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

1. Exercise 4.2.

Solution.
We need only check where $S^e \equiv D \pmod{N}$ in each case. Here’s a transcript of a Python session doing this.

```python
>>> N = 1562501
>>> e = 87953

>>> D = 119812
>>> S = 876453
>>> pow(D,e,N)
574748
>>> pow(S,e,N)
772481
>>> D == pow(S,e,N)
False

>>> D1 = 161153
>>> S1 = 870099
>>> pow(S1,e,N)
161153
>>> pow(S1,e,N) == D1
True

>>> D2 = 586036
>>> S2 = 602754
>>> pow(S2,e,N)
586036
>>> pow(S2,e,N) == D2
True
```

So the second and third signatures are valid, while the first is not.

2. Exercise 4.4.

Solution.
Verification works because assuming Alice has produces $(c, s)$ according to this protocol, they will satisfy

$$
s \equiv \text{Hash}(m)^{d_A} \pmod{N_A}
$$

$$
s^{e_A} \equiv \text{Hash}(m)^{e_A d_A} \pmod{N_A}
\quad \equiv \text{Hash}(m) \pmod{N_A}
$$

since $e_A, d_A$ are an encryption-decryption pair for modulus $N_A$. 
If Eve has a message $m$ that she wishes to send to Bob, claiming that it came from Alice, she could certainly compute $\text{Hash}(m)$ and the ciphertext $c$ (using Bob’s public key), but now she is stymied: she must forge a signature $s$ for the “document” $\text{Hash}(m)$ in the RSA signature scheme, which is believed to be difficult.

Eve might hope instead to achieve a more modest goal: producing a pair $(c, s)$ that verifies as a valid signed and encrypted message from Alice, without necessarily choosing $m$ in advance (similar to the process in problem 5). But now she is stymied by the hash function: she can certainly choose a signature $s$ however she likes, and forge from it a “document” $D \equiv s^{e_A}$ (mod $N_A$). But now she would need to invert the hash function if she wants to figure out what message $m$ goes with the “document” (i.e. solve $\text{Hash}(m) = D$).

It is conceivable that there could be some much more clever approach, in which $c$ and $s$ are created simultaneously, but such an approach would need to rely on some information about the specific hash function being used. A strong hash function is likely to resist such approaches.

3. Exercise 4.5.

**Solution.**

(a) $A \equiv g^a$ (mod $p$); in this case it is $437^{6104}_{6961} = 2065$.

(b) Following table, we can compute as follows (here `inverse` is a function I have previously implemented to find the modular inverse using the extended Euclidean algorithm).

```python
>>> p = 6961
>>> g = 437
>>> a = 6104
>>> A = pow(g,a,p)
>>> D = 5584
>>> k = 4451
>>> s1 = pow(g,k,p)
>>> s1
3534
>>> s2 = inverse(k,p-1)*(D - a*s1) % (p-1)
>>> s2
5888
```

So the signature is $(S_1, S_2) = (3534, 5888)$. Indeed, we can check that it verified correctly as follows.

```python
>>> pow(2065,3534,p) * pow(3534,5888,p) % p
4094
>>> pow(437,5584,p)
4094
```


The following transcript shows the necessary computations.

```python
>>> g = 437
>>> A = 4250

>>> D = 1521
```
>>> S1, S2 = 4129, 5575
>>> pow(A, S1, p) * pow(S1, S2, p) % p
231
>>> pow(g, D, p)
231

>>> D = 1837
>>> S1, S2 = 3145, 1871
>>> pow(A, S1, p) * pow(S1, S2, p) % p
6208
>>> pow(g, D, p)
2081

>>> D = 1614
>>> S1, S2 = 2709, 2994
>>> pow(A, S1, p) * pow(S1, S2, p) % p
2243
>>> pow(g, D, p)
2243

The two sides of the verification equation match in the first and third examples, but not in the second. So the valid signatures are the first and the third.

5. Exercise 4.7.

Solution.

