Final Exam

Due Time and General Instructions:

This is a 72-hours take-home exam. It is available from Monday, December 9 at 9:00 am and is due on Thursday, December 12 at 9 am.

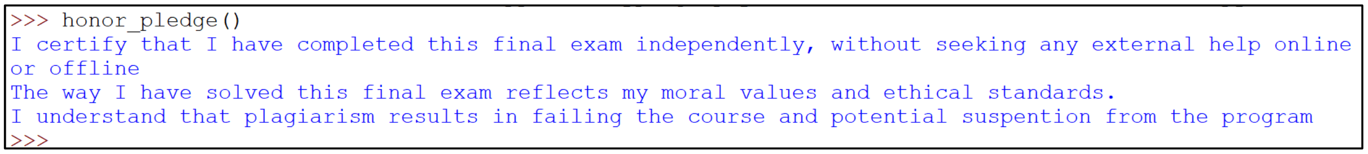
It is an open-book exam. However, seeking any external assistance is not allowed. This includes collaboration with classmates, using online forums and assistance from any other source. Any detected act of plagiarism will lead immediate failure in the course, in addition to other disciplinary matters.

Question Windows:

- Monday (December 9): 2:00-3:00 pm
- Tuesday (December 10): 2:00-3:00 pm
- Wednesday (December 11): 2:00-3:00 pm

Honor Pledge:

All students are required to type in the following statement under honor_pledge() function. Any submission without this will be rejected.



```
>>> honor_pledge()
I certify that I have completed this final exam independently, without seeking any external help online
or offline
The way I have solved this final exam reflects my moral values and ethical standards.
I understand that plagiarism results in failing the course and potential suspention from the program
>>>
```

Also, remember to edit your name and student ID on top of the designated files

Final Exam Files:

You need the following files for your final exam.

File name	Description	Action(s)	Restrictions
template.py	File containing your solution	rename this file to final.py	you can edit the provided functions but you cannot add any new functions

test_final.py	File used for testing your solution	None	Do not edit this file
utilities.py	File hosting utility functions	None	Do not edit this file
mod.py	Modular arithmetic file	Not provided. Import your own file	You may edit content, but not function signatures
SDES.py	Simple DES implementation	Not provided. Import your own file	You can edit as you like
matrix.py	Matrix Utilities	Not provided. Import your own file	You may edit content, but not function signatures
q4_template.py	Solution to Q4	rename this file to q4.py	You can edit as you like
Q3 Supporting Files	sbox1.txt sbox2.txt primes.txt	None	Do not edit
Q4 supporting files	q4A_ciphertext.txt q4B_ciphertext.txt q4C_ciphertext.txt	None	Do not edit
Q5 Supporting files	public_keys.txt	None	Do not edit
Others	engmix.txt	None	Do not edit

Final Exam Structure:

This final is composed of the following five questions.

1- Math Cipher
2- Myszkowski Cryptanalysis
3- SDES Modes
4- Double X
5- Public X

All questions are mandatory and each is 6 pts.

What to submit:

You need to submit ONE zip file <yourname_CP460final.zip> that contains the following 8 files:

1- final.py
2- q4.py
3- mod.py
4- matrix.py
5- SDES.py
6- 3 plaintext files: `q4A_plaintext.txt`, `q4B_plaintext.txt` and `q4C_plaintext.txt`

You do not need to include any other files.

You may not import external libraries other than the ones included.

It is critically important that running the testing file should not cause your program to crash. This means that even if you do not provide a solution, you still need to make sure that the testing file, when executed, does not crash. You can do this by including empty plaintext files, printing a dummy statement

If your program crashes for any reason, you immediately lose 4 points. If the instructor fails to fix the bug quickly, you will lose 3 more points.

NO EXTENSIONS WILL BE GIVEN. DO NOT BEG FOR ONE

Task 1: MathCipher (6 pts)

There are six functions to write for this task:

1- *isValidKey_mathCipher(key)* [0.5 pts]
2- *e_mathCipher(plaintext,key)* [0.75 pts]
3- *d_mathCipher(ciphertext,key)* [0.75 pts]
4- *analyze_mathCipher(baseString)* [1.5 pts]
5- *stats_mathCipher()* [1.25 pt]
6- *cryptanalysis_mathCipher(ciphertext)* [1.25 pts]

An algebraic cipher scheme called *mathCipher* produces ciphertexts through the following equation:

$$y = b(ax + b) - c$$

where the variables a, b and c are any integer.

The above cipher is neither a decimation cipher, nor an affine cipher, but has some characteristics of both. Indeed, in some scenarios, it may act as a decimation cipher or as an affine cipher.

The encryption and decryption functions: *e_mathCipher* and *d_mathCipher* receive two inputs parameters: plainatext/ciphertext and a key. The *key* is a tuple structured as (baseString, *k*). The baseString is a substring from the base string generated by the function get_baseString available at utilities. The minimum length of the baseString should be 2 characters. The *k* element of the key is a list structured as [a,b,c].

