Math 3011 Coding & Cryptography

Lecture Notes (Expanded)

Chris Williams

Last updated November 24, 2025.

University of Nottingham

Contents

Administrative matters

These are the *expanded* notes for the third year undergraduate module *Coding and Cryptography*, running at the University of Nottingham.

The Golden Rule. If you have any questions/feedback/corrections about the course or these notes, then **please email me!** My email is:

`chris.williams1@nottingham.ac.uk`

Not only will feedback be gratefully received, but it helps me improve the course/classes/lecture notes. Ideally taking this course will be interactive, and you will let me know where I need to do better/what I should keep doing.

Notes on these notes. If you've just picked up these lecture notes for the first time, please don't panic about their length. To reiterate, these are the 'expanded' notes. The 'standard' notes, which I will be maintaining throughout the semester, are shorter and exactly cover the lectures and examinable material. You should consider this longer set of notes as a super-targeted textbook to accompany the course. I have tried to ensure that they are the only resource you will need for the course.

More precisely, these notes cover the same material as the lectures, but in much greater detail: the proofs are longer, there is more motivation, there are more examples, and there are extra, non-examinable, remarks. The hope is that if you find something in lectures to be difficult, then you can consult this more detailed document to improve your understanding. The numbering I use here (e.g. 'Theorem 3.12') will match the numbering I use in lectures.

These notes will be updated as the course progresses, to reflect any changes in the material. I will also upload my written notes for each lecture week by week.

Further reading list. If you should find yourself wanting to learn more, or looking for a different take on the examinable material, there are plentiful additional references you could consult.

Iterations of this course exist in many guises, and an internet search for 'Coding and Cryptography Lecture Notes' throws up several excellent, free, online resources. Some cover much the same material; some go much further. I personally recommend the notes of Tom Körner and Chris Wuthrich, which I consulted myself whilst preparing the course.

The following books are also relevant to this course:

- Dominic Welsh, *Codes and cryptography* (QA 269.5 WEL)
- San Ling and Chaoping Xing, *Coding theory : a first course* (QA 268 LIN)
- Raymond Hill, *A first course in coding theory* (QA 269.6 HIL)

- William Stein, *Elementary number theory: primes, congruences, and secrets* (QA241 STE)
- Gareth A. Jones and Mary J. Jones, *Elementary number theory* (QA 241 JON)
- Henry Beker and Fred Piper, *Cipher systems : the protection of communications* (QA 269.5 BEK)
- Simon Singh, *The code book* (QA 76.9 A25 SIN)

Assessment details. This course will be assessed via 100% written examination.

Prerequisites. To understand this course, you should not require mathematics beyond 1st year core modules. In particular, some basic skills in mathematical proofs and reasoning and knowledge of foundational linear algebra are needed. We will also use modular arithmetic (i.e. modulo p) and school-level probability theory (e.g. of the type learnt at A-level in the UK).

Acknowledgements. I inherited this course from previous lecturers of this material at Nottingham, and these notes are heavily based on their courses. The most recent incumbent in the C&C hotseat was Al Kasprzyk, and I'm very grateful to him for sharing his course materials from 2021/22. His notes in turn built on those of Fredrik Stromberg, Chris Wuthrich, and Douglas Woodall; and these were influenced by courses of Tom Körner and Peter Symonds elsewhere.

Motivation

Pure mathematics: a 'useless' subject?

This is a course in pure mathematics. Pure mathematics is often seen as a very *abstract* pursuit, divorced from the 'real world' which we inhabit every day. For example, on a concrete level, why should one care about axiomatising vector spaces, or studying algebraic properties of rings and fields? When do we ever *use* properties of prime numbers in our day-to-day lives?

I myself am a number theorist. Historically, number theory – the study of primes, and of solutions to equations – has existed on an abstract plane of its own, existing only for the sake of mathematical 'beauty'. In 1940, in *A Mathematician's Apology*, the number theorist G.H. Hardy famously asserted that the subject was 'useless':

> *"I have never done anything 'useful'. No discovery of mine has made, or is likely to make, directly or indirectly, for good or ill, the least difference to the amenity of the world... I am interested in mathematics only as a creative art."*
>
> *"I count Maxwell and Einstein, Eddington and Dirac among 'real' mathematicians. [Their achievements are] almost as 'useless' as the theory of numbers."*

On a personal level, I empathise with Hardy's position. I find mathematics to be a beautiful pursuit in its own right, a chase to discover fundamental patterns and connections between different fields and concepts, and a hunt to explain why such connections should exist.

If I was to summarise this course in one sentence, it would be:

> *"Hardy was totally wrong."*

Many strands of abstract pure mathematics have found stunning real-life applications decades after they were first discovered. Even number theory, one of the most abstract of all, is now used fundamentally in our day-to-day lives.

In this course, we will look at two particular applications of pure mathematics in **modern telecommunications**:

- The theory of *mathematical codes*, an application of linear algebra, ring theory and probability to the **accuracy** of communications.
- The theory of *cryptography*, an application of number theory to the **security** of communications.

These topics are most readily associated with things like Cold War espionage. However, both topics now underpin many aspects of everyday modern life. For example, codes are vitally used when saving a document on your computer, accessing a JPEG image on a tablet, or watching a video on YouTube; whilst cryptography is used when accessing an email account, or transferring money via online banking. Often both codes *and* cryptography are used: when you send a WhatsApp message, it is encrypted with cryptography, and the accuracy of its content is ensured by codes.

Part I: Error-correcting codes

The first half of the course will focus on coding theory, and in particular on *error-correcting codes.*

In practice, no method of communication is perfect. Even in face-to-face conversations you can mishear somebody, and if a message is passed from person to person, then by the time it reaches the 10th person it might be so altered as to be unrecognisable from the original message. In practice, if we mishear then our brains can often 'fill in' the missing or changed parts from context or visual cues, thus performing a sort of 'biological error-correction':

fpr exbmple, yiu cbn prabobly unddrstpnd thss swntpnce.

With electronic communication, no such evolutionary error-correction is available. Physical errors (on a hard drive or CD) or transmission errors (via poor signal, or unreliable networks) are a fact of life. If such errors are left unchecked, they can eventually render even the most innocuous communication unreadable. Worse, in a critical system – for example, aboard a NASA spacecraft – electronic errors could compromise safety.

Mathematics allows us to build error-correction into computer systems. There are two aspects to this:

- Can you **detect** that an error has occurred?
- Can you **correct** any errors that occur?

We'll see how to do both of these using codes.

Part II: Cryptography

The second half of the course will be concerned with *cryptography.*

Ancient and not-so-ancient methods of cryptography. Writing and sending messages is an ancient pastime. For as long as people have sent messages to each other, it has been essential to disguise them, so that only the intended recipient can read them.

One method of doing this is *steganography*, trying to hide the *existence* of a message, but leaving its contents readable to any that know it is there. In Greece in 500BC, this was famously done by shaving and tattooing a slave's head, then allowing their hair to grow back to conceal the message. In the same direction, people have used invisible inks for millennia, whilst in the late 19th century, people discovered how to hide messages by shrinking them to the size of a dot.

Usually, in the modern world, steganography is just not practical. Instead, it is better to disguise the *contents* of a message from prying eyes. This is *cryptography.*

In an early example of cryptography, Julius Caesar coded his letters using the 'Caesar cipher', translating all the letters of the alphabet by a fixed amount to render a block of text apparently meaningless: for example, replacing every letter with the one after it in the alphabet (so A→B, B→C, etc) sends the message

$$\boxed{\texttt{pure mathematics is great!}} \longrightarrow \boxed{\texttt{qvsf nbuifnbujdt jt hsfby!}}$$

However, this is very easily broken: there are only 26 possible translations, and one can check them all by hand to decode a message. As the years went by, more advanced ciphers, such as Vigenère ciphers, were developed, which were harder – but still possible – to crack manually.

In the mid 20th century, the rise of computing began to invalidate all the old methods of cryptography. Computers made it suddenly possible to evaluate millions of permutations, and to analyse ciphers in huge detail, making it possible to decode even the most sophisticated ciphers. Famously, the computers developed at Bletchley Park were key to cracking the Enigma cipher used in the second world war.

A practical use for prime numbers. Even though classical ciphers had been cracked, people still needed to send messages secretly. This meant the hunt was on for new methods of cryptography. One needed to balance two things:

- A message needed to be sufficiently encoded that even the fastest computers could not be expected to crack it;
- but it must still be easily accessed by its intended recipient.

It turned out that number theory, and the abstract structures and theorems described as 'useless' by Hardy, provided an ideal solution to both of these needs. For example, the RSA method of cryptography relies on the following mathematical problem, which is computationally extremely difficult to solve:

> Take two large prime numbers p and q, and let $N = pq$ be their product.
>
> **Question:** Given N, find p and q.

Loosely, given a message encoded with RSA, you can only decode it if you know one of p or q. If you only know N, the message is unreadable. We will describe how RSA works later in the course.

As technology has advanced, the need for secure communications has only increased. When speaking with someone over the internet, how can you be sure you are speaking to the right person, and not someone posing as them? When you access your online bank account, how can the bank be sure that it's you trying to manage your money? If an email is intercepted, how can one ensure that its content remains private? When you connect to a mobile phone network, how is the call or text secured? In all of these things, our state-of-the-art cryptographic methods are underpinned by Hardy's 'useless' theory of numbers.

Life beyond prime numbers. Just as computers rendered older methods of cryptography unusable, there is a very real chance that technological advances will render our current cryptographic methods unsafe. In particular, the next major advance has been predicted for decades: that of *quantum computing*. An effective quantum computer would, thanks to 'Shor's algorithm', make short work of our current state-of-the-art cryptography. Because of this, the hunt is on for newer methods of encoding messages and data. Pure mathematics, and number theory, is at the heart of all of these efforts!

Part I

Coding

In Part I, we study mathematical codes, and in particular, *error-correcting codes*. The ubiquitous picture, which we'll see over and over again, is the following:

$$\begin{tikzcd}
\textbf{Alice} & & \textbf{Bob} \\
\boxed{\text{sourceword } \mathbf{w} \in V_k} \arrow[dd, "\textbf{Enc}"] & & \boxed{\text{sourceword } \mathbf{w}' \in V_k} \\
& & \boxed{\textbf{codeword } \mathbf{c}' \in C \subset V_n} \arrow[u, "\textbf{Dec}"'] \\
\boxed{\textbf{codeword } \mathbf{c} \in C \subset V_n} \arrow[rr, dashed, "\text{Noisy channel}"] & & \boxed{\text{received word } \mathbf{x} \in V_n} \arrow[u, "\text{ERROR-CORRECT}"']
\end{tikzcd}$$

Gradually, we'll see how this picture comes together:

- In Chapter 1, we'll introduce this picture, some basic error-correcting codes, and what they do.
- In Chapter 2, we'll develop precise (theoretical) mathematical models underpinning this picture.
- In Chapter 3, we'll analyse properties of codes through the notion of Hamming distance. We'll attach an invariant – the minimal distance – that totally determines how good a code is at correcting errors.
- In Chapter 4, we'll look at how codes are actually used in practice, using linear algebra. This will allow us to make all the arrows in this picture *practical*, ready for use in the real world.
- In Chapter 5, we'll draw all of our theory together to – finally! – write down families of good codes (the Hamming codes and Reed–Muller codes) which are in wide use today.

Chapter 1

Introduction to error-correcting codes

In this chapter, we will introduce codes, and give some first examples of codes (both theoretical codes and practical codes that you find in the wilds of day-to-day life). In the process, we set up the mathematical language and parameters for studying codes. In this chapter I'll assume a basic familiarity with probability theory, and essentially nothing else.

We focus on *Error Correcting Codes*. These are intended to improve reliability of data transfer by minimising the effects of transmission error. In particular, there are two key aspects to this:

Can you **detect** if an error has occurred?
...If an error occurs, can you **correct** it?

Example 1.1. Suppose somebody sends you a message, and you receive:

```
Hgllo!
```

This is not a real word. From this, you *detect* that there has been an error.

Now, what was the intended message? In this example, it's very likely to have been `Hello!`. In this case, we've also *corrected* the error: the `e` of `hello` glitched to become a `g`.

We perform this sort of error-correction frequently on an everyday basis, 'auto-correcting' text messages in our heads. We recognise errors but contextual clues mean we can often correct them.

To give another example: in day-to-day conversation, if you mishear somebody, there are sometimes other clues to help you 'correct' your understanding. You might pick up on tone of voice, or body language, or lip-reading[1]. These are biological methods of 'error-correction'.

However, biological error-correction is not available to computers. Moreover, it's certainly not infallible, as the next example shows.

Example 1.2. Here's another message. You receive

```
the cost will be £9 million.
```

Has there been an error? It doesn't look like it?

...But what if actually, I sent the message

```
the cost will be £1 million.
```

[1]On a personal level, during the covid pandemic I realised how much I sometimes rely on lip-reading; I found it much harder to understand people wearing masks, as I could not see their mouths.

Oops!! A one-digit error has led to you receive a message that looks legitimate, but is likely to lead to major misunderstandings between us. There was no way to *detect* this error, let alone correct it.

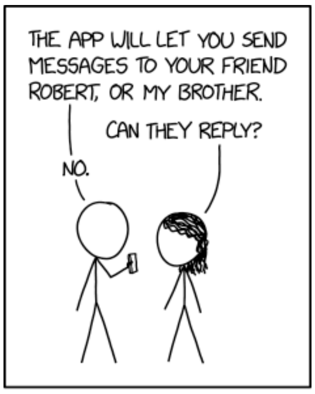
To avoid this kind of catastrophic error, we turn to mathematics!

1.1. What is a code and what should it do?

Our intrepid adventurer, Alice, has a very important message she wants to send Bob digitally:

`the password is ‘math’`

How does she send it? The first thing your phone, computer, or tablet will do is convert the message into *binary*. (Throughout this course, we will focus primarily on binary codes, since this is the language of computers. See the end of this chapter for some more general types of codes, including those governing credit card and ISBN numbers).


THE APP WILL LET YOU SEND MESSAGES TO YOUR FRIEND ROBERT, OR MY BROTHER.
NO.
CAN THEY REPLY?

MY NEW SECURE TEXTING APP ONLY ALLOWS PEOPLE NAMED ALICE TO SEND MESSAGES TO PEOPLE NAMED BOB.

From `xkcd.com`

Definition 1.3 (Binary numbers)**.** A *binary digit*, or *bit*, is either 0 or 1. A *binary number* or *binary string* is a word of the form 0110001010. A *byte* is a binary string of length 8.

We convert text into binary using ASCII, which converts each letter to a 8-digit binary number: e.g.

$$\texttt{a} \longleftrightarrow 01100001.$$

Under this, Alice's message becomes the following 176-digit binary string:

```
01110100 01101000 01100101 00100000 01110000 01100001 01110011 01110011 01110111 01101111 01110010
01100100 00100000 01101001 01110011 00100000 01100000 01101101 01100001 01110100 01101000 00100111.
```

Now she needs some way to send her message.

Definition 1.4. A *channel* is a communication device (e.g. a mobile phone network) that sends individual bits one at a time, (i.e. each transmission is either 0 or 1).

Now Alice sends her message one bit at a time over the channel (so there will be 176 transmissions, each of either a 0 or a 1, to send her message).

In a perfect world, every time Alice sends a 0, Bob receives a 0. In practice, no channel is perfect (for example, maybe the phone signal is poor), and errors will always occur in real life. A better real-world model is:

Definition 1.5. A *noisy* channel is a channel where an errors can occur (so a 0 is sent, but a 1 is received; or a 1 is sent and a 0 is received).

In this case, the message that Bob receives might not be the same as the one Alice sent! For example, the 150th digit of Alice's (binary) message is a 0. If it glitches to a 1, then Bob decodes her message as follows:

$$\begin{tikzcd}
\textbf{Alice} & \textbf{Bob} \\
\texttt{the password is `math'} \arrow[d, "\text{ASCII}"] & \texttt{the password is `meth'} \\
\texttt{01110100}\cdots\texttt{0}\cdots\texttt{0111} \arrow[r, "\textbf{Noisy channel}"] & \texttt{01110100}\cdots\texttt{1}\cdots\texttt{0111} \arrow[u, "\text{ASCII}^{-1}"']
\end{tikzcd}$$

...Oops!

The idea behind codes is to add *redundancy* to messages to mitigate errors. More precisely, you don't just send your message – you also send *more* information that allows Bob to work out that an error has occurred, and then (possibly) to deduce what the intended message was.

Example 1.6 (Repetition codes)**.** Suppose Alice wants to send a 0 over a noisy channel. A natural way to ensure Bob receives a 0 is to send it multiple times: say, send it *three* times, and get Bob to take the majority. This is a 'repetition code'. Our model becomes

$$\begin{tikzcd}
\textbf{Alice} & \textbf{Bob} \\
\texttt{0} \arrow[dd, "\text{repetition code}"] & \texttt{0} \\
 & \texttt{000} \arrow[u, "\text{decode}"'] \\
\texttt{000} \arrow[r, dashed, "\text{Noisy channel}"] & \texttt{010} \arrow[u, "\text{ERROR-CORRECT}"']
\end{tikzcd}$$

Now our problem is solved, right? We can simply send each bit three times, and Bob can correct by taking the majority. If errors are rare (as they often will be in practice), this is now extremely accurate.

...However. Repetition codes are actually **very bad** codes! You have sent *three times* as much information to fix each possible error. This is both very slow and, when you need to buy mobile data or server space, very expensive. We have added far too much redundancy.

Let's consider a more abstract version of the repetition code. In the above, we took `0` – a message of length 1 – and 'encoded' it as `000`, a message of length 3. To work more generally, we need to consider binary numbers of fixed length:

Definition 1.7. We write V_n for the set of all binary numbers of length n. For example,

$$\begin{aligned}
V_1 &= \{0,1\},\\
V_2 &= \{00,01,10,11\},\\
V_3 &= \{000,001,010,100,011,101,110,111\}.
\end{aligned}$$

Note that $V_n = \{0,1\}^n$ has cardinality 2^n.

Definition 1.8. Let $n \geqslant k$.

- A *code* C of message length k and block length n is a set of words
$$C \subset V_n$$
which has cardinality 2^k. (These are sometimes called *block codes*).
- The elements of a code C are called *codewords*.
- Note that C and V_k have the same cardinality. An *encoding map* is a choice of bijection[a]
$$\mathbf{Enc} : V_k \xrightarrow{\text{1-to-1}} C.$$
This bijection will have an inverse, the *decoding map*
$$\mathbf{Dec} : C \xrightarrow{\text{1-to-1}} V_k.$$
- The elements of V_k are called *sourcewords*.

[a]Recall: a bijection is a 1-to-1 map, i.e. it is *injective* (no two elements of V_k are mapped to the same thing in C) and *surjective* (every element of C is mapped to by some element of V_k).

Remark. Note encoding maps are far from unique: there are $2^k! = 2^k \cdot (2^k - 1) \cdot (2^k - 2) \cdots 2 \cdot 1$ choices. Even if k is only 3, this gives over 40,000 possible encoding maps; and for $k = 4$, this jumps to 20 trillion choices! However, we will often see there are 'natural' choices of encoding map.

This might seem a bit abstract, but will hopefully become clear through the following example, where we reinterpret the repetition code above in this more abstract language.

Example 1.9 (Repetition codes)**.** Let $k = 1$, and $n = 3$. Then $C = \{000, 111\} \subset V_3$ is a code, the *3 times repetition code of message length 1.* There is an obvious encoding map

$$\begin{aligned} \mathbf{Enc} : V_1 &\longrightarrow C \subset V_3 \\ \mathtt{x} &\longmapsto \mathtt{xxx}, \end{aligned}$$

i.e. $0 \mapsto 000$ and $1 \mapsto 111$. (We could alternatively have chosen **Enc** to be the map sending $0 \mapsto 111$ and $1 \mapsto 000$, but this would be perverse).

Graphically, the process is:

$$\begin{tikzcd}
\textbf{Alice} & & \textbf{Bob} \\
\boxed{\text{sourceword } 0 \in V_1} \arrow[dd, "\textbf{Enc}"] & & \boxed{\text{sourceword } 0 \in V_k} \\
 & & \boxed{\textbf{codeword } 000 \in V_3} \arrow[u, "\textbf{Dec}"'] \\
\boxed{\textbf{codeword } 000 \in V_3} \arrow[rr, dashed, "\text{Noisy channel}"] & & \boxed{\text{received word } 010 \in V_3} \arrow[u, "\text{ERROR-CORRECT}"']
\end{tikzcd}$$

Here we've used that 010 is much more likely to have come from 000 (1 error was made) than 111 (2 errors were made). In this case, if there is an error in 1 bit, it is correctly corrected.

The abstract version of this model is the following.

$$\begin{tikzcd}
\textbf{Alice} & & \textbf{Bob} \\
\boxed{\text{sourceword } \mathbf{w} \in V_k} \arrow[dd, "\textbf{Enc}"] & & \boxed{\text{sourceword } \mathbf{w}' \in V_k} \\
 & & \boxed{\textbf{codeword } \mathbf{c}' \in C \subset V_n} \arrow[u, "\textbf{Dec}"'] \\
\boxed{\textbf{codeword } \mathbf{c} \in C \subset V_n} \arrow[rr, dashed, "\text{Noisy channel}"] & & \boxed{\text{received word } \mathbf{x} \in V_n} \arrow[u, "\text{ERROR-CORRECT}"']
\end{tikzcd}$$

Here:

- We start with our sourceword $\mathbf{w}$, as before.
- We encode it using **Enc**, obtaining a codeword in V_n.
- We send the message over the noisy channel.
- If there is an error, the likelihood is that the received word is *not a codeword!* As we are only ever sent codewords, we have **detected** an error.
- If the code is chosen cleverly enough, we can attempt to deduce which codeword $\mathbf{c}$ was sent. In this **error-correcting** stage we pass back to a codeword $\mathbf{c}'$.
- We decode $\mathbf{c}'$ as a sourceword $\mathbf{w}'$.

Before looking at other codes, we introduce a very important number attached to a code. Remember that we said that repetition codes are very bad, because we added far too much redundancy. We measure the added redundancy via the following invariant:

Definition 1.10 (Rate)**.** Let C be a code of block length n and message length k. The *rate*

of C is
$$r(C) := \frac{k}{n}.$$

Note $r(C) \leqslant 1$, and the larger $r(C)$ is, the more efficient the code is.

Example. The 3 times repetition code has rate $r(C) = 1/3$, much less than 1.

Example. Here's another example of a very bad code: let $C = \{000, 100, 010, 001\} \subset V_3$. Then:

- C has $2^k = 4 = 2^2$ elements, so we deduce the message length is $k = 2$.
- By definition C is a subset of V_3, so the block length is $n = 3$.
- Thus the rate is $r(C) = k/n = 2/3$.

Why is this a bad code? Well, suppose Alice wants to send `000`. If there is just one error, can she detect it?... No! Because Bob will receive one of `100`, `010` or `001`. These are all codewords in C, so he must assume Alice meant to send them.

...so this is a code where you send 50% extra information, but you can't even *detect* a single error, let alone correct one.

Remark. Whilst error-correction makes it more likely that Bob receives what Alice sent, there is no guarantee! For example, the following might be rare, but is still possible:

$$\begin{tikzcd}
\textbf{Alice} & \textbf{Bob} \\
\boxed{\text{sourceword } 0 \in V_1} \arrow[dd, "\textbf{Enc}"] & \boxed{\text{sourceword } 1 \in V_k} \\
 & \boxed{\textbf{codeword } 111 \in V_3} \arrow[u, "\textbf{Dec}"'] \\
\boxed{\textbf{codeword } 000 \in V_3} \arrow[r, dashed, "\text{Noisy channel}"] & \boxed{\text{received word } 011 \in V_3} \arrow[u, "\text{ERROR-CORRECT}"']
\end{tikzcd}$$

In a noisy channel, no perfect transmission is ever possible: we can only hope to be as close to perfect as possible.

Remark. This discussion is tailored to error-correcting codes, where we *add* redundancy to improve accuracy/allow detection and correction of errors. However, you could also *remove* redundancy by taking $k > n$, which is what you'd do in compression codes (e.g. jpeg or mp3). As an example:

> Ther is actua a lot of redund in langu. I have remo lots of lett from this sent, but it' sti poss to unders it!

Example. Here are some more (bad) codes, and their associated invariants.

- $C = \{00, 11\} \subset V_2$ (the 2 times repetition code). Here C has $2 = 2^1 = 2^k$ elements, so the message length is $k = 1$. The block length is $n = 2$ (since the elements of C are 2-digit binary numbers). The rate is $r(C) = k/n = 1/2$. (How many errors can C detect? How many can it correct?)

- $C = \{0101, 1111, 1000, 0001\} \subset V_4$. Here C has $4 = 2^2 = 2^k$ elements, so the message length is $k = 2$. The block length is $n = 4$. The rate is $1/2$.
- $C = \{10101, 11001\} \subset V_5$. Here $k = 1$ and $n = 5$, and $r(C) = 0.2$. But C cannot correct any errors! If Bob receives 11101, he can't correct it.

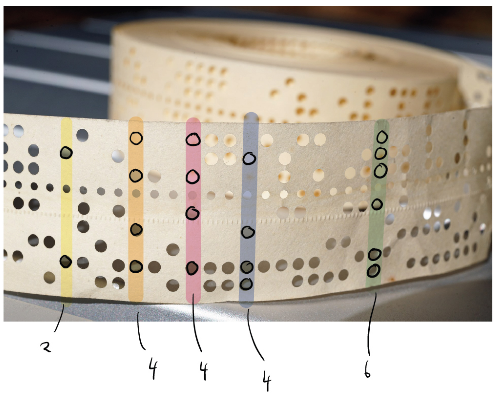
(Don't worry, we will actually see some good codes: starting immediately with the paper tape code below).

1.2. Example: The paper tape code

In the example above, to **correct** an error we had to send three times as much information. We can *detect* an error very easily by only sending a little extra information.

Early computers used hole-punched paper tapes to represent binary. Each tape had 8 'spaces', which could be punched or not punched, and:

- the first 7 spaces were filled with actual program data;
- the final space was used to 'check' the first 7 spaces were filled correctly.



More precisely, if the tape was filled out correctly:

...there should always be an even *number of holes.*

We can represent this with binary. Say a 1 represents a hole, and a 0 represents no hole. Then:

– If Alice wants to send the 7 bits 1001011, then there are already an even number of holes, so the 8th bit should be a 0.

– If Alice wants to send 1001010, then there are currently an odd number of holes, so the 8th bit should be a 1.

This is a code! Here $k = 7$, and $n = 8$. The code is

$$C = \Big\{(x_1, x_2, \dots, x_8) \in \{0,1\}^8 : x_1 + \cdots + x_8 \equiv 0 \pmod 2\Big\} \subset \{0,1\}^8.$$

The encoding map is[2]

$$\begin{aligned} \textbf{Enc} : V_7 &\longrightarrow V_8, \\ (x_1, \dots, x_7) &\longmapsto \Big(x_1, \dots, x_7, \big[x_1 + \cdots + x_7 \pmod 2\big]\Big). \end{aligned}$$

Example. • The message 0000000 is sent by **Enc** to 0000000-0.

[2]If n is an integer, we say $n \equiv 0 \pmod 2$ if n is even, and $n \equiv 1 \pmod 2$ is n is odd. So "$n \pmod 2$" is the remainder when dividing by 2.

- The message `0001000` is sent by **Enc** to `0001000-1` (because there is one 1 in the message).
- The message `0100010` is sent by **Enc** to `0100010-0` (because there are two 1s in the message).

Lemma 1.11. *Alice has a message* $(x_1, ..., x_7)$ *for Bob. She encodes it as* $(x_1, ..., x_7, x_8)$ *using the paper tape code, then sends it. Bob receives the message* $(y_1, ..., y_8)$.

If there is up to one error, Bob can always detect it.

Proof. Let

$$\begin{aligned} d &= x_1 + \cdots + x_8, \\ e &= y_1 + \cdots + y_8. \end{aligned}$$

Because Alice used the paper tape code, we know that d *is even.* If there is no error, then there is nothing to detect.

Suppose there is exactly one error; then either:

- A 0 in Alice's message is replaced by a 1 in the message Bob receives. Then $e = d + 1$.
- Or, a 1 for Alice is replaced by a 0 for Bob. Then $e = d - 1$.

In both cases, as d is even, we must have that e is odd, and hence

$$(y_1, ..., y_8) \notin C.$$

As the message Alice sent *was* in C, Bob knows there has been (at least one) error. □

Example. What can Bob deduce if he receives the following messages?

- `0100101-0`.

  Answer: This is not a codeword, as the sum of the digits is 3, which is odd. Therefore there must have been an error.
- `0100101-1`

  Answer: This is a codeword, so there might be no error...

  ...but there still *might* have been errors. Can you explain why if there has been an error, then there must have been *at least two* errors? (More precisely: can you explain why there must have been an even number of errors?)

This code can detect if there is a single error. Let's say Alice wants to send the code 0000000 (of length 7). They encode this as 0000000-0 using **Enc**. If there is an error in the first bit, then Bob receives 1000000-0. Bob knows that there has been an error, as this is not in the image of **Enc**.

Remark. The final bit in this code contains no actual information; it is only there to 'check' the validity of the other 7 bits. For this reason, it is often called a *check bit* (or *check digit*).

This is an example of an error-**detecting** code. However, it is *not* an error-**correcting** code. The computer knows there is an error, but it has no idea where the error occurs.

Remark. Even without correction, this can be very useful. If Bob receives

`the password is ‘meth’ -` ⚠ (an error has occurred) ⚠

then they can manually ask Alice to send the message again: i.e. ‘have you tried turning it off and on again?’

1.3. Example: Hamming’s code

Hamming was a programmer at US company Bell Laboratories, which used the paper tape code for their machines. One Friday in 1947, Hamming set a large (and slow) program running, and left for the weekend; but when he came back on Monday, the computer had detected an error, and terminated his program. Of course this was hugely frustrating, but it led to huge advances in error-correcting codes.

Hamming reasoned:

If the machine can detect an error, then surely it can correct it?

Of course, this is possible using the 3 times repetition code. However, given how slow programs were in those days, tripling the runtime was simply not an option. Hamming set to work trying to find the *minimum* amount of extra data required to correct an error. In a landmark 1950 paper, he found an answer by introducing *Hamming codes*. These will be a major focus later in the course.

Hamming’s new code took $k = 4$ and $n = 7$ (so required sending 75% extra information). Hamming also found a specific encoding map

$$\mathbf{Enc} : V_4 \longrightarrow V_7.$$

The image C of **Enc** is the set

$$\left\{ (x_1, \ldots, x_7) \in \{0,1\}^7 \;\middle|\; \begin{array}{l} x_1 + x_3 + x_5 + x_7 \equiv 0 \,(\operatorname{mod} 2) \\ x_2 + x_3 + x_6 + x_7 \equiv 0 \,(\operatorname{mod} 2) \\ x_4 + x_5 + x_6 + x_7 \equiv 0 \,(\operatorname{mod} 2) \end{array} \right\}.$$

This looks like it comes totally out of nowhere, but we’ll explain why this is really very natural when we study *linear codes* in later chapters.

Hamming’s code could detect up to 2 errors, and correct up to 1 error, which is the same as the 3 times repetition code, which required sending 200% extra information.

For comparison, here are some practical numbers highlighting the importance of Hamming’s code. Let’s suppose there’s an error on average once every 10,000 transmissions, and that a program is about 100,000 bits long. Then Hamming had to send 100,000 actual bits and around another 14,000 in check bits. The chance of sending this perfectly works out to be 0.001%! If the whole code takes one day to run, it would take an average of *265 years* to complete a single program without errors.

Let’s use the 3 times repetition code instead. Now there will be an error around one in 33 million transmissions, and the chance of completing without error leaps to 99.7%, so Hamming probably only has to run the code once. Now, though, the code requires 3 times as much information (so

300,000 transmissions) and will take three times as long, so it will take an average of 3 days to complete one program.

Hamming's genius was to find a much better error-correcting code. His code still has a 99.5% chance of completing, but he has to send only 1.75 times as much information (so 175,000 transmissions). In other words, the average time to complete a single program is about 1.75 days.

The savings here might not seem much for one program, but when we scale up, the difference becomes huge. If we were to run the code until it had succeeded 1,000 times, then:

- Not using any error-correction, we'd expect this to take over 265,000 years!
- Using a 3-time repetition code, we'd expect this to take 8 years and 3 months.
- Using Hamming's code, we'd expect 4 years and 10 months.

In other words, Hamming's code is essentially as accurate, but saves us 3 years and 5 months.

The following summarises the above, i.e. the probabilities/times required to send a 100,000 bit binary message, assuming that it takes 1 day to send 100,000 bits[3]:

Code used	**Bits sent**	**Prob. completion**	**Est. time (1 time)**	**Est. time (10^3 times)**
Paper tape	114,000	0.001%	265 yrs, 11 mths	265,915 years
$3\times$ repetition	300,000	99.7%	$\sim$3 days	8.3 years
Hamming's code	175,000	99.5%	$\sim$1.75 days	4.8 years

Hamming's code revolutionised computing and launched the entire area of error-correcting codes. One of the goals of this course is to describe why Hamming's code is actually very natural, explaining the mathematics behind it, and where all of these calculations actually come from.

1.4. $(*)$ More general codes

In this (non-examinable) section, we explore some other types of codes, including some that you can actually look at in the wild (e.g. credit card numbers). In particular, we look beyond *binary* codes, and allow more general alphabets:

Definition 1.12. An *alphabet* is a set of symbols $\mathcal{A} = \{x_1, x_2, ...\}$, which we use to construct *words* $\mathbf{x} = x_{i_1}x_{i_2}\dots x_{i_n}$.

Examples.

- $\mathcal{A} = \{$`a,b,c,...,z`$\}$. Then words are finite strings of letters, e.g. `kabehaofja`.
- $\mathcal{A} = \{0, 1\}$. Words are binary numbers/strings.
- $\mathcal{A} = \{0, 1, \dots, 9\}$. Words are decimal numbers.

The following are more general examples of the objects $\mathbf{Enc} : V_k \to C \subset V_n$ we saw above:

[3]Obviously it now takes a fraction of a second to do this, but 100,000 bits is only 12 kilobytes of data!

Definition 1.13.
- A *source* is a set $\mathcal{S}$ of "allowed words". (Think of this as being the set of messages one might want to send).
- Let C be a set of "code words". We call C a *code* (for $\mathcal{S}$) if C and $\mathcal{S}$ have the same cardinality.
- An *encoding map* is a bijective map
$$\mathbf{Enc} : \mathcal{S} \longrightarrow C.$$
and its inverse
$$\mathbf{Dec} : C \longrightarrow \mathcal{S},$$
is the *decoding map.*

When freed from the binary world, we get a much wider array of codes.

Example (Caesar ciphers)**.** Let $\mathcal{A} = \{\mathtt{a},\mathtt{b},\mathtt{c},\dots,\mathtt{z}\}$. We could take our source $\mathcal{S}$ to be "words in English", i.e.
$$\mathtt{hello} \in \mathcal{S}, \qquad \mathtt{helloxz} \notin \mathcal{S}.$$
We could take C to be "words after applying the Caesar cipher
$$\mathtt{a} \mapsto \mathtt{b}, \qquad \mathtt{b} \mapsto \mathtt{c}, \qquad \dots, \qquad \mathtt{z} \mapsto \mathtt{a},\text{"}$$
in which case $\mathtt{ifmmp} \in C$. The encoding map $\mathbf{Enc} : \mathcal{S} \to C$ here sends
$$\mathbf{Enc} : \mathtt{hello} \longmapsto \mathtt{ifmmp}.$$

Example 1.14 (The Luhn Algorithm for credit card numbers)**.** When you enter your credit card details into a website, there's always a short wait whilst you are transferred to your bank's website and your details are verified. Have you ever wondered why, when you type your number incorrectly, a website knows *immediately* that there's a problem, without even contacting the bank? This is because of the *Luhn algorithm*, a clever 'check-digit' code.

Credit card networks see huge amounts of traffic every minute. They don't want to waste time looking for numbers that don't exist, so credit card numbers are chosen in a clever way. In particular, only certain 16 digit numbers are 'valid' credit card numbers. In other words, we are taking
$$\mathcal{A} = \{0,1,\dots,9\}, \qquad C = \{\text{valid card numbers}\} \subsetneq \mathcal{A}^{16}.$$
We want to choose this set C in such a way that no two valid numbers are 'close', so if you type, say, one number incorrectly, or switch around two numbers, then the number you input will no longer be valid.

We'll denote a credit card number by the 16 digits
$$x_1x_2x_3x_4 - x_5x_6x_7x_8 - x_9x_{10}x_{11}x_{12} - x_{13}x_{14}x_{15}x_{16}.$$

Attempt 1: We can do the naive analogue of the paper tape code: say a credit card number is valid if
$$x_1 + x_2 + \dots + x_{16} \equiv 0 \,(\mathrm{mod}\, 10).$$

If we change a single digit, e.g. if we type one digit wrong, then this property is broken, so we can detect single-digit errors. However, another common error might be a 'transposition', to swap two consecutive digits. Our first attempt doesn't catch that.

Attempt 2: We can detect transpositions by *weighting* the sum, and ask instead that a number is valid if

$$2x_1 + x_2 + 2x_3 + x_4 \cdots + 2x_{15} + x_{16} \equiv 0 \pmod{10}.$$

If we swap e.g. x_1 and x_2, then this sum changes by $x_2 - x_1$, and this is $0 \pmod{10}$ if and only if $x_2 = x_1$. So now we catch all transposition errors. However, we no longer detect all the single-digit errors! If $x_1 = 1$, and we mistype it as 6, we cannot catch this error, as $2 \cdot 1 \equiv 2 \cdot 6 \pmod{10}$. In this regime, you can have two valid numbers differing by only one digit.

Attempt 3: What we actually use is the *Luhn algorithm*, a simple modification which detects all single-digit errors and all transpositions except $09 \leftrightarrow 90$.

The problem above was that $2x_1 \equiv 2(x_1 + 5) \pmod{10}$, so any single digit error where we mix up e.g. 1 & 6, or 2 & 7, or 8 & 3, etc. was not caught. The Luhn algorithm gets round this as follows:

For each $i \in \{1, ..., 16\}$, let $Y_i :=$ the sum of the digits in $2x_i$ (so if $x_i = 3$, then $Y_i = 6$; or if $x_i = 7$, then $Y_i = 5$, the sum of the digits in 14).

The credit card number is valid if the sum

$$Y_1 + x_2 + Y_3 + x_4 + \cdots + Y_{15} + x_{16} = \sum_{i=1}^{8} Y_{2i-1} + \sum_{i=1}^{8} x_{2i}.$$

is divisible by 10.

An example of a valid credit card number is: $4727 - 7523 - 6474 - 0928$:

x_i		4	7	2	7		7	5	2	3		6	4	7	4		0	9	2	8	$\sum$
$2x_i$		8	7	4	7		14	5	4	3		12	4	14	4		0	9	4	8	
Y_i		8	7	4	7		5	5	4	3		3	4	5	4		0	9	4	8	80

Exercise. Go and find your credit or debit card. Compute the Luhn weighted sum, and smile to yourself when it is divisible by 10.

