CISS451: Cryptography and Computer Security
Assignment 2

Objectives

- Write Vigenere cipher.
- Perform frequency analysis.
- Break Vigenere ciphertext.

Module

You should collect all the useful code into a file. Let's call it `crypto.py`. This should be included in each directory of an answer to a question.

```
// File  : crypto.py
// Author: smaug

import string

VERBOSITY = 0

def cleanup(cs):
    cs = cs.lower()
    xs = []
    for c in cs:
        if c in string.ascii_lowercase:
            xs.append(c)
    return ''.join(xs)

def to_int(c):
    if not isinstance(c, str): raise ValueError("%s is not str" % c)
    if len(c) != 1: raise ValueError("length of %s is not 1" % c)
    if not('a' <= c <= 'z'): raise ValueError("char %s not in a..z" % c)
    return ord(c) - ord('a')

def to_chr(i):
    if not isinstance(i, int): raise ValueError("%s is not int" % i)
    if i < 0 or i > 25: raise ValueError("int %s not in 0..25" % i)
    return chr(i + ord('a'))

def to_ints(cs):
    ints = []
    for c in cs:
        ints.append(to_int(c))
    return ints

def to_chrs(xs):
    chrs = []
    for x in xs:
        chrs.append(to_chr(x))
    return chrs

# etc.
```

The VERBOSITY will be used to indicate the amount of output to produce while running a function. This is for tracing an algorithm. For instance for a brute force computation of multiplicative inverse:

```
// File  : crypto.py
// Author: smaug

import string
VERBOSITY = 1

# etc.

def inv(a, n):
    """ Returns the multiplicative inverse of a mod n using brute force. """
    for x in range(1, n):
        if VERBOSITY == 1:
            print("inv(%s, %s): testing %s" % (a, n, x))
        if a * x % n == 1:
            if VERBOSITY == 1:
                print("inv(%s, %s): returning %s" % (a, n, x))
            return x
    if VERBOSITY == 1:
        print("inv(%s, %s): returning None" % (a, n))
    return None

# etc.
```

In your bash shell run python3 and if you do this:

```
>>> from crypto import *
>>> VERBOSE = 0
>>> x = inv(3, 26)
inv(3, 26): testing 1
inv(3, 26): testing 2
inv(3, 26): testing 3
inv(3, 26): testing 4
inv(3, 26): testing 5
inv(3, 26): testing 6
inv(3, 26): testing 7
inv(3, 26): testing 8
inv(3, 26): testing 9
inv(3, 26): returning 9
>>> VERBOSITY = 0
>>> x = inv(3, 26)
```

```
>>>
```

Get it?

For the assignment questions, make sure each directory has the `crypto.py` and `main.py` that answers the question. And `main.py` should look like this:

```
// File  : main.py
// Author: smaug

from crypto import *

// answer to question using cleanup(), to_ints(), various
// encription/decryption functions, etc.
```

Here's another way to do it:

```
// File  : main.py
// Author: smaug

import crypto

// answer to question using crypto.cleanup(), crypto.to_ints(), etc.
```

More on printing

You have already seen the print statement:

```
x = 1
y = 2
print(x, y, "buckle my shoe")
```

Note that you have a single space between 1 and 2. There are times when you want to have more control over the printing. For instance say I do not want the spaces between the `1` and `2` and between `2` and `"buckle my shoe"`. You can of course create a string according to what you want. The following returns two strings (from integer values) and join then together:

```
x = 1
y = 2
s = str(x) + str(y)
s += "buckle my shoe"
print(s)
```

Another way to create string from integer, float, bool, char, string values using string formatting:

```
x = 1
y = 2
s = "%s%s%s" % (x, y, "buckle my shoe")
print(s)
```

Here's another example:

```
day = 1
month = 5
year = 1980
t = 'date: %s/%s/%s' % (year, month, day)
print(t)
```

Make sure you run the above examples.

