

"CAESAR SAID WHAT?!": Breaking Substitution Ciphers Programatically

Introduction

One of the most basic and oldest forms of encryption is the substitution cipher. The substitution cipher is a simple encryption technique where every character of some text, known as the **plaintext**, is encrypted by replacing every letter with another letter, forming encrypted text, known as a **ciphertext**. Using substitution ciphers is simple and easy to use but unfortunately it is not secure enough to hold up to more advanced cryptanalysis.

In this project we will be exploring two simple substitution ciphers: The Caesar Cipher and an arbitrary Monoalphabetic Cipher. We'll get a good understanding of each cipher and create programs that could automate the cracking of such simple ciphers. At the same time we'll also exercise our ability to manipulate strings and parse files within program languages.

1 The Caesar cipher

Let's start off by taking a look at the Caesar Cipher. A Caesar Cipher is a simple cipher where each letter within some plaintext is replaced by a letter at some fixed offset down the alphabet. For example, if you were to use a Caesar Cipher with an offset of +4, the letter "A" would be replaced with the letter "E" since "E" is +4 positions after "A". Figure 1 shows the resulting alphabet mapping resulting from a Caesar Cipher with the offset +4.

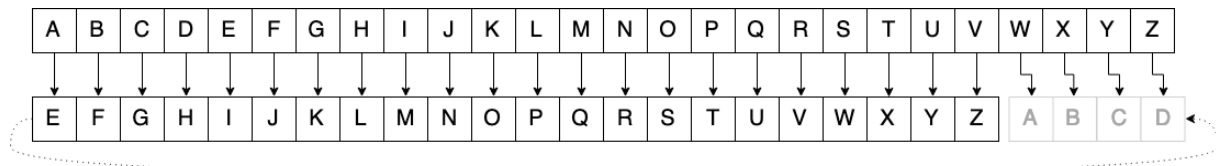


Figure 1: Example of Caesar Cipher with an offset of +4

You can think of the Caesar Cipher as mapping letters based on indexes. We can generalize the caesar cipher with the following equation: Given some letter with an index of x , and some offset k , the index of the new letter $E(x)$ would be:

$$E_n(x) = (x + k) \mod 26$$

1.1 Breaking the Caesar Cipher

Exploiting the limitations of the Caesar Cipher for a Brute Force Attack

We know that the Caesar Cipher simply offsets the original alphabet by some value, so the number of ways of encrypting a message with a caesar would be equal to the amount of ways you can offset the original alphabet. If you notice, starting from offset 0 (which is the original alphabet), you can offset the alphabet 25 times before you return to the original alphabet.

Since we know that there are only 25 ways to possibly offset an alphabet within a Caesar Cipher, we can feasibly brute force the decryption of a ciphertext decrypted with a Caesar Cipher by generating all possible plaintexts at each offset and picking the one that makes the most sense.

Determining the correct plaintext

When using brute force to generate possible decrypted plaintexts, It is easy for a human to determine which plaintext is the correct plaintext as we know what the English language "should" look like. However that is not good enough as we are trying to break the caesar cipher in a completely programmatic way. How can we let computer program figure out which plaintext is correct and makes sense?

A simple and naive way to go about it is to have the computer program check the words of a plaintext against a dictionary/list of actual English words to determine which words "makes sense". This way, a program can determine which decrypted plaintext is most likely the correct plaintext by finding the plaintext with the most amount of words that "make sense".

1.2 Program Implementation

For this part of the assignment you will be creating a program called CrackCaesar that will decrypt a ciphertext using brute force and a dictionary of common words.

