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

Written problems

1. Use the babystep-giantstep algorithm to compute each of the following discrete logarithms. Show your calculations, e.g. in the form of the table on page 83 of the textbook.

   (a) $\log_{10}[13]_{17}$ (that is, solve $10^x \equiv 13 \pmod{17}$)
   (b) $\log_{15}[16]_{37}$
   (c) $\log_{5}[72]_{97}$

Solution.

(a) We are solving $g^n = h \pmod{p}$, where $g = 10$, $h = 13$, and $p = 17$. There is a solution in $[0,16)$, so it’s enough to take $B = 5$.

$g^B \equiv 6 \pmod{17}$. The inverse of this is $3 \pmod{17}$ (as can be found with the extended Euclidean algorithm). So each entry in the giantstep list is $3$ times the previous, modulo 17. So we construct the following table.

<table>
<thead>
<tr>
<th>$i$</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>$10^i \pmod{17}$</td>
<td>1</td>
<td>10</td>
<td>15</td>
<td>14</td>
<td>4</td>
</tr>
<tr>
<td>$13 \cdot 3^i \pmod{17}$</td>
<td>13</td>
<td>5</td>
<td>15</td>
<td>11</td>
<td>16</td>
</tr>
</tbody>
</table>

The common element is 15, at indices 2 and 2. This shows that $10^2 \equiv 13 \cdot 10^{-5.2}$ (mod 17), hence $10^{12} \equiv 13$ (mod 17). So 12 is a solution to this discrete logarithm problem.

(b) We are solving $g^n = h \pmod{p}$, where $g = 15$, $h = 16$, and $p = 37$. There is a solution in $[0,36)$, so it’s enough to take $B = 7$.

$g^B \equiv 35 \pmod{37}$. The inverse of this is $18 \pmod{37}$ (as can be found with the extended Euclidean algorithm). So each entry in the giantstep list is 18 times the previous, modulo 37. So we construct the following table.

<table>
<thead>
<tr>
<th>$i$</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>$15^i \pmod{37}$</td>
<td>1</td>
<td>15</td>
<td>3</td>
<td>8</td>
<td>9</td>
<td>24</td>
<td>27</td>
</tr>
<tr>
<td>$16 \cdot 18^i \pmod{37}$</td>
<td>16</td>
<td>29</td>
<td>4</td>
<td>35</td>
<td>1</td>
<td>18</td>
<td>28</td>
</tr>
</tbody>
</table>

The common element is 1, at indices 0 and 4. This shows that $15^0 \equiv 16 \cdot 15^{-7.4}$ (mod 37), hence $15^{28} \equiv 16$ (mod 37). So 28 is a solution to this discrete logarithm problem.

(c) We are solving $g^n = h \pmod{p}$, where $g = 5$, $h = 72$, and $p = 97$. There is a solution in $[0,96)$, so it’s enough to take $B = 10$.

$g^B \equiv 53 \pmod{97}$. The inverse of this is $11 \pmod{97}$ (as can be found with the extended Euclidean algorithm). So each entry in the giantstep list is 11 times the previous, modulo 97. So we construct the following table.
The common element is 1, at indices 0 and 5. This shows that \( 5^0 \equiv 72 \cdot 5^{-10} \pmod{97} \), hence \( 5^{50} \equiv 72 \pmod{97} \). So 50 is a solution to this discrete logarithm problem.

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

(b) The inverse of 11 modulo 10 is just 1 (since 11 \( \equiv 1 \) itself), which makes the computation a bit simpler.
\[
\begin{align*}
x &= 6 \pmod{11} \\
\Rightarrow x &= 6 + 11k \text{ for some integer } k \\
6 + 11k &\equiv 2 \pmod{10} \\
11k &\equiv -4 \pmod{10} \\
k &\equiv 6 \pmod{10} \\
\Rightarrow k &= 6 + 10h \text{ for some integer } h \\
x &= 6 + 11(6 + 10h) \\
x &= 72 + 110h \\
\Rightarrow x &\equiv 72 \pmod{110}.
\end{align*}
\]
(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.

