Part II
Monoalphabetic substitution ciphers

Unit 12
Monoalphabetic substitution cipher

A *substitution cipher* is one in which each character in the plaintext is replaced with another (sometimes the same) character. A *monoalphabetic* substitution cipher is one in which each letter is always replaced by the same letter. So ‘A’ is always replaced by the same thing, while ‘B’ is always replaced by the same letter, but different to the replacement for ‘A,’ etc. The *key* is the information needed to correctly decipher a ciphertext. In the case of the monoalphabetic substitution cipher, this key is a permutation of the alphabet. When we solve such ciphers by hand, we typically write the key under the alphabet like so:

plaintext:	`abcdefghijklmnopqrstuvwxyz`
ciphertext:	`JIOAFMBYXZESHRDWQLTCPUNKGV`

For this key, ‘A’ in the plaintext is always replaced by ‘J’ in the ciphertext, ‘B’ by ‘I,’ etc. The key is the ciphertext alphabet `JIOAFMBYXZESHRDWQLTCPUNKGV`. Notice that ‘Q’ is mapped into itself with this key.

The *key space* is the set of all possible keys (including the one that enciphers the plaintext into an exact copy of itself). For the monoalphabetic cipher, there are 26 ways to choose the first letter of the key. This leaves 25 choices for the second letter, and 24 remain for the third. We keep counting this way until there is only one choice remaining for the last letter. Thus, the size of the key space is $26 \times 25 \times 24 \times ... \times 1 = 26!$.

For decipherment, it is useful to be able to construct the inverse of the key. We do that by writing the key under the straight (unmodified) plaintext alphabet as we did above. Then we rearrange things so that the ciphertext is straight, while being careful to keep each pair of letters together. For our example above, we would get

plaintext:	`dgtokeymbaxrfwcuqnlsvzpihj`
ciphertext:	`ABCDEFGHIJKLMNOPQRSTUVWXYZ`

The inverse key is `DGTOKEYMBAXRFWCUQNLSVZPIHJ`. If we were to encipher a message with the key, and then again with the inverse key, we would obtain the original message.

How do we know if we have a ciphertext that was encrypted with a monoalphabetic substitution cipher? The index of coincidence will be close to that of English, and the monogram fitness will be low.

Python tips

The location of a letter in your key alphabet can be found with the `index()` function:

```
key = "ZYXWVUTS"
some_number = key.index("X")
```

The n^{th} letter in the key can be found by indexing:

```
some_letter = key[7]
```

Indexing and the `index()` function work for any string.

A character can be added to the end of a string like this:

```
message += "A"
```

Unfortunately, in Python, a character in a string cannot be modified. Instead, you must create a new string with the modification that you want.

You should never hard-code your texts into your scripts. Instead, you should input them on the command line or from a file. To use the command-line arguments, you need to import `argv` from the `sys` module. It is an array containing all of the strings on the command line. The first is `argv[0]`, which is the name of the script/program. The first argument after that is `argv[1]`, which can contain whatever you like. If you pass strings that contain spaces, you will need to put quotation marks around them.

Reading and references

Helen Fouché Gaines, *Cryptanalysis: a study of ciphers and their solution*, New York: Dover, 1956; previously titled *Elementary Cryptanalysis* and published by American Photographic in 1939; archive.org/details/cryptanalysis00gain; chapter IX.

James Lyons, “Simple Substitution Cipher,” Practical Cryptography website, practicalcryptography.com/ciphers/simple-substitution-cipher

Fletcher Pratt, *Secret and Urgent*, New York: Bobbs-Merrill, 1939, chapter III, section IV.

David Kahn, *The Codebreakers: The Story of Secret Writing*, New York: Simon & Schuster, 1967, revised and updated 1996, pages 97-105.

Programming tasks

You might want to put these functions together in a module that deals with monoalphabetic substitution ciphers.

1. Write a function that returns `True` or `False`, indicating whether a ciphertext is likely to have been encrypted with a monoalphabetic substitution cipher.
2. Write a function that inverts an alphabet key.
3. Write a function that enciphers a plaintext with an alphabet key and returns a ciphertext.
4. Write a function that deciphers a ciphertext with an alphabet key and returns a plaintext. Note that the key is the one used in *encipherment*.

Exercises

1. Use your functions to invert this key alphabet: `KNOFPEUCARBMGXQIZWYSTJVDHL`.
2. Use your functions to encipher the phrase “Sally sells seashells by the seashore” with the key `ATSPWGKHVXDLCJZEMFBRYUONIQ`.
3. Use your functions to decipher the message `PFAFIPGPFIPGKLFTCPFKLHVPGKLEFTPFPPFIZ` with the key `CSKTFVRMGQLEXDHPJIZANBOUWY`.

madness's book on classical cryptography
unit 13: atbash cipher
last modified 2020-10-08

Unit 13
Atbash cipher

The *atbash cipher* (formerly spelled *athbash*) is an ancient cipher in which the key alphabet is the reverse of the original:

plaintext:	`abcdefghijklmnopqrstuvwxyz`
ciphertext:	`ZYXWVUTSRQPONMLKJIHGFEDCBA`

Notice that the key is *reciprocal*, i.e., it is its own inverse, and therefore the cipher is also *reciprocal* so that encipherment and decipherment are the same process.

To use the atbash cipher, we can treat it like the general monoalphabetic substitution cipher with the key alphabet given above. Another way is to use simple arithmetic. Consider the plaintext as a sequence of symbols $\{p_i\} = p_0\ p_1\ p_2\ p_3\ \ldots$, and the ciphertext as $\{c_i\} = c_0\ c_1\ c_2\ c_3\ \ldots$. These symbols can be treated as numbers, so that ‘A’ = 0, ‘B’ = 1, ..., ‘Z’ = 25 (modern programmers usually count starting with zero). Then the atbash cipher can be implemented with this simple equation:

$$c_i = 25 - p_i$$

Notice that there is almost no security with the atbash cipher. When using it, your only hope of secrecy is your belief that your opponents do not know the cipher.

Another completely insecure cipher related to atbash is the *albam cipher*. It, too, is reciprocal. It uses this key, in which the two halves of the alphabet have been swapped:

plaintext:	`abcdefghijklmnopqrstuvwxyz`
ciphertext:	`NOPQRSTUVWXYZABCDEFGHIJKLM`

Reading and references

Wikipedia: en.wikipedia.org/wiki/Atbash

Crypto Corner: crypto.interactive-maths.com/atbash-cipher.html

Practical Cryptography: practicalcryptography.com/ciphers/atbash-cipher-cipher

David Kahn, *The Codebreakers: The Story of Secret Writing*, New York: Simon & Schuster, 1967, revised and updated 1996, pages 77-79.

Programming tasks

1. Write functions to encipher and decipher texts with the atbash cipher. Note that they are really the same function. Use either (or both) of the methods discussed above.

Exercises

1. What is the size of the key space for the atbash cipher?

2. Write a short message and encipher it with the atbash cipher. Now decipher it and check that you obtain the original message.

madness's book on classical cryptography
unit 14: modular arithmetic: addition and subtraction
last modified 2020-09-16

Unit 14
Modular arithmetic: addition and subtraction

Consider the integers, $\mathbb{Z}$ = { ..., −3, −2, −1, 0, 1, 2, 3, ... }. Now pick one, say 12, and call it the *modulus*. Next, rename each integer by the remainder when you divide by the modulus. So 0 ⟶ 0, 1 ⟶ 1, 2 ⟶ 2, 3 ⟶ 3, ..., 11 ⟶ 11. But 12 ⟶ 0, since 12 = 1 × 12 + 0, and the remainder is 0. The sequence 0, ..., 11 starts all over again, as 12 ⟶ 0, 13 ⟶ 1, ... The result is a new set $\mathbb{Z}_{12}$ = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11}, which contains only 12 members. This is called the set of *residues*. We say things like “13 modulo 12 is 1” or “13 = 1 modulo 12” or “13 = 1 mod 12.”

