Programming for Cryptologists Part 3

November 2026

Introduction

This part builds on top of part 2. Please read it and make sure you understand tetragrams before continuing.

Part 3 investigates what happens when you can no longer brute force a cipher by looking at the monoalphabetic substitution cipher, and also introduces an algorithm that can be used to break a wide variety of classical ciphers.

Goals

By the end of this guide, you should:

- Understand how a monoalphabetic substitution cipher works
- Have a Python script to encrypt and decrypt a monoalphabetic substitution cipher
- Know what hill climbing is
- Understand how and why hill climbing works
- Have implemented hill climbing to break a substitution cipher
- Understand how to adapt your script to work with various different ciphers

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.
Additionally, knowledge of tetragram scoring is also required, see part 2.

Monoalphabetic substitution cipher

The monoalphabetic substitution cipher acts somewhat similar to a Caesar cipher. Every letter is mapped to a different letter using a rule. However, unlike Caesar, this isn't done by shifting a letter but by using a keyed alphabet.

For example, the key could map the letter A to F and the letter B to Z and the letter C to A and so on.

The best way to see how the key does this is to lay the key underneath an unkeyed (normal) alphabet.

```
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
K E Y W O R D A B C F G H I J L M N P Q S T U V X Z
```

From this, we can see that any A gets turned into K, any B gets turned into E and so on.

Trying to word this in a way that is easier for a computer to understand, A is the first letter of the alphabet and so should be replaced by the first letter of the key.

Now that it has been worded like this, it is not too difficult to rewrite as Python:

```
def monoalphabetic(text, key):
    ALPHA = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
    res = ""
    for l in text:
        res += key[ALPHA.index(l)]
    return res

print(monoalphabetic("ABCDEFG", "KEYWORDABCFGHIJLMNPQSTUVXZ"))
```

Decrypting is just the same as encrypting, just using the inverse key.

To calculate the inverse key, using the example key above, you find where the letter A appears in the keyed alphabet and see what letter maps to it (H). In the inverse key, A now maps to that letter (H), and so H is the first letter of the inverse key. Then you repeat this for every letter alphabetically. In Python this is:

```
def inverse_monoalphabetic_key(key):
    ALPHA = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
    inverse = ""
    for l in ALPHA:
        inverse += ALPHA[key.index(l)]
    return inverse
```

So to encrypt and then decrypt a message, we can now do this:

```
key = "KEYWORDABCFGHIJLMNPQSTUVXZ"
plaintext = "THE QUICK BROWN FOX JUMPS OVER THE LAZY DOG"
plaintext = plaintext.replace(" ", "")
ciphertext = monoalphabetic(plaintext, key)
print(ciphertext)
inverse_key = inverse_monoalphabetic_key(key)
decrypted = monoalphabetic(ciphertext, inverse_key)
print(decrypted)
```

Typically, the keyed alphabet is derived from a keyword using a variety of techniques, in the example key above this was done by writing out the keyword followed by all remaining letters

alphabetically and then removing all duplicates (you cannot have duplicate letters in a monoalphabetic key).

Depending on the challenge, sometimes the keyword is in the alphabet used to encrypt the text and sometimes it is in the key used to decrypt the key. The easy way to find out is to just print both the key you find and the inverse of that key and see if you can spot a keyword.

The hill

In Part 2 I briefly mentioned an interesting property of our tetragram scoring function, that if a text is only slightly worse than the correct text (e.g. a few letters wrong), it only gets a slightly lower score.

To investigate this further, we will use the following setup. It contains our tetragram scoring function from part 2, the monoalphabetic decrypt function we just wrote, a simple function to switch two random letters in the key and a medium sized piece of plaintext (for example the decrypt to this year's 2B).

```
import re
import random

############################################
# Your tetragram code from part 2 goes here
############################################

def swap_letters(key):
    # Convert to a list so we can swap letters
    key_list = list(key)
    # Pick two random indices to swap
    i = random.randint(0, len(key_list) - 1)
    j = random.randint(0, len(key_list) - 1)
    # Swap the letters
    key_list[i], key_list[j] = key_list[j], key_list[i]
    # Convert back to a string and return
    return ''.join(key_list)

def monoalphabetic(text, key):
    ALPHA = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
    res = ""
    for l in text:
        res += key[ALPHA.index(l)]
    return res

plaintext = """ADD YOUR PLAINTEXT HERE"""

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

```
key = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"

print(score_tetras(monoalphabetic(plaintext, key)))
```

In this example, decrypting using the key ABCDEFGHIJKLMNOPQRSTUVWXYZ just gives the exact same plaintext, and so it represents using the 'correct' key. Since this is the true plaintext, it should give the highest score. We call this score the global maximum.

Now, let's make the key slightly worse by switching 2 letters around in it.

```
slightly_worse_key = swap_letters(key)
print(score_tetras(monoalphabetic(plaintext, slightly_worse_key)))
```

You should (hopefully) notice that the score is only slightly less than the global max.
Let's try progressively increase the number of swaps and observe what happens to the score.