Example (ISBN – International book standard number)**.** Books have ISBN codes, the number above and below the bar code:

ISBN 978-3-16-148410-0

9 783161 484100

Here the alphabet is $\mathcal{A} = \{0, 1, \dots, 9\}$.

As with credit card numbers, it's desirable for any two (valid) ISBN codes to be 'far' from each other, so that you don't accidentally buy the wrong book by entering the wrong ISBN. A valid (new) ISBN code is a 13 digit code of the form

$$x_1x_2x_3 - x_4 - x_5x_6x_7 - x_8x_9x_{10}x_{11}x_{12} - x_{13}$$

with $x_i \in 0, 1, \dots, 9$ such that

$$\sum_{i=0}^{6} x_{2i+1} + 3\sum_{i=1}^{6} x_{2i} = x_1 + 3x_2 + x_3 + \cdots + 3x_{12} + x_{13} \equiv 0 \pmod{10}. \tag{1.1}$$

For example 978 - 3 - 16 - 148410 - 0 (as above) is a valid code.

Remark. As with the paper tape code, for credit card and ISBN numbers the final digit does not represent any "real" information (like bank or publisher etc.), but is only a check digit, in that their sole purpose is to determine whether or not the other numbers are correct.

To put this in the abstract language of codes: for credit cards, the source is $\mathcal{A}^{15}$, and the code is the subset of $\mathcal{A}^{16}$ where the Luhn condition holds. For ISBN numbers, the source is $\mathcal{A}^{12}$, and the code is the subset of $\mathcal{A}^{13}$ where (1.1) holds.

Chapter 2

Coding 101: Fundamentals

In this chapter, we'll introduce some basic mathematical concepts with which to study codes. Throughout, we'll continue to assume that Alice is sending messages to Bob. When we talk about using a code C, we assume that both of them have agreed in advance to use this specific code (so they both know what the codewords are).

2.1. Error detection and correction

For a given code C, we want to know:

(1) How many errors can Bob detect?

(2) How many errors can Bob correct?

In both cases, we want to know how many errors Bob is *guaranteed* to be able to detect or correct.

Definition 2.1. We say a code C is:

- *e-error detecting* if Bob can *always* detect (up to and including) e errors in arbitrary positions.
- *e-error correcting* if Bob can *always* correct (up to and including) e errors in arbitrary positions.

Example 2.2. Consider the 3 times repetition code, with $C = \{000, 111\}$. We will assume the probability of an error is < 0.5, so that it is always more likely a bit is sent correctly than incorrectly. If Bob receives 101, then, then 1**1**1 (one error) is more likely to have been sent than **0**0**0** (two errors).

Consider all the possibilities on sending 000:

Sent	Received	Errors	Likely sent?	Detect?	Correct?
000	000	0	000	N/A	N/A
000	001		000	✓	✓
000	010	1	000	✓	✓
000	100		000	✓	✓
000	110		111	✓	✗
000	101	2	111	✓	✗
000	011		111	✓	✗
000	111	3	111	✗	✗

In the first line, there are no errors, so there is nothing to detect or correct and the final columns are vacuous.

The same table appears if we send 111 (with all 1s and 0s interchanged). We see that:

- we can correctly detect if there are at most 2 errors,
- and we can correctly correct if there is at most 1 error.

Thus C is 2-error-detecting and 1-error-correcting.

Example 2.3. Here's an example of a *catastrophically bad* code. Let

$$C = \{\texttt{000}, \texttt{001}, \texttt{101}, \texttt{111}\} \subset V_3.$$

Then $k = 2$ (because $\#C = 2^k = 4$), and $n = 3$.

How many errors can C detect and correct? Well, we look at the analogue of the table above for this code. Suppose Alice sends `000`. Then:

Sent	Received	Errors	Likely sent?	Detect?	Correct?
000	000	0	000	N/A	N/A
000	100		?	✓	✗
000	010	1	000	✓	✓
000	001		001	✗	✗

In the second line, Bob detects an error because 100 is not a codeword. However Bob could reasonably decode 100 as 000 or 101 – both are just one error from 101, so neither is more likely than the other, and we cannot correct this one error.

In the third line, we can both detect and correct, as 010 is only 1 error from 000, but is at least two errors from each of 001, 101 and 111.

In the fourth line, we've had a shocker: there's only one error, but we hit another codeword, so Bob can't even *detect* an error!

Since C is not guaranteed to even detect one error, it is 0-error-detecting. Similarly it is 0-error-correcting.

Example. Let C be the paper tape code from §1.2. In Lemma 1.11, we saw that C always detects up to 1 error. It cannot always detect 2 errors, as for example if Alice sends `00000000` and Bob receives `11000000`, he doesn't know there is an error. Thus C is 1-error-detecting.

The paper tape code can never correct errors, so it is 0-error-correcting.

We can also consider more general repetition codes.

Example 2.4 (Binary repetition codes)**.** Let m be a positive integer. Define the m-times repetition code to be

$$C = \Big\{ \underbrace{00\cdots00}_{m},\ \underbrace{11\cdots11}_{m} \Big\} \subseteq V_m.$$

In other words, we are repeating each individual bit m times.

We usually decode this using *majority decoding*, which is exactly as you'd expect. If we receive a message, then:

- if there are more 0s than 1s: decode as $(0, ..., 0)$.
- if there are more 1s than 0s: decode as $(1, ..., 1)$.
- if there are equally many 1s and 0s: decode as either $(0, ..., 0)$ or $(1, ..., 1)$ "randomly".

Exercise. Explain why this code is $(m-1)$-error-detecting and $\lfloor \frac{m-1}{2} \rfloor$-error-correcting.

2.2. Binary symmetric channels

We now introduce a simple, but convenient, mathematical model of a noisy channel.

Definition 2.5. Let $0 \leqslant p \leqslant 1/2$. A *binary symmetric channel* (or *BSC*) with error probability p transmits individual bits (i.e. either 0 or 1), and:

- in each transmission, the probability that there is an error is p,
- and the probability of an error in a given transmission is independent of all previous and future transmissions.

Remark. The BSC is a good model for most means of digital transmissions, e.g. mobile phones, WiFi, optical fibers, writing to compact or magnetic discs or flash drives. Examples of error probabilities are $p = 10^{-5}$ (compact disk) and $p = 10^{-9}$ (magnetic disks).

In practice, the error probabilities are not always independent since physical errors (cutting a wire, a mechanical error in a hard drive, electrical failure etc.) usually occur in clusters. However, because of the large amount of information being transmitted, this is still a good approximation. Also, certain errors (e.g. due to quantum mechanical effects) are indeed truly random.

We can compute how likely errors are in a BSC. Recall notation from probability: we let

$$\mathbf{P}(A \mid B) \coloneqq \text{probability of } A \text{ happening given } B \text{ has already happened.}$$

For a BSC, we see that

$$\begin{aligned} \mathbf{P}\big(\text{receive } 0 \mid \text{send } 0\big) &= \mathbf{P}\big(\text{receive } 1 \mid \text{send } 1\big) = 1-p \qquad &\text{(no error)},\\ \mathbf{P}\big(\text{receive } 0 \mid \text{send } 1\big) &= \mathbf{P}\big(\text{receive } 1 \mid \text{send } 0\big) = p \qquad &\text{(error)}. \end{aligned}$$

Example. Suppose we send the word `0000` over a BSC of error probability p.

- What's the probability there's no error? Well, in this case the transmissions must be

$$(\text{send } 0,\ \text{receive } 0)\&(\text{send } 0,\ \text{receive } 0)\&(\text{send } 0,\ \text{receive } 0)\&(\text{send } 0,\ \text{receive } 0)$$

  which happens with probability

$$(1-p)\times(1-p)\times(1-p)\times(1-p).$$

  So the probability is $(1-p)^4$.

- What's the probability there's one error? Well, either we received `1000`, `0100`, `0010`, or `0001`. The probability of receiving `1000` is $p\times(1-p)\times(1-p)\times(1-p) = p(1-p)^3$ (and ditto for the others). Thus the probability of exactly one error is

$$p(1-p)^3 + (1-p)p(1-p)^2 + (1-p)^2p(1-p) + (1-p)^3p = 4p(1-p)^3.$$

- The probability of two errors? Well now we receive one of six possibilities: `1100`, `1010`, `1001`, `0110`, `0101`, `0011`. The probability of receiving each is $p^2(1-p)^2$. So the probability is

$$P(2\text{ errors}) = 6p^2(1-p)^2.$$

More generally:

Lemma 2.6. *Alice sends a word* $\mathbf{x}$ *of length* n *through a BSC with error probability* p.

(i) The probability that Bob receives $\mathbf{x}$*, i.e. it is transmitted with no errors, is*

$$\mathbf{P}(\textit{no errors}) = (1-p)^n,$$

and the probability that he does not receive $\mathbf{x}$ *(so there is* some *error) is*

$$\mathbf{P}(\textit{an error}) = 1-(1-p)^n.$$

(ii) Let $0 \leqslant d \leqslant n$ *be an integer. The probability that* $\mathbf{x}$ *is transmitted with exactly* d *errors (in any bits) is*

$$\mathbf{P}(\textit{exactly } d \textit{ errors}) = \binom{n}{d} p^d(1-p)^{n-d}.$$

Proof. (i) Each bit is transmitted correctly independently with probability $1-p$, and there are n bits. The result follows from basic probability: if A and B are independent, then $P(A\&B) = P(A)\times P(B)$.

(ii) The probability that there are errors in exactly d *given* bits, for example the first d bits, is

$$p^d(1-p)^{n-d}.$$

There are

$$\binom{n}{d} = \frac{n(n-1)\cdots(n-d+1)}{d!}$$

different ways of choosing exactly d different bits. The result follows. $\square$

Remark. A sanity check for working with probabilities is to check that when you sum the probabilities of all the possible cases, you get 1. In this case, we have

$$\begin{aligned}1 &= \mathbf{P}(0 \text{ errors}) + \mathbf{P}(1 \text{ error}) + \mathbf{P}(2 \text{ errors}) + \cdots + \mathbf{P}(n \text{ errors}) \\ &= (1-p)^n + \binom{n}{1} p(1-p)^{n-1} + \binom{n}{2} p^2(1-p)^{n-2} + \cdots + p^n = \Big[(1-p)+p\Big]^n = 1^n = 1,\end{aligned}$$

using the binomial theorem

$$(a+b)^n = \sum_{k=0}^{n} \binom{n}{k} a^k b^{n-k}.$$

Rather than exact formulas, we are usually interested in approximations. For example, in the BSC above, the probability there is an error was

$$1-(1-p)^n = -\sum_{k=1}^{n} \binom{n}{k} (-p)^n = np - \frac{n^2-n}{2}p^2 + \cdots .$$

If we assume p is very small relative to n, then approximately we have

$$\mathbf{P}(\text{an error}) = 1-(1-p)^n \approx np.$$

2.3. Decoding schemes

So far in our model, there has been a mysterious step called 'error-correction'. We saw the example of how to do this for repetition codes, but we've not explored this further. In general, this requires decoding schemes.

Definition 2.7. Let C be a code. A *decoding scheme* for C is a method of error-correction.

This fits into our model as follows:

$$\begin{tikzcd}
\textbf{Alice} & \textbf{Bob} \\
\boxed{\text{sourceword} \in V_k} \arrow[dd, "\textbf{Enc}"] & \boxed{\text{sourceword } \mathbf{w} \in V_k} \\
 & \boxed{\textbf{codeword } \mathbf{c} \in C \subset V_n} \arrow[u, "\textbf{Dec}"'] \\
\boxed{\textbf{codeword} \in C \subset V_n} \arrow[r, dashed, "\text{Noisy channel}"] & \boxed{\text{received word } \mathbf{y} \in V_n} \arrow[u, "\text{decoding scheme}"']
\end{tikzcd}$$

We are considering this from Bob's point of view; the only information he knows is that Alice has sent *some* element of C, and the word $\mathbf{y}$ that he has received.

A fundamental question is:

What's the best possible decoding scheme for a given code C? That is, what's the best method of error-correction for C?

We now consider two abstract decoding schemes. These are *theoretical* methods of decoding; we don't yet worry about how to actually decode in practice.

2.3.1. Ideal-observer decoding. The 'best possible' decoding scheme answers the question:

Having received $\mathbf{y}$*, which codeword was most likely to have been sent?*

More precisely, this is:

Definition 2.8 (Ideal-observer decoding)**.**
Find the $\mathbf{c} \in C$ *that maximises the probability* $\mathbf{P}\big(\textit{send } \mathbf{c} \mid \textit{receive } \mathbf{y}\big)$.

That is, find a codeword $\mathbf{c} \in C$ such that

$$\mathbf{P}\big(\text{send } \mathbf{c} \mid \text{receive } \mathbf{y}\big) \geqslant \mathbf{P}\big(\text{send } \mathbf{c}' \mid \text{receive } \mathbf{y}\big) \qquad \forall \mathbf{c}' \in C.$$

This is *ideal-observer* decoding.

In practice ideal-observer is essentially *impossible* to implement without more information! One needs to know all possible probabilities $\mathbf{P}\big(\text{send } \mathbf{c}' \mid \text{receive } \mathbf{y}\big)$. To compute this, you need to know the probability that $\mathbf{c}'$ was sent in the first place. But we don't know what this is without extra assumptions; maybe the probability varies depending on what was sent previously, or some codewords are more likely to be sent than others.

Example 2.9. Consider the 3 times repetition code $C = \{\texttt{000}, \texttt{111}\}$, sent over a BSC with error probability $p = 0.1$. Suppose Bob receives $\mathbf{y} = \texttt{010}$. Then maximising $\mathbf{P}\big(\text{send } \mathbf{c} \mid \text{receive } \texttt{010}\big)$ is the same as finding the maximum of the two numbers

$$\begin{aligned}(\text{when } \mathbf{c} = \texttt{000})\text{:}\quad P(\text{send } \texttt{000} \text{ and receive } \texttt{010}) &= P(\text{send } \texttt{000}) \times \Big[(1-p)\cdot p \cdot (1-p)\Big] \\ &= 0.081 \cdot P(\text{send } \texttt{000}),\end{aligned}$$

and

$$\begin{aligned}(\text{when } \mathbf{c} = \texttt{111})\text{:}\quad P(\text{send } \texttt{111} \text{ and receive } \texttt{010}) &= P(\text{send } \texttt{111}) \times \Big[p \cdot (1-p) \cdot p\Big] \\ &= 0.009 \cdot P(\text{send } \texttt{111}).\end{aligned}$$

Now, what's the maximum? With the information I've given you...
.... *We don't know!!* We can't compute this without knowing $P(\text{send } \texttt{000})$ and $P(\text{send } \texttt{111})$. I've not told you what these are: maybe in this message `111` is much more likely to be sent than `000`, or maybe at this point of time the last 5 transmissions have been `000` and it should be `000` again.

2.3.2. Maximum-likelihood decoding. We can avoid all these subtleties by asking a different question:

Definition 2.10 (Maximum-likelihood decoding)**.**
Find the $\mathbf{c} \in C$ *that maximises* $\mathbf{P}\big(\textit{receive } \mathbf{y} \mid \textit{send } \mathbf{c}\big)$.

That is, find a codeword $\mathbf{c} \in C$ such that

$$\mathbf{P}\big(\text{receive } \mathbf{y} \mid \text{send } \mathbf{c}\big) \geqslant \mathbf{P}\big(\text{receive } \mathbf{y} \mid \text{send } \mathbf{c}'\big) \qquad \forall \mathbf{c}' \in C.$$

This is *maximum-likelihood decoding.*

Example 2.11. Consider again $C = \{\texttt{000}, \texttt{111}\}$ sent over a BSC with error probability $p = 0.1$, and suppose Bob receives `010`. Then

$$\begin{aligned} \mathbf{P}\big(\text{receive } \texttt{010} \mid \text{send } \texttt{000}\big) &= (1-p)\cdot p \cdot (1-p) = 0.9^2 \cdot 0.1 = 0.081, \\ \mathbf{P}\big(\text{receive } \texttt{010} \mid \text{send } \texttt{111}\big) &= p \cdot (1-p) \cdot p = 0.1^2 \cdot 0.9 = 0.009, \end{aligned}$$

so maximum-likelihood decoding corrects `010` to `000` (as you would expect!).

2.3.3. Ideal-observer vs. Maximum-likelihood.

It is natural to ask: when is ideal-observer decoding the same as maximum-likelihood?

Definition 2.12 (Memoryless BSCs)**.** We say a BSC is *memoryless* if, at any time, all codewords are equally likely to be sent, i.e.

$$\mathbf{P}\big(\mathbf{c} \text{ sent}\big) = \frac{1}{\#C}.$$

In other words:

- all source words appear with the same frequency, and
- the probability of a word being sent is not influenced by the words previously sent.

Remark. Not all interesting symmetric channels will be memoryless. For example, in the English language, words like 'the' are much more frequent than others, and words are rarely repeated consecutively. If I've started sending the message 'there was an extraordinary ruckus in...' then the next word is very unlikely to be 'xylophone' or 'salad',

However, in practice this assumption is not as bad as it seems! The sheer volume of data can mean a channel 'averages out' to being memoryless; and one can achieve 'approximately memoryless' channels by using a compression algorithm, e.g. "zip" a file.

Proposition 2.13. *In a memoryless BSC, ideal-observer decoding and maximum-likelihood decoding are the same.*

Proof. Recall Bayes' theorem for relative probability:

$$\mathbf{P}(A \mid B) = \frac{\mathbf{P}(A)}{\mathbf{P}(B)} \cdot \mathbf{P}(B \mid A).$$

Bob has received the word $\mathbf{y}$, which is thus fixed. Bayes says that for any $\mathbf{c}$, we have

$$\begin{aligned}\mathbf{P}\big(\text{receive } \mathbf{y} \mid \text{send } \mathbf{c}\big) &= \frac{\mathbf{P}(\text{receive } \mathbf{y})}{\mathbf{P}(\mathbf{c} \text{ sent})} \cdot \mathbf{P}\big(\text{send } \mathbf{c} \mid \text{receive } \mathbf{y}\big) \\ &= \Big[\#C \cdot \mathbf{P}(\mathbf{y} \text{ received})\Big] \cdot \mathbf{P}\big(\text{send } \mathbf{c} \mid \text{receive } \mathbf{y}\big),\end{aligned}$$

using that $\mathbf{P}(\mathbf{c} \text{ sent}) = (\#C)^{-1}$ because the BSC is memoryless. Let

$$A(\mathbf{y}) := \#C \cdot \mathbf{P}(\mathbf{y} \text{ received}) > 0,$$

noting this is a positive number[a] depending only on $\mathbf{y}$, i.e. it is constant as we vary $\mathbf{c}$. (This constancy crucially depends on the fact that the BSC is memoryless).

Now for any $\mathbf{c}' \in C$, as $A(\mathbf{y}) > 0$, we have

$$\begin{aligned}&\Big[\mathbf{P}\big(\text{send } \mathbf{c} \mid \text{receive } \mathbf{y}\big) \geqslant \mathbf{P}\big(\text{send } \mathbf{c}' \mid \text{receive } \mathbf{y}\big)\Big] \\ &\qquad \iff \Big[A(\mathbf{y})\mathbf{P}\big(\text{send } \mathbf{c} \mid \text{receive } \mathbf{y}\big) \geqslant A(\mathbf{y})\mathbf{P}\big(\text{send } \mathbf{c}' \mid \text{receive } \mathbf{y}\big)\Big] \\ &\qquad \iff \Big[\mathbf{P}\big(\text{receive } \mathbf{y} \mid \text{send } \mathbf{c}\big) \geqslant \mathbf{P}\big(\text{receive } \mathbf{y} \mid \text{send } \mathbf{c}'\big)\Big].\end{aligned}$$

In other words,

$$\mathbf{c} \text{ maximises } \mathbf{P}\big(\text{send } \mathbf{c} \mid \text{receive } \mathbf{y}\big) \iff \mathbf{c} \text{ maximises } \mathbf{P}\big(\text{receive } \mathbf{y} \mid \text{send } \mathbf{c}\big),$$

and the two decoding schemes are the same. □

[a] $A(\mathbf{y}) > 0$ since $\#C$ is positive, and as Bob has *already* received $\mathbf{y}$, we know $\mathbf{P}(\mathbf{y} \text{ received})$ cannot be 0!

Exercise. If we do not assume the BSC is memoryless, ideal-observer and maximum-likelihood are genuinely different!! For example, consider the BSC from §1.1, i.e. $0 \mapsto 000, 1 \mapsto 111$, with error probability p. We suppose that actually 0 is the *only* word that is ever sent: that is,

$$\mathbf{P}(0 \text{ sent}) = 1, \qquad \mathbf{P}(1 \text{ sent}) = 0.$$

(In other words, this BSC is as far from memoryless as possible). In this example:[1]

(i) Investigate how you would decode each of the received words 000, 100, 110, 111 if you used:

(a) ideal-observer decoding;

(b) maximum-likelihood decoding.

(ii) Explain why ideal-observer is *theoretically* better than maximum-likelihood for this BSC.

Remark. This exercise highlights a useful tool when trying to get intuition for probability. Ask yourself: what happens in the most extreme cases? (Here, the most extreme possibility is that 0 is

[1] *Solution: (i) We always have* $\mathbf{P}(0 \text{ sent} \mid \mathbf{y} \text{ received}) = 1$ *and* $\mathbf{P}(1 \text{ sent} \mid \mathbf{y} \text{ received}) = 0$*, no matter what* $\mathbf{y}$ *is. So ideal-observer decoding* always *decodes any message as 0. In particular, 111 is decoded as 0. However, we have*

$$\mathbf{P}(111 \text{ received} \mid 1 \text{ sent}) = (1-p)^3 > p^3 = \mathbf{P}(111 \text{ received} \mid 0 \text{ sent}),$$

so maximum-likelihood decoding will decode the message 111 as 1.

(ii) Since the only message that can be sent is 0, in a perfect system everything should be decoded as 0. This is what ideal-observer does, whilst maximum-likelihood will incorrectly decode the messages 111, 101, 110 and 011 as 1, even though the message 1 is never sent.

always sent, and 1 is never sent. Another extreme case is when the error probability is 0, that is, the BSC is literally perfect. In these cases you should get predictable outcomes).

Readers may have already identified a third decoding scheme, that we have not yet mentioned. A natural scheme, which we have already been subconsciously using in most of our examples, is to attempt to minimise errors; that is, to decode a received word as the codeword for which the fewest errors has occurred. Having set up our theoretical framework, this third scheme will be the focus of the next chapter.

Chapter 3

Hamming distance

Introduced by Hamming in his foundational 1950 paper *Error detecting and error correcting codes*, the notion of *Hamming distance* is a mathematical description of 'number of errors'. In this chapter, we'll introduce Hamming distance. We'll use it to describe another decoding scheme, the *nearest-neighbour* scheme. We'll also introduce the *minimum distance* of a code C, an important invariant that totally describes how many errors it can detect and correct (under maximum-likelihood decoding).

3.1. First properties of Hamming distance

Hamming distance encodes when two codewords are 'close'.

Definition 3.1 (Hamming distance)**.** Let

$$\mathbf{x} = x_1 \cdots x_n \qquad \text{and} \qquad \mathbf{y} = y_1 \cdots y_n \in V_n$$

be two binary strings of length n.

The *Hamming distance* between $\mathbf{x}$ and $\mathbf{y}$, denoted $d(\mathbf{x}, \mathbf{y})$, is the number of i's with $x_i \neq y_i$.

Example. The distance between $\mathbf{x} = 100110$ and $\mathbf{y} = 001010$ is $d(\mathbf{x}, \mathbf{y}) = 3$ – in the following, we mark the differing digits in bold:

$$\begin{array}{ccc} \mathbf{x} & = & (\mathbf{1}, \;\; 0, \;\; \mathbf{0}, \;\; \mathbf{1}, \;\; 1, \;\; 0) \\ \mathbf{y} & = & (\mathbf{0}, \;\; 0, \;\; \mathbf{1}, \;\; \mathbf{0}, \;\; 1, \;\; 0) \end{array}$$

Hamming distance satisfies the axioms of the usual mathematical notion of distance – it is a *metric*, i.e.:

Lemma 3.2. *We have:*

(i) $d(\mathbf{x}, \mathbf{y}) = 0$ *if and only if* $\mathbf{x} = \mathbf{y}$*;*

(ii) $d(\mathbf{x}, \mathbf{y}) = d(\mathbf{y}, \mathbf{x})$*; and*

(iii) $d(\mathbf{x}, \mathbf{z}) \leq d(\mathbf{x}, \mathbf{y}) + d(\mathbf{y}, \mathbf{z})$ (triangle inequality).

Proof. (i) and (ii) follow immediately from the definition.

(iii) For the triangle inequality, suppose

$$d(\mathbf{x},\mathbf{y}) = r, \qquad d(\mathbf{y},\mathbf{z}) = s.$$

This means we need to change r coordinates in $\mathbf{x}$ to obtain $\mathbf{y}$, and then s coordinates in $\mathbf{y}$ to obtain $\mathbf{z}$. In particular, to get from $\mathbf{x}$ to $\mathbf{z}$, at most $r+s$ coordinates have been changed. $\square$

Exercise. Calculate the Hamming distance between the three words $\mathbf{x} = 1010101$, $\mathbf{y} = 1111110$ and $\mathbf{z} = 0000001$ and verify the triangle inequality.

Exercise. Find all words $\mathbf{y} \in V_5$ that satisfy $d(\mathbf{y}, 00000) \leq 3$.

3.2. Nearest-neighbour decoding

We saw, in the last chapter, two purely theoretical decoding schemes:

- *ideal-observer*, finding $\mathbf{c}$ that maximises $\mathbf{P}\big(\text{send } \mathbf{c} \mid \text{receive } \mathbf{y}\big)$. This is the best decoding scheme that is theoretically possible.
- *maximum-likelihood*, finding $\mathbf{c}$ that maximises $\mathbf{P}\big(\text{receive } \mathbf{y} \mid \text{send } \mathbf{c}\big)$.

Proposition 2.13 said that if a channel is memoryless, the two schemes coincide (though this is not true in general).

A third natural decoding scheme would ask to minimise the number of errors. But observe:

Lemma 3.3. *Suppose Alice sends a codeword* $\mathbf{c}$ *over a channel, and Bob receives* $\mathbf{y}$*. The number of errors is exactly the Hamming distance* $d(\mathbf{c},\mathbf{y})$*.*

Example. Alice sends the codeword $\mathbf{c} = 01010$. Bob receives $\mathbf{y} = 01110$. Then there has been one error (in the third digit), and the Hamming distance is $d(\mathbf{c},\mathbf{y}) = 1$.

Thus minimising the number of errors is the same as minimising Hamming distance, motivating:

Definition 3.4 (Nearest-neighbour decoding)**.** Let C be a binary code. Under *nearest-neighbour decoding*, we decode a received word $\mathbf{y}$ as a codeword $\mathbf{c} \in C$ that minimises $d(\mathbf{y},\mathbf{c})$, i.e.

$$d(\mathbf{y},\mathbf{c}) \leqslant d(\mathbf{y},\mathbf{c}') \qquad \forall \mathbf{c}' \in C.$$

If our channel is a BSC with error probability $p < \frac{1}{2}$, then this is just maximum-likelihood in disguise!

Theorem 3.5. *Over a BSC with error probability* $p < \frac{1}{2}$,

$$\Big[\textit{Maximum-likelihood decoding}\Big] = \Big[\textit{Nearest-neighbour decoding}\Big].$$

Proof. Suppose we have received $\mathbf{y}$. By definition, maximum-likelihood will decode $\mathbf{y}$ as the codeword $\mathbf{c}$ which maximises the probability $\mathbf{P}\big(\text{receive } \mathbf{y} \mid \text{send } \mathbf{c}\big)$. When $\mathbf{y}$ has length n, we compute that this is

$$\begin{aligned}\mathbf{P}(\mathbf{y} \text{ received} \mid \mathbf{c} \text{ sent}) &= p^{\#(\text{errors from } \mathbf{c} \text{ to } \mathbf{y})} \cdot (1-p)^{n-\#(\text{errors from } \mathbf{c} \text{ to } \mathbf{y})} \\ &= p^{d(\mathbf{y},\mathbf{c})} \cdot (1-p)^{n-d(\mathbf{y},\mathbf{c})} \\ &= (1-p)^n \cdot \left(\frac{p}{1-p}\right)^{d(\mathbf{y},\mathbf{c})}.\end{aligned}$$

Since $p < \frac{1}{2}$ it follows that $\frac{p}{1-p} < 1$. Hence the above expression is therefore largest (giving maximum-likelihood) when $d(\mathbf{c}, \mathbf{y})$ is smallest (giving nearest-neighbour). □

Remark 3.6. This says that for *memoryless* BSCs, nearest-neighbour is the best possible decoding algorithm! However, for complicated codes it may be very difficult to actually *find* the nearest neighbour (as we definitely do not want to compute $d(\mathbf{y}, \mathbf{c})$ for all $\mathbf{c}$).

In the future:

unless a specific decoding scheme is mentioned, we always assume that nearest-neighbour decoding is used.

(Sometimes, genuinely different decoding schemes turn out to be better; for example, this is true for the so-called Reed-Muller codes. We'll introduce these in Chapter 5, and show that Reed–Muller decoding is not nearest-neighbour in Proposition 5.9).

3.3. Hamming distance and error-correction

The following easy observations are useful. Suppose Alice sends a codeword $\mathbf{c}$ and Bob receives $\mathbf{y}$. Then:

(ED) *Bob knows that an error has occurred*
$\iff$ *the received word* $\mathbf{y}$ *is not a codeword.*

(EC) *Using nearest-neighbour, Bob can correct* $\mathbf{y}$ *to* $\mathbf{c}$
$\iff$ $\mathbf{c}$ *is the* unique *codeword closest to* $\mathbf{y}$*.*

Recall the notions of e-error-detecting and e-error-correcting from Definition 2.1. We can now formalise these properties using Hamming distance (and in particular, Hamming balls).

Definition 3.7. Let $\mathbf{x} \in V_n$ be a binary string of length n, and let $r \geqslant 0$ be an integer. The *Hamming ball (of centre x and radius r)* is the set

$$B(\mathbf{x}, r) := \{\mathbf{y} \in V_n : d(\mathbf{x}, \mathbf{y}) \leqslant r\}.$$

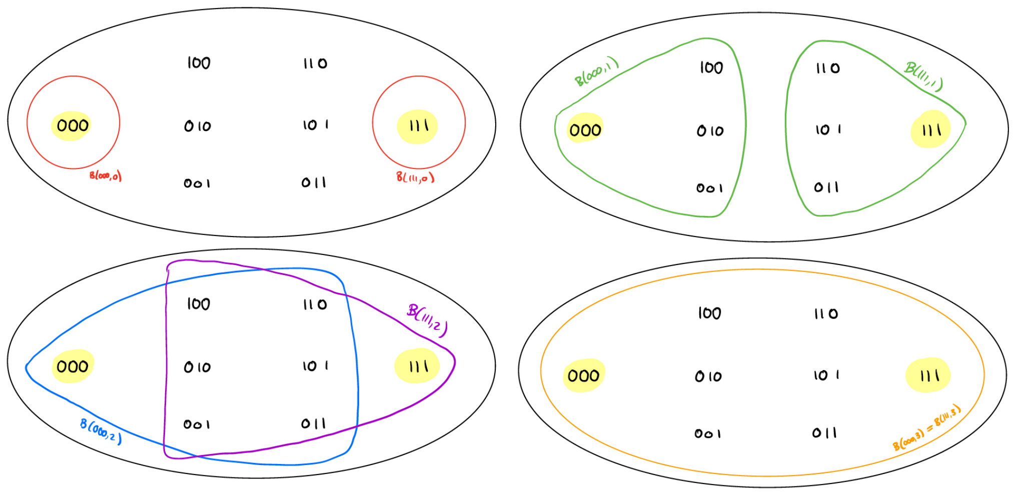
Note that when we think about (V_n, d) as a metric space, then $B(\mathbf{x}, r)$ is the closed ball of radius r around $\mathbf{x}$ (justifying the terminology).

Example. Consider $n = 4$, and $\mathbf{x} = 0101$. Then

$$\begin{aligned}
B(\mathbf{x}, 0) &= \{0101\},\\
B(\mathbf{x}, 1) &= \{0101\} \cup \{1101, 0001, 0111, 0100\},\\
B(\mathbf{x}, 2) &= \{0101\} \cup \{1101, 0001, 0111, 0100\} \cup \{1001, 1111, 1100, 0011, 0000, 0110\},
\end{aligned}$$

and so on. Here note that $B(\mathbf{x}, 2)$ is the union of the three sets where $d(\mathbf{x}, \mathbf{y}) = 0$, $d(\mathbf{x}, \mathbf{y}) = 1$ and $d(\mathbf{x}, \mathbf{y}) = 2$ respectively.

Example. The following illustrates the Hamming balls around 000 and 111 in V_3. The ball drawn are $B(000, r)$ and $B(111, r)$, where $r = 0, 1, 2, 3$ in the top left, top right, bottom left and bottom right respectively.



Now, remember we showed that the code $C = \{000, 111\}$ is 2-error-detecting and 1-error-correcting. We can read this off from the pictures above.

Lemma 3.8. *Using nearest-neighbour decoding, a code $C \subset V_n$ is e-error detecting if and only if*

(∗) for every $\mathbf{c} \in C$, we have
$$B(\mathbf{c}, e) \cap C = \{\mathbf{c}\},$$
that is, the Hamming ball $B(\mathbf{c}, e) \subset V_n$ contains no other codewords in C.

Proof. Suppose (∗) holds. Suppose $\mathbf{c}$ is sent and received as $\mathbf{y}$. If there are $\leqslant e$ errors, then

$$d(\mathbf{c}, \mathbf{y}) \leqslant e \quad \Rightarrow \quad \mathbf{y} \in B(\mathbf{c}, e),$$

so by (∗) $\mathbf{y}$ cannot be another codeword. By (ED) above, C is thus e-error-detecting.

Conversely, suppose C is e-error-detecting. Fix $\mathbf{c}$ and suppose $\mathbf{c}'$ is a different codeword with $\mathbf{c}' \in B(\mathbf{c}, e)$. By definition
$$d(\mathbf{c}, \mathbf{c}') \leqslant e.$$
In particular, if we sent $\mathbf{c}$ and received $\mathbf{c}'$, then less than e errors have occurred. By (ED)

above, in this case we *cannot* detect an error; contradiction. So $\mathbf{c}'$ does not exist, and $(*)$ holds. ☐

Example. Consider the 2-error-detecting code $C = \{000, 111\}$. Then

$$\begin{aligned}
B(000,0) &= \{\mathbf{000}\},\\
B(000,1) &= \{\mathbf{000}, 100, 010, 001\},\\
B(000,2) &= \{\mathbf{000}, 100, 010, 001, 110, 101, 011\},\\
B(000,3) &= \{\mathbf{000}, 100, 010, 001, 110, 101, 011, \mathbf{111}\} = V_3.
\end{aligned}$$

The codewords (elements of C) have been written in bold. Note that $B(000,2)$ contains only one codeword, whilst $B(000,3)$ contains two.

What about error correction?

Lemma 3.9. *A code $C \subset V_n$ is e-error-correcting if and only if*

$(**)$ all the Hamming balls $B(\mathbf{c}, e)$ around $\mathbf{c} \in C$ are disjoint.

Proof. Suppose $(**)$ holds. Suppose $\mathbf{c} \in C$ is sent, and received as $\mathbf{y}$. If there are $\leqslant e$ errors, then $\mathbf{y} \in B(\mathbf{c}, e)$. The assumption $(**)$ ensures that for every other $\mathbf{c}' \in C$, we have

$$\mathbf{y} \notin B(\mathbf{c}', e) \quad \Rightarrow \quad d(\mathbf{y}, \mathbf{c}') > e.$$

In particular,

$$d(\mathbf{c}, \mathbf{y}) \leqslant e < d(\mathbf{c}', \mathbf{y}) \qquad \forall \mathbf{c}' \in C \backslash \{\mathbf{c}\},$$

that is, $\mathbf{c}$ is the closest codeword to $\mathbf{y}$. By (EC) above, we can correct such errors.

Conversely, suppose C is e-error-correcting, but that $(**)$ fails; so there exist $\mathbf{c}, \mathbf{c}' \in C$ and

$$\mathbf{x} \in B(\mathbf{c}, e) \cap B(\mathbf{c}', e).$$

Without loss of generality, suppose $\mathbf{x}$ is closest to $\mathbf{c}'$ (i.e. $d(\mathbf{x}, \mathbf{c}') \leqslant d(\mathbf{x}, \mathbf{c})$). If $\mathbf{c}$ is sent, and received as $\mathbf{x}$, then $\leqslant e$ errors have occurred; but $\mathbf{x}$ would be *incorrectly* corrected to $\mathbf{c}'$. This is a contradiction, so $(**)$ holds. ☐

Example. Again consider the code $C = \{000, 111\}$. We check:

$$\begin{aligned}
B(000,1) \cap B(111,1) &= \varnothing,\\
B(000,2) \cap B(111,2) &= \{001, 010, 100, 110, 101, 011\}.
\end{aligned}$$

This tallies with our knowledge that C is 1-error-correcting.

The difference between error-correcting and error-detecting is illustrated in Figure 3.1. The dots in the centre of the circles represent the code words in V_n, which is drawn as the large black circle. The smaller, coloured circles represent Hamming balls centred at the code-words.

In the first image, suppose the balls have radius 2. Each ball contains only one codeword, so the code is 2-error-detecting. However the balls are not distinct, so the code is not 2-error-correcting.

In the second, suppose the balls have radius 1. All the balls are disjoint, so the code is 1-error-correcting.

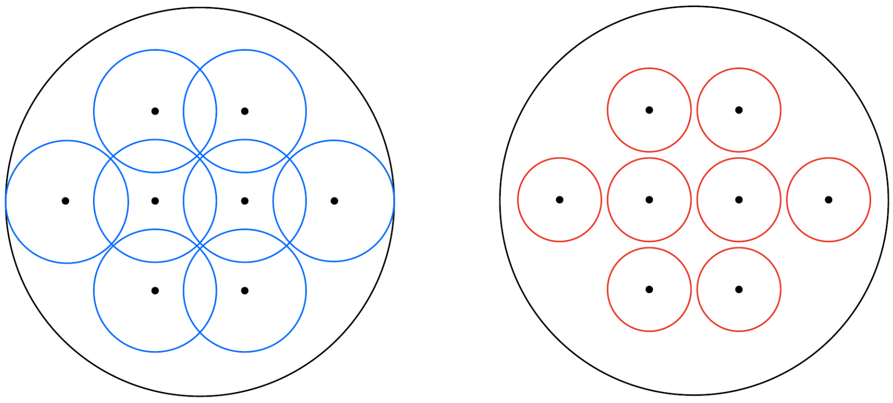


Figure 3.1: Error detecting v/s correcting

3.4. Minimum distance of a code

We now introduce an extremely important invariant of a code: its minimum distance. Henceforth we'll assume that C has at least 2 elements[1].

Definition 3.10. Let C be a binary code. The *minimum distance* of C denoted $d(C)$, is

$$d(C) := \min\left\{d(\mathbf{c}, \mathbf{c}') \,\middle|\, \mathbf{c} \neq \mathbf{c}' \in C\right\}.$$

Examples.