In general, your CrackCaesar program will do the following:

1. Take in two text files as arguments (in order):
 - (a) A text file containing ciphertext needing to be decrypted
 - (b) A text file containing common words to check your plaintext against
2. Break the ciphertext using brute force, determining the correct decoded plaintext by checking if decoded words match words in the text file of common words.
3. Print out the determined offset for the Casesar Cipher and the decoded plaintext. (Note: The case of the output does not matter)

You can implement your program with any of the following languages:

- Python
- Java
- C/C++

Under the following constraints:

- **The name of your implementation should be CrackCaesar exactly.** Our grading scripts will compile the program if necessary, and call the program accordingly with the proper arguments. Using your program should look along the lines of the following:
 - For Python:

```
$ python CrackCaesar.py ciphertext.txt dictionary.txt
```
 - For Java:

```
$ java CrackCaesar ciphertext.txt dictionary.txt
```
 - For C/C++:

```
$ ./CrackCaesar ciphertext.txt dictionary.txt
```
- **Keep your implementation to one file.** This requirement is for those that chose a compiled language. This will make compilation easy and straight forward.
- **You can only use the standard libraries.** Do not use any libraries outside of the standard libraries for your programming language. This is to ensure we don't have download anything extra when grading.

1.3 Example Run

To show a better example imagine, we have a text within the ciphertext called *ciphertext.txt* and our dictionary of common words called *dictionary.txt*.

ciphertext.txt

```
aopz tlzzhnl pz jvtwslalsf jvumpkluaphs.  
p opk aol tvulf ha aol lknl vm aol zhohyh  
klzlya
```

dictionary.txt

```
you  
it  
not  
or  
be  
are  
from  
.....
```

Running the program written in python should look like the following:

Command Line

```
$ python CrackCaesar.py ciphertext.txt dictionary.txt  
7  
this message is completely confidential. i hid the money at the edge  
of the sahara desert
```

1.4 Testing your program

In order to test your program, provided should be a folder called **CaesarSample** with two sample text files

1. **ciphertext-sample.txt** - a sample ciphertext encrypted using a caesar cipher
2. **dictionary-sample.txt** - a text file containing common words separated by line
3. **plaintext-sample.txt** - the correct plaintext that your program should output



Note: The sample dictionary is based on 10,000 of the most common words determined by analysis of Google's Trillion Word Corpus. You can learn more here: <https://github.com/first20hours/google-10000-english>



Correctness: The correct output of your program should when decrypting ciphertext-sample.txt should be similar to the contents of plaintext-sample.txt. Do not worry about the case of the letters if your program outputs different letter cases, they will be ignored. Note that the sample ciphertext is merely a sample and will not be used when testing your program so make sure you implement your program correctly!

2 General Monoalphabetic Substitution Ciphers

The Caesar cipher is just a simple example of what is known as a **monoalphabetic cipher**, a cipher where each letter from the original alphabet text is encrypted using some fixed alphabet mapping. This means each letter is replaced by one other letter. In the case of the Caesar Cipher, the fixed alphabet mapping was simply the original alphabet offset by a certain value, but in general the mapping for a monoalphabetic cipher can be completely arbitrary. Notice in figure 2 and 3 how each letter is mapped to exactly 1 other letter, with no noticeable patterns within the letter assignments.

