Applied Information Security and Cryptography

Lab 01: Breaking classical encryption schemes

In this lab, you will try to decrypt two ciphertexts obtained using classical encryption schemes. As usual with this class of encryption schemes, we will encode the plaintext using lowercase letters (no spaces or punctuation) and the ciphertext using uppercase letters. When representing ciphertext or plaintext letters as numerical values, we will use the convention ‘A’ = ‘a’ = 0, ‘B’ = ‘b’ = 1, … ‘Z’ = ‘z’ = 25, according to standard alphabetical order.

Technical note:

All scripts have been tested with Python 3.13.2. The script AISC_01.py requires NumPy (https://numpy.org).

Monoalphabetic cipher

The first ciphertext is in the file “cryptogram01.txt” and has been obtained using a monoalphabetic cipher, i.e., each plaintext letter is mapped to a ciphertext letter according to a fixed one-to-one mapping. Since there are 26! possible mappings, brute force does not help in this case. However, assuming that the plaintext is English text, a frequency analysis should easily reveal the most likely plaintext.

For this task, you can use the Python script AISC_01.py, which provides some useful tools for performing frequency analysis. Namely, you can open the ciphertext in Python using

```
with open("cryptogram01.txt","r") as text_file:
        cryptogram = text_file.read()
```

Then, the following functions defined in the script can be used to get the topn most frequent letters (monograms), pair of letters (digrams), and group of three letters (trigrams) in the ciphertext:

```
monogram_ranking(cryptogram, topn)
digram_ranking(cryptogram, topn)
trigram_ranking(cryptogram, topn)
```

Guessing the 2-3 most likely trigrams in your plaintext and matching them with the most frequent trigrams of the ciphertext is usually sufficient to start recovering the plaintext. Additional missing letters can be inferred looking at the most frequent digrams and monograms in the ciphertext or looking for common English words. To test your guess, you can simply replace a ciphertext letter with your guess and print the resulting text. In Python, this is simply:

```
guess01 = cryptogram.replace('X', 'a')
print(guess01)
```

Of course, you need to guess a few trigrams and digrams before your guess makes sense.

To help your analysis, the following tables report the most likely letters, digrams, and trigrams in English (taken from http://practicalcryptography.com/cryptanalysis/letter-frequencies-various-languages/english-letter-frequencies/):

A	:	8.55	K	:	0.81	U	:	2.68
B	:	1.60	L	:	4.21	V	:	1.06
C	:	3.16	M	:	2.53	W	:	1.83
D	:	3.87	N	:	7.17	X	:	0.19
E	:	12.10	O	:	7.47	Y	:	1.72
F	:	2.18	P	:	2.07	Z	:	0.11
G	:	2.09	Q	:	0.10			
H	:	4.96	R	:	6.33			
I	:	7.33	S	:	6.73			
J	:	0.22	T	:	8.94			

TH	:	2.71	EN	:	1.13	NG	:	0.89
HE	:	2.33	AT	:	1.12	AL	:	0.88
IN	:	2.03	ED	:	1.08	IT	:	0.88
ER	:	1.78	ND	:	1.07	AS	:	0.87
AN	:	1.61	TO	:	1.07	IS	:	0.86
RE	:	1.41	OR	:	1.06	HA	:	0.83
ES	:	1.32	EA	:	1.00	ET	:	0.76
ON	:	1.32	TI	:	0.99	SE	:	0.73
ST	:	1.25	AR	:	0.98	OU	:	0.72
NT	:	1.17	TE	:	0.98	OF	:	0.71

THE	:	1.81	ERE	:	0.31	HES	:	0.24
AND	:	0.73	TIO	:	0.31	VER	:	0.24
ING	:	0.72	TER	:	0.30	HIS	:	0.24
ENT	:	0.42	EST	:	0.28	OFT	:	0.22
ION	:	0.42	ERS	:	0.28	ITH	:	0.21
HER	:	0.36	ATI	:	0.26	FTH	:	0.21
FOR	:	0.34	HAT	:	0.26	STH	:	0.21
THA	:	0.33	ATE	:	0.25	OTH	:	0.21
NTH	:	0.33	ALL	:	0.25	RES	:	0.21
INT	:	0.32	ETH	:	0.24	ONT	:	0.20

Vigenere cipher

The second ciphertext is in the file “cryptogram02.txt” and has been obtained using a Vigenere cipher. In this case, the key is a passphrase of unknown length which is continuously repeated and used as a keystream: each letter of the plaintext is “XORed” with the corresponding letter of the keystream to obtain the ciphertext.

$$C_i = E(p_i, k) = p_i + k_{i \bmod m} \bmod 26$$

```
key:          deceptivedeceptivedeceptive
plaintext:    wearediscoveredsaveyourself
ciphertext:   ZICVTWQNGRZGVTWAVZHCQYGLMGJ
```

If the key length is m characters, then there are 26^m possible keys, which for large m still makes brute force impractical. However, the fact that a short key is repeated still exposes the frequencies of the plaintext language, in this case English text.