Suppose that $i,j$ are integers with $\gcd(j, p-1) = 1$ and $S_1 \equiv g^i A^j \pmod{m}$ (we will come to the choices of $S_2$ and $D$ later). Then the left side of the verification equation can be expressed:

$$A^{S_1 S_2} \equiv A^{S_1 g^{iS_2} A^{jS_2}} \pmod{p}$$

$$\equiv g^{iS_2} A^{S_1 + jS_2} \pmod{p}$$

Now, as long as $S_2$ is selected to be $-j^{-1} S_1 \pmod{p}$, the exponent $S_1 + jS_2$ of $A$ will be congruent to 0 modulo $p-1$, and in turn the entire left hand side will be congruent to $g^{iS_2}$. This will be congruent to $g^D$ modulo $p$ so long as $D$ is chosen to be congruent to $iS_2$ modulo $p-1$. Indeed, since $S_2 \equiv -j^{-1} S_1 \pmod{p-1}$, this will hold as long as $D$ is chosen to be $-ij^{-1} S_1 \pmod{p-1}$, which is precisely what is prescribed in the problem.

Note. The key insight here is Eve’s idea to start by computing $S_1 \equiv g^i A^j \pmod{p}$. This choice allows her to rig up the choice of $S_2$ to deliberately cancel away the power of $A$ on the left side of the equation, leaving her with a known power of $g$ (that then serves as her document).


Solution.

(a) If Samantha uses the same key $k$ twice, then the same signature element $S_1$ will occur twice, which Eve can tell immediately, if she is looking for it.
Problem Set 7

(b) From the second part of the signing equations, Eve knows the following congruences.

\[ aS_1 + kS_2 \equiv D \pmod{p-1} \]
\[ aS'_1 + k'S'_2 \equiv D' \pmod{p-1} \]

In the case where \( S_1 = S'_1 \), Eve may deduce (assuming that \( g \) is a primitive root) that \( k = k' \). Hence she will have these two congruences.

\[ aS_1 + kS_2 \equiv D \pmod{p-1} \]
\[ aS'_1 + kS'_2 \equiv D' \pmod{p-1} \]

She can take two approaches in trying to find \( a \). She can subtract one congruence from the other, solve for \( k \), and then substitute to find \( a \). Alternatively, she can more directly solve for \( a \) by taking a linear combination of the two congruences. Specifically, she can subtract \( S_2 \) times the second congruence from \( S'_2 \) times the first to obtain

\[ S'_2(aS_1 + kS_2) - S_2(aS_1 + kS'_2) \equiv S'_2D - S_2D' \pmod{p-1} \]
\[ \iff (S'_2S_1 - S_2S_1)a + (S'_2S_2 - S_2S'_2)k \equiv S'_2D - S_2D' \pmod{p-1} \]
\[ \iff (S'_2 - S_2)S_1a \equiv S'_2D - S_2D' \pmod{p-1} \]

In a perfect world (for Eve), she can now solve this congruence by multiplying by the inverse of \((S'_2 - S_2)S_1\). It is possible that this number has factors in common with \( p - 1 \), however; in that case she must first divide through by those common factors, obtain a solutions determined modulo \((p-1)/G\) (where \( G \) is the greatest common divisor), and use trial and error in the equation \( g^a \equiv A \pmod{p} \) to determine the true private key. This is demonstrated in part (c), where indeed there is a common factor to worry about.

(c) In this case, we may use the approach above and compute

\[ (S'_2 - S_2)S_1 \equiv 347960 \pmod{p-1} \]
\[ S'_2D - S_2D' \equiv 252868 \pmod{p-1} \]

For readability, I will write \( m = 347960 \) and \( b = 252868 \). So it is necessary to solve \( ma \equiv b \pmod{p-1} \) for \( a \).

Unfortunately, we compute that \( \gcd(m, p - 1) = 4 \), so there is no inverse of \( m \) modulo \( p - 1 \). But 4 divides the right side also, and we can divide the left side, ride side, and modulus by 4 to obtain \( \frac{m}{4}a \equiv \frac{b}{4} \pmod{\frac{p-1}{4}} \), which can be solved by obtaining an inverse. The inverse of \( \frac{m}{4} \) modulo \( \frac{p-1}{4} \) can be found with the extended Euclidean algorithm. It is \( 37037 \). So we obtain \( a \equiv 37037 \cdot \frac{252868}{4} \pmod{\frac{348149-1}{4}} \), or, simplifying,

\[ a \equiv 72729 \pmod{\frac{p-1}{4}}. \]

Let \( a_0 = 72729 \). At this stage, it is necessary to do some trial and error. The value of \( a \) modulo \( p - 1 \) must be one of \( a_0, a_0 + \frac{p-1}{4}, a_0 + 2\frac{p-1}{4}, a_0 + 3\frac{p-1}{4} \) (if we add any other multiple of \( \frac{p-1}{4} \), the result will differ from one of these by a multiple of \( p - 1 \)). These are 72729, 159766, 246803, 333840. Now, we can test each one to see if it is the correct discrete logarithm of \( A \).

\[
\begin{align*}
113459^{72729} & \equiv 185149 \pmod{348149} \\
113459^{159766} & \equiv 137653 \pmod{348149} \\
113459^{246803} & \equiv 163000 \pmod{348149} \\
113459^{333840} & \equiv 210496 \pmod{348149}
\end{align*}
\]
The public key we are looking for is $A = 185149$, so the first option is the correct one: the private key is 72729.


