CMPUT 331, Fall 2020, Assignment 4 (Version 2)

Before beginning work on this assignment, carefully read the Assignment Submission Specifications posted on eClass.

You will submit four files for this assignment: "a4.pdf" or "a4.txt" (for problem 1), a4p2.py (for problem 2), a4p3.py (for problem 3), and a README file (also in either PDF or plain text), indicating who you worked with, which resources you consulted, and any other relevant information as indicated in the Assignment Submission Specifications posted on eClass. Also include any other files needed to run your code, such as additional modules from the textbook; your submission must be self-contained.

Problem 1

The included file "ciphers_version2.txt" contains twelve ciphertexts[1], one per line, each enciphered independently with one of the keyword variant of the Caesar cipher from assignment 1, the columnar transposition cipher from assignment 2, or the affine cipher from "assign4_affine.py" on eClass. Modify the provided file, "generalHacker.py", to support hacking all of these ciphers and decrypt all twelve ciphertexts. You can assume that the lines of text encrypted using the keyword variant of the Caesar cipher are encrypted using a word from dictionary.txt and lines of text encrypted using the columnar transposition cipher have fewer than 10 columns in their keys. Your a4.pdf or a4.txt should contain the twelve plaintexts *in the same order in which their respective ciphertexts are presented* (e.g. the seventh plaintext in your file should be your solution to the cipher given on the seventh line of "ciphers_version2.txt").

IMPORTANT NOTE: in order to get useful results, you may need to adjust the parameters of the "isEnglish" function which is used in "generalHacker.py". You may also assume that codebook values will be non-negative integers.

Problem 2

As you have seen, the simple substitution cipher is stronger than ciphers covered in previous chapters, but it is certainly not unbreakable. One way of strengthening this cipher is to use a *nomenclator*, which adds word-level substitution to the character-level substitution of the simple substitution cipher.

The key to a nomenclator consists of a key to the simple substitution cipher (as seen in chapter 17), and a *codebook*, which lists substitutions for specific words. To encrypt a message with a nomenclator, first substitute all words found in the codebook with one of their respective codebook values, and then encipher all other words using the simple substitution cipher. For example, suppose we have a codebook `'examination':['4', '5'], 'examinations':['6', '7', '8']`. A nomenclator might replace '`examination`' and '`examinations`' with '4' and '7' respectively, and then encipher the rest of the message using the simple substitution cipher. So, 'examination increases disappoint', using the same substitution cipher key as in the textbook with this codebook might result in the ciphertext '5 PLQRZKBZB MPBKSSIPLC'.

[1] Credit: plaintexts two through nine were copied or adapted from Wikipedia.

Using code from ‘simpleSubCipher.py’ as a starting point, create a python module called ‘nomenclator.py’ which implements the nomenclator cipher.

Your nomenclator.py should contain functions named ‘encryptMessage’ and ‘decryptMessage’, which implement encryption and decryption respectively with a nomenclator. These functions should each take 3 parameters: ‘subKey’, which contains a key to the simple substitution cipher (which is simply a permutation of the alphabet), ‘codeBook’, a dictionary matching some dictionary words to a list of non-letter symbols (for example, a key could be ‘`examinations`’, and its values could be ‘`['4', '5']`’). Given a message to encrypt, you consider each word one at a time: If the word is in the codebook, replace it with one of its corresponding symbols chosen at random. If the word is not in the codebook, encipher it using the simple substitution cipher. The codebook is not case sensitive: if ‘uncomfortable’ is in the codebook, then ‘Uncomfortable’ and ‘UNCOMFORTABLE’ will not be in the codebook, should be replaced by one of the symbols given for ‘uncomfortable’. To decrypt a message, simply reverse this process, determine if each word is a value in the codebook, and either replacing it with the corresponding word, or deciphering it using the substitution cipher key.

IMPORTANT NOTE: You can assume that no symbol value will be found in multiple word keys. If the key ‘`examination`’ has the symbol values ‘`['4', '5']`’, then any other key word WILL NOT have `4` or `5` in their symbol value list.

Here are some sample calls to consider – your module should be able to reproduce this. Note that since we are randomly choosing replacement symbols, you may get slightly different ciphertexts, but they should all decrypt back to the same plaintext.

```
>>> import nomenclator
>>> mySubKey = 'LFWOAYUISVKMNXPBDCRJTQEGHZ'
>>> codebook = {'university':['1', '2', '3'], 'examination':['4', '5'],
                'examinations':['6', '7', '8'], 'WINTER':['9']}
>>> plaintext = 'At the University of Alberta, examinations take place in December
                 and April for the Fall and Winter terms.'
>>> ciphertext1 = nomenclator.encryptMessage(mySubKey, codebook, plaintext)
>>> ciphertext1
'Lj jia 1 py Lmfacjl, 6 jlka bmlwa sx Oawanfac lxo Lbcsm ypc jia Ylmm lxo 9 jacnr.'
>>> ciphertext2 = nomenclator.encryptMessage(mySubKey, codebook, plaintext)
>>> ciphertext2
'Lj jia 3 py Lmfacjl, 7 jlka bmlwa sx Oawanfac lxo Lbcsm ypc jia Ylmm lxo 9 jacnr.'
>>> deciphertext1 = nomenclator.decryptMessage(mySubKey, codebook, ciphertext1)
>>> deciphertext1
'At the university of Alberta, examinations take place in December and April
for the Fall and WINTER terms.'
>>> deciphertext2 = nomenclator.decryptMessage(mySubKey, codebook, ciphertext2)
>>> deciphertext2
'At the university of Alberta, examinations take place in December and April
for the Fall and WINTER terms.'
```

