Written problems

1. The following passage has been encrypted using the Caesar cipher.

Nzky kyv yvcg fw kyv arezkfi yv jtivnuv fekf kyv
jzuv fw kyv uvjb r gvetzc jyrigvevi -- kyrk yzxycp
jrkzjwpzex, yzxycp gyzcjfgyztrc zdgcvdvek kyrk xfvj
kztfeuvifxr-kztfeuvifxr, wvvuzex fe kyv pvccfn wzezjy reu
jnvk nffu, reu veuj lg ze r bzeu fw jfleucvjjcp jgzeezex
vkyrivrc mfzu rj nv rcc dljk.

Determine the secret key $k$ (the number of places each letter has been advanced in the alphabet). You can solve this by hand, or by writing code to make the task easier. Either way, briefly summarize your strategy, including a description of any code that you write.

Solution.

The following bit of code implements the Caesar cipher.

```python
# Rotate a character ch n places, preserving the case.
# If ch isn't a letter, then this function returns it unchanged.
def rotate_char(ch,n):
    if not(ch.isalpha()): return ch #If ch isn't a letter, don't change it.
    #Calculate the position in the alphabet (A is 0, B is 1, etc.)
    loc = ord(ch.upper()) - ord('A')
    #Advance loc by n places, wrapping if necessary
    loc = (loc + n) % 26
    #Return the new letter, in the appropriate case
    if ch.isupper(): return chr( ord('A') + loc )
    else: return chr( ord('a') + loc )

# Rotate an entire string by applying rotate_char to each letter
def rotate(line,n):
    res = ''
    for c in line:
        res += rotate_char(c,n)
    return res

# NOTE: the more "pythonic" one-liner for this function is:
# return ''.join(rotate_char(ch,n) for ch in line)
```

Now, to quickly print all possible decipherings of this passage, you can first write the original passage to a string $s$ and write a loop as follows (output shown, truncated after 20 characters to save space). Note that I rotate $s - k$ places since I am deciphering, rather than enciphering, it.

```python
>>> for k in range(26):
...    print n, rotate(s,-k)[:60], "...
...
```
With the help of the janitor he screwed onto the side of the desk a pencil sharpener – that highly satisfying, highly philosophical implement that goes ticonderoga-ticonderoga, feeding on the yellow finish and sweet wood, and ends up in a kind of soundlessly spinning ethereal void as we all must.

The solution to problem 10 explains how you can write a program to find the secret key without needing to inspect the possibilities yourself.

2. Throughout this course, we will say that an integer \( a \) is an “\( n \)-bit (nonnegative) integer” if \( 0 \leq a < 2^n \) (I will also sometimes use the phrase “exactly \( n \) bits long” to mean that \( 2^n - 1 \leq a < 2^n \), i.e. that \( a \) is an \( n \)-bit integer but not an \((n-1)\)-bit integer).

The Data Encryption Standard (DES) is a private-key encryption algorithm that was a government standard from 1977 to 2002. DES uses 56-bit secret keys. Suppose that Eve attempts a brute-force attack on DES by trying to decrypt an intercepted cipher text with every possible 56-bit key until she finds something that looks like English text. If Eve’s system can try one billion keys per second, how long would it take her to try all of the keys (and thus be sure to break the encryption)?

(By 1999, a distributed system was able to break DES encryption in less than 24 hours. DES was replaced in 2002 by a new standard, called AES, which uses keys of at least 128 bits. For
“top secret” communication, the government uses AES with 256 bit keys.)

**Solution.** Eve requires $2^{56}/10^9$ seconds to try all possible keys. This comes out to

$$2^{56} \cdot \frac{1}{10^9} \text{sec} \cdot \frac{1}{3600} \text{sec} \cdot \frac{1}{24} \text{hour} \cdot \frac{1}{365.25} \text{day} \approx 2.28 \text{ years}$$

This is a while, but short enough that you can see why it was necessary to move to standard
with longer keys.


**Solution.**

(a) If $a \mid b$ and $b \mid c$, then by definition there exist $k, h \in \mathbb{Z}$ such that $b = ka$ and $c = hb$. It follows that $c = (hk)a$, so $a \mid c$.

(b) If $a \mid b$ and $b \mid a$, then there exist integers $k, h$ such that $a = kb$ and $b = ha$. Therefore $b = (hk)b$. Therefore either $b = 0$ or $kh = 1$. If $b = 0$ then $a = kb = 0$ as well, so $a = b$. On the other hand, if $kh = 1$, then both $k$ and $h$ must be $\pm 1$, since these are the only factors of 1. Hence $a = kb = \pm b$.