Since there are too many scenarios for invalid keys, write a function called: isValidKey_mathCipher that gets a key as an input parameter and returns True/ False. The function checks both that the key is passed in the correct format, i.e. tuple:str,[int,int,int], the baseString is at least 2 characters long and the given integers (some or all) are invertible at some mode *m*.

Below is the testing results for the validity check:

```
Testing Q1: MathCipher

Testing validity of keys:
['ab', [1, 1, 1]]  =  False
('', [1, 1, 1])  =  False
('a', [1, 1, 1])  =  False
(123, [1, 1, 1])  =  False
('ab', (1, 1, 1))  =  False
('ab', [1, 1])  =  False
('ab', [1, '1', 1])  =  False
('abcdefghijklmnopqrstuvwxyz', [6, 7, 11])  =  False
('abcdefghijklmnopqrstuvwxyz', [11, 13, 19])  =  False
('abcdefghijklmnopqrstuvwxyz', [11, 7, 13])  =  True
```

The encryption and decryption functions should call the above key checking function. If the key is deemed invalid, the following error message should be printed:

```
Error(e_mathCipher): invalid plaintext
Error(d_mathCipher): invalid ciphertext
```

Also, if the plaintext/ciphertext is non-empty strings:

```
Error(e_mathCipher): invalid plaintext
Error(d_mathCipher): invalid ciphertext
```

Testing the encryption and decryption functions will give the following:

```
Testing Encryption/Decryption:
key =  ('abcdefghijklmnopqrstuvwxyz', [5, 9, 11])
plaintext =   MATH CIPHER
ciphertext=   MSPV EORVQD
plaintext2=   MATH CIPHER

key =  ('abcdefghijklmnopqrstuvwxyz ?!', [45, 5, 16])
plaintext =   Is mission accomplished?
ciphertext=   L!bml!!l?fbjyy?mutl!skrx
plaintext2=   Is mission accomplished?

key =  ('abcdefghijklmnopqrstuvwxyz ?!#', [5, 5, 5])
plaintext =
Error(e_mathCipher): invalid plaintext
Error(d_mathCipher): invalid ciphertext
```

```
key =  ('abcdefghijklmnopqrstuvwxyz?!', [421, 421, 421])
plaintext =   The quick brown fox jumps over the lazy dog
ciphertext=   The quick brown fox jumps over the lazy dog
plaintext2=   The quick brown fox jumps over the lazy dog

key =  ('abcdefghijklmnopqrstuvwxyz#?!', [10, 3, 9])
plaintext =   The quick brown fox jumps over the lazy dog
ciphertext=   The quick brown fox jumps over the lazy dog
plaintext2=   The quick brown fox jumps over the lazy dog

key =  ('abcdefghijklmnopqrstuvwxyz#?!', [0, 1, 1])
plaintext =   This is Q1 of CP460 Final Exam
Error(e_mathCipher): Invalid key
Error(e_mathCipher): Invalid key
```

In order to cryptanalyze any given ciphertext, the key space should be analyzed. This is done through the function analyze_mathCipher() provides an estimation of the required work to brute-force through the entire key space. The function receives a baseString of an arbitrary length, of two or more characters. The objective is to find how many keys, i.e. different combinations of a, b and c, are needed to exhaust the entire space. The function returns the following list:

`[total, illegal, noCipher, decimation, valid]`

`total:` Size of the key space, i.e. how many keys theoretically could be generated.

`illegal:` Number of keys which are illegal, i.e. there is no modular multiplicative inverse such that decryption would not be possible.

`noCipher`: Number of keys which will result in no encipherment, i.e. plaintext will be equal to ciphertext

`decimation`: Number of keys which if used, the scheme will act exactly like a decimation cipher. Note that keys which are illegal or will result in no encipherment are not included in the calculation here.

`valid`: Number of keys which are legal and will result in some form of encipherment which is not identical to decimation cipher.

Testing the above will yield the following:

```
Testing analyze_mathCipher:
Analyze:  ab =  [8, 6, 1, 0, 1]
Analyze:  abc =  [27, 15, 2, 2, 8]
Analyze:  abcd =  [64, 48, 2, 2, 12]
Analyze:  abcde =  [125, 45, 4, 12, 64]
Analyze:  abcdefg =  [343, 91, 6, 30, 216]
Analyze:  abcdefghijklmnopqrstuvwxyz =  [17576, 13832, 12, 132, 3600]
Analyze:  abcdefghijklmnopqrstuvwxyz!#? =  [24389, 1653, 28, 756, 21952]
```

The stats_mathCipher function builds on the results obtained from analyze_mathCipher. The objective is to find out what is the percentage of valid keys against the total key space.

The function receives no parameters and return values. It only print statistical results.

