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

1. Evaluate the discrete logarithm \( \log_{40} [33]_{73} \) 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 \( N = \text{ord}_{73}[40]_{73} \). 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 \( x = \log_{40} [33]_{73} \).

**Solution.**

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

\[
N = 2^33^2.
\]

(b)

\[
\begin{align*}
g_1 & \equiv 40^{N/8} \equiv 40^9 \pmod{73} \\
& \equiv 10 \pmod{73} \\
h_1 & \equiv 33^{N/8} \equiv 33^9 \pmod{73} \\
& \equiv 63
\end{align*}
\]

\[
\begin{align*}
g_2 & \equiv 40^{N/9} \equiv 40^8 \pmod{73} \\
& \equiv 55 \pmod{73} \\
h_2 & \equiv 33^{N/9} \equiv 33^8 \pmod{73} \\
& \equiv 55 \pmod{73}
\end{align*}
\]

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

(c) To evaluate \( \log_{10} [63]_{73} \), 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, \( \log_{55} [55]_{73} \), is immediate: it is simply \( y_2 = 1 \).

(d) We must solve the system

\[
\begin{align*}
x & \equiv 5 \pmod{8} \\
x & \equiv 1 \pmod{9}
\end{align*}
\]

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

\[
\begin{align*}
x &= 5 + 8k \\
8k & \equiv 1 - 5 \equiv 5 \pmod{9} \\
k & \equiv -5 \equiv 4 \pmod{9} \\
x &= 5 + 8(4 + 9h) = 37 + 72h \\
x & \equiv 37 \pmod{72}
\end{align*}
\]
So the desired logarithm is $37 \pmod{72}$.

2. The following sequence of problems will prove part (b) of the theorem from class on Friday 10/14, showing the correctness of the Pohlig-Hellman algorithm in a slightly more general form than Theorem 2.31 in the textbook.

(a) Textbook exercise 1.13.

(b) Prove that if $r_1, r_2, \ldots, r_t$ are pairwise relatively prime (that is, $\gcd(r_i, r_j) = 1$ for all $i \neq j$), and $N = r_1 r_2 \cdots r_t$, then the $t$ numbers $N/r_1, N/r_2, \ldots, N/r_t$ have greatest common divisor 1. (Note: you are proving that the greatest common divisor of the entire list is 1; this is not the same as saying that the numbers are pairwise relatively prime).

(c) Suppose that $g, h$ are elements of a group $G$, $N$ is a positive integer, and that $N$ factors into prime powers as $N = q_1^{e_1} q_2^{e_2} \cdots q_i^{e_i}$. Suppose also that $x$ is an integer such that $g^{x \cdot N/q_i} = h^{N/q_i}$ for $i = 1, 2, \ldots, t$. Deduce from parts (a) and (b) that in fact $g^x = h$.

Note. The proof of Theorem 2.31 in the textbook will likely be helpful. Note however that you should prove part (c) without assuming that the order of $g$ is equal to $N$, so the argument in the textbook cannot be applied verbatim.

Solution.

(a) We prove instead the more general statement:

**Lemma.** If $a_1, a_2, \ldots, a_k$ are integer with greatest common divisor $g$, then there exist integers $u_1, u_2, \ldots, u_k$ such that

$$a_1 u_1 + a_2 u_2 + \cdots + a_k u_k = g.$$ 

The desired result follows immediately from the lemma; it is the case $g = 1$. So we need only prove the lemma. The most straightforward way is to use induction.

The case $k = 2$ is known from the extended Euclidean algorithm. Now assume that the lemma is known for lists of $k - 1$ integers, and we wish to prove it for lists of $k$ integers, where $k \geq 3$.

Let $g' = \gcd(a_1, a_2, \ldots, a_{k-1})$. By induction hypothesis, there are integers $u'_i$ such that

$$a_1 u'_1 + a_2 u'_2 + \cdots + a_{k-1} u'_{k-1} = g'.$$

By the extended Euclidean algorithm, there exist integers $u'', v''$ such that $g' u'' + a_k v'' = \gcd(g', a_k)$. Substituting the previous equation and distributing the multiplication, this gives

$$a_1 (u'_1 u'') + a_2 (u'_2 u'') + \cdots + a_{k-1} (u'_{k-1} u'') + a_k (v'') = \gcd(g', a_k).$$