(c) If $a \mid b$ and $a \mid c$, then there exists integers $k, h$ such that $b = ka$ and $c = ha$. Therefore $b \pm c = (k \pm h)a$. Since $k \pm h$ is an integer, this shows that $a \mid (b \pm c)$.

4. Textbook exercise 1.9, parts (a) and (b).

**Solution.** The following lists of steps were generated automatically using the Euclidean algorithm.

(a)

\[
\begin{align*}
gcd(291, 252) &= gcd(252, 39) \quad (\text{since } 291 - (1) \cdot 252 = 39) \\
                 &= gcd(39, 18) \quad (\text{since } 252 - (6) \cdot 39 = 18) \\
                 &= gcd(18, 3) \quad (\text{since } 39 - (2) \cdot 18 = 3) \\
                 &= gcd(3, 0) \quad (\text{since } 18 - (6) \cdot 3 = 0) \\
                 &= 3
\end{align*}
\]

(b)

\[
\begin{align*}
gcd(16261, 85652) &= gcd(85652, 16261) \quad (\text{since } 16261 - (0) \cdot 85652 = 16261) \\
                   &= gcd(16261, 4347) \quad (\text{since } 85652 - (5) \cdot 16261 = 4347) \\
                   &= gcd(4347, 3220) \quad (\text{since } 16261 - (3) \cdot 4347 = 3220) \\
                   &= gcd(3220, 1127) \quad (\text{since } 4347 - (1) \cdot 3220 = 1127) \\
                   &= gcd(1127, 966) \quad (\text{since } 3220 - (2) \cdot 1127 = 966) \\
                   &= gcd(966, 161) \quad (\text{since } 1127 - (1) \cdot 966 = 161) \\
                   &= gcd(161, 0) \quad (\text{since } 966 - (6) \cdot 161 = 0) \\
                   &= 161
\end{align*}
\]

5. Textbook exercise 1.10, parts (a) and (b).
Solution. Keeping track of the linear combinations giving rise to each intermediate number from the previous problem, we obtain the following (again, this output has been automatically generated).

(a)  
\begin{align*}
291 &= 1 \cdot 291 + 0 \cdot 252 \\
252 &= 0 \cdot 291 + 1 \cdot 252 \\
39 &= 1 \cdot 291 - 1 \cdot 252 \\
18 &= -6 \cdot 291 + 7 \cdot 252 \\
3 &= 13 \cdot 291 - 15 \cdot 252 \\
0 &= -84 \cdot 291 + 97 \cdot 252
\end{align*}

So one possible answer is (13,-15).

(b)  
\begin{align*}
16261 &= 1 \cdot 16261 + 0 \cdot 85652 \\
85652 &= 0 \cdot 16261 + 1 \cdot 85652 \\
16261 &= 1 \cdot 16261 + 0 \cdot 85652 \\
4347 &= -5 \cdot 16261 + 1 \cdot 85652 \\
3220 &= 16 \cdot 16261 - 3 \cdot 85652 \\
1127 &= -21 \cdot 16261 + 4 \cdot 85652 \\
966 &= 58 \cdot 16261 - 11 \cdot 85652 \\
161 &= -79 \cdot 16261 + 15 \cdot 85652 \\
0 &= 532 \cdot 16261 - 101 \cdot 85652
\end{align*}

So one possible answer is (-79,15). You may notice an odd artifact at the beginning of these steps: the third line is identical to the first. This is because the smaller of the two numbers 16261, 85652 was listed first, so the first thing the algorithm did (in effect) was swap the two values. You also see this occurring in the previous problem.

6. Textbook exercise 1.11.

Solution.

(a) Suppose that $au + bv = 1$, where $a, u, b, v$ are all integers. Let $g$ be the greatest common divisor of $a$ and $b$. Then we can write

$$1 = g \cdot \left( \frac{a}{g}u + \frac{b}{g}v \right).$$

Since $a/g$ and $b/g$ are both integers, it follows that 1 is equal to $g$ times an integer, i.e. $g|1$. The only positive integer than divides 1 is 1 itself, hence $g = 1$, as desired.

(b) No, it is not necessarily true. For example, if $a = 3$ and $b = 2$, then we can take $u = 6, v = -6$ and obtain $au + bv = 6$, even though gcd(3, 2) = 1.