Base strings from length 2 characters up to 100 characters, inclusive both ends, are to be examined. For each base length, the percentage of valid keys against the total key space is computed. Then the average of these values, i.e. across all base lengths, is computed along with the best case scenario and worst case scenario. The function provides statistics for prime lengths of base string, and no-prime lengths of base string. The statistics are to be displayed.

Below is the expected output:

```
Testing stats_mathCipher:
For all numbers:
    Average = 40.30%
    Best = 6.87%
    Worst = 96.94%
For Primes:
    Average = 83.07%
    Best = 12.50%
    Worst = 96.94%
For non Primes:
    Average = 25.85%
    Best = 6.87%
    Worst = 71.97%
```

From the above, you can see that if the base string's length is some value between 2 and 100, then on average 40.3% of the key space should be tested.

In the best scenario, only 6.87% would be needed, and at worst 96.94% is needed.

The final step, is to write a cryptanalysis function. The function receives a ciphertext and attempts to find the key. Base strings are assumed to be at least as long as the lower case alphabet, and at most the same length as the base string returned by get_baseString() in the utilities file. Note that only “valid keys” should be considered. Testing the function will give:

```
Testing cryptanalysis:
key found after  9 attempts
Key =  ('abcdefghijklmnopqrstuvwxyz', [1, 1, 9])
plaintext =  There are two kinds of cryptography in this world:
cryptography that will stop your kid sister from reading your files, and
cryptography that will stop major governments from reading your files.

ciphertext:  Uxcnoicostplqpcampcaptaugec ktptcxolcamutcdiptauog,cnoicyurrcxughctozpcmugat
key found after  3731 attempts
Key =  (HIDDEN, HIDDEN)
plaintext = <HIDDEN> some hints
```

Note that for the last cryptanalysis case, the recovered key is hidden. Only the last two words of the recovered plaintext are displayed.

Task 2: Myszkowski Cipher Cryptanalysis (6 pts)

The implementation of the Myszkowski cipher was conducted in Assignment 3. You wrote three functions: *`e_myszkowski`*, *`d_myszkowski`* and *`get_keyOrder_myszkowski`*. Refer to Assignment 3 PDF file for the specifications. These specifications apply here too.

Copy the above three functions into your final.py file. If you have missed this part in the assignment, you need to write these functions first before proceeding.

Automation of Myszkowski cipher is not a simple task. However, if we know specific facts about the ciphertext, the cryptanalysis process could be automated. In this task, you will consider three different scenarios which you need to write a cryptanalysis function for.

<u>Scenario 1: Brute Force, key length = 3 (2 pts):</u>

Assume that you know that the ciphertext was produced through a key which is composed of three alphabetical characters. This is the smallest key size possible for the Myszkowski cipher to work (you should know why!!).

Write a brute-force scheme to break such cipher. Consider carefully the number of keys that you will be using in your brute-force.

Assuming that you provide convincing description, and you successfully decrypt the given two messages, you will get the following points:

- If worst case is >= 16,000 keys → 0 points
- If worst case is (16,000, 2,000] keys → 1 point
- If worst case is (2,000,500] keys → 1.5 points
- If worst case is < 500 keys → 2 points

For this part, you need to write two functions:
`cryptanalysis1_myszkowski(ciphertext)` and `q2_description1().`

Running the testing function will give:

```
-------------------------------------------
Testing Q2: Myskowski Cryptanalysis

Scenario 1:
ciphertext =  Bd ims av ascenifc ale.Thsear ocasinsa oo larerwoldno
mssqatehe itivu e ecso gden u ti.
Starting Cryptanalysis:
key found after <HIDDEN> attempts
Key =  <HIDDEN>
plaintext =  <HIDDEN>

ciphertext =  omnoeemti nv o  g; nexre nreergN aout f xprienatonca
eerprvemeriht asigl epeimntca pov m won.
Starting Cryptanalysis:
key found after <HIDDEN> attempts
Key =  <HIDDEN>
plaintext =  <HIDDEN>

Cryptanalysis Strategy (Brute Force 3 characters):
Put your description here
Worst Case Scenario: X attempts
```

Scenario 2: Dictionary Attack (2 pts):

Assume that you know that the ciphertext was produced through a key which is one of the words in the dictionary file “engmix.txt”. Also, assume that you know the length of the key.

The cryptanalysis function should construct a list of words, read from the dictionary file, which match the given length and are valid Myszkowski keys. Then, it should brute-force through this list to find the proper key.

For this part, you need to write two functions: `cryptanalysis2_myszkowski(ciphertext,length)` and `q2_description2()`.

In the description function, briefly describe your method for filtering the words in the dictionary. Also, mention at what key length does this method result in the worst scenario? List both the key length and number of keys/attempts.

