Written problems

1. Write some code to answer the following question: given an integer \( n \), what is the probability that two numbers \( a, b \) chosen randomly from 1 to \( n \) inclusive satisfy \( \gcd(a, b) = 1 \)? Determine, to six decimal places, this probability for \( n = 10, 100, 1000, \) and 10000 (your computer will probably have to think for awhile on the last one). Guess the trend as \( n \) goes to infinity.

Solution. The following code can be used to find the desired probability, up to any particular bound \( N \).

```python
from fractions import gcd

# Count pairs (a,b), each from 1 to N inclusive, that are relatively prime.
def count(N):
    num = 0
    for a in xrange(1,N+1):
        for b in xrange(1,N+1):
            if gcd(a,b) == 1:
                num += 1
    return num

def prob(N):
    return float(count(N)) / (N*N)
```

The four desired probabilities can now be found in the Python interpreter as follows.

```python
>>> prob(10)
0.63
>>> prob(100)
0.6087
>>> prob(1000)
0.608383
>>> prob(10000)
0.60794971
```

These data suggest that this probability is tending towards a nonzero limit near 0.608. In fact, it is possible to prove that this limit exists and is equal to \( 6/\pi^2 \), which is approximately 0.607927. Later in the course, we may discuss how to prove this fact.

2. The Euclidean algorithm is built into Python in the `fractions` module. To read its source code and documentation, type the following three commands into the python interpreter.

```python
>>> import fractions
>>> import inspect
>>> print inspect.getsource(fractions.gcd)
```

State carefully the exact circumstances under which \( \gcd(a,b) \) will return a negative result (the comment in source code gives most of the answer), and explain why. You may need to look up the precise behavior of the operator \( \% \) when negative numbers are used. Briefly explain whether this function will even return 0, and why.
Solutions. The rather concise implementation of gcd is simply the following.

```python
def gcd(a, b):
    while b:
        a, b = b, a%b
    return a
```

When \( a \nmid b \), the sign of \( a\%b \) is the same as the sign of \( b \). Therefore, in each iteration of the while loop, the sign of \( b \) is unchanged (until it is eventually set to 0). The value of \( a \) at the end of the function will be equal to a previous value of \( b \), and therefore have the same sign as the original value of \( b \), except in one situation: when the while loop iterates 0 times, i.e. the original value of \( b \) was 0. In this situation, the function simply returns the original value of \( a \).

Therefore this function returns a negative number in two situations.

- \( b = 0 \) and \( a < 0 \).
- \( b < 0 \).

The function can return 0, but only for one input: \((0, 0)\). The value of \( b \) must be 0, because otherwise the while loop would iterate at least once, and \( a \) is always reset to be a nonzero value when the while loop iterates. But when \( b = 0 \), the function returns \( a \) itself. So indeed \((0, 0)\) is the only input that returns 0.

3. Textbook exercise 1.16, parts (a),(b),(c).

**Solution**

(a)

<table>
<thead>
<tr>
<th>+</th>
<th>0</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>( \cdot )</th>
<th>0</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

(b)

<table>
<thead>
<tr>
<th>+</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>5</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>( \cdot )</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>2</td>
<td>4</td>
<td>0</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>4</td>
<td>2</td>
<td>0</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

(c)

<table>
<thead>
<tr>
<th>( \cdot )</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>5</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>5</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>4</td>
<td>8</td>
<td>1</td>
<td>5</td>
<td>7</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>8</td>
<td>7</td>
<td>2</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>1</td>
<td>2</td>
<td>7</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>7</td>
<td>5</td>
<td>1</td>
<td>8</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>8</td>
<td>7</td>
<td>5</td>
<td>4</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>


**Solution**
By the Extended Euclidean Algorithm, there must exist integers \( u, v \) such that \( au + bv = \gcd(a, b) \). By multiplying \( g^u \) by itself \( u \) times (or its inverse by itself \( -u \) times if \( u < 0 \)), we obtain \( g^{au} \equiv 1 \pmod{m} \). Similarly, \( g^{bv} \equiv 1 \pmod{m} \). Multiplying these together gives \( g^{au+bv} \equiv 1 \pmod{m} \), which is the same thing as \( g^\gcd(a, b) \equiv 1 \pmod{m} \).