To break Vigenere, we will explore an automated approach, based on two observations.

The first observation is that a letter every m (the key length) is encrypted using the same key letter, so the same plaintext letter is always mapped to the same ciphertext letter. This means that if we extract from the ciphertext a subsequence taking a letter every m letters, that is C[0], C[m], C[2*m], C[3*m], and m is the correct key length, the relative frequencies of those letters will be very close to the relative frequencies of single letters in English.

The second observation is that each of these subsequences is actually encrypted using a Caesar cipher, that is, if the key letter if k, a ciphertext letter is obtained by selecting the letter placed k positions after the corresponding plaintext letter. Hence, if P denotes the letter frequencies of the plaintext and Q denotes the letter frequencies of the ciphertext, we have P(x) = Q(x+k mod 26). By looking at Q, and assuming that P is that of English, it should be quite easy to infer k.

Write a Python function performing the following steps:

1. Extract from the ciphertext a subsequence taking one letter every keylen letters. You can easily achieve this with slicing:

   ```
   subsequence = cryptogram[0::keylen]
   ```

   Compute the relative frequencies of the letters in the subsequence. In order to have comparable frequencies for different keylen values, it is recommended to consider subsequences with the same length. A subsequence with at least 100 characters is recommended to have meaningful statistics. Hence, it is suggested to take only the first 100 characters of the extracted subsequence for each keylen. The script AISC_01.py provides a function to compute frequency of letters in a subsequence:

```
crypto_freq(subsequence)
```

2. Compute the following score: $S = \sum_{i=0}^{25} Q[i]^2$ where Q[i] denotes the frequency of the ith letter. If Q is close to English text, S is about 0.065. If Q denotes randomly chosen letters, then Q[i] = 1/26 and S = 1/26 = 0.038. (*<u>Variant</u>: to obtain a more robust statistic, you can compute the above score for each of the keylen possible subsequences that you extract from the ciphertext, that is, starting at position 0, 1, … up to leylen-1. Then, your final score will be the average of the keylen computed scores*)

Now, call this function for different keylen values. Since you want to extract subsequences of at least 100 characters, the maximum keylen value that you should test is bounded by length(cryptogram)/100. For the correct keylen value you should obtain S about 0.065, whereas for the wrong values you should obtain S about 0.038. Use this observation to estimate the key length. (*Hint: You should not expect to get exactly those values, especially if your subsequence is not very long, however for the correct keylen, and for multiples of this value, you should see higher values than in the other cases. Try to plot the obtained scores as a function of keylen: it should be evident which is the most likely keylen value.*).

Once you have the key length, write a Python function to crack the key. The function should do the following steps:

1. Cycle over all possible offsets between 0 and keylen-1.

2. Extract a ciphertext subsequence at offset j with step keylen, e.g., C[j], C[j + keylen], C[j + 2*keylen], …

3. Compute the relative frequencies of the letters in the subsequence.

4. Compute the circular correlation between the above frequencies and the frequencies of English text. The circular correlation between P and Q is:

$$R[k] = \sum_{i=0}^{25} P[i]Q[i + k \, mod \, 26]$$

   For k = 0,1,…,25. The script AISC_01.py has a function to facilitate this:

```
periodic_corr(x, y)
```

   Moreover, you can find frequencies for English text in the vector:

```
english_letter_freqs
```

5. Find the k maximizing R. This should be the key letter at offset j. The rationale is that for the correct k you should observe R[k] about 0.065, whereas for the other values R[k] is about 0.038.

Now you should have found the key. Test it with the function:

```
Vigenere_decrypt(cryptogram, key)
```

Tip: key should be encoded like a Python string, all lowercase letters.

Questions (Answer these in your report)

Describe the strategy used for breaking the monoalphabetic cipher. Which were the first guesses? How did you manage to guess the other letters?

Considering the above strategy, how could you write a function to automate this? Would the provided frequency tables be sufficient for such a function? If not, what other prior information could be useful? (think of the way you made your guesses to answer this)

Estimate the complexity of the algorithm used to break the Vigenere cipher as a function of the key length. Is this polynomial in the key length? (You can make assumptions on the maximum key length that you are expecting)

If you should break Vigenere by hand, using only tools that estimate the frequencies of ciphertext letters as in the case of monoalphabetic cipher, would this be more or less complex than the monoalphabetic cipher? Explain your point in the answer.

Bonus question: how has been cryptogram03.txt generated? Describe how did you find out.

Bonus question: how would you use the “Vigenere cracking algorithm” for attacking a stream cipher where the same key has been re-used for multiple encryptions? (*You will be able to answer this after the lecture on stream ciphers*)