For this part, you will get 0.5 pts for each matching cryptanalysis result, and 0.5 for the description.

```
Scenario 2:
ciphertext =  cinc desno kowit dbttoimgiatonseeo tn se  ani
Starting Cryptanalysis:
key found after 12 attempts
Key =  baa
plaintext =  [redacted]

ciphertext =  e  ifa ornaonl tf cyo, psdesaic p eaiolb n inoa sc rosal oomcessn.eneuodstinn
Starting Cryptanalysis:
key found after [redacted] attempts
Key =  worthlessness
plaintext =  science is not only a disciple of reaosn but, also one of romance and passion.

ciphertext =  cuq ndohqot u asodchiean t cqqat uiq iyclqTsh
Starting Cryptanalysis:
key found after 678 attempts
Key =  [redacted]
plaintext =  Touch a scientist and you touch a child

Cryptanalysis Strategy (Dictionary Attack):
Put your description here
Worst Case Scenario: X attempts
```

Scenario 3: 5-Triple (2 pts):

Assume that you know that the ciphertext was produced through a key which is five characters long. Also, you know that one of the characters appear three times in the key.

Examples of keys that satisfy the above are: *cccde*, *yxxxz* and *rtsrr*.

First, study the brute-force scenario. Show how many keys, at most, you need to try.

Hints:

1- Focus only on one scenario which is when the character *c* appears 3 times in the string, and work through all possible scenarios.

2- The number of keys you need to try is some number below 100.

Second, construct a list of words from the dictionary that are valid Myszkowski keys, of length 5, and contain one character that is repeated 3 times. How many words does your list contain? Is that more or less than the brute-force space you have calculated above? Write your observations in the description function.

Use the method (brute-force or dictionary attack) with least keys to break through the given ciphertexts.

For this part, you need to write two functions: `Cryptanalysis3_myszkowski(ciphertext,length)` and `q2_description3()`.

You will get 1 point for correct output, and 1 point for valid description.

Running the testing function will give:

```
Scenario 3:
ciphertext =  irtTi:Dont atemchtmeo asngequsioqF  osuini tsp w    len
Starting Cryptanalysis:
key found after [redacted] attempts
Key =  geese
plaintext =  First Tip: Do not waste much time on a single question

ciphertext =  Sondip:avenou brks d sep ursqqe   gealhqcTHehaneoq
Starting Cryptanalysis:
key found after [redacted] attempts
Key =  fluff
plaintext =  Second Tip: Have enough breaks and sleep hours

ciphertext =  iitwtauoiqTrd p: artithhe sy estns rstqqhTS  eqifq
Starting Cryptanalysis:
key found after [redacted] attempts
Key =  emcee
plaintext =  Third Tip: Start with the easy questions first

-------------------------------------------
```

Task 3: Simple DES Modes
(6 pts)

In Assignment 5, you have worked on implementing a simplified version of DES. In this task, you will support the implementation with the three basic modes of operation: ECB, CBC and OFB.

Throughout this task, the encoding will be B6 code. However, the block-size, key-size and rounds may vary.

<u>Mode 1: Electronic Code Book (ECB) (2 pts):</u>

The implementation you wrote for A5 actually correspond to the ECB mode. In this part, you will do the following:

Step 1: Editing SDES.py

Copy your SDES.py file into the same folder as your final exam. There are import statements at the final.py and test_final.py files to include SDES. Rename:

- e_SDES(plaintext,key) to e_SDES_ECB(plaintext,key)
- d_SDES(ciphertext,key) to d_SDES_ECB(ciphertext,key)

In these functions, edit the error messages to reflect the new function name. You do not need to change the other functions as they will be the same for all modes.

Step 2: Generic functions:

In final.py, create two functions called: `e_SDES(plaintext,key,mode)` and `d_SDES(ciphertext,key,mode)`. These functions should act as generic functions that call the appropriate encryption/decryption functions. The supported modes should be ECB, CBC and OFB. An error message should printed for other modes. You will also need to error checking for validity of plaintext/ciphertext and key. It is good to catch errors before passing it to the other functions.

Step 3: Testing: Ensure that both ECB function and generic functions are working and produce correct results. Running the testing module will give:

```
-------------------------------------------
Testing Q3: SDES modes

Case 0:
    e_SDES(CP460 Final Exam,) = Error(e_SDES): undefined mode
    d_SDES(Cfpzch3mN4f0,) = Error(d_SDES): undefined mode

    e_SDES(CP460 Final Exam,111000111) = Error(e_SDES): Invalid key
    d_SDES(,111000111) = Error(d_SDES): Invalid input

Testing mode ECB:
case 1:
Configuration: [['rounds', '2'], ['key_size', '9'], ['block_size', '12'],
['encoding_type', 'B6']]
    e_SDES(xRxR,111000111) = b9b9
    d_SDES(b9b9,111000111) = xRxR

Case 2:
Configuration: [['rounds', '2'], ['key_size', '9'], ['block_size', '12'],
['encoding_type', 'B6'], ['p', '103'], ['q', '199']]
    e_SDES(DES is a legacy Cipher,111000111) = hB2AmSQDvS4X6SkMwmkLZp
    d_SDES(hB2AmSQDvS4X6SkMwmkLZp,111000111) = DES is a legacy Cipher

Case 3:
Configuration: [['rounds', '5'], ['key_size', '15'], ['block_size', '24'],
['encoding_type', 'B6'], ['p', '103'], ['q', '199']]

    e_SDES(speed car,111000111001010) = p0L9SWwSHSi0
    d_SDES(p0L9SWwSHSi0,111000111001010) = speed car

Case 4:
    e_SDES(AAAAAA,) = cZRWXHvn
    d_SDES(cZRWXHvn,) = AAAAAA
```