What if you want more control? Notice that the above print statement alway prints a newline (at the end). You can get print to omit the newline at the end by doing this:

```
day = 1
```

```
month = 5
year = 1980
t = 'date: %s/%s/%s' % (year, month, day)
print(t, end='')
```

Another way to print is to send a string to the standard output stream. Here's how you do it:

```
import sys

x = 1
y = 2
s = "%s%s%s" % (x, y, "buckle my shoe")
sys.stdout.write(s)
```

You'll see that in this case there's no newline. If you want a newline at the end, you would have to do this:

```
import sys # Do this only once at the top

x = 1
y = 2
s = "%s%s%s" % (x, y, "buckle my shoe\n")
sys.stdout.write(s)
```

Note that you can only send strings to `sys.stdout`. Try this (you'll get an error):

```
import sys

sys.stdout.write(42)
```

Pretty print

Create a directory a02 for this assignment and for each question create a directory. For instance for Q1, create directory a02q01 and put all your python files in this directory. Etc. For this assignment, each question has one file: `main.py`. (In the future, we will organize all the useful functions into separate python files.) You can put all the useful functions from a01 in each `main.py` for this assignment.

The following function is useful and can be added to your `crypto.py`. The function assumes that the string is made up of lowercase characters.

```
import sys

def printstr(s, columns=50, omit_int=False):
    print("     ", end='')
    for i in range(columns):
        if i % 10 == 0:
            sys.stdout.write("%s" % (i//10))
        else:
            sys.stdout.write(" ")
    if not omit_int:
        sys.stdout.write(" ")
        for i in range(columns):
            if i % 10 == 0:
                sys.stdout.write("%3s" % (i//10))
            else:
                sys.stdout.write("   ")
    print()
    print("     ", end='')
    for i in range(0, columns):
        sys.stdout.write("%s" % (i % 10))
    if not omit_int:
        sys.stdout.write(" ")
        for i in range(0, columns):
            sys.stdout.write("%3s" % (i % 10))
    print()
    def drawline():
        sys.stdout.write("    +" + "-" * columns + "+")
        if not omit_int:
            sys.stdout.write("---" * columns + "+")
        print()
    drawline()
    line = 0
    while s != "":
        t = s[:columns]
        t = t + " " * (columns - len(t)) + "|"
        sys.stdout.write("%4s|%s" % (line, t))
        if not omit_int:
            t = s[:columns]
            t = to_ints(t)
            sys.stdout.write("".join(["%3s" % _ for _ in t]))
            sys.stdout.write("   " * (columns - len(t)))
            sys.stdout.write("|")
        print()
        s = s[columns:]
        line += 1
    drawline()
```

Here's a test:

```
from crypto import *

print("test printstr ...")
s = """
Dear reader! it rests with you and me whether, in our two fields of action
similar things shall be or not. Let them be!
We shall sit with lighter bosoms on the hearth, to see the ashes
 of our fires turn grey and cold.
"""
s = cleanup(s)
printstr(s, 60, True)
```

The output of the test is

```
     0         1         2         3         4         5
     012345678901234567890123456789012345678901234567890123456789
    +------------------------------------------------------------
   0|dearreaderitrestswithyouandmewhetherinourtwofieldsofactionsi
   1|milarthingsshallbeornotletthembeweshallsitwithlighterbosomso
   2|nthehearthtoseetheashesofourfiresturngreyandcold
```

Make sure you also try `printstr(s, 30, True)`

Note that in the code you see `i//10`. In python3, `//` means integer division. You should try

`1/10`. You'll see that `/` is *floating point* division. (In my opinion, this was a big mistake in the design of python3.)

Here's another test with the last argument set to `False`:

```
printstr(s, 20, False)
```

The output is

```
   0         1            0                             1
   01234567890123456789   0  1  2  3  4  5  6  7  8  9  0  1  2  3  4  5  6  7  8  9
  +--------------------+------------------------------------------------------------+
 0|dearreaderitrestswit|  3  4  0 17 17  4  0  3  4 17  8 19 17  4 18 19 18 22  8 19|
 1|hyouandmewhetherinou|  7 24 14 20  0 13  3 12  4 22  7  4 19  7  4 17  8 13 14 20|
 2|rtwofieldsofactionsi| 17 19 22 14  5  8  4 11  3 18 14  5  0  2 19  8 14 13 18  8|
 3|milarthingsshallbeor| 12  8 11  0 17 19  7  8 13  6 18 18  7  0 11 11  1  4 14 17|
 4|notletthembeweshalls| 13 14 19 11  4 19 19  7  4 12  1  4 22  4 18  7  0 11 11 18|
 5|itwithlighterbosomso|  8 19 22  8 19  7 11  8  6  7 19  4 17  1 14 18 14 12 18 14|
 6|nthehearthtoseetheas| 13 19  7  4  7  4  0 17 19  7 19 14 18  4  4 19  7  4  0 18|
 7|hesofourfiresturngre|  7  4 18 14  5 14 20 17  5  8 17  4 18 19 20 17 13  6 17  4|
 8|yandcold            | 24  0 13  3  2 14 11  3                                    |
  +--------------------+------------------------------------------------------------+
```

Make sure you understand the above code and try out the test code to understand how to use the functions.

