CMPUT 331, Winter 2026, Assignment 4: Cryptanalysis

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

Introduction

In previous assignments, you have learned various cryptographic techniques, each requiring the implementation and analysis of different cipher mechanisms. This assignment builds on that foundation, allowing you to explore these techniques further and deepen your understanding of cipher-breaking strategies.

You will submit the following for this assignment:

- **a4p1.py, a4p2.py, a4p3.py**: Python solutions to problems 1, 2, and 3, respectively.
- **decrypted.txt**: Output file of problem 1.
- a README file (**README.txt** or **README.pdf**)
- any other files needed to run your code, such as additional modules from the textbook; your submission must be self-contained.

Submit your files in a zip file named 331-as-4.zip

IMPORTANT NOTE: Please do not modify starter code (i.e., renaming/removing provided functions)

Problem 1

The included file "ciphers.txt" contains twelve encrypted messages[1], one per line, each prefixed by its cipher type (C, T, or A) followed by a semicolon and the ciphertext, with each independently encrypted using the encryption algorithms indicated by cipher type which are the algorithms that you have learned so far from previous assignments:

- C = The Caesar cipher (Assignment 1, Problem 3)
- T = The columnar transposition cipher (Assignment 2, Problem 3)
- A = The affine cipher

Modify the provided file, "a4p1.py", to support hacking all of these ciphers and decrypt all twelve ciphertexts. You can add python files with solution from previous assignments to the assignment folder. You can assume that:

1. The lines of text encrypted using the Caesar cipher are encrypted using an alphabet from "ABCDEFGHIJKLMNOPQRSTUVWXYZ".
2. The lines of text encrypted using the columnar transposition cipher have fewer than 10 columns in their keys.

Your "a4p1.py" should print out only the twelve plaintexts (do not include cipher type) to an output file "decrypted.txt" *in the same order in which their respective ciphertexts are presented* (e.g. the seventh plaintext in "decrypted.txt" should be your solution to the cipher given on the seventh line of "ciphers.txt") and should preserve any punctuations, if present.

Important note:

1. In order to get useful results, you may need to adjust the parameters of the "isEnglish" function which is used in "detectEnglish.py". It is possible to get the correct decryption for all twelve ciphertexts using the same set of parameters.
2. The runtime limit for this problem should not exceed 30 seconds.

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

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, 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 (also referred to as symbols), and then encipher all other words using the simple substitution cipher. For example, suppose we have the following codebook:

```
"year": [‘4’, ‘5’],
"years": [‘6’, ‘7’, ‘8’]
```

A nomenclator might replace **“year”** and **“years”** with ‘4’ and ‘7’, respectively, and then encipher the rest of the message using the simple substitution cipher. So, “Happy new year!” might result in the ciphertext “Gzoox mdv 5!”, if we have each plaintext character shifted to the left by one position as the substitution cipher key (i.e., A $\rightarrow$ Z, B $\rightarrow$ A, and so on).

You are given a starter file“a4p2.py” and are tasked to implement the nomenclator cipher. Your “a4p2.py” should contain functions named “encryptMessage” and “decryptMessage”, which implement encryption and decryption, respectively, with a nomenclator. These functions should each take 3 parameters:

1. **key**: a key to the simple substitution cipher (which is simply a permutation of the alphabet),
2. **message**: the string containing the text to encrypt or decrypt,
3. **codebook**: a dictionary matching some dictionary words to a list of non-letter symbols.

Some hints to consider:

1. 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.
2. To decrypt a message, simply reverse this process, determine if each word is a value in the codebook, and either replace it with the corresponding word, or decipher it using the substitution cipher key.