What we can say for sure, following the logic of the previous problem, is that if $au + bv = 6$, then

$$6 = g \cdot \left( \frac{a}{g}u + \frac{b}{g}v \right),$$

where $g = \text{gcd}(a, b)$ as before. Therefore certainly $g|6$. In fact, this is the the most that we can say, as the converse is true: if gcd$(a, b)$ divides 6, then 6 can be written $au + bv$. 

due Thursday 15 September, at 4:30pm.
To see this, simply observe that we can first write \( g = au' + bv' \), and multiply both sides by \( 6/g \) (an integer) to obtain 6 as a combination of \( a \) and \( b \).

Therefore the possible values of \( \gcd(a, b) \), given that 6 = \( au + bv \) for some \( u \) and \( v \), are 1, 2, 3, and 6.

(c) Given two such solutions, it follows that

\[
au_1 + bv_1 = au_2 + bv_2,
\]
\[
a(u_1 - u_2) = b(v_2 - v_1).
\]

Therefore \( a \mid b(v_2 - v_1) \). By a proposition proved in class, if \( a \mid bc \) and \( \gcd(a, b) = 1 \), then \( a \mid c \). In this situation, this proposition implies that \( a \mid (v_2 - v_1) \). A similar argument (reversing the roles of \( a \) and \( b \)) shows that \( b \mid (u_1 - u_2) \).

(d) As in part (c), it follows that if \( (u_1, v_1), (u_2, v_2) \) are two solutions to \( au + bv = g \), then \( a(u_1 - u_2) = b(v_2 - v_1) \). Let \( g = \gcd(a, b) \). A stronger form of the proposition proved in class states that if \( a \mid bc \) and \( \gcd(a, b) = g \), then \( \frac{a}{g} \mid c \). In this situation, this implies that \( \frac{a}{g} \mid (v_2 - v_1) \). In other words, there exists an integer \( k \) such that \( v_2 - v_1 = k\frac{a}{g} \), i.e. \( v_2 = v_1 + ka/g \). Solving the equation \( a(u_1 - u_2) = b(v_2 - v_1) \) for \( u_2 \) shows that, for this same integer \( k \), we have \( u_2 = u_1 - kb/g \).

Therefore if \( (u_1, v_1) \) is any one solution, then every other solution must have the form \( (u_1 - kb/g, v_1 + ka/g) \) for some integer \( k \), which is equivalent to the statement in the book.

7. This problem is meant to allow you to think about how the sizes of an input to a program influence its runtime, in a concrete setting. The purpose is for you to try to make some educated guesses for now; you do not need to be correct to receive full points.

(a) Consider the following function. It takes a positive integer \( N \), and returns the number of divisors of \( N \).

```python
def divisors(N):
    count = 0
    for d in xrange(1,N+1):
        if N%d == 0:
            count += 1
    return count
```

To be marked as correct on hackerrank, a program must finish within 10 seconds. Try to guess (roughly) how long \( N \) can be (in bits) before the function above will not finish in 10 seconds. One useful fact to keep in mind: a modern CPU completes approximately \( 2^{31} \) clock cycles per second.

\textit{Note.} Using \texttt{xrange} instead of \texttt{range} has essentially the same functionality and prevents certain errors. Search for “xrange versus range” (or similar) to read more about this.

(b) Test your guess by going to the “count divisors” challenge at the following link. The test cases are numbers 0 through 99, where test case \( k \) is an integer exactly \( k + 1 \) bits long. Therefore the first test case to fail will tell you the point at which this function no longer can finish in ten seconds.

\texttt{https://www.hackerrank.com/m158-2016-demos}
(c) Now consider the following function. Briefly explain why this also correctly calculates the number of divisors of $N$.

```python
def divisors(N):
    count = 0
d = 1
    while d*d < N:
        if N%d == 0:
            count += 2
        d += 1
    if d*d == N:
        count += 1
    return count
```

(d) Predict how many bits long $N$ must be before the code in part (c) cannot finish in less than 10 seconds. Test your prediction by entering this code and submitting it on hackerrank.

**Solution.**

There are various ways to go about this sort of estimation. The following is an example.

(a) The loop in this function will run $N$ times, and performs 2 instructions in each loop: computing $N%d$ and (possibly) incrementing `count`. In fact, it must do at least one other instruction: setting $d$ to a new value at the beginning of each time through the loop. If we assume that each time through the loop takes $k$ clock cycles on average, we can assume that $k$ is relatively small. The exact value of $k$ depends on a number of factors. So the function will require about $Nk$ clock cycles (plus some small number of cycles for the setup and cleanup before and after the loop, which we can probably ignore since it only happens once).

