What is Computer Science?

An Information Security Perspective

Daniel Page ⟨dan@phoo.org⟩ and Nigel P. Smart ⟨csnps@bristol.ac.uk⟩

git # 148def3 @ 2018-07-11

USING SHORT PROGRAMS TO MAKE AND BREAK HISTORICAL CIPHERS

It might seem hard to imagine, but in early 1587 Mary Stewart (or Mary Queen of Scots) [6] was sitting in a jail cell, most likely cursing the subject of **cryptography**. Around a year or so beforehand, Mary was imprisoned in Chartley Hall as a result of her increasingly tense relationship with the then Queen, Elizabeth I. Having been placed under close observation, Mary was only able to communicate with her allies using messages smuggled in and out of the jail inside beer barrels. However, to prevent the messages being used against her should they be discovered, Mary used a system of encoding that substituted characters and common words with a variety of symbols. This gave Mary enough confidence that, while still in jail, she instigated a plot to overthrow Elizabeth: what we now know as the **Babington Plot** is named after her chief conspirator, Anthony Babington. Unbeknown to them, messages sent between Mary and Babington were being intercepted by a double agent, then analysed by an espionage team established by Sir Francis Walsingham. The messages were processed by Thomas Phelippes, who carefully copied their content before resealing and sending them to their intended recipient. Phelippes eventually worked out the encoding system used, and the plot was uncovered when Phelippes took a real message from Mary and added a forged postscript that asked Babington for the names of the conspirators. The resulting proof of conspiracy led to the arrest of Babington and subsequent execution of Mary.

Historical significance aside, the same underlying issue in this story has reoccurred again and again. In 2006, for instance, the famed Mafia boss Bernardo Provenzano was captured for much the same reason as Mary Stewart was executed. Provenzano's encoded notes, including orders to his henchmen and so on, were decoded by police who were then able to capture him. The issue, basically, is that while both Stewart *and* Provenzano clearly understood the need for secrecy, they lacked the range of formal techniques offered by modern cryptography.

Almost all terms used above can be translated into a modern setting. We still talk about encoding messages, but now call this **encryption**, a technique that is used to ensure the secrecy of messages. An unencrypted message is called a **plaintext**, whereas an encrypted message is called a **ciphertext**. In the context of Mary Stewart, a plaintext was formed from characters of the English alphabet while a ciphertext was formed from an alphabet of abstract symbols. We still use the term alphabet to describe the symbols used to form plaintext and ciphertext, but they are more likely to be sequences of bits or bytes that can be processed by a computer. Put another way, we still think about a **sender** and a **receiver** who are communicating messages, but the message is more likely to be an electronic file communicated over a network such as the Internet than written on paper. As a result, either the sender and/or receiver might also be a computer rather than a human. Finally, interception and attempted decryption of messages also has an analogy in communication as we use it today. We call the party trying to do this an **attacker** or **adversary**, while the art of trying to decrypt a message that should be kept secret would be termed **cryptanalysis**. Whereas the encoding methods used by Stewart and Provenzano were eventually **broken** via cryptanalysis of some sort, the design of modern encryption schemes is intended to resist even the most capable attacker.

As with any topic that has evolved in this way, study of basic techniques can still offer insight into modern practise. Our aim here is to look at two types of historical **cipher** (i.e., methods of encryption) in a very practical way. In each case we describe how the cipher works, how it can be broken via cryptanalysis, and how both

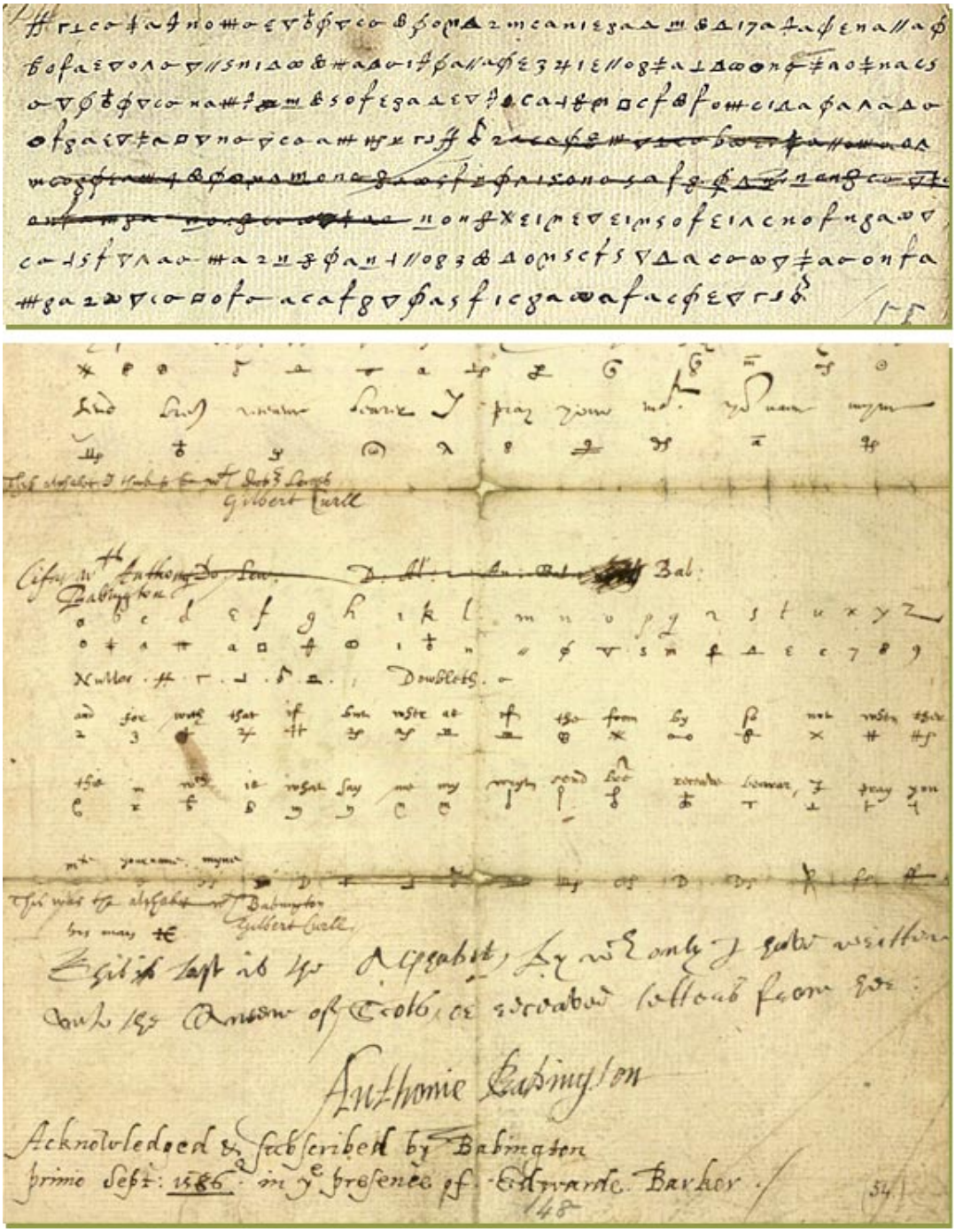

Figure 1: *The message that uncovered the Babington Plot: masquerading as part of a message from Mary Stewart, the postscript asks Babington to reveal the names of the conspirators using the broken cipher (public domain image, source:* http://en.wikipedia.org/wiki/File:Babington_postscript.jpg*).*

aspects can be reproduced using single-line (or at least very short) BASH commands.

