Written problems

1. Textbook exercise 3.12.

Solution.

(a) Eve must pose as Alice to Bob and Bob to Alice at each stage of the ElGamal cryptosystem. Here is how this might look.

- Suppose that the parameters p, g (p a prime, g a unit of $\mathbf{Z}/p$) have been chosen, and are known to all parties.
- Alice selects her secret number a and computes $A \equiv g^a \pmod{p}$. She attempts to publish the public key A, but Eve intercepts it and prevents it from reaching Bob.
- Eve creates her own secret number e, computes $E \equiv g^e \pmod{p}$, and sends E to Bob, pretending to be Alice sending her public key.
- Bob chooses a message m. He believes that Alice's public key is E, so he chooses a random number b and computes a ciphertext (c_1, c_2), where

$$\begin{aligned} c_1 &\equiv g^b \pmod{p}, \\ c_2 &\equiv E^b m \pmod{p}. \end{aligned}$$

- Eve intercepts (c_1, c_2) before it reaches Alice. She uses her secret number e to decrypt m as $c_2 c_1^{-e} \pmod{p}$. She then re-encrypts m using Alice's real public key, by choosing her own ephemeral key f and computing (c_1', c_2'), where

$$\begin{aligned} c_1' &\equiv g^f \pmod{p}, \\ c_2' &\equiv A^f m \pmod{p}. \end{aligned}$$

  She sends this ciphertext to Alice, pretending to be Bob.
- Alice decrypts (c_1', c_2') as usual to obtain m.

At the end of this process, Alice and Bob are none the wiser: Alice has received Bob's message m is exactly the manner that she expected. However, Eve has successfully read the message en route.

(b) As before, Eve must insert herself between Alice and Bob and imitate each one to the other.

- Alice selects a public key (N, e) (where she knows the prime factors of N) and a private decrypting exponent d. She attempts to publish (N, e), but Eve intercepts it and prevents it from reaching Bob.
- Eve chooses her own public key (N', e') and private decrypting exponent d'. She sends (N', e') to Bob, pretending to be Alice.
- Bob chooses a message m, which he wishes to send to Alice. He believes that (N', e') is Alice's public key, so he computes

$$c \equiv m^{e'} \pmod{N'}$$

  and sends it to "Alice," who is actually Eve.

- Eve computes m as $c^{d'} \pmod{N'}$. She then re-encrypts it as $c' \equiv m^e \pmod{N}$ using Alice's real public key, and sends it to Alice (pretending to be Bob).
- Alice receives c' and computes m as $(c')^d \pmod{N}$.

Now Alice has received Bob's message m, and to all appearances nothing is amiss. However, Eve has also learned the contents of m in the process.

Practical note. A student pointed out that these attacks do not sound realistic, since public key cryptography allows public keys to be widely distributed and confirmed by multiple sources, making it seemingly impossible for Eve to sabotage the initial key exchange. Indeed, this is one of the many strengths of public key cryptography – man-in-the-middle attacks are much harder to execute than they would be in symmetric cryptography. A successful attack would likely involve something more invasive that the outline above: Eve might, for example, hack into the directory that Bob uses to store public keys, and overwrite Alice's public key in that directory, prior to inserting herself in the middle of subsequent transmissions from Alice to Bob. Depending on where and precisely how Bob stores the public keys, and how often he checks it for tampering, this attack may be more or less plausible.

2. Prove that if $N = pq$, where p, q are distinct *odd* prime numbers, then the congruence $x^2 \equiv 1 \pmod{N}$ has exactly *four* distinct solutions (modulo N).

   Hint. Use the Chinese remainder theorem.

   Solution.

   First observe that $x^2 \equiv 1 \pmod{pq}$ if and only if $x^2 \equiv 1 \pmod{p}$ and $x^2 \equiv 1 \pmod{q}$ (this amounts to saying that pq divides $x^2 - 1$ if and only if both p and q divide $x^2 - 1$). From our discussion of the Miller-Rabin test, we know that $x^2 \equiv 1 \pmod{p}$ if and only if $x \equiv \pm 1 \pmod{p}$, and similarly for q.

   Conversely, if $x \equiv \pm 1 \pmod{p}$ and also $\pmod{q}$, then $x^2 \equiv 1 \pmod{p}$ and also $\pmod{q}$, and consequently $x^2 \equiv 1 \pmod{pq}$.