By the way, in the above, we're not using any long integers in Python. So the above can be easily translated into for instance C++ if you wish.

Python Programming: table lookup and statistical/probabilistic frequency analysis

In C++ you have the concept of arrays. If you have an array of size 10. An array is like a table lookup. For instance suppose `x` is an array of size 5 with the following values

```
+-------+-------+
| index | value |
+-------+-------+
|     0 |   100 |
|     1 |   -50 |
|     2 |    26 |
|     3 |   990 |
|     4 |    -1 |
+-------+-------+
```

You have already seen that Python has lists which are like arrays. There's actually another table lookup called the dictionary. Each entry in a dictionary is key-value pair or a row. The difference between dictionary and lists is the you access row using a key value instead of an index value. Run the following in Python:

```
freq = {} # empty dictionary
freq['a'] = 0
freq['b'] = 0
print(freq)

freq['a'] = freq['a'] + 1
freq['b'] = freq['b'] + 23
print(freq)
```

Here's how you think of the `freq` dictionary:

```
+-----+-------+
| key | value |
+-----+-------+
| 'a' |     1 |
| 'b' |    23 |
+-----+-------+
```

You can iterate through a dictionary if you have all the keys:

```
freq = {} # empty dictionary
freq['a'] = 0
freq['b'] = 0
```

```
print(freq)

freq['a'] = freq['a'] + 1
freq['b'] = freq['b'] + 23
print(freq)

for key in ['a', 'b']:
    print(freq[key])
```

You can also obtain the list of keys from the dictionary:

```
freq = {} # empty dictionary
freq['a'] = 0
freq['b'] = 0
print(freq)

freq['a'] = freq['a'] + 1
freq['b'] = freq['b'] + 23
print(freq)

for key in ['a', 'b']:
    print(freq[key])

for key in freq.keys():
    print(freq[key])
```

If you access a row using a non-existent key you get an error. For instance the last statement will get you into trouble:

```
freq = {} # empty dictionary
freq['a'] = 0
freq['b'] = 0
print(freq)

freq['a'] = freq['a'] + 1
freq['b'] = freq['b'] + 23
print(freq)

for key in ['a', 'b']:
    print(freq[key])

for key in freq.keys():
    print(freq[key])

print(freq['c'])
```

If you lookup a diciontionary with a key that is not available, you will get an error. Instead of getting error, you can get the dictionary to return a default value when the key is not available. For instance:

```
freq = {} # empty dictionary
freq['a'] = 0
freq['b'] = 0
print(freq)

freq['a'] = freq['a'] + 1
freq['b'] = freq['b'] + 23
print(freq)

for key in ['a', 'b']:
    print(freq[key])

for key in freq.keys():
    print(freq[key])

print(freq['c'])        # ERROR: 'c' is not a key
print(freq.get('c', 0)) # if 'c' is not a key, gimme 0
```

Dictionaries can be used for frequency analysis in ciphers. For instance this creates a frequency table:

```
s = "helloworld"

freq = {}
for c in s:
    print("processing", c)
    freq[c] = freq.get(c, 0) + 1

import string
print("char freq analysis of s ...")
for c in string.ascii_lowercase:
    print(c, freq.get(c, 0))
```

Here's an analysis of 2–grams

```
s = "tobeornottobethatisthequestion"

freq = {}
```

```
for i in range(len(s) - 1):
    key = s[i:i+2]
    print("processing", key)
    freq[key] = freq.get(key, 0) + 1

import string
print("2-gram freq analysis of s ...")
for key in freq.keys():
    print(key, freq[key])
```

The syntax

```
s[i:i+2]
```

returns a substring from string `s` beginning at index `i` and ending at `i+2`, i.e., this is a substring of length 2 starting at index `i`. You can also do this with lists. Here are some examples for you to try out:

```
xs = [11,22,33,44,55,66,77,88,99]
ys = xs[3:8]
print(ys)

xs = "hello world"
ys = xs[3:8]
```

You can also specify the amount of step:

```
xs = [11,22,33,44,55,66,77,88,99]
ys = xs[3:10:2] # every other character from index 3 to 10
print(ys)

xs = "hello world"
ys = xs[3:10:2]
```

You should also try `xs[3:10:3]` and `xs[3:10:4]`.