Things get interesting when we start to do arithmetic with the set of residues. For addition, we simply perform the operation as if the numbers are regular integers, but then take the remainder of the result. Continuing with our example of $\mathbb{Z}_{12}$, we can add 7 and 9 to get 16. But 16 is 4 modulo 12. So in $\mathbb{Z}_{12}$, 7 + 9 = 4.

You might think that subtraction works the same way as addition in modular arithmetic, and you would be correct, but it is worthwhile to look at subtraction in a more rigorous way. After all, there are no negative numbers in modular arithmetic. This approach will also help us when we try to understand division. First, let us generalize and take some modulus *m* without specifying its value. Notice that we have that

$$x + 0 = x$$

for all x in $\mathbb{Z}_m$. Because 0 does not change the value of a number under addition, we call 0 the *additive identity element*. Now, in regular arithmetic, $x - x = x + (-x) = 0$, so we call $-x$ the *additive inverse* of x. In the set of residues, however, we do not have negative numbers, so there is a *positive* number that takes the role of x’s inverse. We might write that number as “*-x*,” but that is just notation for additive inverse. What we call “subtraction” is actually addition using the additive inverse.

How do we find the additive inverse of a number x? Well, we want the number y such that

$$x + y = 0 \bmod m$$

Even though the modulus m itself does not appear in the set of residues, we can calculate y as

$$y = 0 - x = m - x \pmod m$$

In our example of $\mathbb{Z}_{12}$, the inverse of 7 is $12 - 7 = 5$, or $-7 = 5$, so that $7 + 5 = 12 = 0 \pmod{12}$.

Python tips

When dividing integers, the operator `//` gives the quotient, and the operator `%` gives the remainder. Both operators return an integer. For example, `13 // 5` is evaluated as 2, and `13 % 5` is evaluated as 3, since $13 = 2 \times 5 + 3$.

In programming, modular arithmetic is not as complicated as in the text above. Feel free to use the operators `+` and `-` as you normally would, but then modulate the result with the `%` operator.

```
answer = (7 + 9) % 12
answer = (7 - 9) % 12
```

Programming tasks

1. Go outside and take a walk. If it's raining, open a window so you can hear it while you play a game with someone else in your house.

Exercises

1. Check carefully your work on programming task #1 and re-do it if necessary.
2. Evaluate:
   a. 528147 + 790378 modulo 62
   b. 72177 − 162737 mod 81
3.
   a. Find an element of $\mathbb{Z}_{18}$ that is its own (additive) inverse.
   b. Find an element of $\mathbb{Z}_{19}$ that is its own inverse.
   c. What condition must we put on n such that $\mathbb{Z}_n$ has such an element?

Unit 15
Caesar shift cipher

The *Caesar shift cipher*, also known simply as the *Caesar cipher* or the *additive cipher*, can be viewed in two ways. Given its key *k*, which is a number from 0 to 25, we can think of the Caesar cipher as a monoalphabetic substitution in which the ciphertext alphabet is shifted left by *k* letters, with wrap-around. For example, if the key is 5, we have

plaintext:	`abcdefghijklmnopqrstuvwxyz`
ciphertext:	`FGHIJKLMNOPQRSTUVWXYZABCDE`

Another way to view the Caesar cipher is with modular arithmetic. The plaintext and ciphertext are equivalent to sequences of numbers $\{p_i\}$ and $\{c_i\}$. Encipherment is done with addition:

$$c_i = p_i + k \bmod 26$$

Decipherment is done with subtraction:

$$p_i = c_i - k \bmod 26$$

When the shift is 13, the cipher is often called *ROT13*. This shift has a special name because 13 is half of 26. Furthermore, since 13 is its own inverse modulo 26, this cipher is reciprocal, i.e., encipherment is the same process as decipherment. Also notice that ROT13 is the same as the albam cipher.

Reading and references

David Kahn, *The Codebreakers: The Story of Secret Writing*, New York: Simon & Schuster, 1967, revised and updated 1996, pages 83-84.

Wikipedia: en.wikipedia.org/wiki/Caesar_cipher

Crypto Corner: crypto.interactive-maths.com/caesar-shift-cipher.html

Practical Cryptography: practicalcryptography.com/ciphers/caesar-cipher

Programming tasks

1. Write a function to generate the key alphabet for a Caesar cipher from the key number. You might not use this function if you work with modular arithmetic, but it is good practice to write the function.

2. Write a function to encipher a plaintext with the Caesar cipher and a given key. Use whatever method you like.

3. Write a function to decipher a ciphertext with the Caesar cipher and a given key. Use whatever method you like.

Exercises

1. What is the size of the key space for the Caesar cipher?

2. Encipher this text with the key 11:

   > A horse is a horse, of course of course, and no one can talk to a horse, of course, unless, of course, that horse of course is the famous Mister Ed.

3. Decipher this text with the key 15:

```
WTLPHDCRTPAXIIATVGTTCQPAADURAPNVJBQNQJINDJHWDJASHTT
LWPIWTRPCSDIDSPNVJBQNWTRPCLPAZXCIDBPCNQDDZHLXIWWXHE
DCTNEPAEDZTNIDDPCSXUNDJWPKTPWTPGIIWTCVJBQNHPEPGIDUN
DJVJBQN
```

madness's book on classical cryptography
unit 16: brute-force attack on the caesar cipher
last modified 2020-08-21

Unit 16
Brute-force attack on the Caesar cipher

A *brute-force attack* involves trying every possible key until an acceptable decryption is found. Since the Caesar cipher has only 26 possible keys (even when we include the identity operation [that leaves the plaintext unaltered]), a brute-force attack takes very little time.

Programming tasks

1. Write a function that does a brute-force attack on a ciphertext that was encrypted with the Caesar cipher. Use your tetragram-fitness function to find the best possible decryption. Your function should return the key or the plaintext or both. Also write a wrapper around your function that accepts a ciphertext and prints the plaintext or writes it to a file.

Exercises

1. Perform a brute-force attack on this ciphertext:

   ```
   WLALYWPWLYWPJRLKHWLJRVMWPJRSLKWLWWLYZ
   ```

   Remember this text; we will see it again.

2. Perform a brute-force attack on this ciphertext:

   ```
   KLYJKPMCLDUFXAESCZFRSSZZADLEESPKZZ
   ```

   Remember this text; we will see it again.

madness's book on classical cryptography
unit 17: attacking the caesar cipher with cribs
last modified 2020-07-04

Unit 17
Attacking the Caesar cipher with cribs

A *crib* is a word or phrase that you have good reason to believe is contained in the plaintext. To use a crib to break a ciphertext that was encrypted with the Caesar cipher, we try to fit the crib in all possible positions in the ciphertext. For each position, we find the shift needed to decrypt that portion of the ciphertext to match each letter of the crib. If each letter requires the same shift, then we have a good candidate for the key. It is not guaranteed to be the true key, but if we find several candidates, then we can decipher the text with each and use tetragram fitness to pick out the best plaintext.

Let us look at a short example. Here is a piece of encrypted text that we promised you would see again (and you will see it again at least once more):

```
WLALYWPWLYWPJRLKHWLJRVMWPJRSLKWLWWLYZ
```

Let's try to break it with the crib `PIPER`. We line up the crib with the ciphertext at each position until we find one at which all the shifts are equal:

```
ciphertext:   W L A L Y W P W L Y W P J R ...

crib:         P I P E R _ _ _ _ _ _ _ _ _ ...
shifts:       7 3 11 7 7

crib:         _ P I P E R _ _ _ _ _ _ _ _ ...
shifts:         22 18 22 20 5

⋮

crib:         _ _ _ _ _ P I P E R _ _ _ _ ...
shifts:               7 7 7 7 7
```

We now have a good candidate for the key: 7. If we decipher with that key, we find that the plaintext looks like English, and we are satisfied that we have found the correct key.

