $288, 234, 305, 535, 607$. The inverse of 607 (mod 919) is 162. Therefore the second row of the table will consist of $162 \cdot 162^e \% 919$. The resulting two lists are as follows.

e	0	1	2	3	4
$288^e \pmod{919}$	1	288	234	305	535
$162 \cdot 288^{-5e} \pmod{919}$	162	512	234	229	338

Sure enough there is a collision: $288^2 \equiv 234 \equiv 162 \cdot 288^{-5\cdot 2} \pmod{919}$, hence $288^{12} \equiv 162 \pmod{919}$. So the desired discrete logarithm is 12.

3. Evaluate the discrete logarithm $\log_{40} 33$ for the prime $p = 73$ using the Pohlig-Hellman algorithm, according to the following steps (see the statement of Theorem 2.31 in the textbook for details on the notation). You may use, without proof, the fact that 40 is a primitive root modulo 73.

   (a) Let $N = \mathrm{ord}_{73}(40)$. Factor N into prime powers as $N = q_1^{e_1} \cdots q_t^{e_t}$.

   (b) Determine the numbers g_i and h_i for each i from 1 to t inclusive. For each i, what is the order of g_i modulo 73?

   (c) For each i, evaluate the discrete logarithm $y_i = \log_{g_i} h_i$, using a method of your choice.

   (d) Solve the system of congruences $x \equiv y_i \pmod{q_i^{e_i}}$ to obtain the discrete logarithm $x = \log_{40} 33$.

Solution.

(a) Since 40 is a primitive root, its order modulo 73 is $N = 72$. Factoring into prime powers,

$$N = 2^3 3^2.$$

(b)

$$\begin{aligned} g_1 &\equiv 40^{N/8} \equiv 40^9 \pmod{73} \\ &\equiv 10 \pmod{73} \\ h_1 &\equiv 33^{N/8} \equiv 33^9 \pmod{73} \\ &\equiv 63 \end{aligned}$$

$$\begin{aligned} g_2 &\equiv 40^{N/9} \equiv 40^8 \pmod{73} \\ &\equiv 55 \pmod{73} \\ h_2 &\equiv 33^{N/9} \equiv 33^8 \pmod{73} \\ &\equiv 55 \pmod{73} \end{aligned}$$

The order of g_1 is 8 and the order of g_2 is 9, by construction.

(c) To evaluate $\log_{10}(63)$, we can use the fact that 10 has order 8 modulo 73. This is small enough that it's reasonable to just list the first seven powers of 10 to see which one is 63. These powers are $10, 27, 51, 72, 63, 46, 22$; so the desired logarithm is $y_1 = 5$.

The second logarithm, $\log_{55}(55)$, is immediate: it is simply $y_2 = 1$.

(d) We must solve the system

$$\begin{aligned} x &\equiv 5 \pmod{8} \\ x &\equiv 1 \pmod{9} \end{aligned}$$

which we can perform in the same manner as in the first problem.

$$\begin{aligned} x &= 5 + 8k \\ 8k &\equiv 1 - 5 \equiv 5 \pmod 9 \\ k &\equiv -5 \equiv 4 \pmod 9 \\ x &= 5 + 8(4 + 9h) = 37 + 72h \\ x &\equiv 37 \pmod{72} \end{aligned}$$

So the desired logarithm is $\log_{40} 33 = 37 \pmod{72}$.

4. Consider the set $\mathbf{N}$, equipped with the following operation.

$$x \star y = \max(x, y)$$

Show that $(\mathbf{N}, \star)$ 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 $x, y \in \mathbf{N}$, then $\max(x, y)$ is again a positive integer, so $\star$ does induce a well-defined operation $\mathbf{N} \times \mathbf{N} \to \mathbf{N}$.

The **identity law holds.** To see this, observe that for any $x \in \mathbf{N}$, $x \star 1 = 1 \star x = \max(x, 1) = x$, 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 $a \star (b \star c)$ and $(a \star b) \star c$ amount to the same thing: the maximum of the set $\{a, b, c\}$.