This will help in cutting up strings/lists into pieces when you do Vigenere.

Q1.

Write a function `Vigenere_E(k, s)` where `k` is a string, the key, and `s` is a string of lowercase letter to be encrypted and returns the ciphertext of `s` using the Vigenere encryption algorithm.

You are given the following message m:

Dear reader! It rests with you and me whether, in our two fields of action similar things shall be or not. Let them be! We shall sit with lighter bosoms on the hearth, to see the ashes of our fires turn grey and cold.

First do the following on your own (this is not graded).

1. Clean it up (i.e. change uppercase to lowercase, remove all characters which are not `a-z`.
2. Encrypt the message from 1. above using Vigenere with the key of

   ```
   times
   ```

3. Print the ciphertext from the above using the `printstr` function with `columns` parameter set to 50 and last parameter set to `True`, i.e. do not print the mod 26 integers. In other words if the ciphertext as a string is `s`, execute `printstr(s, 50, True)`.

Now do the following (this is graded): write a program `main.py` so that on executing the program it prints the Vigenere encryption of the above string using the above key (The output is of course a string of only lowercase letters.)

Q2. Write a function `Vigenere_D(k, s)` that performs the Vigenere decryption of `s` using key `k`.

(It's a good idea to test your `Vigenere_D` function with the ciphertext produced by `Vigenere_E` in Q1. You should get back the original plaintext in Q1.)

The following goal is to find the key used for the cipher that produced this ciphertext:

```
mpqzalqfsjaihmfzafvgetqhlhbtiobvpsotvpfwbvsxzxvqry
toqhagtascbvsgskmxikltksmmemwslczqgompfqmpuwafxdik
lqhiwgbdcslumruhcxhhhaemtegnizxafsgwetmkmturymwtme
lmxjobbtedeqyeybvmfdxkasdgmewobbtlalpmxkmqxpggizhs
vmdxsbvmmjhnqbztcexahvgtggpuqagxmvltzuwagorvgfmjgw
lauzwlcyqwkizhagxmvlyzaqwqkqwkbdqkwgbupamgrsjbbiek
mwnikxmzaamptedyiziqxbtelamiektbtsjhcslyxvfpwfizqs
wmfslamysvxtajlamfmexeqejrwrinxzkxzbvsefwxgxlbvsrg
fwdixtqflagizclaqzklaizpmvqrij
```

and to decrypt the ciphertext.

First break the above cipher to discover the key on your own. Second decrypt the above message.

When you done write a program `main.py` so that on executing the program, it prints the key used to produce the above ciphertext. For instance if the key is `hotdog` and the decrypted message is `to be or not to be`, then your program does this:

```
// Name: smaug
// File: main.py

print("hotdog")
print("to be or not to be")
```

To get the key, you can any method mentioned in class. For this question, you do not need to explain how you arrive at the key.

Q3.

Here's the ciphertext from Q2 again:

```
mpqzalqfsjaihmfzafvgetqhlhbtiobvpsotvpfwbvsxzxvqry
toqhagtascbvsgskmxikltksmmemwslczqgompfqmpuwafxdik
lqhiwgbdcslumruhcxhhhaemtegnizxafsgwetmkmturymwtme
lmxjobbtedeqyeybvmfdxkasdgmewobbtlalpmxkmqxpggizhs
vmdxsbvmmjhnqbztcexahvgtggpuqagxmvltzuwagorvgfmjgw
lauzwlcyqwkizhagxmvlyzaqwqkqwkbdqkwgbupamgrsjbbiek
mwnikxmzaamptedyiziqxbtelamiektbtsjhcslyxvfpwfizqs
wmfslamysvxtajlamfmexeqejrwrinxzkxzbvsefwxgxlbvsrg
fwdixtqflagizclaqzklaizpmvqrij
```

For this question we will use the I and M numbers mentioned in class (the Kasiski method.)