Python tips

To take a “slice” of a string (a substring), we use indexing. So, for example, to make a substring that takes the fifth through tenth character:

```
newString = oldString[4:10]
```

Remember that we count from zero, so the fifth character has index 4. The last character that we want to include has index 9, but in Python we have to add one to the index when we specify the last place in a string or array. To duplicate a string (or an array), we also use indexing, but leave the indices out:

```
newString = oldString[:]
```

Programming tasks

1. Write a function or script that takes a ciphertext and a crib and breaks the ciphertext.

Exercises

1. Break this ciphertext with the crib `CUSTOM`.

   ```
   RCCXRLCZJUZMZUVUZEKFKYIVVGRIKJKYVWZIJKZJZEYRSZKVU
   SPSVCXZREJKYVJVTFEUSPKYVRHLZKREZREJNYFTRCCKYVDJVC
   MVJTVCKJREUKYVKYZIUSPKYVXRLCJKYVJVUZWWVIWIFDFEVRE
   FKYVIZECREXLRXVTLJKFDJREUCRNJKYVIZMVIXRIFEEVJVGRI
   RKVJKYVXRLCJWIFDKYVTVCKJKYVIZMVIJDRIEVREUJZVEVJVG
   RIRKVKYVSVCXZREJWIFDKYVFKYVIJ
   ```

2. Break this ciphertext. It may or may not include some of these words: `VICTORY SPAIN DISCO EUROVISION`.

   ```
   MFFTUEFUYQOMQEMDIMEQXQOFQPRADFTQRAGDFTFUYQMEPUOFM
   FADMZPIMEOAYUZSFAEBMUZFARUZUETFTQIMDAZFTQIMKTQIME
   YQFNKMYNMEEMPADERDAYOADPANMITATMPPQEQDFQPSQZQDMXB
   AYBQKFTQKUZRADYQPTUYFTMFUFIAGXPNQQMEUQEFFAFMWQFTQ
   OUFKMFZUSTFNQOMGEQFTQQZQYKTMPNKFTQZZAWZAIXQPSQARP
   UEOAADARFTQZUSTFXURQADARFTQNAASUQ
   ```

Unit 18 (optional)
Attacking the Caesar cipher with monogram frequencies (χ^2)

Here is an attack on the Caesar cipher that uses only monogram frequencies. The technique is to make a table of the monogram frequencies from the ciphertext, then to shift that frequency table until it resembles that of typical English. In this unit, we will use monogram fitness based on the the χ^2 statistic to measure how well the frequencies match.

Let's work through an example. Consider this ciphertext:

```
LIKHKDGDQBWKLQJFRQILGHQWLDOWRVDBKHZURWHLWLQFLSKHUWKDWLVE
BVRFKDQJLQJWKHRUGHURIWKHOHWWHUVRIWKHDOSKDEHWWKDWQRWDZRUG
FRXOGEHPDGHRXWLIDQBRQHZLVKHVWRGHFLSKHUWKHVHDQGJHWDWWKHLU
PHDQLQJKHPXVWVXEVWLWXWHWKHIRXUWKOHWWHURIWKHDOSKDEHWQDPHO
BGIRUDDQGVRZLWKWKHRWKHUV
```

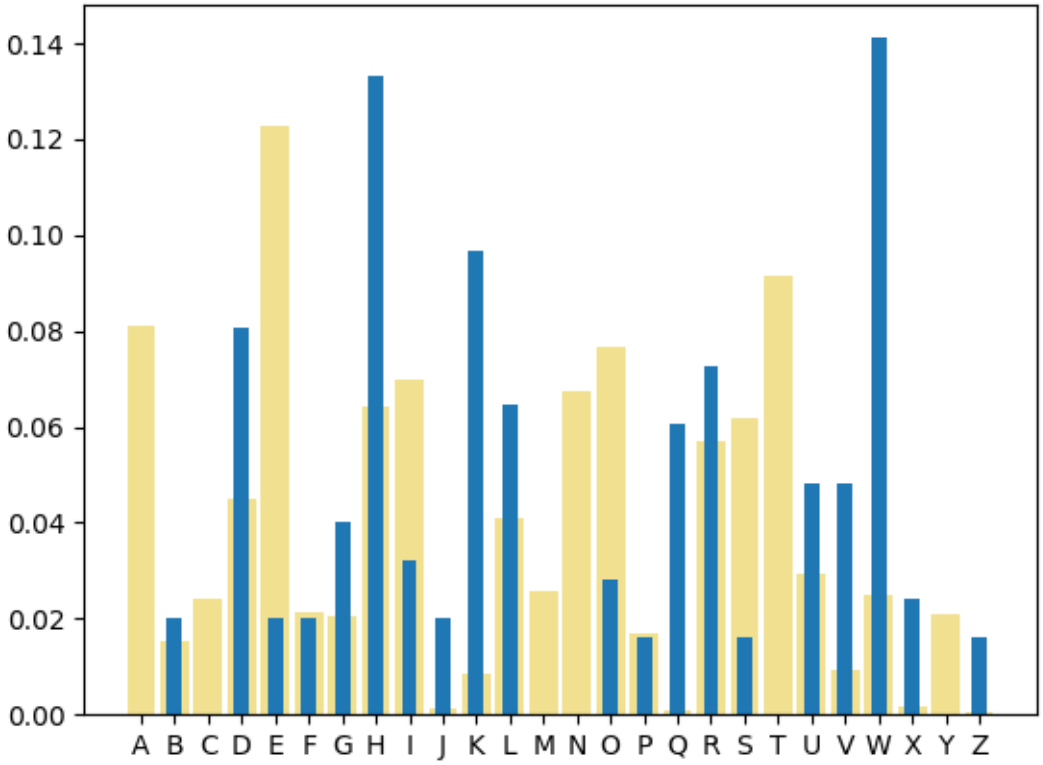
The frequencies of letters in this ciphertext are

A	0	J	0.020	S	0.016
B	0.020	K	0.097	T	0
C	0	L	0.065	U	0.048
D	0.081	M	0	V	0.048
E	0.020	N	0	W	0.141
F	0.020	O	0.028	X	0.024
G	0.040	P	0.016	Y	0
H	0.133	Q	0.060	Z	0.016
I	0.032	R	0.073		

Let's write the table as a list:

0.000, 0.020, 0.000, 0.081, 0.020, 0.020, 0.040, 0.133, 0.032, 0.020, 0.064, 0.065, 0.000, 0.000, 0.028, 0.016, 0.060, 0.073, 0.016, 0.000, 0.048, 0.048, 0.141, 0.024, 0.000, 0.016

Here is a graph of those frequencies (in **blue**), along side typical English frequencies (in yellow):



We can already see that a shift of the blue bars left by three will give a good match. But let's use the χ^2 statistic and see how that works. Our monogram table for English looks like this (yours may be slightly different) (rounded to three decimal places):

0.081, 0.015, 0.024, 0.045, 0.123, 0.021, 0.021, 0.064, 0.070, 0.001, 0.009, 0.041, 0.016,
0.068, 0.077, 0.017, 0.001, 0.057, 0.062, 0.091, 0.029, 0.009, 0.025, 0.002, 0.021, 0.001

