Written problems

1. Textbook exercise 3.1 (same in first edition).

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:

$$\begin{aligned} x^{19} &\equiv 36 \pmod{97} \\ \Rightarrow x^{19\cdot 91} &\equiv 36^{91} \pmod{97} \\ \Rightarrow x &\equiv 36^{91} \pmod{97} \quad (\text{since } 19\cdot 91 \equiv 1 \pmod{\phi(97)}) \\ x &\equiv 36 \pmod{97} \end{aligned}$$

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.

$$\begin{aligned} x^{137} &\equiv 428 \pmod{541} \\ \phi(541) &= 540 \text{ (541 is prime)} \\ 137^{-1} &\equiv 473 \pmod{540} \\ \Rightarrow x &\equiv 428^{473} \pmod{541} \\ x &\equiv 213 \pmod{541} \end{aligned}$$

(c) Following the same procedure:

$$\begin{aligned} x^{73} &\equiv 614 \pmod{1159} \\ 1159 &= 19\cdot 61 \\ \phi(1159) &= 18\cdot 60 = 1080 \\ 73^{-1} &\equiv 577 \pmod{1080} \\ \Rightarrow x &\equiv 614^{577} \pmod{1159} \\ x &\equiv 158 \pmod{1159} \end{aligned}$$

(d) Following the same procedure:

$$\begin{aligned} x^{751} &\equiv 677 \pmod{8023} \\ 8023 &= 71\cdot 113 \\ \phi(8023) &= 70\cdot 112 = 7840 \\ 751^{-1} &\equiv 7151 \pmod{7840} \\ \Rightarrow x &\equiv 677^{7151} \pmod{8023} \\ x &\equiv 1355 \pmod{8023} \end{aligned}$$

(e) Following the same procedure:

$$\begin{aligned}
x^{38993} &\equiv 328047 \pmod{401227} \\
401227 &= 607 \cdot 661 \\
\phi(401227) &= 606 \cdot 660 = 399960 \\
38993^{-1} &\equiv 265457 \pmod{399960} \\
\Rightarrow x &\equiv 328047^{265457} \pmod{401227} \\
x &\equiv 36219 \pmod{401227}
\end{aligned}$$

2. Textbook exercise 3.2 (same in first edition).

Solution.

(a) By the primitive root theorem, there is a primitive root g modulo p. Since both x and c must be units modulo p, we can defined the following discrete logarithms.

$$\begin{aligned}
y &= \log_g(x) \\
d &= \log_g(c)
\end{aligned}$$

As shown on an earlier homework (problem 7 on PSet 2), these logarithms are well-defined as elements of $\mathbf{Z}/(p-1)$. We can then express the congruence we wish to solve as follows.

$$\begin{aligned}
x^e &\equiv c \pmod{p} && (1) \\
\Leftrightarrow g^{ey} &\equiv g^d \pmod{p} && (2) \\
\Leftrightarrow ey &\equiv d \pmod{p-1} && (3)
\end{aligned}$$

The third line follows by taking discrete logarithms of both sides with respect to g. Solving congruence **??** for $x \in \mathbf{Z}/p$ is precisely equivalent to solving congruence **??** for $y \in \mathbf{Z}/(p-1)$, since y uniquely determined x and vice versa.

Therefore it suffices to show that the congruence $ey \equiv d \pmod{p-1}$ has exactly $\gcd(e, p-1)$ solutions, assuming that it has any at all.

We have seen in problems 4 and 10 of problem set 2 that a linear congruence of this form:

1. has a solution if and only if $\gcd(e, p-1)$ divides d, and
2. if so, the solution may be expressed in the form $y \equiv \ldots \pmod{\frac{p-1}{\gcd(e,p-1)}}$ (where the $\ldots$ is a expression that is calculated using the Exteneded Euclidean algorithm).

We are assuming that a solution exists, so it follows that the set of all solutions is described by a congruence class modulo $\frac{p-1}{\gcd(e,p-1)}$. So if y_0 is one possible value of y, then the set of all values is $\{y_0 + k \cdot \frac{p-1}{\gcd(e,p-1)}\}$. This gives $\gcd(e, p-1)$ distinct solutions modulo $p-1$, since $k = 0, 1, 2, \cdots, \gcd(e, p-1)$ all give values of y that are distinct modulo $p-1$, while any other value of k gives a value of y differing by a multiple of $p-1$ from one of these.

Therefore there are $\gcd(e, p-1)$ possible choices of y (modulo $p-1$) solving congruence **??**, hence also $\gcd(e, p-1)$ possible choices of x (modulo p) solving congruence **??**.