Write a program that does the following:

1. For $m = 1$, print `"m:1, I:"`, compute and print the I–value for the above string.

   (Recall from the notes that m is the number of substrings you get by subdividing the plaintext.)

2. For $m = 2$, print `"m:2, I:"`, compute the I–values for the two strings `mqa...` and `pzl...`, compute their average, and finally print the average.

3. For $m = 3$, print `"m:3, I:"`, compute the I–values for the three strings, compute the average I–values of the three strings, print the average. Print a newline.

4. Do the same for $m = 4$.

5. Do the same for $m = 5$.

6. Do the same for $m = 6$.

7. Print a newline

8. Print the m, I values but sorted. (See format below.)

At this point, the printout would look like this:

```
m:1, I:0.03
m:2, I:0.02
m:3, I:0.01
```

```
m:4, I:0.04
m:5, I:0.05
m:6, I:0.06

m:6, I:0.06
m:5, I:0.05
m:4, I:0.04
m:1, I:0.03
m:2, I:0.02
m:3, I:0.01
```

(This is just to show you the format of the output. The values are of course incorrect.)

Note that to sort by the I value, you save the m, I pairs of values as (I, m) (tuples) in a python list and then execute sort. Here's an example:

```
xs = [(0.1, 1), (0.5, 2), (0.3, 3), (0.8, 4), (0.7, 5)]
xs.sort(reverse=True) # sort in descending order

for (I, m) in xs:
    print("m:%s, I:%s" % (m, I))
```

I'll call the largest m value the "best m", the second largest m value the "second best m". (Remember that everything is probabilistic – the largest m value might not be the correct one. Remember that we're hoping that the best m (or maybe the second best m) is the length of the key.

1. Compute the M values for y_0, y_1 for shifts $g = 0, 1, 2, ..., 25$ where y_0 is the first substring of the orginal ciphertext, y_1 is the first substring of the orginal ciphertext, etc. Save the (M, g) values in a list, sort (in reverse order) and print the entry with the largest M value. The format of the output might look like this:

```
m:1, I:0.03
m:2, I:0.02
m:3, I:0.01
m:4, I:0.04
m:5, I:0.05
m:6, I:0.06

m:6, I:0.06
m:5, I:0.05
m:4, I:0.04
```

```
m:1, I:0.03
m:2, I:0.02
m:3, I:0.01

m:6
0,1: g:15, M:0.8
```

This assume that the best shift relative shift is $g = 15$. (This is just to show you the format of your output. The values are not correct).

2. Repeat the above for y_0, y_2. The format of the output might look like this:

```
m:1, I:0.03
m:2, I:0.02
m:3, I:0.01
m:4, I:0.04
m:5, I:0.05
m:6, I:0.06

m:6, I:0.06
m:5, I:0.05
m:4, I:0.04
m:1, I:0.03
m:2, I:0.02
m:3, I:0.01

m:6
0,1: g:15, M:0.08
0,2: g:4, M:0.07
```

3. Repeat the above again until you have the data for y_0, y_{m-1} where m is "the best m". For instance assume the best m is 6, the output might look like this:

```
m:1, I:0.03
m:2, I:0.02
m:3, I:0.01
m:4, I:0.04
m:5, I:0.05
m:6, I:0.06

m:6, I:0.06
m:5, I:0.05
```

```
m:4, I:0.04
m:1, I:0.03
m:2, I:0.02
m:3, I:0.01

m:6
0,1: g:15, M:0.08
0,2: g:4, M:0.07
0,3: g:6, M:0.06
0,4: g:8, M:0.09
0,5: g:0, M:0.07
```