Now, if we shift the frequencies from the ciphertext leftward by one, we get

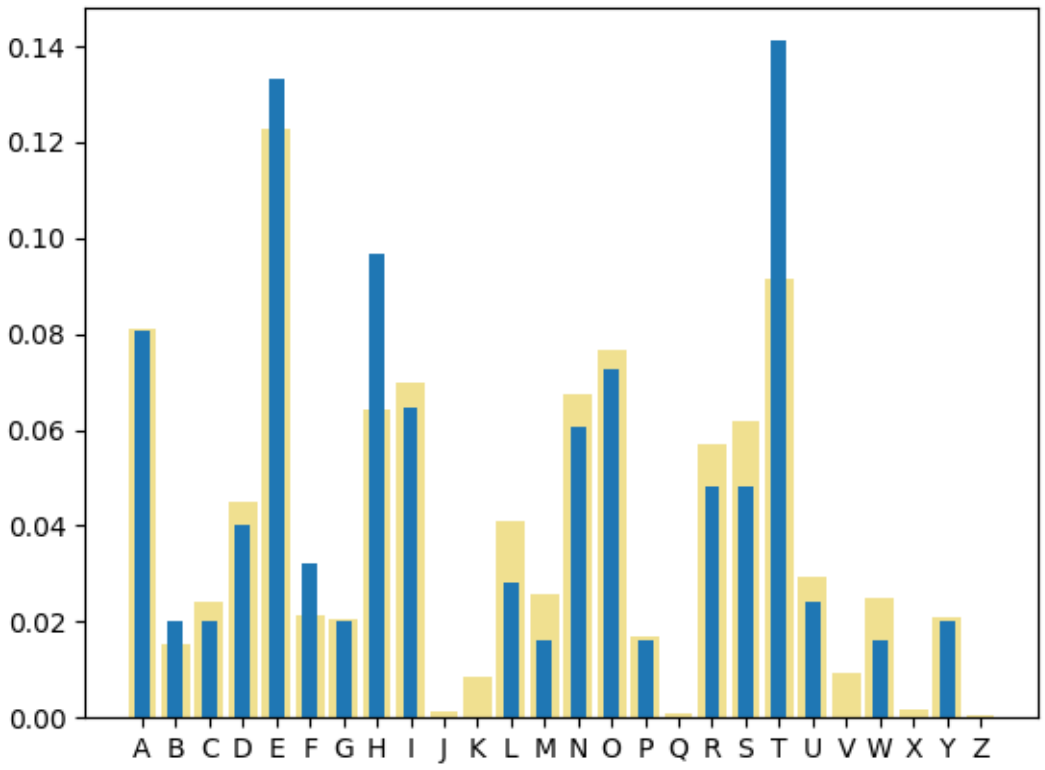
0.020, 0.000, 0.081, 0.020, 0.020, 0.040, 0.133, 0.032, 0.020, 0.064, 0.065, 0.000, 0.000,
0.028, 0.016, 0.060, 0.073, 0.016, 0.000, 0.048, 0.048, 0.141, 0.024, 0.000, 0.016, 0.000

The χ^2 between this and the English table is 18.02, indicating a poor fit. We can continue in this manner, and for each shift we obtain these values for χ^2:

shift	χ^2
0	6.51
1	18.02
2	6.73
3	0.10
4	9.81
5	5.17
6	25.14
7	7.67
8	24.11

9	3.91
10	10.27
11	13.92
12	10.99
13	31.47
14	3.91
15	2.90
16	3.91
17	22.50
18	7.88
19	3.83
20	19.38
21	9.52
22	4.00
23	24.56
24	21.31
25	11.63

The best fit (lowest χ^2) is for a shift of three. We conclude that the key is 3. To get a feel for what a good fit looks like, here is a graph of the shifted frequency table compared to typical English:



Programming tasks

1. Write a function or script that implements the attack. Make sure that you shift the table in the correct direction; generate some ciphertexts and test it to be certain.

Exercises

1. Finish decrypting the ciphertext in the example above.
2. Apply the attack to this ciphertext:

```
QEBZXBPXOZFMEBOFPKXJBAXCQBOGRIFRPZXBPXOTELXZZLOAFKDQLP
RBQLKFRPRPBAFQTFQEXPEFCQLCQEOBBQLMOLQBZQJBPPXDBPLCJFIF
QXOVPFDKFCFZXKZBTEFIBZXBPXOPTXPQEBCFOPQOBZLOABARPBLCQE
FPPZEBJBLQEBOPRYPQFQRQFLKZFMEBOPXOBHKLTKQLEXSBYBBKRPBA
BXOIFBOFQFPRKHKLTKELTBCCBZQFSBQEBZXBPXOZFMEBOTXPXQQEBQ
FJBYRQFQFPIFHBIVQLEXSBYBBKOBXPLKXYIVPBZROBKLQIBXPQYBZX
RPBJLPQLCZXBPXOPBKBJFBPTLRIAEXSBYBBKFIIFQBOXQB
```

3. Apply the attack to this ciphertext, which we have seen before:

```
WLALYWPWLYWPJRLKHWLJRVMWPJRSLKWLWWLYZ
```

   You should find that the attack fails. Why did it fail?

4. Apply the attack to this ciphertext, which we have seen before:

```
KLYJKPMCLDUFXAESCZFRSSZZADLEESPKZZ.
```

   You should find that the attack fails. Why did it fail? Now you should understand why we do not recommend using the χ^2 statistic for cracking ciphertexts.

Unit 19
Attacking the Caesar cipher with monogram frequencies

Here is an attack on the Caesar cipher that uses only monogram frequencies. The technique is to make a table of the monogram frequencies from the ciphertext, then to shift that frequency table until it resembles that of typical English. In this unit, we will use monogram fitness based on the cosine of the angle between vectors to measure how well the frequencies match.

Let’s work through an example. Consider this ciphertext:

```
LIKHKDGDQBWKLQJFRQILGHQWLDOWRVDBKHZURWHLWLQFLSKHUWKDWLVE
BVRFKDQJLQJWKHRUGHURIWKHOHWWHUVRIWKHDOSKDEHWWKDWQRWDZRUG
FRXOGEHPDGHRXWLIDQBRQHZLVKHVWRGHFLSKHUWKHVHDQGJHWDWWKHLU
PHDQLQJKHPXVWVXEVWLWXWHWKHIRXUWKOHWWHURIWKHDOSKDEHWQDPHO
BGIRUDDQGVRZLWKWKHRWKHUV
```

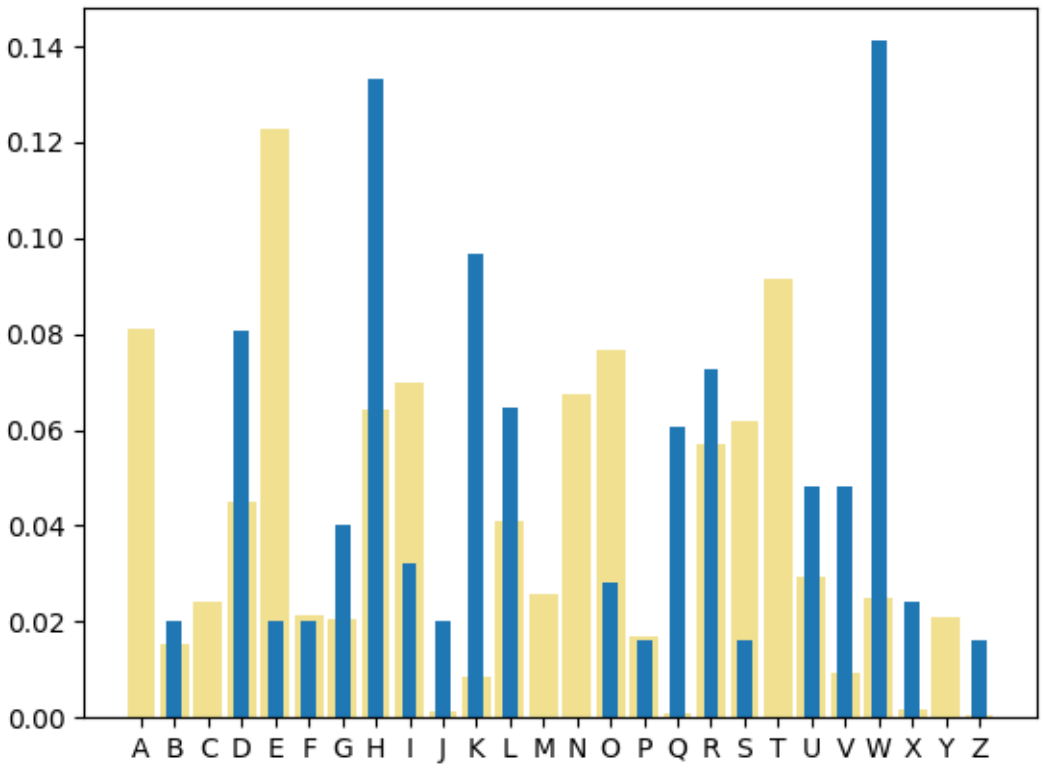
The frequencies of letters in this ciphertext are