Solution.

(a) The verification key is $A \equiv 4488^{674} \equiv 4940 \pmod{22531}$.

(b) To compute $S_1$, Alice first computes $g^k \equiv 4488^{574} \equiv 7954 \pmod{p}$, and then takes the remainder modulo $q = 751$ to obtain $S_1 = 444$.

To obtain $S_2$, Alice finds $k^{-1} \equiv 297 \pmod{751}$ (note that this is the inverse modulo $q$, not modulo $p$), and $D + aS_1 \equiv 602 \pmod{q}$, hence $S_2 \equiv k^{-1}(D + aS_1) \equiv 297 \cdot 602 \equiv 56 \pmod{q}$.

8. Consider the following implementation of one trial of Pollard’s algorithm (from solutions to last week’s problem set).

```python
def pollard_findfact(N):
    a = random.randrange(1,N)
    # Check first whether a is a unit. If not, you have a factor.
    if fractions.gcd(a,N) != 1:
        return fractions.gcd(a,N)
    j = 2
    while fractions.gcd(a-1,N) == 1:
        a = pow(a,j,N)
        j += 1
    return fractions.gcd(a-1,N)
```

(a) Suppose that this function is called on an input $N = pq$, a product of two distinct primes. Prove that in principle (i.e. given an unbounded amount of time), this function will always return some factor of $N$ other than 1. Under what circumstances will it return $N$, rather than a proper factor?

(b) Suppose that this function is called on $N = pq$, where both $p - 1$ and $q - 1$ have at least one prime factor greater than $2^{256}$. Estimate how large you expect $j$ to grow before this function will return an answer, and explain why. The result will depend on the random value of $a$ this is chosen; try to justify why the estimate you give will be correct with very high probability.

Solution.

(a) There are only two situations when the function will return: when the initial value of $a$ has a common factor with $N$ (larger than 1), or when a later value of $a$ is such that $a - 1$ is a common factor with $N$ that is larger than 1. In both cases, that common factor of $N$ is returned. So if the function returns, it returns a factor of $N$ larger than 1. We still need to check that the function will definitely return some value eventually (i.e. that it won’t loop forever) also also to analyze when the value returned is $N$ itself.

Let $a_0$ be the initial value of $a$ (the random number from 1 to $N - 1$ inclusive). One possibility (though unlikely) is that $a$ already has a common factor with $N$; in this case
the function returns a value immediately (which cannot be \(N\), since \(a_0\) is never chosen to be a multiple of \(N\)).

The other possibility is that \(a_0\) is a unit modulo \(N\), i.e. it is not divisible by \(p\) or \(q\). In this case, the while loop will iterate some number of times. Each time the value of \(a\) is updated, it’s value becomes \(a \equiv a_0^{j!} \pmod{N}\). The function will return a value as soon as \(a_0^{j!} - 1\) is either divisible by \(p\) or divisible by \(q\), i.e. as soon as either \(a^{j!} \equiv 1 \pmod{p}\) or \(a^{j!} \equiv 1 \pmod{q}\). From Fermat’s little theorem, we know that as soon as \(p - 1\) divides \(j!\), it will follow that \(a^{j!} \equiv 1 \pmod{p}\) (and similarly for \(q\)). This is certain to happen eventually; certainly it will happen once \(j \geq p - 1\), for example. Usually it will happen much sooner.

So the function is sure to return eventually. The remaining question is when it will return \(N\) itself. This can only happen when \(a - 1\) becomes divisible by \(p\) and by \(q\) simultaneously. In other words, this occurs when the minimum value of \(j\) such that \(a^{j!} \equiv 1 \pmod{p}\) is the same as the minimum value of \(j\) such that \(a^{j!} \equiv 1 \pmod{q}\).

(b) I claim that except for very special values of \(a\) (which are unlikely to be chosen at random), this function would need to iterate at least \(2^{256}\) times (i.e. until \(j \geq 2^{256}\)) before returning an answer. Note that this is much longer than could ever complete during your lifetime (or the age of the universe, for that matter).

