

Programming for Cryptologists

Part 2

October 2026

Introduction

This part builds on top of part 1. If you haven't read part 1, you should probably read it first.

At this stage, you should be familiar with brute-forcing a cipher (like the Caesar cipher) and scoring how good each candidate is.

Part 2 focuses on a better way of scoring candidates, this will be extremely useful in future parts.

Goals

By the end of this guide, you should:

- Understand how tetragrams can be used to score a candidate
- Compile a large corpus of text for generating expected frequencies
- Generate your own tetragram score table
- Have a score function to score how close to English a decrypt is
- Have a Python script to encrypt and decrypt a block transposition cipher
- Have put together your own Python script to automatically break a block transposition cipher (small key lengths only)

Prerequisites

For this guide you will need some basic knowledge of Python. There are many amazing existing guides to help you learn this.

You should also have Python installed (or run it online) if you want to follow along.

Why letter frequencies are not enough?

In part 1, we found letter frequencies as a useful way to score how good our Caesar candidates are (and for those who did the extension, Affine as well).

However, there are some limitations. There are two that we will focus on.

Firstly, the expected frequencies are just an average, the true plaintext might not perfectly match the letter frequency of English in general.

What does this mean? It means that in some cases, a candidate might fit the frequencies better than the true decrypt itself. This happens most with the medium frequency letters where they have very similar frequencies as each other.

Secondly, letter frequencies do not care about the order of letters.

An extreme example of this is a transposition cipher. A transposition cipher does not affect the letter frequencies, it just rearranges the existing letters. Ultimately, we would like the score function to actually take this into account.

Tetragrams

Tetragrams are blocks of four letters. Like letter frequencies, in English you can expect some tetragrams to be more frequent than others. For example, you expect "TION" to appear a lot more than "XXXX". Importantly, "NOIT" would not be as common as "TION", therefore we know that this statistic takes letter order into account.

We typically use a sliding window to generate tetragrams, instead of just taking consecutive blocks of four. This gives us 4x the number of tetragrams from the same text.

Here is an example:

```
HELLOWORLD
HELL-----
-ELLO-----
--LOW-----
---LOW-----
----OWOR--
-----WORL-
-----ORLD
```

Now, we want some way of rewarding 'good' tetragrams and punishing 'bad' tetragrams.

To do that, we need to know which tetragrams appear most frequently.

Language corpus

If we want to build up a table of how frequently we expect each tetragram to appear, we first need to count them up in a sample of English text. To do this, we would like to put together a large language corpus containing a lot of English plaintext.

There are many ways of going about this, you could for example use a sample of Wikipedia or a book. However, there is another nice option that might even be better for National Cipher Challenge, using all the past plaintexts.

Madness has kindly (thank you) preserved all the plaintexts that have previously appeared in the National Cipher Challenge here: <https://github.com/themaddoctor/BritishNationalCipherChallenge>

Your first task is to go to that GitHub repository, press the dropdown next to the green code button and press 'Download ZIP'.

Now that you have a copy of all the past challenges in your downloads, either by hand or preferably programmatically, copy all the `plaintext.txt` files into a single `NCC\_Plaintexts.txt` file.

This should leave you with a single file with a LOT of text in the style that NCC uses. We can then use this to collect data about plaintext.

Generating tetragram table

Heading back to Python, we want to build up a reusable file containing the scores for each tetragram so that we don't need to generate it every time.

The first step is loading all the plaintext we put together into Python.

If you have not seen it before, here is how Python's file reading system works. Make sure that the txt file you created is in the same folder as the Python script.

```
with open("NCC_Plaintexts.txt", "r") as file:
    text_corpus = file.read()

print(text_corpus)
```

At least for now, we do not care about spaces or punctuation (you might in future), so we should clean this text up. Here is a simple way to do it using regex, makes the text uppercase and then removes everything except letters.

```
import re
cleaned_corpus = re.sub(r'[^A-Z]', '', text_corpus.upper())
```

Now to take groups of four letters at a time. The easiest way of doing this is, for every starting index *i*, take letters *i* to *i*+3.

```
for i in range(len(cleaned_corpus) - 3):
    # Takes a slice of 4 chars
    tetra = cleaned_corpus[i:i + 4]
    print(tetra)
```