However, the **inverse law does not hold.** For a specific counterexample, $2 \star n \geq 2$ for all $n \in \mathbf{N}$ (since the maximum of 2 and n is certainly at least as large as 2). In particular, $2 \star n$ 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 $\star$-inverse.

5. (a) Consider the set M consisting of all 2×2 matrices with integer entries, with the operation $\cdot$ being ordinary matrix multiplication. Show that $(M, \cdot)$ is *not* a group.

Solution.

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

(b) Let S denote the subset of M consisting of those matrices with determinant equal to 1. Show that $(S, \cdot)$ *is* a group. (This group is usually denotes $\mathrm{SL}_2(\mathrm{Z})$ and is called the *special linear group of degree* 2 *over* $\boldsymbol{Z}$).

Solution.

The operation of matrix multiplication gives a well-defined operation on S 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 1 (and integer entries), hence it again lies in S. So $\cdot$ is a well-defined operation on S.

The identity law holds with identity element $I = \left(\begin{smallmatrix} 1 & 0 \\ 0 & 1 \end{smallmatrix}\right)$, as is verified by checking that $\left(\begin{smallmatrix} a & b \\ c & d \end{smallmatrix}\right) I = I \left(\begin{smallmatrix} a & b \\ c & d \end{smallmatrix}\right) = \left(\begin{smallmatrix} a & b \\ c & d \end{smallmatrix}\right)$. 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 2×2 matrix. There are numerous ways to obtain this formula, such as row-reduction or an easy application of Cramer's rule.

$$\left(\begin{smallmatrix} a & b \\ c & d \end{smallmatrix}\right)^{-1} = \frac{1}{ad - bc} \left(\begin{smallmatrix} d & -b \\ -c & a \end{smallmatrix}\right)$$

If $M = \left(\begin{smallmatrix} a & b \\ c & d \end{smallmatrix}\right) \in S$, then by definition $ad - bc = 1$. Therefore we have the following particularly simple formula.

$$\left(\begin{smallmatrix} a & b \\ c & d \end{smallmatrix}\right) \in S \Rightarrow \left(\begin{smallmatrix} a & b \\ c & d \end{smallmatrix}\right)^{-1} = \left(\begin{smallmatrix} d & -b \\ -c & a \end{smallmatrix}\right)$$

In particular, this inverse again has integer entries, and its determinant is again $ad - bc = 1$, so it too lies in S. So S indeed forms a group.

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

Solution.

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

$$\begin{aligned} A &:= \left(\begin{smallmatrix} 1 & 1 \\ 0 & 1 \end{smallmatrix}\right) \\ B &:= \left(\begin{smallmatrix} 1 & 0 \\ 1 & 1 \end{smallmatrix}\right) \end{aligned}$$

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.

$$\begin{aligned} A \cdot B &= \left(\begin{smallmatrix} 2 & 1 \\ 1 & 1 \end{smallmatrix}\right) \\ B \cdot A &= \left(\begin{smallmatrix} 1 & 1 \\ 1 & 2 \end{smallmatrix}\right) \end{aligned}$$

Since $AB \neq BA$, it follows that the commutative law does not hold in S.

Programming problems

6. Write a program which solves the a discrete logarithm problem, where the base of the exponentiation has a *known* order considerably smaller than the prime number p. Specifically, your program will read four integers p, g, a, N, where p is a 1024 bit prime, g, a are elements of $\mathbf{Z}/p$, and N is a 32-bit integer guaranteed to be equal to the order of g modulo p. It is further guaranteed that some power of g is congruent to $a \pmod{p}$. Your program should print an element e of $\mathbf{Z}/N$ such that $g^e \equiv a \pmod{p}$.

Solution.