- For $C = \{000, 111\}$, the minimum distance is $d(C) = 3 = d(000, 111)$. It is 2-error-detecting and 1-error-correcting.
- For the paper tape code
$$C = \{\mathbf{x} \in V_8 : \text{even number of 1s}\},$$
we have $d(C) = 2$ (since if you take a codeword and change 1 digit, you must change at least 2 to get another codeword, or there will be an odd number of 1s). This code is 1-error-detecting and 0-error-correcting.
- Recall the 'catastrophic' code of Example 2.3,
$$C = \{000, 001, 101, 111\}.$$
The minimum distance is $d(C) = 1$, realised as e.g. $d(000, 001)$ (or $d(001, 101)$, or $d(101, 111)$). We showed this code is 0-error-correcting and 0-error-detecting.
- Let $m \geqslant 1$. For the m times repetition code $\{0 \cdots 0, 1 \cdots 1\}$, we have $d(C) = m$. It was $m - 1$ error-detecting and $\lfloor \frac{m-1}{2} \rfloor$-error-correcting.

[1]The minimum distance is only well-defined when $\#C \geqslant 2$. The case where C is a singleton set is, in any case, uninteresting/useless in practical terms.

Theorem 3.11. *Let C be a code with minimum distance d. Then using nearest-neighbour decoding:*

(i) C is exactly $(d-1)$-error-detecting;

(ii) C is exactly $\lfloor \frac{d-1}{2} \rfloor$-error-correcting.

In other words, C cannot always detect or correct arbitrary errors longer than this.

Proof. (i) As the minimum distance is d, each Hamming ball $B(\mathbf{c}, d-1)$ around a codeword contains no other codewords; so C is $(d-1)$-error-detecting by Lemma 3.8.

The same lemma shows C is not d-error-detecting, since if $d(\mathbf{c}, \mathbf{c}') = d$ realises the minimum distance, we have $\mathbf{c}' \in B(\mathbf{c}, d)$.

(ii) The Hamming balls $B(\mathbf{c}, \lfloor \frac{d-1}{2} \rfloor)$ for $\mathbf{c} \in C$ are all distinct, so C is $\lfloor \frac{d-1}{2} \rfloor$-error-correcting by Lemma 3.9. (To see this: suppose there exists

$$\mathbf{x} \in B\left(\mathbf{c}, \lfloor \tfrac{d-1}{2} \rfloor\right) \cap B\left(\mathbf{c}', \lfloor \tfrac{d-1}{2} \rfloor\right).$$

Then $d(\mathbf{c}, \mathbf{c}') \leqslant d(\mathbf{c}, \mathbf{x}) + d(\mathbf{x}, \mathbf{c}') \leqslant d - 1$, contradiction).

To see C cannot do any better than this, note that if $d(\mathbf{c}, \mathbf{c}') = d$ realises the minimum distance, then

$$B\left(\mathbf{c}, \lfloor \tfrac{d-1}{2} \rfloor + 1\right) \cap B\left(\mathbf{c}', \lfloor \tfrac{d-1}{2} \rfloor + 1\right) \neq \varnothing. \tag{3.1}$$

Hence C is not $(\lfloor \frac{d-1}{2} \rfloor + 1)$-error-correcting by Lemma 3.9.

(To see (3.1): $\mathbf{c}$ and $\mathbf{c}'$ differ at exactly d places. Define a sequence

$$\mathbf{c} = \mathbf{x}_0, \mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_{d-1}, \mathbf{x}_d = \mathbf{c}',$$

by changing one bit at a time, so that $d(\mathbf{c}, \mathbf{x}_i) = i$ and $d(\mathbf{x}_i, \mathbf{c}') = d - i$. There are two cases:

- d is even, so $\lfloor \frac{d-1}{2} \rfloor + 1 = d/2$, and $\mathbf{x}_{d/2}$ lies in both Hamming balls;
- d is odd, so $\lfloor \frac{d-1}{2} \rfloor + 1 = (d+1)/2$, and $\mathbf{x}_{(d-1)/2}$ and $\mathbf{x}_{(d+1)/2}$ lie in both Hamming balls.) $\square$

Remark. The following example illustrates the final argument in this proof. Consider the $5\times$ repetition code $C = \{\mathbf{0}, \mathbf{1}\}$, where $\mathbf{0} = 00000$ and $\mathbf{1} = 11111$. Then $d = 5$, and the Theorem says the code is exactly 2-error-correcting.

We have $d(\mathbf{0}, \mathbf{1}) = d = 5$, and a sequence between them is

$$\mathbf{x}_0 = 00000,\ \mathbf{x}_1 = 10000,\ \mathbf{x}_2 = 11000,\ \mathbf{x}_3 = 11100,\ \mathbf{x}_4 = 11110,\ \mathbf{x}_5 = 11111.$$

Then

$$\{\mathbf{x}_2, \mathbf{x}_3\} \subset B(\mathbf{0}, 3) \cap B(\mathbf{1}, 3) \neq \varnothing.$$

By Lemma 3.9 the code cannot be 3-error-correcting.

Corollary 3.12. *Using nearest-neighbour decoding:*

(i) If C is exactly e-error-detecting, then its minimum distance is exactly $e+1$.

(ii) If C is exactly e-error-correcting, then its minimum distance is either $2e+1$ or $2e+2$.

3.5. (n, k, d)-codes

We've seen that the minimum distance completely describes the error-detecting and error-correcting properties of a code. In particular, it's a useful classifier of a code.

Definition 3.13. If C is a code with:

- $C \subset V_n$,
- $\#C = 2^k$,
- and minimum distance d,

we call it an (n, k, d)*-code*.

Examples.

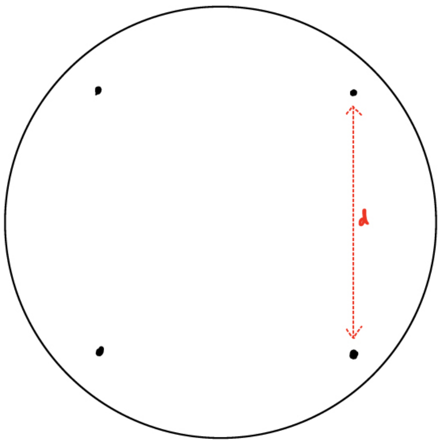
- The $3\times$ repetition code $000, 111$ is a $(3, 1, 3)$-code.
- More generally, the m-times repetition code (see Example 2.4) is an $(m, 1, m)$-code.

A fundamental question is: for arbitrary integers n, k and d, does there exist an (n, k, d)-code? Not always! For starters, we already specified that $k \leqslant n$. Further:

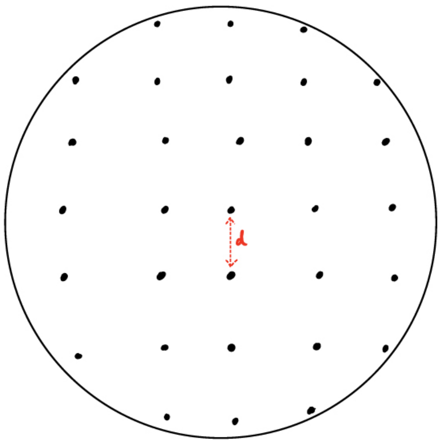
Example.

- Can there exist a $(3, 1, 4)$-code? *No:* the distance between two elements of V_3 is at most 3, so there can never be two elements of $C \subset V_3$ with distance 4.
- More generally, the same argument shows that if an (n, k, d)-code exists, then $d \leqslant n$.
- Thus $k \leqslant n$ and $d \leqslant n$. Can they both be equal to n? A little thought shows *no*. If k gets large, then C has more elements, and they are more 'tightly packed' into V_n – there is less room to 'spread them out'. This will mean that the minimum distance gets smaller.

The picture is the following.



Here k is small, so $\#C$ is small, and the elements of C can be spread out more, meaning d can be large.



Here k is large, so there is less room to spread the elements out, so d must be small.

More precisely, this relationship is described by the *Singleton bound.*

Theorem 3.14 (The Singleton[a] bound)**.** *Let C be an (n, k, d)-code. Then*

$$k + d \leqslant n + 1.$$

[a]The Singleton bound is named after the mathematician R. C. Singleton, *not* the singleton set!

Proof. Define a map

$$\begin{aligned} f : V_n &\longrightarrow V_{n-d+1} \\ (v_1, \dots, v_{n-d+1}, v_{n-d+2}, \dots v_n) &\longmapsto (v_1, \dots, v_{n-d+1}) \end{aligned}$$

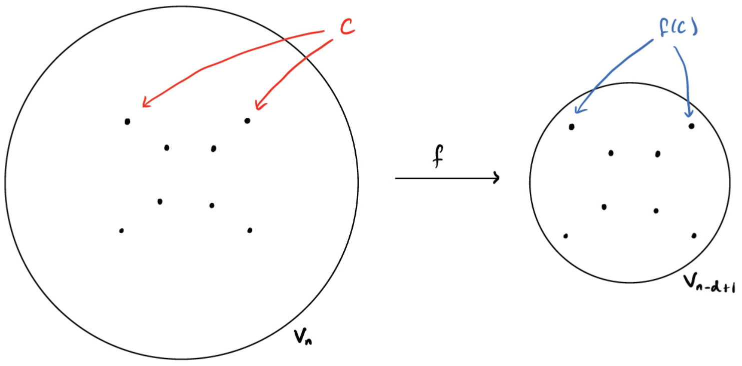
by cutting off the last $d - 1$ bits.

I claim that if $\mathbf{c} \neq \mathbf{c}' \in C$, then

$$f(\mathbf{c}) \neq f(\mathbf{c}').$$

To see this, note that if $f(\mathbf{c}) = f(\mathbf{c}')$, then $\mathbf{c}$ and $\mathbf{c}'$ differ only in the last $d - 1$ digits, so $d(\mathbf{c}, \mathbf{c}') \leqslant d - 1$ – but we know $d(\mathbf{c}, \mathbf{c}') \geqslant d$, as the minimum distance of C is d.

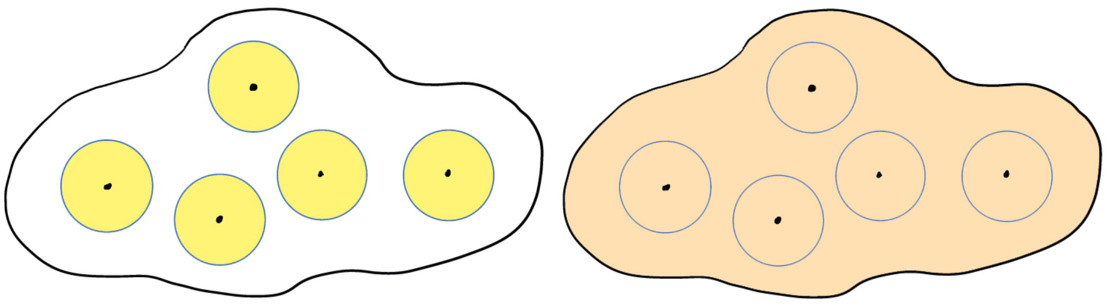
The above means that f is injective. Since any function is surjective onto its image $f(C)$, the function f thus induces a bijetion between C and $f(C)$. This means that C and $f(C)$ have the same size, namely 2^k. The picture is the following:



Thus $f(C) \subset V_{n-d+1}$ is a subset of size 2^k, so $2^k \leqslant 2^{n-d+1}$, whence $k \leqslant n-d+1$. Adding d gives $k+d \leqslant n+1$. □

We can obtain a different bound by recalling Lemma 3.9, that

$$C \text{ is } e\text{-error-correcting} \iff \text{all } B(\mathbf{c}, e) \text{ around } \mathbf{c} \in C \text{ are disjoint.}$$



From the picture, we see that

$$\begin{aligned}(\text{size of yellow area}) &= \#C \cdot \#B(\mathbf{c}, e) \\ &\leqslant \#V_n = (\text{size of orange area}).\end{aligned} \tag{3.2}$$

We know that $\#C = 2^k$, and $\#V_n = 2^n$. So what is $\#B(\mathbf{c}, e)$?

Fact 3.15. For a fixed radius r, the Hamming balls

$$B(\mathbf{x}, r) = \{\mathbf{y} \in V_n : d(\mathbf{x}, \mathbf{y}) \leqslant r\}$$

all have the same number of elements independent of the centre $\mathbf{x}$. More precisely, we have

$$\#B(\mathbf{x}, r) = \sum_{j=0}^{r} \binom{n}{j} = \binom{n}{0} + \cdots + \binom{n}{r}.$$

Why it this? Well, note we have

$$\begin{aligned}B(x, r) &= \{y \in V_n : d(x, y) \leqslant r\} \\ &= \{y \in V_n : d(x, y) = 0\} \cup \{y \in V_n : d(x, y) = 1\} \cup \cdots \cup \{y \in V_n : d(x, y) = r\}.\end{aligned}$$

Now observe

$$\binom{\{y \in V_n : d(x,y) = 0\} = \{y\} \text{ has size } 1 = n}{0.}$$

How big is $\{y \in V_n : d(x,y) = 1\}$? This is the set of all words in V_n that differ from x in exactly one place. There are $\binom{n=n}{1}$ possible places for this.

Similarly $\{y \in V_n : d(x,y) = 2\}$ is the set of all words that differ from x in exactly two places. There are n possible places that could be changed, so there are $\binom{n}{2}$ possible pairs of places that can be changed, so

$$\#\{y \in V_n : d(x,y) = 2\} = \binom{n}{2}.$$

Continuing similarly, we see that

$$\#\{y \in V_n : d(x,y) = r\} = \binom{n}{r}.$$

Combining all of the above gives the fact.

A more precise version of (3.2) is thus:

Theorem 3.16 (Sphere-packing bound/Hamming bound)**.** *Let C be an (n,k,d)-code, and let $e = \lfloor \frac{d-1}{2} \rfloor$ (so C is e-error-correcting by Theorem 3.11). Then*

$$\sum_{j=0}^{e} \binom{n}{j} \leqslant 2^{n-k}. \tag{3.3}$$

In particular, we have

$$\#C \leqslant \frac{2^n}{\sum_{j=0}^{e} \binom{n}{j}}.$$

Note this says that for fixed n, as e increases, to be e-error-correcting the code must get smaller (i.e. k must get smaller). For $n = 7$, we see that a 0-correcting code has $k \leqslant 7$, a 1-correcting code has $k \leqslant 4$, a 2-correcting code has $k \leqslant 2$, and a 3-correcting code has $k \leqslant 1$. (Exercise: write down a 3-correcting code with $n = 7$ and $k = 1$).

Proof. There are 2^k codewords. Since C is e-error-correcting, by Lemma 3.9 all of the 2^k Hamming balls $B(\mathbf{c}, e)$ around $\mathbf{c} \in C$ are disjoint; and by Fact 3.15, they all have the same size. Thus

$$2^k \cdot \#B(\mathbf{c}, e) \leqslant \#V_n = 2^n.$$

Dividing through by 2^k, we see that

$$\#B(\mathbf{c}, e) = \sum_{j=0}^{e} \binom{n}{j} \leqslant 2^{n-k},$$

as required. □

Exercise. Alice wants to send messages of length $k = 4$, and she wants to send them using a 1-error-correcting code. Explain why she must use a code with block length $n \geqslant 7$.

This shows that Hamming's code with $k = 4$, $n = 7$ – which we introduced in §1.3, but haven't explained properly yet – is the most efficient possible 1-error-correcting code with $k = 4$.

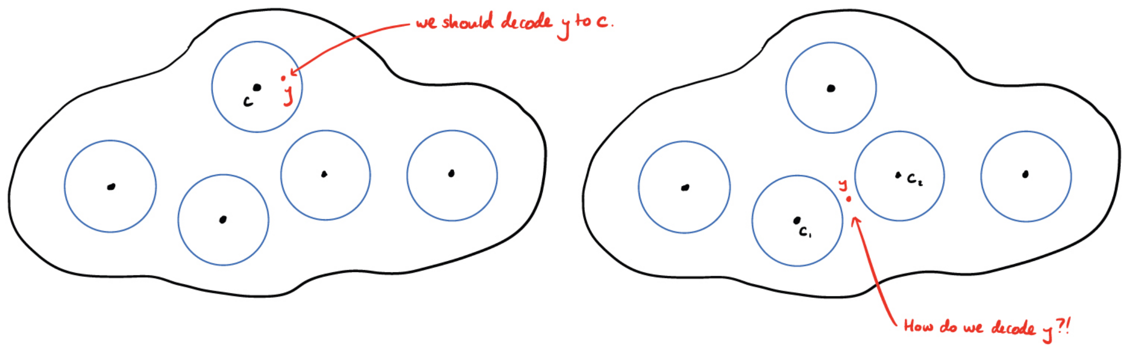
3.6. Perfect codes

Recall that a code C is e-error-correcting if and only if all the Hamming balls $B(\mathbf{c}, e)$ around codewords are disjoint. Now suppose Alice sends a message, and Bob receives $\mathbf{y}$.

(1) If $\mathbf{y}$ lies in one of the Hamming balls $B(\mathbf{c}, e)$, then Bob knows to decode it to $\mathbf{c}$.

(2) ...but what if $\mathbf{y}$ is *not* in one of these balls? It might be equally close to several $\mathbf{c}$'s, and there is ambiguity in decoding.

Possibilities (1) and (2) are sketched (on the left and right respectively) in the pictures below:



It's desirable to find codes where the second possibility (2) *never* occurs.

Definition 3.17. An e-error-correcting code is called *perfect* if

$$V_n = \bigcup_{\mathbf{c} \in C} B(\mathbf{c}, e).$$

We could also state this as follows: *a code C is called perfect if there exists an integer e such that*

$$V_n = \bigsqcup_{\mathbf{c} \in C} B(\mathbf{c}, e),$$

that is, that every $\mathbf{x} \in V_n$ is distance $\leqslant e$ from one, and only one, $\mathbf{c} \in C$.

If this second, equivalent, definition holds, then the Hamming balls $B(\mathbf{c}, e)$ are all disjoint, so C is e-error-correcting.

Lemma 3.18. *If C is a perfect e-error-correcting code, then $d(C) = 2e + 1$ is odd.*

Proof. As C is e-error-correcting, we know $d(C) = 2e + 1$ or $2e + 2$ (by Corollary 3.12)

Suppose $d(C)$ is even. Then $d(C) = 2e + 2$. If $\mathbf{c}$ and $\mathbf{c}'$ are two codewords with $d(\mathbf{c}, \mathbf{c}') = 2e + 2$, then there exists $\mathbf{x} \in V_n$ with

$$d(\mathbf{c}, \mathbf{x}) = e + 1 = d(\mathbf{x}, \mathbf{c}').$$

(Since: $\mathbf{c}$ and $\mathbf{c}'$ differ in $2e+2$ places. As in the proof of Theorem 3.11, we construct such an $\mathbf{x}$ by starting from $\mathbf{c}$ and swapping only $e+1$ of the places).

Now $\mathbf{x}$ does not lie in $B(\mathbf{c}, e)$ or $B(\mathbf{c}', e)$. As C is perfect, it must lie in $B(\mathbf{c}'', e)$ for some $\mathbf{c}'' \in C$. But then by the triangle inequality

$$\begin{aligned} d(\mathbf{c}, \mathbf{c}'') &\leqslant d(\mathbf{c}, \mathbf{x}) + d(\mathbf{x}, \mathbf{c}'') \\ &= (e+1) + e \\ &= 2e+1, \end{aligned}$$

which is a contradiction to the minimum distance being $2e+2$. Therefore $d(C)$ must have been odd all along. $\square$

Lemma 3.19. *If C is an $(n, k, 2e+1)$-code, then C is perfect if and only if the sphere-packing inequality* (3.3) *is an equality, i.e.*

$$\sum_{j=0}^{e} \binom{n}{j} = 2^{n-k}. \tag{3.4}$$

Proof. Suppose C is perfect, i.e. there exists an integer e such that

$$V_n = \bigsqcup_{\mathbf{c} \in C} B(\mathbf{c}, e).$$

Then C is e-error-correcting. Counting both sides, we get

$$\begin{aligned} 2^n &= \#C \times \#B(\mathbf{c}, e) \\ &= 2^k \cdot \sum_{i=0}^{e} \binom{n}{i}, \end{aligned}$$

where we count the size of the Hamming ball using Fact 3.15. Dividing by 2^k gives the required equality in the inequality (3.3).

Conversely, suppose that (3.3) is an equality. We know already that if C is e-error-correcting, then

$$\bigsqcup_{\mathbf{c} \in C} B(\mathbf{c}, e) \subset V_n$$

(this is how we actually proved the Hamming bound). But both sides have the same size, namely 2^n; so this inclusion is an equality, and hence C is perfect. $\square$

Examples.

- The *trivial code* $C = V_n$ is perfect. It is an $(n, n, 1)$-code.
- Recall the k-times-binary repetition code $C = \{0 \cdots 0, 1 \cdots 1\}$ is a $(k, 1, k)$-code. If $k = 2e+1$ is odd, then one can check that (3.4) holds, and C is perfect. (Exercise: prove this!)

  If k is even, then $d(C) = k$ is not odd, so C cannot be perfect.

3.7. Summary so far

Remember the guiding question we considered in the introduction. In a code:

*One must **maximise** the chance that the correct message is received...*
*...whilst **minimising** the amount of data that needs to be sent.*

We can now describe this more mathematically. We want to find an (n,k,d)-code C with parameters such that:

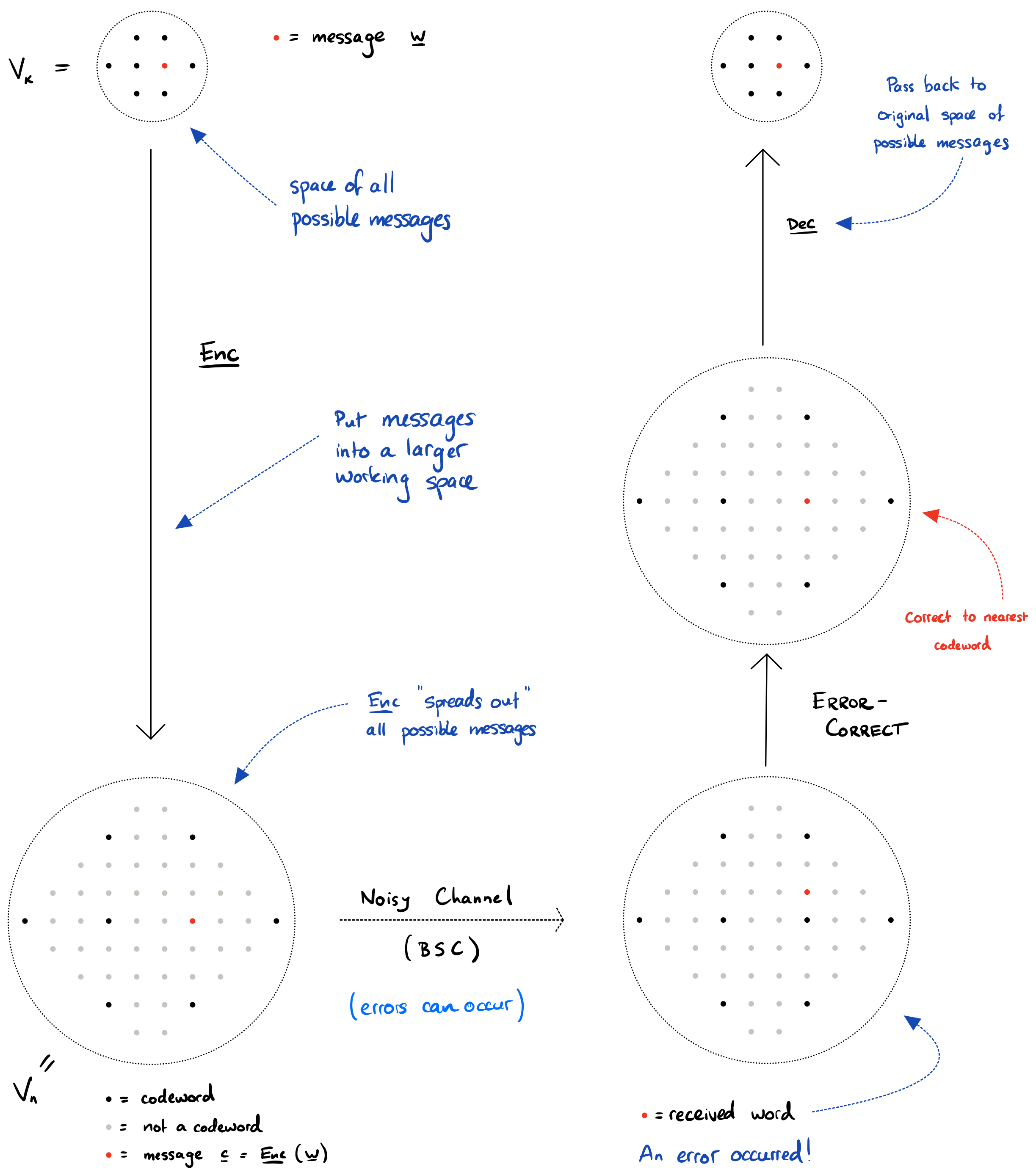
- **Maximising**: C detects/corrects the most possible errors. By Theorem 3.11, this means we want its minimum distance d to be as large as possible.
- **Unique correction:** We don't want ambiguity in decoding schemes. Thus we want C to be perfect, i.e. $d = 2e+1$ should be odd, and n, k and e should satisfy the equality (3.4).
- **Minimising:** We want n to be 'not much larger' than k. In other words, we want the *rate* k/n to be as large as possible.

For real-world applications, we can also throw in:

- **Practical:** It should be efficient to put the code into a computer; in other words, the encoding and decoding maps **Enc** and **Dec** should be easy to write down and compute.

The art of finding good codes is to keep all these considerations in balance; but for different applications, certain aspects are more important than others. One might want to optimise speed over error correction, in which case one must prioritise keeping the rate low at the cost of d remaining small. Alternatively, one might want to optimise error correction, at the cost of additional time; so one allows a smaller rate to increase d.

In the next chapters, we'll actually write down such codes using linear algebra.



V_k =
• = message $\underline{w}$
space of all possible messages
Enc
Put messages into a larger working space
Enc "spreads out" all possible messages
V_n =
• = codeword
• = not a codeword
• = message $\underline{c}$ = Enc ($\underline{w}$)
Noisy Channel
(BSC)
(errors can occur)
• = received word
An error occurred!
Error-Correct
Correct to nearest codeword
Dec
Pass back to original space of possible messages

Chapter 4

Linear codes

So far, we've studied mathematical properties of abstract codes, but the only real examples we've seen have been of repetition codes. We now introduce a class of codes – the *linear codes* – that give many more examples, including Hamming's code.

The theory of linear codes rests on the observation:

> *We may view the set of binary numbers $V_n = \{0,1\}^n$ as a vector space over the field with two elements!*

A *linear code* is a vector subspace $C \subset V_n$. In practice, *all* codes used in the wild are linear codes, because we can then use linear algebra, meaning:

- encoding and decoding are easy (using matrices);
- error-correction may be easier (when $n - k$ is small);
- it is easier to calculate minimum distances.

In particular, during this chapter we'll see that *encoding* is done via a generator matrix, *decoding* via the standard form, and *error correction* via syndromes, using a *parity-check matrix.* The minimum distance can be easily read off from this parity-check matrix. Thus:

$$\begin{tikzcd}
\boxed{\text{sourceword } \mathbf{w} \in V_k} \arrow[dd, "\textbf{Enc}", "\text{(generator matrix)}"'] & & \boxed{\text{sourceword } \mathbf{w}' \in V_k} \\
& & \boxed{\text{codeword } \mathbf{c}' \in V_k} \arrow[u, "\text{(standard form)}", "\textbf{Dec}"'] \\
\boxed{\text{codeword } \mathbf{c} \in V_k} \arrow[rr, dashed, "\text{Noisy channel}"] & & \boxed{\text{received word } \mathbf{x} \in V_k} \arrow[u, "\text{(syndromes/ parity-check matrix)}", "\text{error-correct}"']
\end{tikzcd}$$

4.1. Linear algebra review

The theory of linear codes depends foundationally on basic linear algebra (of the kind you met in core first year courses, or in some cases even at school). We quickly recap the most vital results and notions.

4.1.1. Fields and vector spaces.

Definition. A *field* is a set K where you can do:

- addition,
- subtraction,
- multiplication,
- and division of everything except 0

(with all the usual rules, e.g. associativity $(a+b)+c = a+(b+c)$, commutativity $a \times b = b \times a$, etc).

For example, the usual fields you meet are the field $\mathbf{Q}$ of rational numbers, $\mathbf{R}$ of real numbers, and $\mathbf{C}$ of complex numbers. (We'll be interested in the field $\mathbf{F}_2 = \{0, 1\}$, which we'll introduce below).

Definition. Given a field K, and $n \geqslant 0$, we have the n-dimensional *vector space*

$$V = K^n = \{(x_1, ..., x_n) \in K^n\}$$

of all n-tuples in K. We can do addition as usual:

$$(x_1, ..., x_n) + (y_1, ..., y_n) = (x_1 + y_1, ..., x_n + y_n).$$

4.1.2. Bases of vector spaces.

Definition. A *basis* of a vector space V is a set $\{\mathbf{e}_1, ..., \mathbf{e}_n\}$ such that every element $\mathbf{x} \in V$ can be written uniquely as

$$\mathbf{x} = \lambda_1\mathbf{e}_1 + \cdots + \lambda_n\mathbf{e}_n, \qquad \lambda_i \in K.$$

The 'standard' basis is $\mathbf{e}_1 = (1, 0, ..., 0), \mathbf{e}_2 = (0, 1, 0, ..., 0), ..., \mathbf{e}_n = (0, ..., 0, 1)$, but there are many choices: for example, $\{(0, 1), (1, 1)\}$ is another basis of K^2.

If the definition of basis does not bring back memories (or at least does not bring back *fond* memories), then be reassured we will revisit it later in these notes, at the end of §4.3.

4.1.3. Matrices, Images, and Kernels.

Definition.
- We write $M_{k\times n}(K)$ for the space of $k \times n$ matrices with entries in K (i.e. k rows and n columns), e.g.

$$\begin{pmatrix} 1 & 2 & 6 \\ 4 & 1 & 10 \end{pmatrix} \in M_{2\times 3}(\mathbf{Q}).$$

- If $A \in M_{k\times n}(K)$, write $A^t \in M_{n\times k}(K)$ for its transpose, i.e. its 'reflection in the diagonal'. For example

$$\begin{pmatrix}1 & 2 & 6\\ 4 & 1 & 10\end{pmatrix}^t = \begin{pmatrix}1 & 4\\ 2 & 1\\ 6 & 10\end{pmatrix}.$$

We may view $\mathbf{x} \in K^n$ as a $1 \times n$ matrix $(x_1 \ \cdots \ x_n)$. Its transpose is an $n \times 1$ matrix (a column vector)

$$\mathbf{x}^t = \begin{pmatrix}x_1\\ \vdots\\ x_n\end{pmatrix}.$$

We recall the image and kernel of a matrix.

Definition. Recall that if

$$A = \begin{pmatrix}a_{11} & \cdots & a_{1n}\\ \vdots & \ddots & \vdots\\ a_{k1} & \cdots & a_{kn}\end{pmatrix} \in M_{k\times n}(K)$$

is a matrix with k rows and n columns, then matrix multiplication on the left defines a map

$$\begin{aligned} K^n &\longrightarrow K^k\\ \mathbf{v} = (v_1, \dots, v_n) &\longmapsto A\mathbf{v} = \begin{pmatrix}a_{11} & \cdots & a_{1n}\\ \vdots & \ddots & \vdots\\ a_{k1} & \cdots & a_{kn}\end{pmatrix}\begin{pmatrix}v_1\\ \vdots\\ v_n\end{pmatrix}\\ &= (a_{11}v_1 + \cdots a_{1n}v_n, \dots, a_{k1}v_1 + \cdots a_{kn}v_n). \end{aligned}$$

Definition. The *image* is the set of all possible images of the above map, i.e. the set of all $\mathbf{x} \in K^k$ such that $\mathbf{x} = A\mathbf{v}$ for some $\mathbf{v} \in K^n$. In other words, it is the set

$$\mathrm{Im}(A) := \{A\mathbf{v} \in K^k : \mathbf{v} \in K^n\} \subset K^k.$$

The image is a vector subspace of K^k (it is closed under addition and scalar multiplication, and contains $\mathbf{0}$).

Another way of considering this definition is:

Lemma. *The image* $\mathrm{Im}(A)$ *of a matrix* A *is the set of all linear combinations of the columns of* A*.*

More precisely, if we write

$$A = \begin{pmatrix}a_{11} & \cdots & a_{1n}\\ \vdots & \ddots & \vdots\\ a_{k1} & \cdots & a_{kn}\end{pmatrix} \in M_{k\times n}(K),$$

then the image is the subset of all vectors in K^k that can be written in the form

$$\mathrm{Im}(A) = \left\{ v_1 \begin{pmatrix} a_{11} \\ \cdots \\ a_{k1} \end{pmatrix} + \cdots + v_n \begin{pmatrix} a_{1n} \\ \cdots \\ a_{kn} \end{pmatrix} : v_1, ..., v_n \in K \right\}.$$

We normally write this as the *span* (over K) of the columns of A, written

$$\mathrm{Im}(A) = \mathrm{Span}_K(\text{columns of } A) \subset K^k.$$

Example. Let

$$A = \begin{pmatrix} 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

We compute that for any $\mathbf{v} = (v_1, ..., v_4) \in K^4$, we have

$$\begin{aligned} A\mathbf{v} &= \begin{pmatrix} 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} v_1 \\ v_2 \\ v_3 \\ v_4 \end{pmatrix} \\ &= (v_1 + v_3 + v_4, \quad v_2 + v_3 + v_4, \quad v_4), \end{aligned}$$

so

$$\mathrm{Im}(A) = \{(v_1 + v_3 + v_4, \quad v_2 + v_3 + v_4, \quad v_4) : v_i \in K\}.$$

As mentioned above we can write this as the span of the columns:

$$\mathrm{Im}(A) = \left\{ v_1 \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} + v_2 \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} + v_3 \begin{pmatrix} 1 \\ 1 \\ 0 \end{pmatrix} + v_4 \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} : v_i \in K \right\}.$$

Note here that

$$\begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} + \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} = C1 + C2 = C3 = \begin{pmatrix} 1 \\ 1 \\ 0 \end{pmatrix},$$

so we can simplify this: we have

$$\begin{aligned} v_1 \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} + v_2 \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} + v_3 \begin{pmatrix} 1 \\ 1 \\ 0 \end{pmatrix} + v_4 \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} &= (v_1 + v_3) \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} + (v_2 + v_3) \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} + v_4 \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} \\ &= v_1' \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} + v_2' \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} + v_3' \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}, \end{aligned}$$

where $v_1' = v_1 + v_3, v_2' = v_2 + v_3$, and $v_3' = v_4$. You can't do this any further, as $\{(1, 0, 0), (0, 1, 0), (1, 1, 1)\}$ is a basis of K^3, so $\mathrm{Im}(A)$ is a 3-dimensional vector space. Indeed, we see that $\mathrm{Im}(A) = K^3$ is everything (so A is surjective).

Definition. The *kernel* of a matrix A is

$$\mathrm{Ker}(A) := \{\mathbf{x} \in K^n : A\mathbf{x} = 0\}.$$

It is a vector subspace of K^n.

Example. Again let's consider

$$A = \begin{pmatrix} 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

An element of the kernel is $(1, 1, -1, 0)$, because

$$\begin{aligned} \begin{pmatrix} 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 \\ 1 \\ -1 \\ 0 \end{pmatrix} &= (1-1+0, \quad 1-1+0, \quad 0) \\ &= (0, 0, 0). \end{aligned}$$

In general, the elements of the kernel are exactly those $\mathbf{v} = (v_1, \dots, v_n)$ such that

$$A\mathbf{v} = (v_1 + v_3 + v_4, \quad v_2 + v_3 + v_4, \quad v_4) = (0, 0, 0).$$

This forces $v_4 = 0$. Moreover if $v_1 + v_3 + v_4 = v_1 + v_3 = 0$, and $v_2 + v_3 + v_4 = v_2 + v3 = 0$, then we see $v_1 = v_2 = -v_3$. So the most general element of the kernel has the form

$$(v, v, -v, 0) \in K^4,$$

that is, $\mathrm{Ker}(A)$ is the 1-dimensional subspace $\{(v, v, -v, 0) : v \in K\}$ of all scalar multiples of the vector above.

Note that in this example, $\mathrm{Im}(A)$ was 3-dimensional, whilst $\mathrm{Ker}(A)$ is 1-dimensional, and $3+1 = 4$ is the number of columns (i.e. n) for this matrix. The *Rank–Nullity theorem* says this is true in general.

Theorem (Rank–Nullity). *If $A \in M_{k\times n}(K)$ is a matrix with k rows and n columns, then the dimensions of the image and kernel sum to n, i.e.*

$$\dim_K[\mathrm{Im}(A)] + \dim_K[\mathrm{Ker}(A)] = n.$$

Remark. Note that *right*-multiplication by A also defines a map on vector spaces, but in the other way:

$$\begin{aligned} K^k &\longrightarrow K^n \\ (w_1, \dots, w_k) &\longmapsto (w_1, ..., w_k) \begin{pmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{k1} & \cdots & a_{kn} \end{pmatrix} \\ &= (a_{11}w_1 + \cdots + a_{k1}w_k, \dots, a_{1n}w_1 + \cdots a_{kn}w_k). \end{aligned}$$

For example,

$$(1,2)\begin{pmatrix}1 & 2 & 6\\ 4 & 1 & 10\end{pmatrix} = (9,4,26).$$

We will use this fact later to write down encoding maps for codes!

4.2. The field with two elements

For us, we are interested in a very particular field K: the *field* $\mathbf{F}_2$ *with two elements.* This is directly relevant to the study of binary numbers.

Definition 4.1 (The field $\mathbf{F}_2$ of 2 elements)**.** We can equip the set $\{0,1\}$ with addition and multiplication modulo 2, i.e.

Binary operations:

$\times$	0	1
0	0	0
1	0	1

$+$	0	1
0	0	1
1	1	0

With these operations, $\{0,1\}$ becomes a field, denoted $\mathbf{F}_2$ – the *field with two elements.*

Proposition 4.2. *We can see the space V_n of binary numbers of length n as an n-dimensional vector space over $\mathbf{F}_2$, via*

$$\begin{aligned} V_n &\longrightarrow \mathbf{F}_2^n \\ x_1\cdots x_n &\longmapsto (x_1,...,x_n) \end{aligned}$$

For example, the binary number 011001 of length 6 corresponds to the vector

$$011001 \mapsto (0,1,1,0,0,1) \in \mathbf{F}_2^6.$$

We can 'transfer' the addition structure on $\mathbf{F}_2^n$ to give a vector space addition structure on the binary numbers V_n, for example

$$1+1=0, \qquad 10+11=01, \qquad 11010+10010=01000,$$

and more generally (recalling all additions are mod 2)

$$\Big(x_1\cdots x_n\Big) + \Big(y_1\cdots y_n\Big) = (x_1+y_1)\ (x_2+y_2)\cdots(x_n+y_n).$$

Note that if $\mathbf{x}$ and $\mathbf{y}$ are binary numbers, then $\mathbf{x}+\mathbf{y} = \mathbf{x}-\mathbf{y}$, so addition and subtraction are the same!