\[
x = 2 + 3k \\
k \equiv 1 - 2 \equiv 9 \pmod{10} \\
k \equiv 3 \pmod{10} \\
x = 2 + 3(3 + 10h) = 11 + 30h \\
30h \equiv 3 - 11 \equiv 6 \pmod{7} \\
2h \equiv 6 \pmod{7} \\
h \equiv 4 \cdot 6 \equiv 3 \pmod{7} \\
x = 11 + 30(3 + 7j) = 101 + 210j \\
x \equiv 101 \pmod{210}
\]

(d) We proceed similarly to the previous part.

\[
x = 6 + 8k \\
8k \equiv 3 - 6 \equiv 6 \pmod{9} \\
k \equiv -6 \equiv 3 \pmod{9} \\
x = 6 + 8(3 + 9h) = 30 + 72h \\
72h \equiv 17 - 30 \equiv 4 \pmod{17} \\
4h \equiv 4 \pmod{17} \\
h \equiv 1 \pmod{17} \\
x = 30 + 72(1 + 17j) = 102 + 1224j \\
x \equiv 102 \pmod{1224}
\]

3. Textbook exercise 2.21 (this provides an alternative proof of the “uniqueness” part of the Chinese remainder theorem from the counting argument I presented in class).

**Solution.**

(a) Since \( a \mid c \), there exists an integer \( k \) such that \( c = ak \). Since \( b \mid c \), we can write \( c \equiv 0 \pmod{b} \). Using the expression \( c = ak \), it follows that \( ak \equiv 0 \pmod{b} \). Since \( \gcd(a, b) = 1 \), there is an inverse \( u \) of \( a \) modulo \( b \). Multiplying on both sides by \( u \), it follows that \( (au)k \equiv 0 \pmod{b} \), i.e. \( k \equiv 0 \pmod{b} \). So there exists an integer \( h \) such that \( k = hb \). Therefore \( c = ak = abh \). Since \( h \) is an integer, it follows that \( ab \mid c \), as desired.

(b) If \( c, c' \) are two solutions of all of the congruences in 2.7, then it follows that \( c - c' \equiv 0 \pmod{m_i} \) for each \( i \). Therefore \( m_1 \mid (c - c'), m_2 \mid (c - c'), m_3 \mid (c - c'), \ldots, m_k \mid (c - c') \). Using part (a), \( m_1 \mid (c - c') \) and \( m_2 \mid (c - c') \) imply together that \( m_1m_2 \mid (c - c') \) (note we are using that \( \gcd(m_1, m_2) = 1 \)). Since \( \gcd(m_1m_2, m_3) = 1 \), we can use part (a) again to deduce from \( m_1m_2 \mid (c - c') \) and \( m_3 \mid (c - c') \) that \( m_1m_2m_3 \mid (c - c') \). In general, if we now that \( m_1m_2 \cdot m_\ell \mid (c - c') \) and \( m_{\ell+1} \mid (c - c') \), then we can deduce that...
Let \( G \) be a group, and \( g \) an element of order \( L \) in \( G \). I will write \( g^n \) to be the \( n \)th power of \( g \) with respect to the group operation.

(a) Prove that if \( n \) is an integer dividing \( L \), then \( \text{ord}(g^n) = L/n \).

(b) Prove that \( n \) is an integer relatively prime to \( L \), then \( \text{ord}(g^n) = L \).

(c) Prove that if \( n \) is any integer, then \( \text{ord}(g^n) = L/\gcd(n,L) \). (Observe that this formula mutually generalizes (a) and (b).)

Solutions.

(a) Since \( L \) is the order of \( g \), we know that \( g^L = e \) (where \( e \) is the identity of the group), so certainly \( (g^n)^{L/n} = g^{nL/n} = g^L = e \). On the other hand, if \( m \) is any positive integer less than \( L/n \), then \( mn < L \), so \( (g^n)^m = g^{mn} \neq e \), since \( L \) is the minimum positive integer such that \( g^k = e \). Therefore \( L/n \) is the minimum positive integer \( m \) such that \( (g^n)^m = e \). In other words, \( L/n \) is the order of \( g \).