1 Shift ciphers

Apparently, Julius Caesar knew about cryptography. Chronicling the life of the Roman leader in *De Vita Caesarum, Divus Iulius*, Suetonius wrote:

> *If he had anything confidential to say, he wrote it in cipher, that is, by so changing the order of the letters of the alphabet, that not a word could be made out. If anyone wishes to decipher these, and get at their meaning, he must substitute the fourth letter of the alphabet, namely D, for A, and so with the others.*

This should sound familiar: Caesar was doing something similar to Mary Stewart in the sense that he was translating characters in a plaintext message into other characters to form a ciphertext message. We use the name **shift cipher** [2] to describe the method of translation used by Caesar. ROT13 [10], a modern day equivalent of the same method, is still used to hide solutions to puzzles in newspapers and so on.

1.1 Encryption and decryption

1.1.1 3-place shifts

We can describe the method used by Caesar using two functions

$$\textsc{Enc}(x) = \begin{cases} \text{'d' if } x = \text{'a'} \\ \text{'e' if } x = \text{'b'} \\ \text{'f' if } x = \text{'c'} \\ \text{'g' if } x = \text{'d'} \\ \vdots \\ \text{'z' if } x = \text{'w'} \\ \text{'a' if } x = \text{'x'} \\ \text{'b' if } x = \text{'y'} \\ \text{'c' if } x = \text{'z'} \end{cases} \qquad \textsc{Dec}(x) = \begin{cases} \text{'a' if } x = \text{'d'} \\ \text{'b' if } x = \text{'e'} \\ \text{'c' if } x = \text{'f'} \\ \text{'d' if } x = \text{'g'} \\ \vdots \\ \text{'w' if } x = \text{'z'} \\ \text{'x' if } x = \text{'a'} \\ \text{'y' if } x = \text{'b'} \\ \text{'z' if } x = \text{'c'} \end{cases}$$

Encryption works by examining each character of the plaintext message in turn. For example, where we see 'a' and want to encrypt it, we use Enc('a') to look-up the result 'd'. Or, in the other direction, if we want to decrypt 'd' then we use Dec('d') to look-up the result 'a'. The term shift cipher comes from the fact that what we are actually doing is shifting the alphabet around: in this case the shifting moves characters by three places.

To demonstrate the process on a larger example, we need something to act as plaintext. As in Chapter 2, we use text downloaded from Project Gutenberg

`http://www.gutenberg.org/`

There are numerous worthy examples we could use, but opt for *The Merchant of Venice* by Shakespeare. Using '△' to make it clear where the spaces are, encrypting the plaintext

't' 'h' 'e' '△' 'm' 'e' 'r' 'c' 'h' 'a'
'n' 't' '△' 'o' 'f' '△' 'v' 'e' 'n' 'i'
'c' 'e'

yields the ciphertext

'w' 'k' 'h' '△' 'p' 'h' 'u' 'f' 'k' 'd'
'q' 'w' '△' 'r' 'i' '△' 'y' 'h' 'q' 'l'
'f' 'h'

One can imagine the cipher being like a big codebook; to encrypt or decrypt messages, Caesar probably employed a trusted slave to apply the translation using tables in the codebook for reference. Obviously this is very tedious, so an important question to resolve (given we lack a slave, but on the other hand have computers) is whether encryption and decryption might be automated. One way would be to write a dedicated program for the task; since we want to focus on the concepts rather than teach programming, we will instead try to automate the process using BASH commands. First we need some plaintext to encrypt. The text for *The Merchant of Venice* will do fine: we save it as the file `A.txt` before translating all characters to lower-case so that our job is made a little easier (since we no longer need to consider the upper-case characters as distinct):

```
<A.txt 'http://www.gutenberg.org/dirs/etext97/1ws1810.txt'
bash$ cat A.txt | tr [:upper:] [:lower:] > B.txt
bash$
```

Although we aim to process the whole file, this is tricky to demonstrate because of the length. Therefore, we focus on a seven line extract starting at line #274:

```
bash$ cat B.txt | tail -n +274 | head -n 7
bash$
```

To encrypt and decrypt we employ the `tr` command, which we already saw in Chapter 2. Recall that `tr` reads lines of input, translates characters in those lines based on rules supplied by the user, and writes the result as output. As before, the rule is given by two sequences: all instances of a given character in the first sequence are translated into the corresponding character in the second sequence. This is easy to see with a simple example. Imagine we want to translate from ⟨'a', 'b', 'c'⟩ into ⟨'g', 'h', 'i'⟩:

```
bash$ cat | tr [a-c] [g-i]
abcdef
ghidef
bash$
```

The input "abcdef" is typed by the user; the first three characters ('a', 'b' and 'c') match those in the first sequence and are thus translated by `tr` into the corresponding characters in the second sequence ('g', 'h' and 'i'). Notice that the next three characters ('d', 'e' and 'f') do not match any in the first sequence so are passed through unaltered. Using this technique, we can encrypt and decrypt files using a 3-place shift cipher as follows:

```
bash$ cat B.txt | tr [a-cd-z] [d-za-c] > C.txt
bash$ cat C.txt | tr [d-za-c] [a-cd-z] > D.txt
bash$
```

In the first command we start with `B.txt` (the original file turned into lower-case), feed this to `tr`, and direct the output into the file `C.txt` which represents the ciphertext. The rule for translation is given by the sequences `[a-cd-z]`, which are short-hand for

$$\langle \text{'a'}, \text{'b'}, \text{'c'}, \text{'d'}, \ldots, \text{'w'}, \text{'x'}, \text{'y'}, \text{'z'} \rangle,$$

i.e., the standard alphabet, and `[d-za-c]`, meaning

$$\langle \text{'d'}, \text{'e'}, \text{'f'}, \text{'g'}, \ldots, \text{'z'}, \text{'a'}, \text{'b'}, \text{'c'} \rangle.$$

In other words, the first command reads input and translates 'a' into 'd', 'b' into 'e', 'c' into 'f', 'd' into 'g' and so on. In the second command we reverse the process by starting with `C.txt` (the ciphertext), feed this to `tr` (where the sets for translation are reversed), and direct the output into the file `D.txt`. Inspecting the relevant lines in the encrypted and decrypted files shows the result:

```
bash$ cat C.txt | tail -n +274 | head -n 7
bash$ cat D.txt | tail -n +274 | head -n 7
bash$
```

Just by looking at the text, we can see the 3-place shift cipher at work. For example, the 'a' of "antonio" in `B.txt` has become a 'd' in `C.txt`. The file `D.txt` which is the decryption *should* match the original file `B.txt` which was encrypted. Although the decrypted extract looks the same as the original, we can *prove* that the whole file is the same using the `diff` command to perform a file comparison:

```
bash$ diff B.txt D.txt
bash$ echo

bash$
```

The lack of output (and exit code, as printed by the `echo` command) indicate there are no differences, i.e., the encryption and subsequent decryption were successful.