Mode 2: Cipher Block Chaining (CBC) (2 pts):

Implement the CBC mode for SDES. Place your implementation at the SDES file. Your new functions are: `e_SDES_CBC(plaintext,key)` and `d_SDES_CBC (ciphertext,key)`. You can assume that the plaintext/ciphertext and key are valid inputs.

Since, the CBC and OFB modes require the use of an initial vector (IV), a special function is needed for this. Write the function: `get_IV(block_size)` which creates an IV of the given size. The function calls the Blum random algorithm with the parameters p = 643 and q = 131. The number of bits shall be the same as the block size, which is defined in the configuration file.

Running the testing function will give:

```
Testing mode CBC:
case 2:
Configuration: [['rounds', '2'], ['key_size', '9'], ['block_size', '12'],
['encoding_type', 'B6']]
IV =  101111001101
    e_SDES(xRxR,111000111) = fY0R
    d_SDES(fY0R,111000111) = xRxR

Case 2:
Configuration: [['rounds', '2'], ['key_size', '9'], ['block_size', '12'],
['encoding_type', 'B6'], ['p', '103'], ['q', '199']]
IV =  101111001101
    e_SDES(DES is a legacy Cipher,111000111) = LgWXKRl66v5S9 G6j32W8O
    d_SDES(LgWXKRl66v5S9 G6j32W8O,111000111) = DES is a legacy Cipher

Case 3:
Configuration: [['rounds', '5'], ['key_size', '15'], ['block_size', '24'],
['encoding_type', 'B6'], ['p', '103'], ['q', '199']]

IV =  101111001101110010000110
    e_SDES(speed car,111000111001010) = lV4GZVlG9Uta
    d_SDES(lV4GZVlG9Uta,111000111001010) = speed car

Case 4:
    e_SDES(AAAAAA,) = RNw4ii6v
    d_SDES(RNw4ii6v,) = AAAAAA
```

Mode 3: Output Feedback Mode (2 pts):

Implement the OFB mode for SDES. Place your implementation at the SDES file. Your new functions are: `e_SDES_OFB(plaintext,key)` and `d_SDES_OFB` `(ciphertext,key)`. You can assume that the plaintext/ciphertext and key are valid inputs. Use the same IV generation method as in CBC.

Remember that in OFB mode, there is no padding. Therefore, the last block has to be handled differently.

Running the testing function will give:

```
Testing mode OFB:
case 3:
Configuration: [['rounds', '2'], ['key_size', '9'], ['block_size', '12'],
['encoding_type', 'B6']]
IV =  101111001101
    e_SDES(xRxR,111000111) = uzX1
    d_SDES(uzX1,111000111) = xRxR

Case 2:
Configuration: [['rounds', '2'], ['key_size', '9'], ['block_size', '12'],
['encoding_type', 'B6'], ['p', '103'], ['q', '199']]
IV =  101111001101
    e_SDES(DES is a legacy Cipher,111000111) = o IaLFWvQp8CYxVF34c
KI
    d_SDES(o IaLFWvQp8CYxVF34c
KI,111000111) = DES is a legacy Cipher

Case 3:
Configuration: [['rounds', '5'], ['key_size', '15'], ['block_size', '24'],
['encoding_type', 'B6'], ['p', '103'], ['q', '199']]

IV =  101111001101110010000110
    e_SDES(speed car,111000111001010) = v4s51zwWe
    d_SDES(v4s51zwWe,111000111001010) = speed car

Case 4:
    e_SDES(AAAAAA,) = lmmHUo
    d_SDES(lmmHUo,) = AAAAAA

-------------------------------------------- Testing mode OFB:
case 3:
Configuration: [['rounds', '2'], ['key_size', '9'], ['block_size', '12'], ['encoding_type', 'B6']]
IV =  101111001101
    e_SDES(xRxR,111000111) = uzX1
    d_SDES(uzX1,111000111) = xRxR
```

```

Case 2:
Configuration: [['rounds', '2'], ['key_size', '9'], ['block_size', '12'],
['encoding_type', 'B6'], ['p', '103'], ['q', '199']]
IV =  101111001101
    e_SDES(DES is a legacy Cipher,111000111) = o IaLFWvQp8CYxVF34c
KI
    d_SDES(o IaLFWvQp8CYxVF34c
KI,111000111) = DES is a legacy Cipher

Case 3:
Configuration: [['rounds', '5'], ['key_size', '15'], ['block_size', '24'],
['encoding_type', 'B6'], ['p', '103'], ['q', '199']]

IV =  101111001101110010000110
    e_SDES(speed car,111000111001010) = v4s51zwWe
    d_SDES(v4s51zwWe,111000111001010) = speed car

Case 4:
    e_SDES(AAAAAA,) = lmmHUo
    d_SDES(lmmHUo,) = AAAAAA

-------------------------------------------
```

Task 4: DoubleX Cipher (6 pts)

In this task, you are given three ciphertext files. Your job is to decrypt them.