So to finish in time when $N$ is $\ell$ bits long, we need $2^\ell k \leq 10 \cdot 2^{31} \approx 2^{34}$, so roughly speaking this will work if $\ell \leq 34 - \log_2(k)$. If $k = 1$ (somehow each iteration of the loop takes only a single clock cycle), we'd get up to 34 bits, while if $k = 1000$ (i.e. if each operation takes a couple hundred clock cycles), we'd get up to about 24 bits (since $\log_2(1000) \approx 10$).

It’s hard to guess the actual value of $k$ without some more experience with Python, but tentatively we can say that this program will probably make it to somewhere between 24 and 34 bit integers.

(b) In fact, this program succeeds as long as $N$ is at most 27 bits long. So our estimate was in the right ballpark. In fact, this experiment tells us that the value of $k$ is probably about $2^7 = 128$. So each iteration of the loop probably takes about 100 clock cycles.

(c) Examining this function, we see that it counts the number of divisors less than $\sqrt{N}$, adding 2 to the total for each one. It also adds 1 if $\sqrt{N}$ itself is an integer. This counts divisors successfully because every divisor $d$ greater than $\sqrt{N}$ corresponds to a *unique* divisors less than $\sqrt{N}$, namely $N/d$. So by counting divisors less than $\sqrt{N}$ twice and ignoring all divisors greater than $\sqrt{N}$, we will get the correct total.

(d) Everything except the while loop only happens once, so it is unlikely to contribute much to the runtime. The loop runs $\sqrt{N}$ times (rounded down), and each iteration is three or four instructions. So it probably takes a similar number $k$ of clock cycles per iteration.
as the loop in part (a). Based on what we saw in part (d), we might guess that this will succeed as long as \( k\sqrt{N} \leq 2^{34} \). If \( N \approx 2^\ell \), this is equivalent to \( \ell \leq 64 - 2 \log_2(k) \).

In other words, this function will probably succeed for integers roughly twice as long as that for which the code in (a) succeeds. Since the code in (a) succeeded up to 27 bits, we might guess that this will succeed up to 54 bits.

The code in part (a) succeeds as long as \( N \) is less than 27 bits long. The code in part (c) succeeds as long as \( N \) is less than 53 bits long.

**Programming problems**

Full specifications and online submission: [http://www.hackerrank.com/m158-2016-pset-1](http://www.hackerrank.com/m158-2016-pset-1)

8. Write a program which receives an integer \( N \) as input, guaranteed to be a product of two 16-bit prime numbers, and determines the two prime factors.

**Solution.**

The simplest approach is trial and error: starting with 2, look for numbers that are a factor of \( N \). Once a single factor \( p \) is found, it must be the smallest prime factor, so \( N/p \) is the other. Here is an implementation.

```python
N = int(raw_input())

def firstprime(n):
    p = 2
    while N%p != 0:
        p += 1
    return p

def prime_factors(N):
    p = firstprime(N)
    return p, N/p

N = int(N)

if N < 2**16:
    factors = prime_factors(N)
else:
    factors = (1, N)

print factors
```

**Remark.** You may have encountered a bug on larger input if you wrote this using a `for` loop, e.g. `for p in range(2,N):`. The reason is that in Python 2, `range` must create a list with all the numbers to be iterated over all at once, which will be too large to fit in memory. One way to fix this is to use `xrange` instead, which does not allocate the entire list in memory at once. Either way, you should make sure to break the loop once a single prime factor is found, since this means you only need to find the smaller prime factor (which will be less than \( \sqrt{N} \)), rather than iterating \( N \) times. As we saw in problem 7, the difference between \( N \) iterations and \( \sqrt{N} \) iterations is about a factor of 2 in terms of how many bits long the input can be.

9. Write a program which takes a list of 1024-bit positive integers (i.e. each integer \( a \) in the list satisfies \( 1 \leq a < 2^{1024} \)) as input, and prints their greatest common divisor.

**Suggestion.** Begin with a function that given the greatest common divisor of any two numbers, then figure out how to use this function to find the GCD of a longer list.