Exercise. Let $\mathbf{x}$ be a binary number. What is $\mathbf{x}+\mathbf{x}$?

4.3. First properties of linear codes

Definition 4.3 (Linear code)**.**
- A *linear code* of block length n and message length k is a k-dimensional linear subspace of V_n. In other words, C is a code such that:
  - C has size 2^k,
  - and if $\mathbf{c}$ and $\mathbf{c}'$ are elements of C, then $\mathbf{c} + \mathbf{c}' \in C$.
- If an (n, k, d)-code is linear, we call it an $[n, k, d]$-code.

Example.
- Repetition codes are also linear codes! (Don't worry, we're going to move away from them eventually). For example, consider 3 times repetition $C = \{000, 111\}$. We see that

$$\begin{aligned} 000 + 000 &= 000 \in C,\\ 000 + 111 &= 111 \in C,\\ 111 + 111 &= 000 \in C. \end{aligned}$$

  So C is closed under addition.
- The paper tape code $C = \{\mathbf{x} \in V_8 :\ \text{sum of the digits is even}\}$ is linear. One can check that if $\mathbf{x}$ and $\mathbf{y}$ have even digit-sums, then $\mathbf{x} + \mathbf{y}$ does too, so $\mathbf{x} + \mathbf{y} \in C$.
- The very bad code $C = \{000, 001, 101, 111\}$ is *not* linear, as for example $001 + 101 = 100 \notin C$.

4.3.1. Bases for linear codes. To define a code of message length k, in general one must write down all 2^k codewords. A linear code is determined by much less information: a linear code C is entirely determined by a *basis* of the vector space C. We recall what basis means in our context.

Definition 4.4. Let C be an $[n, k, d]$-code. A *basis* of C is any set $B \subset C$ such that:
- $\#B = k$,
- If $B = \{\mathbf{e}_1, ..., \mathbf{e}_k\}$, then every $\mathbf{c} \in C$ can be written (uniquely) as a sum of some of the $\mathbf{e}_i$. In other words, we have

$$\begin{aligned} C = \{\mathbf{0}, \mathbf{e}_1,\ \ ...,\ \ \mathbf{e}_k, & \qquad\qquad \text{(all sums of one of the } \mathbf{e}_i)\\ \mathbf{e}_1 + \mathbf{e}_2,\ \ ...,\ \ \mathbf{e}_r + \mathbf{e}_s,\ \ ... & \qquad\qquad \text{(all sums of two of the } \mathbf{e}_i\text{'s)}\\ \mathbf{e}_1 + \mathbf{e}_2 + \mathbf{e}_3,\ \ ...,\ \ \mathbf{e}_r + \mathbf{e}_s + \mathbf{e}_t,\ \ ..., & \qquad\qquad \text{(all sums of three of the } \mathbf{e}_i\text{'s)}\\ ... & \\ \mathbf{e}_1 + \cdots + \mathbf{e}_k\}. & \end{aligned}$$

Example. Let C be the $3\times$-repetition code of message length 2, i.e. $k = 2$, $n = 6$, and $C = \{000000, 000111, 111000, 111111\}$. Then $\{\mathbf{e}_1 = 111000, \mathbf{e}_2 = 000111\}$ is a basis of C, since

$$\begin{aligned} 000000 &= 0 \times 111000 + 0 \times 000111 = [\text{sum of none of the } \mathbf{e}_i\text{'s}],\\ 111000 &= 1 \times 111000 + 0 \times 000111 = \mathbf{e}_1,\\ 000111 &= 0 \times 111000 + 1 \times 000111 = \mathbf{e}_2,\\ 111111 &= 1 \times 111000 + 1 \times 000111 = \mathbf{e}_1 + \mathbf{e}_2, \end{aligned}$$

i.e. every element of C can be written uniquely as a linear combination of $\mathbf{e}_1$ and $\mathbf{e}_2$.

Example. In the paper tape code from §1.2, a basis is given by

$$\begin{aligned}\mathbf{e}_1 &= 11000000,\\ \mathbf{e}_2 &= 00110000,\\ \mathbf{e}_3 &= 00001100,\\ \mathbf{e}_4 &= 00000011,\\ \mathbf{e}_5 &= 10100000,\\ \mathbf{e}_6 &= 10001000,\\ \mathbf{e}_7 &= 10000010.\end{aligned}$$

Note that bases give us a quick way of telling whether a code is linear. We do not want to be checking every possible sum $\mathbf{c}+\mathbf{c}'$ for codewords $\mathbf{c},\mathbf{c}' \in C$ to verify that C is closed under addition, as this involves checking 2^{2k} different possible values of $\mathbf{c}+\mathbf{c}'$. In practice, it is easier to use:

Lemma 4.5. *A code C is linear if and only if it has a basis.*

Proof. If C is linear, it is a vector subspace of V_n; and vector spaces have bases.

If C has a basis $\{\mathbf{e}_1, ..., \mathbf{e}_k\}$ in the sense of the last Definition, then every codeword is a sum of basis vectors, i.e. a sum of $\mathbf{e}_i$'s. Thus the sum of any two codewords is a sum of $\mathbf{e}_i$'s, and hence (by definition of basis) a codeword. □

Example. Consider

$$C = \{00000, 10101, 00010, 10111, 11000, 01101, 11010, 01111\} \subset V_5.$$

Let's show C is linear by finding a basis.

(1) First, let $\mathbf{e}_1$ be any choice of non-zero codeword, e.g.

$$\mathbf{e}_1 = 10101.$$

Then we have

00000	10101	00010	10111	11000	01101	11010	01111
$\mathbf{0}$	$\mathbf{e}_1$						

(2) Now pick $\mathbf{e}_2$ to be any codeword we haven't yet accounted for e.g.

$$\mathbf{e}_2 = 00010.$$

If C is linear, then we must have $\mathbf{e}_1 + \mathbf{e}_2 \in C$, which we do (it's the codeword 10111). Now we have accounted for:

00000	10101	00010	10111	11000	01101	11010	01111
$\mathbf{0}$	$\mathbf{e}_1$	$\mathbf{e}_2$	$\mathbf{e}_1+\mathbf{e}_2$				

(3) Now pick $\mathbf{e}_3$ to be any codeword we haven't accounted for, e.g. $\mathbf{e}_3 = 11000$. Now all we have to check is that adding $\mathbf{e}_3$ to the codewords we already have keeps us in the code:

$$00000 + \mathbf{e}_3 = 11000 \in C, \qquad 10101 + \mathbf{e}_3 = 01101 \in C,$$
$$00010 + \mathbf{e}_3 = 11010 \in C, \qquad 10111 + \mathbf{e}_3 = 01111 \in C.$$

Our code is then:

00000	10101	00010	10111	11000	01101	11010	01111
$\mathbf{0}$	$\mathbf{e}_1$	$\mathbf{e}_2$	$\mathbf{e}_1+\mathbf{e}_2$	$\mathbf{e}_3$	$\mathbf{e}_1+\mathbf{e}_3$	$\mathbf{e}_2+\mathbf{e}_3$	$\mathbf{e}_1+\mathbf{e}_2+\mathbf{e}_3$

So $\{10101, 00010, 11000\}$ is a basis of C, and hence C is linear.

Note that if any of the steps in this example *failed* – that is, if (for example) $10101 + \mathbf{e}_3$ had not been a codeword – then this would be a proof that C is *not* linear.

4.3.2. Writing down linear codes. Bases also give a good way of writing down linear codes. The following inductive algorithm is how I write down examples for your exercise sheets; you can use it to generate your own examples (and exercises!).

- Choose n.

  For example, take $n = 5$.

- Choose any element $\mathbf{e}_1 \in V_n$. This the first basis element. Then
  $$C_1 = \{\mathbf{0}, \mathbf{e}_1\}$$
  is a linear code of message length 1 and basis $\{\mathbf{e}_1\}$.

  For example, we could take $\mathbf{e}_1 = 11011$*, so* $C_1 = \{00000, 11011\}$.

- Choose any element $\mathbf{e}_2 \in V_n$ not already in C_1 (so $\mathbf{e}_2 \neq \mathbf{0}, \mathbf{e}_1$). This is your second basis element. To get C_2, add $\mathbf{e}_2$ to all the elements of C_1 and put them in your set, giving
  $$C_2 = \{\mathbf{0}, \mathbf{e}_1, \quad \mathbf{e}_2, \mathbf{e}_1 + \mathbf{e}_2\}.$$
  This is a linear code of message length 2 and basis $\{\mathbf{e}_1, \mathbf{e}_2\}$.

  In our example, we must take $\mathbf{e}_2 \neq 00000, 11011$. *Let's choose* $\mathbf{e}_2 = 11000$. *Then*
  $$C_2 = \{00000, 11011, \quad 11000, 00011\}$$
  is a linear code with basis $\{11011, 11000\}$.

- Choose any element $\mathbf{e}_3 \in V_n$ not already in your set C_2 (so $\mathbf{e}_3 \neq \mathbf{0}, \mathbf{e}_1, \mathbf{e}_2, \mathbf{e}_1 + \mathbf{e}_2$. This is your third basis element. To get C_3, add $\mathbf{e}_3$ to all elements of C_2, so
  $$C_3 = \{\mathbf{0}, \mathbf{e}_1, \mathbf{e}_2, \mathbf{e}_1 + \mathbf{e}_2, \quad \mathbf{e}_3, \mathbf{e}_1 + \mathbf{e}_3, \mathbf{e}_2 + \mathbf{e}_3, \mathbf{e}_1 + \mathbf{e}_2 + \mathbf{e}_3\}.$$
  This is a linear code of message length 3 and basis $\{\mathbf{e}_1, \mathbf{e}_2, \mathbf{e}_3\}$.

  In our example, we must have $\mathbf{e}_3 \notin \{00000, 11011, 11000, 00011\}$. *Take for example* $\mathbf{e}_3 = 11111$. *Then we find*
  $$C_3 = \{00000, 11011, 11000, 00011, \quad 11111, 00100, 00111, 11100\}$$
  is a linear code with basis $\{11011, 11000, 11111\}$.

- Continuing the same way, you can define linear codes C_k of arbitrary message length k, with basis $\{\mathbf{e}_1, ..., \mathbf{e}_k\}$.

4.3.3. Linear codes and Hamming weights. Let C be a linear code. Since C is closed under addition, we have $\mathbf{0} \in C$ is a codeword. The zero vector has an important role to play, giving a nicer description of minimum distances, based on *Hamming weights*.

Definition 4.6.

- The *weight* of $\mathbf{x} \in V_n$, denoted $w(\mathbf{x})$, is the number of 1's in $\mathbf{x}$, or equivalently the Hamming distance

$$w(\mathbf{x}) = d(\mathbf{x}, \mathbf{0}).$$

- The *minimum weight* of a code C, denoted $w(C)$, is

$$w(C) := \min\big\{w(\mathbf{c}) \,\big|\, \mathbf{0} \neq \mathbf{c} \in C\big\}.$$

Example.

- We have

$$w(00011101) = 4, \qquad w(001) = 1, \qquad w(10101) = 3.$$

- The minimum weight of $C = \{000, 111\}$ is $w(C) = 3$. This is equal to its minimum distance!
- The minimum weight of the paper tape code is $w(C) = 2$, again equal to the minimum distance.

Lemma 4.7. *(i) For* $\mathbf{x}, \mathbf{y} \in V_n$*, we have*

$$d(\mathbf{x}, \mathbf{y}) = w(\mathbf{x} - \mathbf{y}) = w(\mathbf{x} + \mathbf{y}).$$

(ii) Let C be a linear code. The minimum distance of C is equal to its minimum weight, i.e. $d(C) = w(C)$.

Proof. (i) Recall $d(\mathbf{x}, \mathbf{y})$ is the number of i's such that $x_i \neq y_i$. In $\mathbf{F}_2$, we have

$$x_i - y_i = x_i + y_i = \begin{cases} 0 & : x_i = y_i \\ 1 & : x_i \neq y_i \end{cases},$$

so

$$\begin{aligned} w(\mathbf{x} + \mathbf{y}) &= (\text{number of 1s in } \mathbf{x} + \mathbf{y}) \\ &= (\text{number of } i \text{ with } x_i \neq y_i) = d(\mathbf{x}, \mathbf{y}). \end{aligned}$$

(ii) Let $\mathbf{x}, \mathbf{y} \in C$ be codewords such that $d(C) = d(\mathbf{x}, \mathbf{y})$, i.e. that realise the minimum distance. As C is linear, $\mathbf{x} + \mathbf{y} \in C$ is also a codeword; and it is non-zero, because $\mathbf{x} \neq \mathbf{y}$ (by definition of minimum distance). Moreover $w(\mathbf{x} + \mathbf{y}) = d(\mathbf{x}, \mathbf{y})$ by (i), so we have

$$w(C) \leqslant w(\mathbf{x} + \mathbf{y}) = d(C),$$

where the first inequality follows as $w(\mathbf{x} + \mathbf{y})$ is a weight in C, and $w(C)$ is the smallest such weight.

Also, if $\mathbf{z} \in C$ is any codeword, we have $w(\mathbf{z}) = d(\mathbf{z}, \mathbf{0})$, so

$$\begin{aligned} w(C) &= \min\{d(\mathbf{z}, \mathbf{0}) : \mathbf{0} \neq \mathbf{z} \in C\} \\ &\geqslant \min\{d(\mathbf{x}, \mathbf{y}) : \mathbf{x} \neq \mathbf{y} \in C\} = d(C). \end{aligned}$$

Here the inequality follows as we have $\{d(\mathbf{z}, \mathbf{0})\} \subset \{d(\mathbf{x}, \mathbf{y})\}$, so the minimum of the latter set is at most the minimum of the former set.

As $w(C) \leqslant d(C)$ and $w(C) \geqslant d(C)$, we must have $w(C) = d(C)$. $\square$

4.4. Efficient encoding: the generator matrix

Given a basis $\{\mathbf{e}_1, ..., \mathbf{e}_k\}$ of a linear code, we can form a matrix with rows $\mathbf{e}_1, ..., \mathbf{e}_k$. This 'generator matrix' plays a very special role in the theory: it gives a convenient (and easily computable) encoding maps for linear codes.

Definition 4.8. Let $C \subset V_n$ be a linear code of dimension k. A matrix

$$G \in M_{k \times n}(\mathbf{F}_2)$$

is said to be a *generator matrix* for C if the row vectors $\{\mathbf{e}_1, ..., \mathbf{e}_k\}$ form a basis of C.

Example 4.9. Let C be the $3\times$-repetition code of message length 2, i.e. $k = 2$, $n = 6$, and $C = \{000000, 000111, 111000, 111111\}$ with encoding map

$$\begin{aligned} 00 &\longmapsto 000000 \\ 01 &\longmapsto 000111 \\ 10 &\longmapsto 111000 \\ 11 &\longmapsto 111111. \end{aligned}$$

Then $\{111000, 000111\}$ is a basis of C, so

$$G = \begin{pmatrix} 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 \end{pmatrix}$$

is a generator matrix. This is not unique! For example, we could also take the basis $\{111111, 000111\}$ and get a generator matrix

$$G = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 \end{pmatrix} \tag{4.1}$$

Example 4.10. In the paper tape code, the basis given above gives the generator matrix

$$G = \begin{pmatrix} 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix}.$$

Generating matrices give efficient encoding maps for linear codes. This is based on the following observation: let C be an $[n,k,d]$-code. If $\mathbf{w} \in V_k$ (a possible message for C), then $\mathbf{w}G \in V_n$. Even better, $\mathbf{w}G$ is in the span of the rows of G, so it must be a codeword!

Example. Consider Example 4.9 above. Then

$$(0,1)\begin{pmatrix}1 & 1 & 1 & 0 & 0 & 0\\ 0 & 0 & 0 & 1 & 1 & 1\end{pmatrix} = (0,0,0,1,1,1) = 000111,$$

whilst

$$(1,0)\begin{pmatrix}1 & 1 & 1 & 0 & 0 & 0\\ 0 & 0 & 0 & 1 & 1 & 1\end{pmatrix} = (1,1,1,0,0,0) = 111000.$$

Both are codewords.

Lemma 4.11. *Let C be a k-dimensional linear code, and let G be a generator matrix. Then the map*

$$\begin{aligned} G : V_k &\longrightarrow V_n, \\ \mathbf{w} &\longmapsto \mathbf{w}G, \end{aligned}$$

has image[a] *C. In other words, G defines an encoding map* $\mathbf{Enc} : V_k \xrightarrow{\text{1-to-1}} C \subset V_n$.

[a]Note this is *not* the usual definition of 'image of a matrix', which here would be a subset of V_k. It means C is the image of this specific map, as a subset of V_n.

Proof. This follows essentially straight from the definitions. The standard basis elements $e_i = (0,...,0,1,0,...,0)$ of V_k (with a 1 in the ith place) are mapped to the row vectors of G, i.e. to a basis $\{\mathbf{e}_1,...,\mathbf{e}_k\}$ of C. In symbols, we have $e_iG = \mathbf{e}_i$.

Here's a more detailed argument for why this map is bijective onto C:

- If $\mathbf{c} \in C$, then write
$$\mathbf{c} = \lambda_1\mathbf{e}_1 + \cdots + \lambda_k\mathbf{e}_k.$$
Then
$$\begin{aligned} \mathbf{c} &= \lambda_1\mathbf{e}_1 + \cdots + \lambda_k\mathbf{e}_k \\ &= \lambda_1(e_1G) + \cdots + \lambda_k(e_kG) \\ &= (\lambda_1e_1 + \cdots + \lambda_ke_k)G \\ &= (\lambda_1,...,\lambda_k)G. \end{aligned}$$
In particular, every $\mathbf{c} \in C$ is in the image, so the map is surjective onto C.
- The map is injective; recall that this is equivalent to its kernel being trivial (equal to 0). If
$$(\lambda_1,...,\lambda_k)G = \lambda_1\mathbf{e}_1 + \cdots + \lambda_k\mathbf{e}_k = 0,$$
then $\lambda_1 = \cdots = \lambda_k = 0$, since $\{\mathbf{e}_1,...,\mathbf{e}_k\}$ is a basis. Thus G is injective.

Thus G defines a bijective map $V_k \to C$, so it's an encoding map. □

Example. Consider $C = \{000, 111\}$. A basis is given by 111, so the generator matrix is $\begin{pmatrix}1 & 1 & 1\end{pmatrix}$. The encoding map is given by

$$0 \mapsto (0)\begin{pmatrix}1 & 1 & 1\end{pmatrix} = (0,0,0) = 000,$$
$$1 \mapsto (1)\begin{pmatrix}1 & 1 & 1\end{pmatrix} = (1,1,1) = 111,$$

as, of course, we expected.

Exercise. What is the encoding map if we instead use the generator matrix G from (4.1)? Compare this to the remark on page 14: the fact that generator matrices are not unique is closely related to the fact that encoding maps are not unique. Also convince yourself that not every possible encoding map for a linear code has the form $\mathbf{v} \mapsto \mathbf{v}G$ (hint: consider the 3 times repetition example immediately above).

4.5. Efficient decoding: the standard form

We've seen that generator matrices give an efficient *encoding* map for linear codes. We now give an efficient *decoding* map, too.

Example 4.12. Let C be the code with generator matrix

$$G = \begin{pmatrix} 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 \end{pmatrix}.$$

This code has $k = 3$ and $n = 6$. The associated encoding map is given by

$$x_1x_2x_3 \mapsto (x_1, x_2, x_3)\begin{pmatrix} 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 \end{pmatrix} = (x_1, x_2, x_3, *, *, *)$$

(the last three entries are irrelevant for this example). Then the decoding map is super-simple: the first three digits are *unchanged*, so to decode, we simply chop off the last 3 digits.

The reason this works is that the matrix G above starts with the 3×3 identity matrix (with diagonal 1s and 0s elsewhere), i.e. G starts with

$$G = \begin{pmatrix} 1 & 0 & 0 & \cdots \\ 0 & 1 & 0 & \cdots \\ 0 & 0 & 1 & \cdots \end{pmatrix}.$$

More generally, if G starts with the $k \times k$ identity matrix

$$G = \left(\begin{array}{ccc|ccc} 1 & & & a_{11} & \cdots & a_{1,n-k} \\ & \ddots & & \vdots & \ddots & \vdots \\ & & 1 & a_{k1} & \cdots & a_{k,n-k} \end{array}\right),$$

then the encoding map always starts

$$x_1 \cdots x_n \mapsto (x_1, ..., x_n, *, ..., *).$$

Note: When I write

$$I_k = \begin{pmatrix} 1 & & \\ & \ddots & \\ & & 1 \end{pmatrix},$$

this is shorthand for the $k \times k$ matrix which has 1s on the diagonal with all other entries being 0. So

$$I_3 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}.$$

Definition 4.13. A generator matrix G is said to be in *standard form* if it has the form

$$G = \left(\begin{array}{ccc|ccc} 1 & & & a_{11} & \cdots & a_{1,n-k} \\ & \ddots & & \vdots & \ddots & \vdots \\ & & 1 & a_{k1} & \cdots & a_{k,n-k} \end{array} \right)$$

for values $a_{i,j} \in \mathbf{F}_2$.

A natural question therefore is:

Can we always find a generator matrix in standard form?

The answer is *not always*. For example, consider the code $C = \{0000, 0011, 1100, 1111\}$. A generator matrix in standard form for this code would need to have a top row that starts $(1\ 0\ \cdots)$. However, all the codewords start with $(1\ 1\ \cdots)$ or $(0\ 0\ \cdots)$, and one of them must be the top row. So C does not admit a generator matrix in standard form.

However, a slight modification of C *does* admit a generator matrix in standard form. **Let's define a new code C' by swapping around the 2nd and 3rd digits in every codeword**, so

$$\begin{aligned} 0000 &\mapsto 0000, \\ 1100 &\mapsto 10101, \\ 0011 &\mapsto 0101, \\ 1111 &\mapsto 1111. \end{aligned}$$

The new code is

$$C = \{0000, 0101, 1010, 1111\}.$$

It is easy to see this is still a linear code. Moreover, it now admits a generator matrix in standard form, namely

$$G' = \begin{pmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{pmatrix}.$$

In this process, we have replaced C with an *equivalent code*. More generally:

Definition 4.14. Two codes C and C' in V_n are *equivalent* if C' can be obtained by applying the same position-wise permutation of digits to every element of C.

Example. The codes

$$\{00000, 10101, 11100, 01001\} \qquad \text{and} \qquad \{00000, 10101, 01101, 11000\}$$

are equivalent by applying the permutation (15), i.e. swap the 1st and 5th digits.

The codes

$$\{000, 110, 001, 111\} \qquad \text{and} \qquad \{000, 101, 100, 001\}$$

are not equivalent. Applying any permutation to 111 will give 111, but that is not in the second code.

If C and C' are equivalent, then the minimum distances $d(C) = d(C')$ and rates $r(C) = r(C')$ are the same; so for all practical purposes, we can always replace a code with an equivalent one.

Now, what does passing to an equivalent code do to generator matrices? In the codes C and C' above, swapping the 2nd and 3rd digits meant swapping the 2nd and 3rd columns of the corresponding generator matrix, i.e.

$$G = \begin{pmatrix} 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \end{pmatrix} \text{ (for } C) \qquad \text{became} G' = \begin{pmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{pmatrix} \text{ (for } C').$$

In particular, we could put our generator matrix into standard form using a standard matrix column operation. More generally, this motivates the following theorem:

Theorem 4.15 (Standard form)**.** *By replacing C with an equivalent code if necessary, we can choose*

$$G = (I_k A) = \left(\begin{array}{ccc|ccc} 1 & & & a_{11} & \cdots & a_{1,n-k} \\ & \ddots & & \vdots & \ddots & \vdots \\ & & 1 & a_{k1} & \cdots & a_{k,n-k} \end{array} \right).$$

Proof. Let G be a $k \times n$ generator matrix for C. Note that:

- Row operations (i.e. we add one row to another) do not change the code at all (since it corresponds to taking a different basis for C), and
- Permuting columns corresponding to permuting the bits in C, so replaces C with an equivalent code.

This is now an exercise in standard linear algebra – you will have already seen it, called either *Gaussian elimination* or *row reduction.* No row of G is all zero; so permute the columns so that $g_{11} = 1$, and then clear the first first column by adding the first row to other rows as necessary. Now repeat for g_{22}; the first entry of the second row is 0, and since the second row is not entirely 0, up to swapping the second column with a later column we may assume $g_{22} = 1$, and then clear the rest of the second column. Carrying on in this way we get G in the correct form. $\square$

We emphasise again the point of standard form. We saw that generator matrices give efficient encoding maps. If G is in standard form, then **Enc** takes the form

$$\begin{aligned} \mathbf{Enc} : V_k &\longrightarrow V_n \\ \mathbf{x} = (x_1, \ldots, x_k) &\longmapsto \mathbf{x}G = (\mathbf{x}, A\mathbf{x}) \\ &\qquad = (x_1, \ldots, x_k, y_{k+1}, \ldots, y_n), \end{aligned}$$

for some y_i's. In this case, the decoding map is simply 'chop off the last $n-k$ bits'. Since putting a matrix into standard form is efficient, this gives an efficient decoding map for linear codes.

Example 4.16. Consider the code given by generator matrix

$$G = \begin{pmatrix} 1 & 1 & 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 0 \end{pmatrix}.$$

(Actually, this turns out to be Hamming's code, from §1.3). We put this into standard form.

Already, we have $g_{11} = 1$, so we proceed to clear the first column, by adding row 1 to rows 3 and 4, yielding:

$$G = \begin{pmatrix} 1 & 1 & 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 & 0 & 1 \end{pmatrix}.$$

We also have $g_{22} = 1$, so clear the second column by adding row 2 to rows 1 and 3:

$$G = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 & 0 & 1 \end{pmatrix}.$$

Now $g_{33} = 0$, so we swap columns 3 and 4:

$$G = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 & 0 & 1 \end{pmatrix}.$$

Clear column 3 by adding row 3 to rows 2 and 4:

$$G = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 0 \end{pmatrix}.$$

Now we're in standard form.

($*$) **Remark.** You might ask if the standard form of a generator matrix is unique. This is an excellent question, with a subtle answer.

If you allow changing to an equivalent code, then the standard form is not in general unique. For example, if C is a code with $n = 6$ and $k = 4$, and G is a matrix in standard form for C, then you can often switch the last two columns of G and get another matrix in standard form, for an equivalent code.

If you don't allow equivalence, however, then the standard form is unique (if it exists; it might not, as we saw in lectures). In other words, you can only do row operations on the matrix, which correspond to changing the basis (and no column swaps, which pass to equivalent codes). In this case, any generator matrix is in reduced row echelon form, and such form is unique (exercise: look this up; it's on Wikipedia).

Finally, note that you can do Gaussian elimination to put any generator matrix into the unique reduced row echelon form for the code, without doing any column swaps. Then you get efficient encoding and decoding maps, just not quite as clean a form as we saw in this section. As an exercise, write down such a row-reduced matrix, defining a code, and write down the corresponding encoding and decoding maps.

4.6. Dual codes: the parity-check matrix

We now show an important method of building new codes out of old ones, via the *dual code.*

The *scalar product* of two vectors $\mathbf{x} = (x_1, \dots, x_n)$, $\mathbf{y} = (y_1, \dots, y_n) \in V_n$ is

$$\mathbf{x} \cdot \mathbf{y} = x_1 y_1 + \cdots + x_n y_n \in \mathbf{F}_2.$$

Note that this is the same as the matrix multiplication

$$\mathbf{x}\mathbf{y}^t = \begin{pmatrix} x_1 & \cdots & x_n \end{pmatrix} \begin{pmatrix} y_1 \\ \vdots \\ y_n \end{pmatrix}.$$

Definition 4.17. Let $C \subset V_n$ be a linear code. Its *dual code* is the orthogonal complement of C under scalar product, namely

$$C^\perp := \Big\{ \mathbf{x} \in V_n : \mathbf{x} \cdot \mathbf{c} = 0 \ \text{ for all } \mathbf{c} \in C \Big\}.$$

Example. Consider our old friend the 3 times repetition code $C = \{000, 111\}$. What is $C^\perp$?

We can compute a table of scalar products between all possible $\mathbf{x} \in V_3$ and $\mathbf{c} \in C$:

		$\mathbf{c}$		
	$\cdot$	000	111	$\mathbf{x} \in C^\perp$?
$\mathbf{x}$	000	0	0	✓
	001	0	1	✗
	010	0	1	✗
	100	0	1	✗
	110	0	0	✓
	101	0	0	✓
	011	0	0	✓
	111	0	1	✗

To calculate these entries, we have, for example, computed:

- $001 \cdot 111 = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} \cdot \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} = (0 \times 1) + (0 \times 1) + (1 \times 1) = 1,$

- $101 \cdot 111 = \begin{pmatrix}1\\0\\1\end{pmatrix} \cdot \begin{pmatrix}1\\1\\1\end{pmatrix} = (1 \times 1) + (0 \times 1) + (1 \times 1) = 0$, remembering that we always work modulo 2 (i.e. $1 + 1 = 0!$).

The $\mathbf{x}$ marked in red are those for which $\mathbf{x} \cdot \mathbf{c} = 0$ for all $\mathbf{c} \in C$, i.e. theses are the elements of $C^\perp$. We see

$$C^\perp = \{000, 110, 101, 011\}.$$

This is a linear code with $n = 3$, $k = 2$.

Theorem 4.18. *Let C be a linear code of dimension k.*

(i) We have $C^\perp = \ker(G)$, where G is any generator matrix of C.

(ii) $C^\perp$ is a linear code of dimension $n - k$.

(iii) We have $(C^\perp)^\perp = C$.

Proof. (i) Let $\{\mathbf{c}_1, ..., \mathbf{c}_k\}$ be a basis of C, giving a generator matrix G of C. We have $\mathbf{x} \in C^\perp$ if and only if $\mathbf{x}$ is orthogonal to each $\mathbf{c}_i$. This happens if and only if

$$\begin{pmatrix}\mathbf{c}_1 \cdot \mathbf{y}\\ \vdots \\ \mathbf{c}_k \cdot \mathbf{x}\end{pmatrix} = G\mathbf{x}^t = 0,$$

i.e. $\mathbf{x} \in \ker(G)$. Thus $C^\perp = \ker(G)$.

(ii) Kernels are linear subspaces, so $C^\perp$ is a linear code by (i). Moreover G has rank k (as it has k linearly independent rows), so by the Rank–Nullity Theorem its kernel $C^\perp$ has dimension $n - k$.

(iii) Let $\mathbf{c} \in C$. For all $\mathbf{x} \in C^\perp$, we have $\mathbf{c} \cdot \mathbf{x} = 0$, so $\mathbf{c} \in (C^\perp)^\perp$. Hence $C \subset (C^\perp)^\perp$. Thus they are equal, as they both have dimension k. □

Example. Let C be the 3 times repetition code. This has $n = 3$ and $k = 1$, so the dual code $C^\perp$ should have dimension $n - k = 3 - 1 = 2$. We computed this dual above to be $\{000, 101, 110, 011\}$, which is indeed 2-dimensional, with a basis given by $\{110, 101\}$.

Exercise.
- Let $C = \{00000000, 11111111\}$ be the 8 times repetition code. Show that the dual code $C^\perp$ is the paper tape code (from §1.2).
- Let $C_n := \{\mathbf{x} \in V_n : w(\mathbf{x}) \text{ is even}\}$. What is $C_n^\perp$?

We've seen that a generator matrix G attached to a code C has 'image'[1] C and kernel $C^\perp$. We get the symmetric-looking picture

	C	$C^\perp$	
$G =$ generator matrix of C	'Image'	Kernel	
	Kernel	'Image'	$H =$ generator matrix of $C^\perp$

[1]In a sense! The image of G is strictly a subset of V_k. However we can identify C as the image of the map $V_k \to V_n$ given by $\mathbf{x} \mapsto \mathbf{x}G$, so it's sort-of an image of G.

This suggests that we should study generator matrices of $C^\perp$.

Definition 4.19. A *parity-check matrix* (or just *check matrix*) for C is a generator matrix

$$H \in M_{(n-k)\times n}(\mathbf{F}_2)$$

for the dual code.

Lemma 4.20. *Let C be a k-dimensional linear code, and H a parity-check matrix. Then C is the kernel of H, i.e.*

$$C = \ker(H) = \Big\{\mathbf{x} \in V_n : H\mathbf{x}^t = 0\Big\}.$$

Proof. This is just Theorem 4.18(i), using that $C = (C^\perp)^\perp$ (by Theorem 4.18(iii)). □

Example. For the 3 times repetition code, we found that $\{110, 101\}$ was a basis of the dual code. Thus a parity-check matrix is given by

$$H = \begin{pmatrix} 1 & 1 & 0 \\ 1 & 0 & 1 \end{pmatrix}.$$

Example 4.21. We saw Hamming's code in the first chapter: it is the subset

$$C := \left\{ x_1 \cdots x_7 \in V_7 \;\middle|\; \begin{array}{l} x_1 + x_3 + x_5 + x_7 \equiv 0 \,(\mathrm{mod}\, 2) \\ x_2 + x_3 + x_6 + x_7 \equiv 0 \,(\mathrm{mod}\, 2) \\ x_4 + x_5 + x_6 + x_7 \equiv 0 \,(\mathrm{mod}\, 2) \end{array} \right\}.$$

This is a 4-dimensional linear subspace of V_7. Moreover, it is the kernel of the matrix

$$H = \begin{pmatrix} 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{pmatrix}.$$

So: Hamming's code is a linear code with $n = 7$ and $k = 4$, with parity-check matrix H.

Note that the columns of H are exactly the non-zero elements of V_3! This is the defining feature of Hamming's code.

In Hamming's code, we saw that all the columns of the parity-check matrix are different. Moreover this matrix has the most possible distinct non-zero columns for a binary matrix with 3 rows (if we added another column, it *must* have appeared already). Actually there is a *very good reason for this.*

In Theorem 3.11, we saw that the error-detecting and error-correcting properties of a code are entirely governed by its minimum distance. For a linear code, we can read this off from the parity-check matrix without even computing a single codeword!

Proposition 4.22. *Let C be a linear code with parity-check matrix H. The minimum distance of the code is equal to the minimum number of columns in H that sum to zero.*

Proof. As $C = \ker(H)$, for any $\mathbf{c} \in C$, we know

$$\begin{aligned} 0 &= \mathbf{c}H^t \\ &= \big[\text{sum of the columns of } H \text{ where } \mathbf{c} \text{ has a } 1\big]. \end{aligned}$$

Thus if j columns sum to 0, then there is a corresponding vector of weight j in $\mathbf{c}$.

Since C is linear, the minimum distance $d(C)$ is the smallest weight of a non-zero vector in C (by Lemma 4.7(ii)). The result follows. $\square$

From this proposition, we see easily that a linear code C has:

- minimum distance 1 $\iff$ H has a zero column;
- minimum distance 2 $\iff$ H has a repeated column;
- minimum distance $\geqslant 3$ $\iff$ H has all distinct columns.

In particular, for a code to be at least 1-error-correcting, the columns of H must be all different and non-zero.

Example. (i) All the columns in Hamming's code are different, so it has minimum distance $d(C) \geqslant 3$. It's easy to find three columns that sum to zero, so in fact it has $d(C) = 3$. Hence it is 2-error-detecting and 1-error-correcting by Theorem 3.11.

(ii) In the code of Example 4.31 (which we'll see below), there is a repeated column: so the minimum distance is 2. By Theorem 3.11, it is thus 1-error-detecting and 0-error-correcting.

So we see that the parity-check matrix is extremely useful for analysing codes: it makes it very easy to see the minimum distance. But:

How do we actually write down a parity-check matrix?

The following results give an algorithm to write down H (up to equivalence), exploiting standard form.

Lemma. *Let C be a linear code with generator matrix $G \in M_{k\times n}(\mathbf{F}_2)$. If $H \in M_{(n-k)\times n}(\mathbf{F}_2)$ has rank $(n-k)$ and*

$$HG^t = 0$$

is the zero $(n-k)\times k$ matrix, then H is a parity-check matrix for C.

Proof. Since H has rank $n-k$, it is the generator matrix of an $(n-k)$-dimensional linear code $C' \subset V_n$. The vanishing statement says that each row of H is orthogonal to the rows of G, so $C' \subset C^\perp$. But they have the same dimension, so $C' = C^\perp$ and H is a generator matrix of $C^\perp$, i.e. a parity-check matrix of C. $\square$

This gives us a algorithm computing parity-check matrices (using standard form).

Proposition 4.23. *Suppose C is a linear code that admits a generator matrix $G = (I_k \mid A)$*

in standard form[a], *with* $A = (a_{ij}) \in M_{k\times(n-k)}(\mathbf{F}_2)$. *Then*

$$H = (A^t I_{n-k}) = \left(\begin{array}{ccc|ccc} a_{11} & \cdots & a_{1k} & 1 & & \\ \vdots & \ddots & \vdots & & \ddots & \\ a_{n-k,1} & \cdots & a_{n-k,k} & & & 1 \end{array}\right)$$

is a parity-check matrix for C.

[a]Recall by Theorem 4.15 that this is always possible up to replacing C with an equivalent code.

Proof. Now let $H = (A^t|I_{n-k})$, a matrix of rank $n-k$ (all $n-k$ rows are linearly independent). Note that

$$\begin{aligned} HG^t = (A^t|I_{n-k})\begin{pmatrix} I_k \\ A^t \end{pmatrix} &= A^t + A^t \\ &= 2A^t = 0 \in M_{(n-k)\times k}(\mathbf{F}_2). \end{aligned}$$

So H is a parity-check matrix for C by the above Lemma. $\square$

Example. Consider the code from (the end of) Example 4.16, defined by the generator matrix

$$G = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 0 \end{pmatrix} = (I_4|A),$$

where

$$A = \begin{pmatrix} 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 0 \end{pmatrix}$$

We want to find a corresponding parity-check matrix. Note that

$$A^t = \begin{pmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 0 \end{pmatrix}.$$

Since $n = 7$ and $k = 4$, we have $n - k = 3$, so the proposition says that we can take

$$H = (A^t|I_3) = A^t = \begin{pmatrix} 0 & 1 & 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 \end{pmatrix}.$$

Note that up to reordering the columns, this is the same as the matrix defining Hamming's code: the column vectors are exactly all the non-zero binary strings of length 3. So this shows that the code we are studying is equivalent to Hamming's code.

Example. We show how the above results give a neat algorithm for computing minimum distance.

Let C be the linear code with generator matrix

$$G = \begin{pmatrix} 1 & 0 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 & 1 \end{pmatrix}.$$

This has $n = 6$ (as there are 6 columns), $k = 3$ (3 rows), and $n - k = 3$. We put this into standard form using the usual method (this is very easy in this example; we just need to add row 1 to row 2):

$$G = \begin{pmatrix} 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 & 1 \end{pmatrix} = \left(\begin{array}{c|ccc} & 1 & 0 & 0 \\ I_3 & 1 & 1 & 1 \\ & 1 & 1 & 1 \end{array}\right).$$

Then we get a parity-check matrix by

$$H = \left(\begin{array}{ccc|c} 1 & 1 & 1 & \\ 0 & 1 & 1 & I_3 \\ 0 & 1 & 1 & \end{array}\right) = \begin{pmatrix} 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 & 1 \end{pmatrix}.$$

We see that:

- There is no zero column, so $d(C) > 1$.
- There is a repeated column (the 1st and 4th, or the 2nd and 3rd) so $d(C) = 2$.