A	0	J	0.020	S	0.016
B	0.020	K	0.097	T	0
C	0	L	0.065	U	0.048
D	0.081	M	0	V	0.048
E	0.020	N	0	W	0.141
F	0.020	O	0.028	X	0.024
G	0.040	P	0.016	Y	0
H	0.133	Q	0.060	Z	0.016
I	0.032	R	0.073		

Let’s write the table as a 26-dimensional vector:

(0.000, 0.020, 0.000, 0.081, 0.020, 0.020, 0.040, 0.133, 0.032, 0.020, 0.064, 0.065, 0.000, 0.000, 0.028, 0.016, 0.060, 0.073, 0.016, 0.000, 0.048, 0.048, 0.141, 0.024, 0.000, 0.016)

Here is a graph of those frequencies (in **blue**), along side typical English frequencies (in yellow):



We can already see that a shift of the blue bars left by three will give a good match. But let's use the monogram fitness and see how that works. Our monogram table for English looks like this (yours may be slightly different) (rounded to three decimal places):

(0.081, 0.015, 0.024, 0.045, 0.123, 0.021, 0.021, 0.064, 0.070, 0.001, 0.009, 0.041, 0.016,
0.068, 0.077, 0.017, 0.001, 0.057, 0.062, 0.091, 0.029, 0.009, 0.025, 0.002, 0.021, 0.001)

Now, if we shift the frequencies from the ciphertext leftward by one, we get

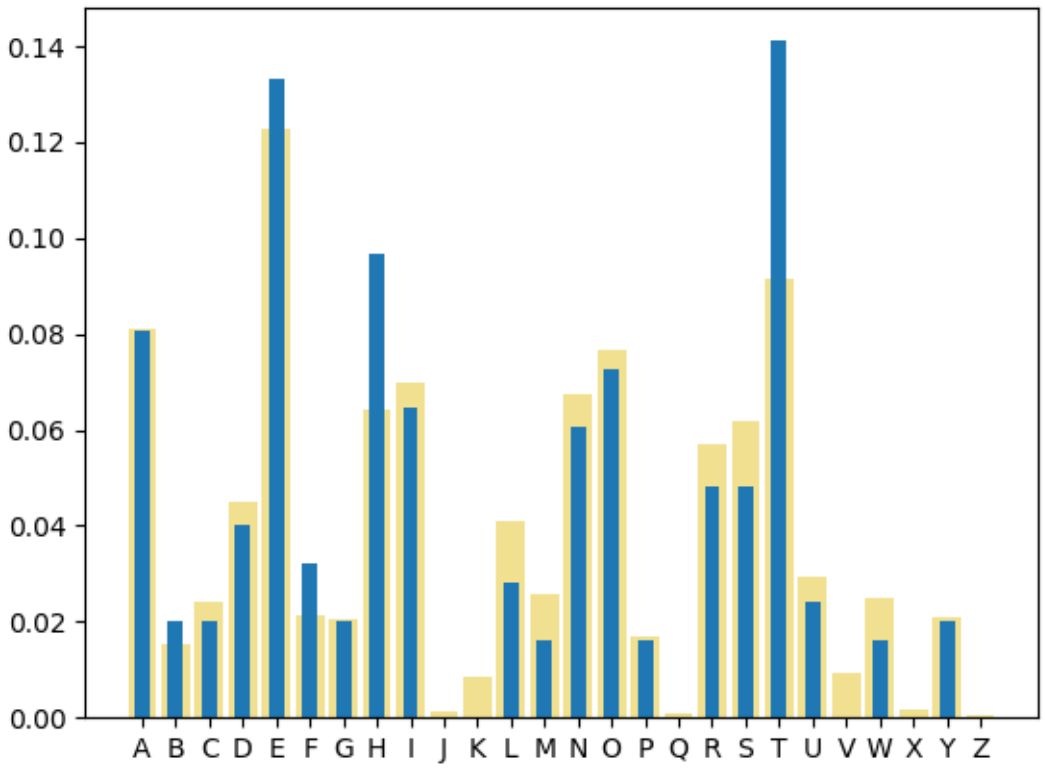
(0.020, 0.000, 0.081, 0.020, 0.020, 0.040, 0.133, 0.032, 0.020, 0.064, 0.065, 0.000, 0.000,
0.028, 0.016, 0.060, 0.073, 0.016, 0.000, 0.048, 0.048, 0.141, 0.024, 0.000, 0.016, 0.000)

The cos θ between this vector and the English table is 0.393, indicating a poor fit. We can continue in this manner, and for each shift we obtain these values for cos θ:

shift	cos θ
0	0.493
1	0.393
2	0.544
3	0.970
4	0.593
5	0.443
6	0.485
7	0.589
8	0.493

9	0.512
10	0.547
11	0.452
12	0.434
13	0.478
14	0.648
15	0.580
16	0.584
17	0.550
18	0.693
19	0.569
20	0.443
21	0.435
22	0.608
23	0.499
24	0.444
25	0.600

The best fit (largest cos θ) is for a shift of three. We conclude that the key is 3. To get a feel for what a good fit looks like, here is a graph of the shifted frequency table compared to typical English:



Programming tasks

1. Write a function or script that implements the attack. Make sure that you shift the table in the correct direction; generate some ciphertexts and test it to be certain.

Exercises

1. Finish decrypting the ciphertext in the example above.
2. Apply the attack to this ciphertext:

```
QEBZXBPXOZFMEBOFPKXJBAXCQBOGRIFRPZXBPXOTELXZZLOAFKDQLP
RBQLKFRPRPBAFQTFQEXPEFCQLCQEOBBQLMOLQBZQJBPPXDBPLCJFIF
QXOVPFDKFCFZXKZBTEFIBZXBPXOPTXPQEBCFOPQOBZLOABARPBLCQE
FPPZEBJBLQEBOPRYPQFQRQFLKZFMEBOPXOBHKLTKQLEXSBYBBKRPBA
BXOIFBOFQFPRKHKLTKELTBCCBZQFSBQEBZXBPXOZFMEBOTXPXQQEBQ
FJBYRQFQFPIFHBIVQLEXSBYBBKOBXPLKXYIVPBZROBKLQIBXPQYBZX
RPBJLPQLCZXBPXOPBKBJFBPTLRIAEXSBYBBKFIIFQBOXQB
```

3. Apply the attack to this ciphertext, which we have seen before:

```
WLALYWPWLYWPJRLKHWLJRVMWPJRSLKWLWWLYZ
```

   You should find that the attack fails. Why did it fail? When we used the brute-force attack, we succeeded. From this we learn that using monogram fitness is useful for longer ciphertexts, but not very reliable for shorter texts. Nevertheless, this attack is necessary if you have a ciphertext that has been encrypted with both a Caesar cipher and a transposition cipher, which rearranges the letters of the text and thereby makes it impossible to use tetragram fitness.

4. Apply the attack to this ciphertext, which we have seen before:

```
KLYJKPMCLDUFXAESCZFRSSZZADLEESPKZZ.
```

   You should find that the attack succeeds. When we used the monogram fitness based on the χ^2 statistic, we failed to decrypt this ciphertext.

Unit 20
Greatest common divisor

The *greatest common divisor* (gcd) of two integers is the largest integer that divides them both evenly (without remainder).

One way to find the gcd of two integer is to write out the prime-number factorization of each and select the largest set of factors that is contained in both. For example,

$$84 = 2 \times 2 \times 3 \times 7$$
$$360 = 2 \times 2 \times 2 \times 3 \times 3 \times 5$$

The largest subset contained in both factorizations is {2, 2, 3}, so the gcd of 84 and 360 is $2 \times 2 \times 3 = 12$.