(b) We saw in the solution to (a) that the congruence **??** has a solution if and only if $\gcd(e, p-1)$ divides $d = \log_g(c)$. Therefore d must have the form $k \cdot \gcd(e, p-1)$ for some integer k. Two valued of k give values of d that are congruent modulo $p-1$ if and only if they differ by a multiple of $\frac{p-1}{\gcd(e,p-1)}$. Therefore there are $\frac{p-1}{\gcd(e,p-1)}$ possible values of d for which congruence **??** has a solution. In turn, there are also $\frac{p-1}{\gcd(e,p-1)}$ distinct (modulo p) nonzero values of c for which the original congruence has a solution.

Alternate solution. Consider the function $(\mathbf{Z}/p)^\times \to (\mathbf{Z}/p)^\times$ given by $x \to x^e$. Part (a) implies that, for any element in the image of this map, the inverse image of that element has exactly $\gcd(e, p-1)$ elements in it. Since the source of the map has exactly $(p-1)$ elements in it, this means that the map partitions these $(p-1)$ elements into classes of size $\gcd(e, p-1)$ elements each, according to their image under the map. Therefore there are $\frac{p-1}{\gcd(e,p-1)}$ such classes (since the number of classes times the number of elements per class must be $p-1$), hence this is equal to the number of points in the image, i.e. the number of elements c for which an eth root exists.

3. Textbook exercise 3.7 (3.6 in first edition).

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

4. Textbook exercise 3.8 (3.7 in first edition).

   Solution.

   By trial and error, Eve may discover that $N = 12191 = 73 \cdot 167$. Therefore $\phi(N) = 72 \cdot 166 = 11952$, and the decrypting exponent is $e^{-1} \pmod{\phi(N)}$, i.e. $37^{-1} \equiv 11629 \pmod{11952}$. Therefore the plaintext that Alice sent to Bob was $c^d \pmod N$, i.e. $587^{11629} \pmod{12191}$, which works out to 4894.

5. Textbook exercise 3.13 (3.12 in first edition).

   Solution.

   Eve may compute, using the extended Euclidean algorithm, that

$$e_1 u + e_2 v = 1,$$

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

$$\begin{aligned} m &\equiv m^1 \pmod N \\ &\equiv m^{e_1 u + e_2 v} \pmod N \\ &\equiv (m^{e1})^u (m^{e_2})^v \pmod N \\ &\equiv c_1^u c_2^v \pmod N \\ &\equiv 1244183534^{252426389} \cdot 732959706^{-496549570} \pmod{1889570071} \end{aligned}$$

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).

$$\begin{aligned} m &\equiv 1244183534^{252426389} \cdot 1873807620^{496549570} \pmod{1889570071} \\ &\equiv 1031756109 \cdot 603385073 \pmod{1889570071} \\ &\equiv 1054592380 \pmod{1889570071} \end{aligned}$$

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 $m^{e_1} \equiv c_1 \pmod N$ and $m^{e_2} \equiv c_2 \pmod N$.

6. The textbook mentions that choosing $e = 3$ as the encryption exponent can help increase the efficiency of RSA encryption, likely without compromising security. If Bob chooses $e = 3$ in his public key, what conditions should he be sure to satisfy when choosing his modulus in order to have a valid public key?

   Solution.

   Bob must ensure that $\gcd(3, (p-1)(q-1)) = 1$. Since 3 is prime, the only possible values of this gcd are 1 and 3, so this requirement amounts to saying that 3 does not divide $(p-1)(q-1)$. Equivalently, 3 must divide neither $p-1$ nor $q-1$. This is conveniently stated in terms of congruences as follows.

$$\begin{aligned} p &\not\equiv 1 \pmod 3 \\ q &\not\equiv 1 \pmod 3 \end{aligned}$$

   Since Bob is (hopefully) not going to choose p or q to be 3 itself, this amounts to saying that p and q should both be $2 \pmod 3$.

7. Textbook exercise 3.11(a) (3.10(a) in first edition). 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 the first programming problem.

Solution.

Observe that by Fermat's little Theorem,

$$\begin{aligned} g_1 &\equiv (g^{p-1})^{r_1} \pmod p \\ &\equiv 1^{r_1} \pmod p \\ &\equiv 1 \pmod p \\ \text{and similarly, } g_2 &\equiv 1 \pmod q \end{aligned}$$

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 $\mathbf{Z}/N$, this means that indeed Alice has constructed the original message m.

Programming problems

8. Write a program to break the cryptosystem described in problem 3.11 (3.10 in first edition). 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 $p | (g_1 - 1)$. 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 `merge` and `gcd` functions, since have already been written for previous problem sets.

```
def analyze(N,g1,g2,c1,c2):
        if g1%N == 1: return c1 # Easy special case
        p = ext_euclid(g1-1,N)[0] # GCD must equal p
        q = N/p
        return merge(c1,p,c2,q)[0]

# Extended Euclidean algorithm
```

```
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 (from PSet 4)
def merge(a1,m1,a2,m2):
        g,u,v = ext_euclid(m1,m2)
        return ( (v*m2*a1 + u*m1*a2)%(m1*m2), m1*m2 )

# I/O
N,g1,g2 = map(int,raw_input().split())
c1,c2 = map(int,raw_input().split())
```

9. Write a program which determines the two prime factors p, q of an RSA modulus $N = pq$, given N and $\phi(N)$. This demonstrates that computing phi is not any easier than factoring.