   Now, $1 \not\equiv -1 \pmod{p}$ (that is why we must assume that $p, q \neq 2$!), so there are two distinct choices for the class of $x \pmod{p}$ and two distinct choices for the class of $x \pmod{q}$. Now, suppose that ϵ_1, ϵ_2 are two elements of $\{-1, 1\}$. Then the Chinese Remainder Theorem guarantees that the pair of congruences

   $$\begin{aligned} x &\equiv \epsilon_1 \pmod{p} \\ x &\equiv \epsilon_2 \pmod{q} \end{aligned}$$

   has a unique solution modulo pq; the resulting x satisfies $x^2 \equiv 1 \pmod{pq}$. There are four different ways to choose ϵ_1, ϵ_2 (two choices for each), resulting in four choices of x. These choices are distinct modulo pq since any two of them differ either modulo p or modulo q. We have seen above that *any* solution to $x^2 \equiv 1 \pmod{pq}$ must solve the above pair of congruences for one of these four choices. Hence there are exactly four such choices of x.

3. For each number n between $100,000$ and $100,019$ (inclusive), determine how many numbers $a \in \{1, 2, \cdots, n-1\}$ are Miller-Rabin witnesses (you should write some code to do this; briefly summarize how your code works and simply copy out the resulting numbers). Which of these

numbers are prime? Which number has the lowest proportion of witnesses, and what it this portion?

Solution.

We can use the following function definition (as written on the board in class) to determine whether a particular integer is a witness or not.

```
def is_witness(a,n):
        q = n-1
        k = 0
        while q%2 == 0:
                q /= 2
                k += 1
        b = pow(a,q,n)
        if b == 1: return False
        for i in xrange(k):
                if b == n-1: return False
                b = b*b % n
        return True
```

In addition, we can write the following function to count the number of witnesses for a particular integer n.

```
def num_witnesses(n):
        num = 0
        for a in range(1,n):
                if is_witness(a,n): num += 1
        return num
```

Having defined these two functions, we can obtain the desired figures quickly in the Python terminal as shown.

```
>>> for n in range(100000,100020):
...     wits = num_witnesses(n)
...     print n, "has", wits, "witnesses."
...
100000 has 99998 witnesses.
100001 has 99950 witnesses.
100002 has 100000 witnesses.
100003 has 0 witnesses.
100004 has 100002 witnesses.
100005 has 100002 witnesses.
100006 has 99990 witnesses.
100007 has 100004 witnesses.
100008 has 100006 witnesses.
100009 has 99954 witnesses.
```

```
100010 has 100008 witnesses.
100011 has 100008 witnesses.
100012 has 100010 witnesses.
100013 has 100010 witnesses.
100014 has 100012 witnesses.
100015 has 100008 witnesses.
100016 has 100014 witnesses.
100017 has 100014 witnesses.
100018 has 100014 witnesses.
100019 has 0 witnesses.
>>>
```

We can immediately identify the two primes: they are the ones with no witnesses, 100003 and 100019.

When I wrote this question, I meant to ask which *composite* number has the lowest proportion of witnesses, i.e. the smallest probability of a "false negative" when testing whether it is composite. Since I instead asked which numbers (composite or prime) have the lowest proportion, the answer is that the two prime numbers 100003 and 100019 have the lowest proportion: 0.

The answer to the question I meant to ask is: among the 18 composites on the list, $100,009$ has the lowest portion of witnesses, at about 99.946%. So we can observe in this data that the 75% guarantee in the theorem in the textbook is quite conservative as an estimate on the prevalence of witnesses.

4. Divide the integers from 1 to $1,000,000$ into ten equal intervals. For each interval, determine how many primes are in that interval. Also determine how many of these primes are congruent to 1 (mod 4) and how many are congruent to 3 (mod 4). Which of these tends to be a larger number?

   Note. You can use the Miller-Rabin test for this problem, but you are free to use any other method you prefer. For example, the Sieve of Eratosthenes may be faster for this particular problem.

   Solution.