This is a simple task: take any *other* plaintext of your choice (e.g., a text file you wrote, or another one from Project Gutenberg) and reproduce the steps above to encrypt then decrypt it.

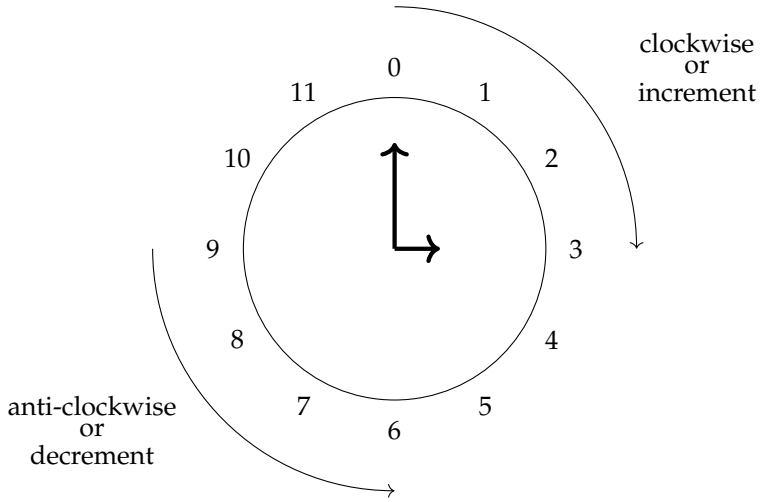

Figure 2: *An analogue clock face showing the time three o'clock.*

1.1.2 *k*-place shifts

At the moment, the Enc and Dec functions must be kept secret: since they are fixed to performing 3-place shifts, if the attacker can work out their behaviour he can decrypt all messages encrypted with them. This is often called "security through obscurity" and is frowned upon in modern cryptography. More usually, we try to build schemes whose security relies only on the secrecy of some **key** rather than the actual method of encryption. This philosophy was articulated by Auguste Kerckhoffs in the late 1800s, and is often called the **Kerckhoffs Principle** [5].

Kerckhoffs actually cited *six* design principles for ciphers. Find out what the others are. Based on how we communicate today compared with 1883, do you think all the principles are still relevant? If any are no longer relevant, what has changed to make this so? Are there any new principles you could add to the list?

Fortunately, we can easily generalise Enc and Dec by adding a key parameter called k that allows more general k-place shifts; the resulting functions are written Enc_k and Dec_k. To describe the generalised functions easily, we first need a way to convert characters to numbers. We could use ASCII, as in Chapter 2, but to make things easier we will use the functions

$$\text{Ord}(x) = \begin{cases} 0 \text{ if } x = \text{'a'} \\ 1 \text{ if } x = \text{'b'} \\ 2 \text{ if } x = \text{'c'} \\ 3 \text{ if } x = \text{'d'} \\ \vdots \\ 22 \text{ if } x = \text{'w'} \\ 23 \text{ if } x = \text{'x'} \\ 24 \text{ if } x = \text{'y'} \\ 25 \text{ if } x = \text{'z'} \end{cases} \qquad \text{Chr}(x) = \begin{cases} \text{'a' if } x = 0 \\ \text{'b' if } x = 1 \\ \text{'c' if } x = 2 \\ \text{'d' if } x = 3 \\ \vdots \\ \text{'w' if } x = 22 \\ \text{'x' if } x = 23 \\ \text{'y' if } x = 24 \\ \text{'z' if } x = 25 \end{cases}$$

where $\text{Ord}(x)$ takes a character x and returns the associated number, and $\text{Chr}(x)$ does the reverse by taking a number x and returning the associated character.

To describe what is going on, it is easier to view the scheme as another application of the **modular arithmetic** [7] we first touched on in Chapter 2. This topic appears frequently in cryptography, so it makes sense to describe it more fully now. Fortunately, we have a nice analogy to help: we are doing what is sometimes called **clock arithmetic**. Imagine someone says to you "the time is now ten o'clock; I will meet you in four hours", what time do they mean? You might say "two o'clock" instinctively, or maybe getting this answer by looking at a clock face such as Figure 2 and moving clockwise 4 hours starting at 10 in order to get to 2. More formally we write this as

$$(10 + 4) = 14 \equiv 2 \pmod{12}$$

so that mod can be read to mean "remainder after division": 14 and 2 are **equivalent** modulo 12 because 14 divided by 12 gives the remainder 2. Of course, we might alter our diagram so it describes a 24-hour clock face

instead; this would illustrate that the equivalence we are discussing "wraps around". We can see this easily by looking at a number line detailing x and $x \pmod{12}$:

$$\begin{array}{lcccccccccccccccccc} x & = & \cdots & -2 & -1 & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & \cdots \\ x \pmod{12} & = & \cdots & 10 & 11 & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 0 & 1 & 2 & \cdots \end{array}$$

Notice that as you would expect $13 \equiv 1 \pmod{12}$, i.e., the time 13 : 00 means one o'clock, and also that two useful facts pop out of looking at the number line:

1. Taking any x and adding 12 is the same as adding 0 because $12 \equiv 0 \pmod{12}$. For example $1 + 12 \equiv 1 \pmod{12}$.
2. If x is negative, $x \pmod{12}$ is given by $12 + x$. For example $-2 \pmod{12}$ is given by $12 + (-2) = 10$. To answer the question "the time is now four o'clock, what time was it six hours ago?" we know $4 - 6 = -2$ and hence $-2 \equiv 10 \pmod{12}$.