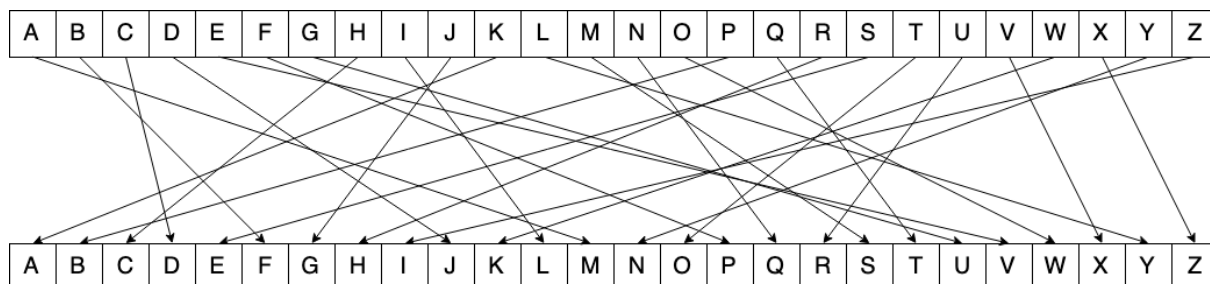


Figure 2: Example of an arbitrary mapping for a monoalphabetic cipher

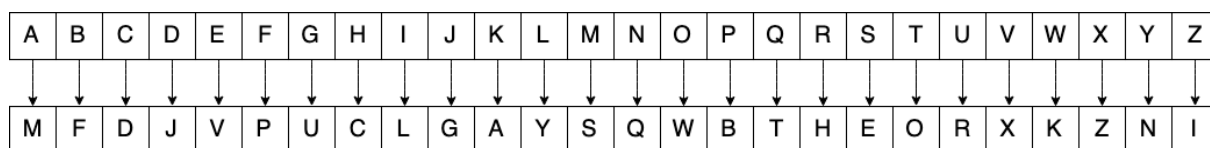


Figure 3: Resulting alphabet mapping from figure 2

2.1 Breaking a Monoalphabetic Cipher

Using Brute Force

Breaking the Caesar Cipher using brute force is easy as alphabet mappings are determined by an offset making the alphabet key space to search was limited to 26 possible offsets. (or 25 if you exclude offsetting by 0). But what if the alphabet mappings were arbitrary? By this we mean, where letter assignment is not bounded by any order/offset?

In order to crack the an arbitrary substitution cipher using brute force we will have to try every alphabet mapping possible to decrypt every possible plaintext. Assigning a letter to letters A-Z, we would assign 26 of the possible letters to letter A, then assign one of the remaining 25 letters to B and so on. In the end we would can see that number of alphabet mappings to search is:

$$26 \times 25 \times 24 \times \cdots \times 1 = 26! = 4.03 \times 10^{26}$$

To try out 403 septillion alphabet mappings is a lot to do, even for a computer. To put that into a more relatable perspective, 403 septillion is a bit more than the the square of 20 trillion (20×10^{12}) and far greater than the number of sand grains on planet Earth (7.5×10^{18}). Nonetheless, you can still try and crack an arbitrary monoalphabetic cipher using brute force, but it will take a long, long, long, time.

Frequency Analysis: A Better Approach

A better approach to breaking a monoalphabetic substitution cipher is to use frequency analysis. **Frequency analysis** is the study of the frequency of letters or sets of letters. Like every language, the English language has rules that determine what letters we can be used together to form words. Because of these rules, the English language tends to exhibit certain characteristics in terms of letter frequency. By examining the letter frequencies, we can potentially get some idea of which letters map to which.

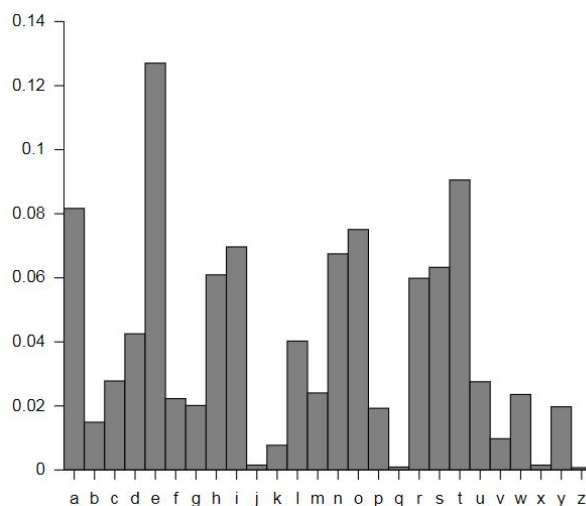


Figure 4: Typical distribution of letters in the English language (source: wiki)

Getting Accurate Statistics for Frequency Analysis

Figure 4 shows the typical letter distribution of letters within the English language. These frequencies could be used to help decipher most ciphertexts but the use of these letter frequencies may be too general to be used for more elaborate ciphertexts. How can we get better statistical results to aid in deciphering a ciphertext in question? One approach to this problem is to perform a frequency analysis on texts that potentially relate to the ciphertext. The idea is that related texts could exhibit similar sentence structures and word use. Because of this, using related known texts could potentially give us better statistical results that could be used to decipher the ciphertext faster.