We've computed $d(C)$ (via H), an encoding map (via G) and a decoding map (via standard form) without ever writing down the elements of C.

4.7. Error correction for linear codes

One crucial reason for using linear codes is that when $n - k$ is small, there is an efficient algorithm for error correction.

Let C be a linear code of dimension k. Suppose we have received $\mathbf{y}$ and want to decode. Under nearest-neighbour/maximum-likelihood decoding, we want to find the codeword $\mathbf{c_y} \in C$ such that:

(1) $d(\mathbf{y}, \mathbf{c_y})$ is smallest,

(2) which by Lemma 4.7 is when $w(\mathbf{y} + \mathbf{c_y})$ is smallest.

(3) Thus we want to find the element of smallest weight in

$$\mathbf{y} + C := \{\mathbf{y} + \mathbf{c} : \mathbf{c} \in C\}.$$

(4) Suppose $\mathbf{e_y}$ is the element of smallest weight in $\mathbf{y} + C$. Then $\mathbf{e_y} = \mathbf{y} + \mathbf{c_y}$, and we should decode $\mathbf{y}$ as

$$\mathbf{c_y} = \mathbf{y} + (\mathbf{y} + \mathbf{c_y}) = \mathbf{y} + \mathbf{e_y}.$$

Note that in step (4), *we do not need to find* $\mathbf{c_y}$ *itself* – we only need to find $\mathbf{e_y}$, the element of smallest weight in $\mathbf{y} + C$, as from this we can *compute* $\mathbf{c_y}$ by adding $\mathbf{y}$. This observation is extremely important!!

Definition 4.24.
- The *coset* attached to $\mathbf{y} \in V_n$ is $\mathbf{y} + C$.
- The *coset leader* attached to $\mathbf{y}$ is the element $\mathbf{e_y}$ of smallest weight in $\mathbf{y} + C$.

Remark. Note that the coset leader will not be unique. Indeed, as an exercise, prove that

$$\text{every coset } \mathbf{y} + C \text{ has a unique coset leader} \iff C \text{ is perfect.}$$

We summarise the above discussion in the following theorem.

Theorem 4.25. *We can pass from any $\mathbf{y} \in V_n$ to its nearest codeword $\mathbf{c_y}$ by adding the corresponding coset leader, i.e.*

$$\mathbf{c_y} = \mathbf{y} + \mathbf{e_y}.$$

Example 4.26. Consider once again the 3 times repetition code $C = \{000, 111\}$, and $\mathbf{y} = 100$. The coset attached to $\mathbf{y}$ is

$$\mathbf{y} + C = \{100 + 000, 100 + 111\} = \{100, 011\}.$$

The coset leader is $\mathbf{e_y} = 100$. We decode $\mathbf{y}$ as

$$\mathbf{c_y} = \mathbf{y} + \mathbf{e_y} = 100 + 100 = 000.$$

If instead we started with $\mathbf{z} = 011$, then note that the coset

$$\mathbf{z} + C = \{011, 100\} = \mathbf{y} + C$$

is the same! So we also have $\mathbf{e_z} = \mathbf{e_y} = 100$, and we should decode $\mathbf{z}$ as

$$\mathbf{c_z} = \mathbf{z} + \mathbf{e_z} = 011 + 100 = 111.$$

Returning to the general setting, we make the following observation.

Lemma 4.27. *Let $C \subset V_n$ be a linear code of dimension k.*

(i) For any $\mathbf{y} \in V_n$, we have $\mathbf{y} \in \mathbf{y} + C$.

(ii) If $\mathbf{y}, \mathbf{y}' \in V_n$, then

$$\mathbf{y} + C = \mathbf{y}' + C \quad \iff \quad \mathbf{y} - \mathbf{y}' \in C.$$

(iii) Any two cosets $\mathbf{y} + C$ and $\mathbf{y}' + C$ are equal or disjoint.

(iv) There are exactly 2^{n-k} different cosets for C in V_n, each of size 2^k.

Proof. (i) We have $\mathbf{y} = \mathbf{y} + \mathbf{0} \in \mathbf{y} + C$, because $\mathbf{0}$ is always a codeword in a linear code.

(ii) $\Rightarrow$ If $\mathbf{y} + C = \mathbf{y}' + C$, then $\mathbf{y} \in \mathbf{y} + C = \mathbf{y}' + C$ by (i). In particular $\mathbf{y} = \mathbf{y}' + \mathbf{c}$ for some $\mathbf{c} \in C$. Thus $\mathbf{y} - \mathbf{y}' = \mathbf{c} \in C$.

$\Leftarrow$ Suppose $\mathbf{y} - \mathbf{y}' \in C$. Then there exists $\mathbf{c}' \in C$ such that $\mathbf{y} - \mathbf{y}' = \mathbf{c}'$, i.e. $\mathbf{y} = \mathbf{y}' + \mathbf{c}'$. Then for any $\mathbf{c} \in C$, we have $\mathbf{y} + \mathbf{c} = \mathbf{y}' + (\mathbf{c} + \mathbf{c}') \in \mathbf{y}' + C$, since $\mathbf{c} + \mathbf{c}' \in C$ (by linearity). Thus $\mathbf{y} + C \subset \mathbf{y}' + C$.

But we can argue in reverse, since also $\mathbf{y}' = \mathbf{y} - \mathbf{c}' = \mathbf{y} + \mathbf{c}'$. This would show $\mathbf{y}' + C \subset \mathbf{y} + C$, hence $\mathbf{y} + C = \mathbf{y}' + C$.

Figure 4.1: A very bad lookup table for Hamming's code:

Label	**Coset** $\mathbf{y}+C$	**Coset leader** $\mathbf{e_y}$
D_1	0000000, 1110000, 1001100, 0111100, 0101010, 1011010, 1100110, 0010110, 1101001, 0011001, 0100101, 1010101, 1000011, 0110011, 0001111, 1111111	0000000
D_2	1000000, 0110000, 0001100, 1111100, 1101010, 0011010, 0100110, 1010110, 0101001, 1011001, 1100101, 0010101, 0000011, 1110011, 1001111, 0111111	1000000
D_3	0100000, 1010000, 1101100, 0011100, 0001010, 1111010, 1000110, 0110110, 1001001, 0111001, 0000101, 1110101, 1100011, 0010011, 0101111, 1011111	0100000
D_4	0010000, 1100000, 1011100, 0101100, 0111010, 1001010, 1110110, 0000110, 1111001, 0001001, 0110101, 1000101, 1010011, 0100011, 0011111, 1101111	0010000
D_5	0001000, 1111000, 1000100, 0110100, 0100010, 1010010, 1101110, 0011110, 1100001, 0010001, 0101101, 1011101, 1001011, 0111011, 0000111, 1110111	0001000
D_6	0000100, 1110100, 1001000, 0111000, 0101110, 1011110, 1100010, 0010010, 1101101, 0011101, 0100001, 1010001, 1000111, 0110111, 0001011, 1111011	0000100
D_7	0000010, 1110010, 1001110, 0111110, 0101000, 1011000, 1100100, 0010100, 1101011, 0011011, 0100111, 1010111, 1000001, 0110001, 0001101, 1111101	0000010
D_8	0000001, 1110001, 1001101, 0111101, 0101011, 1011011, 1100111, 0010111, 1101000, 0011000, 0100100, 1010100, 1000010, 0110010, 0001110, 1111110	0000001

(iii) Suppose $\mathbf{z} \in (\mathbf{y}+C) \cap (\mathbf{y}'+C)$ (i.e. suppose these cosets are not disjoint). Then $\mathbf{z} = \mathbf{y}+\mathbf{c} = \mathbf{y}+\mathbf{c}'$ for some $\mathbf{c}, \mathbf{c}' \in C$. But then $\mathbf{y}-\mathbf{y}' = \mathbf{c}'-\mathbf{c} \in C$, so $\mathbf{y}+C = \mathbf{y}'+C$ by (ii).

(iv) Each coset is in bijection with C, so has 2^k elements. By (iii), any two cosets are either equal or disjoint. Also every element $\mathbf{y}$ of V_n is in a coset (e.g. $\mathbf{y}+C$). So V_n, which has size 2^n, is covered by cosets, each of size 2^k. There must therefore be $2^n/2^k = 2^{n-k}$ cosets. □

Note that if $\mathbf{y}$ and $\mathbf{y}'$ have the same coset, then they have the same coset leader, as this depends only on the coset! This also means that $\mathbf{y}+C = \mathbf{e_y}+C$, so that any coset can be 'generated' by a coset leader.

Let's call the distinct cosets $D_1, ..., D_{2^{n-k}}$, and compute a coset leader for each D_i. We store all of these in a *lookup table.* For example, for Hamming's code, this naive table is computed in Figure 4.1.

A (very bad!) algorithm for decoding $\mathbf{y}$ might now be:

- Search through the 'coset' column of the table until you find the row containing $\mathbf{y}$.
- Read off the corresponding coset leader $\mathbf{e_y}$.
- Decode $\mathbf{y}$ as $\mathbf{c_y} = \mathbf{y}+\mathbf{e_y}$.

This is bad because searching the table for $\mathbf{y}$ is not at all efficient. We want a better way to find the coset leader attached to $\mathbf{y}$.

Idea: Suppose can we find an efficient function S on V_n such that $S(\mathbf{y}) = S(\mathbf{y}')$ if and only if $\mathbf{y}+C = \mathbf{y}'+C$. Then we can very easily tell which coset $\mathbf{y}$ lies in: *we just have to compute* $S(\mathbf{y})$. Such a function is given by *syndromes.*

Definition 4.28. Let C be a linear code with parity-check matrix H. For $\mathbf{y} \in V_n$, the *syndrome* of $\mathbf{y}$ is

$$S(\mathbf{y}) = \mathbf{y}H^t \in V_{n-k}.$$

Lemma 4.29. *Two elements* $\mathbf{y}, \mathbf{y}'$ *in* V_n *have the same syndrome if and only if* $\mathbf{y}+C = \mathbf{y}'+C$ *(i.e. they lie in the same coset).*

Proof. Recall C is the kernel of H. We know:

$$\begin{aligned} \mathbf{y} \text{ and } \mathbf{y}' \text{ lie in the same coset} &\iff \mathbf{y} - \mathbf{y}' \in C \\ &\iff H(\mathbf{y} - \mathbf{y}')^t = 0 \\ &\iff (\mathbf{y} - \mathbf{y}')H^t = 0 \\ &\iff \mathbf{y}H^t = \mathbf{y}'H^t. \end{aligned}$$ □

Remark. Note that there are 2^{n-k} possible syndromes, and 2^{n-k} possible cosets. Thus the lemma says that every element of V_{n-k} is a syndrome for exactly one coset.

Given a $\mathbf{y}$, computing its associated syndrome is very efficient. Thus a much better lookup table stores *not the coset*, but its associated syndrome, as in Figure 4.2.

Figure 4.2: An excellent lookup table for Hamming's code:

Label	**Syndrome** $\mathbf{y}H^t$	**Coset leader** $\mathbf{e_y}$
D_1	$(0,0,0)$	0000000
D_2	$(1,0,0)$	1000000
D_3	$(0,1,0)$	0100000
D_4	$(1,1,0)$	0010000
D_5	$(0,0,1)$	0001000
D_6	$(1,0,1)$	0000100
D_7	$(0,1,1)$	0000010
D_8	$(1,1,1)$	0000001

Now we have a good error-correction algorithm!

Algorithm 4.30 (Error-correction in a linear code)**.**

I: One-time work.

(i) Write down a parity-check matrix for C.

(ii) For each coset $\mathbf{y} + C$, compute its associated syndrome $\mathbf{y}H^t$ and associated coset leader $\mathbf{e_y}$.

(iii) Store all the syndromes and associated coset leaders in a lookup table.

II: Every-time work. To decode a given $\mathbf{y}$:

(1) Compute the syndrome $\mathbf{y}H^t$.

(2) Look up the syndrome in the table, and read off the corresponding $\mathbf{e_y}$.

(3) Return the codeword $\mathbf{c_y} = \mathbf{y} + \mathbf{e_y}$.

Remark. As steps (i), (ii) and (iii) are one-time work, it's ok if they take a long time: once they are done, we need never do them again. (However, see below to see how we can compute (ii)).

In the every-time work, step (3) is always efficient. Step (1) can take a long time if n is very large, as the matrix multiplication takes $O(n(n-k))$ operations[2]. Step (2) is efficient when $n-k$ is small (e.g. for Hamming's code); but as $n-k$ grows, the size of the lookup table – which has 2^{n-k} entries – grows *exponentially*, and the efficiency of the algorithm suffers accordingly.

Whilst step (ii) is one-time work, it's still worth finding an algorithm that's better than 'list every element of all the cosets'. In practice, the coset leader will have very small weight, so we can start by computing the syndromes of all the vectors of weight 1, and then weight 2, etc.

Example 4.31 (Writing down a lookup table)**.** Let C be the linear code with parity-check matrix

$$H = \begin{pmatrix} 1 & 1 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 & 1 \end{pmatrix}.$$

This has $k = 4$ and $n = 7$, like Hamming's code; but we will see it is not as good[3].

There are $2^{n-k} = 8$ syndromes, so the lookup table will have 8 rows. We compute syndromes of vectors of low weight:

- 0000000 is the only vector of weight 0, and clearly it has syndrome 000.
- We find the syndromes of weight 1 vectors are:

$$\begin{aligned} (1,0,0,0,0,0,0)H^t &= (1,1,0) = \text{1st column of } H, \\ (0,1,0,0,0,0,0)H^t &= (1,0,0) = \text{2nd column of } H, \\ &\vdots \\ (0,0,0,0,0,0,1)H^t &= (0,1,1) = \text{last column of } H. \end{aligned}$$

In particular, we can populate most of our lookup table already:

[2]Recall big-Oh notation: if

$$f : \mathbf{R} \longrightarrow \mathbf{C} \qquad \text{and} \qquad g : \mathbf{R} \longrightarrow \mathbf{C}$$

are two functions, we say that f is *big-Oh* of g, written $f = O(g)$, as x tends to 0 (or ∞) if there exist a constant c such that

$$|f(x)| \leq c \cdot |g(x)| \quad \text{as } x \to 0 \text{ (or } x \to \infty).$$

[3]Indeed, whilst Hamming's code is 2-error-detecting and 1-error-correcting, this new code is 1-error-detecting and 0-error-correcting!

Label	**Syndrome** $\mathbf{y}H^t$	**Coset leader** $\mathbf{e_y}$
D_1	$(0,0,0)$	0000000
D_2	$(1,1,0)$	1000000 *or* 0010000
D_3	$(1,0,0)$	0100010
D_4	$(0,0,1)$	0001000
D_5	$(0,1,0)$	0000100
D_6	$(1,1,1)$	0000010
D_7	$(0,1,1)$	0000001

Note that the 1st and 3rd columns of H are the same, so 1000000 and 0010000 have the same syndrome. We could have chosen *either* as the coset leader in D_2: both are equally valid.[4]

- There is one syndrome in V_3 missing: $(1,0,1)$. Since we have already computed all the syndromes $\mathbf{y}H^t$ with $w(\mathbf{y}) = 1$, the coset leader must have weight *at least* 2. Since $(1,0,1)$ is the sum of the 2nd and 4th columns of H, we see that

$$(0,1,0,1,0,0,0)H^t = (1,0,1),$$

so the last line of our lookup table is

Label	**Syndrome** $\mathbf{y}H^t$	**Coset leader** $\mathbf{e_y}$
D_8	$(1,0,1)$	0101000

It is worth emphasising here: this is also not a unique coset leader. Since the 1st and 7th columns also sum to (1,0,1), we could also have taken 1000001 (or, indeed, 00100001, or 0000110, or...)

Example 4.32. Let's use the lookup table from Example 4.31 to decode some messages. Alice and Bob agree to use the code C from that example. Alice sends a message.

- Suppose Bob receives $\mathbf{y} = 1001011$. To decode, he computes the syndrome:

$$\mathbf{y}H^t = (1,0,0,1,0,1,1)\begin{pmatrix}1&1&0\\1&0&0\\1&1&0\\0&0&1\\0&1&0\\1&1&1\\0&1&1\end{pmatrix} = (0,1,1).$$

From our lookup table, the corresponding coset leader is $\mathbf{e_y} = 0000001$. So we decode $\mathbf{y}$ as

$$\mathbf{c_y} = \mathbf{y} + \mathbf{e_y} = 1001010.$$

To double-check all our computations, we can check that $H\mathbf{c}_{\mathbf{y}}^t = 0$, to be sure $\mathbf{c_y}$ is a codeword: and this is true.

[4]The fact there is no unique choice of coset leader means that the code is not perfect, and we sometimes have to make a choice in error-correction.

It's worth doing this check when you compute things by hand. Whilst I was writing down this example, I made an error, and got the syndrome to be $(0,1,0)$*, giving* $\mathbf{c_y} = 1001111$*. But then* $H\mathbf{c_y}^t = (0,0,1) \neq 0$*, so this value of* $\mathbf{c_y}$ *is* not *a codeword, so the decoding step must have gone wrong somewhere. I was then able to easily correct my mistake.*

- Now suppose Bob receives 0101000. Now the syndrome is

$$\mathbf{y}H^t = (0,1,0,0,1,0,0)\begin{pmatrix} 1 & 1 & 0 \\ 1 & 0 & 0 \\ 1 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 1 & 1 \\ 0 & 1 & 1 \end{pmatrix} = (1,1,0).$$

This is the bad coset where we don't have a unique coset leader. We can choose either $\mathbf{e_y} = 1000000$ or 0010000, and decode 0101000 as either

$$\mathbf{c_y} = \mathbf{y} + 1000000 = 1100100 \qquad \text{or} \qquad \mathbf{c_y} = \mathbf{y} + 0010000 = 01101000.$$

Both are equally valid, and equally close. Bob has no reason to believe Alice sent one over the other. (This, of course, makes this a very bad code indeed, as there has probably only been one error).

4.8. (∗) Further topics

4.8.1. Existence of $[n,k,d]$-codes. Given an (n,k,d)-code, we have obtained bounds (sphere-packing and Singleton) on n, k and d. Proposition 4.22 allows us to prove a kind of converse, namely an existence theorem for linear codes.

Theorem 4.33 (Gilbert–Garsharmov)**.** *Fix n, k and e. If*

$$2^{n-k} > \sum_{j=0}^{2e-1} \binom{n-1}{j},$$

there is an $[n,k,2e+1]$-code (i.e. a k-dimensional linear e-error-correcting code of length n).

Proof. A k-dimensional code of length n is determined by an $(n-k)\times n$ parity-check matrix. By Proposition 4.22, it suffices to write down such a matrix such that no linear combination of $2e$ (or fewer) columns sums to 0. We construct such a matrix inductively.

For $r \leqslant n-1$, suppose we have written down r columns $\mathbf{x}_1, \cdots, \mathbf{x}_r \in V_{n-k}$ with this property. We must write down a vector $\mathbf{x}_{r+1}$ that is not equal to the sum of up to $2e-1$ of the other $\mathbf{x}_i$'s. There are exactly $\binom{r}{0}$ ways of writing down a sum of zero vectors; exactly $\binom{r}{1}$ ways of writing down a sum $\mathbf{x}_i$, exactly $\binom{r}{2}$ ways of writing down sums $\mathbf{x}_i + \mathbf{x}_j$, and so on; so there are

$$\binom{r}{0}+\binom{r}{1}+\binom{r}{2}+\cdots+\binom{r}{2e-1}=\sum_{j=0}^{2e-1}\binom{r}{j}$$

ways of writing down sums of up to $2e-1$ vectors.

Now, by the condition we have

$$2^{n-k}>\sum_{j=0}^{2e-1}\binom{n-1}{j}\geqslant\sum_{j=0}^{2e-1}\binom{r}{j},$$

using that $n-1\geqslant r$. So there is always a column vector $\mathbf{x}_{r+1}\in V_{n-k}$ that is not a sum of up to $2e-1$ of the $\mathbf{x}_1,...,\mathbf{x}_r$. We can thus build all n columns $\mathbf{x}_1,...,\mathbf{x}_n\in V_{n-k}$ by induction. $\square$

4.8.2. Error probabilities for linear codes. Finally, we conclude this chapter by describing error probabilities for linear codes.

Definition 4.34. Given a code C, let

$$\mathbf{P}_{\text{err}}:=\mathbf{P}(\text{there is an error, even after correction}).$$

Suppose we work over a binary symmetric channel of error probability p. For perfect codes, one can compute $\mathbf{P}_{\text{err}}$ directly, but this is rather more difficult for general linear codes (for example, the code from Example 4.31). However, you can easily compute $\mathbf{P}_{\text{err}}$ from a syndrome lookup table:

Theorem 4.35. *Let C be a linear code, and fix a syndrome lookup table for C. For each $i\geqslant 0$, let a_i be the number of coset leaders that have weight i. Then*

$$\mathbf{P}_{\text{err}}=1-\sum_{i=0}^{n}a_ip^i(1-p)^{n-i}.$$

Proof. Suppose $\mathbf{c}$ is sent. If $\mathbf{y}$ is received, it is correctly decoded if and only if it has the form

$$\mathbf{y}=\mathbf{c}+\mathbf{e},$$

where $\mathbf{e}$ is one of the coset leaders. For a fixed $\mathbf{e}$ of weight i, the probability that the word $\mathbf{c}$ is sent to exactly $\mathbf{c}+\mathbf{e}$ is

$$\mathbf{P}\big(\text{receive }\mathbf{c}+\mathbf{e}\mid\text{send }\mathbf{c}\big)=p^i(1-p)^{n-i}.$$

Thus the probability there is *no* error is

$$\sum_{i=0}^{n}a_ip^i(1-p)^{n-i},$$

but $\mathbf{P}(\text{no error})=1-\mathbf{P}_{\text{err}}$. $\square$

Example 4.36. Suppose we send a message of length $k=4$.

- If we perform no error correction at all, this is the same as taking $n = k = 4$. There is just one coset, C itself, with coset leader $(0, 0, 0, 0)$. Thus $a_0 = 1$, and $a_i = 0$ for $i \geqslant 1$, and

  $$\begin{aligned}\mathbf{P}_{\text{err}} &= 1 - \Big[1 \times (1-p)^4\Big] \\ &= 4p - 6p^2 + \cdots \approx 4p\end{aligned}$$

  (when p is small).

- In Hamming's code, we have $a_0 = 1$, $a_1 = 7$, and $a_i = 0$ for $i \geqslant 2$. Thus

  $$\begin{aligned}\mathbf{P}_{\text{err}} = 1 - \Big[1 \times (1-p)^7\Big] - \Big[7 \times p(1-p)^6\Big]& \\ = 21p^2 - 70p^3 + \cdots \approx 21p^2&.\end{aligned}$$

  Thus Hamming's code is hugely improve for small p. (Note we will always have a factor of np coming from the one coset leader of weight 0. Crucially here, there are n coset leaders of weight 1, which contributes a $-np$ that cancels, leaving only the terms p^2 and higher).

- In the bad code of Example 4.31, we have $n = 7$, and $a_0 = 1, a_1 = 6, a_2 = 1$ and $a_i = 0$ for $i \geqslant 3$. Now the factors of p do not cancel, and we get

  $$\begin{aligned}\mathbf{P}_{\text{err}} &= 1 - \Big[1 \times (1-p)^7\Big] - \Big[6 \times p(1-p)^6\Big] - \Big[1 \times p^2(1-p)^5\Big] \\ &= p + 14p^2 + \cdots \approx p.\end{aligned}$$

  We've improved on no error correction only by a factor of 7. Even 3 times repetition is better than this!

Chapter 5

Families of good codes

We now use the theory of linear codes, as developed in the previous chapter, to write down two families of high quality codes: *Hamming codes*, which are perfect 1-error-correcting codes with very efficient rate, and *Reed–Muller codes*, which are highly accurate codes historically used by NASA.

Hamming codes allow you to to trade greater efficiency for lower accuracy; whilst Reed–Muller codes allow you to trade greater accuracy for lower efficiency.

5.1. Hamming codes

We have already seen one version of Hamming's code: it was the code with parity-check matrix

$$H = \begin{pmatrix} 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{pmatrix}.$$

This is the matrix where the columns are exactly the 7 non-zero elements of V_3.

Using the previous chapter, we see where Hamming's code comes from. Whenever we try and construct a good code, we want to ensure:

– **we don't send too much extra information** (i.e. the rate $r(C) = k/n$ is large);

– C **corrects as many errors as possible** (i.e. the minimum distance $d(C)$ is large, by Theorem 3.11).

– **We have good algorithms for encoding, decoding and error correction** (i.e. we should use a linear code).

Then note:

- Any linear code C of dimension k will be the kernel of an $(n-k) \times n$ parity-check matrix H.
- If you fix $s = n - k$, the rate k/n is larger as n increases (e.g. $3/6 < 4/7 < 5/8$ etc. when $s = 3$). Thus, for the best efficiency, you want the most possible columns in H.
- For $d(C) \geqslant 3$, by Proposition 4.22 all the columns of H must be different and non-zero.
- For fixed $s = n - k$, the maximum number of different non-zero columns is $2^s - 1$, where the columns of H are all the non-zero elements of V_s. This defines a parity-check matrix

$$H_s \in M_{s\times(2^s-1)}(\mathbf{F}_2).$$

The matrix H_s is the parity-check matrix of a linear code C with parameters

$$n = 2^s - 1, \qquad k = (2^s - 1) - s, \qquad d(C) = 3.$$

In other words, you get a $[2^s - 1,\ 2^s - s - 1,\ 3]$-code.

Definition 5.1. The *Hamming code* $\mathrm{Ham}(s)$ is the linear code defined by the parity-check matrix H_s, i.e.

$$\mathrm{Ham}(s) := \ker(H_s) \subset V_{2^s-1}.$$

Given all the theory we developed for linear codes, the following is straightforward to prove; I leave the proof as an exercise.

Theorem 5.2. *The Hamming code* $\mathrm{Ham}(s)$ *is a perfect 1-error-correcting* $[2^s-1,\ 2^s-s-1,\ 3]$-*code.*

Hamming's original code is the case where $s = 3$, giving a perfect $[7, 4, 3]$-code.

Remark 5.3. Note that the rate

$$\frac{2^s - s - 1}{2^s - 1} \longrightarrow 1$$

tends to 1 as $s \to \infty$; i.e. as s increases, the efficiency tends to perfect. However, for *all* values of s, the code $\mathrm{Ham}(s)$ can only correct **one** error. This means that as s grows, $\mathrm{Ham}(s)$ becomes less accurate. Precisely, by Theorem 4.35 we can work the error probability out to be

$$\mathbf{P}_{\mathrm{err}}[\mathrm{Ham}(s)] = \binom{2^s - 1}{2} p^2 + O(p^3),$$

which is growing with s. Thus as s grows, the improvement in rate is offset by a reduction in accuracy.

To illustrate this, compare:

- For $s = 10$, for example, we have $n = 1023$;
- For $s = 3$, we have $n = 7$.

Thus $\mathrm{Ham}(10)$ can correct up to 1 error in every block of 1023 transmissions, whilst $\mathrm{Ham}(3)$ can correct up to 1 error in every block of 7 transmissions. In concrete terms, if $p = 10^{-5}$ and we send a 1MB message:

- if we send using $\mathrm{Ham}(10)$ then we have only around 1% redundancy but the probability of error-free completion is only around 66%.
- if we send using $\mathrm{Ham}(3)$, then we have 75% redundancy, but the probability of error-free completion rises to 99.8%.

Note that if we sent using no error-correction at all, then the probability of error-free completion is $0.00\cdots003\%$ (35 zeros), which is essentially 0. So using $\mathrm{Ham}(10)$ is a huge win: for only 1% redundancy it immensely improves the accuracy.

5.2. Reed–Muller codes

For the Hamming code Ham(s), as $s \to \infty$, we saw you can trade increased efficiency for reduced accuracy.

What if you want to go the other way? If you are transmitting over a particularly noisy channel, e.g. transmitting from a NASA spacecraft, then errors will be more frequent and improved accuracy of error-correction becomes paramount. Even the most accurate Hamming codes can only correct one error (they have an $O(p^2)$-error probability), and for 'larger' p this is still unacceptably high. Instead, to transmit pictures of Mars back to Earth, the Mariner and Viking NASA spacecrafts used *Reed–Muller codes*, where efficiency is sacrificed for improved accuracy.

We will describe a family $R(1, m)$ of Reed–Muller codes here. NASA used the code $R(1, 5)$, for which the headline is:

The Reed–Muller code $R(1, 5)$ *is a* $[32, 6, 16]$*-code.*

The rate here is 0.1875, which is bad; but Theorem 3.11 tells us $R(1, 5)$ is *7-error-correcting*, so it is stunningly accurate.

Example (Reed–Muller vs. other codes)**.** Suppose we want to send a message that is 8000 bits, or 1 kilobyte, long.

- If $p = 0.05$, which is *terrible*, then the probability of sending error-free via no code, a Hamming code, and a Reed–Muller code are:

Code used	**P**(sending 1kb error-free)
None	$0.00\cdots006\%$ (176 zeros)
Ham(3)	$0.00\cdots004\%$ (37 zeros)
$R(1,5)$	96.6%.

How is this computed?

  – If we send with no code, the probability of sending with no error is

$$(1-p)^{8000} = 0.95^{8000} \approx 6 \cdot 10^{-179}.$$

  – Sending with Ham(3) we break the message into 2000 messages of length 4, each of which sends correctly with probability $P(\text{no errors}) + P(1 \text{ error}) = 0.95^7 + 7 \cdot 0.95^6 \cdot 0.05$. So the probability of sending the whole message with no errors is

$$[0.95^7 + 7 \cdot 0.95^6 \cdot 0.05]^{2000} \approx 4 \cdot 10^{-40}.$$

For Reed–Muller, we break this into 250 messages of length 32, each of which sends correctly with probability

$$P(\text{no error}) + P(1 \text{ error}) + \cdots + P(7 \text{ errors}).$$

This gives the claimed percentage: we must now compute

$$[0.95^{32} + 32 \cdot 0.95^{31} \cdot 0.05 + \binom{32}{2} 0.95^{30} \cdot 0.05^2 + \cdots + \binom{32}{7} 0.95^{25} 0.05^7]^{250} \approx 0.966.$$

(Disclaimer: I may have entered this wrong into WolframAlpha as this equation is too big for it).

- If we dropped p to be 0.01, which is still bad, then the probabilities become:

Code used	$\mathbf{P}$(sending 1kb error-free)
None	$0.00\cdots001\%$ (32 zeros)
Ham(3)	1.7%
$R(1,5)$	99.9999%.

Because the rate of $R(1,5)$ is only 0.1875, we needed to send 5.3 times as much information to get this accuracy. Despite this, over a very noisy channel, this is a vastly superior code.

Definition 5.4. For $m \geqslant 1$, define a sequence of matrices

$$G_m \in M_{(m+1)\times 2^m}(\mathbf{F}_2)$$

recursively as follows:

- $G_1 = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}$,
- $G_2 = \left(\begin{array}{cc|cc} \multicolumn{2}{c|}{G_1} & \multicolumn{2}{c}{G_1} \\ \hline 0 & 0 & 1 & 1 \end{array}\right) = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \end{pmatrix}$,
- $G_3 = \left(\begin{array}{cccc|cccc} \multicolumn{4}{c|}{G_2} & \multicolumn{4}{c}{G_2} \\ \hline 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{array}\right) = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{pmatrix}$,
- and so on, with

$$G_m = \left(\begin{array}{ccc|ccc} \multicolumn{3}{c|}{G_{m-1}} & \multicolumn{3}{c}{G_{m-1}} \\ \hline 0 & \cdots & 0 & 1 & \cdots & 1 \end{array}\right)$$

The *Reed–Muller code* $R(1,m)$ is the linear code with generator matrix G_m.

To see that G_m has the dimensions claimed, we work inductively. Clearly $G_1 \in M_{(1+1)\times 2^1}(\mathbf{F}_2)$, so the base case holds. Assume $G_m \in M_{(m+1)\times 2^m}$. Note G_{m+1} has twice as many columns as G_m, i.e. 2^{m+1}; and it has one extra row (the bottom one), so it has $m+2 = (m+1)+1$ rows.

We deduce immediately that $R(1,m)$ is a code of message length $k = m+1$ and block length $n = 2^m$.

Remark. One can prove that the ith column of G_m is a 1 followed by the 'inverse' binary representation of $i-1$; so if

$$i - 1 = a_0 + a_1 2 + a_2 2^2 + \cdots + a_{m-1}2^{m-1},$$

the ith column is

$$\begin{pmatrix} 1 \\ a_0 \\ a_1 \\ \vdots \\ a_{m-1} \end{pmatrix}.$$

The following table lists all the elements of $R(1,m)$ for small m:

$R(1,1)$	{00,	01,	10,	11}				
$R(1,2)$	{0000,	0101,	1010,	1111,	0011,	0110,	1001,	1100}
$R(1,3)$	{00000000,	01010101,	10101010,	11111111,	00110011,	01100110,	10011001,	11001100,
	00001111,	01011010,	10100101,	11110000,	00111100,	01101001,	10010110,	11000011}

Note that every codeword in $R(1,1)$ (apart from $\mathbf{0}$ and $\mathbf{1}$) has weight $1 = 2^0$; every codeword in $R(1,2)$ (apart from $\mathbf{0}$ and $\mathbf{1}$) has weight $2 = 2^1$; and every codeword in $R(1,3)$ (apart from $\mathbf{0}$ and $\mathbf{1}$) has weight $4 = 2^2$.

In these examples, all the codewords (except for $\mathbf{0}$ and $\mathbf{1}$) have weight $w(\mathbf{c}) = 2^{m-1}$, that is, they have exactly 2^{m-1} 1's and 2^{m-1} 0's. This continues.

Proposition 5.5. *In $R(1,m)$, every codeword (apart from $\mathbf{0}$ and $\mathbf{1}$) has weight 2^{m-1}.*

To get a hint at the proof, look closely again at the table above. All the words in $R(1,3)$ are obtained by either repeating a word in $R(1,2)$ twice (the first row of $R(1,3)$), or taking such a word and adding 00001111 (the second row of $R(1,3)$). To prove the proposition, we prove that this pattern also continues.

Proof. We use induction. The result is true for $R(1,1)$ (and $R(1,2)$ and $R(1,3)$) by the examples above. Suppose it is true for $R(1,m-1)$.

We have a specific basis $\{\mathbf{e}_1, ..., \mathbf{e}_m\}$ of $R(1,m-1)$, given by the rows of G_{m-1}. The basis of $R(1,m)$ is then

$$\{\mathbf{e}_1\mathbf{e}_1,\ \mathbf{e}_2\mathbf{e}_2,\ ..., \mathbf{e}_m\mathbf{e}_m,\ 0\cdots01\cdots1\},$$

where for any $\mathbf{c} = c_1 \cdots c_{2^{m-1}}$, by $\mathbf{cc}$ we mean the word $c_1 \cdots c_{2^{m-1}} c_1 \cdots c_{2^{m-1}}$ of length 2^m where we just repeat $\mathbf{c}$.

This means that any codeword $\mathbf{d} \in R(1,m)$ not equal to $\mathbf{0}$ or $\mathbf{1}$ has one of two forms; either:

- $\mathbf{d}$ is in the span of $\{\mathbf{e}_1\mathbf{e}_1, ..., \mathbf{e}_m\mathbf{e}_m\}$, in which case it has the form $\mathbf{cc}$ for some $\mathbf{c} \in R(1,m)$. Since $\mathbf{c}$ has weight 2^{m-2} by the induction hypothesis, we deduce $\mathbf{cc}$ has weight 2^{m-1}.

- Or, $\mathbf{d}$ is *not* in this span, in which case it has $0\cdots01\cdots1$ in its expansion. Such $\mathbf{d}$ are of the form $\mathbf{d} = \mathbf{cc} + 0\cdots01\cdots1$, for $\mathbf{c} \in R(1,m-1)$. For such a $\mathbf{c}$, let $\mathbf{c}^- = \mathbf{c} + 1\cdots1 \in R(1,m)$. Note $\mathbf{c}^-$ has weight 2^{m-2} as well (it has 1's and 0's exactly opposite to $\mathbf{c}$). So $\mathbf{d} = \mathbf{cc}^-$ has weight $2^{m-2} + 2^{m-2} = 2^{m-1}$, as required.

Thus any codeword (apart from $\mathbf{0}$ and $\mathbf{1}$) in $R(1,m)$ has weight 2^{m-1}, completing the induction step. $\square$

Corollary 5.6. *The minimum distance of the Reed–Muller code is*

$$d[R(1,m)] = 2^{m-1}.$$

Thus under nearest-neighbour decoding it is $(2^{m-2}-1)$*-error-correcting.*

Proof. Apart from the vectors $\mathbf{0}$ and $\mathbf{1}$, every codeword $\mathbf{c}$ in $R(1,m)$ has weight $w(\mathbf{c}) = 2^{m-1}$, which is thus the minimum weight. But the minimum weight is equal to the minimum distance by Lemma 4.7. □

5.3. (∗) Error-correction for Reed–Muller codes

In $R(1,m)$ the length n is growing exponentially, whilst k grows only linearly, so the rate decreases rapidly. Moreover, the syndrome lookup table has a doubly exponentially growing number of entries, namely $O(2^{2^m})$ entries! Even for $m = 5$ this is 67,108,864, and searching line-by-line through a table of this size is not of practical use in error-correction.

There is, however, a much better error-correction algorithm for Reed–Muller codes, exploiting the following evident structure in G_m:

Lemma 5.7. *In* G_m*:*

- *The top row is all* 1*s.*
- *The 2nd row is* $0101\cdots01$*, alternating* 0*s and* 1*s.*
- *The 3rd row is* $00110011\cdots0011$*, with two* 0*s followed by two* 1*s.*
- $\cdots$
- *The* j*th row, for* $2 \leqslant j \leqslant m+1$*, repeats* 2^{j-2} 0*s, followed by* 2^{j-2} 1*s.*

For clarity, we will illustrate this error-correction algorithm with $m = 3$. Then $2^{m-2} - 1 = 1$, so $R(1,m)$ should be 1-error-correcting.

Algorithm 5.8 (Reed–Muller/majority-logic decoding)**.** Let $\mathbf{w} = (w_1, w_2, w_3, w_4)$ be a sourceword, and $\mathbf{c} = (c_1, c_2, \ldots, c_8)$ be the corresponding codeword: that is,

$$\mathbf{c} = (w_1, w_2, w_3, w_4)\begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{pmatrix}.$$

We send this over a noisy channel, and receive $\mathbf{x}$. We want to error-correct this to a codeword $\mathbf{c}'$ (which we hope is $\mathbf{c}$!) and then decode this to a sourceword $\mathbf{w}' = (w_1', w_2', w_3', w_4')$ (which we hope is $\mathbf{w}$).