While talking about clock faces, 12 is the natural **modulus** to use. In general however, we can select (more or less) any integer modulus we like. What links this fact to the way we have performed encryption and decryption so far, is that both $\textsc{Enc}_k(x)$ and $\textsc{Dec}_k(x)$ can be described using modular arithmetic by setting the modulus to 26. More specifically, we can write

$$\begin{aligned} \textsc{Enc}_k(x) &= \textsc{Chr}(\textsc{Ord}(x) + k \pmod{26}) \\ \textsc{Dec}_k(x) &= \textsc{Chr}(\textsc{Ord}(x) - k \pmod{26}) \end{aligned}$$

This is starting to get a bit complicated, so it makes sense to look at what is going on in more detail. Say we select $k = 3$ to mimic our original functions, and want to encrypt the plaintext 'a':

1. First turn 'a' into a number using $\textsc{Ord}(\text{'a'}) = 0$.
2. Next add k to get $0 + 3 = 3$, and then reduce it modulo 26 to get $3 \bmod 26 = 3$.
3. Finally, translate the number back into a character to get the result $\textsc{Chr}(3) = \text{'d'}$.

In short, we have encrypted 'a' into the ciphertext 'd'. What about decryption? Essentially the same technique applies, meaning we can decrypt 'd' as follows:

1. First turn 'd' into a number using $\textsc{Ord}(\text{'d'}) = 3$.
2. Next subtract k to get $3 - 3 = 0$, and then reduce it modulo 26 to get $0 \bmod 26 = 0$.
3. Finally, translate the number back into a character to get the result $\textsc{Chr}(0) = \text{'a'}$.

Notice that by opting to include the mod 26 operation we have ensured that when adding or subtracting k to a number, the result will "wrap around" to the start or end of the alphabet when turning the number back into a character. We can see this more clearly with another example, this time we encrypt the plaintext 'x':

1. First turn 'x' into a number using $\textsc{Ord}(\text{'x'}) = 23$.
2. Next add k to get $23 + 3 = 26$, and then reduce it modulo 26 to get $26 \bmod 26 = 0$.
3. Finally, translate the number back into a character to get the result $\textsc{Chr}(0) = \text{'a'}$.

The corresponding decryption of the ciphertext 'a' is as follows:

1. First turn 'a' into a number using $\textsc{Ord}(\text{'a'}) = 0$.
2. Next subtract k to get $0 - 3 = -3$, and then reduce it modulo 26 to get $-3 \bmod 26 = 23$.
3. Finally, translate the number back into a character to get the result $\textsc{Chr}(23) = \text{'x'}$.

One can view all this as a mechanism to generate new codebooks. Whereas before Caesar just had one codebook, he can now generate a new one for *each* value of k. Either way, we now have a situation where only k need be kept secret. An attacker might know the method of encryption but without the value of k, he cannot decrypt messages. Better still, we are free to select a *different* k for each message so that even if an attacker recovers the key for one message, he still would not necessarily know the key for another message.

1.2 Cryptanalysis

We already fixed one problem with the initial 3-place shift cipher by making it into a general k-place version; this gave us a cipher that at least made some attempt to comply with the Kerckhoffs Principle. You might have guessed, however, that this is not really enough, and that the cipher is *still* easy to cryptanalyse:

- Although we could have included space as a character in the plaintext alphabet, we did not do so. As such, the word structure of the plaintext is retained in the ciphertext when we encrypt it. We can identify words in both simply by spotting where the space characters are.
- A given character is translated the same way *every* time it occurs. For example if we encrypt one 'a' in the plaintext into a 'd' in the ciphertext, we know that *all* occurrences of 'a' will encrypt to 'd'.
- Even though we are free to select k, there are not that many choices. Selecting $k = 27$ and shifting the alphabet by 27 places, for example, is the same as $k = 1$ because of the "wrap around" effect. So basically there are only 26 possible keys. In fact with $k = 0$ the ciphertext is the same as the plaintext, so actually there are probably more like 25 useful keys.

How can we use these features to cryptanalyse the cipher, for example to decrypt an encrypted message we have intercepted? When presented with the ciphertext we might first employ a **brute-force** attack [1]. This means that we just try *all* the keys. Although laborious, there are only 26 of them so if the key is worth finding it could be worthwhile. Caesar would probably just have 26 slaves try one key each in order to speed things up. The problem is, how do we know when we have got the right key? In our case, we have been assuming that the underlying plaintext is English so as soon as a trial decryption yields text which forms English words we know we have got the right one.

Imagine either we do not have the time to perform the brute-force attack or are just lazy and want a short cut. The next thing we could do is apply a technique called **frequency analysis** [4]. The basic idea is that for a language like English, single characters and combinations of characters occur with varying frequencies. In fact, given a large enough sample, the frequencies are characteristic of *any* text written in that same language: this means, for example, we can make a good guess if the text is English (versus German say), or predict the frequency with which characters occur in one text given frequencies in another (if they are written in the same language).

By retrieving a fresh copy of *The Merchant of Venice*

```
<A.txt 'http://www.gutenberg.org/dirs/etext97/1ws1810.txt'
bash$ cat A.txt | tr [:upper:] [:lower:] > B.txt
bash$
```

we can use some further BASH commands to demonstrate this:

```
bash$ cat B.txt | fold -w 1 | grep [[:alpha:]] | sort | uniq -c | paste -s

bash$
```

The last command in particular needs some explanation. The first part of the command pipeline feeds the file `B.txt` (the original text turned into lower-case) into `fold` which splits each line into one character per-line. Since this is the first time we have used `fold`, a simpler example might be helpful:

```
bash$ cat | fold -w 1
abcd
a
b
c
d
bash$
```

Notice that the single line input of four characters typed by the user has been split into four lines each of one character. As used originally, the output of `fold` is then filtered to leave only alphanumeric characters (or letters and numbers), then sorted using `sort` and fed to `uniq` to count how many duplicates appear (i.e., how many times a given character exists). Finally we format the output nicely using `paste`, and after all that effort, hope something useful came out! The interesting thing is that some occur much more frequently than others. For example 'e' is the most used by some distance, followed by 'o', 't', 'a' and so on. You can view the result above as being somewhat indicative of English in general. Okay, it is *Shakespearean* English but more or less the same frequencies occur in modern text as well, bar the odd "ye olde" or two.

The claim above is that our character frequencies are indicative of the language. Test this claim by performing the same type of analysis on a text file written in something other than English; Project Gutenberg offers a number of French books for instance.