If letter frequencies from known texts don't help up give the write alphabet mapping, we can further analyze the frequencies of groups of letters called N -grams, which are sets of N consecutive letters. N -grams could give a deeper insight of patterns within texts and can give more hints on possible letter mappings.

2.2 Program Implementation

For this part of the assignment we will only focus on individual letter frequencies. You will create a program called *CrackCipher* that will analyze a set of known texts and decrypt a ciphertext by matching up letter frequencies. You can assume that relative letter frequencies of "known texts" will be perfectly representative of the relative letter frequencies within the ciphertext. By "perfect", we mean that matching letters based on relative letter frequencies will result in a correct alphabet mapping.

To make it slightly harder provided ciphertexts will also have spaces substituted. This means the substitution cipher allows for the space character to be used to substitute any other letter and vice versa. This is a more realistic way of using a substitution cipher as doing this will hide the separation between words making it impossible to deduce letter mappings based on word lengths in ciphertexts. Fortunately despite being able to hide words, it is quite easy to determine what letter maps to the space character with frequency analysis as more often than not, the space character is the most frequent character within a given text.

Your *CrackCipher* program will do the following:

1. Take in two arguments (in order)
 - (a) A text file containing ciphertext needing to be decoded
 - (b) A fairly long text file containing known text related to the plaintext trying to be recovered
2. Perform frequency analysis on the letters within the known text (including spaces).

3. Perform frequency analysis on the letters within the ciphertext (including spaces).
4. Generate an alphabet mapping to use based on similar letter frequencies (which includes mapping the space character).
5. Print out the decoded plaintext using the determined alphabet key. (Note: The case of the output does not matter)

Note: When parsing texts, be sure not to confuse newline characters with spaces

Similar to the last program, you can implement your program with any of the following languages:

- Python
- Java
- C/C++

Under the same following constraints:

- **The name of your implementation should be CrackCipher exactly.** Our grading scripts will compile the program if necessary, and call the program accordingly with the proper arguments. Using your program should look along the lines of the following:
 - For Python:

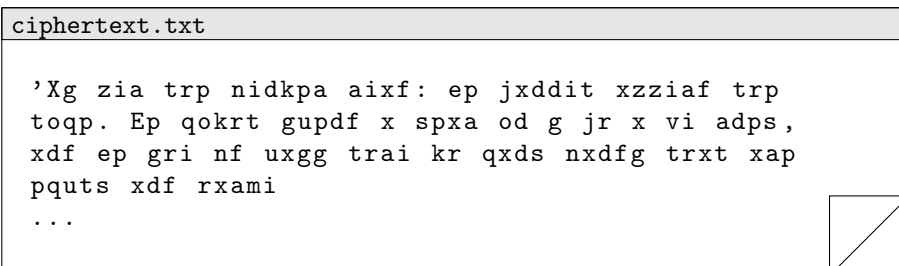
```
$ python CrackCipher.py ciphertext.txt knownplaintext.txt
```
 - For Java:

```
$ java CrackCipher ciphertext.txt knownplaintext.txt
```
 - For C/C++:

```
$ ./CrackCipher ciphertext.txt knownplaintext.txt
```
- **Keep your implementation to one file.** This requirement is for those that chose a compiled language. This will make compilation easy and straight forward.
- **You can only use the standard libraries.** Do not use any libraries outside of the standard libraries for your programming language. This is to ensure we don't have download anything extra when grading.