(b) Suppose that \( m \) is a positive integer such that \( (g^n)^m = e \), i.e. \( g^{mn} = e \). Then by Proposition 2.12, \( mn \) is divisible by the order \( L \) of \( g \). But since \( L \mid mn \) and \( \gcd(L,n) = 1 \), it follows that \( L \mid m \) (one way to see this quickly to notice that \( mn \equiv 0 \pmod{L} \) and multiply by the inverse of \( n \) modulo \( L \), which exists since \( \gcd(n,L) = 1 \)). In particular, \( m \geq L \), since it is a positive multiple of \( L \). So the order of \( g^n \) is at least \( L \). On the other hand, \( (g^n)^L = (g^L)^n = e^n = e \), so the order is also at most \( L \). Therefore the order of \( g^n \) is exactly \( L \).

(c) First, notice that \( (g^n)^{L/\gcd(n,L)} = g^{L(n/\gcd(n,L))} \), and \( n/\gcd(n,L) \) is an integer, so this is equal to \( e^{n/\gcd(n,L)} = e \). So the order of \( g^n \) is at most \( L/\gcd(n,L) \).

To see that it is at least \( L/\gcd(n,L) \), observe that if \( m \) is a positive integer such that \( g^{mn} = e \), then it follows from Proposition 2.12 that \( mn \) is divisible by \( L \). By a result proved in class, this implies that \( m \) is divisible by \( L/\gcd(L,n) \). Since \( m \) is assumed to be positive, this shows that \( m \geq L/\gcd(L,n) \). So the order of \( g^n \) is at least \( L/\gcd(n,L) \), since no smaller value of \( m \) can work.

Therefore the order of \( g^n \) is exactly \( L/\gcd(n,L) \).

Note. You can also, of course, first prove the result in part (c), and then deduce (a) and (b) as special cases.

5. I mentioned in class that in the Elgamal cryptosystem, Bob should create a new (and truly random) ephemeral key (denoted \( k \) on page 72 of the textbook) each time he enciphers a message to Alice using her public key. Otherwise, he exposes himself to a “known-plaintext attack” from Eve. In this problem, we will see why he also should not just perform “small variations” on a previously-used ephemeral key.

(a) Suppose that Bob previously used an ephemeral key \( k_1 \) to send Alice a message \( m_1 \), and that Eve knows what this message is (as well as the enciphered version that Bob
sent to Alice). Suppose Bob now enciphers another message \( m_2 \) to Alice, using the ephemeral key \( k_2 = k_1 + 1 \). Explain how Eve can efficiently detect that this is how Bob has obtained \( k_2 \) from \( k_1 \) (without necessarily determining what \( k_1 \) is), and efficiently extract the message \( m_2 \).

(b) Suppose instead that Bob obtains \( k_2 \) as \( 42k_1 \). Explain how Eve can efficiently detect this, and extract the message \( m_2 \).

(c) Suppose instead that Bob obtains \( k_2 \) as \( k_1 - 5 \). Explain how Eve can efficiently detect this, and extract the message \( m_2 \).

(d) Suppose instead that Bob obtains \( k_2 \) as \( 13k_1 + 2 \). Explain how Eve can efficiently detect this, and extract the message \( m_2 \).

Note. This problem seems make what looks like a strange assumption: that Eve knows the plaintext of a previously-sent message to Alice. In fact, this happens quite often; for example, Bob might sent the same message to many people (e.g. a boilerplate introduction or header information), including both Eve and Alice. For this reason, it is important to make sure that cryptosystems used in practice are not vulnerable to these so-called “known-plaintext” attacks.

Solution.

(a) Suppose that \((c_{11}, c_{12})\) is the first cipher text, \(m_1\) is the first plaintext, and \((c_{21}, c_{22})\) is the second cipher text. Notice that since Eve knows \(m_1\), and she knows that \(c_{12} \equiv A^{k_1} m_1 \pmod{p}\), she can deduce that \(A^{k_2} \equiv c_{12} m_1^{-1} \pmod{p}\) (where she computes \(m_1^{-1}\) with the extended Euclidean algorithm). So throughout, Eve knows the value of \(A^{k_1} \pmod{p}\).