   We can begin by creating a list `prime` of length $1,000,001$ (so that it accepts indices up to and including 1 million) telling whether or not each number is prime. One way to do with is using the Miller-Rabin test (as implemented in problem 11). For variety, here's another: the following code implements an ancient algorithm to list primes called the Sieve of Eratosthenes. I'll omit the details of how it works, but you can read about it online if you wish.

```
def erat(N):
        prime = [True]*(N+1)
        prime[0] = prime[1] = False
        for n in range(2,N+1):
                if prime[n]:
                        k = n
```

```
                        while k*n <= N:
                                prime[k*n] = False
                                k += 1
        return prime
```

Now we can obtain the desired counts as follows in the Python terminal.

```
>>> for k in range(10):
...     rem1 = 0
...     rem3 = 0
...     tot = 0
...     for n in range(100000*k+1,100000*(k+1)+1):
...             if prime[n]:
...                     if n%4 == 1: rem1 += 1
...                     if n%4 == 3: rem3 += 1
...                     tot += 1
...     print "Block",k,"has",tot,"primes, including",rem1,"that are 1mod4 and",
        rem3,"that are 3mod4."
...
Block 0 has 9592 primes, including 4783 that are 1mod4 and 4808 that are 3mod4.
Block 1 has 8392 primes, including 4194 that are 1mod4 and 4198 that are 3mod4.
Block 2 has 8013 primes, including 4003 that are 1mod4 and 4010 that are 3mod4.
Block 3 has 7863 primes, including 3920 that are 1mod4 and 3943 that are 3mod4.
Block 4 has 7678 primes, including 3831 that are 1mod4 and 3847 that are 3mod4.
Block 5 has 7560 primes, including 3791 that are 1mod4 and 3769 that are 3mod4.
Block 6 has 7445 primes, including 3726 that are 1mod4 and 3719 that are 3mod4.
Block 7 has 7408 primes, including 3667 that are 1mod4 and 3741 that are 3mod4.
Block 8 has 7323 primes, including 3669 that are 1mod4 and 3654 that are 3mod4.
Block 9 has 7224 primes, including 3591 that are 1mod4 and 3633 that are 3mod4.
>>>
```

Note that these figures agree appear to be consistent with the predictions of the Prime Number theorem: $\frac{1}{\ln(100{,}000)} \approx 0.08685$ and $\frac{1}{\ln(1{,}000{,}000)} \approx 0.07238$, which predicts a steady drop from about 8700 primes per $100,000$ to about 7200 primes per $100,000$ over the course of this range, which is reasonably close to the observed data.

As for the relative numbers of $1 \pmod 4$ primes and $3 \pmod 4$, two things are apparent:

(a) The figures are very close to each other in all ten cases.

(b) In seven out of ten cases (all but blocks $5, 6$ and 8), the $3 \pmod 4$ primes have a narrow lead.

The first point is consistent with the known fact that asymptotically, these two counts should be identical (primes distribute evently among all invertible residue classes). The second point is not too remarkable on its own, since our sample size is small.

The question of whether, for a given n, there are more primes congruent to $1 \pmod 4$ or $3 \pmod 4$, is related to some surprisingly deep problems in number theory; surprisingly, 3

(mod 4) is in the lead for "most" values of n (in some sense), even though that lead must be extremely narrow. A nice survey of some related mathematics can be found in the paper "Prime Number Races" by Granville and Martin (you can find it free online). Briefly: the reason you might expect 3 (mod 4) to win is that all perfect squares are 1 (mod 4), and no perfect squares are prime, so a random 3 (mod 4) number is very slightly less likely to be a square than usual, and thus slightly more likely to be prime.

5. Textbook exercise 3.19.

Solution.

(a) Denoting the number of primes up to n as $\pi(n)$ as usual, we can express $P(N)$ as follows.

$$\begin{aligned} P(N) &= \frac{\pi(\frac{3}{2}N) - \pi(\frac{1}{2}N)}{\frac{3}{2}N - \frac{1}{2}N} \\ &= \frac{\pi(\frac{3}{2}N) - \pi(\frac{1}{2}N)}{N} \end{aligned}$$

(Actually, this formula is only correct on the nose if N is odd, so that neither $\frac{3}{2}N$ nor $\frac{1}{2}N$ are integers. If N is even, then the numerator would be $\pi(\frac{1}{2}N - 1)$ and the denominator should be increasing by 1 since both endpoints of the interval should be included. Since these differences change the numerator and denominator by at most 1 each, and both tend to infinity, this quibble doesn't affect asymptotics, so I will permit myself to assume the formula is correct as written and not fuss over it.)