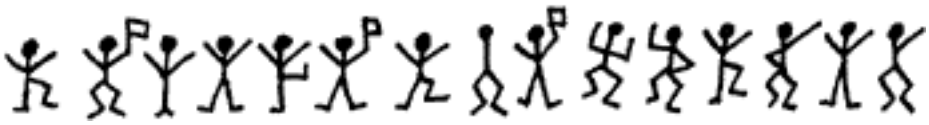

Figure 3: *An example of the "dancing men" used as a cipher in* The Adventure of the Dancing Men *(public domain image, source: http://en.wikipedia.org/wiki/File:Dancing_men.png).*

So imagine someone hands us some ciphertext and challenges us to tell them the key it was encrypted with. To simulate this, we will retrieve a copy of *A Midsummer Night's Dream* again by Shakespeare

```
<A.txt 'http://www.gutenberg.org/dirs/etext97/1ws1710.txt'
bash$ cat A.txt | tr [:upper:] [:lower:] > B.txt
bash$
```

and imagine that someone takes the plaintext `B.txt`, selects a k and then encrypts `B.txt` to give the ciphertext `C.txt`. Our task is to determine the unknown k given *only* `C.txt`. We call this scenario a **ciphertext only** attack [3] since we are given (rather than choose) the ciphertext, and also do not get the corresponding plaintext. We do not present the file `C.txt`, but you could obtain your own version by encrypting some text and then following the same analysis we do; the results you obtain may be slightly different, but the general method should still work.

We first examine what happens if we apply the same frequency analysis to the ciphertext (rather than the plaintext as above). All the cipher does is shift around the alphabet, basically just rearranging the table of frequencies. Employing the method as above, we get a different set of numbers; this should not be surprising since this was a different plaintext originally:

```
bash$ cat C.txt | fold -w 1 | grep [[:alpha:]] | sort | uniq -c | paste -s

bash$
```

However, the crucial thing to notice is that the relative frequency of the characters in the new results should match those in the old results. For example, in this case 'r' is the most used by some distance, followed by 'g', 'b', 'n' and so on. If we consider 'r', 'g', 'b' and 'n' to be the only reasonable way one could have encrypted 'e' then we narrow the range of possible keys to $k = 13$, $k = 2$, $k = 23$ or $k = 9$.

Consider some more evidence in the shape of the eleven line extract of ciphertext starting at line #959:

```
bash$ cat C.txt | tail -n +959 | head -n 11
bash$
```

On the second line of the output, we find a 1-character word 'v'. Not many of these exist in English, so basically we know either 'a' or 'i' must encrypt to 'v' which suggests $k = 21$ or $k = 13$. By this point, $k = 13$ is looking like a good choice: we could now try to decrypt the ciphertext using this key, and see what we get. Selecting $k = 13$ means translating 'a' to 'n' and so on, which means we just need to select the right sequences for `tr` and we are done:

```
bash$ cat C.txt | tr [n-za-m] [a-mn-z] > D.txt
bash$ cat D.txt | tail -n +959 | head -n 11
bash$
```

Even if we did not have `diff` to confirm the result as follows

```
bash$ diff B.txt D.txt
bash$ echo

bash$
```

it would be fairly bad luck to have selected the wrong key and get perfect Shakespearean as output, so we can conclude $k = 13$ was the right key. If we had intercepted `C.txt` from Caesar then we could decrypt his messages and get the jump on him the next time he invaded our country.

2 Substitution ciphers

After generalising the 3-place shift cipher into a k-place version and still failing to produce something which can secure our messages, the next step is something called a **substitution cipher** [11]. In The Adventure of the Dancing Men, Sir Arthur Conan Doyle had his character Sherlock Holmes, *the* arch detective, encounter a cipher of this type. Holmes and Watson are confronted with a number of pictures of dancing men:

> *These hieroglyphics have evidently a meaning. If it is a purely arbitrary one, it may be impossible for us to solve it. If, on the other hand, it is systematic, I have no doubt that we shall get to the bottom of it.*

Of course, the pictures are symbols which encode a message; Holmes and Watson eventually decode the messages and solve yet another case [12]. So if substitution ciphers are important enough for the great Sherlock Holmes to worry about, then they are good enough for us as well.

2.1 Encryption and decryption

Imagine we have a sequence or list of characters

$$A = \langle \text{'a'}, \text{'b'}, \text{'c'}, \text{'d'} \rangle.$$

Given such a sequence, the concept of a **permutation** [9] is central: if we permute the elements in a source sequence, we basically reorder them to produce a target sequence. This means that each element occurs once, but there is some translation from the source to the target sequence. We can describe an example permutation P as follows:

$$P(X) = \langle X_1, X_2, X_3, X_0 \rangle.$$

Put more simply, if we apply P to some source sequence X then the target sequence we get back has X_1 as the 0-th element, X_2 as the 1-st element, X_3 as the 2-nd element and X_0 as the 3-rd element. Applying P to the sequence A above therefore gives us

$$P(A) = \langle \text{'b'}, \text{'c'}, \text{'d'}, \text{'a'} \rangle.$$

Since P is simply reordering the elements, we could write down the **inverse permutation** P^{-1} which performs translation in the opposite direction. In this case

$$P^{-1}(X) = \langle X_3, X_0, X_1, X_2 \rangle$$

which means that

$$P^{-1}(A) = \langle \text{'d'}, \text{'a'}, \text{'b'}, \text{'c'} \rangle$$

and, more importantly, that

$$P^{-1}(P(A)) = \langle \text{'a'}, \text{'b'}, \text{'c'}, \text{'d'} \rangle = A.$$

The basic idea is that the key for a substitution cipher is a permutation; we still write k as the key just for continuity, so you can think of it as a name (or index) that identifies the permutation we use among all those available. The permutation tells us how to translate characters from a source sequence (i.e., the plaintext alphabet) into characters in a target sequence (i.e., the ciphertext alphabet). Of course, one can view the shift cipher as a particular form of permutation. However, the fact that the permutation is of such a particular form makes the cipher weak. We already saw that there are only 26 possible keys, which is far from ideal: a substitution cipher generalises the idea, relaxing the need for a particular form of permutation and allowing *any* permutation at all.

How many possible keys would there be using this generalised approach? We can get the answer by looking at a more general question: say we have an n-element source sequence, how many different permutations of those elements are there? The answer is n factorial or

$$n! = n \cdot (n-1) \cdot (n-2) \cdots 3 \cdot 2 \cdot 1.$$

Why is this is the case? We start with n elements in the source sequence, so there are n choices for the first element in the target sequence. When we remove one of those choices, there are $n-1$ choices left for the second element in the target sequence, $n-2$ choices left for the third element and so on. So given we have $n = 26$ possible characters in our plaintext and ciphertext alphabets there are a total of

$$26! = 403291461126605635584000000$$

possible keys for the substitution cipher. This is now too big to allow searching for the key by brute-force: we need to think a bit harder if we want to break this scheme.