Another way to find the gcd is with Euclid's algorithm, which is easy to implement in a script. Here is how the algorithm works for two integers *m* and *n*:

1. while *n* is not 0
    a. set *m* equal to *m* modulo *n*
    b. swap *m* and *n*
2. once *n* is 0, output *m*

The *least common multiple* (lcm) of two integers is the smallest number that is a multiple of both of them. We can find the lcm from the prime-number factorizations as well, by selecting the smallest set of factors that is a superset of each factorizations. For our example of 84 and 360, we see that we need three 2's to accommodate the 2's in 360, two 3's, one 5 and one 7. So the lcm of 84 and 360 is $2 \times 2 \times 2 \times 3 \times 3 \times 5 \times 7 = 2520$. An interesting fact is that

$$\gcd(m, n) \times \text{lcm}(m, n) = m \times n$$

We say that two numbers are *coprime* if their gcd is one. In this case, the two numbers have no factors in common, and neither one evenly divides the other.

Python tips

Python has a built-in limit on the number of levels of recursion. If you are working with large numbers, and trying to find a gcd, then you could exceed this limit and get an error. (Recursion is when a function calls itself. When some condition becomes true, the function exits all the way back to the main control flow.) We recommend that you implement Euclid's algorithm without recursion.

Python allows variables to hold boolean values. These values are either `True` or `False`. Furthermore, functions can return boolean values. Notice that the following two short code blocks are equivalent; both return `True` if the value of `x` is one and `False` otherwise.

code block 1:

```
if x == 1:
      return True
else:
      return False
```

code block 2:

```
return (x == 1)
```

Reading and references

Wikipedia

en.wikipedia.org/wiki/Greatest_common_divisor
en.wikipedia.org/wiki/Euclidean_algorithm
en.wikipedia.org/wiki/Least_common_multiple

Programming tasks

1. Write a function that calculates the gcd of two integers. Use whatever method you prefer.
2. Write a function that finds the lcm of two integers. Feel free to use your function for the gcd.
3. Write a function that returns a boolean value that indicates whether two integers are coprime. This can be a very short function.

Exercises

1. Randomly generate a few pairs of positive integers. For each pair, find the gcd and lcm. Verify that the product of the gcd and lcm equals the product of the pair.

Unit 21
Modular arithmetic: multiplication and division

Let's return to our example of $\mathbb{Z}_{12} = \{0, 1, ..., 11\}$, and construct a multiplication table. Remember that what we need to do is find the product of two members of the set, and then find the remainder when we divide the product by the modulus 12.

	0	1	2	3	4	5	6	7	8	9	10	11
0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6	7	8	9	10	11
2	0	2	4	6	8	10	0	2	4	6	8	10
3	0	3	6	9	0	3	6	9	0	3	6	9
4	0	4	8	0	4	8	0	4	8	0	4	8
5	0	5	10	3	8	1	6	11	4	9	2	7
6	0	6	0	6	0	6	0	6	0	6	0	6
7	0	7	2	9	4	11	6	1	8	3	10	5
8	0	8	4	0	8	4	0	8	4	0	8	4
9	0	9	6	3	0	9	6	3	0	9	6	3
10	0	10	8	6	4	2	0	10	8	6	4	2
11	0	11	10	9	8	7	6	5	4	3	2	1

That wasn't so hard. But what about division? Fractions do not exist in $\mathbb{Z}_{12}$, so what are we to do? In the same way that we defined the additive inverse, we must now do the same for multiplication. The *multiplicative inverse* of some number x is another number y such that $x \cdot y = 1$. The *multiplicative identity element* is 1. We write "x^{-1}" for the inverse of x, but we do not intend that it should be interpreted as "1 over x" or "1 divided by x." From this point of view, division is actually multiplication by an inverse.

If we look at the multiplication table for $\mathbb{Z}_{12}$ we can see, for example, that $5 \times 5 = 1$, so the inverse of 5 is 5 itself. But what about the inverse of 3? There is no number y such that $3 \cdot y = 1$. In this case, we are forced to say that 3 does not have an inverse. This means that we cannot divide by three in $\mathbb{Z}_{12}$. But what if we try to do it anyway? Notice that $3 \times 3 = 9$, and $3 \times 7 = 9$, and $3 \times 11 = 9$. Then if we

want to find 9 / 3, we have to face the fact that there are three possible answers. Since the quotient is not well defined, it cannot exist.

In general, an element x of $\mathbb{Z}_m$ has an inverse if and only if gcd (x, m) = 1. This means that if m is prime, all elements of $\mathbb{Z}_m$ are invertible.

To find the inverse of an element in $\mathbb{Z}_m$ (or to determine whether it exists), Euclid comes to the rescue again. His *extended Euclidean algorithm* for finding the inverse of x in $\mathbb{Z}_m$ is presented here:

1. set $t = 0$, $t' = 1$, $r = m$, $r' = x$
2. while r' is nonzero
   a. set $q = r / r'$ (discard the remainder)
   b. set new values of t and t' to the current values of t' and $t - q \cdot t'$ (respectively)
   c. set new values of r and r' to the current values of r' and $r - q \cdot r'$ (respectively)
3. if r is nonzero, then x is not invertible; exit
4. if $t < 0$, add m to t
5. t is the inverse of x

Python tips

Python allows parallel assignment. For example, to set `x` to 1 and `y` to old value of `x`, this one statement suffices:

```
x, y = 1, x
```

This technique should make implementing steps 2b and 2c in Euclid's extended algorithm easier.

Reading and references

en.wikipedia.org/wiki/Extended_Euclidean_algorithm

Programming tasks

1. Write a function that finds the multiplicative inverse (if it exists) of an integer with a given modulus. If the inverse does not exist, it should have a way of telling the program that called the function that it does not; one way is to return the boolean value `False`.

Exercises

1. Randomly generate some integer pairs such that the first of each pair is smaller than the second. For each pair find the multiplicative inverse of the first number modulo the second.

Unit 22
Affine cipher

The *affine cipher* extends the Caesar cipher to include the use of modular multiplication. When we treat the characters of the plaintext $\{p_i\}$ and ciphertext $\{c_i\}$ as integers modulo 26, then encipherment under the affine cipher is done with this equation:

$$c_i = a\ p_i + b \bmod 26$$

Decipherment is accomplished by solving the above equation for p_i:

$$p_i = a^{-1}\ (c_i - b) \bmod 26$$

where the key consists of the multiplier a and the shift b. Note that a must invertible, so gcd (a, 26) must be equal to one.

Another way to handle the affine cipher is to construct a key alphabet and treat the cipher as a monoalphabetic substitution (which it *is*). It's not complicated. Start with 'A' and take every a^{th} letter; then shift (with rollover) so that the letter whose number is b is at the beginning. For example, if a = 3 and b = 5, write down every third letter, starting with 'A':

```
ADGJMPSVYBEHKNQTWZCFILORUX
```

Then, shift so that the fifth letter (starting from zero) of the normal alphabet is at the beginning:

```
FILORUXADGJMPSVYBEHKNQTWZC
```

There are some special cases of the affine cipher:

- a = 1, b = 0: the identity operation
- a = 1: Caesar shift cipher with key = b
- a = 25, b = 25: atbash cipher
- b = 0: this is called the *multiplicative cipher*

Reading and references

en.wikipedia.org/wiki/Affine_cipher

practicalcryptography.com/ciphers/affine-cipher

crypto.interactive-maths.com/affine-cipher.html

Programming tasks

1. Write a function that verifies a key for the affine cipher and returns a boolean value indicating whether the key is valid or invalid.
2. Write a function that takes a key for an affine cipher and returns an equivalent alphabet key for a monoalphabetic substitution. This function should first call the function from the previous task to verify the key before proceeding.
3. Write a function that enciphers a plaintext with the affine cipher and a given key. Use whatever method you prefer. You should verify the key before proceeding.
4. Write a function that deciphers a ciphertext with the affine cipher and a given key. Use whatever method you prefer. You should verify the key before proceeding.