If Bob has chosen \(k_2 = k_1 + 1\) then it will follow that \(c_{21} \equiv g^{k_1 + 1} \equiv g^{k_1} \cdot g \equiv c_{11} g \pmod{p}\). Therefore Eve can detect this fact (without learning \(k_1\)) by noticing that \(c_{21} = (c_{11} g) \pmod{p}\). Once Eve knows that \(k_2 = k_1 + 1\), she can make the following inferences.

\[
A^{k_2} \equiv A^{k_1} \cdot A \pmod{p} \\
\Rightarrow A^{k_2} \equiv c_{12} m_1^{-1} A \pmod{p} \\
\Rightarrow c_{22} \equiv c_{12} m_1^{-1} A m_2 \pmod{p} \\
\Rightarrow c_{22} c_{12}^{-1} m_1 A^{-1} \equiv m_2 \pmod{p}
\]

Since Eve knows all of the terms on the left side (some immediately, some by inverting using the Euclidean algorithm), she can efficiently compute the value of \(m_2\).

To summarize:

If Eve observes that \(c_{21} \equiv c_{11} g \pmod{p}\) Then she can quickly compute \(m_2\) as \(c_{22} c_{12}^{-1} m_1 A^{-1} \pmod{p}\).

(b) If Bob chooses \(k_2 = 42k_1\), then it follows that \(c_{21} \equiv (c_{11})^{42} \pmod{p}\). Eve can detect this relationship between \(k_1\) and \(k_2\) by observing this congruence.

If Eve observes that \(c_{21} \equiv (c_{11})^{42} \pmod{p}\), she may deduce (by raising both sides to the power \(a\)) that \(A^{k_2} \equiv (A^{k_1})^{42} \pmod{p}\). Therefore she may deduce that

Due the night of Thursday 10/13 (hard deadline 4am on 10/14).
\[
c_{22} \equiv A^{k_2} m_2 \pmod{p} \\
\equiv (A^{k_1})^{42} m_2 \pmod{p} \\
\equiv (c_1 m_1^{-1})^{42} m_2 \pmod{p} \\
\Rightarrow c_{22} (c_1 m_1^{-1})^{42} \equiv m_2 \pmod{p}.
\]

Summarizing:

If Eve observes that \( c_{21} \equiv c_{11}^{42} \pmod{p} \)

Then she can quickly compute \( m_2 \) as \( c_{22} (c_1 m_1^{-1})^{42} \pmod{p} \).

(She should use the fast-powering algorithm for the exponentiation, as usual.)

(c) Bob’s choice will leave a fingerprint in the fact that \( c_{21} \equiv c_{11} g^{-5} \pmod{p} \) (where as usual, \( g^{-5} \) denotes the fifth power of \( g^{-1} \pmod{p} \)). Eve can detect this fact about Bob’s choice by noticing that this congruence holds.

If Eve detects this congruence, she may make the following inferences. The first one follows by raising both sides of \( c_{21} \equiv c_{11} g^{-5} \pmod{p} \) to the power \( a \).

\[
A^{k_2} \equiv A^{k_1} A^{-5} \pmod{p} \\
\equiv c_1 m_1^{-1} A^{-5} \pmod{p} \\
\Rightarrow c_{22} \equiv (A^{k_1} A^{-5}) m_2 \pmod{p} \\
\equiv c_1 m_1^{-1} A^{-5} m_2 \pmod{p} \\
\Rightarrow c_{22} c_1^{-1} m_1 A^{-5} \equiv m_2 \pmod{p}.
\]

Summarizing:

If Eve observes that \( c_{21} \equiv c_{11} g^{-5} \pmod{p} \)

Then she can quickly compute \( m_2 \) as \( c_{22} c_1^{-1} m_1 A^{-5} \pmod{p} \).

(d) Bob’s choice will leave a fingerprint in the congruence \( c_{21} \equiv (c_{11})^{13} g^2 \pmod{p} \). Eve can detect this choice by noticing that this congruence holds.