Hence we wish to evaluate the limit of the following expression.

$$\frac{P(N)}{1/\ln(N)} = \frac{\pi(\frac{3}{2}N)}{N/\ln(N)} - \frac{\pi(\frac{1}{2}N)}{N/\ln(N)} \tag{1}$$

From the Prime Number Theorem, we know the following two limits.

$$\begin{aligned} \lim_{N\to\infty} \frac{\pi(\frac{3}{2}N)}{\frac{3}{2}N/\ln(\frac{3}{2}N)} &= 1 \\ \lim_{N\to\infty} \frac{\pi(\frac{1}{2}N)}{\frac{1}{2}N/\ln(\frac{1}{2}N)} &= 1 \end{aligned}$$

In order to compare these two limits to the limits of the expressions appearing in equation (1), we must compare $\ln(\frac{3}{2})$ and $\ln(\frac{1}{2}N)$ to $\ln N$. This can be done by observing that for *any* constant c (e.g. $\frac{1}{2}$ or $\frac{3}{2}$),

$$\begin{aligned} \lim_{N\to\infty} \frac{\ln N}{\ln(cN)} &= \lim_{N\to\infty} \frac{\ln N}{\ln c + \ln N} \\ &= 1 \end{aligned}$$

We can use this fact to evaluate the needed limits as follows. Here c is any constant; we will need to use $c = \frac{1}{2}$ and $c = \frac{3}{2}$.

$$\begin{aligned}
\lim_{N\to\infty} \frac{\pi(cN)}{N/\ln(N)} &= \lim_{N\to\infty} \frac{\pi(cN)}{N/\ln(N)} \cdot \lim_{N\to\infty} \frac{\ln(cN)}{\ln(N)} \\
&= \lim_{N\to\infty} \left[\frac{\pi(cN)}{N/\ln(N)} \cdot \frac{\ln(cN)}{\ln(N)} \right] \\
&= \lim_{N\to\infty} \frac{\pi(cN)}{N/\ln(cN)} \\
&= c \cdot \lim_{N\to\infty} \frac{\pi(cN)}{cN/\ln(cN)} \\
&= c
\end{aligned}$$

Hence in our case, this gives

$$\lim_{N\to\infty} \left[\frac{\pi(\frac{3}{2}N)}{N/\ln(N)} - \frac{\pi(\frac{1}{2}N)}{N/\ln(N)} \right] = \frac{3}{2} - \frac{1}{2}$$

which gives the desired result: $\lim_{N\to\infty} \dfrac{P(N)}{1/\ln(N)} = 1$.

(b) The denominator should actually be the same as in part (a): it is $1/\ln(N)$. For any choice of c_1, c_2, the limit of this probability is approximately the "probability that the number N is prime," i.e. 1 out of $\ln N$. To prove this statement (which you were not required to do for this problem), you can follow the exact same logic as in part (a), showing that

$$\frac{P(c_1, c_2; N)}{1/\ln N} = \frac{1}{c_2 - c_1} \left[\frac{\pi(c_2 N)}{N/\ln N} - \frac{\pi(c_1 N)}{N/\ln N} \right]$$

and the limit of this expression as $N \to \infty$ is $\frac{1}{c_2 - c_1} [c_2 - c_1] = 1$.

6. Textbook exercise 3.20, parts (a), (b) and (c).

Solution. Note that in case, you need only *formulate* the statement; you do not need to prove it. In all three cases, the numbers $\frac{1}{2}$ and $\frac{3}{2}$ can be replaced with any two positive constants with $c_1 < c_2$, with no other changes to the statement.

(a) Let $P_{\text{odd}}(N)$ denote the probability that an odd integer, chosen uniformly at random from all of the odd numbers in $[\frac{1}{2}N, \frac{3}{2}N]$, is prime. Then $\lim_{N\to\infty} \dfrac{P_{\text{odd}}(N)}{2/\ln N} = 1$.

(b) Let $P_{1 \bmod 3}(N)$ denote the probability that an integer, chosen uniformly at random among all integers n congruent to 1 (mod 3) in the interval $[\frac{1}{2}N, \frac{3}{2}N]$, is prime. Then $\lim_{N\to\infty} \dfrac{P_{1 \bmod 3}(N)}{3/(2\ln N)} = 1$.

(c) Let $P_{1 \bmod 6}(N)$ denote the probability that an integer, chosen uniformly at random among all integers n congruent to 1 (mod 6) in the interval $[\frac{1}{2}N, \frac{3}{2}N]$, is prime. Then $\lim_{N\to\infty} \dfrac{P_{1 \bmod 6}(N)}{3/\ln N} = 1$.