5. Textbook exercise 1.24, part (a).

**Solution.**

The congruence \( x \equiv 3 \pmod{7} \) holds if and only if there exists an integer \( y \) such that \( x = 3 + 7y \). For any such choice of \( x \) and \( y \), \( x \equiv 4 \pmod{9} \) holds if and only if \( 3 + 7y \equiv 4 \pmod{9} \), which is equivalent to \( 7y \equiv 1 \pmod{9} \). By trial and error (or the extended Euclidean algorithm), the inverse of 7 modulo 9 is 4. Therefore 7 \( y \equiv 4 \pmod{9} \), which is to say there exists an integer \( z \) such that \( y = 4 + 9z \).

Therefore any \( x \) satisfying both congruences must be equal to \( 3 + 7y \), where \( y = 4 + 9z \), where \( z \) is an integer. In this case, \( x = 3 + 7(4 + 9z) = 31 + 63z \), so \( x \equiv 31 \pmod{63} \).

So one single value \( x \) is \( x = 31 \); indeed \( 31 \equiv 7 = 3 \) and \( 31 \equiv 9 = 4 \).  

**Note.** This problem is a simple example of a general and very useful procedure, which we will study in detail when we discuss the Chinese Remainder Theorem.


**Solution.**

(a) We are given that \( [b]_p = [a]_p^{(p-1)/q} \). Raising both sides to the power \( q \), \( [b]_p^q = [a]_p^{p-1} \equiv [1]_p \), by Fermat’s little theorem. By Proposition 1.29, this means the order of \( [b]_p \) is a divisor of \( q \). Since \( q \) is prime, its only positive integer divisors are 1 and \( q \). So the order of \( [b]_p \) is either 1 or \( q \). But its order is 1 only if \( [b]_p^1 = [1]_p \), i.e. \( [b]_p = [1]_p \). Therefore either \( [b]_p = [1]_p \), or \( [b]_p \) has order \( q \).

(b) By Theorem 1.30, there is a primitive root \([g]_p \) modulo \( p \), and the set of all units modulo \( p \) is equal to the set of powers of \([g]_p \). Therefore

\[
\#\{[a]_p \in (\mathbb{Z}/p)^* : [a]_p^{(p-1)/q} \neq [1]_p\} = \#\{e \in \{0, 1, 2, \ldots, p-2\} : ([g]_p^e)^{(p-1)/q} \neq [1]_p\}.
\]

In effect, this equation amounts to counting the units modulo \( p \) in a different order: instead of counting them by adding 1 repeatedly, we count be multiplying by \([g]_p \) repeatedly. This method makes it much easier to analyze the desired quantity.

The order of \([g]_p \) is \((p-1)\), so \([g]_p^{k_p} = [1]_k \) if and only if \((p-1) \mid k \) (by Proposition 1.29). Therefore \(([g]_p^e)^{(p-1)/q} = [1]_p \) if and only if \( e(p-1)/q \) is divisible by \((p-1)\), i.e. if and only if \( e/q \) is an integer, which is the same as saying that \( q \mid e \).

Therefore the number \#\{\( [a]_p \in (\mathbb{Z}/p)^* : [a]_p^{(p-1)/q} \neq [1]_p \}\} is simple the number of integers \( e \in \{0, 1, 2, \ldots, p-2\} \) that are not multiples of \( q \). There are \((p-1)/q \) multiples of \( q \), and therefore

\[
\frac{\#\{[a]_p \in (\mathbb{Z}/p)^* : [a]_p^{(p-1)/q} \neq [1]_p\}}{\#(\mathbb{Z}/p)^*} = \frac{(p-1) - (p-1)/q}{p-1} = 1 - \frac{1}{q}.
\]
Therefore the probability that the number \( a^{(p-1)/q} \) is an element of order \( q \) is \( 1 - \frac{1}{q} \). This is quite high, especially if \( q \) is large. Therefore this exercise shows that this randomized procedure gives a highly effective way to produce an element of order \( q \) (this is a crucial task in creating parameters for several cryptographic systems, especially the Digital Signature Algorithm).