Each key specifies a different permutation; you can think of this as each k specifying a different, secret pair of encryption and decryption functions. Consider an example and imagine that some k specifies the functions:

$$\text{Enc}_k(x) = \begin{cases} \text{'z' if } x = \text{'a'} \\ \text{'y' if } x = \text{'b'} \\ \text{'x' if } x = \text{'c'} \\ \text{'w' if } x = \text{'d'} \\ \vdots \\ \text{'d' if } x = \text{'w'} \\ \text{'c' if } x = \text{'x'} \\ \text{'b' if } x = \text{'y'} \\ \text{'a' if } x = \text{'z'} \end{cases} \qquad \text{Dec}_k(x) = \begin{cases} \text{'a' if } x = \text{'z'} \\ \text{'b' if } x = \text{'y'} \\ \text{'c' if } x = \text{'x'} \\ \text{'d' if } x = \text{'w'} \\ \vdots \\ \text{'w' if } x = \text{'d'} \\ \text{'x' if } x = \text{'c'} \\ \text{'y' if } x = \text{'b'} \\ \text{'z' if } x = \text{'a'} \end{cases}$$

Although these functions are not complete (so they can fit on a page), the general idea acts as a mechanism to generate a codebook; as was the case of the k-place shift cipher, security is based on knowledge of k rather than the actual method of encryption.

Automating *this* encryption method is almost as simple as for the shift cipher. We again fetch the text of *The Merchant of Venice* text and save it as `A.txt` before translating all characters to lower-case:

```
<A.txt 'http://www.gutenberg.org/dirs/etext97/1ws1810.txt'
bash$ cat A.txt | tr [:upper:] [:lower:] > B.txt
bash$
```

Next we set up two strings, which essentially define source and target sequences, and hence the permutation we want to use. This means we can again use `tr` to perform the encryption and decryption:

```
bash$ S='abcdefghijklmnopqrstuvwxyz'
bash$ T='zyxwvutsrqponmlkjihgfedcba'
bash$ cat B.txt | tr   > C.txt
tr: missing operand
Try 'tr --help' for more information.
bash$ cat C.txt | tr   > D.txt
tr: missing operand
Try 'tr --help' for more information.
bash$
```

What does this mean? Essentially what we are saying is that the i-th character in the source sequence `S` should be translated into the i-th character in target sequence `T` so, for example, in this case an 'a' encrypts to a 'z'.

As before, we can focus on a seven line extract starting at line #274 to show the process is working as expected:

```
bash$ cat C.txt | tail -n +274 | head -n 7
bash$ cat D.txt | tail -n +274 | head -n 7
bash$
```

or use `diff` to show that the decrypted file is the same as the original plaintext:

```
bash$ diff B.txt D.txt
bash$ echo

bash$
```

2.2 Cryptanalysis

Since we have improved upon the shift cipher using the stronger permutation cipher, we need need to consider better forms of cryptanalysis to attack it: we cannot say it prevents the previous attack, so therefore is secure! In particular, we need to consider an improved form of frequency analysis that relies on further properties of language.

The concepts of **bigrams** and **trigrams** are special cases of something called an **n-gram** [8]. An n-gram is a sub-sequence of length n taken from some other sequence; a bigram is the case where $n = 2$ and a trigram is the case where $n = 3$. Imagine we want to find all the bigrams of

$$A = \langle \text{'a'}, \text{'b'}, \text{'c'}, \text{'d'} \rangle.$$

We can formulate a solution by saying we want all the sub-sequences which look like

$$\langle A_i, A_{i+1} \rangle.$$

That is, we want all the sub-sequences formed by taking the i-th element and the $(i + 1)$-th element of A. Of course we cannot select element three as the i-th element since the $(i + 1)$-th element would not be valid, but apart from this the sub-sequences we can form are

$$\langle \text{'a'}, \text{'b'} \rangle, \langle \text{'b'}, \text{'c'} \rangle, \langle \text{'c'}, \text{'d'} \rangle.$$

You can think of a text file as just a long string, so it is easy to imagine that we could work out all the bigrams within such a file. How can we do this in practical terms? Using only existing BASH commands demands a cunning approach; imagine we have a file called `A.txt` which has one character per-line:

```
bash$ cat > A.txt
a
b
c
d
bash$
```

Now imagine we take the file and paste it next to itself; we can achieve this easily using the `paste` command:

```
bash$ paste -d ' ' A.txt A.txt
a a
b b
c c
d d
bash$
```

Believe it or not, we are almost there. All we need now do is "skew" the second column by one character. That is, if we skewed the column so that we started at 'b' rather than 'a' we would (more or less) have the bigrams one per-line. To perform the skewing, we use `tail` to take `A.txt` and copy the content starting at the second line. Pasting the result, which we call `B.txt`, alongside the original `A.txt` gives:

```
bash$ tail -n +2 A.txt > B.txt
bash$ paste -d ' ' A.txt B.txt
a b
b c
c d
d
bash$
```

Of course, we might want to eliminate the last line (this is the result of including an "invalid" index) but other than that we can build a list of all bigrams in `A.txt`. The case for trigrams is quite similar but we need to skew the original file by two characters and include that in our `paste` command as well. Suppose we apply this to *The Merchant of Venice*. First we retrieve the text and turn it into lower-case as usual; then we split the characters from `B.txt` into a file called `E.txt` where there is one character per-line and finally skew this file by one and two lines to get `F.txt` and `G.txt`:

```
<A.txt 'http://www.gutenberg.org/dirs/etext97/1ws1810.txt'
bash$ cat A.txt | tr [:upper:] [:lower:] > B.txt
bash$ cat B.txt | fold -w 1 > E.txt
bash$ tail -n +2 E.txt > F.txt
bash$ tail -n +3 E.txt > G.txt
bash$
```

Now we are ready to construct the bigrams and trigrams. Using `paste` as above we take the files `E.txt`, `F.txt` and `G.txt` and paste them into place next to each other. Then we use `grep` to throw away any invalid lines. We do this by specifying that we only want lines with two or three alphabetic characters on them (for the respective bigram and trigram case). Finally we remove the inter-character spacing using `tr` and get the bigrams and trigrams in `H.txt` and `I.txt`:

```
bash$ paste -d ' ' E.txt F.txt       | tr -d ' ' | grep ..  > H.txt
bash$ paste -d ' ' E.txt F.txt G.txt | tr -d ' ' | grep ... > I.txt
bash$
```

Next we can apply a similar approach to analysis of the bigram and trigram frequencies as we previously applied to single character frequencies. We take the input file, sort it using `sort` and feed the output to `uniq` to count how many duplicates exists (i.e., how many times a given bigram or trigram exists). Unlike single character frequencies where there were not many (since there are not many characters), there are a huge number of bigrams and trigrams: we feed the result through `sort` and `tail` to produce only the 20 most frequent:

```
bash$ cat H.txt | sort | uniq -c | sort -n -r | head -n 20 | paste -s

bash$ cat I.txt | sort | uniq -c | sort -n -r | head -n 20 | paste -s

bash$
```

If you think about it, the results are what we would expect: "th" is obviously going to occur more often than "tz" for example, and it should not be a surprise that three character words such as "the" and "and" are the most popular trigrams.