Note first of all that there is the possibility that \(j\) never grows at all, if \(\gcd(a, N) \neq 1\) at the very beginning. However, this is quite unlikely: there are \(q - 1\) multiples of \(p\) in \(\{1, 2, \cdots, N - 1\}\), and \(p - 1\) multiples of \(q\), so the probability that a non-unit \(a\) is chosen is \(\frac{(q-1)(p-1)}{N-1}\). Since \(p, q\) are roughly \(\sqrt{N}\) in size, this probability will be in the neighborhood of \(2/\sqrt{N}\) – extremely small when \(N\) has at least 1000 bits in it, for example.

So now assume that \(a\) is chosen to be a unit. Denote by \(p'\) the largest prime factor of \(p - 1\), and by \(q'\) the largest prime factor of \(q - 1\). So we are assuming that \(p', q' \geq 2^{256}\).

We need to know how large \(j\) is before \(a^{j!} \equiv 1 \pmod{p}\). Now, \(a^{j!} \equiv 1 \pmod{p}\) if and only if \(\ord_{p}(a) \mid j\). Observe that the only prime factors of \(j!\) are the primes \(\leq j\). So \(a^{j!} \equiv 1 \pmod{p}\) can only occur if all of the prime factors of \(\ord_{p}(a)\) are less than or equal to \(j\). Now, I claim that with very high probability, the prime \(p'\) divides the order of \(a\) modulo \(p\). If this is so, then \(a^{j!} \equiv 1 \pmod{p}\) would imply that \(j \geq p' \geq 2^{256}\), so that \(j\) would need to grow at least to \(2^{256}\).

We know from Fermat’s little theorem that \(\ord_{p}(a)\) always divides \(p - 1\), of which \(p'\) is a large prime factor. It seems plausible that elements of small order are unusual, and hence that for most values of \(a\), the order of \([a]_p\) is divisible by \(p'\). For this assignment, you are not expected to prove this formally, but here is a proof for your interest.

**Claim.** If \(a\) is a randomly chosen unit modulo \(N\), then the probability that \(p' \mid \ord_{p}(a)\) is 1 – \(\left(\frac{1}{p'}\right)^e\), where \(e\) is the exponent with which \(p'\) appears in the prime factorization of \(p - 1\).

**Proof.** By the Chinese remainder theorem, choosing a unit modulo \(N\) is equivalent to choosing a unit modulo \(p\) and a unit modulo \(q\). Therefore \(a\) is equally likely to be in any of the nonzero congruence classes modulo \(p\). Let \(g\) be a primitive root modulo \(p\). Then
the nonzero congruence classes modulo $p$ are $\{[g]_p^0, [g]_p^1, [g]_p^2, \cdots, [g]_p^{p-1}\}$. So $[a]_p$ is one of these classes, with each one equally likely. Suppose that $[a]_p = [g]_p^i$. By problem set 4, question 4(c), the order of $a$ is equal to \(\text{ord}[g]_p^i = \text{ord}[g]_p / \gcd(\text{ord}[g]_p, i) = (p-1)/\gcd(i, p-1)\). Now, this is divisible by $p'$ unless $\gcd(i, p-1)$ is divisible by $(p')^e$ (and hence cancels all of the copies of $p'$ from $p-1$), which is true if and only if $(p')^e \mid i$. But there are only $(p-1)/(p')^e$ multiples of $(p')^e$ between 0 and $p-2$ inclusive. Therefore, since $i$ is equally likely to be any number in $\{0, 1, \cdots, p-2\}$, the probability that $\text{ord}[a]_p$ is not divisible by $p'$ is $1/(p')^e$, so the probability that is is divisible by $p'$ is $1 - \left(\frac{1}{p}\right)^e$. 

So the probability that $a^{j!} \equiv 1 \pmod{p}$ for some $j < 2^{256}$ is at most $1 - \frac{1}{p^2}$, which is at most $1 - \frac{1}{2^{256}}$ (essentially negligible). Similarly, the probably that $a^{j!} \equiv 1 \pmod{q}$ for some $j < 2^{256}$ is at most $1 - \frac{1}{2^{256}}$. Therefore, this function will have to run at least $2^{256}$ iterations (far too many to finish before the world ends) of the while loop before it returns, except in four unusual circumstances: $p \mid a$, $q \mid a$, $p' \nmid \text{ord}[a]_p$, or $q' \nmid \text{ord}[a]_q$. Each of these circumstances occurs with essentially negligible probability, as is argued above.