```
print(score_tetras(monoalphabetic(plaintext, key)))

for switch in range(20):
    key = swap_letters(key)
    print(score_tetras(monoalphabetic(plaintext, key)))
```

If you run it a few times, you should notice that the score progressively decreases as the key gets further and further from the correct key, until some point that is it just so far that they are all almost equally bad (a bit after 10 swaps).

You can kind of imagine this forming a hill around the global maximum, where the correct key is the highest point and keys near the correct key getting higher the closer they are to the correct key.

Hill climbing

So far we have only gone down the hill. It would be nice if we had some way of climbing the hill and so getting closer and closer to the correct key. There is!

Hill climbing is an optimisation technique. It involves taking incremental steps and, if the change produces an increased score, continuing making incremental changes to the new solution until no further improvements are made. A lot of bits will be simplified in this guide and we will focus entirely on stochastic hill climbing, there are many other variants to explore if you are interested.

Imagine you are climbing a hill. This algorithm says take a small step in a random direction. If you have gone up, you keep the new position, otherwise you go back. Then you keep repeating it until you stop improving. You can see how this slowly moves you up the hill until you reach the top.

To make a small step, we mutate the current key. For the monoalphabetic substitution cipher, switching 2 random letters is sufficient.

Hill climbing in code

Try having a go at this by yourself first. Your code needs to start with an initial key (for example the unkeyed alphabet), create a new candidate key by copying the best key and then switching 2 letters at random, and check if the candidate decrypt is better than the current best score. If it is, then that becomes the new best key, otherwise it is just discarded. Your code should also keep track of the number of changes since the last improvement (a counter increased each attempt and reset if a better key is found), and break out of the loop if it exceeds a threshold (try 3000).

Here is what I put together:

```
ciphertext = """
PASTE YOUR MONOALPHABETIC CIPHERTEXT HERE
"""

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



key = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"

best_score = float('-inf')
best_plaintext = ""
best_key = key

num_attempts_without_improvement = 0

while True:
    num_attempts_without_improvement += 1
    if num_attempts_without_improvement > 3000:
        break

    candidate_key = swap_letters(best_key)
    candidate_plaintext = monoalphabetic(ciphertext, candidate_key)
    candidate_score = score_tetras(candidate_plaintext)
    if candidate_score > best_score:
        best_score = candidate_score
        best_plaintext = candidate_plaintext
        best_key = candidate_key
        num_attempts_without_improvement = 0
        print("New best score:", best_score)


print("Best score:", best_score)
print("Best key:", best_key)
print("Inverse key:", inverse_monoalphabetic_key(best_key))
```

```
print("Best plaintext:\n", best_plaintext)
```

Local maxima

If you run your code many times, you might notice that it doesn't find the correct plaintext every time. This is because there isn't just a single hill, there are actually many different smaller hills. If our algorithm climbs to the top of a lower hill, it gets stuck as it can't go any higher.

There are different ways to stop it getting stuck. One of the simplest is allowing it to occasionally take a step down. The exact parameters may need some adjustment and vary from cipher to cipher, however for monoalphabetic, here is a good starting point from a bit of testing: if the score is within 2% of the best score, then there is a 1% chance that you accept it.

In code, that looks like just adding this elif clause:

```
    elif candidate_score != best_score and abs((best_score -
candidate_score) / best_score) < 0.02 and random.random() < 0.01:
        best_score = candidate_score
        best_key = candidate_key
        num_attempts_without_improvement = 0
        print("Dropped to:    ", best_score)
```

You should notice that your program is now a lot better at getting unstuck. Escaping local maxima is a major focus when trying to hill climb more complex ciphers, these parameters will probably not work on other ciphers.

Key prediction

Another major way to speed up our monoalphabetic breaker is to give it a good starting point. Currently we are using something random as the start point, there is a chance it is very far from the hill and our algorithms will have a hard time getting to the correct solution.

Since we know what letter frequencies we expect, we can use this to approximate a good key. We can guess that the most common letter in the ciphertext maps to E, the next most common to T and so on.

Try write a few extra lines of code that run just before the hill climbing algorithm that use a quick and cheap frequency count to take an initial guess at the key. Be careful that you don't accidentally try and guess the inverse of the key, you may need to test this a bit to make sure. You should notice that your script is now noticeably faster and doesn't hit local maxima as often. If not, you should probably test the prediction logic again.

How to adapt this for other ciphers?

A lot of this algorithm can be reused to break many different classical ciphers, though some changes need to be made.

Firstly, hopefully this one is obvious, you need to swap the decrypt function out for something that can decrypt the cipher you want to break.

Next, you will probably need to change how you mutate the key each round. Switching 2 random letters in the key might be enough for substitution however many ciphers require more advanced key mutations. For example, my Playfair code has more than 10 different ways of mutating the Playfair key each step. The best way to decide on these is to think how two keys can be close and make a function that is able to move between the two.

You may need to change the scoring function. For most classical ciphers, tetragrams are usually a safe choice, however for ciphers like the Wheatstone clock cipher you may need to change it out for a different function.

Random drops downs are something you need to tune very carefully to make sure you avoid getting stuck at local maxima but don't escape the global maximum. Sometimes you might use additional techniques to help with this like simulated annealing.

You may also want to change how long the algorithm can go without seeing an improvement, too long and it takes a lot of time to run, too short and it might give up too early.

Congratulations

You just broke a cipher with 403,291,461,126,605,635,584,000,000 possible keys. You will find that hill climbing will be extremely useful in breaking many classical ciphers. With a bit of modification, you can use it to break some of the past 10B challenges (no promises about this year's).

Get ready for part 4!

Extensions:

You now know the core algorithm behind hill climbing, try applying it to a different cipher. Start easy, I might suggest tackling the transposition cipher from the previous part, see if you can get it to work for large key lengths like 9 to 15.

Have a look at this book written by madness:
https://github.com/themaddoctor/classical_crypto_book/blob/main/version2.pdf.
It might be a lot to take in, especially if this is unfamiliar to you, but if you pick one of the simpler ciphers and apply the same hill climbing you just learnt, you should be able to break quite a few of them.

Explore different variants of hill climbing. In particular, simulated annealing, fastest ascent and shotgun hill climbing. In my experience, some seem to work better for some ciphers but not for others so it is worth trying different methods and seeing which works best for each cipher.