Similar to what you did last time with letter frequencies, we want to count up how frequently each tetragram appears. This is a good time to introduce the `myDict.get(key, default)` function. If the key does not appear in the dictionary, it returns the default instead of an error.

```
counts = {}
total = 0
for i in range(len(cleaned_corpus) - 3):
    tetra = cleaned_corpus[i:i + 4]
    counts[tetra] = counts.get(tetra, 0) + 1
    total += 1

print(list(counts.items())[:10])
```

Now we can get the frequency of each tetragram by dividing count by total.

```
for tetra in counts:
    frequency = counts[tetra] / total
    print(f"{tetra}: {frequency:.6%}")
```

Aside - Allocating a score to each tetragram

To make scoring easier and importantly quicker, we would like the scoring function to loop over every tetragram in the text, and every time it sees a common one increase the score and every time it sees a rare one, decrease the score. This means that we do not need to generate or store counts and frequencies, just slide a window over the text and keep track of the score so far.

We could just use the raw count as the score and you will find that this often works, however a nicer way is using the logarithm of the expected frequency `log(freq)`. The reasons for this are beyond the scope of this guide, but what you need to know is that $\log(\text{high_freq}) > \log(\text{low_freq})$.

This is only relevant when we are building the tetragram table, when actually scoring text, all we care about is summing together the precomputed score for each tetragram we see.

Generating tetragram table

Knowing this, we can now generate a score for each tetragram using its frequency in the corpus.

```
import math
scores = {}
for tetra in counts:
    frequency = counts[tetra] / total
    score = math.log(frequency)
    scores[tetra] = score

print(sorted(scores.items(), key=lambda x: x[1], reverse=True)[:10], "...")
```

Finally, once we have computed these tetragram scores, we can save all of this into a file so that we don't have to do this every time we run one of our scripts.

```
with open("tetras.txt", "w") as output_file:
    for tetra, score in scores.items():
        output_file.write(f"{tetra}: {score}\n")
```

Scoring a candidate

First, we need to load all these tetragram scores into our solver. Again, we can use Python's file handler and read each line.

```
with open("tetras.txt", "r") as file:
    for line in file.readlines():
        line = line.strip()
        # If line is not empty
        if line:
            print(line)
```

Now what we want to do is to separate the tetragram and the score because they are currently both in the same string. We can also tell Python that score is a number using `float()`.

```
with open("tetras.txt", "r") as file:
    for line in file.readlines():
        line = line.strip()
        # If line is not empty
        if line:
            tetra, score = line.split(": ")
            score = float(score)
            print(f"Tetra: {tetra}, Score: {score}")
```

Then, we just add all these entries into a dictionary so that we can quickly lookup the score of any tetragram.

```
tetras = {}

with open("tetras.txt", "r") as file:
    for line in file.readlines():
        line = line.strip()
        # If line is not empty
        if line:
            tetra, score = line.split(": ")
            score = float(score)
            tetras[tetra] = score

print(tetras)
```

One additional thing we can do, since our language corpus did not contain every possible tetragram, we want to know what score to give a tetragram like 'XXXX' which we did not generate a score for since we have never seen it before. The easiest way of doing this is to just give it the same score as the lowest score of the ones we know. You can compute this using:

```
LOWEST_SCORE = min(tetras.values())
print(f"Lowest score: {LOWEST_SCORE}")
```

Now, scoring the text is pretty simple. All we need to do is to use the same sliding window code and, every time we see a tetragram, add its score to the running text score.

```
def score_tetras(text):
    total_score = 0
    for i in range(len(text) - 3):
        tetra = text[i:i + 4]
        # Get the score for the tetragram, if it doesn't exist default to
        LOWEST_SCORE
        total_score += tetras.get(tetra, LOWEST_SCORE)
    return total_score / len(text)
```

Putting it all together and making sure that the text is cleaned first, we get this.

```
import re

tetras = {}

with open("tetras.txt", "r") as file:
    for line in file.readlines():
        line = line.strip()
        # If line is not empty
        if line:
            tetra, score = line.split(": ")
            score = float(score)
            tetras[tetra] = score

LOWEST_SCORE = min(tetras.values())

def score_tetras(text):
    total_score = 0
    for i in range(len(text) - 3):
        tetra = text[i:i + 4]
        # Get the score for the tetragram, if it doesn't exist default to
        LOWEST_SCORE
        total_score += tetras.get(tetra, LOWEST_SCORE)
    return total_score / len(text)

text = """My dear Mademoiselle Bernhardt, I am sorry to tell you that
despite our best efforts we have been unable to repair the damaged plate. I
fear that the fumes from the adhesives we applied may have damaged it
irretrievably, since even though we were able to reassemble the parts, the
image is badly fogged..."""

# Clean text
text = re.sub(r'[^A-Z]', '', text.upper())

print(score_tetras(text))

ciphertext = text[::-1] # Reverse the text

print(score_tetras(ciphertext))
```

Try scoring different plaintexts and ciphertexts, see what you notice.