Now imagine we play the same game as before. Someone hands us some ciphertext and challenges us to tell them the key it was encrypted with. To simulate this, we first retrieve a fresh copy of *A Midsummer Night's Dream*

```
<A.txt 'http://www.gutenberg.org/dirs/etext97/1ws1710.txt'
bash$ cat A.txt | tr [:upper:] [:lower:] > B.txt
bash$
```

and imagine someone takes the plaintext `B.txt`, selects k then encrypts `B.txt` to give the ciphertext `C.txt`: our task is again to determine k given only `C.txt`. To avoid having to inspect the whole file, we will again focus on the eleven lines of the ciphertext starting at line #959:

```
bash$ cat C.txt | tail -n +959 | head -n 11
bash$
```

By now, we have a range of techniques available to us. The first step is to run a single character frequency analysis

```
bash$ cat C.txt | fold -w 1 | grep [[:alpha:]] | sort | uniq -c | paste -s

bash$
```

and then to extract the most common bigrams and trigrams:

```
bash$ cat C.txt | fold -w 1 > E.txt
bash$ tail -n +2 E.txt > F.txt
bash$ tail -n +3 E.txt > G.txt
bash$ paste -d ' ' E.txt F.txt       | tr -d ' ' | grep ..  > H.txt
bash$ paste -d ' ' E.txt F.txt G.txt | tr -d ' ' | grep ... > I.txt
bash$ cat H.txt | sort | uniq -c | sort -n -r | head -n 20 | paste -s

bash$ cat I.txt | sort | uniq -c | sort -n -r | head -n 20 | paste -s

bash$
```

Based on all this information we can start to make some guesses about how the ciphertext was produced:

- We still do not consider spaces during encryption, so the cipher still retains the word structure and we know that '△' decrypts to '△'.
- Based on the single character frequency analysis we can be reasonably sure about at least the three most frequent characters and say that 'i' decrypts to 'e', 't' decrypts to 't' and 'y' decrypts to 'o'.
- The bigram and trigram analysis confirms the guesses above because, for example, the most frequent trigram in the ciphertext is "tfi" and so if we match this against "the" (the most frequent trigram in some general text) we confirm the likelihood of 't' decrypting to 't'. Based on further similar matching we can guess that 'f' decrypts to 'h', 'm' decrypts to 'a', 'z' decrypts to 'n' and 'j' decrypts to 'd'.
- Given we already guessed 'm' decrypts to 'a', we can guess that 'e' decrypts to 'i' since on the second line we have a one letter word and we know it cannot decrypt to 'a'.

Based on these initial guesses, and without too much effort, we can already be fairly confident about roughly a third of the key; we can start taking the ciphertext and performing a partial decryption. Considering just the first two lines of our example text:

'y' 'l' 'i' 'v' 'y' 'z' '.' '△' 'e' '△'
'x' 'v' 'm' 'o' '△' 't' 'f' 'i' 'i' '△'
'g' 'e' 'r' 'i' '△' 'e' 't' '△' 'a' 'i'
'.' 'e' '△' 'c' 'z' 'y' 'q' '△' 'm' '△'
'l' 'm' 'z' 'c' '△' 'q' 'f' 'i' 'v' 'i'
'△' 't' 'f' 'i' '△' 'q' 'e' 'b' 'j' '△'
't' 'f' 'o' 'a' 'i' '△' 'l' 'b' 'y' 'q'
'u' ','

we can already decrypt portions of it to read:

'o' 'l' 'e' 'v' 'o' 'n' '.' '△' 'i' '△'
'x' 'v' 'a' 'o' '△' 't' 'h' 'e' 'e' '△'
'g' 'i' 'r' 'e' '△' 'i' 't' '△' 'a' 'e'
'.' 'i' '△' 'c' 'n' 'o' 'q' '△' 'a' '△'
'l' 'a' 'n' 'c' '△' 'q' 'h' 'e' 'v' 'e'
'△' 't' 'h' 'e' '△' 'q' 'i' 'b' 'd' '△'
't' 'h' 'o' 'a' 'e' '△' 'l' 'b' 'o' 'q'
'u' ','

At this point we need to start working harder ... but we can still lean on some existing tools to help us. The basic idea is to start looking at the words which we know part of and narrow down the possibilities for the parts we do not know based on which real words fit the template. This is sort of like the process of filling in a crossword. For example, we can see two partially decrypted words "ae" and "thoae". We know that 'a' probably decrypts to something which will make both of these examples real words. So first we can search the standard dictionary [13] file for all two letter words which end in 'e':

```
ticulture     vituperate      vituperative    vivace  vocalize        vocative
      vociferate      vogue   voice   voile   volatile        vole    voltage vo
luble volume  vote    votive  vouchsafe       voyage  vulcanize       vulgarize
      vulnerable      vulture vulvae  waddle  wade    waffle  wage    waggle  wa
istline       waive   wake    wale    walleye wane    wangle  wannabe warble  wa
rdrobe        ware    warehouse       warfare warhorse        warlike wartime wa
shable        wastage waste   watercourse     waterline       waterside       wa
```

```
ttage wattle  wave    wayside we      we're   we've   wearable        wearisome
      weatherize      weave   website wedge   wee     welcome welfare were    we
sternize      whale   whalebone       wheedle wheelbase       wheeze  whence  wh
ere   wherefore       whetstone       while   whine   whistle white   whittle wh
o're  who've  whole   wholesale       wholesome       whoopee whore   whorehouse
      whose   wide    wife    wiggle  wildfire        wildlife        wile    wi
mple  wince   windowpane      windpipe        wine    winsome winterize       wi
ntertime      wipe    wire    wise    wiseacre        wishbone        wive    wo
bble  woe     woebegone       woke    wolverine       womanize        womanlike
      woodbine        woodpile        woolie  wore    workable        workfare
      workforce       workhorse       workhouse       workmanlike     workplace
      worldwide       wormhole        worrisome       worse   worthwhile      wo
uld've        wove    wrangle wreathe wreckage        wrestle wriggle wrinkle wr
itable        write   writhe  wrote   xylophone       yardage yarmulke        ye
      yippee  yoke    yore    you're  you've  yule    yuletide        yuppie  zo
mbie  zone    zygote  épée  étude
bash$
```

This does not help much; the dictionary does not seem much good for this case! For example, the words "ve" and "qe" might be real, but do not really seem realistic possibilities for someone writing English. If we eliminate the words beginning with characters we are already confident about, this helps to reduce the possibilities; probably only "be", "me", "we" and "ye" remain. Now we can search for all five letter words that start with "th" and end with either "be", "me", "we" or "ye":

```
bash$ cat /usr/share/dict/words | grep - ih.be$ | sort | uniq | paste -s
grep: ih.be$: No such file or directory

bash$ cat /usr/share/dict/words | grep - ih.me$ | sort | uniq | paste -s
grep: ih.me$: No such file or directory

bash$ cat /usr/share/dict/words | grep - ih.we$ | sort | uniq | paste -s
grep: ih.we$: No such file or directory

bash$ cat /usr/share/dict/words | grep - ih.ye$ | sort | uniq | paste -s
grep: ih.ye$: No such file or directory

bash$
```