7. Textbook exercise 1.34, parts (c),(d),(e).

**Solution.**

(c) We can check whether any given number is a primitive root by listing its first \( p - 2 \) powers \( \pmod{p} \) and making sure that none of them are congruent to 1.

```python
>>> def is_pr(g,p):
...     if g%p == 0: return False
...     power = 1
...     for i in range(p-2):
...         power = power * g % p
...         if power == 1:
...             return False
...     return True
...```

Using this function we can start trying potential values of \( g \) until we find a primitive root.

```python
>>> is_pr(2,23)
False
>>> is_pr(3,23)
False
>>> is_pr(4,23)
False
>>> is_pr(5,23)
True
>>> is_pr(2,29)
True
>>> is_pr(2,41)
False
>>> is_pr(3,41)
False
>>> is_pr(4,41)
False
>>> is_pr(5,41)
False
>>> is_pr(6,41)
True
>>> is_pr(2,43)
False
```

So 5 is a primitive root modulo 23. Similarly, we can find a primitive root for each of the other primes mentioned this way.
>>> is_pr(3,43)
True
so 2 is a primitive root modulo 29, 6 is a primitive root modulo 41, and 3 is a primitive root modulo 43.
For reference, here are lists of all prime roots for each of these primes.

\[
\begin{align*}
p &= 23 & 5, 7, 10, 11, 14, 15, 17, 19, 20, 21 \\
p &= 29 & 2, 3, 8, 10, 11, 14, 15, 18, 19, 21, 26, 27 \\
p &= 41 & 6, 7, 11, 12, 13, 15, 17, 19, 22, 24, 26, 28, 29, 30, 34, 35 \\
p &= 43 & 3, 5, 12, 18, 19, 20, 26, 28, 29, 30, 33, 34 \\
\end{align*}
\]

(d) Using the \texttt{is\_pr} function written above, we can quickly produce this list as follows.

```python
>>> for g in range(1,11):
    if is_pr(g,11): print g,
2 6 7 8
```
Indeed, there are 4 of them, and \(\phi(10) = 4\) (the numbers relatively prime to 10 are 1,3,7,9).

(e) We can define the following function to return all the primitive roots of a given prime as a list.

```python
>>> def all_prs(p):
    res = []
    for g in range(1,p):
        if is_pr(g,p): res += [g]
    return res
```
Calling this function on \(p = 229\) gives the full list.

```python
>>> all_prs(229)
```
We can quickly determine the length of this list as follows.

```python
>>> len(all_prs(229))
72
```
So there are 72 primitive roots modulo 229. To check that this is equal to \(\phi(228)\), we can quickly put together a naive (but efficient enough for primes of this size) algorithm to commute \(\phi\) directly from the definition as follows.