Once Eve detects this fact, she can raise both sides to the power \( a \) (even though she does not know what \( a \) is, she still knows that the resulting congruence must hold!) and make the following inferences.

\[
A^{k_2} \equiv (A^{k_1})^{13} A^2 \pmod{p} \\
\equiv (c_1 m_1^{-1})^{13} A^2 \pmod{p} \\
\Rightarrow c_{22} \equiv (c_1 m_1^{-1})^{13} A^2 m_2 \pmod{p} \\
\Rightarrow c_{22} (c_1 m_1^{-1})^{13} A^{-2} \equiv m_2 \pmod{p}.
\]
Summarizing:

If Eve observes that \( c_{21} \equiv (c_{11})^{13} g^2 \pmod{p} \)

Then she can quickly compute \( m_2 \) as \( c_{22} (c_{12}^{-1} m_1)^{13} A^{-2} \pmod{p} \).

6. Write a function to do the following task: generate a sequence of random \( B \) bit nonnegative numbers (that is, integers with \( 0 \leq n < 2^B \), where \( B \) is given as input), until a number is repeated. The function should return the number of numbers generated (including the repeat at the end). Write a second function that runs your first function 1000 times and averages the results. Run your second function for all values of \( B \) from 1 to 20 and report the results. See if you can identify a pattern. (You may already know what pattern to expect to see; the result is slightly counterintuitive and is referred to as the “birthday paradox.” This pattern turns out to be of crucial importance in studying the danger posed by certain randomized attacks on cryptosystems.)

**Solution.**

The following two functions can be used to generate this data.

```python
import random
random.seed()

def trial(N):
    elts = set()
    cnt = 0
    while True:
        n = random.randrange(N)
        if n in elts:
            return cnt+1
        elts.add(n)
        cnt += 1

def avg_trial(N,num_trials):
    tot = 0
    for i in xrange(num_trials):
        tot += trial(N)
    return float(tot)/num_trials
```

Here is the output when I run this from \( B = 1 \) to \( B = 20 \).

```python
>>> for B in xrange(1,21):
...    print B,avg_trial(2**B, 1000)
...
1 2.479
2 3.187
3 4.278
4 5.777
5 7.806
```

Due the night of Thursday 10/13 (hard deadline 4am on 10/14).
6 10.634
7 14.952
8 20.634
9 29.872
10 40.698
11 57.696
12 79.572
13 115.719
14 157.334
15 232.017
16 313.552
17 441.47
18 649.568
19 936.197
20 1277.512

The curious feature here is that, although there are $2^{20}$, approximately 1 million, 20-bit integers, you only have to list a little over a thousand of them before you encounter the same integer twice. In general, if you are selecting from $N$ possibly numbers, you will choose the same number twice after you have sampled a small multiple of $\sqrt{N}$ elements at random.

7. Go to the following demonstration assignment, and open the problem called "DLP benchmarking." This problem allows you to benchmark the effectiveness of different algorithms to solve the discrete logarithm problem.

https://www.hackerrank.com/m158-2016-demos/

Note that case number $n$ uses a prime of length exactly $\left\lfloor \frac{n}{2} \right\rfloor + 17$ bits. So by seeing which cases a particular algorithm solves, you can gauge the length of prime it is able to handle within the hackerrank time limit.

(a) Before submitting any code, estimate the number of test cases that you think a naive trial-and-error approach (i.e. testing all possible exponents, starting from 0, until one is found that works) will correctly solve. Then implement such an approach (perhaps using your submission to Problem Set 3, if you used a trial-and-error approach there), submit it, and check how close your estimate was.

(b) Estimate how many cases an implementation of Babystep-Giantstep will complete correctly. If you have a working BSGS implementation (e.g. after completing the coding portion of the assignment), use it to check your answer, but do not submit the code until you have made your estimate.

Note. For this assignment, you will receive full point if you make a good faith effort and your reasoning makes sense. I may ask estimations like this on future exams; in this case, I would mark your answer correct if you estimate the number of bits (in the length of $p$) that the program can handle within 10 bits of the actual figure.

Solution.