There is a *lot* of structure here. For example, the first two columns sum to

$$\begin{pmatrix}1\\0\\0\\0\end{pmatrix} + \begin{pmatrix}1\\1\\0\\0\end{pmatrix} = \begin{pmatrix}0\\1\\0\\0\end{pmatrix},$$

so

$$c_1 + c_2 = w_2.$$

Similarly, the 3&4 locums, the 5&6 columns, and the 7&8 columns sum to give $(0,1,0,0)^t$, so we have

$$\begin{aligned} w_2 &= c_3 + c_4 \\ &= c_5 + c_6 \\ &= c_7 + c_8. \end{aligned}$$

If $\mathbf{c}$ is sent over a *perfect, noiseless* channel, and $\mathbf{x} = (x_1, \dots, x_8)$ is received, then $\mathbf{x} = \mathbf{c}$ and hence

$$w_2 = x_1 + x_2 = x_3 + x_4 + x_5 + x_6.$$

If $\mathbf{c}$ is sent over a noisy channel, $\mathbf{x}$ is received and there is $\leqslant 1$ error, then:

At least two of the values $x_1 + x_2, x_3 + x_4, x_5 + x_6$ are still equal to w_2!

In other words:

Step 1: Determine w_2'.

We should decode $\mathbf{x}$ to $(-, w_2', -, -)$, where w_2' is the most common value in the triple $x_1 + x_2,\ x_3 + x_4,\ x_5 + x_6$.

By the above discussion, we will have $w_2' = w_2$ if there is at most one error.

We can do something similar for w_3 and w_4, since

$$\begin{aligned} w_3 &= c_1 + c_3 = c_2 + c_4 = c_5 + c_7 = c_6 + c_8, \qquad \text{and} \\ w_4 &= c_1 + c_5 = c_2 + c_6 = c_3 + c_7 = c_4 + c_8. \end{aligned}$$

In particular:

Step 2: Determine $w_3', ..., w_{m+1}'$.

We should decode $\mathbf{x}$ to $(-, w_2', w_3', w_4')$, where

$$\begin{aligned} w_3' &= \text{the most common value in } \{x_1 + x_3,\ x_2 + x_4,\ x_5 + x_7\}, \\ w_4' &= \text{the most common value in } \{x_1 + x_5,\ x_2 + x_6,\ x_3 + x_7\}. \end{aligned}$$

Again, we will have $(-, w_2', w_3', w_4') = (-, w_2, w_3, w_4)$ if there is at most one error.

It remains to find w_1, which must be handled differently. Let $\mathbf{r}_i$ be the ith row of G_3. Note that

$$\begin{aligned} \mathbf{c} + w_2\mathbf{r}_2 + w_3\mathbf{r}_3 + w_4\mathbf{r}_4 &= \Big(w_1\mathbf{r}_1 + w_2\mathbf{r}_2 + w_3\mathbf{r}_3 + w_4\mathbf{r}_4\Big) + w_2\mathbf{r}_2 + w_3\mathbf{r}_3 + w_4\mathbf{r}_4 \\ &= w_1\mathbf{r}_1 + 2\Big(w_2\mathbf{r}_2 + w_3\mathbf{r}_3 + w_4\mathbf{r}_4\Big) \\ &= w_1\mathbf{r}_1 \\ &= (w_1, w_1, \dots, w_1), \end{aligned}$$

so if we have correctly decoded w_2, w_3 and w_4, then w_1 is likely to be the most-occurring digit in

$$\mathbf{x} + w_2\mathbf{r}_2 + w_3\mathbf{r}_3 + w_4\mathbf{r}_4.$$

This motivates:

Step 3: Determine w_1'.

We should decode $\mathbf{x}$ *to* (w_1', w_2', w_3', w_4')*, where*

$$w_1' = \text{most common digit in } \mathbf{x} + w_2'\mathbf{r}_2 + w_3'\mathbf{r}_3 + w_4' + \mathbf{r}_4. \qquad (5.1)$$

If there is at most one error, then w_2', w_3' and w_4' have all been chosen correctly, whence at least 3 of the digits in (5.1) are equal to w_1, so that w_1' is also chosen correctly. In particular, if there is at most one error, then this algorithm correctly decodes $\mathbf{x}$ as $\mathbf{w}$. In particular:

For $R(1,3)$, the Reed–Muller decoding scheme is 1-error correcting.

Remark. For general m, from the recursive definition of G_m, we see that

$$\begin{aligned} w_2 &= c_1 + c_2 = c_3 + c_4 = \cdots = c_{2^m - 1} + c_{2^m}, \\ w_3 &= c_1 + c_3 = c_2 + c_4 = \cdots = c_{2^m - 2} + c_{2^m}, \\ &\vdots \\ w_{m+1} &= c_1 + c_{1+2^{m-1}} = c_2 + c_{2+2^{m-1}} = \cdots = c_{2^{m-1}} + c_{2^m}. \end{aligned}$$

So to decode $w_2, ..., w_{m+1}$, we can choose the majority from a collection of $2^{m-1} - 1$ equations; and this will always give the correct answer if there are at most $2^{m-2} - 1$ errors. In particular:

Reed–Muller decoding is $(2^{m-2} - 1)$-error-correcting.

Here, though, is a curious thing. This is *not* one of the decoding schemes we've seen before.

Proposition 5.9. *Reed–Muller decoding is not the same as nearest-neighbour/maximum-likelihood decoding.*

Proof. The proof of this Proposition is a guided exercise.

Consider the Reed–Muller code $R(1, 4)$. Suppose we receive the word

$$\mathbf{x} = (1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 1, 1, 1).$$

(1) Prove that the Reed–Muller scheme decodes $\mathbf{x}$ to the codeword $\mathbf{0}$.

(2) Prove that
$$\mathbf{r}_5 = (0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1) \in C$$
is closer to $\mathbf{x}$ than $\mathbf{0}$.

(3) Deduce that Reed–Muller and nearest-neighbour decode $\mathbf{x}$ to different codewords.

□

Remark. In particular, if C is a memoryless Reed–Muller code, then Reed–Muller decoding is not the 'best possible' (since in this case, the best possible is ideal-observer = maximum-likelihood = nearest-neighbour).

However, this is not a problem. In the example above, if we receive $\mathbf{x}$, then there have been at least 5 errors, whilst $R(1, m)$ is 3-error correcting. The difference between Reed–Muller decoding and nearest-neighbour only arises when there have been at least 2^{m-2} errors. In particular, there are a vanishingly small number of cases where Reed–Muller does not give the best possible decoding, and this is *more* than made up by the hugely increased efficiency of Reed–Muller decoding vs. nearest-neighbour for large n.

Exercise. For which m is the Reed–Muller $R(1, m)$ code perfect?

(Solution below[1]).

Exercise. We saw that under Reed–Muller decoding, the code $R(1, m)$ is (at least) $(2^{m-2} - 1)$-error-correcting. Can it do any better than this? How many errors can it *detect*? (Note that Proposition 5.9 tells us that we *cannot* just quote Theorem 3.11 here, as that theorem applies only for *nearest-neighbour* decoding).

(Solution below[2]).

[1]Solution: Any perfect code has odd minimal distance. Since the minimal distance of $R(1, m)$ is 2^{m-1}, it cannot be perfect for $m \geqslant 2$. For $m = 1$, we get the trivial code $C = V_2$, which is perfect (but totally uninteresting).

[2]You cannot do better than this: as with nearest-neighbour, it is *exactly* $(2^{m-2} - 1)$-error-correcting. For example, with $R(1, 3)$, suppose we try and send $\mathbf{w} = \mathbf{0}$, and receive $(1, 0, 1, 0, 0, 0, 0, 0)$, with 2 errors. Then we decode $w_2' = \text{mode}\{1, 1, 0\} = 1$, which is wrong. This is an example where 2 errors cannot be corrected.

In the form described above, Reed–Muller decoding for $m = 3$ is exactly 2-error detecting. (Contrast this with nearest-neighbour, which is 3-error-detecting). We will detect an error if any of the sets

$$\{x_1 + x_2, x_3 + x_4, x_5 + x_6\}$$
$$\{x_1 + x_3, x_2 + x_4, x_5 + x_7\}$$
$$\{x_1 + x_5, x_2 + x_6, x_3 + x_7\}$$

have different entries (e.g. if one of them is, say, $\{0, 0, 1\}$ rather than $\{0, 0, 0\}$). If there are at most 2 errors, then it is impossible that these will all be incorrect but still equal. However, if there are errors in x_2, x_3 and x_5, then we will not detect this: for example, if we send $\mathbf{w} = (0, 0, 0, 0)$, but receive $(0, 1, 1, 0, 1, 0, 0, 0)$, then the three sets above would be $\{1, 1, 1\}$, $\{1, 1, 1\}$, $\{1, 1, 1\}$.

We could improve the error-detecting properties of Reed–Muller decoding by also considering the sum $x_7 + x_8$ when we compute w_2. For all four elements of $\{x_1 + x_2, x_3 + x_4, x_5 + x_6, x_7 + x_8\}$ to be the same and *wrong*, there need to have been at least 4 errors; so this improved version can always detect up to 3 errors, making this as good

($*$) **Remark.** This remark is highly non-examinable.

The Reed–Muller codes we present here are *first-order* Reed–Muller codes. There are higher-order Reed–Muller codes: one can define a code $R(r, m)$ for any $r \leqslant m$, generalising $R(1, m)$ above. These are defined as follows: let

$$\mathbf{v}_0 = (1, ..., 1) = [\text{first row of } G_m],$$

and

$$\mathbf{v}_1 = [\text{second row of } G_m], \qquad \dots, \qquad \mathbf{v}_m = [(m+1)\text{st row of } G_m].$$

Then:

- $R(2, m)$ has generator matrix whose rows are

  $$\mathbf{v}_0, \mathbf{v}_1, \dots, \mathbf{v}_m \qquad \text{and} \qquad \{\mathbf{v}_i \wedge \mathbf{v}_j : 1 \leqslant i < j \leqslant m\}.$$

  (Here the notation $\mathbf{x} \wedge \mathbf{y}$ means $(x_1y_1, ..., x_ny_n)$. For example, $(1, 1, 0) \wedge (1, 0, 1) = (1, 0, 0)$).
- $R(3, m)$ has generator matrix whose rows are

  $$\mathbf{v}_0, \mathbf{v}_1, \dots, \mathbf{v}_m, \{\mathbf{v}_i \wedge \mathbf{v}_j : 1 \leqslant i < j \leqslant m\} \qquad \text{and} \qquad \{v_i \wedge v_j \wedge v_k : 1 \leqslant i < j < k \leqslant m\}.$$

- $R(r, m)$ has generator matrix whose rows are $\mathbf{v}_0$ and all possible wedge products of up to r of the $\mathbf{v}_1, ..., \mathbf{v}_m$.

The minimum distance of $R(r, m)$ is 2^{m-r}.

Completing this picture, we also define the '0th order Reed–Muller code' $R(0, m)$ to have generator matrix just $\mathbf{v}_0$. This is just the code $\{0 \cdots 0, 1 \cdots 1\}$, i.e. it is just the 2^m-repetition code.

One can recover 'extended' Hamming codes via higher-order Reed–Muller codes: the code $R(m-2, m)$ is an 'extension' of the Hamming code $\text{Ham}(m)$. There is a nice table on the Wikipedia page for Reed–Muller codes, explaining that Reed–Muller codes $R(r, m)$ incorporate many other families of good codes.

as nearest-neighbour.

Part II

Cryptography

A *plaintext message* is a message which you want to send, e.g. `what should we eat for dinner?`. The art of cryptography aims to disguise a plaintext message as *ciphertext*, making it unreadable to any but the intended recipient.

There are multiple aims of cryptography. Suppose Alice and Bob want to communicate, and that a malicious attacker, Eve, is lurking in the background. Then Alice and Bob should ensure the following.

- **Secrecy**. They want to ensure that Eve cannot read their messages.
- **Integrity**. They want to ensure that Eve cannot *alter* their messages.
- **Authenticity**. Bob wants to ensure that the messages he's receiving really are from Alice, and vice versa.
- **Non-repudiation**. Bob wants to be able to prove to third parties that the messages they've received really did come from Alice (if, for example, Alice decides to deny ever sending them!)

Before starting this part of the course, let me declare a caveat. Several of these concepts are harder to describe theoretically than to discover through examples. I have tried to include a large number of worked examples in the second half of the course, and encourage students to generate more examples of your own: there are resources for doing this listed in the text below.

Chapter 6

Ciphers

In a *cipher*, a message is encrypted 'by applying a bijection to it.' In other words, every letter of a message is replaced by another in a systematic way. To be a useful cipher:

(1) the recipient must be able to apply the inverse bijection to decrypt the message.

(2) a third-party should not be able to decrypt the message.

In this section, we describe some ciphers of varying strengths.

Before starting this chapter, let me declare a caveat. Many of these concepts are harder to describe than to discover through examples. I have tried to include a large number of worked examples in this chapter, and encourage students to generate more examples of your own: there are resources for doing this listed in the text below.

6.1. Caesar and affine ciphers

6.1.1. Caesar ciphers. One of the most famous, and easy to grasp, ciphers to ever exist is the *Caesar cipher*, named after Julius Caesar. He disguised his messages by a simple translation of the alphabet, shifting each letter by a fixed amount D, and wrapping around back to the start. For example, if you shift by $\kappa = 3$ letters, then in a plaintext message you replace:

$$\texttt{a} \text{ with } \texttt{d}, \quad \texttt{b} \text{ with } \texttt{e}, \quad \texttt{c} \text{ with } \texttt{f}, \quad \dots, \quad \texttt{z} \text{ with } \texttt{c}.$$

In other words, the plain letters below are mapped to their cipher equivalents:

plain	a	b	c	⋯	w	x	y	z
cipher	D	E	F	⋯	Z	A	B	C

In this case, the message `Caesar cipher` becomes `FDHVDU FLSKHU`.

This is very easy to crack. There are only 26 possible values of κ, and one of these – the case $\kappa = 0$ – doesn't even change the message! So even without a computer, you could try all the possibilities very quickly. It provides essentially no information security.

6.1.2. Modular arithmetic. Caesar ciphers were secretly using a basic number theory device: *modular arithmetic*, in particular modulo 26. This relies on the following convention:

Convention 6.1. For the rest of this chapter, we will identify the alphabet with the set $\{0, 1, 2, ..., 25\}$,, via the following table:

a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25

Via the rules of modular arithmetic, which we recap below, this allows us to add letters together, e.g.

$$\mathtt{c} + \mathtt{f} = 2 + 5 = 7 = \mathtt{h}.$$

If we get a result beyond 26, we ‘wrap around’ back to 0 and start again, so the numbers go ...23, 24, 25, 0, 1, 2, ... This is working modulo 26. For example,

$$\mathtt{p} + \mathtt{t} = 15 + 19 = 34 \sim 8 = \mathtt{i}.$$

To describe more complicated ciphers, we manipulate the integers $\{0, 1, ..., 25\}$ and thus get permutations of the alphabet. For this we must use modular arithmetic, which precisely describes this ‘wrapping around’ process, and which we recall now. In particular, the set $\{0, 1, 2, ..., 25\}$ is nothing other than $\mathbf{Z}/26\mathbf{Z}$, the integers modulo 26.

Definition. Let $n > 1$ be an integer. For any integers a and b, we say a is *congruent to b modulo n*, written

$$a \equiv b \pmod{n},$$

if $a - b$ is divisible by n.

Lemma. *Every integer a is congruent to a unique element of $\{0, 1, \dots, n-1\}$, the remainder after dividing a by n.*

Hence $\pmod{n}$ defines an equivalence class on the integers. We denote the set of equivalence classes by $\mathbf{Z}/n\mathbf{Z}$. Note that $\mathbf{Z}/n\mathbf{Z}$ is a *ring*, i.e. we can do addition, subtraction and multiplication on its elements:

$$\begin{aligned}[a \pmod{n}] + [b \pmod{n}] &= (a + b) \pmod{n},\\ [a \pmod{n}] - [b \pmod{n}] &= (a - b) \pmod{n},\\ [a \pmod{n}] \times [b \pmod{n}] &= (a \times b) \pmod{n}.\end{aligned}$$

Usually we only write ‘$\pmod{n}$’ at the *end* of the equation.

Example. Let’s work modulo 12. Then

$$\begin{aligned}&3 + 7 = 10 \pmod{12},\\ &5 + 7 = 12 = 0 \pmod{12},\\ &8 + 9 = 17 = 5 \pmod{12},\\ &3 \times 9 = 27 = 3 \pmod{12},\\ &1 - 3 = -2 = 10 \pmod{12}.\end{aligned}$$

However, **in general we cannot do division** in the usual way. For example, what value should we give to

$$5 \div 4 \pmod 6?$$

If we call this x, then we find

$$5/4 = x \pmod 6 \qquad \Rightarrow \qquad 4x = 5 \pmod 6$$

(because we do multiplication in the usual way). But we can just check all the values of $x \pmod 6$:

$$\begin{aligned} &4 \times 0 = 0 \pmod 6, \quad 4 \times 1 = 4 \pmod 6, \quad 4 \times 2 = 2 \pmod 6, \\ &4 \times 3 = 0 \pmod 6, \quad 4 \times 4 = 4 \pmod 6, \quad 4 \times 5 = 2 \pmod 6. \end{aligned}$$

So there is *no solution* to $4x = 5 \pmod 6$, so '$\div 4$' isn't a valid operation on $\mathbf{Z}/6\mathbf{Z}$.

The problem here is that 4 and 6 have a common factor bigger than 1 (namely, 2).

Proposition 6.2. *Let $n > 1$ and let a be an integer. Then there exists $b \in \mathbf{Z}$ such that $ab \equiv 1 \pmod n$ if and only if a and n are coprime (i.e. a and n have no common factors greater than 1).*

Definition. If there exists b with $ab \equiv 1 \pmod n$, we say a is *invertible modulo n*, and we say $b = a^{-1} \pmod n$ is the *inverse of a modulo n*.

If a is invertible modulo n, with inverse b, then 'division by a' is the same as multiplication by b. So we can do division by a in $\mathbf{Z}/n\mathbf{Z}$ if and only if a is coprime to n.

Example. If $n = 6$, and $a = 2$, then a and n are not coprime. We can check directly that a is not invertible, by computing all the possibilities:

$$\begin{aligned} 2 \times 0 &\equiv 0 \pmod 6, \\ 2 \times 1 &\equiv 2 \pmod 6, \\ 2 \times 2 &\equiv 4 \pmod 6, \\ 2 \times 3 &\equiv 0 \pmod 6, \\ 2 \times 4 &\equiv 2 \pmod 6, \\ 2 \times 5 &\equiv 4 \pmod 6. \end{aligned}$$

None of these are 1, so there is no b with $2b \equiv 1 \pmod 6$.

Remark. If $n = p$ is a prime number, then a and p have a common factor if and only if p divides a. Hence division by $1, 2, \dots, p-1$ is possible in $\mathbf{Z}/p\mathbf{Z}$. As all the non-zero elements have inverses, we see that $\mathbf{Z}/p\mathbf{Z}$ is a *field* (the finite field of p elements). This generalises the example of $\mathbf{Z}/2\mathbf{Z}$ we saw in Chapter 4. It will also be extremely relevant in the later chapters.

Particularly relevant to us is the case $n = 26$ (the number of letters in the alphabet!). The following table is a list of all the elements, with their inverses (where they exist):

element	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
inverse	✗	1	✗	9	✗	21	✗	15	✗	3	✗	19	✗	✗	✗	7	✗	23	✗	11	✗	5	✗	17	✗	25

Moving forward with the course, I will assume familiarity with all of the above facts. If you would like to read more, then please consult Chapter 2 of William Stein's book *Elementary Number Theory: primes, congruences and secrets: a computational approach.*

Then the Caesar cipher works as follows:

Encryption (Caesar cipher)**.** (1) Fix $\kappa \in \{0, 1, \dots, 25\}$ (the 'private key').

(2) Take a letter (or equivalently, by the table, a number x in $\{0, 1, ..., 25\}$).

(3) Encode this as the letter corresponding to $x + \kappa \pmod{26}$.

Decryption (Caesar cipher)**.** Given the private key κ, and an encrypted letter corresponding to $y \in \{0, 1, ..., 25\}$, decrypt y as the letter corresponding to $y - \kappa \pmod{26}$.

To decrypt you need the private key κ; but as mentioned above, this this is easy to crack.

6.1.3. The affine cipher.

It is not hard to improve on the Caesar cipher. In the *affine cipher*, in step (3) we replace the function $f(x) = x + \kappa$ with a more general linear function

$$\begin{aligned} f : \mathbf{Z}/26\mathbf{Z} &\longrightarrow \mathbf{Z}/26\mathbf{Z} \\ x &\longmapsto \lambda x + \kappa, \end{aligned}$$

for some values of $\lambda, \kappa \in \{0, 1, \dots, 25\}$.

Encryption (Affine cipher)**.** (1) Fix $\lambda, \kappa \in \{0, 1, \dots, 25\}$ (the 'private key').

(2) Take a letter (or equivalently, by the table, a number x in $\{0, 1, ..., 25\}$).

(3) Encode this as the letter corresponding to $\lambda x + \kappa \pmod{26}$.

Example 6.3. If we took $\lambda = 7, \kappa = 3$, then we get the following encryption table:

plain	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z
x	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
f(x)	3	10	17	24	5	12	19	0	7	14	21	2	9	16	23	4	11	18	25	6	13	20	1	8	15	22
cipher	D	K	R	Y	F	M	T	A	H	O	V	C	J	Q	X	E	L	S	Z	G	N	U	B	I	P	W

Thus the message

```
the affine cipher is better than the caesar cipher
```

becomes

```
GAF DMMHQF RHEAFS HZ KFGGFS GADQ GAF RDFZDS RHEAFS.
```
(6.1)

Decryption for the affine cipher. How do we decrypt? We must recover x from $y = \lambda x + \kappa \pmod{26}$. If λ is invertible modulo 26, that is if $\lambda^{-1} \pmod{26}$ exists, we can simply solve:

$$\begin{aligned} x = \lambda^{-1}(y-\kappa) &= \lambda^{-1}y - \lambda^{-1}\kappa \pmod{26} \\ &= \mu y + \nu \pmod{26}, \end{aligned}$$

where $\mu = \lambda^{-1} \pmod{26}$ and $\nu = -\lambda^{-1}\kappa \pmod{26}$.

What if λ is not invertible modulo 26? Then the 'cipher' is not a bijection, and hence does not give a valid cipher at all, as decryption becomes impossible. For example, suppose $\lambda = 13$ (which is not invertible modulo 26) and $\kappa = 3$. Then the encryption table becomes

plain	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z
x	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
$f(x)$	3	16	3	16	3	16	3	16	3	16	3	16	3	16	3	16	3	16	3	16	3	16	3	16	3	16
cipher	D	Q	D	Q	D	Q	D	Q	D	Q	D	Q	D	Q	D	Q	D	Q	D	Q	D	Q	D	Q	D	Q

This 'cipher' encrypts:

$$\texttt{i love you} \longmapsto \texttt{D QDQD DDD},$$
$$\texttt{i hate you} \longmapsto \texttt{D QDQD DDD}.$$

...oops! This is impossible to decrypt, even if we know the private key (λ, κ), making it useless for cryptography. Thus if λ is not invertible, then the affine cipher is not a valid cipher at all.

Decryption (Affine cipher)**.** Given the private key (λ, κ) with λ invertible modulo 26:

(1) Compute $\mu := \lambda^{-1} \pmod{26}$ and $\nu = -\mu\kappa$.

(2) Given an encrypted letter corresponding to $y \in \{0, 1, ..., 25\}$, decrypt y as the letter corresponding to $\mu y + \nu \pmod{26}$.

Remark. Note that the decryption map is another affine cipher!

Breaking the affine cipher. It should already be clear that the affine cipher is better than the Caesar cipher; it looks a lot more random. Since there are 12 invertible λ's modulo 26, and 26 possible κ's, there are $12 \times 26 = 312$ possibilities for the private key (λ, κ). A brute force attack is thus a tall order to do by hand (though of course a computer would make short work of it).

However, the affine cipher is still easily broken, even without brute force. If we decrypt just one letter, it yields a linear equation in the secret parameters λ and κ, and another linear equation in the parameters μ and ν. For example, suppose we know that our cipher encrypts `c` as `R`. Then we get the equation

$$\texttt{c} = 2 \xrightarrow{x \mapsto \lambda x + \kappa} \texttt{R} = 17 \qquad \Rightarrow \qquad 2\lambda + \kappa = 17 \pmod{26}$$

for the encryption key (λ, κ), and the equation

$$\texttt{R} = 17 \xrightarrow{y \mapsto \mu y + \nu} \texttt{c} = 2 \qquad \Rightarrow \qquad 17\mu + \nu = 2 \pmod{26}$$

for the decryption key (μ, ν).

Now, this equation is likely to have multiple solutions. But if we can guess just a few letters, then we get a collection of linear equations involving the unknowns (λ, κ) or (μ, ν). The more letters we guess, the better the chance the system has a unique solution; and we can solve this system of simultaneous equations to compute (our guess of!) the secret key. This is slightly more subtle than solving equations over, say, $\mathbf{R}$, as not everything has an inverse modulo 26, which means not every system has a unique solution; we'll see this by example.

Once we have a guess at the secret key, it can then be checked by using it to attempt to decrypt the message.

To decrypt some letters, we can lean on our knowledge of the underlying language.

Example 6.4. Consider the message (6.1) above, and suppose we know the decrypted message is in English. The word 'the' is very common in the English language, so we might guess it appears in our cipher text. In our message, the three-letter word `GAF` appears twice, so it's reasonable to guess this is 'the'. There are now two approaches to using our guess:

Approach 1: Find the encryption key.
If our guess is true, we would encrypt `t` as `G`, `h` as `A` and `e` as `F`, yielding the three equations

$$\begin{aligned} \texttt{t} = 19 \mapsto \texttt{G} = 6 \qquad &\Rightarrow \qquad 19\lambda + \kappa = 6 \,(\mathrm{mod}\, 26), \\ \texttt{h} = 7 \mapsto \texttt{A} = 0 \qquad &\Rightarrow \qquad 7\lambda + \kappa = 0 \,(\mathrm{mod}\, 26), \\ \texttt{e} = 4 \mapsto \texttt{F} = 5 \qquad &\Rightarrow \qquad 4\lambda + \kappa = 5 \,(\mathrm{mod}\, 26). \end{aligned}$$

In this case, taking the third equation from the first equation yields the equation

$$15\lambda = 1 \,(\mathrm{mod}\, 26).$$

Now, this has a **unique** solution, because:

- 15 is invertible modulo 26 (as it has no common factors with 26),
- so we can apply $15^{-1}\,(\mathrm{mod}\, 26)$ to both sides,
- yielding $\lambda = 15^{-1}\,(\mathrm{mod}\, 26)$.

Of course we must now find 15^{-1} as an integer in $\{0, 1, 2, ..., 25\}$. A simple (if unwieldy) way to do this is to keep adding 15 to itself until we get 1; for example,

$$\begin{aligned} &1 \times 15 = 15 \,(\mathrm{mod}\, 26), \\ &2 \times 15 = 15 + 15 = 30 = 4 \,(\mathrm{mod}\, 26), \\ &3 \times 15 = 4 + 15 = 19 \,(\mathrm{mod}\, 26), \\ &4 \times 15 = 19 + 15 = 34 = 8 \,(\mathrm{mod}\, 26), \\ &5 \times 15 = 8 + 15 = 23 \,(\mathrm{mod}\, 26), \\ &6 \times 15 = 23 + 15 = 38 = 12 \,(\mathrm{mod}\, 26), \\ &7 \times 15 = 12 + 15 = 27 = 1 \,(\mathrm{mod}\, 26), \end{aligned}$$

so the inverse of $15\,(\mathrm{mod}\, 26)$ is $7\,(\mathrm{mod}\, 26)$, and therefore

$$\lambda = 7 \,(\mathrm{mod}\, 26).$$

Plugging this back into our equations yields three equations for κ, namely

$$\begin{aligned} 19\times 7+\kappa &= 6 \pmod{26},\\ 7\times 7+\kappa &= 0 \pmod{26},\\ 4\times 7+\kappa &= 5 \pmod{26}. \end{aligned}$$

Each yields the answer $\kappa = 3 \pmod{26}$. So finally we guess the encryption mapping is

$$x \mapsto 7x + 3 \pmod{26},$$

which we can use to try to decrypt the message.

Approach 2: Find the decryption key.
Alternatively, we would decrypt `G` as `t`, `A` as `h`, and `F` as `e` or in other words,

$$\begin{aligned} \texttt{G} = 6 \mapsto \texttt{t} = 19 \quad &\Rightarrow \quad 6\mu+\nu = 19 \pmod{26},\\ \texttt{A} = 0 \mapsto \texttt{h} = 7 \quad &\Rightarrow \quad 0\mu+\nu = 7 \pmod{26},\\ \texttt{F} = 5 \mapsto \texttt{e} = 4 \quad &\Rightarrow \quad 5\mu+\nu = 4 \pmod{26}. \end{aligned}$$

Equation (2) now tells us, immediately!, that $\nu = 7 \pmod{26}$. We now want to solve for μ. Plugging our value of ν into the two equations involving μ, we find

$$\begin{aligned} 6\mu+7 = 19 \pmod{26} \quad &\Rightarrow \quad 6\mu = 12 \pmod{26},\\ 5\mu+7 = 4 \pmod{26} \quad &\Rightarrow \quad 5\mu = 23 \pmod{26}. \end{aligned}$$

Here your intuition might need a jolt.

Warning: be wary of your intuition!. Looking at the first equation, you might reasonably conclude 'as $6\mu = 12 \pmod{26}$, we have $\mu = 2 \pmod{26}$'. Here you are doing an automatic calculation, honed over years of arithmetic: but we must examine the steps. Implicitly, this assertion comes down to:

- If $6\mu = 12 \pmod{26}$,
- **Then dividing both sides by 6,**
- we find $\mu = 2 \pmod{26}$.

But the bold step is nonsense. We cannot divide by 6 modulo 26, because 6 and 26 share a common factor, namely 2.

In fact, the equation $6\mu = 12 \pmod{26}$ has **two** possible solutions: we could have $\mu = 2 \pmod{26}$, *or* we could have $\mu = 15 \pmod{26}$, as $6\times 15 = 90 = 12 \pmod{26}$. (One may check these are the only possible solutions. Note that they differ by $13 = 26/2$, and that 2 was the GCD of 6 and 26; this is a feature of modular equations). Without more information we can't know which of 2 and 15 is the 'true' value of μ.

Luckily, we do have more information, though: namely, we have the second equation $5\mu = 23 \pmod{26}$. Here we **do** have a unique solution, because:

- 5 and 26 share no common factor, so 5 is invertible modulo 26;
- therefore applying $5^{-1} \pmod{26}$ to both sides,

- we find $\mu = 5^{-1} \times 23 \pmod{26}$.

Again we must find $5^{-1} \pmod{26}$. We could do the same as above, which will eventually yield the answer 21 (oof!). Or we could use a trick. We know that

$$5 \times 5 = 25 = -1 \pmod{26},$$

so

$$-5 \times 5 = 1 \pmod{26},$$

so

$$5^{-1} = -5 = 21 \pmod{26}.$$

Thus we find

$$\mu = 21 \times 23 = (-5) \times (-3) = 15 \pmod{26}.$$

Here I didn't fancy computing 21×23, so I simplified both terms using mod 26 (as $21 = -5 \pmod{26}$, and $23 = -3 \pmod{26}$) before I did the computation. These kinds of clever manipulations of modular arithmetic take time to come naturally, but the more experience you have with it, the easier and more intuitive it becomes.

Finally, then, our guess at the decryption mapping is

$$y \mapsto 15y + 7 \pmod{26}.$$

Equipped with this guess, we apply it to the rest of the ciphertext, and (if our guess was correct) we recover the original message.

Remark 6.5. There are other simple techniques to attack the affine cipher. If a one-letter word appears, it can only be decrypted to either `i` or `a`. There are also not many common 2-letter words:

`an`, `as`, `at`, `be`, `by`, `do`, `go`, `he`, `hi`, `if`, `in`, `is`, `it`, `me`, `my`, `no`, `of`, `oh`, `ok`, `on`, `or`, `ow`, `pa`, `so`, `to`, `up`, `us`, `we`.

A simple way to defend this line of attack, looking for common short words, is to remove all the spaces! The example above would become

`GAFDMMHQFRHEAFSHZKFGGFSGADQGAFRDFZDSRHEAFS`.

We can still look for repeated sequences, e.g. `GAF` appears twice (the word 'the'), as does `RHEAF` ('cipher'). The longer the ciphertext the more likely you are to see patterns of this kind. (In any case, the affine cipher suffers the same weakness as *all* monoalphabetic ciphers: it is very weak against frequency analysis, which we'll describe in the next section).

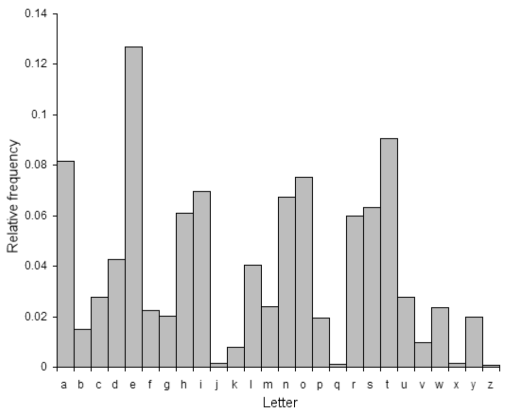
Exercise. The Caesar cipher used one of 26 secret keys κ and the encryption function $x \mapsto x + \kappa$, and was easy to break with brute force (e.g. by decrypting one letter) The affine cipher used one of 312 secret keys (λ, κ) and the encryption function $x \mapsto \lambda x + \kappa$, and this was stronger, requiring decryption of two letters.

It seems natural to consider a 'quadratic cipher', where the secret key is a triple (η, λ, κ) and the encryption function $x \mapsto \eta x^2 + \lambda x + \kappa$, which – by the same method – would require knowing *three* letters to decrypt.

Explain why this 'quadratic cipher' can never be decrypted.[1]

[1]Solution: the map $\mathbf{Z}/26\mathbf{Z} \to \mathbf{Z}/26\mathbf{Z}$ given by $x \mapsto \eta x^2 + \lambda x + \kappa$ is never a bijection.

6.2. Monoalphabetic ciphers and frequency analysis



Both the Caesar cipher and affine cipher are examples of *monoalphabetic ciphers*, where each specific letter of the alphabet with another *fixed* specific letter. In particular, any monoalphabetic cipher corresponds to a permutation of the alphabet.
Any such permutation defines an element of the symmetric group S_{26} on $\{0, 1, \cdots, 25\}$, a group of size $26! \approx 4 \times 10^{26}$. However, only 26 permutations come from Caesar ciphers, and 312 from affine ciphers, so we have been operating in a tiny part of this group.

What if we use an arbitrary permutation π of $\{0, 1, ..., 25\}$? That is, π is a bijection from $\{0, 1, ..., 25\}$ to itself.

Encryption (Monoalphabetic cipher)**.** Encrypt `a` as the letter corresponding to $\pi(0)$, encrypt `b` as the letter corresponding to $\pi(1)$, encrypt `c` as the letter corresponding to $\pi(2)$, and so on.

Decryption (Monoalphabetic cipher)**.** If a letter corresponds to the number $x \in \{0, 1, ..., 25\}$, decrypt it as the letter corresponding to $\pi^{-1}(x)$.

This is a cipher where we replace every instance of `a` with some other (arbitrary) letter, every instance of `b` with a (different, arbitrary) letter, every instance of `c` with another letter, and so on. This is why there are 26! possibilities: there are 26 choices of where to send `a`, then 25 to send `b`, then 24 to send `c`, and so on.

In this case, a brute force attack is impossible: there's no way we can evaluate all the 26! possibilities. Since the permutation is totally random, we cannot exploit mathematical structure as we did before. However, we can still easily attack this using *frequency analysis.*

Certain letters in the English language appear much more frequently than others; e.g. in decreasing order, the most common letters are `e`, `t`, `a`, `o`, `i`, `n`, `s`, `h`, `r`, `d`, and `l`. This is illustrated in the chart above right, which measures the relative frequency of each letter.

Similarly, certain pairs and triples of letters are much more likely to appear together than others, as in the following table:

digram frequencies				trigram frequencies			
th	3.21%	es	1.21%	the	2.00%	dth	0.35%
he	3.05%	ou	1.16%	and	0.93%	ent	0.34%
in	1.83%	to	1.12%	ing	0.74%	eth	0.32%
er	1.74%	at	1.09%	her	0.58%	for	0.32%
an	1.73%	en	1.07%	tha	0.47%	nth	0.31%
re	1.37%	on	1.07%	hat	0.44%	thi	0.30%
nd	1.28%	ea	1.06%	his	0.41%	she	0.30%
ed	1.28%	nt	1.05%	you	0.40%	was	0.29%
ha	1.23%	st	1.04%	ere	0.39%	hes	0.29%

Example. I took the script from the film *Harry Potter and the Philosopher's Stone*, which is roughly 114,000 characters long, and computed the following letter frequencies. Even though this is a film script, so it's peppered with wizard names and doesn't follow typical sentence structures, there is a still a strong convergence to the norm.

Letter	Freq. in HP	Rel. freq. in English	Letter	Freq. in HP	Rel. freq. in English
A	7.7%	8.2%	N	6.8%	6.7%
B	1.7%	1.5%	O	8.1%	7.5%
C	2.2%	2.8%	P	1.8%	1.9%
D	3.8%	4.3%	Q	0.1%	0.1%
E	10.9%	12.7%	R	7.7%	6.0%
F	2.0%	2.2%	S	7.0%	6.3%
G	2.6%	2.0%	T	8.2%	9.1%
H	5.8%	6.1%	U	2.8%	2.8%
I	6.5%	7.0%	V	0.9%	1.0%
J	0.2%	0.2%	W	2.0%	2.4%
K	1.0%	0.8%	X	0.2%	0.2%
L	4.6%	4.0%	Y	2.8%	2.0%
M	2.3%	2.4%	Z	0.1%	0.1%

Given some plaintext, `e` is likely to be the most common letter. For long enough plaintext, this is essentially guaranteed. If we apply *any* permutation of the alphabet, then this maps all the `e`'s to some fixed letter. If this letter is (say) `F`, then `F` will appear the most times in the ciphertext. In this way we can guess what `e` has been encrypted to. Repeating this for enough letters, you can decrypt large parts of the message.

Frequency analysis: a complete worked example. Here's a complete worked example of decryption via frequency analysis, generated using the online tools at

`https://www.dcode.fr/frequency-analysis`.