Although "thebe" *might* be a reasonable Shakespearean word, it seems more likely that 'a' decrypts to 'm' given the only other two words in the dictionary support this choice. Updating our partial decryption we get:

'o'	'l'	'e'	'v'	'o'	'n'	'.'	'△'	'i'	'△'
'x'	'v'	'a'	'o'	'△'	't'	'h'	'e'	'e'	'△'
'g'	'i'	'r'	'e'	'△'	'i'	't'	'△'	'm'	'e'
'.'	'i'	'△'	'c'	'n'	'o'	'q'	'△'	'a'	'△'
'l'	'a'	'n'	'c'	'△'	'q'	'h'	'e'	'v'	'e'
'△'	't'	'h'	'e'	'△'	'q'	'i'	'b'	'd'	'△'
't'	'h'	'o'	'm'	'e'	'△'	'l'	'b'	'o'	'q'
'u'	','								

The process can continue in a similar way, using the frequency analysis to back up our guesses. Although we are working in a known ciphertext scenario, we can bend the rules a bit by considering some knowledge about the plaintext (actually we already did this by assuming it was English). Why is this not cheating? Imagine you get an encrypted email. In this case you know that there is a high chance that the start of the email includes the headers "to", "from", "subject" and so on, and this type of information can help conclude our search more quickly.

Fast-forwarding a little then, we would eventually recover the whole key and hence the method of substitution between plaintext and ciphertext characters. Using this, and in the same way as the shift cipher example, we can test if the result of a trial decryption looks reasonable:

```
bash$ S='abcdefghijklmnopqrstuvwxyz'
bash$ T='mlkjihgfedcbazyxwvutsrqpon'
bash$ cat C.txt | tr   > D.txt
tr: missing operand
Try 'tr --help' for more information.
bash$ cat D.txt | tail -n +959 | head -n 11
bash$
```

Find someone to work with. One of you act as the sender of some secret plaintext message, and the other as the cryptanalysist:

1. the sender selects k, i.e., a permutation, and encrypts the plaintext message to produce a ciphertext, then
2. the cryptanalysist is given the ciphertext, and asked to recover k (or at least the plaintext message, partial or otherwise).

Did it work? Several things can go wrong: can you think why the process might fail? For example, what assumptions do we make and therefore depend on being true?

References

[1] *Wikipedia: Brute-force attack*. http://en.wikipedia.org/wiki/Brute_force_attack (see p. 9).

[2] *Wikipedia: Caesar cipher*. http://en.wikipedia.org/wiki/Caesar_cipher (see p. 5).

[3] *Wikipedia: Ciphertext-only attack*. http://en.wikipedia.org/wiki/Ciphertext-only_attack (see p. 10).

[4] *Wikipedia: Frequency analysis*. http://en.wikipedia.org/wiki/Frequency_analysis (see p. 9).

[5] *Wikipedia: Kerckhoffs' principle*. http://en.wikipedia.org/wiki/Kerckhoffs'_principle (see p. 7).

[6] *Wikipedia: Mary I of Scotland*. http://en.wikipedia.org/wiki/Mary_I_of_Scotland (see p. 3).

[7] *Wikipedia: Modular arithmetic*. http://en.wikipedia.org/wiki/Modular_arithmetic (see p. 7).

[8] *Wikipedia: N-gram*. http://en.wikipedia.org/wiki/N-gram (see p. 12).

[9] *Wikipedia: Permutation*. http://en.wikipedia.org/wiki/Permutation (see p. 11).

[10] *Wikipedia: ROT13*. http://en.wikipedia.org/wiki/ROT13 (see p. 5).

[11] *Wikipedia: Substitution cipher*. http://en.wikipedia.org/wiki/Substitution_cipher (see p. 10).

[12] *Wikipedia: The Adventure of the Dancing Men*. http://en.wikipedia.org/wiki/The_Adventure_of_the_Dancing_Men (see p. 11).

[13] *Wikipedia:* `words`. http://en.wikipedia.org/wiki/Words_(Unix) (see p. 14).

I have a question/comment/complaint for you. Any (positive *or* negative) feedback, experience or comment is very welcome; this helps us to improve and extend the material in the most useful way.

However, keep in mind we are far from perfect; mistakes are of course possible, although hopefully rare. Some cases are hard for us to check, and make your feedback even more valuable: for instance

1. minor variation in software versions can produce subtle differences in how some commands and hence examples work, and
2. some examples download and use online resources, but web-sites change over time (or even might differ depending on where you access them from) so might cause the example to fail.

Either way, if you spot a problem then let us know: we will try to explain and/or fix things as fast as we can!

Can I use this material for something ? We are distributing these PDFs under the Creative Commons Attribution-ShareAlike License (CC BY-SA)

http://creativecommons.org/licenses/by-sa/4.0/

This is a fancy way to say you can basically do whatever you want with them (i.e., use, copy and distribute it however you see fit) as long as a) you correctly attribute the original source, and b) you distribute the result under the same license.

Is there a printed version of this material I can buy? Yes: Springer have published selected Chapters in

http://www.springer.com/computer/book/978-3-319-04041-7

while *also* allowing us to keep electronic versions online. Any royalties from this published version are donated to the UK-based Computing At School (CAS) group, whose ongoing work can be followed at

http://www.computingatschool.org.uk/

Why are all your references to Wikipedia? Our goal is to give an easily accessible overview, so it made no sense to reference lots of research papers. There are basically two reasons why: research papers are often written in a way that makes them hard to read (even when their intellectual content is not difficult to understand), and although many research papers are available on the Internet, many are not (or have to be paid for). So although some valid criticisms of Wikipedia exist, for introductory material on Computer Science it certainly represents a good place to start.

I like programming; why do the examples include so little programming? We want to focus on interesting topics rather than the mechanics of programming. So even when we include example programs, the idea is to do so in a way where their meaning is fairly clear. For example it makes more sense to use pseudo-code algorithms or reuse existing software tools than complicate a description of something by including pages and pages of program listings.

But you need to be able to program to do Computer Science, right? Yes! But only in the same way as you need to be able to read and write to study English. Put another way, reading and writing, or grammar and vocabulary, are just tools: they simply allow us to study topics such as English literature. Computer Science is the same. Although it *is* possible to study programming as a topic in itself, we are more interested in what can be achieved *using* programs: we treat programming itself as another tool.