2.3 Example Execution

Imagine we have some text file which has encrypted text from the book "The Lord of the Rings: Fellowship Of The Ring" named *ciphertext.txt*



```
ciphertext.txt

'Xg zia trp nidkpa aixf: ep jxddit xzziaf trp
toqp. Ep qokrt gupdf x spxa od g jr x vi adps,
xdf ep gri nf uxgg trai kr qxds nxdfg trxt xap
pquts xdf rxami
...
```

We also have a file named *knownplaintext.txt* with a large amount of text from the book "The Lord of the Rings: Fellowship Of The Ring":

knownText.txt

```
'You speak of what you do not know, when you
liken Moria to the stronghold of Sauron,'
answered Gandalf. 'I alone of you have ever
been in the dungeons of the Dark Lord, and
only in his older and lesser dwelling in
Dol Guldur....'
```

Running our program should be like the following:

Command Line

```
$ python CrackCipher.py ciphertext.txt knownText.txt
'As for the longer road: we cannot afford the time. We might spend a
year in such a journey, and we should pass through many lands that are empty
and harbourless. Yet they would not be safe. The watchful eyes both of
Saruman and of the Enemy are on them. When you came north, Boromir, you
...
```

2.4 Testing your program

In order to test your program, provided should be a folder called **CipherSample** with a sample ciphertext along with its corresponding plaintext and known text.

1. **ciphertext-sample.txt** - a sample ciphertext encrypted using an arbitrary letter mapping
2. **knownText-sample.txt** - a text file relating to the original plaintext
3. **plaintext-sample.txt** - the correct plaintext that your program should output



Correctness: The correct output of your program should when decrypting ciphertext-sample.txt should be similar to the contents of plaintext-sample.txt. Do not worry about the case of the letters if your program outputs different letter cases, they will be ignored. Note that the sample ciphertext is merely a sample and will not be used when testing your program so make sure you implement your program correctly!

3 Smarter Cracking (Optional)

Like mentioned in the previous section, although monoalphabetic ciphers are vulnerable to frequency attacks, frequency matching letters may not be enough.

3.1 More in Depth Frequency Analysis

Instead of single letter frequency analysis, we can use analyze **N-grams**, or sets of N consecutive items (letters, syllables, words, etc.) to perform a deeper analysis and get more insights to help us deduce a suitable alphabet mapping. By analyzing sets of letters, we can try and extract more possible patterns from the texts. The more information we have the high chance of guessing the best alphabet mapping.

If perhaps you were able to determine the spaces between words, you can gain insight on possible letter mappings based on frequent words of a given length whose letters can possibly map to common words of the same length. In the end you can use all this information to find a suitable alphabet mapping.

Try to determine alphabet mappings using frequency analysis on one single letters, bi-grams, and tri-grams and see how accurate the mappings are for each. You can then try to combine all three to determine an alphabet mapping based on similar frequencies from each of the analysis.

You can learn more about N -grams here:

- **N-Grams:**
<https://en.wikipedia.org/wiki/N-gram>
- **N-Gram Frequency Counts:**
<http://practicalcryptography.com/cryptanalysis/text-characterisation/monogram-bigram-and-trigram-frequency-counts/>
- **Cryptoanalysis with N-Grams:**
<https://jeremykun.com/2012/02/03/cryptanalysis-with-n-grams/>

3.2 Markov Chain Monte Carlo Methods

Using the previously mentioned approaches for figuring out a alphabet mapping for some monoalphabetic substitution cipher are great but unfortunately we have only talked about using these frequencies to determine a single suitable mapping. It is quite possible that the mapping we determined using our implementations for frequency analysis may be wrong. In this case it may be helpful to use iterative methods to improve our initial guess. The class of Markov Chain Monte Carlo Methods could be useful in our case to find more suitable alphabet mappings based on frequencies of within the known text.

You can learn more about Markov Chain Monte Carlo Methods here:

- **Using MCMC Methods to break Classical Ciphers:**
<https://link.springer.com/article/10.1007/s11222-011-9232-5>
- **Markov Chain Monte Carlo:**
https://en.wikipedia.org/wiki/Markov_chain_Monte_Carlo
- **Simulation and Solving Substitution Codes:**
<http://www-users.york.ac.uk/~sbc502/decode.pdf>

3.3 An AI Approach

Another way we can approach finding the right alphabet mapping is to use an AI approach by turning our problem into a search problem. If we think of an alphabet mapping for a substitution cipher as a "key", our AI will be in charge of searching for a suitable key. In this case, searching neighbors would be searching similar alphabet mappings. This is a general approach and this program can be modeled in many different ways and it's up to you how you would want to model the search. It is important to note that you should use previous approaches to find a good alphabet mapping for your AI to start searching from. If you have experience in AI, this could be a great approach to try out.