I took the first 100,000 characters of J.R.R. Tolkien's *The Fellowship of the Ring*, and encrypted it using a random monoalphabetic cipher. The corresponding ciphertext starts:

```
QWYIXCBLPBKBNOOLIDKFBNOYIJNIIKGIAYJEWNEWYQKGPJ
DWKCEPTBYAYPYBCNELIOWLDYPYRYIETFLCDEBLCEWJNTQL
EWNUNCETKFDUYALNPXNOILFLAYIAYEWYCYQNDXGAWENPMN
IJYSALEYXYIELIWKBBLEKIBLPBKQNDRYCTCLAWNIJRYCTU...
```

In the full ciphertext, the frequencies of the letters that appear, and the frequencies of the letters in the English language, are:

Letter	Frequency in cipher	Letter	Rel. freq. in English
Y	11.88%	e	12.7%
E	8.81%	t	9.1%
N	8.38%	a	8.2%
K	7.98%	o	7.5%
I	6.96%	i	7.0%
L	6.67%	n	6.7%
W	6.12%	s	6.3%
D	5.90%	h	6.1%
C	5.58%	r	6.0%
J	5.21%	d	4.3%
P	4.45%	l	4.0%
G	2.78%	c	2.8%
O	2.58%	u	2.8%
F	2.49%	m	2.4%
Q	2.48%	w	2.4%
X	2.24%	f	2.2%
T	2.17%	g	2.0%
B	2.14%	y	2.0%
A	1.61%	p	1.9%
U	1.36%	b	1.5%
M	0.98%	v	1.0%
R	0.96%	k	0.8%
S	0.08%	j	0.2%
V	0.08%	x	0.2%
Z	0.07%	q	0.1%
H	0.05%	z	0.1%

The most common digrams and trigrams appearing in the ciphertext are:

cipher digram frequencies				cipher trigram frequencies			
EW	3.12%	KG	1.19%	EWY	1.91%	WNE	0.36%
WY	2.79%	YD	1.08%	NIJ	1.28%	WLD	0.34%
LI	1.91%	EK	1.08%	LIO	0.80%	WYD	0.32%
NI	1.85%	YI	1.04%	YCY	0.52%	JEW	0.32%
IJ	1.74%	YN	1.03%	WYC	0.45%	LIE	0.31%
YC	1.68%	IE	1.01%	EWN	0.44%	IEW	0.31%
CY	1.36%	IO	1.00%	QND	0.41%	YDN	0.30%
YJ	1.21%	LE	0.99%	TKG	0.40%	RYC	0.29%
WN	1.20%	WL	0.98%	NPP	0.37%	FKC	0.28%

And, for convenience, here's the table of English frequencies again:

English digram frequencies				English trigram frequencies			
th	3.21%	es	1.21%	the	2.00%	dth	0.35%
he	3.05%	ou	1.16%	and	0.93%	ent	0.34%
in	1.83%	to	1.12%	ing	0.74%	eth	0.32%
er	1.74%	at	1.09%	her	0.58%	for	0.32%
an	1.73%	en	1.07%	tha	0.47%	nth	0.31%
re	1.37%	on	1.07%	hat	0.44%	thi	0.30%
nd	1.28%	ea	1.06%	his	0.41%	she	0.30%
ed	1.28%	nt	1.05%	you	0.40%	was	0.29%
ha	1.23%	st	1.04%	ere	0.39%	hes	0.29%

Given these tables, let's attempt to decrypt by hand.

- By far the most common letter is `Y`, which we should decrypt as `e`.
- By considering the digrams `EW` and `WY`, and the trigram `EWY`, it's clear that we should decode `E` as `t` and `W` as `h`.
- Both `LI` and `NI` end in the same letter appear very high in the digram frequencies, and comparing to English, they should correspond to `in` and `an`; so `I` decodes as `n`, and $\{\texttt{L},\texttt{N}\}$ decodes as $\{a, i\}$ (in some order).
- In the trigrams, the words $\{\texttt{NIJ},\texttt{LIO}\}$ will decode to $\{\texttt{and},\texttt{ing}\}$. To decided which is which, note that `J` occurs 5.21% of the time, whilst `O` only appears 2.58%; these must decode to `d` (4.3%) and `g` (2.0%). It's very likely, then, that `J` $\mapsto$ `d` and `O` $\mapsto$ `g`, and then we further deduce that `N` $\mapsto$ `a` and `L` $\mapsto$ `i`.

Having decrypted some of the letters, we see that our tables of single letters, double letter and triple letter frequencies in the cipher is now as follows, with unencrypted letters capital and bold:

letter frequencies		cipher digram frequencies				cipher trigram frequencies			
e	11.88%	th	3.12%	**KG**	1.19%	the	1.91%	hat	0.36%
t	8.81%	he	2.79%	e**D**	1.08%	and	1.28%	hiD	0.34%
a	8.38%	in	1.91%	t**K**	1.08%	ing	0.80%	heD	0.32%
K	7.98%	an	1.85%	en	1.04%	e**C**e	0.52%	dth	0.32%
n	6.96%	nd	1.74%	ea	1.03%	he**C**	0.45%	int	0.31%
i	6.67%	e**C**	1.68%	nt	1.01%	tha	0.44%	nth	0.31%
h	6.12%	**C**e	1.36%	n**O**	1.00%	**Q**a**D**	0.41%	eDa	0.30%
D	5.90%	e**J**	1.21%	it	0.99%	**TKG**	0.40%	ReC	0.29%
C	5.58%	ha	1.20%	hi	0.98%	aPP	0.37%	**FKC**	0.28%

We see that the dynamics has shifted: having only decrypted a handful of letters, we've obtained an enormous amount of information about the remaining ones. For example:

- Of the top 7 most frequent letters in the cipher, the only one we haven't assigned is `K` (7.98%), and in the 6 most common in English, we haven't decrypted anything to `o` (7.5%), so this is very likely. The next most common unassigned letters appear significantly less often in both lists.

  Further evidence that `K` $\mapsto$ `o` is the cipher digram `tK`, which essentially has to come from `to`.
- The digrams `YC` and `CY` are likely to be `er` and `re`, so `C` $\mapsto$ `r`. (This tallies with the high occurrence of `WYC` $\mapsto$ `her`).

- To find the encryption of `s`, we consider the popular English trigram `his` (0.41%), which we know encrypts as `WL-`. In the cipher, the only likely candidate is `WLD`, so we guess `D` $\mapsto$ `s`. To further support this, note that `WYD` (0.32%) would decrypt to `hes` (0.29%). Also, `D` appears 5.90%, whilst `s` appears 6.3%: so we can be confident in our guess.
- The unique occurrence of `Q` in the cipher digrams/trigrams is in `QND`, which decrypts to `-as`. By looking at the English ones, we see that the unique occurrence of `w` is in `was`, so we decrypt `Q` $\mapsto$ `w`.
- We've now decrypted the top 10 most common cipher letters, and assigned the top 10 most used English letters. The next two are `P` (4.45%) and `G` (2.78%), whilst in plaintext the next two are `l` (4.0%) and `c`(2.8%). It thus seems likely that `P` $\mapsto$ `l`. This tallies with the trigram `NPP` $\mapsto$ `all`.
- The trigram `TKG` decrypts to `-o-`. The two trigrams of a similar shape in the English table are `you` and `for`. But we have already assigned `r` to `C`, so likely we have `T` $\mapsto$ `y` and `G` $\mapsto$ `u`. Checking against the single-letter frequencies, we see `T` and `y` appear 2.17% and 2.0% respectively, and `G` and `u` appear 2.78% and 2.8% respectively, so our guesses seem sound.
- `FKC` decrypts as `-or`, so `F` (2.49%) likely decrypts as `f` (2.2%).

At this point we've decoded *everything* on our small list of common digrams and trigrams. Our decryption table is now:

cipher	A	B	C	D	E	F	G	H	I	J	K	L	M
plaintext			r	s	t	f	u		n	d	o	i	
cipher	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
plaintext	a	g	l				y			h		e	

We could continue decrypting just by looking at the frequencies of more bigrams and trigrams in the cipher (for example, 'qu' will be rare amongst all digrams, but hugely more common than 'qa', 'qb', etc.).

We've not yet *touched* the actual ciphertext other than to count frequencies. However, we've already done enough to understand the entire message. If we substitute the letters we've already found, we get:

```
QhenXrBilBoBagginsofBagendannounAedthatheQould
shortlyBeAeleBratinghiseleRentyfirstBirthdayQi
thaUartyofsUeAialXagnifiAenAethereQasXuAhtalMa
ndeSAiteXentinhoBBitonBilBoQasReryriAhandReryU...
```

Now it's possible to recognise partial words: for example, `Qhen` is very likely to be `when` (whence `Q` $\mapsto$ `w`), `announAed` is probably `announced` (so `A` $\mapsto$ `c`), and `Birthday` is (helpfully) `birthday` (`B` $\mapsto$ `b`). After making these changes, we can isolate the phrase

`birthdaywithaUartyofsUecialXagnifence`,

at which point it's clear that `U` $\mapsto$ `p` and `X` $\mapsto$ `m`. From `bilbowasReryrichandRery...` we deduce `R` $\mapsto$ `v`. Finally `therewasmuchtalMandeScitement` yields `M` $\mapsto$ `k` and `S` $\mapsto$ `x`. Our table is now

cipher	A	B	C	D	E	F	G	H	I	J	K	L	M
plaintext	c	b	r	s	t	f	u		n	d	o	i	k

cipher	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
plaintext	a	g	l	w	v	x	y	p		h	m	e	

and the start of the ciphertext runs

```
whenmrbilbobagginsofbagendannouncedthathewould
shortlybecelebratinghiseleventyfirstbirthdaywi
thapartyofspecialmagnificencetherewasmuchtalka
ndexcitementinhobbitonbilbowasveryrichandveryp...,
```

or rather,

When Mr. Bilbo Baggins of Bag End announced that he would shortly be celebrating his eleventy first birthday with a party of special magnificence, there was much talk and excitement in Hobbiton. Bilbo was very rich and very p...

Of course, we still need to decrypt `H`, `V` and `Z`, which will go to `j`, `z` and `q` in some order. A little further on, we find the lines

```
...oldandstiffintheVointstheVob...
...craHyaboutstoriesoftheolddays...
...somethingZuiteexceptional...,
```

From which we deduce a complete decryption table:

cipher	A	B	C	D	E	F	G	H	I	J	K	L	M
plaintext	c	b	r	s	t	f	u	z	n	d	o	i	k

cipher	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
plaintext	a	g	l	w	v	x	y	p	j	h	m	e	q

Remark. It is worth emphasising here that everything we did above was very systematic. It is easy to put this sort of thing into a computer and automate the process; indeed, you can ask the website

`https://www.dcode.fr/monoalphabetic-substitution` (6.2)

to do it.

Remark. You might guess that this only worked because we took such a long message. What if we start with something much smaller? I ran the same experiment again, using only the first 100 characters, and asked the website (6.2) to decrypt automatically. It did it perfectly. When I reduced this to 50 characters, it failed.

Exercise. Repeat this example with another block of plaintext, and another random monoalphabetic cipher. This is a fun exercise akin to doing a Sudoku. In the process, convince yourself that monoalphabetic ciphers are *really* easy to break using frequency analysis.

6.3. Polyalphabetic ciphers and the Vigenère cipher

We've seen that any monoalphabetic cipher is very weak, and can be broken using frequency analysis. The *Vigenère cipher*, introduced by Blaise de Vigenère, gets around this by adding more randomness. For a long time, it was thought to be unbreakable.

Recall the addition operation on letters from Convention 6.1 (e.g. `m` + `p` = `b`, corresponding to $12 + 15 \equiv 1 \pmod{26}$).

Encryption (Vigenère cipher)**.** (1) Choose a word to be the *key*, e.g. `hobbit`..

(2) Add this word repeatedly to the key text: e.g.

plain	`whenmrbilbobagginsofbagendannouncedhewould`
+ keyword	`hobbithobbithobbithobbithobbithobbithobbit`
= cipher	`DVFOUKIWMCWUHUHJVLVTCBOXURBOVHBBDFLALKPVTW`

(i.e. `w` + `h` = `D`, and `h` + `o` = `V`, etc.)

Decryption (Vigenère cipher)**.** If you know the keyword, you decrypt by repeatedly subtracting it from the ciphertext.

The Vigenère cipher is a collection of k Caesar ciphers, where k is the length of the keyword. Crucially each letter can be encrypted to up to k different letters, depending on its position in the plaintext; for example, in the cipher above, the first `b` is encrypted as `b` + `h` = `I`, whilst the second goes to `b` + `b` = `C`, and the third to `b` + `t` = `U`. This makes the Vigenère cipher into a *polyalphabetic cipher*. A general polyalphabetic cipher takes the form:

Encryption (Polyalphabetic cipher)**.** (1) Choose a key length k.

(2) Choose k permutations $\pi_1, ..., \pi_k$ of the alphabet.

(3) Encrypt the nth character of the plaintext using $\pi_{n \pmod k}$.

For example, if $k = 3$, we encrypt the message `hello` as

$$\texttt{hello} \longmapsto \pi_1(\texttt{h}) \cdot \pi_2(\texttt{e}) \cdot \pi_3(\texttt{l}) \cdot \pi_1(\texttt{l}) \cdot \pi_2(\texttt{o}).$$

Decryption (Polyalphabetic cipher)**.** Given the secret key $(\pi_1, \ldots, \pi_k)$, decrypt the nth character of a ciphertext message by applying $\pi^{-1}_{n \pmod k}$.

Polyalphabetic ciphers foil basic frequency analysis by 'averaging out' the relative frequencies.

Example 6.6. Returning to *The Fellowship of the Ring*, I again took the first 100,000 characters and encrypted them using the Vigenère cipher, with some secret key. (We'll try and crack this key soon!) The letter frequences in the whole ciphertext were:

Letter	Frequency in cipher	Letter	Frequency in cipher
R	5.92%	F	3.82%
V	5.85%	Z	3.80%
E	4.72%	N	3.72%
W	4.48%	L	3.69%
X	4.44%	O	3.47%
B	4.40%	G	3.18%
C	4.37%	J	3.06%
K	4.09%	A	3.03%
S	3.97%	T	3.03%
Y	3.97%	H	2.99%
M	3.94%	U	2.89%
I	3.94%	Q	2.78%
P	3.87%	D	2.59%

6.3.1. Breaking a polyalphabetic cipher. To break a polyalphabetic cipher, we use the following observation.

Lemma 6.7. *If we know the key length k, then any polyalphabetic cipher can be reduced to k monoalphabetic ciphers.*

Moreover, as explained in the previous section, we can break any monoalphabetic cipher using frequency analysis. In particular, to break most polyalphabetic ciphers, *all we need to do is find k.*

6.3.2. Finding k: Babbage's method. Whilst frequency analysis is less useful for a direct attack on the Vigenère cipher, Babbage realised that it can be useful for finding k. We illustrate this first with an example.

Example. Suppose we want to encode the message

The cat is on the mat. If and when the cat eats rats, the cat gets fat

and choose the Vigenère keyword `story`, so that $k = 5$. After stripping out spaces and punctuation, we obtain the following ciphertext:

```
thecatisonthematifandwhenthecateatsratsthecatgetsfat

LASTYLBGFLLASDYLBTRLVPVVLLASTYLXOKQJTHJRZXQRRYXHJDSM
```

There are four instances of the word `the` in the plaintext. Moreover, *three of them are encrypted to the same thing*, namely `LAS`. This is exactly the kind of bad/repetitive structure we were hoping to avoid by using polyalphabetic ciphers.

What's going on? Observe that by the way the cipher is constructed, the first three instances of `the` appear at the same point in the (mod 5) cycle, so they are all encoded in *the same way*:

```
  theca|tison|thema|tifan|dwhen|theca|teats|ratst|···

+ story|story|story|story|story|story|story|story|···
-----------------------------------------------------
= LASTY|LBGFL|LASDY|LBTRL|VPVVL|LASTY|LXOKQ|JTHJR|···
```

So in each case, `the` is being encoded as

$$(\mathtt{t}+\mathtt{s})(\mathtt{h}+\mathtt{t})(\mathtt{e}+\mathtt{o}) = \mathtt{LAS}.$$

How does this help us find k? Well, two words appear at the same point in the (mod 5) cycle if the number of characters between them is a multiple of 5. Here, we find that

$$\underbrace{\mathtt{thecatison}}_{\text{10 characters}}\,|\,\underbrace{\mathtt{thematifandwhen}}_{\text{15 characters}}\,|\,\underbrace{\mathtt{thecateatsrats}}_{\text{14 characters}}\,|\mathtt{thecatgetsfat},$$

which in the ciphertext translates into

$$\underbrace{\mathtt{LASTYLBGFL}}_{\text{10 characters}}\,|\,\underbrace{\mathtt{LASDYLBTRLVPVVL}}_{\text{15 characters}}\,|\mathtt{LASTYLXOKQJTHJRZXQRRYXHJDSM}.$$

Let's turn this into an attack to find k. Observe:

- Common trigrams, e.g. `the`, will still likely be the most common in the plaintext.
- There are k possible cipher trigrams that `the` can be sent to, depending on the position (modulo k) of the letter `t`.
- If `the` appears starting at the mth character, and again starting at the nth character, and $m \equiv n \pmod k$, then both instances of `the` are encrypted to the same cipher trigram (e.g. say `BLR`).
- On average, this will happen for every kth instance of the word `the`.

In this example, the upshot is:

> *Certain fixed cipher trigrams should appear regularly, and frequently the distance between occurrences should be a multiple of k.*

So: we should look for repeated/common triples of letters in the ciphertext, and count the number of characters between each instance. This is likely to be a multiple of k; so it we compute a bunch of such distances, then k is likely to be the GCD of them.

Example (Finding polyalphabetic key length using Babbage's method)**.** We return to Example 6.6. The most common trigrams in the cipher are are `GQV`, `HSO`, `BLR`, and `OYN`, ringed in black, grey, blue and red respectively below. The ciphertext starts as follows:

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

N F N Z O C P S E C C P U S N O T I O A R K B B R R Q W R U V M X B E I N M E B K

S B F E R L E L F Z K A O A K Z X L X W J I S N S I P Z J X E B T T M M R P N K Y

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

C H K A Z R A P I B Q S K V H I N I P I S N E T M N A R E W V L N J I R W K Y X K

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

O U E E T R S E S O S A E C Y V R K O Y M M E A M L E X L A O K X R M I V M I C X

$\boxed{\mathtt{B\ L\ R}}$ A Z T A S D R M L N M S I H I R R B F N L B W K C X R Q W G A R M X Z D R

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

I Y R V M X R H R I X R E V I M V P Y T H S X C B F W R R B W N H K Y T H E R M L

V U C R M P L Q M R Q F R J Y I W V W J G D E E X Z D C B Y S O V U P W E R B V R

J J L K S L X L M S C Y R M K L C V S G N E F N U S P W V S J D V M V P B M A N B

R C L C S S A T E X C F G U P N D M T X L I M C X K Z Z R U R K M W X O E S E N F

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

T H Y S V I G H Y V P O D W C G U C Y V L O X O I W N C W Z Y H J K B R V W V K R

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

S K V K R M N F N Z O R I Z R K V V G B P K Z I E C Y V F O C U $\boxed{\texttt{B L R}}$ A V N X F

P C W Q R C Y R M G S Y W O $\boxed{\texttt{G Q V}}$ Z K V P K L W N W U K A C F Q P X G Q Z J P O

D D W S Z D T Y H T L Q W S Q C Y Z G U T D A I R V V U N B Q K Q V G Q R K T B J

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

A A R U C R L F P Z C X R M C P B B P H P E H B K Z U Z P G M E Y C Y Z M K T V T

L N E V K H P P Z I M Q O F I M V P I A E V M Z K B G Y D V E G D I R E $\boxed{\texttt{O Y N}}$ B

V B D S C X K T V T G B V V F Y W E L C X F X W R K H C Y C F Y N Y R W B Z D K S

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

B K G X C A V M A R A V N B Z W S V K G X W F K U T F M L V V Y Z L C O N Q X V N

J R G R S S A K B X U W H F E E V I U N I V F O T X M H B W M Z L W E S V K G N I

D L K T D P L V B I V E O E S D I F N O T X D E Y N G B D I J X $\boxed{\texttt{H S O}}$ A E P T M

Z E Z P L I K T R E J X G L X L L R Q R U F O Y I L I I X K V W O O W Q V R A J R

F C Y Q $\boxed{\texttt{B L R}}$ Q F S U W E C W J C X F I T B O E V M Z Y F I M O Y D N E Z R C Z

X G M E $\boxed{\texttt{B L R}}$ Q R U G C N V W W R O I Z X B O C C R G R C J H A P Y N L V B P F

N B R O Z G B D J Z G G M O O E A C F X K C H E X X U N V C W S D D W J $\boxed{\texttt{G Q V}}$ J

X $\boxed{\texttt{O Y N}}$ J M Y K F J Y O G Y C V V C V N T G J Y C R T O I F W C M K O K V W J N

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

S T B I R Q K I F N U S D P M Z C F C B J P K B F N P V E W $\boxed{\texttt{O Y N}}$ $\boxed{\texttt{B L R}}$ Q F G X

G Z P $\boxed{\texttt{B L R}}$ B R T D J T V T I O J X X B B D O A A R A V W B B L V T C Q J J Y X

R M S T F B J E U Y F Z N W L N Y G V G S O D W L N E V K A S D K U I O R I K A R

L I A I C C V D U S C X L C B D Y R W P P D B I E L F D X $\boxed{\texttt{O Y N}}$ T M I...

We observe:

- The two instances of `GQV` are separated by 550 characters.
- The four instances of `OYN` are separated by 370, 110, and 120 characters.
- The instances of `BLR` are separated by 400, 460, 40, 190, and 10 characters.

All of these differences are divisible by 10 (but nothing larger). Based on this, using Babbage's method we can be very confident that the key length k is equal to 10.

We illustrate how easy it is to break the Vigenère cipher once you know k. Break the text into 10 Caesar ciphers. In these 10 ciphers, the (overwhelmingly) most common letters are respectively

$$\texttt{N, V, V, X, S, P, O, M, I, R}. \tag{6.3}$$

In each cipher, this is likely to be the encrypted image of `e`. Subtracting `e` from each letter in (6.3), out pops our candidate keyword:

$$\text{Guessed keyword} = \texttt{jrrtolkien}.$$

If we attempt to decrypt with this, then we find the familiar phrase *When Mr. Bilbo Baggins...* popping out: we've cracked this Vigenère cipher.

6.3.3. Finding k: Friedman's method. William F. Friedman found another, rather beautiful and surprising, method for finding the key length k. His method relies on *coincidence indices.*

Take a string S of text, e.g.

$$S := \texttt{whenmrbilbobagginsofbagendannouncedthathewouldshortlybecelebratinghis}$$

For each integer n, let S_n be the nth character of S. We can ask:

For an integer $r > 0$, how often is $S_n = S_{n+r}$?

You can visualise this. Put two copies of the string one above the other, and move the bottom one along by r characters. Then the nth and $(n+r)$th characters agree exactly at places where a letter appears over itself. For example, when $r = 4$ we get 4 instances:

$$\texttt{whenmrbilbobagginsofbagendan}\boxed{\texttt{n}}\texttt{ou}\boxed{\texttt{n}}\texttt{cedthathewouldshortlybecel}\boxed{\texttt{e}}\texttt{bratingh}\boxed{\texttt{i}}\texttt{s}$$

$$\texttt{whenmrbilbobagginsofbage}\boxed{\texttt{n}}\texttt{da}\boxed{\texttt{n}}\texttt{nouncedthathewouldshortlyb}\boxed{\texttt{e}}\texttt{celebrat}\boxed{\texttt{i}}\texttt{nghis}$$

In this example, there are 4 coincidences out of a possible 61. In other words, this coincidence occurs 6.56% of the time. This is entirely in line with expectations[2]:

Proposition 6.8. *Let S be a generic string of English text, and let $r > 0$. For each n, we have*

$$\mathbf{P}\Big(S_n = S_{n+r}\Big) = 0.0657.$$

The same is true if we replace S with any monoalphabetic cipher of S.

Proof. Let

$$\begin{aligned}\mathbf{P}_{\texttt{a}} &:= \text{probability that a random letter in } S \text{ is } \texttt{a} = 0.082,\\ \mathbf{P}_{\texttt{b}} &:= \text{probability that a random letter in } S \text{ is } \texttt{b} = 0.015,\end{aligned}$$

etc., where we use the relative frequency percentages in the English language (listed in the expanded lecture notes).

We now find

$$\begin{aligned}\mathbf{P}\Big(S_n = S_{n+r}\Big) &= \mathbf{P}\Big(\text{both are } \texttt{a}\Big) + \mathbf{P}\Big(\text{both are } \texttt{b}\Big) + \cdots + \mathbf{P}\Big(\text{both are } \texttt{z}\Big)\\ &= \mathbf{P}_{\texttt{a}}^2 + \mathbf{P}_{\texttt{b}}^2 + \cdots + \mathbf{P}_{\texttt{z}}^2\\ &= 0.082^2 + 0.015^2 + \cdots + 0.001^2 = 0.0657,\end{aligned}$$

as required.

Now consider a monoalphabetic cipher $\pi(S)$ of S, corresponding to a permutation π of the alphabet. Note that

$$\mathbf{P}(\pi(S)_n = \texttt{a}) = \mathbf{P}(S_n = \pi^{-1}(\texttt{a})) = \mathbf{P}_{\pi^{-1}(\texttt{a})}.$$

[2] In this proof, we are implicitly assuming that, on average, the letters in the English language are randomly and independently distributed with fixed probabilities $\mathbf{P}_{\texttt{a}}, \mathbf{P}_{\texttt{b}}$, etc., i.e. the probability that the next letter is a is always 8.2%, regardless of the letters before it. This is, of course, false: consider the common digrams/trigrams! However as the shift length tends to infinity, this assumption becomes more and more reasonable. For example, the chance of a letter being a should not have a strong correlation with the letter 100 characters earlier. Additionally, the data supports this assumption even for low values of r.

Then by the same argument, we also have

$$\begin{aligned}\mathbf{P}\Big(\pi(S)_n = \pi(S)_{n+r}\Big) &= \mathbf{P}^2_{\pi^{-1}(\mathtt{a})} + \mathbf{P}^2_{\pi^{-1}(\mathtt{b})} + \cdots + \mathbf{P}^2_{\pi(\mathtt{z})} \\ &= \mathbf{P}^2_{\mathtt{a}} + \mathbf{P}^2_{\mathtt{b}} + \cdots + \mathbf{P}^2_{\mathtt{z}} \\ &= 0.0657,\end{aligned}$$

since the top sum is just a rearrangement of the bottom sum. □

In other words, for a string S in English, and any $r > 0$, we expect coincidences around 6.57% of the time. If we take $S =$ [first 100,000 characters of *The Fellowship of the Ring*], and compute the proportion of coincidences for various values of r, we get the following table:

Coincidence frequencies: English plaintext

r	1	2	3	4	5	6	7	8	9	10
% $S_n = S_{n+r}$	6.23%	6.24%	6.24%	6.24%	6.25%	6.25%	6.22%	6.23%	6.22%	6.25%
r	11	12	13	14	15	16	17	18	19	20
% $S_n = S_{n+r}$	6.25%	6.25%	6.25%	6.22%	6.25%	6.24%	6.23%	6.23%	6.24%	6.25%
r	21	22	23	24	25					
% $S_n = S_{n+r}$	6.25%	6.27%	6.24%	6.25%	6.26%					

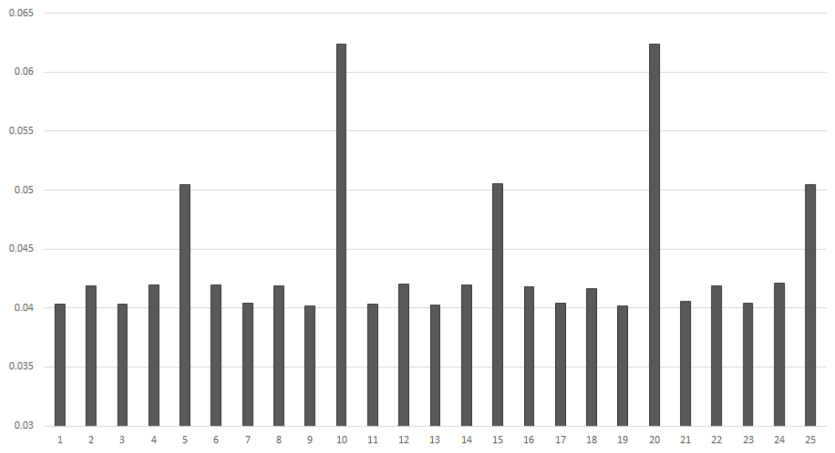
Remark. All these numbers are remarkably similar, but different from the average 6.57% we gave in Proposition 6.8. To explain this, note the distribution of letters across this passage of text will be close to, but not the same as, the averages across the whole English language (see, for example, the table on page 97). More precisely, the proportion of `a`'s will be slightly different in our specific text than in English. Using the *actual* frequencies for this passage in the proof of Proposition 6.8 will give a number around 0.0623.

Now encrypt S with the Vigenère cipher with keyword `jrrtolkien`, of length $k = 10$, as above. Let S' be the resulting ciphertext. Now the numbers are:

Coincidence frequencies: after Vigenère cipher with $k = 10$

r	1	2	3	4	5	6	7	8	9	10
% $S'_n = S'_{n+r}$	4.03%	4.19%	4.03%	4.19%	5.04%	4.19%	4.04%	4.19%	4.02%	**6.24%**
r	11	12	13	14	15	16	17	18	19	20
% $S'_n = S'_{n+r}$	4.04%	4.20%	4.02%	4.20%	5.06%	4.18%	4.04%	4.17%	4.02%	**6.24%**
r	21	22	23	24	25					
% $S'_n = S'_{n+r}$	4.05%	4.19%	4.04%	4.21%	5.05%					

These values are generally much smaller than before, *except* at $r = 10$ and 20, which are *multiples of the key length.* Graphically, this is clear:



Again, this is entirely in line with expectations. If π is a permutation of the alphabet, and S a string of letters, then let $\pi(S)$ be the string after applying π to each letter in turn.

Proposition 6.9. *(i) Let S be a generic string of English text, and $r > 0$. Let π be a non-trivial permutation of the alphabet. Then*

$$\mathbf{P}\Big(\pi(S)_n = S_{n+r}\Big) < 0.0657.$$

(ii) If $\pi \neq \pi'$ are two permutations, then

$$\mathbf{P}\Big(\pi(S)_n = \pi'(S)_{n+r}\Big) < 0.0657.$$

Proof. (i) We have

$$\mathbf{P}\Big(\pi(S)_n = S_{n+r}\Big) = \mathbf{P}_{\pi^{-1}(\mathtt{a})}\mathbf{P}_{\mathtt{a}} + \mathbf{P}_{\pi^{-1}(\mathtt{b})}\mathbf{P}_{\mathtt{b}} + \cdots + \mathbf{P}_{\pi(\mathtt{z})}\mathbf{P}_{\mathtt{z}}$$

We must show this is *strictly less than* $\mathbf{P}(S_n = S_{n+r})$. To see this, compute

$$\mathbf{P}\Big(S_n = S_{n+r}\Big) - \mathbf{P}\Big(\pi(S)_n = S_{n+r}\Big) = \sum_{i=\mathtt{a}}^{\mathtt{z}} \Big[\mathbf{P}_i^2 - \mathbf{P}_{\pi^{-1}(\mathtt{i})}\mathbf{P}_{\mathtt{i}}\Big].$$

Now we do something cheeky. Since

$$\mathbf{P}_{\mathtt{a}}^2 + \cdots + \mathbf{P}_{\mathtt{z}}^2 = \mathbf{P}_{\pi^{-1}(\mathtt{a})}^2 + \cdots + \mathbf{P}_{\pi^{-1}(\mathtt{z})}^2,$$

we can write

$$\sum_{i=\mathtt{a}}^{\mathtt{z}} \mathbf{P}_i^2 = \tfrac{1}{2}\sum_{i=\mathtt{a}}^{\mathtt{z}} \Big(\mathbf{P}_i^2 + \mathbf{P}_{\pi^{-1}(i)}^2\Big),$$

and hence

$$\begin{aligned}\mathbf{P}\Big(S_n = S_{n+r}\Big) - \mathbf{P}\Big(\pi(S)_n = S_{n+r}\Big) &= \frac{1}{2}\sum_{i=\mathtt{a}}^{\mathtt{z}} \mathbf{P}_i^2 + \frac{1}{2}\sum_{i=\mathtt{a}}^{\mathtt{z}} \mathbf{P}_{\pi^{-1}(i)}^2 - \sum_{i=\mathtt{a}}^{\mathtt{z}} \mathbf{P}_{\pi^{-1}(i)}\mathbf{P}_i \\ &= \frac{1}{2}\sum_{i=\mathtt{a}}^{\mathtt{z}} \Big(\mathbf{P}_i^2 - 2\mathbf{P}_{\pi^{-1}(i)}\mathbf{P}_i + \mathbf{P}_{\pi^{-1}(i)}^2\Big) \\ &= \frac{1}{2}\sum_{i=\mathtt{a}}^{\mathtt{z}} \Big(\mathbf{P}_i - \mathbf{P}_{\pi^{-1}(i)}\Big)^2 \\ &> 0,\end{aligned}$$

since $\pi \neq 1$. Thus

$$\mathbf{P}\Big(S_n = S_{n+r}\Big) > \mathbf{P}\Big(\pi(S)_n = S_{n+r}\Big).$$

(ii) We have

$$\mathbf{P}\Big(\pi(S)_n = \pi'(S)_{n+r}\Big) = \mathbf{P}\Big(S_n = \pi^{-1}\pi'(S)_{n+r}\Big) < 0.0657,$$

where the equality is the second part of Proposition 6.8 and the inequality is (i). □

Now let S be a string of English, and let S' be the same string encrypted by a Vigenère cipher of length k, corresponding to k permutations $\pi_1, ..., \pi_k$. For any r, we have

$$\mathbf{P}\Big(S'_n = S'_{n+r}\Big) = \mathbf{P}\Big(\pi_{n \ (\mathrm{mod}\ k)}(S)_n = \pi_{n+r \ (\mathrm{mod}\ k)}(S)_{n+r}\Big).$$

Now if r is a multiple of k, then $n \equiv n + r \ (\mathrm{mod}\, k)$, so the permutations $\pi_{n \ (\mathrm{mod}\ k)}$ and $\pi_{n+r \ (\mathrm{mod}\ k)}$ are the same; and otherwise, they are (in general) different. By Propositions 6.8 and 6.9, we have

$$\mathbf{P}\Big(S'_n = S'_{n+r}\Big)\begin{cases} = 0.0657 & : r \text{ is divisible by } k \\ < 0.0657 & : \text{otherwise}\end{cases}$$

In other words, we should expect around 6-7% coincidences when r is a multiple of k, and strictly fewer otherwise: which is what we observed on page 108!

6.3.4. A perfect Vigenère cipher. In the previous sections I claimed that if we can find k, and hence obtain k different Caesar ciphers, then we can crack the Vigenère cipher. This is certainly true *if k is small relative to the size of the ciphertext.* Suppose for example that the ciphertext has length N; then each of the Caesar ciphers has length N/k, and if N/k is large enough, we can perform frequency analysis.

However, if we take k to be *large* – for example, taking $k = N$ – then new ideas are needed, as each of the Caesar ciphers will contain only a single letter. One method of doing this is to take the key to be, say, a page of a book. However, this is also susceptible to frequency analysis, as for example `e` will be the most common letter of the plaintext, and also of the key.

We can foil frequency analysis altogether by taking a completely *random*[3] key of length $k = N$. If you use this key exactly once, this is a perfect, unbreakable Vigenère cipher. This is method of encryption is called the *one-time pad*, and was used by Soviet agents during the cold war.

Exercise. Why does the one-time pad become less safe if use the random key more than once?

[3]Generating genuinely random keys is harder than it sounds! We will not describe why in detail.

6.4. Better ciphers

All of the attacks we have outlined exploit structure in the cipher – for example, the mathematical function used to encode in the affine cipher, statistical analysis of the English language in monoalphabetic ciphers, or the length of the key in the Vigenère cipher. In passing from mono- to poly-alphabetic ciphers, we also saw:

Any cipher can be improved by adding more randomness.

(In this case, the randomness averaging out against frequency analysis).

Other methods for increasing randomness to foil frequency analysis include:

- Use multiple languages/codenames/acronyms/abbreviations.
- Delete every instance of the word `the`, and other common words.
- Rndomly mispll wrds, for exmpl by deltng commn vowls.
- Replace each space with one of the rare letters `v`, `k`, `q`, `j`, `x`, `z` chosen at random.
- Do these things *after* encryption.
- After encryption, encode the ciphertext using an e-error-correcting code. Then make e deliberate, random, 'errors' in every codeword. Any legitimate readers will know they need to decode before decrypting, but illegitimate attackers attempting conventional attacks are likely to be confused.

Most modern ciphers are *block ciphers.* As in Part I, these work with binary strings considered as vectors over $\mathbf{F}_2$, blending ideas from linear algebra with the alphabetic ciphers we just described. For example:

Example 6.10 ("Binary Vigenère cipher")**.** (1) Break your message into 256-bit binary strings (e.g. using ASCII).

(2) Pick a random integer $0 < k < 2^{256}$, and write $k \in V_{256}$ in its binary form.

(3) For each string $m \in V_{256}$ in the message, encrypt m as the ciphertext $c = m + k \in V_{256}$.

(4) To decrypt, you compute $m = c + k$.

This is a very, very basic step in the current gold standard in encryption, namely AES-256. This, like all Vigenère ciphers, is not secure by itself. In AES-256, it is combined with a number of linear algebra steps and repeated up to 14 times, each time scrambling the message more and more and confusing attackers.

To attack AES-256, an attacker really needs the key k. AES-256 is so strong that it is practically impossible to determine k, *even if the attacker is able to generate their own ciphertext from plaintext.* In other words, you can toy with attackers: if they give you any plaintext of their choice, and you tell them the corresponding ciphertext using your chosen value of k, they still can't determine k. (Note this is absolutely not true of any of the ciphers we've covered in detail in this chapter. If you can generate your own ciphertext from a chosen plaintext, then you can immediately determine the key for any mono- or poly-alphabetic cipher). This kind of strength is now considered essential for a cipher to be secure.

We don't have time to describe AES-256 in detail in this course, but it is easy to find accounts of it online, for example `https://en.wikipedia.org/wiki/Advanced_Encryption_Standard`

This uses keys of length between 128 and 256 bits. This cipher is very secure and efficient, and the encryption and decryption steps are so quick that it can be used to encrypt a live video stream (see e.g. `https://www.dacast.com/blog/aes-video-encryption`).

Chapter 7

A number theory toolkit

All the ciphers we've encountered so far are examples of *symmetric key ciphers* – they use the same key to encrypt and decrypt the message. For example, in the Vigenère cipher the message is encrypted and decrypted with the same keyword (we used the key `jrrtolkien` in the examples of §6.3). As we mentioned in §6.4, secure versions of symmetric key ciphers are widely used in modern cryptography.

However, all symmetric key ciphers have one **big problem**. Suppose Alice and Bob want to exchange secret messages using a symmetric key cipher, say a Vigenère cipher with key `jrrtolkien`. Since this key is used both to encrypt and decrypt the message:

Before Alice and Bob can exchange messages, they both *need to know the key!*

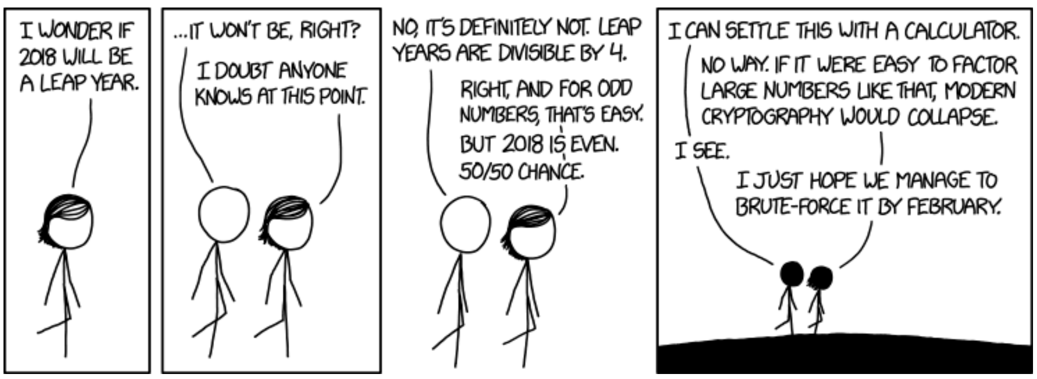
This is a real chicken-and-egg situation: it means that

Before Alice and Bob can start sending secure messages to each other...