It only remains to show that $\gcd(g', a_k) = \gcd(a_1, a_2, \ldots, a_k)$. Observe first of all that $\gcd(g', a_k)$ is certainly a divisor of all of $a_1, a_2, \ldots, a_k$, since $g'$ is a divisor of $a_1, \ldots, a_{k-1}$ and $\gcd(g', a_k)$ must divide both $g'$ and $a_k$. On the other hand, any common divisor $d$ of $a_1, a_2, \ldots, a_k$ must divide any integer linear combination of $a_1, a_2, \ldots, a_k$, so by the equation above, $d$ divides $\gcd(g', a_k)$. Hence any common divisor of $a_1, a_2, \ldots, a_k$ is at most $\gcd(g', a_k)$. So $\gcd(g', a_k)$ is indeed the greatest common divisor.
Therefore, taking
\[ u_1 = u_1' u'', \ u_2 = u_2' u'', \ \ldots, \ u_{k-1} = u_{k-1}' u'', \ u_k = v'' \]
we have proved that \( a_1 u_1 + \cdots + a_k u_k = \gcd(a_1, \cdots, a_k) \), as desired.

Remark. This proof provides a straightforward way to efficiently compute the numbers \( u_i \), e.g. in the following couple lines of code (assuming the extended Euclidean algorithm is already implemented).

```python
def compute_u(a_list):
    g = 0
    u_list = []
    for ak in a_list:
        u1,v1,g = ext_euclid(g,ak)
        u_list = [u1*ui for ui in u_list] + [v1]
    return u_list
```

(b) I claim that there can be no prime number dividing all of the numbers \( N/r_i \). Suppose otherwise, and let \( p \) be such a prime. Then since \( p \) divides \( N/r_1 = r_2 r_3 \cdots r_t \), it must divides one of the numbers \( r_i \) \((i \neq 1)\). But \( p \) must also divides \( N/r_i \), which is \( r_1 r_2 \cdots r_{i-1} r_{i+1} \cdots r_t \), so \( p \) must divide a second number \( r_j \) \((j \neq i)\). But then \( p \) must divide both \( r_i \) and \( r_j \), contradicting the assumption that \( r_i \) and \( r_j \) are relatively prime.

This is impossible, so it follows that there is no prime dividing all of the numbers \( N/r_i \). But this shows that the greatest common divisor of the numbers \( N/r_i \) is 1, since any set of number with some common divisor (greater than 1) would have to have some prime common divisor.

(c) By part (b), the exponents \( N/q_i^{e_i} \) have greatest common divisor 1. So by part (a), there exist integers \( u_i \) such that \( \sum_{i=1}^{t} \frac{N}{q_i^{e_i}} u_i = 1 \). Now observe that:

\[
\prod_{i=1}^{t} \left( g^{x \cdot \frac{N}{q_i^{e_i}}} ight)^{u_i} = \prod_{i=1}^{t} \left( h^{N/q_i^{e_i}} ight)^{u_i} \\
g^{x \cdot \sum_{i=1}^{t} N u_i / q_i^{e_i}} = h^{\sum_{i=1}^{t} N u_i / q_i^{e_i}} \\
g^x = h,
\]

where the last line uses the fact that \( \sum_{i=1}^{t} N u_i / q_i^{e_i} = 1 \).

3. Textbook exercise 3.1, parts (a),(b),(c).

Solution.

(a) 97 is prime, so \( \phi(97) = 96 \). To take the 19th root, we should compute \( 19^{-1} \pmod{96} \),
which is 91, and compute as follows:

\[
x^{19} \equiv 36 \pmod{97}
\]
\[
\Rightarrow x^{19 \cdot 91} \equiv 36^{91} \pmod{97}
\]
\[
\Rightarrow x \equiv 36^{91} \pmod{97} \quad \text{(since } 19 \cdot 91 \equiv 1 \pmod{\phi(97)})
\]
\[
x \equiv 36 \pmod{97}
\]

So in fact 36 is its own 19th root modulo 97.

(b) We follow the same procedure as before. I will be somewhat more brief in the shown steps.

\[
x^{137} \equiv 428 \pmod{541}
\]
\[
\phi(541) = 540 \quad \text{(541 is prime)}
\]
\[
137^{-1} \equiv 473 \pmod{540}
\]
\[
\Rightarrow x \equiv 428^{473} \pmod{541}
\]
\[
x \equiv 213 \pmod{541}
\]

(c) Following the same procedure:

\[
x^{73} \equiv 614 \pmod{1159}
\]
\[
1159 = 19 \cdot 61
\]
\[
\phi(1159) = 18 \cdot 60 = 1080
\]
\[
73^{-1} \equiv 577 \pmod{1080}
\]
\[
\Rightarrow x \equiv 614^{577} \pmod{1159}
\]
\[
x \equiv 158 \pmod{1159}
\]

4. Textbook exercise 3.5, parts (a),(b),(c),(d).

\textit{Hint for part (c):} Prove that \([n]_{MN}\) is a unit if and only if both \([n]_M\) and \([n]_N\) are units. Then apply the Chinese Remainder Theorem.