Exercises

1. How large is the key space for the affine cipher? Include only valid keys for which the multiplier is invertible. Include the identity operation (the key that leaves the plaintext unchanged under encipherment).
2. Encipher this text with multiplier 11 and shift 9:

   > An affine transformation is an automorphism of an affine space which preserves both the dimension of any affine subspaces and the ratios of the lengths of parallel line segments. Consequently, sets of parallel affine subspaces remain parallel after an affine transformation. An affine transformation does not necessarily preserve angles between lines or distances between points, though it does preserve ratios of distances between points lying on a straight line.

3. Decipher this text with multiplier 15 and shift 3:

   ```
   TSLCZFRCEFRPECCETNUMDTQCLKCVFRMWTQGFMGLNFBLBDCELBDCTHDMQFQNLQ
   NLCEDCQFFQLRQWLYNCDQWNTQNCLDWELYLTNDIFXLVEZWTWCELHETHXLQHYFNN
   CELBFLSTRNNCYTUCFPLCCFCELNDBLNTWL
   ```

4. If we extend the alphabet with a space character so that it now contains 27 letters, how many valid keys are there?

madness's book on classical cryptography
unit 23: brute-force attack on the affine cipher
last modified 2020-07-04

Unit 23
Brute-force attack on the affine cipher

The brute-force attack on the affine cipher tries all possible keys and chooses the deciphered plaintext with the best textual fitness.

Programming tasks

1. Implement the attack. Use tetragram fitness. Be sure to try only valid keys.

Exercises

1. Break this ciphertext:

```
HAGDGKDJKQNDFICEIVZOSABBNWIJIDIBWIDIBWSTITIDWTABD
JKPDNIDGDIJDWZTJSYDNKGDJSPKCPSJDIFSIJZDNKGDKVOGNK
PDNWYIDWUIGIYKQNDOGIKBKVYIVDNWGEKPPWJFJIXWIVZGAJW
TKXWPIGGWVQWJGGWDGIKBDNIDZIOTSJIDNJWWNSAJDSAJIDNJ
WWNSAJDSAJDNWUWIDNWJGDIJDWZQWDDKVQJSAQNDNWDKVOGNK
PUIGDSGGWZKTVSDTSJDNWCSAJIQWSTDNWTWIJBWGGCJWUDNWY
KVVSUUSABZFWBSGDDNWYKVVSUUSABZFWBSGDDNWGNKPGWDQJS
AVZSVDNWGNSJWSTDNKGAVCNIJDWZZWGWJDKGBWUKDNQKBBKQI
VDNWGEKPPWJDSSDNWYKBBKSVIKJWIVZNKGUKTWDNWYSXKWGDI
JDNWPJSTWGGSJIVZYIJOIVVNWJWSVQKBBKQIVGKGBW
```

madness's book on classical cryptography
unit 24: attacking the affine cipher with cribs
last modified 2020-07-22

Unit 24
Attacking the affine cipher with cribs

If we can match two letters from the ciphertext with two letters from the plaintext, then it is possible to solve for the multiplier and shift of the affine cipher. If all of the letters in a crib can be found from a section of the ciphertext with the same multiplier and shift, then we have found a candidate key for the cipher. This resembles the attack on the Caesar cipher, but whereas the Caesar cipher has only one parameter, the affine cipher has two.

Let's run through a short example to see how the algebra works. Suppose our crib is `CRIB`, and that we are matching it to the sequence `KFIT` in the ciphertext. First, we replace the letters with their numbers, where we begin the alphabet at 'A' = 0.

`C`	→	2	`K`	→	10
`R`	→	17	`F`	→	5
`I`	→	8	`I`	→	8
`B`	→	1	`T`	→	19

These give us four equations involving the parameters a and b of the affine cipher's key:

$$\begin{aligned} 2a + b &= 10 \mod 26 \\ 17a + b &= 5 \mod 26 \\ 8a + b &= 8 \mod 26 \\ a + b &= 19 \mod 26 \end{aligned}$$

Suppose we subtract the first equation from the second to get

$$15a = -5 = 21 \mod 26$$

The multiplicative inverse of 15 modulo 26 is 7, so we multiply both sides by 7:

$$\begin{aligned} 7 \cdot 15a &= 7 \cdot 21 \mod 26 \\ a &= 17 \end{aligned}$$

Then, from the last of our four original equations, $b = 2$. If we encipher `CRIB` with this key, we do indeed get `KFIT`, so we have a good candidate for the cipher's key. Finally, notice that if we had subtracted the first equation from the third, we would have found

$$6a = -2 = 24 \mod 26$$

but we cannot remove the coefficient of a because 6 is not invertible modulo 26.

Programming tasks

1. Write a function or script that attacks a ciphertext encrypted with an affine cipher by using a crib. Be careful that you only try to remove coefficients that are invertible.

Exercises

1. Try your program on this ciphertext. A good crib is `CRIB`.

```
OYFSTGLYYRSBXPTCLLIRSBSLZANYSGYNXXPFYXTONWRTVYRAYLDJRYLQ
OWLBLSOCLSLTTFSQIYLVRTRNSNGFSLUICNTRELNYQSFSVLQRTINTFCOL
VWSRVRFSRGRGFRCLQWLZNJCQZFHLJIZNJCQSOBNYRBWOAFVHONTCLLIF
SQNGOLSIYNOLTOLQCNJQCP
```

Unit 25
Attacking the affine cipher with monogram frequencies

This attack is very fast, but requires a ciphertext that is lengthy enough to do the required statistics. It is also a necessary attack if we are confronted with a ciphertext that was encrypted with an affine cipher and a transposition cipher (which changes positions of characters so that tetragram fitness is useless), since it can break the affine cipher so that we can then break the transposition cipher.

The method of attack begins by tabulating the monogram frequencies in the ciphertext. They are then shuffled in the same way that the key alphabet is generated in the affine cipher for given multiplier and shift. The multiplier and shift that give the best match to English frequencies is taken as the cipher's key. Armed with the key, we can then decipher the text.

The shuffling of the frequencies can be done by replacing the n^{th} entry in the table with the $(an + b)^{\text{th}}$ entry (modulo 26), where a and b are the multiplier and shift of the affine cipher.

Programming tasks

1. Implement the attack.

Exercises

1. Use your implementation to break this ciphertext:

   ```
   ZTYWFFAJYKAXTYLASNYLIKQJSZLWAJYRWJRZTYLYFQLYYWSIZQDLYWO
   UAZTSZWZASZAKSYNYJUAZTWZYBZZTASSTQLZ
   ```

2. The following ciphertext was encrypted with an affine cipher and a transposition cipher in which each block of three letters has been reversed. Use your implementation of the attack to break the affine cipher, then reverse each three-letter block to reveal the plaintext.

   ```
   MNRVVCMPWHWUZMNJIWKZMPYUZRIPWCCDMRDPZMNYVMMMZKICLYRCMZ
   MSOKMZMAYMHBWRUCEPVWDZMVRPMCXHRPWRBM
   ```

madness's book on classical cryptography
unit 26: keyword substitution cipher
last modified 2020-10-09

Unit 26
Keyword substitution cipher

The *keyword substitution cipher*, also called the *keyed substitution cipher*, or simple the *keyword cipher*, is a monoalphabetic substitution cipher in which the key alphabet is constructed from a keyword. The keyword is placed at the beginning of the key, its repeated letters are removed, and then the remainder of English letters are added to the key. The three most common ways of filling the key are these:

- Add the remaining letters in alphabetical order. For example, if the keyword is `AUTOMOBILE`, then the key alphabet is (remember to drop the repeated ‘O’)

  `AUTOMBILECDFGHJKNPQRSVWXYZ`

- Start adding letters from the alphabetically next after the last letter of the keyword. For the keyword AUTOMOBILE, we start with the next letter after ‘E,’ which is ‘F.’ When we reach ‘Z’ we place the remaining missing letters ‘C’ and ‘D.’

  `AUTOMBILEFGHJKNPQRSVWXYZCD`

- Start adding letters from the next letter after the alphabetically last letter of the keyword. The keyword `AUTOMOBILE` has ‘U’ as its alphabetically last letter. So we fill in starting with ‘V.’