Each file is encrypted using a double encryption, i.e. encrypting first using an encryption scheme and the output is encrypted using a second but different scheme.

The set of ciphers include five schemes:

1- Atbash Cipher
2- Shift Cipher
3- Columnar Transposition Cipher
4- Polybius Cipher
5- Hill Cipher

You do not know the order of how these schemes are used in encrypting each file. This leaves you with 20 possible combinations of the above. Your job is to find the two schemes used for each file, and the order they were applied to produce the ciphertext.

Below are some notes about each scheme:

Atbash Cipher: There is no key used in this scheme. The base string is the lower case alphabet. However, the case of letters should be persevered. All non-alpha characters are excluded from encryption/decryption.

Shift Cipher: The base string is the lower case alphabet. The key is a number between 0 and 25. The case of letters is preserved. All non-alpha characters are excluded from encryption/decryption.

Columnar Transposition: The encryption/decryption is applied to all characters including spaces and punctuations. The key is a two-letter. When padding is required, lower case ‘q’ is used. You can assume that columnar transposition will always perform some shuffling between columns, i.e. the key is not alphabetically ordered.

Polybius Cipher: The key is a Polybius square. The 8x8 square used in Assignment 1 (Question 4) is to be used here. The scheme is destructive, such that characters not defined in the Polybius square are removed. An exception is '\n', which is preserved. Also, the scheme produces upper case plaintexts.

Hill Cipher: The key is a four-letter string from the alphabet: abcde. This means the key corresponds to a 2x2 matrix composed of numbers in the range [0,4]. The modular inverse of interest is 26. The encryption scheme is case insensitive and produces upper case ciphertexts and lower case plaintexts. If padding is necessary, use upper case 'Q'.

Copy the matrix.py file that you wrote in Assignment 4 (Question 5) into your final exam folder. Do not change the contents of this file. If you need to add any new functions, do that in the final.py file.

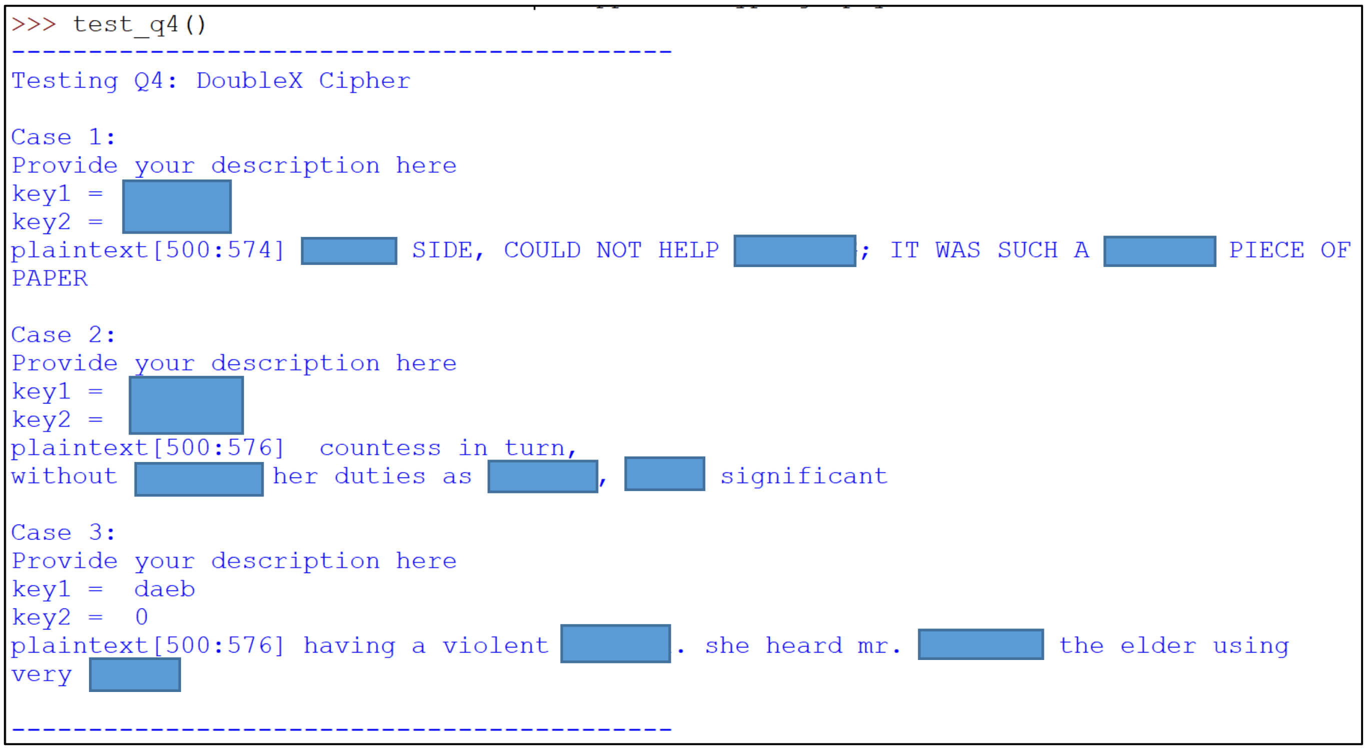
In final.py, complete three functions q4A, q4B and q4C. Each of the functions receives a ciphertext, which is read from the given ciphertext files. The recovered plaintext along with the key should be returned. Your implementation should have a description and code.