\textbf{Solution.}

(a) As seen in class, \(\phi(pq) = pq - p - q + 1 = (p-1)(q-1)\). We know that \(\phi(p) = p - 1\) and \(\phi(q) = q - 1\). So \(\phi(pq) = \phi(p)\phi(q)\).

(b) A number from 0 to \(p^j - 1\) is relatively prime to \(p^j\) if and only if it is not divisible by \(p\). The numbers divisible by \(p\) are precisely \(0, p, 2p, \ldots, p^j - p\). There are \(p^j - 1\) such numbers. So \(\phi(p^j) = p^j - p^j - 1\). In particular, \(\phi(p^2) = p^2 - p\).

(c) First, I prove the statement from the hint: \(n\) is a unit modulo \(MN\) if and only if it is both a unit modulo \(M\) and a unit modulo \(N\).

Note that \(n\) is a unit modulo \(MN\) if and only if none of its prime factors divide \(MN\). But this is equivalent to saying that none of its prime factors divide either \(M\) or \(N\), i.e. to saying that \(n\) is relatively prime to both \(M\) and \(N\). So \(n\) is relatively prime to \(MN\) if and only if it is relatively prime to both \(M\) and \(N\). In other words, \([n]_{MN}\) is a unit if and only if both \([n]_M\) and \([n]_N\) are units.
Now, we remarked in class the one interpretation of the Chinese remainder theorem is that, if \( \gcd(M, N) = 1 \), then the function
\[
\mathbb{Z}/MN \rightarrow \mathbb{Z}/M \times \mathbb{Z}/N
\]
given by \([a]_{MN} \mapsto ([a]_M, [a]_N)\) is one-to-one and onto (i.e., it is a bijection). The previous paragraph shows that when this function is restricted only to the units modulo \( MN \), its image is exactly equal to the set of pairs \(([a_1]_M, [a_2]_N)\) where both \([a_1]_M\) and \([a_2]_N\) are units. Since the function is one-to-one, this shows that the number of units modulo \( MN \) is equal to the number of pairs consisting of a unit modulo \( M \) and a unit modulo \( N \). The number of such pairs is simply \( \phi(M)\phi(N) \), since there are \( \phi(M) \) choices for the first element of the pair, and \( \phi(N) \) choices for the second. This shows that \( \phi(MN) = \phi(M)\phi(N) \).

(d) We are assuming \( p_1, \ldots, p_r \) are the primes dividing \( N \). Let \( e_1, \ldots, e_r \) be the exponents of these primes, i.e.
\[
N = p_1^{e_1} p_2^{e_2} \cdots p_r^{e_r}.
\]
Applying the formula from (c) repeatedly, we obtain
\[
\phi(N) = \phi(p_1^{e_1}) \phi(p_2^{e_2}) \cdots \phi(p_r^{e_r})
\]
\[
\cdots
\]
\[
= \phi(p_1^{e_1}) \phi(p_2^{e_2}) \cdots \phi(p_r^{e_r}).
\]
Applying the formula from part (a), this becomes
\[
\phi(N) = \prod_{i=1}^{r} \phi(p_i^{e_i})
\]
\[
= \prod_{i=1}^{r} \left( p_i^{e_i} - p_i^{e_i-1} \right)
\]
\[
= \prod_{i=1}^{r} p_i^{e_i} \left( 1 - \frac{1}{p_i} \right)
\]
\[
= \prod_{i=1}^{r} p_i^{e_i} \prod_{i=1}^{r} \left( 1 - \frac{1}{p_i} \right)
\]
\[
= N \cdot \prod_{i=1}^{r} \left( 1 - \frac{1}{p_i} \right),
\]
which is the desired formula.

5. Textbook exercise 3.7.

Solution.

(a) Bob sends \( m^e \pmod N \), i.e. \( 892383^{103} \pmod{2038667} \), which works out to 45293.
(b) The other prime is \( q = N/p = 2038667/1301 = 1567 \). So \( \phi(N) = (1301 - 1)(1567 - 1) = 2035800 \), and the decryption exponent is \( e^{-1} \pmod{\phi(N)} \), i.e. \( 103^{-1} \pmod{2035800} \), which works out (using the extended Euclidean algorithm) to \( d = 810367 \).