### Problem Set 7

#### Programming problems

Full formulation and submission: [https://www.hackerrank.com/m158-2016-pset-7](https://www.hackerrank.com/m158-2016-pset-7)

9. Write a program that verifies whether or not a given DSA signature is valid. You will be given the public parameters and public key, and a document with a purported signature.

**Solution.**

We can follow Table 4.3 as follows. The inverse of $S_2 \pmod{q}$ is obtained using the extended Euclidean algorithm, as implemented on previous assignments.

```python
### Omitted: source for function ext_euclid

def verify(p, q, g, A, D, s1, s2):
    s2inv = ext_euclid(s2, q)[0] % q
    v1 = s2inv * D % q
    v2 = s2inv * s1 % q
    return pow(g, v1, p) * pow(A, v2, p) % p % q == s1

# I/O
p, q, g, A = map(int, raw_input().split())
D, s1, s2 = map(int, raw_input().split())
if verify(p, q, g, A, D, s1, s2):
    print 'valid'
else:
    print 'invalid'
```

10. Problem 6 showed that a pair of ElGamal signatures using the same ephemeral key can accidentally give away the signer’s private key. The same is true in DSA – write a program which take public parameters and a public key for DSA, along with two signed documents that have been signed with the same ephemeral key, and computes the signer’s private key from this information.

Due the night of Thursday 11/3 (hard deadline 4am on 11/4).
Solution.

Suppose that \((D, S_1, S_2)\) is one document with a valid signature, and \((D', S'_1, S'_2)\) is another. Let \(k, k'\) be the ephemeral keys used to make these two signatures. If these were accidentally made with the same ephemeral key, then \(k = k'\) and \(S_1 = S'_1\). From the second signing congruence in Table 4.3, we obtain the following pair of congruences.

\[
kS_2 \equiv D + aS_1 \pmod{q} \\
k'S_2 \equiv D' + aS_1 \pmod{q}
\]

As in problem 6, we can cancel \(k\) from these congruences by multiplying the first by \(S'_2\), and subtracting \(S_2\) times the second from it, to obtain

\[
0 \equiv (S'_2D - S_2D') + a(S'_2 - S_2)S_1 \pmod{q}
\]

We are given that \(S_1 \not\equiv 0 \pmod{q}\) and that \(S'_2 - S_2 \not\equiv 0 \pmod{q}\), hence both are units modulo \(q\). So we can obtain \(a\) as

\[
a \equiv (S_1(S_2 - S'_2))^{-1} (S'_2D - S_2D') \pmod{q}.
\]

This is implemented below.

### Omitted: source for the function ext_euclid

```python
def extract(p,q,g,A,D1,S11,S21,D2,S12,S22):
    assert(S11 == S12)
    coeff = S11 * (S21-S22) % q
    cinv = ext_euclid(coeff,q)[0] % q
    return cinv * (D1*S22 - D2*S21) % q
```

11. Suppose that Samantha and Victor are using a variant of Elgamal signatures, in which the verification congruence that Victor will use is \(S_1S_1gS_2 \equiv AD \pmod{p}\). Given the public parameters, Samantha’s secret signing key, and a document \(D\), produce a valid signature for this system.

**Solution.**

The first step, generation of \(S_1\), can be taken in the same manner as in Elgamal: choose \(k\) at random and set \(S_1 \equiv g^k \pmod{p}\). Since the online specifications required that \(S_1 \neq 1\), we should choose \(k \neq 0 \pmod{p-1}\) (in practice, the probability that this would occur is negligible), so below we choose \(k\) from \(\{1, 2, \cdots, p-2\}\).

The second step is to choose \(S_2\) so that \(S_1^S_1 g^{S_2} \equiv AD \pmod{p}\). Taking discrete logarithms, this congruence is equivalent to \(kS_1 + S_2 \equiv aD \pmod{p-1}\). Therefore we can compute \(S_2\) using \(S_2 \equiv aD - kS_1 \pmod{p-1}\).

The implementation is fairly short.
from random import randrange

def sign(p,g,a,D):
    k = randrange(1,p-1)
    s1 = pow(g,k,p)
    s2 = (a*D - s1*k)%(p-1)
    return s1,s2

# I/O
p,g,a,D = map(int,raw_input().split())
s1,s2 = sign(p,g,a,d)
print s1,s2