At the start of the function, provide a print statement with your cryptanalysis description. List the two schemes you have selected and their order. Explain how you came to that conclusion, and why you have eliminated the other selections.

The code in q4A, q4B and q4C need only to be customized to the solution files provided.

Each function should return recovered plaintext along with a tuple of two keys (order is indifferent). For Atbash and Polybius, you can return an empty string as the key.

Running the testing function will give the following:



```
>>> test_q4()
-------------------------------------------
Testing Q4: DoubleX Cipher

Case 1:
Provide your description here
key1 = [redacted]
key2 =
plaintext[500:574] [redacted] SIDE, COULD NOT HELP [redacted]; IT WAS SUCH A [redacted] PIECE OF
PAPER

Case 2:
Provide your description here
key1 = [redacted]
key2 =
plaintext[500:576]  countess in turn,
without [redacted] her duties as [redacted], [redacted] significant

Case 3:
Provide your description here
key1 =  daeb
key2 =  0
plaintext[500:576] having a violent [redacted]. she heard mr. [redacted] the elder using
very [redacted]

-------------------------------------------
```

Task 5: Public X (6 pts)

This task is related to topics studied under Public Key Cryptography.

Step 1: RSA Key (1 pt)

Inspect the file: public_keys.txt, which has the public keys for everyone in the class. Each line of the file is formatted as: <name m e>, where *m* is the multiplication of two large prime numbers *p* and *q*, and e is the encryption exponent which is an integer relatively prime to *n*.

Record your *m* and *e*, and also the instructor's public key.

Then, email the instructor during one of the question windows requesting your private key. The subject of the email should be: 'CP460: Private Key Request'. Include your name and student ID in the email body. Do not include any other content, because it will be ignored.

When you receive your private key, it is actually your *p* and *q* numbers.

Perform the following:

1- Verify that `m = p*q` (if it is not equal: contact the instructor during the question window).
2- Compute *n*
3- Verify that *e* is relatively prime to *n* (if it is not equal: contact the instructor during the question window).
4- Compute *d*

Edit get_RSAKey function with the above information. The output will look like this:

```
-------------------------------------------
Testing Q5: Public X

Key Information:
name =  John Smith
p =  16779839
```

```
q =  29844557
m =  500786861486323
n =  500786814861928
e =  32452869
d =  55562361680029
```

Step 2: Modular Exponentiation (1 pt):

In final.py, implement the function `LRM(b,e,m)`. The function computes

$$x = b^e \, mod \, m$$

using the left-to-right method taught in class. (Note: most online resources do not represent the same method taught in class).

Testing the function will give:

```
Testing LRM exponentiation:
(66**13) mod 20 = 16
(5**117) mod 19 = 1
(4**60) mod 69 = 58
(1234567890**151515151515151515) mod 190 = 20
```

Note that the LRM method is very efficient, therefore, the computation should be instantaneous. If it is taking you long time (few seconds), then something is wrong in the implementation.

Step 3: Mod96 Encoding (1 pt)

In RSA, everything is a number. Therefore, we need a method to convert text into numbers. We will extend the method taught in class about mode BAAAA cipher, which converted every 5 characters, from base string length = 26, into a number.

In final.py, implement the two functions: `encode_mod96(text)` and `decode_mod96(num,block_size)`.

Your base string is the 96 character returned by the utility function: `get_RSA_baseString()`. In this alphabet, *a* corresponds to *0*, and *ba* corresponds to *96* (96*1+0*1).