Important Note:

1. You may assume that codebook values will always be non-negative integers.
2. Punctuations should remain unchanged (i.e., it will appear in the same position in both the plaintext and ciphertext without being encrypted or decrypted).
3. You can also 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 keyword WILL NOT have 4 or 5 in their symbol value list.
4. The codebook is not case sensitive: if 'uncomfortable' is in the codebook, then 'Uncomfortable' and 'UNCOMFORTABLE' will not be in the codebook, but all variant of this word should be replaced by one of the symbols given for 'uncomfortable'.
5. Cases should also remain unchanged for words NOT in the codebook. For example, if 'HeLLo' is in the plaintext and is NOT in the codebook, then the resulting ciphertext should be capitalized in the first, third, and fourth position. The same logic applies on decryption.
6. Words conjoined by special characters are counted as the same word. Words followed or preceded by special characters should be matched in the codebook.
   For example, ['"hello"', 'hello;'] with codebook [hello:['1']] would be encrypted as ['"1"', '1;'].

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 a4p2
>>> 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 = a4p2.encryptMessage(mySubKey, codebook, plaintext)
>>> ciphertext1
'Lj jia 1 py Lmfacjl, 6 jlka bmlwa sx Oawanfac lxo Lbcsm ypc jia Ylmm lxo 9 jacnr.'
>>> ciphertext2 = a4p2.encryptMessage(mySubKey, codebook, plaintext)
>>> ciphertext2
'Lj jia 3 py Lmfacjl, 7 jlka bmlwa sx Oawanfac lxo Lbcsm ypc jia Ylmm lxo 9 jacnr.'
>>> deciphertext1 = a4p2.decryptMessage(mySubKey, codebook, ciphertext1)
>>> deciphertext1
'At the university of Alberta, examinations take place in December and April
for the Fall and WINTER terms.'
>>> deciphertext2 = a4p2.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 :

Your task is to refine the output of a simple substitution cipher solver. The "simpleSub-Hacker.py" program provides an initial decipherment and a key, but as shown in Chapter 17 of your textbook, some letters may remain unresolved (i.e., leaving blanks in the plaintext). Your goal is to fill in these missing letters using dictionary-based pattern matching while ensuring the one-to-one constraint of the cipher key is maintained.

To illustrate this, look at the example below:

```
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
```

Running "simpleSubHacker.py" program on the text above produces the following partial decipherment:

```
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
```

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 that begins with 'closel' is, in fact, 'closely'. By making a reasonable assumption that the correct decipherment of 'wmpramh' is a word in "dictionary.txt", the decipherment of 'wmpramh' must be 'closely', and thus, the cipher letter 'h' must map to the plaintext 'y'. This deduction also reveals that 'nhjir' and 'elh', which are partially decrypted 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 deciphered to 'acce_t', which could either be 'accent' or 'accept'. However, the initial decipherment pass mapped the cipher letter 'x' to the 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 the cipher letter 'b' maps to the 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 the cipher letter ‘g’ to the plaintext ‘x’ and the cipher letter ‘f’ to the plaintext ‘b’.

In short, by making reasonable assumptions, 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
```

You have learned about regular expressions (RegEx) and how they can be used to identify whether or not a given word matches a given pattern. Use this knowledge to complete the given file “a4p3.py”, containing a function ‘hackSimpleSub’, which first runs the decipherment algorithm used by “simpleSubHacker.py”, then repeatedly refines the output by filling in blanks until no unresolved mappings remain. Your function should return the fully deciphered plaintext as a string.

Assignment 4 video guide should be helpful in solving this problem.

Important note:

1. The dictionary contains words that are purely made up of alphabets. Therefore, you must find a way to decrypt words containing special characters such as “-Bertrand”, “way.”, and “closely,”. It is not guaranteed that these special characters appear only at the start and end of the word, they may appear in the middle of some words (e.g. X-ray).
2. Punctuations should remain unchanged (i.e., it will appear in the same position in both the plaintext and ciphertext without being encrypted or decrypted).
3. Cases should also remain unchanged such that if the first character of the first word in the ciphertext is capitalized, then the first character of the first word in the plaintext should also be capitalized.