You will find yourself copy pasting this into most of your cipher scripts in the near future.

Close to plaintext

Our tetragram scoring system has an interesting property.

Investigate what happens if only a few letters are wrong (e.g. you got the monoalphabetic substitution key slightly wrong).

Since a small mistake only affects a few tetragrams, the score is lower than the true plaintext but still closer than a decrypt with more mistakes. This gives us a function that not only lets us decide which text is the plaintext, but also tells us if we are getting closer or further from the correct decrypt. This will be extremely useful in the future when we can no longer brute force all keys.

Block transposition cipher

Block transposition is a type of transposition cipher. This means that it does not change the letters but instead rearranges the letters according to a rule. It is a perfect cipher to test our new scoring function.

The way the block transposition cipher encrypts a text is by taking blocks of N letters at a time and the rearranging them according to the key. Rearranging can be done by taking the kth letter of the block where k is determined by the nth value of key.

An example might help:

```
PLAINTEXT: ATTACK AT DAWN  
KEY: (3,0,2,1)
```

```
BLOCKS:
```

```
ATTA
```

```
CKAT
```

```
DAWN
```

```
REARRANGED BLOCKS:
```

```
AATT
```

```
TCAK
```

```
NDWA
```

```
CIPHERTEXT:
```

```
AATTTCAKNDWA
```

Decrypting is exactly the same but using the inverse of the key. Try to convince yourself of that, and put together an algorithm to inverse a given key.

To actually implement this in code, it is surprisingly easy. First you want to get blocks on n letters at a time.

```
def decrypt_block_transposition(ciphertext, key):
    plaintext = ""

    n = len(key)

    # Take multiples of n
    for i in range(0, len(ciphertext), n):
        # Take a block of size n
        block = ciphertext[i:i + n]
        print(block)

    ciphertext = "AATTTC AKNDWA"
    key = (1,3,2,0)

    print(decrypt_block_transposition(ciphertext, key))
```

And then, following the key, we take the kth letter of each block and add it to the plaintext.

```
def decrypt_block_transposition(ciphertext, key):
    plaintext = ""

    n = len(key)

    # Take multiples of n
    for i in range(0, len(ciphertext), n):
        # Take a block of size n
        block = ciphertext[i:i + n]

        for col in key:
            if col < len(block):
                plaintext += block[col]

    return plaintext

    ciphertext = "AATTTC AKNDWA"
    key = (1,3,2,0)

    print(decrypt_block_transposition(ciphertext, key))
```

You now have a script to decrypt block transposition ciphers. One more step left, brute forcing the key.

Breaking the block transposition cipher

At this point, the only thing in the way of you breaking the block transposition cipher is generating the keys.

Try write this bit yourself.

If you need a hint, you can use a for loop to try different key lengths, and then Python has a library called itertools which provides you with the permutations of a list.

Then just put everything you have built today together, like in part 1 and print the best key and best decrypt.