You can learn more here:

- **Cryptoanalysis with N-Grams:**
<https://jeremykun.com/2012/02/03/cryptanalysis-with-n-grams/>
- **Automated Cryptoanalysis of Classical Ciphers:**
<http://what-when-how.com/artificial-intelligence/automated-cryptanalysis-of-classical-ciphers-artificial-intelligence/>

3.4 Program Implementation

The goal of this optional part is to try and create a new program called *SmartCrackCipher* that improve your original CrackCipher program in order for it to be able to handle imperfect frequency correlations. It takes in the same inputs at the CrackCipher program. Feel free to try any of the approaches above or come up with your own approaches.

3.5 Testing your Implementation

In order to test your program, provided should be a folder called **SmartSample** with a sample ciphertext along with its corresponding plaintext and known text.

1. **ciphertext-sample.txt** - a sample ciphertext encrypted using a arbitrary letter mapping
2. **known-text-sample.txt** - a text file relating to the original plaintext
3. **plaintext-sample.txt** - the correct plaintext that your program should output

These sample texts are based on the same sample texts from part 2 of this assignment. The only difference between these texts and the texts from part 2 is that the known text is not perfectly representative of the plaintext trying to be recovered. Since the known text is not perfectly representative of the plaintext, there will be a few incorrect letter mappings. Since the ciphertext is exact same ciphertext from part 2, you can use the correct letter mapping determined in part 2 to evaluate the accuracy of your SmartCrackCipher program.

The basic idea for evaluation would be as follows:

1. Run your SmartCrackCipher program against the ciphertext and known text provided.
2. Compare the letter mapping your SmartCrackCipher program generated to the correct letter mapping determined in part 2.

3.6 Write Up

Along with your implementation you should submit a **PDF** that answers the following questions:

1. Briefly describe the approach you took.
2. If you incorporated n-gram analysis:
 - What length of n-grams did you use and why?
3. If you incorporated an search/iterative approach:
 - What letter mapping does your program start off with? and how does your program iteratively try to find better letter mappings?
 - What metric did you use in order to grade letter mappings being searched. Another way to think about this question is, "How does your program figure out which letter mapping to keep or which one to reject?"
 - How does your program determine when to stop?
4. How did your program perform on the sample cipher text? Was Is it more accurate at recovering plaintext compared to straight frequency matching done in your CrackCipher program in part 2? If so, how much better? If not, why do you think that is? Do you think you can make it better?

4 Submission

To submit your project, simply submit your source code for CrackCaesar, CrackCipher, and SmartCrackCipher/write up (if you tried part 3) directly to sakai as is. **Do not compress your source code**, as it will be easier for us to grade. Also, do not upload any of the sample text files supplied, including the dictionary of common words. We will use different ones for grading.

5 Additional Notes and Hints



Querying the set of common words: There are many ways to manipulate and query the list of common words. You can search the text file every time you want to do the comparison or you can import the words into some data structure that is easy to go through. Ciphertexts may be lengthy so make sure to use a suitable data structure to make your program as efficient as possible.

(Hint: Dictionaries and Hashmaps offer fast lookups by using efficient indexing)



Be Careful When Comparing Strings: Take into account that the plaintext can be upper case and/or lower case. The same can be said for the words within the text file of common words. This can be problematic as, upper case letters and lower case letters have different ASCII values. Be sure to make strings are all upper-case or all lower-case when doing any string comparison.



When in doubt, Google: If you're having trouble figuring out something, Google it. Some other developer somewhere probably had the same issue as issue as you. There are plenty of guides you can look up to do all sorts of things! Look around and see what works for you!

6 Additional Questions

If you have any questions about the project or are having any issues, email me at David.Domingo@rutgers.edu