**Solution.** First implement the `gcd` function using the Euclidean algorithm. Then, we can begin the first element in the list and successively “shrink” it by replacing it with its greatest common divisor with each subsequence entry. Here’s one implementation.

```python
from math import gcd

def gcd_list(L):
    if len(L) == 1:
        return L[0]
    else:
        return gcd(gcd_list(L[1:]), L[0])

L = [int(a) for a in input().split()]

print(gcd_list(L))
```
```python
def gcd(a, b):
    while a != 0:
        a, b = b % a, a
    return b

def gcd_list(ls):
    res = ls[0]
    for n in ls:
        res = gcd(res, n)
    return res

ls = map(int, raw_input().split())
print gcd_list(ls)
```

Note that there is also a very “pythonic” one-liner that can also be used to implement `gcd_list`, as follows. It uses the fact that \( \text{gcd}(n, 0) = n \) for all \( n \neq 0 \).

```python
def gcd_list(ls):
    return reduce(gcd, ls, 0)
```

10. *(Extra credit).* Write a program which can solve problem 1 of this assignment automatically (i.e. decrypt a passage of English text encrypted with a Caesar cipher). (Contact me to receive a hint, after you think about what sorts of methods might work).

**Solution.** There are a number of ways to approach this problem. Using the code from problem 1, it is easy to generate the 26 “candidates” for the plaintext; the hard part is determining which one is in English.

Students in the class found several different approaches to this; you should browse the submissions to see what sort of things were done. I’ll describe one method here.

One easy-to-code approach is to use frequency analysis. The basic idea is that the 26 letters do not occur equally often in English: some are seen more often than others. On page 6 of the textbook you can find the frequency of each of the 26 letters. You can similarly form the list of frequencies of all 26 letters in a candidate text, and attempt to measure how closely these frequencies mirror those of English. A simple way to quantify them is to regard both frequency lists as vectors with 26 coordinates and taking their dot product (if you then divide by the magnitude of the two vectors, you will obtain the cosine of the angle between them, i.e. the correlation coefficient of the two frequency lists; since the magnitudes of the vectors are the same for all 26 candidates, we can dispense with this last step and just compare the dot products). If implemented, this approach is good enough to break the Caesar cipher in all of the test cases online.

A slightly more theoretically-grounded approach (that is more robust for use in other applications) is to compute a “likelihood score” for each candidate text. This score is computed as follows: if \( p_i \) denotes the probability that a randomly chosen letter from an English text is the \( i \)th letter of the alphabet (so \( p_0 \) is the probability of \( a \), \( p_1 \) is the probability of \( b \), and so forth), and the letters of a text correspond to indices \( i_0, i_1, i_2, \ldots, i_{l-1} \), then the likelihood of this string is defined to be the product

\[
p_{i_0}p_{i_1}\cdots p_{i_{l-1}}.
\]
This product tells you the probability that you would obtain this particular sequence of letters if you were to choose the letters randomly from the “English letter distribution.” It should be higher for English passages than non-English passages. Since this number will grow very small very quickly, it is better in practice to compute its logarithm, the log-likelihood.

$$\log p_i_0 + \log p_i_1 + \cdots + \log p_i_{l-1}$$

If you compute the log-likelihood of each of the 26 candidate decipherings, you will generally find that one of these “scores” beats the others by a long shot; this will be the English plaintext. Here is an implementation (note that this also uses the code written in problem 1, which I will not reproduce here). I use a structure called a dict to record the letter frequencies for later computation; you could just as easily use an array, and access it by first computing the index of a letter in English (from 0 to 25).

```python
import math

# A dictionary of letter frequencies, copied from p. 6 of the textbook.
freq = {
    'A': 0.0815, 'B': 0.0144, 'C': 0.0276, 'D': 0.0379,
    'E': 0.1311, 'F': 0.0292, 'G': 0.0199, 'H': 0.0526,
    'I': 0.0635, 'J': 0.0013, 'K': 0.0042, 'L': 0.0339,
    'M': 0.0254, 'N': 0.0710, 'O': 0.0800, 'P': 0.0198,
    'Q': 0.0012, 'R': 0.0683, 'S': 0.0610, 'T': 0.1047,
    'U': 0.0246, 'V': 0.0092, 'W': 0.0154, 'X': 0.0017,
    'Y': 0.0198, 'Z': 0.0008
}

def freq_score(line):
    res = 0
    for char in line:
        char = char.upper()
        if char in freq:
            res += math.log(freq[char])
    return res

def decrypt(line):
    # Initially, best score is minus infinity
    # (to ensure that any score will appear better).
    best_score = float("-inf")
    # Try each of 26 rotations, and remember the best frequency score.
    for n in range(26):
        rot_line = rotate(line,n)
        score = freq_score(rot_line)
        if score > best_score:
            best_score = score
            best_rot = rot_line
```

due Thursday 15 September, at 4:30pm.
return best_rot

# I/O code for online submission
line = raw_input()
print decrypt(line)