Problem 3

In chapter 18, you saw a method for hacking the simple substitution cipher. However, this method cannot always decipher every letter, sometimes resulting in blanks sprinkled throughout the recovered plaintexts. For example, hacking the message

```
Sy l nlx sr pyyacao l ylwj eiswi upar lulsxrj
isr sxrjsxwjr, ia esmm rwctjsxsza sj wmpramh,
lxo txmarr jia aqsoaxwa sr pqaceiamnsxu, ia
esmm caytra jp famsaqa sj. Sy, px jia pjiac
ilxo, ia sr pyyacao rpnajisxu eiswi lyypcor l
calrpx ypc lwjsxu sx lwwpcolxwa jp isr
sxrjsxwjr, ia esmm lwwabj sj aqax px jia
rmsuijarj aqsoaxwa. Jia pcsusx py nhjir sr
agbmlsxao sx jisr elh. -Facjclxo Ctrramm
```

with the simpleSubHacker.py program given in Chapter 18 results in the following decryption:

```
If a man is offered a fact which goes against
his instincts, he will scrutinize it closel_,
and unless the evidence is overwhelming, he
will refuse to _elieve it. If, on the other
hand, he is offered something which affords a
reason for acting in accordance to his
instincts, he will acce_t it even on the
slightest evidence. The origin of m_ths is
e__lained in this wa_. -_ertrand Russell
```

So, the mapping of the cipher letters is incomplete, leaving some blanks in our deciphered text.

We can see that, for example, the cipher letter ‘h’ has not been mapped to the plaintext letter ‘y’, and so the cipher word ‘wmpramh’ is only partially deciphered as ‘closel_’. However, looking at dictionary.txt, the only word which begins with ‘closel’ is, in fact, ‘closely’, so, making the reasonable assumption that the correct decipherment of ‘wmpramh’ is a word in dictionary.txt, the decipherment of ‘wmpramh’ *must* be ‘closely’, and thus cipher ‘h’ must map to plaintext ‘y’. This deduction also reveals that ‘nhjir’ and ‘ely’, initially partially deciphered as ‘m_ths’ and ‘wa_’, should decipher to ‘myths’ and ‘way’ respectively, even without an additional dictionary lookup.

Going one step further, ‘lwwabj’ is initially incompletely deciphered to ‘acce_t’, which could be ‘accent’ or ‘accept’. However, the initial decipherment pass mapped cipher ‘x’ to plaintext ‘n’. Since a simple substitution cipher key must be one-to-one, the only option for ‘acce_t’ is ‘accept’, so we can add to our mapping that cipher ‘b’ maps to plaintext ‘p’.

Finally, ‘e__lained’ can only be ‘explained’, and ‘_ertrand’ can only be ‘Bertrand’ (again, assuming the plaintext does not contain any strange words), thus mapping cipher ‘g’ to plaintext ‘x’ and cipher ‘f’ to plaintext ‘b’.

In short, by making a reasonable assumption, we were able to add to the mapping produced by simpleSubHacker.py, further mapping ‘h’ to ‘y’, ‘b’ to ‘p’, ‘g’ to ‘x’, ‘f’ to ‘b’. With this extended key, we can complete the decipherment:

```
If a man is offered a fact which goes against
his instincts, he will scrutinize it closely,
and unless the evidence is overwhelming, he
will refuse to believe it. If, on the other
hand, he is offered something which affords a
reason for acting in accordance to his
instincts, he will accept it even on the
slightest evidence. The origin of myths is
explained in this way. -Bertrand Russell
```

In this problem, your task is to automate this ‘second pass’ for solving simple substitution ciphers. You should not alter the original mapping produced by the first pass; your code should only add decipherments for symbols not deciphered by the first pass.

You have learned about regular expressions, and how they can be used to identify whether or not a given word matches a given pattern. Use this knowledge to create a Python module ‘modifiedSimpleSubHacker.py’, containing a function ‘hackSimpleSub’, which first runs the decipherment algorithm used by ‘simpleSubHacker.py’, obtaining both the decipherment and the key it produces. Then, if there are any blanks in this initial decipherment, your code should use regular expressions to identify words in ‘dictionary.txt’ which match the patterns of the words containing those blanks (for example, the pattern induced by ‘acce_t’ is matched by ‘accent’ and ‘accept’), and, if possible (i.e. if doing so does not violate the one-to-one constraint of the simple substitution key), adds any induced mappings to the key. Once this is done, the hackSimpleSub subroutine in modifiedSimpleSubHacker.py should produce a final decipherment, in which as many of the blanks as possible are filled in, and return the string containing this decipherment.

For example, given the ciphertext from the above example, the ‘hackSimpleSub’ function in your ‘modifiedSimpleSubHacker.py’ should return the final, underscore-free plaintext above.