(a) Working with $b$-bit primes, a trial and error approach will have to loop over up to approximately $2^b$ possible solutions before finding the correct one. If each iteration of
the look takes $k$ clock cycles, this means the algorithm will take $2^b k$ clock cycles to finish, which will need to be less than about $10 \cdot 2^{31} \approx 2^{34}$ to finish within 10 seconds. Each iteration through the loop involves three arithmetic operations, plus some overhead for the loop. Guided by problem 7 on the first problem set (where each iteration through a loop also required just one or two arithmetic operations), we might guess that $k$ is somewhere on the order of $128 = 2^7$ clock cycles by loop. So we need $2^7 \cdot 2^b \leq 2^{34}$, i.e. $b \leq 27$.

Running the following straightforward implementation of trial-and-error, I found that it started to fail on test case number 20, i.e. when $p$ was 27 bits long (so it succeeded for all primes up to 26 bits in length). So the estimate above was in fact very close to exactly correct.

```
p,g,h = map(int,raw_input().split())

  powg = 1
  e = 0
  for d in xrange(p-1):
    if powg == h:
      print e
      break
    e += 1
    powg = powg*g % p
```

Note that other implementations give slightly different results. For example, the following implementation is slower since it computes each power of $g$ from scratch with each iteration (instead of multiplying the previous one by $g$):

```
p,g,h = map(int,raw_input().split())

  for d in xrange(p-1):
    if pow(g,d,p) == h:
      print d
      break
```

It first fails on test case 19, i.e. when $p$ is 26 bits long. So the performance difference can be detected, but it is quite minor.

(b) Babystep-giantstep involves construction two lists of length roughly $\sqrt{p}$, and each entry of each list requires two arithmetic operations to find (one multiplication, one reduction modulo $p$). Then we must check for a collision; the speed of this will depend on implementation, but we can assume that this takes a small multiple of $\sqrt{p} \log \sqrt{p}$ operations, which will probably overshadow the operations needed to create the lists in the first place. So if $p$ is $b$ bits long, the whole algorithm will require a small multiple of $b \cdot 2^{b/2}$ arithmetic operations. Since we know that our final answer is unlikely to be much more that 64 bits (since $2^{64/2} = 2^{32}$ already), we might make it easy on ourselves and just guess that we’ll have to do about $642^{b/2} = 2^{b/2+6}$ arithmetic operations. Since we’ve seen previously that each arithmetic operation seems to take on the order of 30 clock cycles, we might guess that this will take about $2^{b/2+6+5} = 2^{b/2+11}$ clock cycles. Like in part (a), this will probably need to be less than $2^{34}$ or so, i.e. $b$ should be at most 46.

In fact, when I run my sample implementation (in problem 8, below) of BSGS (using
a \texttt{dict()} to find collisions), it first fails in test case 54, i.e. for a 44 bit prime. So the estimate above was optimistic, but not far off the mark.

**Programming problems**

Full formulation and submission: [https://www.hackerrank.com/m158-2016-pset-4](https://www.hackerrank.com/m158-2016-pset-4)

8. Solve the discrete logarithm problem, where the modulus is a 36 bit prime number.

**Solution.**

Here is a sample implementation of babystep-giantstep, using a \texttt{dict()} to find collisions, that runs fast enough for 36 bit primes (in fact, for up to 43 bit primes, as found above).

```python
import math

# 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 \([u0,v0,r0]\)

# Babystep-Giantstep algorithm
# Finds a solution \(g^n = h \mod p\) with \(0 \leq n < \text{max\_soln}\), if possible
# Returns None if there is no such solution
def bsgs(p,g,h,max_soln):
    blist = []
glist = []
    B = int(math.sqrt(max_soln))+1
    gb = pow(g,B,p)
gbinv = ext_euclid(gb,p)[0]%p
    bs = 1 # Next item to add to babystep list
    gs = h # Next item to add to giantstep list
    for i in xrange(B):
        blist += [bs]
glist += [gs]
        bs = bs * g % p
        gs = gs * gbinv % p
    # Make reverse look-up dictionary for the giantstep list
    grev = dict()
    for j in xrange(B):
        grev[glist[j]] = j
    # Find a collision and return the result
    for i in xrange(B):
        # Check for collision
        if grev[blist[i]] != i:
            return blist[i]
```

Due the night of Thursday 10/13 (hard deadline 4am on 10/14).
if bslist[i] in gsrev:
    return i + B*gsrev[bslist[i]]
# If we get to this line, then no collision was found
return None

p,g,h = map(int,raw_input().split())
print bsgs(p,g,h,p-1)

9. From a list of congruences \( x \equiv a_i \pmod{m_i} \), where the integers \( m_i \) are pairwise relatively prime, determine integers \( a, m \) such that the list is equivalent to the single congruence \( x \equiv a \pmod{m} \).