7. Textbook exercise 3.21.

Solution.

(a) Integrating the dt term by parts, we obtain:

$$\begin{aligned}\int_2^X \frac{dt}{\ln t} &= \left[\frac{t}{\ln t}\right]_2^X - \int_2^X t \cdot \left(-\frac{1}{(\ln t)^2}\frac{1}{t}\right) dt \\ &= \frac{X}{\ln X} - \frac{2}{\ln 2} + \int_2^X \frac{1}{(\ln t)^2} dt\end{aligned}$$

Since $\frac{2}{\ln 2}$ is constant, it is certainly bounded by a constant, so it is $\mathcal{O}(1)$.

(b) We can break the integral into two pieces, one from 2 to $\sqrt{X}$ and the other from $\sqrt{X}$ to X, and bound each one using the fact that

$$\int_a^b f(t)dt \le (b-a) \cdot (\text{maximum value of } f(t) \text{ on } [a,b]),$$

which can be visualized by placing the area under the curve underneath a rectangle. Since $\frac{1}{(\ln t)^2}$ is a decreasing function, the maximum value of each piece is easy: it is just $\frac{1}{(\ln t)^2}$ evaluated at the left endpoint. Using the suggested endpoints gives the following bounds.

$$\begin{aligned}\int_2^{\sqrt{X}} \frac{1}{(\ln t)^2} dt &\le \frac{1}{(\ln 2)^2}(\sqrt{X} - 2) \\ &\le \frac{\sqrt{X}}{(\ln 2)^2} \\ \int_{\sqrt{X}}^X \frac{1}{(\ln t)^2} dt &\le \frac{1}{\left(\ln \sqrt{X}\right)^2}(X - \sqrt{X}) \\ &\le \frac{X}{\left(\frac{1}{2}\ln X\right)^2} \\ &\le \frac{4X}{(\ln X)^2} \\ \Rightarrow \int_2^X \frac{1}{(\ln t)^2} dt &\le \frac{\sqrt{X}}{(\ln 2)^2} + \frac{4X}{(\ln X)^2}\end{aligned}$$

From this it follows that

$$\begin{aligned}
\frac{\int_2^X \frac{1}{(\ln t)^2}dt}{X/\ln X} &\leq \frac{\sqrt{X}}{(\ln 2)^2}\frac{\ln X}{X} + \frac{4X}{(\ln X)^2}\frac{\ln X}{X} \\
&\leq \frac{\ln X}{(\ln 2)^2\sqrt{X}} + \frac{4}{\ln X} \\
\Rightarrow \lim_{X\to\infty} \frac{\int_2^X \frac{1}{(\ln t)^2}dt}{X/\ln X} &\leq \lim_{X\to\infty}\left(\frac{\ln X}{(\ln 2)^2\sqrt{X}} + \frac{4}{\ln X}\right).
\end{aligned}$$

(assuming that both limits exist, which will be a consequene of what follows).

Now, observe that $\lim_{X\to\infty} \frac{\ln X}{\sqrt{X}} = 0$ (this follows, for example, from applying l'Hôpital's rule), and $\lim_{X\to\infty} \frac{1}{\ln x} = 0$ since the denominator tends to infinity. So it follows that (since it is a positive function of X bounded above by a function tending to 0),

$$\lim_{X\to\infty} \frac{\int_2^X \frac{1}{(\ln t)^2}dt}{X/\ln X} = 0.$$

From this it follows, using part (a), that

$$\begin{aligned}
\lim_{X\to\infty} \frac{\mathrm{Li}(X)}{X/\ln X} &= \lim_{X\to\infty}\left(\frac{X/\ln X}{X/\ln X} - \frac{2/\ln 2}{X/\ln X} + \frac{\int_2^X \frac{dt}{(\ln t)^2}}{X/\ln X}\right) \\
&= \lim_{X\to\infty}(1) - \frac{2}{\ln 2}\lim_{X\to\infty}\left(\frac{\ln X}{X}\right) + \lim_{X\to\infty}\left(\frac{\int_2^X \frac{dt}{(\ln t)^2}}{X/\ln X}\right) \\
&= 1 - 0 + 0 \\
&= 1.
\end{aligned}$$