Solution.

We know that

$$\begin{aligned} N &= pq \\ \phi &= (p-1)(q-1) \\ &= N - p - q + 1 \end{aligned}$$

Hence we know the sum and the product of p and q.

$$\begin{aligned} p + q &= N - \phi + 1 \\ pq &= N \end{aligned}$$

Now, we can recover p and q by solving a quadratic equation. Specifically, p, q are the two solutions of the equation

$$\begin{aligned} 0 &= (x-p)(x-q) \\ &= x^2 - (p+q)x + pq \\ &= x^2 - (N - \phi + 1)x + N \end{aligned}$$

Since we know $p+q$ and pq, we know the coefficients of this quadratic equation, hence we can solve it (e.g. with the quadratic formula) and recover p and q as the two roots. The following code implements this.

```
import math

def extract(N,phi):
        sum = N - phi + 1
        diff = int(math.sqrt(sum*sum-4*N))
        return (sum-diff)/2,(sum+diff)/2

# I/O
N,phi = map(int,raw_input().split())
p,q = extract(N,phi)
print p,q
```

10. Implement the Pohlig-Hellman algorithm. You have written all of the main ingredients in previous programming problems. 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 all of the function from problems 6 and 7 of the previous problem set (babystep-giant for known-order elements, and merging lists of congruences). To save space, I will omit this course code; see last week's solutions for the source. What is needed from those solutions are the functions `merge_list` and `dip_bsgs`, together with the auxiliary functions needed in these.

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.

```
### Omitted: source for the functions merge_list(a,m) and dlp_bsgs(p,g,a,N).
### See last week's solutions for the code.

# Gives a list of the prime-power factors of n.
# Similar in strategy to problem 6 of problem set 1.
def pp_factors(n):
        res = []
        d = 2
        while n>1:
                dpow = 1
                while n%d == 0:
                        dpow *= d
                        n /= d
                if dpow != 1: res += [dpow]
                d += 1
        return res

def pohlig_hellman(p,g,a):
        # Make list of q^e values
        qe = pp_factors(p-1)
        # Make list of gi values and hi values (l for "list")
        # Determine and solve each subordinate DLP; store results in a list y.
        y = []
        for i in range(len(qe)):
                gi = pow(g,(p-1)/qe[i],p)
                hi = pow(a,(p-1)/qe[i],p)
                yi = dlp_bsgs(p,gi,hi,qe[i])
                y += [yi]
        x,m = merge_list(y,qe)
        return x

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

11. Write a program to print the last five digits of F_n, the n^{th} Fibonacci number. These are defined by $F_0 = 0,\ F_1 = 1$, and the recurrence $F_n = F_{n-1} + F_{n-2}$ for $n > 1$.

    Hint. Express the *vector* $\binom{F_n}{F_{n+1}}$ in terms of the vector $\binom{F_{n-1}}{F_n}$ using a matrix equation. Using this equation, try to reformulate the problem in terms of one of the programming problems from last week.

    Solution.

    The Fibonacci recurrence may be expressed in terms of pairs of consecutive Fibonacci numbers using the following equation.

$$\begin{pmatrix} F_n \\ F_{n+1} \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} F_{n-1} \\ F_n \end{pmatrix}$$

    On the surface, this equation does not seem to help: we still will need to do approximately 2^{64} matrix multiplications to get to F_n from F_0 and F_1. However, the key observation is that the net effect of all these multiplications is to take a single large power of the matrix. Explicitly,

$$\begin{pmatrix} F_n \\ F_{n+1} \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^n \begin{pmatrix} F_0 \\ F_1 \end{pmatrix}$$

Since $F_0 = 0$ and F_1, it follows that the vector $\binom{F_n}{F_{n+1}}$ is simply the second column of the matrix $\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^n$, and in turn the number F_n is the first entry of this column. So if we can compute this matrix, we can extract F_n as the upper-left entry.

Since we ultimately only need the remainder of F_n modulo $100,000$, we can reduce all matrices modulo $100,000$ as well. Thus we can use the fast-powering algorithm for matrices modulo m developed last week to compute the matrix modulo $100,000$, and then return the upper-right entry of it. An implementation is below.

Note that it is convenient to use "formatted printing" (as noted in the online problem statement) to automatically pad the resulting number with 0s in case it has fewer than 5 digits initially.

```
### Omitted: source for the function fast_pow(A,e,m)
###      for powers of 2x2 matrices modulo m.
### See last week's solutions for the code.

def fn_mod(n,m):
        A = [[0,1],[1,1]]
        An = fast_pow(A,n,m)
        return An[0][1]

# I/O
n = int(raw_input())
print '%05d'%fn_mod(n,100000)
```