**Solution.**

Here is sample implementation, which merges the congruences two at at time. The method used to merge two congruences is the idea I showed in class, using the output of the Extended Euclidean algorithm directly.

# 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]

# Merge two congruences
def merge_two(a1,m1,a2,m2):
    [u,v,g] = 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]

10. Alice and Bob use Diffie-Hellman key exchange on a regular basis, but they are not choosing their secret numbers $a$ and $b$ randomly. As a result, the secret numbers that Alice chooses during different key exchanges are usually close to each other; Bob make the same mistake. More precisely: you may assume that if Alice uses $a_0$ as her secret number one day and $a_1$ on another day, then $|a_0 - a_1| \leq 2^{20}$ (and similarly with Bob’s numbers).

Eve has managed to learn one of Alice and Bob’s previous shared secret $S_0$, corresponding two exchanged numbers $A_0, B_0$ (from Alice to Bob and vice versa). Later, she intercepts two more exchanged numbers $A_1, B_1$, and wishes to extract the new shared secret $S_1$ corresponding to these. From all of this information, and the knowledge about how Alice and Bob are choosing their secret numbers, determine $S_1$.

*Hint.* Use similar ideas to those used in problem 5.

*Solution.*

Eve knows that, whatever $a_0$ and $a_1$ are, they differ by some difference $d_a$ that is at most $2^{20}$ in magnitude. To determine what this difference is, Eve could simply compute $A_1 \cdot A_0^{-1}$ (mod $p$), and see if it is equal to $g^d$ (mod $p$), trying all values of $d$ from $-2^{20}$ to $2^{20}$ inclusive. She can similarly identify the difference $d_b = b_1 - b_0$. Note that strictly speaking, she only determines this difference modulo the order of $g$, but this is good enough for her purposes, since it will not affect the final result.

This extraction of $d_a$ and $d_b$ is encapsulated in the function `expDiff` below. Note that to write it, I’ve written an auxiliary function `powmod(g, e, p)` which behaves just like the built-in `pow(g, e, p)` except that it can handle nonpositive exponents. This removes the need for some casework elsewhere.

Finally, Eve can compute the second shared secret as follows.

$$
S_1 \equiv g^{a_1b_1} \equiv g^{(a_0 + d_a)(b_0 + d_b)} \pmod{p} \\
\equiv g^{a_0b_0 + a_0d_b + d_a b_0 + d_a d_b} \pmod{p} \\
\equiv S_0 A_0^{d_a} B_0^{d_b} g^{d_a d_b}
$$

All of the variables in this last expression are known to Eve, so she can compute $S_1$.

An implementation is shown below.

```python
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
```

\[\text{Due the night of Thursday 10/13 (hard deadline 4am on 10/14).} \]
u0,v0,r0 = u1,v1,r1
u1,v1,r1 = u2,v2,r2
return [u0,v0,r0]

def expDiff(p,g,C0,C1):
e = 0
ge = 1  # g^e mod p
while True:
    if C0 * ge % p == C1: return e
    if C1 * ge % p == C0: return -e
    e += 1
    ge = ge * g % p

# Powering function that can accept positive or negative exponents
# Assumes that g is a unit modulo p
def powmod(g,e,p):
    if e >= 0: return pow(g,e,p)
    else:
        ginv = ext_euclid(g,p)[0]%p
        return pow(ginv,-e,p)

def extract(p,g,A0,B0,S0,A1,B1):
da = expDiff(p,g,A0,A1)
db = expDiff(p,g,B0,B1)
return S0 * powmod(A0,db,p) * powmod(B0,da,p) * powmod(g,da*db,p) % p

# I/O
p,g = map(int,raw_input().split())
A0,B0,S0 = map(int,raw_input().split())
A1,B1 = map(int,raw_input().split())

print extract(p,g,A0,B0,S0,A1,B1)