(c) According to formula 3.12, if the Riemann Hypothesis is true, then

$$\pi(X) = \int_2^X \frac{dt}{\ln t} + \mathcal{O}(\sqrt{X}\ln X).$$

If this is true, then it follows that

$$\begin{aligned}
\frac{\pi(X)}{X/\ln X} &= \frac{\int_2^X \frac{dt}{\ln t}}{X/\ln X} + \mathcal{O}\left(\frac{\sqrt{X}\ln X}{X/\ln X}\right) \\
&= \frac{\int_2^X \frac{dt}{\ln t}}{X/\ln X} + \mathcal{O}\left(\frac{(\ln X)^2}{\sqrt{X}}\right).
\end{aligned}$$

This means that for sufficiently large X, there exists a constant C such that

$$\frac{\int_2^X \frac{dt}{\ln t}}{X/\ln X} - C \cdot \frac{(\ln X)^2}{\sqrt{X}} \leq \frac{\pi(X)}{X/\ln X} \leq \frac{\int_2^X \frac{dt}{\ln t}}{X/\ln X} + C \cdot \frac{(\ln X)^2}{\sqrt{X}}$$

Now $\lim_{X\to\infty} \frac{(\ln X)^2}{\sqrt{X}} = 0$ (e.g. by taking the square root and applying l'Hôpital's rule once). Therefore the limit of the left side and the limit of the right side both equal $\lim_{X\to\infty} \frac{\int_2^X \frac{dt}{\ln t}}{X/\ln X}$, which is equal to 1 by part (b). From the squeeze theorem, it follows that $\lim_{X\to\infty} \frac{\pi(X)}{X/\ln X} = 1$, which is the prime number theorem.

8. Textbook exercise 4.1.

   Solution.

   (a) The modulus is $N = 541 \cdot 1223 = 661643$. The private signing key is the inverse of $e = 159853$ modulo $\phi(N) = (541-1)(661643-1) = 659880$, which is $d = 561517$.

   (b) The signature for $D = 630579$ is $D^d \pmod N$, i.e. $630579^{561517} \equiv 206484 \pmod{661643}$. So the signature is $S = 206484$.

9. Textbook exercise 4.2.

   Solution. We can check each signature as follows.

$$\begin{aligned} S^e &\equiv 876453^{87953} \equiv 772481 \pmod{1562501} \\ (S')^e &\equiv 870099^{87953} \equiv 161153 \pmod{1562501} \\ (S'')^e &\equiv 602754^{87953} \equiv 586036 \pmod{1562501} \end{aligned}$$

   The first signature is not valid, since S^e is not congruent to D, which is 119812. The second and third are correct signatures, since the result of the exponentiation matches the document.

   Programming problems

10. Write a program to decipher an RSA message sent to you using your public key. You will be given your public key, as well as the two prime numbers that you used to create it, and a cipher text c.

    Solution.

    To decrypt, you must first finding an inverse d of e modulo $\phi(N) = (p-1)(q-1)$, and then compute $c^d \pmod N$. This is done in the following code.

    Remark. As the textbook points out, you might obtain a mild performance improvement by first calculating $L = \frac{(p-1)(q-1)}{\gcd(p-1,q-1)}$ (this is the least common multiple of $p-1$ and $q-1$) and letting d be $e^{-1} \pmod L$ rather than $e^{-1} \pmod{(p-1)(q-1)}$. This change does not affect the correctness of the decryption.

```
# Inverses modulo m.
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

# Finds the deciphering exponent d, using p,q and the enciphering exponent.
def private_exp(p,q,e):
        phi = (p-1)*(q-1)
        return inv_mod(e,phi)

def decipher(c,p,q,e):
        d = private_exp(p,q,e)
        return pow(c,d,p*q)

# I/O
N,e = map(int,raw_input().split())
p,q = map(int,raw_input().split())
c = int(raw_input())

print decipher(c,p,q,e)
```

11. Write a program that determines whether a given integer (up to 1024 bit in size) is prime or not.

    We first need the function `is_witness(a,n)` written out in class. To check primality, we can just try 50 possible witnesses, and return false if we find a single witness. In practice, many fewer than 50 would work just as well (5 would be enough, even 3 would probably be fine).

```
import random
random.seed()

def is_witness(a,n):
        q = n-1
        k = 0
        while q%2 == 0:
                q /= 2
                k += 1
        b = pow(a,q,n)
        if b == 1: return False
```

```
        for i in xrange(k):
                if b == n-1: return False
                b = b*b % n
        return True

def is_prime(n):
        for i in xrange(50): # 50 is many more than really necessary
                a = random.randrange(1,n)
                if is_witness(a,n): return False
        return True

# I/O
n = int(raw_input())
if is_prime(n): print 'prime'
else: print 'composite'
```

12. Write a program that generates three numbers p, q, g with the following properties.

    - p and q are primes of specified length in bits.
    - $p \equiv 1 \pmod{q}$
    - $g \in \mathbf{Z}/p$ has order q modulo p.