Putting it all together, full example

```
import re
from itertools import permutations

tetras = {}

with open("tetras.txt", "r") as file:
    for line in file.readlines():
        line = line.strip()
        if line:
            tetra, score = line.split(": ")
            score = float(score)
            tetras[tetra] = score

LOWEST_SCORE = min(tetras.values())

def score_tetras(text):
    total_score = 0
    for i in range(len(text) - 3):
        tetra = text[i:i + 4]
        total_score += tetras.get(tetra, LOWEST_SCORE)
    return total_score / len(text)

def decrypt_block_transposition(ciphertext, key):
    plaintext = ""

    n = len(key)

    for i in range(0, len(ciphertext), n):
        block = ciphertext[i:i + n]

        for col in key:
            if col < len(block):
                plaintext += block[col]

    return plaintext
```

```

ciphertext = ""CHOTA IELRF MMORA MYYRD RCAEH LIRAE AVHIE ADERT REEHP TORON
ERPOA ONITM CURER NDAYA MPIMR SESED WWOHE ITLLW EXSAE TEUCD HAEWV LRAEE
YBDAE NMAGA FAUNC RIUTN HETGS VERER OMRFT PLEHA ANSND ISTIA EARLD ROPYV
GENIX EMRTE USYLE LIUFH EVWOE MDARI PPASO TENID ATHTY FAUOI DTELO OPRPE
YDLRE ROTS Y LILAN IMRCI TIANN OCDGU NTEMS ATGAH TISAL ASHLE UGONH ESITI
RIBNT HISIN LLETI NCEGE HAOTV OUFEN UTODT TBAHO HASSS CKIPE POU DN PRYME
OUIVS TTELE OYTRO EGRUA INDRG INOGG EAHAD THIWT OPEHE TIARO TWINA ECDSE
LYTNE RYCNP DWETI ASHTU TISBT IOTUN PHICE OOSRN AFYLE EMMWB SOREF SSOBW
LBLIE LEBAT ECDOR TIPYT DENAV FENEW WIREL EABLB TOELL KINIT CKABT SBUOU
HITTS ULOCD AMEBA RMOJI AKTSE ATGAH SKIAE INPEG LOCAS YEEEE HETNR ORPET
DWNAI WALLR SIUNF YONAN ANMEA STEGO CKIPU NOOPU RATRI TIILS LINUK YTLEH
THTAI ILWSL ADEL T UROOD COSIV YBREU TIITS ILTSL UNNAN ESCES YRRAI THKSA
ESWTH LDUOA TOMIA IDOVA OUYSH ESVAE THNEE WHENE QUDAA ERTRS VEAHB NCEEO
LEPMT ANDED STSIA INTRG GEOTT ANPUD NNURI ITGNW LBLIE TTIFE ITWDH ENHTE
UPSWE OMC RP ERTUS RTOHL NDAYS EWWOI BELLA ETLBO NAURH ENGUU ERBMO OMC FP
ATTUI STNOH RESIA YILLN EARCS THSEE PAACB ITLII OFSEO OPRUE TIARO TOSNT
POEHI OFTNB NGIEA ETLBO ACRCK OUHTS DSNAO ILMFI RYATM SASEG ASSEE NDOCW
AVHEE ARTST TODEM EOVOU EMMRB STREO ISHTH DQAEU TERAR HETSS ENICT TSSIA
ENDNG EENIR FOOSU PEORR IOTAN REASA EARLD ITS YU EDTAI HETNU ERDNG UNORD
BIALN EFHTA LIICT YTBYH IMTEE UROYE IVECE ISHTM SASEG EWWEI ALLLR DYAEH
EBVAE NTUGH REPEL INMIA STYRA SOEGF RNUOE OPTXE TIARO OPSNL SEAEK PUEEN
RTEDH ADREA ORFRT UPEHC INMOG TUUFR EWWEI COLLN CTATY WHUOE TIINS CEENS
RYASF YOROU PLOTA OUY YR RTAPI HITNS EAHTT CAIRL STAME IEPRC HETEU ALUSC
NNAHE OFSLC MUMON ATCII WINOL ECBLL EDSOF ASROH TWROH ESLIO TINUL UHOYE
OTRAH WIRES LEPEA KEESE OPTPR OCTOO OUFLR XIISN DEROR STOTA IDHYD FRNEO
OSBMS UW OYI HELLA ROFRM BESUF EARON AJMYO APHRP INNEG ARMSY OLOBE ""

```

```

ciphertext = re.sub(r'^[A-Z]', '', ciphertext.upper())

```

```

best_score = float('-inf')

```

```

best_plaintext = ""

```

```

best_key = None

```

```

for key_length in range(2, 7):

```

```

    for candidate_key in permutations(range(key_length)):

```

```

        plaintext = decrypt_block_transposition(ciphertext, candidate_key)

```

```

        score = score_tetras(plaintext)

```

```

        if score > best_score:

```

```

            best_score = score

```

```

            best_plaintext = plaintext

```

```

            best_key = candidate_key

```

```

print("Best plaintext:")

```

```

print(best_plaintext)

```

```

print(f"With key {best_key}")

```

Congratulations

Congratulations, you now have an improved scoring function that you will find extremely useful going forwards and also have a script that can break block transposition ciphers. Next time we will start looking at ciphers that cannot be brute forced anymore.

Until part 3!

Extensions:

How far can you push your script? How long a key length can you break? Can you optimise it further?

Change your script to break columnar transposition ciphers as well.