  `AUTOMBILEVWXYZCDFGHJKNPQRS`

Here are some other variations. They can be used alone or in combinations.

- Fill in the remaining letters after the keyword in reverse alphabetical order.
- Put the keyword at the end of the key.
- Use a keyword for the plaintext alphabet rather than the ciphertext alphabet.
- Use keywords for both the plaintext and ciphertext alphabets.

Reading and references

David Kahn, *The Codebreakers: The Story of Secret Writing*, New York: Simon & Schuster, 1967, revised and updated 1996, pages 103-104.

Programming tasks

1. Write a function to generate an alphabet key from a keyword. It should be able to do so in the three most common ways, and there should be some way to tell it which method to use.

2. Write a function to encipher a plaintext when given a keyword and a key-filling method.

3. Write a function to decipher a ciphertext when given a keyword and a key-filling method.

4. Write a function to generate an alphabet key if the keyword is used in the plaintext alphabet rather than the ciphertext alphabet.

5. Write a function to generate an alphabet key if keywords are used in both the plaintext and ciphertext alphabets. Note that you can combine tasks 1, 4, and 5 into one function if you like. Make new versions of your enciphering and deciphering functions that can handle keywords for both alphabets.

Exercises

1. Encipher this text with the keyword `KNIGHTS`. Use the first key-filling method that we discussed above.

   > Before Cai was born, Cynyr made the prophesy that his son would have a frozen heart and be extremely stubborn. He added the prediction that no one would be able to endure fire or water as well as his son.

2. Decipher this text with the keyword `ROUNDTABLE`. Use the second key-filling method that we discussed above.

   ```
   AVLIDWDPDAVLIDWDPDXBLSBDPQBRGGLAJHVQSLDINVPDQJVSSDPGZRGJ
   IDSJNPRAHZGLTDJVSBRVISDNDWDPHJPDOZKRGDRUUVQLIAQKDUSPDQJT
   NDRNFILABSQLURIIJSGDRWDSBDDKLSZHZNDQKRLPAXLIDWDPD
   ```

3. This ciphertext is an example from Gaines's book. It uses a keyword for both the plaintext and ciphertext alphabets. In the plaintext, 'J' is used as a space between words. Break it by hand and reconstruct both alphabets to obtain the two keywords.

   ```
   ROVLL ABTLD LBCQM PXLBA FBTCT ATCOR LTOLC RHPDT XLYOA
   ELBXP HLXBT XXQLD RGLTK XRLGD BKLDP PLOHL YOAEL KOMXB
   LHOEL VCRRC RJLTK DTLRC INXPL LLTKX LRCIN XPLVD BLVOR
   LPORJ LDJOL FYLIO PORXP LMDEN XELKC TTLVK OLOHH XEXGL
   TOLIO QMEOQ CBXLH OELTV OLIXR TBLBC RIXLK XLVDB LDFPX
   LTOLB XRGLT KXLBO PATCO RLFYL EXTAE RLQDC PLLBT CPPLC
   TLVOA PGLFX LVOET KLDRO TKXEL RCINX PLTOL HCRGL OATLT
   ```

KXLNX YLLTK CBLQA BTLFX LTKXL XWMPD RDTCO RLOHL TKXLE
XHXEX RIXLT OLDLI EOORX ELDRG LTKXL XQMKD BCBLO RLDLG
DTXLL MLBLT KXLTV OLIXR TBLKD BLROT LYXTL FXXRL MDCGL

madness's book on classical cryptography
unit 27: dictionary attack on the keyword substitution cipher
last modified 2020-11-27

Unit 27
Dictionary attack on the keyword substitution cipher

A *dictionary attack* is an attack in which one tries to decrypt a ciphertext by using a list of possible keywords.

Python tips

Remember that to split a text and store the pieces in an array, we use the `split()` function. To read a list of words that are separated by newline characters (`\n`), you can do something like this:

```
words = open("dictionary.txt","r").read().split("\n")
```

If you are working on a Max or Windows computer, then the ends of lines might be `\n\r` or `\r\n` instead of simply `\n`. You should experiment to determine what is the correct thing to put inside the `split()` function. In addition, to avoid an empty word at the end of the list, you might try keeping only up to the second to last entry, which in Python is denoted as the -1^{st} entry:

```
words = open("dictionary.txt","r").read().split("\n")[:-1]
```

Programming tasks

1. Write a function or script to implement a dictionary attack on a substitution cipher whose key was generated with a keyword. Remember that there are several ways to fill the key. Use the word lists that you compiled earlier. Optionally, allow for the use of a custom word list. Use tetragram fitness of the plaintext to determine if and when you have found the correct keyword.

Exercises

1. Perform a dictionary attack and break this ciphertext. The keyword is a common English word.

```
UHENTWVVENLCMCAONTAIOWTNETTYBTLIGHUEVWPFSOMYIUHINBNVIT
BYINBXIXIVYBCHOYUHEYHOLEWNIXESTEYBTMBVEWPOFPBSUIALETOF
```

MBUESIBLYHIAHNOMBUUESHOYVWLLBNVLIFELETTUHECMIGHUTEEMYE
SENEXESUHELETTFILLEVYIUHUHITINUENTEBNVXIUBLREBWUC

Unit 28
Stochastic hill-climbing attack on monoalphabetic substitution ciphers

This unit describes an attack on the monoalphabetic substitution cipher from Jakobsen's paper in 1995. It is called *stochastic* because it makes random choices as it goes along, and is *hill-climbing* because it works to maximize some function. The function that it maximizes is textual fitness, and for us that means the tetragram fitness that we defined earlier.

The algorithm begins by choosing a key alphabet as the "parent" key and a plaintext is found by deciphering the ciphertext with it. From the parent, a "child" key is obtained by swapping two randomly chosen letters in the parent. The ciphertext is deciphered with the child key, and if the fitness of the resulting plaintext is higher than the parent's plaintext, then the child becomes the new parent, and the process repeats. This continues until the fitness does not improve for a few thousand trials.

Here is the algorithm, so you can't complain that something was unclear:

1. choose a parent key, such as `ABCDEFGHIJKLMNOPQRSTUVWXYZ`
2. decipher the ciphertext with the parent key to obtain the parent's plaintext
3. calculate the fitness of the parent's plaintext
4. set counter to 0
5. while the counter is less than a limit of around 10,000
    a. set the child key as a copy of the parent key
    b. randomly choose two distinct numbers x and y in 0, ..., 25
    c. swap the x^{th} and y^{th} letters in the child key
    d. decipher the ciphertext with the child key
    e. calculate the fitness of the new plaintext
    f. if the fitness of the new plaintext is larger than the fitness of the parent's plaintext
        i. set the parent key as a copy of the child key
        ii. set the parent's plaintext as a copy of the child's plaintext
        iii. set the parent's fitness equal to the child's fitness
        iv. set the counter back to 0
    g. add 1 to the counter
6. output the parent key and/or the parent's plaintext

Python tips

While you cannot modify characters in a string, you can modify items in a list.

Bad:

```
myString = "abcdef"
myString[3] = "z"
```

Good:

```
myArray = ["a", "b", "c", "d", "e", "f"]
myArray[3] = "z"
```

Reading and references

Thomas Jakobsen, “A fast method for cryptanalysis of substitution ciphers,” *Cryptologia* 19:3 (1995) 265-274. DOI: 10.1080/0161-11959188394

Programming tasks

1. Implement the attack.

Exercises

1. Break this ciphertext:

```
IDSIYUDHJZXIXTOQOXUSVOROMNSRMOREXOESGOMMSVOMNSRSUSDHJS
YSTNIJOUQSTSMKSREMNUDJTNIISRJNIDOUKQLHMNGUXLUINIXINHRG
NCDSTIDSUNQCUHRUZSOIXTSVOMNSRSUSNRHRSSCNUHVSIDSUSGTSIQ
SUUOESIHMVYNSJSTUIHJOIGDZXIXTOQOIDXTUVOKUOIISR
```