This problem can be solved with the babystep-giantstep algorithm. The only modification needed from the 36-bit discrete logarithm problem on the previous problem set is that our discrete logarithm function must accept one additional argument (the order N of the element g), and it should use N is place of $p-1$. Here is an implementation, which differs only slightly from the solution to last week's problem.

```
import math

# Computes modular inverse, using the Euclidean algorithm
def inverse(a,p):
        pre = (p,0)
        cur = (a,1)
        while cur[0] > 0:
                k = pre[0]/cur[0]
                nex = (pre[0]-k*cur[0],pre[1]-k*cur[1])
                pre = cur
                cur = nex
        assert(pre[0] == 1) # Make sure the inverse was found
        return pre[1]%p

# Finds indices of a common element of two lists
def coll_ind(l1, l2):
        ind_of = dict()
        for i in xrange(len(l1)):
                ind_of[ l1[i] ] = i
        for j in xrange(len(l2)):
                if l2[j] in ind_of:
                        return (ind_of[l2[j]],j)
        # Return none if no collision found
        return None

# Babystep-Giantstep algorithm, where we assume order(g) <= N.
def dlp_bsgs(p,g,a,N):
        # Choose a "base"
        B = int(math.sqrt(N))+1

        # Make the list of babysteps (powers of g mod p)
        ge = 1 #Current power of g
        bs = []
        for e in xrange(B):
                bs += [ge]
                ge = ge*g % p

        # Compute g^(-B) modulo p.
```

```
        # Note that ge is currenlty set to g^B%p, so we can just invert it.
        h = inverse(ge,p)

        # Make the giantstep list
        nex = a
        gs = []
        for e in xrange(B):
                gs += [nex]
                nex = nex*h % p

        # Find the collision and compute the result
        (i,j) = coll_ind(bs,gs)
        return i + B*j

# I/O
p,g,a,N = map(int,raw_input().split())
print dlp_bsgs(p,g,a,N)
```

7. Write a program which takes as input an integer n and n pairs of integers y_i, m_i, and prints a pair of integers x, m, where $x \pmod m$ is the solution to the system of congruences $x \equiv y_i \pmod{m_i}$. The n integers m_i are guaranteed to be pairwise relatively prime.

   Solution.

   This problem can be solved by first writing a function `merge`, which converts two congruence conditions $(a_1, m_1), (a_2, m_2)$ into a single congruence condition (assuming that m_1 and m_2 are relatively prime), and then performing this function repeatedly to collapse the full list of congruences into one.

   There are various ways to implement the merging function. One option is to mirror the method used in problem 1. A slightly slicker way, well-suited to a computer program, is to observe that once we use the Euclidean algorithm to solve the equation

$$m_1 u + m_2 v = 1,$$

   (as we will need to do regardless, to find the inverse of one modulus with respect to the other), we can observe that

$$\begin{aligned} m_1 u &\equiv 0 \pmod u & \qquad m_2 v &\equiv 1 \pmod u \\ m_1 u &\equiv 1 \pmod v & \qquad m_2 v &\equiv 0 \pmod v \end{aligned}$$

   from which it follows that the number $a_2 m_1 u + a_1 m_2 v$ satisfies both congruences. This idea is implemented in the following code.

```
# Returns (g,u,v), where g = gcd(a,b) and g = a*u + b*v
def ext_euclid(a,b):
```

```
        pre = (a,1,0)
        cur = (b,0,1)
        while cur[0] != 0:
                k = pre[0] / cur[0]
                nex = (pre[0]-k*cur[0],pre[1]-k*cur[1],pre[2]-k*cur[2])
                pre = cur
                cur = nex
        return pre

# Merge two congruences
def merge_two(a1,m1,a2,m2):
        g,u,v = ext_euclid(m1,m2)
        assert(g == 1) # Doesn't currently handle the non-coprime case
        return ( (v*m2*a1 + u*m1*a2)%(m1*m2), m1*m2 )

# Takes lists a and m; merges the conditions a[i] mod m[i]
def merge_list(a,m):
        # Initialize result to the "trivial congruence" 0 mod 1
        res = (0,1)
        for i in range(len(a)):
                res = merge_two(res[0],res[1],a[i],m[i])
        return res

# I/O
n = int(raw_input())
y = []
m = []
for i in range(n):
        yy,mm = map(int,raw_input().split())
        y += [yy]
        m += [mm]
res = merge_list(y,m)
print res[0],res[1]
```

8. Let m be any positive integer, and let G denote the set of all 2×2 matrices A with entries chosen from $\mathbf{Z}/m$ such that the determinant of A (which you can regard as an element of $\mathbf{Z}/m$) is a unit modulo m. Then G, with usual matrix multiplication (where you should reduce each entry modulo m after computing it in the usual way) forms a group, usually denoted $\mathrm{GL}_2(\mathbf{Z}/m)$. Write a program which takes an integer m and the four entries of an element of G and prints the inverse element in G, represented as the four entries of the matrix, all reduced modulo m.