The decoding method works in the reverse order, but always produces strings that exactly fit the *given block_size*. This means, the string will be pre-padded with a's

Observe and verify your implementation using the following test cases:

```
Testing mod96 Encoding/decoding:
encode_mod96(a) = 0
encode_mod96(A) = 26
encode_mod96(aB) = 27
encode_mod96(bb) = 97
encode_mod96(ABC DE) = 214315391742

decode_mod96(0,4) = aaaa
decode_mod96(26,3) = aaA
decode_mod96(27,6) = aaaaaB
decode_mod96(97,6) = aaaabb
decode_mod96(214315391742,8) = aaABC DE
```

Step 4: Ecryption (1 pt)

In final.py, implement the function `e_RSA(plaintext,key)`. The key is the tuple `(m,e)`.

The encryption process is as follows:

1- Divide the plaintext into blocks of 6 characters. If last block is less than 6 characters, pad it with 'q'.
2- Convert each block to a number using the encoding scheme developed in Step 3.
3- For each block, compute the corresponding cipher block using the RSA encryption formula: $y = x^e \bmod m$. You need to invoke your LRM function, developed in Step 2.
4- Convert each block back into character stream, using the decode function developed in in Step 3. Note that each cipher block should be 8 characters.
5- The ciphertext is a the concatenation of all strings produced by the previous step.

Step 5: Decryption (1 pt)

In final.py, implement the function `d_RSA(ciphertext,key)`. The key is the tuple `(m,d)`. [Note: whether the number passed to the function is *e* or *d*, this is indifferent to the implementation. The same applies to encryption].

The decryption works in the reverse order. Be careful of the fact that it reads blocks of 8 characters and produces 6 character blocks. Also, remember to get rid of the 'q' padding at the tail, if it exist.

Below is the testing for the encryption/decryption schemes:

```
Testing Encryption/Decryption:

Case 1: Use instructor's public key:

Encryption using instructor public key:
plaintext:  Your theory is crazy, but it's not crazy enough to be true
key:  (535093796775623, 32452999)
ciphertext:  e2i92301cWgDIQSSgHmsT'<obBiHt7acf'M%_DI0basX;UN
f3{%cM&<gZnx`U=xd5=f[,:MdgxYI![e

Decryption using instructor public key:
ciphtertext:  e!WL;i^Vf*-QQFk\dpE(xLNwgW^6TM\Xa9lFss-
fa6#k`ZD(dV|0u~bEa;XR\Z'Ea|#vB<fZcXS7\_F_ek,Y5U`sb_gi1RkvaF7);1]2gJ7).kfX
key =  (535093796775623, 32452999)
plaintext:  Exams test your memory, life tests your learning; others will
test your patience

Case 2: Use your  public/private key:

Encryption using your public key:
plaintext:  Your theory is crazy, but it's not crazy enough to be true
key:  (500786861486323, 32452869)
ciphertext:  aZ,
knI=gZT40443b4^jLWu,c[btwX~@gxk;',0-
a)Empk0agHqjpe<~coY'p@1Pat\])eHJbej~q/)P

Decryption using your private key:
ciphtertext:  aZ,
knI=gZT40443b4^jLWu,c[btwX~@gxk;',0-
a)Empk0agHqjpe<~coY'p@1Pat\])eHJbej~q/)P
```

```
key =  (500786861486323, 55562361680029)
plaintext:  Your theory is crazy, but it's not crazy enough to be true
```

Note that in case 2, the public/private pair for *John Smith* were used. However, the testing file will use your own key pair.

Step 6: Digital Signature Verification (1 pt)

In public key cryptography, digital signatures are used the same way as encryption/decryption. The only difference is the keys used. In encryption, if Alice wants to send a private message to Bob, it uses Bob's public key. Bob uses his private key to decrypt and retrieve the message. On the other hand, in digital signatures, Alice signs a message through encrypting the message using her private key. Bob verifies by decrypting using Alice's public key.

In final.py, implement the function `verify_RSA(message)`. The function receives some message which has been signed by someone in the class. Your function should find out the name of the person who signed that message. If it was not signed by someone from the class, return `‘Unknown’,‘’`. The function should return both the name of the signer and the decrypted message (no need for the key).

Below is the testing:

```
Testing signature verification:

Message 1:
h6}~I:7:hKwX:grYdCTDu*v\dL6BI|0tbU#s&n@2a?F1^S+:iwNn}OM6iMbZ8c) bS-
WE1V&eJ{_VA*)bk>HzElUaRX%8LzDd/>PyHC9g6&^OTsBhUOc'{cWdLZT"rU}gC_Y
Gs#cm"*tCkahjv?vMw`a;4LA{2o
Message Author:  <HIDDEN>
Message:  <HIDDEN>

Message 2:
hhh'j`Z|d(v?Sj-3gvj3X6=idV^+2NV5h)p9mrmbf|6XS`@(be0/xd#kgO4>}ll.d"(K-
7c.gae`[6[2i]yBjd][eBly'wA#ihw;<IAKh""zv~U*jtMz?.erea-
],g:Pc;^*\]O#gkTdLMm_db7y$)U+gM=^xHG=
Message Author:  <HIDDEN>
Message:  <HIDDEN>
```