    Your program will receive the desired lengths (number of bits) for p and q, and should print p, q and g.

    Hint. See the solution to problems 5 and 6 on problem set 3; these should help you see how to construct the number g once you have chosen p and q.

    Solution.

    We can proceed in three stages.

    1) Choose integers q at random from $[2^{\texttt{qbits}-1}, 2^{\texttt{qbits}})$ until we find one that is prime.
    2) Choose integers k at random from $[\frac{2^{\texttt{pbits}-1}-1}{q}, \frac{2^{\texttt{pbits}}-1}{q})$ until we find one such that $p = kq + 1$ is prime. Use this value as p.
    3) Choose integers a at random from $(0, p)$ until we find one such that $a^{(p-1)/q} \not\equiv 1 \pmod{p}$. Then let $g \equiv a^{(p-1)/q} \pmod{p}$; it is guaranteed to have order q.

    The first stage is self-explanatory; it merely requires the use of the code from the previous problem. The second step has one tricky aspect to it: the endpoints of this interval are not integers, so which performing integer division there is a risk that you will round the wrong direction and end up with inadmissable value of k in being chosen. There are a few ways to fix this. Perhaps the easiest to code is the following observation: if you want to compute the *ceiling* of a quotient a/b (where a, b are integers), i.e. the fraction rounded *up* to the nearest integer, one way to achieve this in code is to write `(a+b-1)/b`. Adding $b-1$ to the numerator causes it to get rounded up to the next multiple of b, except in the case where a is already a multiple of b, in which case the result will be rounded down to example a/b. Other ways to

deal with the rounding issue include computing the fractions as floating-point numbers and then rounding them up or down, or simply rounding down as usual and then adding 1 to the lower bound.

The first two steps are implemented as the functions `make_q(qbits)` and `make_p(pbits,q)` below.

The third step also bears some explanation. We saw on homework 3 that *if* a is a primitive root, then $a^{(p-1)/q}$ is an element of order q. This suggests one way to find an element of order q: first find a primitive root a, then exponentiate it. This is undesirable, however, since to find a primitive root usually requires factoring $p-1$, which it would be best not to have to do.

The key observation is that we actually don't *need* a to be a primitive root. All we need is that $g \equiv a^{(p-1)/q} \not\equiv 1 \pmod{p}$. If this is so, then certainly $g^q \equiv 1 \pmod{p}$, so g will have order q since its order is not 1. So we can just choose values of a at random. We know that we're likely to succeed (since there's an ample supply of primitive roots, and any one of them will do), but we can tell whether we've succeeded without knowing whether we actually chose a primitive root or not (possibly we didn't, but we don't care since we still got a working value of g).

The third step is implemented as `make_g(p,q)` below. Finally, we assemble these three steps using the function `make_params(qbits,pbits)`.

```
import random
random.seed()

### Omitted: source for is_prime(n) (see previous problem),
###    and its helper function, is_witness(a,n).

def make_q(qbits):
        while True:
                q = random.randrange(2**(qbits-1),2**qbits)
                if q%2 == 0: q += 1
                if is_prime(q): return q

def make_p(pbits,q):
        while True:
                minP = 2**(pbits-1)
                maxP = (2**pbits)-1
                minK = (minP+q-2)/q # This is the ceiling of (minP-1)/q
                maxK = (maxP-1)/q
                k = random.randrange(minK,maxK+1)
                p = k*q + 1
                if is_prime(p): return p

def make_g(p,q):
        while True:
```

```
                b = random.randrange(2,p)
                a = pow(b,(p-1)/q,p)
                if a != 1: return a

def make_params(qbits,pbits):
        q = make_q(qbits)
        p = make_p(pbits,q)
        g = make_g(p,q)
        return p,q,g

# I/O
qbits,pbits = map(int,raw_input().split())
p,q,g = make_params(qbits,pbits)
print p,q,g
```