```python
>>> import fractions
>>> def phi(m):
    res = 0
    for a in range(m):
        if is_pr(a,m): res += 1
    return res
```
Due the night of Thursday 9/22 (hard deadline 4am on 9/23).
... if fractions.gcd(a,m) == 1: res += 1
... return res
...

>>> phi(228)
72
So indeed the number of primitive roots modulo 229 is equal to $\phi(228)$.

8. Textbook exercise 2.4.

Solution.

(a) The powers of 2 modulo 23 are 2, 4, 8, 16, 9, 18, 3, 6, 12, 1, · · · (period 11). Therefore 
$\log_2 13 = 7$ is one solution. More generally, since the order of 2 modulo 23 is 11 (as seen above), this discrete logarithm can be described by the class 7 (mod 11).

(b) The powers of 10 modulo 47 are 10, 6, 13, 36, 31, 28, 45, 27, 35, 21, 22, · · · (this sequence turns out to have period 46, though I won’t copy it all out), so $\log_{10}(22) = 11$ (or more generally, 11 (mod 46)).


Solution. Suppose first that the element $[a]_p$ has a square root modulo $p$. This means that there exists an element $[x]_p$ such that $[x]^2_p = [a]_p$. There exists some integer $e$ (the discrete logarithm of $[x]_p$ with base $[g]_p$) such that $[x]_p = [g]^e_p$. It follows that $[g]^{2e}_p = [a]_p$. In other words, $\log_g[a]_p = 2e$, which is even. (More precisely, we should say the $2e$ is a discrete logarithm, since anything congruent to it modulo $p - 1$ will also work; but since $p$ is odd all such numbers are even, so this ambiguity does not affect the result).

Conversely, suppose that the discrete logarithm $\log_g[a]_p$ is even. Then there is an integer $e$ such that this discrete logarithm is $2e$. It follows that $[g]^{2e}_p = [a]_p$, and therefore $[a]_p = ([g]_p^e)^2$. Hence $[a]_p$ has a square root modulo $p$.

Programming problems

Full formulation and submission: https://www.hackerrank.com/m158-2016-pset-2

10. Write a problem to solve linear congruences of the form $ax \equiv b \pmod{M}$. If the congruence has solutions, your program should give a single congruence $x \equiv c \pmod{N}$ that describes them all. If the congruence has no solutions, your program must detect this.

Solution

In the special case gcd($a, M$) = 1 (as in the first ten test cases), a solution is guaranteed to exist, and it can be found in the same way as in usual algebra: multiplying be an inverse of $a$. The congruence $ax \equiv b \pmod{M}$ is true if and only if $x \equiv ba^{-1} \pmod{M}$. The value $a^{-1} \pmod{M}$ can be found quickly as the integer $u$ in a solution to $au + Mv = 1$.

In the general case, where gcd($a, M$) ≠ 1, a solution may not exist. The congruence $ax \equiv b \pmod{M}$ is equivalent to saying that there exists an integer $k$ such that $ax = b + kM$. If $g = \gcd(a, M)$, then $g$ divides both $ax$ and $kM$, so it must divide $b$ in order for a solution to exist. Therefore if $g \nmid b$, we can immediately conclude that there are no solutions and terminate the program.

On the other hand, if $g \mid b$, then we can reduce to the case gcd($a, M$) = 1 as follows.
\[ ax \equiv b \pmod{M} \]
\[ \iff \exists k \in \mathbb{Z}: \ ax - b = kM \]
\[ \iff \exists k \in \mathbb{Z}: \ \frac{a}{g}x - \frac{b}{g} = k \]
\[ \iff \frac{a}{g}x \equiv \frac{b}{g} \pmod{M/g}. \]

Since \( \frac{a}{g}u + \frac{M}{g}v = 1 \) (for the same \( u \) and \( v \) that give \( au + Mv = g \)), it is now the case that \( \frac{a}{g} \)

is a unit modulo \( \frac{M}{g} \); its inverse is \( u \). Therefore the original congruence is equivalent to

\[ x \equiv \frac{u}{g}b \pmod{M/g}. \]

Hence we need to solve the equation \( au + bM = g \), which can be done at the same time that \( g \)

is computed (using the extended Euclidean algorithm); then the desired answer will

be \( N = \frac{M}{g} \), \( c = \left( \frac{u}{g}b \right) \% N \). Here is an implementation.

```python
# 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]

# Returns None if ax = b mod M has no solutions.
# Otherwise returns a pair (c,N) where the solution is x = c mod N.
def solve(a,b,M):
    u,v,g = ext_euclid(a,M)
    if b%g != 0: return None
    return ( ((b/g)*u)%(M/g), M/g )

# Read the input
[a,b,M] = map(int,raw_input().split())

# Compute and print the output
soln = solve(a,b,M)
if soln == None: print 'None'
else: print soln[0],soln[1]
```

11. Write a program that performs Diffie-Hellman key exchange, as in Table 2.2. You will be
given \( p, g, \) and \( A \), where \( p \) is guaranteed to be a 1024-bit prime; you must produce \( B \) and the
shared secret. You should look up how to generate random numbers (but you do not need to find a “cryptographically secure” way for purposes of this assignment), and the autograder will check to make sure that your program is not deterministic.

Solution.

As described in the book, the values $B$ and $S$ are computed by choosing an ephemeral key $b \in \mathbb{Z}/(p - 1)$ and then computing $B \equiv g^b \pmod{p}$ and $S \equiv A^b \pmod{p}$. Once we have a fast-powering algorithm, this can be done in a few lines as follows.