(c) The plaintext should be \( e^d \pmod{N} \), i.e. \( 317730^{810367} \pmod{2038667} \), which works out (using fast powering) to \( 514407 \).

6. Textbook exercise 3.11, part (a). You should also solve part (b), but you don’t need to write it up; instead you will write a program to break the cryptosystem in one of the programming problems.

Solution.

Observe that by Fermat’s little Theorem,

\[
    g_1 \equiv (g^{p-1})^{r_1} \pmod{p} \\
    \equiv 1^{r_1} \pmod{p} \\
    \equiv 1 \pmod{p}
\]

and similarly,

\[
    g_2 \equiv 1 \pmod{q}
\]

It follows form this that \( c_1 \equiv m \cdot 1^{s_1} \equiv m \pmod{p} \) and similarly \( c_2 \equiv m \pmod{q} \). Therefore, if \( m' \) is the number that Alice obtains by applying the Chinese Remainder theorem to solve \( m' \equiv c_1 \pmod{p} \) and \( m' \equiv c_2 \pmod{q} \), it follows that \( m \equiv m' \) both modulo \( p \) and modulo \( q \). So \( p \) and \( q \) both divide \( m - m' \), implying that \( N = pq \) divides \( m - m' \), i.e. \( m \equiv m' \pmod{N} \). Since the message is defined as an element of \( \mathbb{Z}/N \), this means that indeed Alice has constructed the original message \( m \).


Note. You should compare your solution to this problem to your solution to problem 4 on PSet 2. Both use a very similar idea that is quite versatile.

Solution.

Eve may compute, using the extended Euclidean algorithm, that

\[
    e_1u + e_2v = 1,
\]

where \( u = 252426389 \) and \( v = -496549570 \). Therefore she can obtain the original message \( m \) as follows.

\[
    m \equiv m^1 \pmod{N} \\
    \equiv m^{e_1u + e_2v} \pmod{N} \\
    \equiv (m^{e_1})^u(m^{e_2})^v \pmod{N} \\
    \equiv c_1^u c_2^v \pmod{N} \\
    \equiv 1244183534^{252426389} \cdot 732959706^{-496549570} \pmod{1889570071}
\]
We can compute that \( c_2^{-1} \equiv 1873807620 \pmod{1889570071} \) with the extended Euclidean algorithm, and therefore re-express the negative power as a positive power and compute \( m \) as follows (using the fast-powering algorithm twice).

\[
m \equiv 1244183534^{252426389} \cdot 1873807620^{196549570} \pmod{1889570071} \\
\equiv 1031756109 \cdot 603385073 \pmod{1889570071} \\
\equiv 1054592380 \pmod{1889570071}
\]

So this was Bob’s plaintext; Eve has been able to recover it without factoring the modulus. She may check her answer by verifying that \( m^{c_1} \equiv c_1 \pmod{N} \) and \( m^{c_2} \equiv c_2 \pmod{N} \).

**Programming problems**

8. Write a program to break the cryptosystem described in problem 3.11. Your program will receive a public key (but not the corresponding private key) and a cipher text, and it should print the original plaintext.

**Solution.**

The key to breaking the cryptosystem lies in the first observation of the solution to (a), namely that

\[ g_1 \equiv 1 \pmod{p}. \]

(Remember that Eve knows \( g_1 \), since it is part of the public key.) In other order \( p \mid (g_1 - 1) \). So Eve knows a multiple of \( p \). But because of the Euclidean algorithm, this is almost as good as knowing \( p \) itself: she can compute \( \gcd(N, g_1 - 1) \). This greatest common divisor must be a multiple of \( p \) (since \( p \) is a common divisor), and a factor of \( N = pq \), so it is equal to either \( p \) or \( pq = N \). In the second case, \( g_1 \) would have to be 1 \pmod{N}, in which case Eve can simply observe that \( m = c_1 \) and not do any more work. But if \( g_1 \not\equiv 1 \pmod{N} \), then this \( \gcd \) will equal \( p \). So Eve can compute \( p \), then compute \( q \) as \( N/p \). She now knows everything that Alice knows, and hence can decipher messages.