   Note. The group $G = \mathrm{GL}_2(\mathbf{Z}/m)$ is usually called the *general linear group of degree* 2 *over* $\boldsymbol{Z}/m$. In contrast to "special linear" groups, "general linear" groups are defined by the weaker condition that the determinant of the matrix is invertible, rather than the stronger condition

that this determinant is exactly equal to 1.

Solution.

We can use the same formula as usual for inversion of a 2×2 matrix:

$$\left(\begin{smallmatrix} a & b \\ c & d \end{smallmatrix}\right)^{-1} = (ad - bc)^{-1} \left(\begin{smallmatrix} d & -b \\ -c & a \end{smallmatrix}\right).$$

The only difference here, compared to the real-number case, is that we should interpret $(ad - bc)^{-1}$ as the inverse *modulo* m, rather than the ordinary reciprocal. As usual, we can compute this with the extended Euclidean algorithm. Here is an implementation.

```
# Returns the inverse of a modulo m, using the Euclidean algorithm
# Assumes that gcd(a,m) = 1.
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

# Inverts the matrix A = ((a,b),(c,d)) modulo m.
def inv_mat(A,m):
        d = A[0][0]*A[1][1] - A[0][1]*A[1][0]
        di = inv_mod(d,m)
        res =  [ [A[1][1]*di % m, -A[0][1]*di % m],
                 [-A[1][0]*di % m, A[0][0]*di % m] ]
        return res

# I/O
m = int(raw_input())
a,b = map(int,raw_input().split())
c,d = map(int,raw_input().split())
A = [[a,b],[c,d]]
B = inv_mat(A,m)
print B[0][0],B[0][1]
print B[1][0],B[1][1]
```

9. Let G be as in the previous problem. Write a function which takes as input the integer m, the four entries of an element A of G, and an integer n (which may be positive or negative) and returns the element A^n of G. Note that n may be quite large; you should use the fast-powering algorithm to ensure that your program will finish in time.

Solution.

We first need to implement matrix multiplication, i.e. a function `mult(A,B)` which takes two matrices (represented, for example, by two-dimensional arrays) and returns their product (represented the same way), as well as matrix inverse (as in the previous problem). Once this is done, we can implement the fast powering algorithm exactly as we did before in modular arithmetic. The only additional wrinkle is that we should accommodate negative exponents, which can be achieved by first inverting the matrix and flipping the sign of the exponent if the exponent is negative. Here is an implementation.

```
### Include the functions inv_mod and inv_mat from the previous problem.

def mult_mat(A,B,m):
        C = [[0,0],[0,0]]
        for i in range(2):
                for j in range(2):
                        C[i][j] = A[i][0]*B[0][j] + A[i][1]*B[1][j]
                        C[i][j] %= m
        return C

# The fast-powering algorithm, slightly modified to handle negative powers
def fast_pow(A,e,m):
        # Switch to inverse for a negative power
        if e<0:
                A = inv_mat(A,m)
                e = -e
        C = [[1,0],[0,1]]
        while e>0:
                if e%2 == 1:
                        C = mult_mat(C,A,m)
                A = mult_mat(A,A,m)
                e /= 2
        return C

# I/O
m = int(raw_input())
a,b = map(int,raw_input().split())
c,d = map(int,raw_input().split())
e = int(raw_input())

A = [[a,b],[c,d]]
P = fast_pow(A,e,m)
print P[0][0],P[0][1]
print P[1][0],P[1][1]
```