```python
import random
random.seed()

# Fast powering algorithm
# Returns (a^e)%m
def fastpow(a,e,m):
    res = 1
    while e>0:
        if e%2: res = (res*a)%m
        a = (a*a)%m
        e /= 2
    return res

# Read the input
[p,g,A] = map(int,raw_input().split())

# Generate the ephemeral key
b = random.randrange(1,p-1)

# Compute the public number B and shared secret S
print fastpow(g,b,p), fastpow(A,b,p)
```

Note. I’ve implemented a fast modular powers function here, in order to show how it works. However, this is actually built into Python as simply `pow(a,e,m)`, so you can just use that rather than implementing it yourself.

12. You play Eve in this problem. You have intercepted six encrypted messages sent from Alice to Bob. The cryptosystem they are using converts a string (the plaintext) into an integer (the ciphertext); so the data you have intercepted consists of six integers. You have also obtained the source code Alice and Bob are using for their encryption; it is reproduced below on the last page (you can also find it in the Python starter code on hackerrank). Alice and Bob have a secret key $k$, which is a 1024-bit integer.

Write a program to break Alice and Bob’s encryption, and print the original six plaintext messages.

Note. Because Eve accomplishes this attack using only a selection of a few enciphered messages, this is called a ciphertext-only attack.

Hint 1. You don’t need to focus on what’s going on the in `encode` and `decode` functions (they use some syntax that may not be familiar). All you need to know about them is that they
convert strings into integers and back. Focus on what happens in the other two functions.

*Hint 2.* You may find that you can’t think of a completely infallible way to break this encryption. That is OK; in this problem (and in the real world) an attack that succeeds with high probability is good enough, and such attacks exist in this case.

```python
# Encodes a given string as an integer.
def encode(text):
    code = 0
    for (loc, ch) in enumerate(text):
        code += ord(ch) * (1 << loc*8)
    return code

# Decodes an integer to a string.
def decode(code):
    text = ''
    while code > 0:
        byte = code & 0xFF
        text += chr(byte)
        code >>= 8
    return text

# Enciphers a given string (text) using a secret key (a positive integer).
def encipher(text, key):
    return key * encode(text)

# Deciphers a given integer (cipher) to return the original string.
def decipher(cipher, key):
    return decode(cipher / key)
```

**Solution.**

The encryption can be broken by extracting the secret key from the ciphertexts. The key observation is that each cipher text is of the form \(\text{key} \cdot \text{encode(text)}\), hence they all share a common factor of \(\text{key}\). It is possible that they share an even larger common factor: specifically, the gcd of all six ciphertexts will be equal to \(\text{key}\) times the gcd of the six encoded plaintexts. However, it turns out that it is extremely unlikely that six reasonably well-distributed numbers have a common factor, as you might suspect after doing problem 1 on this set (if the integers are large enough, the probability that all six share a common factor is about \(1/50\); the exact value is \(1 - 945/\pi^6\)).

Therefore, with a small chance of error, we will extract the key and hence break the encryption by simply taking the gcd of the six ciphertexts. You can use your code from a problem on the previous problem set to do this. Here is an example implementation (you should also include Alice and Bob’s original source; I have omitted here for clarity).

```python
from fractions import gcd

# Extract the key and decipher a list of ciphertexts
```
```python
def analyze(ciph):
    # Find the gcd of all ciphertexts
    # Start with the first, then take gcd with each subsequent number
    g = ciph[0]
    for n in ciph:
        g = gcd(g,n)
    key = g
    return [decipher(n,key) for n in ciph]

# Read the input
ciph = map(int,raw_input().split())

# Print the output
for plain in analyze(ciph): print plain
```

*Note.* A more concise and “pythonic” syntax to compute `key` would be

```
key = reduce(gcd,ciph,0).
```

*Note.* If you want a stronger algorithm that could work with fewer than 6 plaintexts, and have a reduced probability of error in any case, you could compute the gcd and then successively try integer multiples of it. This would require some algorithm to tell whether the resulting deciphered texts “look like” valid plaintext or not, which would depend somewhat on what sort of data they are (e.g. if they are English text, frequency analysis would likely be effective).