Here is an implementation. I have omitted the code for the `ext_euclid` and `merge` functions, since these have already been written for previous problem sets (see the solution to problem 9 on problem set 4).

```python
# Omitted: functions ext_euclid and merge_two (see PSet 4 problem 9 solution)

def analyze(N,g1,g2,c1,c2):
    if g1 % N == 1: return c1 # Easy special case
    # If g1 % N != 1, we know that p = gcd(g1-1,N), and can factor N
    p = ext_euclid(g1-1,N)[2]
    q = N/p
    return merge_two(c1,p,c2,q)[0]

# I/O
N,g1,g2 = map(int,raw_input().split())
c1,c2 = map(int,raw_input().split())
print analyze(N,g1,g2,c1,c2)
```
9. Write a program which takes an integer \( N \) (possibly up to 1024 bits long) that is guaranteed to factor into prime powers of at most 16 bits in length, and prints those prime power factors. If these prime power factors are \( q_i^{e_i} \) (for \( i = 1, 2, \ldots, t \)), then your program should print them in order of \( q_i \) (i.e. in order of the primes, not in order of prime powers).

**Solution.** One way to do this (by no means the most efficient possible, but good enough for 16-bit factors) is to remove the prime factors from \( N \) one by one, as follows; start counting from 2, and every time you encounter a factor \( p \) of \( N \), divide \( N \) by it as many times as possible to obtain a new value \( N' = N/p^e \). Now add \( p^e \) to a running list of factors, increment \( p \), and continue. Eventually \( N \) will be replaced by 1, and we will be finished.

There is one thing you have to worry about in this algorithm: how do we know that each factor \( p \) found in this way is prime? The answer is that since we try potential factors in order, so by the time we check whether \( N \) is divisible by \( p \), we have already removed form \( N \) all prime factors smaller than \( p \). So \( p \) is necessarily prime, because otherwise none of its prime factors could divide \( N \) anymore.

Here is an implementation.

```python
def ppfacts(N):
    res = []
    p = 2
    while N > 1:
        if N%p == 0:
            q = 1
            while N%p == 0:
                q *= p
                N /= p
            res += [q]
        p += 1
    return res
```

---

10. 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 \( \mathbb{Z}/p \), 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 \( a \) (mod \( p \)). Your program should print an element \( e \) of \( \mathbb{Z}/N \) 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

# Omitted: implementation of ext_euclid

# Babystep-Giantstep algorithm
# Finds a solution g^n = h mod p with 0 <= n < max_soln, if possible
# Returns None if there is no such solution
def bsgs(p, g, h, max_soln):
    bslist = []
gslist = []
    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):
        bslist += [bs]
gslist += [gs]
        bs = bs * g % p
        gs = gs * gbinv % p
    # Make reverse look-up dictionary for the giantstep list
    gsrev = dict()
    for j in xrange(B):
        gsrev[gslist[j]] = j
    # Find a collision and return the result
    for i in xrange(B):
        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, a, N = map(int, raw_input().split())
print bsgs(p, g, h, N)

11. Implement the Pohlig-Hellman algorithm. You have written all of the main ingredients in previous programming problems (including the previous two problems on this assignment). Specifically, you will be given a discrete logarithm problem for with the modulus p is a “weak prime” in the sense that p – 1 factors into small prime powers (all 16 bits or smaller).

Solution.

We make use of the following functions, written in previous homework solutions. To save space, I will not recopy the code below.

- The extended euclidean algorithm ext_euclid(a, b) (e.g. in PSet 4 #9)
- BSGS with known element order bsgs(p, g, h, max_soln) (e.g. # 10 on this problem set)
- The function merge_list (and its helper function, merge_two) implementing the Chinese remainder theorem (e.g. PSet 4 # 9).
- The function \texttt{ppfact(N)}, extracting the prime-power factors of a given integer \(N\) (e.g. \# 9 on this problem set).

The last helper function we need is a function to factor \(p - 1\) and return its prime-power factors. We can implement this by trial division, similarly to the solution of problem 6 on PSet 1.

From here, we need only assume these pieces according to the steps listed in Theorem 2.31 of the text. The implementation is shown below.

```python
### Omitted: source for the functions ext_euclid, bsgs, merge_list, merge_two, and ppfact. (Note that bsgs requires "import math")

def ph_dlp(p, g, a):
    pp = ppfacts(p-1)
    # a[i] mod m[i] will be the congruences that we will merge
    y = []
    m = []
    for qe in pp: # qe is a prime-power factor q^e of p-1
        gi = pow(g, (p-1)/qe, p)
        hi = pow(a, (p-1)/qe, p)
        yi = bsgs(p, gi, hi, qe)
        # Add the congruence yi mod qe to the list
        y += [yi]
        m += [qe]
    return merge_list(y, m)[0]

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

Due the night of Thursday 10/20 (hard deadline 4am on 10/21).