4. Now your program write down $m - 1$ mod 26 equations from the previous step:

```
m:1, I:0.03
m:2, I:0.02
m:3, I:0.01
m:4, I:0.04
m:5, I:0.05
m:6, I:0.06

m:6, I:0.06
m:5, I:0.05
m:4, I:0.04
m:1, I:0.03
m:2, I:0.02
m:3, I:0.01

m:6
0,1: g:15, M:0.08
0,2: g:4, M:0.07
0,3: g:6, M:0.06
0,4: g:8, M:0.09
0,5: g:0, M:0.07

k1 = k0 - 15 = k0 + 11 (mod 26)
k2 = k0 - 4 = k0 + 22 (mod 26)
k3 = k0 - 6 = k0 + 20 (mod 26)
k4 = k0 - 8 = k0 + 18 (mod 26)
k5 = k0 - 0 = k0 + 0 (mod 26)

k = (k0, k0 + 11, k0 + 22, k0 + 20, k0 + 18, k0) (mod 26)
```

5. For $k_0 = 0, 1, 2, ..., 25$, your program now prints all the possible values for $k0$ (0, 1, 2,

..., 25) and the first decryption of the first 40 characters of the ciphertext:

```
m:1, I:0.03
m:2, I:0.02
m:3, I:0.01
m:4, I:0.04
m:5, I:0.05
m:6, I:0.06

m:6, I:0.06
m:5, I:0.05
m:4, I:0.04
m:1, I:0.03
m:2, I:0.02
m:3, I:0.01

m:6
0,1: g:15, M:0.08
0,2: g:4, M:0.07
0,3: g:6, M:0.06
0,4: g:8, M:0.09
0,5: g:0, M:0.07

k1 = k0 - 15 = k0 + 11 (mod 26)
k2 = k0 - 4 = k0 + 22 (mod 26)
k3 = k0 - 6 = k0 + 20 (mod 26)
k4 = k0 - 8 = k0 + 18 (mod 26)
k5 = k0 - 0 = k0 + 0 (mod 26)

k = (k0, k0 + 11, k0 + 22, k0 + 20, k0 + 18, k0) (mod 26)

k0 = 0: k = alwusa  p = sldlfksjdfsldkflswoefpwoqpslgkdflzfgshfk
k0 = 1: k = bmxvtb  p = oritueorjfoefoeighoiurghosjsoeijfspoepse
...
```

Don't worry if you can't find the key. Remember that everything is probabilistic. For a complete solution, look at the next few parts.

Complete solution – Optional

For a complete solution, you would need to know backtracking so that when there's a failure you backtrack and if there's a success, keep going. It's even better if your program have a way to check if a string is made up of meaningful words, i.e., your program can perform language recognition. That requires knowledge of the dynamic programming technique – see CISS358. (In fact the automatic cutting up of string into meaningful English words is a problem in my notes for CISS358. If you have taken CISS358, you might remember that problem.) This can be solved without dynamic programming too, but the asymptotic worst case runtime is exponential and by the time your program is done, earth is long gone, our sun will be a white dwarf, ... So for an efficient and completely automated algorithm to solve Vigenere, you have to know dynamic programming. The specific thing you need to do is a simple natural language processing step that inserts spaces into your decrypted ciphertext to create words:

`"tobeornottobethatisthequestion"` $\mapsto$ `"to be or not to be that is the question"`

For this algorithm, you'll also need a dictionary of English words.

OK ... for the those who want to be totally challenged ... keep going ...

Q4. [OPTIONAL]

Repeat the above, except that for the steps that compute the relative shifts, if an M value is too low (< 0.04), the process for the best m stops and moves to the next best m:

```
m:1, I:0.03
m:2, I:0.02
m:3, I:0.01
m:4, I:0.04
m:5, I:0.05
m:6, I:0.06

m:6, I:0.06
m:5, I:0.05
m:4, I:0.04
m:1, I:0.03
m:2, I:0.02
m:3, I:0.01

m:6
0,1: g:15, M:0.08
0,2: g:4, M:0.01
```

```
REJECT m = 6

m:5
0,1: g:2, M:0.07
...
```

The process stops if the I value for an m is < 0.04.

Q5. [OPTIONAL]

It's possible for an m, and for the pair y_0, y_1, there are *several* good relative shifts. Handle these cases as well.

```
m:1, I:0.03
m:2, I:0.02
m:3, I:0.01
m:4, I:0.04
m:5, I:0.05
m:6, I:0.06

m:6, I:0.06
m:5, I:0.05
m:4, I:0.04
m:1, I:0.03
m:2, I:0.02
m:3, I:0.01

m:6
0,1: g:15, M:0.08
0,2: g:4, M:0.07
0,2: g:8, M:0.08
...
```