...they have to send a secure message to each other (containing the key).

This conundrum was first[1] solved in 1976 by Whitfield Diffie and Martin Hellman. Their solution used number theory, G.H.Hardy's famously 'useless' subject. Their work founded an area now known as *public key cryptography*.

In this chapter, we'll develop a toolkit of number-theoretic functions and results that we will later use to build our public key cryptographic algorithms.


I WONDER IF 2018 WILL BE A LEAP YEAR.
...IT WON'T BE, RIGHT?
I DOUBT ANYONE KNOWS AT THIS POINT.
NO, IT'S DEFINITELY NOT. LEAP YEARS ARE DIVISIBLE BY 4.
RIGHT, AND FOR ODD NUMBERS, THAT'S EASY. BUT 2018 IS EVEN. 50/50 CHANCE.
I CAN SETTLE THIS WITH A CALCULATOR.
NO WAY. IF IT WERE EASY TO FACTOR LARGE NUMBERS LIKE THAT, MODERN CRYPTOGRAPHY WOULD COLLAPSE.
I SEE.
I JUST HOPE WE MANAGE TO BRUTE-FORCE IT BY FEBRUARY.

From `https://xkcd.com/1935`

[1] Actually, it was solved much earlier, in 1969, by James Ellis, Clifford Cox and Malcolm Williamson. However, they were working for GCHQ (the British signals intelligence agency), so their solution was classified 'top secret' for decades, only being declassified in 1997!

7.1. The factoring trapdoor

I've developed an algorithm guaranteed to crack any password in the world:

Algorithm 7.1.
- *Try all possible one-letter words (with lower case, capitals, punctuation, symbols, numbers...). Here, by a 'word' I mean* any *string of characters, not just actual English words!*
- *Now try all possible two-letter words.*
- *Now try all possible three-letter words.*
- *...continue until you've cracked the password.*

Any password necessarily has finite length N, and there are only finitely many words of length up to N; so in some finite amount of time, this algorithm will eventually find any password.

Of course, there's a big problem with my algorithm. There are *so many* possible words. Even if I check a word every second, and use only lower- and upper-case letters, it will take me over 30,000 years even to check up to 7 letters.

The idea behind public-key cryptography is to find *mathematical problems* that cannot be solved quickly. The classic example is factoring integers:

If I give you a very large number $N = pq$, it is very difficult to quickly *find p and q.*

You can always find p and q by trial division: first try and divide by 2, then by 3, then by 4, and so on, but if N is very big, this takes a *looooong* time. There has been a huge amount of work over many decades trying to find better methods, but even the *best* modern factoring algorithms are too impossibly slow.

Example 7.2. The 'RSA factoring challenge', created in 1991, gave a list of values of pq and offered cash prizes for those who could successfully factor to find p and q.

As of September 2022 the largest successfully factored number from the list had 250 digits (RSA-250): namely, the number

$$\begin{gathered}2140324650240744961264423072839333563008614715144755017797754920881418023447\\140136643345519095804679610992851872470914587687396261921557363047454770520\\8051190564931066876915900197594056934574522305893259766974716817380693648946\\9987157849497593749793 7.\end{gathered}$$

They found this to be equal to the product

$$\begin{gathered}6413528947707158027879019017057738908482501474294344720811685963202453234463\\0238623598752668347708737661925585694639798853367\\\times\\3337202759497815655622601060535511422794076034476755466678452098702384172921\\0037080257448673296881877565718986258036932062711.\end{gathered}$$

The computation ‘involved tens of thousands of machines worldwide, and was completed in a few months’.[2]

For context, in 2009, the record stood at 232 digits: so the community have increased it by only 18 digits in around a decade. The numbers used in cryptography are now over 600 digits long, so there’s no hope of factoring them using current technology!

One of the curiosities of factoring is that going the other way is incredibly quick and easy. If I give you p and q, you can write down the product pq, even by hand, very quickly. In other words, we have the picture

$$\begin{tikzcd} \boxed{p \text{ and } q} \arrow[r, bend left, "\text{super-fast}"] & \boxed{pq} \arrow[l, bend left, "\text{impossibly slow}"] \end{tikzcd}$$

We say multiplication/factoring is a *trapdoor function*. You can step/fall through in one direction, but once you done that, you can’t easily get out again.

This concept is *incredibly* useful when it comes to cryptography. We want a system that:

- Take so long to attack that it keeps out attackers;
- But that can be unlocked rapidly if you know the key/password.

Nobody wants to sit around for half an hour every time their password is verified, or queue on their bank’s website whilst it does maths. Thus our cryptographic standards need to revolve around functions that a computer can do incredibly quickly, but which cannot be quickly undone.

7.2. What is fast and what is slow?

We need a more precise way to measure how long an algorithm takes (since ‘very quick’, ‘a bit quicker’, and ‘a long time’ don’t cut it in the real world). For this, it is useful to recall big-Oh notation again.

Definition. Let

$$f : \mathbf{R} \longrightarrow \mathbf{C} \qquad \text{and} \qquad g : \mathbf{R} \longrightarrow \mathbf{C}$$

be two functions. We say that f is *big-Oh* of g, written $f = O(g)$, as x tends to 0 (or ∞) if there exist a constant c such that

$$|f(x)| \leq c \cdot |g(x)| \quad \text{as } x \to 0 \text{ (or } x \to \infty).$$

Example. Here are some runtimes of standard operations.

- Addition: computing $a + b$ is $O\Big(\max(\text{size}(a), \text{size}(b))\Big)$.
- Multiplication: computing ab is $O\Big(\text{size}(a)\text{size}(b)\Big)$.

The following are two major classes of algorithm runtimes.

[2]The factorisation was completed in 2020 by Fabrice Boudot, Pierrick Gaudry, Aurore Guillevic, Nadia Heninger, Emmanuel Thomé, and Paul Zimmermann. See:
`https://www.schneier.com/blog/archives/2020/04/rsa-250_factore.html`.

Definition 7.3. Let $f : \mathbf{N}^k \to \mathbf{R}$ be a function.

"What is slow?" We say a function $f : \mathbf{N}^k \to \mathbf{R}$ has *exponential runtime* if $f(n_1, ..., n_k)$ can be computed in time

$$O(n_1^{e_1} \cdots n_k^{e_k}),$$

for some $e_1, ..., e_k > 0$.

It is also slightly unnatural to continue to talk about the 'number of digits' a number has in its decimal expansion. Given computers work in binary, it makes much more sense to define the size of a number in terms of the number of digits in its *binary* expansion.

Definition 7.4. Let $a \in \mathbf{N}$. The *size* of a, written $\mathrm{size}(a)$, is the number of digits(/bits) in its binary expansion. Equivalently, we have

$$\mathrm{size}(a) = \lfloor \log_2 a \rfloor + 1.$$

Definition 7.5. ***"What is fast?"*** We say f has *polynomial runtime* if $f(n_1, ..., n_k)$ can be computed in time

$$O(\mathrm{size}(n_1)^{e_1} \cdots \mathrm{size}(n_k)^{e_k}),$$

for some $e_1, ..., e_k > 0$.

Remark. This might be confusing: note that

$$\text{an algorithm is polynomial} \iff \text{its runtime is polynomial in } \mathrm{size}(n_i),$$

and

$$\text{an algorithm is exponential} \iff \text{its runtime is polynomial in } n_i.$$

Example 7.6. The difference between these is *enormous.*

- Polynomial time is *rapid.*
- Exponential time is *very, very slow.*

For example:

- Multiplication is polynomial time. A computer can multiply the two enormous factors in Example 7.2 in a fraction of a second.
- Trial division factoring is exponential time. Suppose we tried to factor the product in Example 7.2 by trial division. After performing as many trial divisions as there are atoms in the universe, we would have checked less than $0.000 \cdots 00001\%$ (with *one hundred* 0s) of all possible factors. Gulp!

Example. Trial division is hopeless as a factoring algorithm. There are many better factoring algorithms out there, and the best currently known is the *number field sieve*, which was used for the records above. This is faster than exponential time, but slower than polynomial time (we call such algorithms *sub-exponential*). Thus no known polynomial-time factoring algorithm exists, and this is why factoring is difficult: it is 'not efficient'.

($*$) **Remark.** In 1994 Peter Shor proposed a polynomial-time algorithm for factoring numbers. Wait! Surely this means factoring is efficient, so no longer secure for cryptography? Well, no, your bank details are still secure. Shor's algorithm has a major catch: it requires a *quantum computer.*

Quantum computers have been in development for decades. However, existing quantum computers are very weak, and a quantum computer that can run Shor's algorithm on large numbers is still a long way away. To give a flavour of just how far away: in 2019, Shor's algorithm was successfully used[3] to factor 15 and 21; but if failed on 35, due to 'cumulative errors'.

When a viable quantum computer is first built, then multiplication/factoring will cease to be a trapdoor function: it will be very much a two-way door, and cryptographic standards will need to be changed to remain secure. The hunt for quantum-secure cryptography is very much on; for example, there is an ongoing process organised by the US National Institute of Standards and Technology – NIST – to approve new methods of quantum-secure cryptography. Pure mathematics remains at the heart of all of the proposals. Some require much more abstract/difficult mathematics than present standards: for example, one of the proposals is based on isogenies of supersingular elliptic curves over finite fields. This particular proposal will only make sense to students who have studied Master's level pure maths courses in Algebraic Geometry and Elliptic Curves.

7.3. The number theory toolkit

We're ready to populate our toolkit of number theory. We're looking to build algorithms using both

- efficient operations, i.e. things that can be done in polynomial time;
- and trapdoor operations, i.e. operations that are polynomial in one direction, but slow to reverse.

We've already put a hammer in our toolkit:

Tool 1. Multiplication of integers.
A polynomial-time algorithm to compute $a \times b$ from a, b.

Fundamental to our other tools will be modular arithmetic, as we first met in §6.1.2.

We already recapped some basic results about modular arithmetic in §6.1.2. Most fundamentally, recall that if a, b and n are integers, then we wrote $a \equiv b \pmod n$ if n divides $a - b$.

7.3.1. Euclid's algorithm: finding inverses modulo n.

Recall that an element $a \pmod n$ is *invertible* if there exists b such that

$$ab \equiv 1 \pmod n,$$

in which case we say $b = a^{-1} \pmod n$; and that

$$a \text{ is invertible } \pmod n \iff a \text{ is coprime to } n.$$

The screwdriver in our toolkit is:

[3]https://arxiv.org/pdf/1903.00768.pdf

Tool 2. Euclid's algorithm.
A polynomial-time algorithm that computes $a^{-1} \pmod n$ when a is invertible modulo n.

More specifically, given two integers m and n, Euclid's algorithm finds integers x and y with

$$xm + yn = \gcd(m, n).$$

In the special case where m and n are coprime – that is, when m is invertible $\pmod n$ – we have

$$xm + yn = 1, \qquad \text{so} \qquad xm \equiv 1 \pmod n,$$

so $x = m^{-1} \pmod n$ is the required inverse.

Algorithm 7.7 (Euclid's algorithm)**.** Let m and n be natural numbers, and assume $n > m$.

(1) For (n, m), use division with remainder to find two integers q_1, r_1, with $0 \leqslant r_1 < m$, such that

$$n = q_1 \cdot m + r_1.$$

(2) Now repeat with (m, r_1), finding two integers q_2 and $0 \leqslant r_2 < r_1$ such that

$$m = q_2 \cdot r_1 + r_2.$$

(3) Repeat with (r_2, r_1), then (r_3, r_2), etc., at each stage finding (q_i, r_i) with $0 \leqslant r_i < r_{i-1}$, until...

(4) Since $r_1 > r_2 > \cdots \geqslant 0$, and all the r_i's are integers, at some point, we must have

$$\begin{aligned} r_{k-2} &= q_k r_{k-1} + r_k, \qquad (7.1)\\ r_{k-1} &= q_{k+1} r_k + 0, \end{aligned}$$

that is, $r_{k+1} = 0$. Then $r_k = \gcd(m, n)$ is the greatest common divisor of m and n.

Now we work backwards to write r_k as a linear combination of m and n.

(5) By (7.1), we have

$$\gcd(m, n) = r_k = r_{k-2} - q_k r_{k-1} \tag{7.2}$$

is a linear combination of r_{k-2} and r_{k-1}. Similarly

$$r_{k-1} = r_{k-3} - q_{k-1} r_{k-2} \tag{7.3}$$

is a linear combination of r_{k-3} and r_{k-2}. Substituting (7.3) into (7.2), we obtain $\gcd(m, n)$ as a linear combination of r_{k-3} and r_{k-2}, namely

$$\gcd(m, n) = r_{k-2} = q_k \Big(r_{k-3} - q_{k-1} r_{k-2} \Big).$$

(6) We repeat this with r_{k-2}, r_{k-3}, etc., eventually finding r_2 is a linear combination of m and r_1 (so $\gcd(m, n)$ is a linear combination of m and r_1) and r_1 is a linear combination of m and n (so $\gcd(m, n)$ is a linear combination of m and n).

The algorithm then outputs $\gcd(m, n)$ and integers x, y with

$$xm + yn = \gcd(m, n).$$

Example 7.8. We want to compute the inverse of 22 modulo 31. We find the greatest common divisor by

$$\begin{array}{ccccc} \boxed{31} & = & 1\cdot & \boxed{22} & + & 9 \\ \boxed{22} & = & 2\cdot & \boxed{9} & + & 4 \\ \boxed{9} & = & 2\cdot & \boxed{4} & + & \mathbf{1} \\ \boxed{4} & = & 4\cdot & \boxed{1} & + & 0 \end{array}$$

The boxed terms are the r_i's – so

$$n = 31 \mid m = 22 \mid r_1 = 9 \mid r_2 = 4 \mid r_3 = 1 \mid r_4 = 0.$$

We see (unsurprisingly) that

$$\gcd(31, 22) = r_3 = 1,$$

i.e. 31 and 22 are coprime.

Now we work our way backward, refining each line using the previous one. We find

$$\begin{array}{ccccc} 1 & = & & \boxed{9} & -2\cdot & \boxed{4} \\ & = & & \boxed{9} & -2 & \left(\boxed{22} - 2\cdot\boxed{9}\right) \\ & = & 5\cdot & \boxed{9} & -2\cdot & \boxed{22} \\ & = & 5\cdot & \left(\boxed{31} - 1\cdot\boxed{22}\right) & -2\cdot & \boxed{22} \\ & = & 5\cdot & \boxed{31} & -7\cdot & \boxed{22} \end{array}$$

In the final step we got

$$1 = 5 \cdot 31 + (-7) \cdot 22.$$

From this we deduce that

$$\begin{aligned} 1 &\equiv (-7) \cdot 22 \pmod{31} \\ &\equiv 24 \cdot 22 \pmod{31}. \end{aligned}$$

It follows that the inverse of 22 modulo 31 is 24.

7.3.2. Fast modular exponentiation. In our toolbox, the ever-useful handheld drill is:

Tool 3. Fast modular exponentiation.
Let $0 < a < N$ and $k \in \mathbf{N}$. Fast modular exponentiation is a polynomial-time algorithm to compute $a^k \pmod{N}$.

How can we compute this?

- The naive method is to just compute a^k by repeatedly multiplying by a, and then take its remainder modulo N. This takes k multiplications, and if k is large, this is slow (exponential time in k).
- Slightly better: one can do this in only **7 multiplications** by repeatedly squaring, i.e. compute

  $$3 \mapsto 3^2 \mapsto 3^4 \mapsto 3^8 \mapsto 3^{16} \mapsto 3^{32} \mapsto 3^{64} \mapsto 3^{128},$$

  where each arrow is 'square the previous number'. Then reduce the end number modulo 31 to get the answer.

The second *looks* efficient. However, the numbers get very big, very fast, and thus **every individual multiplication becomes slow**. For example $3^{16} = 43046721$. The next step involves squaring this, which I certainly would not want to do by hand. Eventually we'd get

$$3^{128} = 11790184577738583171520872861412518665678211592275841109096961,$$

which we need to reduce modulo 31; something I don't want to attempt by hand. Moreover 128 is quite a small value of k, nothing like cryptographic size.

In particular, even though we perform only 7 multiplications, the size of the numbers involved make each multiplication step slow, and not feasible by hand.

How do we fix this? We use modular arithmetic to turn big numbers into small numbers.

Example 7.9. *Compute* $3^{128} \pmod{31}$. In 7 small multiplications:

- First compute $3^2 = 9 \pmod{31}$.
- Then compute $3^4 = (3^2)^2 = 9^2 = 19 \pmod{31}$.
- Then compute $3^8 = (3^4)^2 = 19^2 = 20 \pmod{31}$.
- Then compute $3^{16} = (3^8)^2 = 20^2 = 28 \pmod{31}$.
- Then compute $3^{32} = (3^{16})^2 = 28^2 = 9 \pmod{31}$.
- Then compute $3^{64} = (3^{32})^2 = 9^2 = 19 \pmod{31}$.
- Then compute $3^{128} = (3^{64})^2 = 19^2 = 20 \pmod{31}$.

Because we reduced modulo 31 after each step, in each multiplication the numbers are small, and this is easy to compute by hand.

Of course, this is a very special case, because

$$128 = 2^7$$

is a power of 2 – so we could just keep squaring until the exponent was 128. This works if we want to compute $a^{2^n} \pmod{N}$ for any n. For general k, we use binary expansions.

Example 7.10. *Compute* $3^{40} \pmod{31}$. The binary expansion of 40 can be written as $40 = 8+32 = 2^3 + 2^5$. Thus

$$3^{40} = 3^{2^3+2^5} = 3^{2^3} \times 3^{2^5}.$$

To compute this fast:

(1) By repeated squaring, compute and store the powers

$$\begin{aligned} 3^2 &= 9 \pmod{31}, \\ 3^{2^2} &= 19 \pmod{31}, \\ 3^{2^3} &= 20 \pmod{31}, \\ 3^{2^4} &= 28 \pmod{31}, \\ 3^{2^5} &= 9 \pmod{31}, \end{aligned}$$

(2) We compute $3^{40} \equiv 3^{2^3} \times 3^{2^5} \pmod{31}$, which is $20 \times 9 = 180 \equiv 25 \pmod{31}$.

Using binary expansions, we can always do something like this. More precisely:

Algorithm 7.11 (Fast modular exponentiation)**.** Suppose we want to compute $a^k \pmod N$.

(1) First write k in binary expansion

$$k = k_0 + k_1 \cdot 2 + \cdots + k_r \cdot 2^r,$$

where by definition $k_r = 1$.

(2) Repeatedly squaring and reducing $\pmod N$, compute and store all the values $a^{2^j} \pmod N$ for $j = 0, 1, 2, \ldots, r$.

(3) Compute the product of all $a^{2^j} \pmod N$ where $k_j = 1$; that is, compute

$$a^k \pmod N = \left(a^{2^0}\right)^{k_0} \times \left(a^{2^1}\right)^{k_1} \times \left(a^{2^2}\right)^{k_2} \times \cdots \times \left(a^{2^r}\right)^{k_r} \pmod N.$$

This product is the desired value $a^k \pmod N$.

Here we need r squarings in step (2), and at most $r-1$ multiplications in step (3); and (provided we reduce modulo N after each multiplication) each of these multipplications is small. Thus this has runtime $O(r) = O(\log_2(k))$, so it is a *polynomial-time* algorithm in k.

Example. For instance suppose we want to compute 2^{100} modulo the prime number $N = 101$. We have $100 = 2^6 + 2^5 + 2^2 = 1100100_2$ in binary. We compute and store:

j	0	1	2	3	4	5	6
k_j	0	0	1	0	0	1	1
$2^{2^j} \pmod{101}$	2	4	16	54	88	68	79.

So

$$\begin{aligned}2^{100} \equiv 16 \times 68 \times 79 &\equiv 16 \times -33 \times -22 \pmod{101}\\ &= 16 \times 11^2 \times 3 \times 2 \pmod{101}\\ &\equiv 16 \times 20 \times 6 \pmod{101}\\ &\equiv 96 \times 20 \pmod{101}\\ &\equiv -5 \times 20 \equiv -100 \equiv 1 \pmod{101}.\end{aligned}$$

(We could have easily computed this last step on a calculator, but the above illustrates a method of doing such computations by hand).

Note that at each stage we have reduced modulo 101, which keeps all the numbers small; this is much better than first computing

$$2^{100} = 1267650600228229401496703205376$$

and *then* reducing $\pmod{101}$.

An improved version of Fast Modular Exponentiation. The algorithm above has the advantage of being very intuitive. However we can optimise it further.

Above, we needed to compute and **store** the values of a^{2^j} for $j = 0, 1, 2, 3, \ldots, r = \log_2(k)$. In particular, if we were implementing this on a computer, we'd need to use memory to store these values; and then when performing the final step, where we multiply the values together, we'd have to go into the memory to look up the values. This all takes a little more time, and slows down the algorithm.

The following modification has the same fast runtime, but is more efficient, as it does not have the same memory requirements: at no stage do we need to store/look up multiple pieces of data.

Example. Again, let's compute $3^{40} \pmod{31}$. Recall $40 = 2^3 + 2^5 = 2^3(1+2^2)$. Thus

$$3^{40} = 3^{(2^2+1)2^3} = \left(3^{2^2+1}\right)^{2^3} = \left[3^{2^2} \times 3\right]^{2^3}.$$

To compute this fast, without storing extra data:

(1) We compute $3^{2^2} \pmod{31}$ by repeated squaring.

(2) Multiply by 3 to give $3^{2^2} \times 3 \pmod{31}$.

(3) Then we compute $(3^{2^2} \times 3)^{2^3} = 3^{40} \pmod{31}$ by repeated squaring again.

Again using binary expansions, we can always compute $a^k \pmod N$ by repeatedly either

- squaring the current value, or
- multiplying the current value by a.

More precisely:

Algorithm (Fast modular exponentiation, memory efficient version)**.** Suppose we want to compute $a^k \pmod N$.

(1) First write k in binary expansion

$$k = k_0 + k_1 \cdot 2 + \cdots + k_r \cdot 2^r,$$

where by definition $k_r = 1$.

(2) Start with $b = a$.

(3) For each $i = n-1, n-2, \ldots, 0$, either:

(3i) If $k_i = 1$ replace b with $a \cdot b^2 \pmod N$;

(3ii) If $k_i = 0$ replace b with $b^2 \pmod N$.

After the final step with $i = 0$, we have $b = a^k \pmod N$.

The idea is simply illustrated by the following equation

$$a^k = a^{k_0} \cdot \left(a^{k_1} \cdot \left(a^{k_2}\left(\cdots\left(a^{k_{r-1}} \cdot (a^{k_r})^2\right)^2 \cdots\right)^2\right)^2\right)^2.$$

As all we need is r squarings and at most $r-1$ multiplications by a (modulo N), this has runtime $O(r) = O(\text{size}(k))$, so is polynomial-time in k. At any one point in time, rather than storing all the powers $a^{2^n} \pmod N$ in memory, we only need to store the current value of b.

Example. Suppose we want to compute 2^{100} modulo the prime number $N = 101$. As $100 = 2^6 + 2^5 + 2^2 = 1100100_2$ in binary, we get

i	6	5	4	3	2	1	0
k_i	1	1	0	0	1	0	0
Step	$b = 2$	$b \mapsto 2b^2$	$b \mapsto b^2$	$b \mapsto b^2$	$b \mapsto 2b^2$	$b \mapsto b^2$	$b \mapsto b^2$
New b	2	8	64	$56 \pmod{101}$	$10 \pmod{101}$	$-1 \pmod{101}$	$1 \pmod{101}$

So $2^{100} \equiv 1 \pmod{101}$, as we obtained before.

Remark. In the MATH3011 exam, when asked to perform or explain Fast Modular Exponentiation, either of the given algorithms would score full marks if used or described correctly.

7.3.3. Fermat's little theorem. The last example was curious, if you have not seriously studied modular arithmetic before. We did all that work to compute $2^{100} \pmod{101}$ and found that it all cancelled down to give the answer 1. Actually we could have skipped straight to the answer, as this is *always* true.

Let p be a prime. Recall the elements of $\mathbf{Z}/p\mathbf{Z}$ are

$$\mathbf{Z}/p\mathbf{Z} = \Big\{0 \pmod{p},\ 1 \pmod{p},\ 2 \pmod{p},\ \ldots,\ p-1 \pmod{p}\Big\}.$$

Definition. Let

$$\begin{aligned}(\mathbf{Z}/p\mathbf{Z})^\times &= \mathbf{Z}/p\mathbf{Z} - \{0 \pmod{p}\}\\ &= \{1 \pmod{p},\ \ldots,\ p-1 \pmod{p}\}.\end{aligned}$$

Note that this is the set of all elements $a \pmod{p} \in \mathbf{Z}/p\mathbf{Z}$ where a is coprime to p. In particular, by Proposition 6.2, this is the subset of invertible elements, i.e.

$$\Big[a \pmod{p} \in (\mathbf{Z}/p\mathbf{Z})^\times\Big] \iff \Big[\exists b \in \mathbf{Z}/p\mathbf{Z} \text{ such that } ab \equiv 1 \pmod{p}\Big].$$

Remark. In other words, $\mathbf{Z}/p\mathbf{Z}$ is a *field*, commonly denoted $\mathbf{F}_p$: you can add, subtract and multiply any two numbers, and divide by anything except 0.

Of course, we've already seen one of these fields: when $p = 2$, we have $\mathbf{Z}/2\mathbf{Z} = \{0, 1\}$ is the field we know and love from our study of linear codes.

The next theorem[4] is very useful when we study modular exponentiation.

Tool 4 (Fermat's little theorem)**.** Let p be a prime number and let a be an integer coprime to p. Then

$$a^{p-1} \equiv 1 \pmod{p}.$$

Proof. (Non-examinable). The set $(\mathbf{Z}/p\mathbf{Z})^\times$ of invertible elements in $\mathbf{Z}/p\mathbf{Z}$ is a group. Let d be the order of $a \pmod{p}$ in this group, i.e. let d be the smallest positive integer such that $a^d \equiv 1 \pmod{p}$.

Note that $(\mathbf{Z}/p\mathbf{Z})^\times$ has size $p - 1$; indeed, since every $0 < b < p$ is invertible, we have

$$(\mathbf{Z}/p\mathbf{Z})^\times = \Big\{1 \pmod{p},\ 2 \pmod{p},\ \ldots,\ p-1 \pmod{p}\Big\}.$$

[4]Fermat's little theorem is rather less famous than his last theorem; but thankfully its proof did fit in the margin, and didn't take 350 years to 'rediscover'.

By Lagrange's theorem, the order d of $a \pmod p$ must divide $p-1$, say $p-1 = d \cdot e$ for some integer e. Then

$$\begin{aligned} a^{p-1} = a^{de} &= (a^d)^e \\ &\equiv 1^e = 1 \pmod p, \end{aligned}$$

as required. $\square$

Here's another proof, which doesn't appeal to group theory. Consider the values $\{a, 2a, .., (p-1)a \pmod p\}$. I claim that this is a rearrangement of $\{1, 2, ..., p-1\}$. Indeed, if $xa = ya \pmod p$, then $(x-y)a = 0 \pmod p$, so p divides $(x-y)a$. Since p does not divide a, and p is prime, p must divide $x-y$. But then $x = y \pmod p$, and as $0 < x, y < p$, we must have $x = y$. Thus no two elements of the set $\{a, 2a, ..., (p-1)a \pmod p\}$ are the same. As both sets thus have the same side, and by definition the left-hand set is contained in the right-hand set, we thus have

$$\{a \pmod p, 2a \pmod p, ..., (p-1)a \pmod p\} = \{1, 2, ..., p-1\},$$

as claimed.

Now multiply all the elements of these sets together. On the left-hand side, we get

$$a \cdot 2a \cdot 3a \cdots (p-1)a = (p-1)!a^{p-1}.$$

On the right-hand side, we just get $(p-1)!$. As the two sets are the same, we thus get

$$(p-1)!a^{p-1} \equiv (p-1)! \pmod p.$$

As $(p-1)!$ is coprime to p, it is invertible modulo p; and cancelling it gives $a^{p-1} \equiv 1 \pmod p$, as claimed. $\square$

Remark. Let us hint at the relevant consequences of this theorem. Suppose we have picked an integer $N = pq$, and that we know p and q. If a is an integer coprime to N, then

$$a^{(p-1)(q-1)} \equiv 1^{q-1} \equiv 1 \pmod p, \qquad a^{(p-1)(q-1)} \equiv 1^{p-1} \equiv 1 \pmod q,$$

and by the Chinese Remainder Theorem this means

$$a^{(p-1)(q-1)} \equiv 1 \pmod N.$$

We'll exploit this later when we define the RSA cryptographic system.

7.4. Trapdoor #2: discrete logarithms

The second trapdoor function we'll consider is *modular exponentiation.* For this, we specialise to arithmetic $\pmod p$, where p is a (usually large) prime number. We've already seen that computing powers modulo p is a very fast operation (via fast modular exponentiation, Tool 3 of our kit, described in Algorithm 7.3.2):

$$\begin{tikzcd} \boxed{a,\ p, \text{ and } k} \arrow[r, bend left=10, "\text{super-fast}"] & \boxed{a,\ p, \text{ and } a^k \pmod p} \end{tikzcd}$$

What about going the other way? That is, how fast can we solve the question:

Find k such that $2^k \equiv 12 \pmod{23}$.

There's no easy computation to just 'do' in this case. Without having a clever idea, the only thing that really comes to mind is to compute the values

$$2^1 \equiv 2 \pmod{23}, \quad 2^2 \equiv 4 \pmod{23}, \quad 2^3 \equiv 8 \pmod{23}, \quad \dots$$

until we find a value k with $2^k \equiv 12$. Eventually, we'll get the answer $k = 10$.

Now scale up:

Find k such that $2^k \equiv 7123457 \pmod{49979687}$.

...gulp. I genuinely don't know the answer without using a computer. Even then, there is no known polynomial-time algorithm for computing this.

We see that modular exponentiation is another trapdoor function:

$$\begin{tikzcd} \boxed{a,\ p,\ \text{and}\ k} \arrow[r, bend left, "\text{super-fast}"] & \boxed{a,\ p,\ \text{and}\ a^k \pmod p} \arrow[l, bend left, "\text{impossibly slow}"] \end{tikzcd}$$

Definition 7.12. (i) Given numbers a, p and $c = a^k \pmod p$, the question of determining the value of k from a, p and c is called the *discrete logarithm problem.*

(ii) Such a k is called the *discrete logarithm of* $c \pmod p$ *with respect to* a.

Example. Let $p = 7$, and $a = 3$. Then

$$\begin{aligned} a^1 &\equiv 3 \pmod 7, \\ a^2 &\equiv 2 \pmod 7, \\ a^3 &\equiv 6 \pmod 7, \\ a^4 &\equiv 4 \pmod 7, \\ a^5 &\equiv 5 \pmod 7, \\ a^6 &\equiv 1 \pmod 7. \end{aligned}$$

Thus (with respect to $a = 3$, modulo $p = 7$) the discrete logarithm of 3 is 1, the discrete logarithm of 2 is 2, the discrete logarithm of 6 is 3, etc.

7.4.1. Primitive elements. Suppose we are given the data a, p and $c = a^k \pmod p$, and we want to find the discrete logarithm k of c. Naively we can do this by computing the values

$$\{a^1 \pmod p, a^2 \pmod p, a^3 \pmod p, \dots, a^{p-1} \pmod p\}, \tag{7.4}$$

since we know c is an element of this set. This seems to take $p - 1$ computations, making it at least $O(p)$ work, i.e. an exponential algorithm (in p).

However. This security relies in the set in (7.4) being very large, so it is computationally very hard to find which element corresponds to c. A priori, this set seems to have $p - 1$ elements. But this is not always true: you must be very careful about your choice of a.

Example. Let $p = 31$, and $a = 2$. Then

$$a^0 = 1 \pmod{p}, \quad a^1 = 2 \pmod{p}, \quad a^2 = 4 \pmod{p}, \quad a^3 = 8 \pmod{p}, \quad a^4 = 16 \pmod{p},$$
$$a^5 = 1 \pmod{p}, \quad a^6 = 2 \pmod{p}, \quad a^7 = 4 \pmod{p}, \quad \dots$$

This will repeat: so the only possible values of $a^k \pmod{p}$ are 1,2,4,8,16, so we are operating in a very small part of the group $\mathbf{Z}/31\mathbf{Z}$. Of course, the problem here is that $2^5 = 1 \pmod{31}$.

Scaling up: consider the Mersenne prime $p = 2^{2203} - 1$. This is genuinely of cryptographic size, and has over 600 digits. The discrete logarithm problem seems safe and secure at this size. However, if you choose $a = 2$, then $a^{2203} \equiv 1 \pmod{p}$, so *there are only 2203 possible values of* $c = a^k \pmod{p}$, no matter what we choose for k. A computer can easily check all of these possibilities to solve the discrete logarithm problem by brute force.

In these examples, the problem was that there is an integer d much smaller than p with $a^d \equiv 1 \pmod{p}$. (Note d is the multiplicative order of $a \pmod{p}$). By Fermat's little theorem – or more precisely, by its non-examinable proof – we see that such a d must divide $p - 1$ (by Lagrange's theorem, the order of any element of a finite group must divide the size of the group).

Definition 7.13. Let p be a prime. We say that a is a *primitive element* modulo p if a has multiplicative order $p - 1$. In other words, $p - 1$ is the smallest positive integer d such that

$$a^d \equiv 1 \pmod{p}.$$

Another description of this is as follows:

Lemma. *If a is a primitive element $\pmod{p}$, then*

$$\{a^1 \pmod{p}, a^2 \pmod{p}, a^3 \pmod{p}, \dots, a^{p-1} \pmod{p}\} = \{1, 2, 3, \dots, p - 1\}.$$

In other words, for all $0 \neq c \in \mathbf{Z}/p\mathbf{Z}$, there exists a k such that $a^k = c \pmod{p}$.

Proof. It suffices to show that the map

$$\mathbf{Z}/(p-1)\mathbf{Z} \longrightarrow (\mathbf{Z}/p\mathbf{Z})^\times$$
$$k \mapsto a^k \pmod{p}$$

is bijective. The kernel of this map is $\{d \pmod{p-1} : a^d \equiv 1 \pmod{p}\} = \{0\}$, since a is primitive (so there is no $0 < d < p - 1$ with $a^d \equiv 1 \pmod{p}$). So the map is an injective homomorphism of groups; but both sides have the same size (namely $p - 1$), so it must be bijective (in fact, it is an isomorphism of groups). $\square$

Thus when p is large and a is a primitive element, the discrete logarithm problem is hard. Happily, it is always possible to find a primitive element for any prime p:

Theorem 7.14. *Let p be a prime. There always exists a primitive element modulo p.*

Proof. (Non-examinable). High-powered proof: the group $(\mathbf{Z}/p\mathbf{Z})^\times$ is a finite multiplicative subgroup of the field $\mathbf{F}_p = \mathbf{Z}/p\mathbf{Z}$, so it is cyclic. Any generator is a primitive element. □

What's really going on here? Here's a direct proof, which is essentially the same as above but with more detail. Note $(\mathbf{Z}/p\mathbf{Z})^\times$ is a finite abelian group, hence by the structure theorem for finitely generated abelian groups it can be written as a product

$$(\mathbf{Z}/p\mathbf{Z})^\times = (\mathbf{Z}/q_1^{r_1}\mathbf{Z}) \times \cdots (\mathbf{Z}/q_m^{r_m}\mathbf{Z}),$$

where q_i is prime and $r_i \geqslant 1$. We claim that all the q_i's are different; in which case we have

$$(\mathbf{Z}/p\mathbf{Z})^\times = \mathbf{Z}/(q_1^{r_1}\cdots q_m^{r_m})\mathbf{Z} = \mathbf{Z}/(p-1)\mathbf{Z}$$

is cyclic, by the Chinese remainder theorem.

To prove the q_i's are all different, we need to use the fact that $K = \mathbf{Z}/p\mathbf{Z}$ is a field (i.e. every non-zero element has an inverse; essentially, a field is a set 'in which we can add, subtract, multiply and divide' as we would like). Also we'll need:

If $f(X) \in K[X]$ is a polynomial of degree d, then it has at most d *roots*[a] *in K.*

Now suppose one of the q_i's is repeated, say without loss of generality that $q_1 = q_2 = q$. Then $(\mathbf{Z}/p\mathbf{Z})^\times$ contains the subgroup $\mathbf{Z}/q\mathbf{Z} \times \mathbf{Z}/q\mathbf{Z}$. Every element of this subgroup is a root of the polynomial $X^q - 1$, which then has at least q^2 roots in $(\mathbf{Z}/p\mathbf{Z})^\times$, hence in $\mathbf{Z}/p\mathbf{Z}$. But this contradicts the result above; so no prime is repeated.

Now since

$$(\mathbf{Z}/p\mathbf{Z})^\times \cong \mathbf{Z}/(p-1)\mathbf{Z}$$

is cylic, it must have a generator; which is then a primitive element.

This proof was much harder than others in the course. It's very non-examinable. □

[a]Why? Suppose it has more than d roots, say $\alpha_1, ..., \alpha_{d+1}, ...$ Then $(X-\alpha_1)\cdots(X-\alpha_{d+1})$ is a polynomial of degree $d+1$ that divides f – impossible!

Given p and $a \pmod p$, a fundamental question is:

When is $a \pmod p$ a primitive element?

Lemma 7.15. *An element $a \pmod p$ is primitive if and only if for every prime divisor ℓ of $p-1$, we have*

$$a^{\frac{p-1}{\ell}} \neq 1 \pmod p.$$

Proof. If $d > 0$ is the smallest integer with $a^d \equiv 1 \pmod p$, then by Lagrange's theorem d divides $p-1$. If $d < p-1$, this means d must divide $(p-1)/\ell$, for some prime divisor ℓ of $p-1$. In other words, $(p-1)/\ell = d \cdot e$, for some integer e. But this would mean $a^{\frac{p-1}{\ell}} = a^{de} = (a^d)^e = 1^e = 1 \pmod p$.

We deduce that if $a^{\frac{p-1}{\ell}} \neq 1 \pmod p$ for every ℓ, then d does not divide $(p-1)/\ell$, so $d = p-1$ and a is primitive. □

Example. Let $p = 31$. We've shown that 2 is not primitive modulo 31. We now show that $a = 3$ is primitive modulo 31.

To see this, note the prime divisors of $p-1 = 30$ are $\ell = 2, 3$ and 5, giving $(p-1)/\ell = 15, 10$ and 6. We compute $3^{15} = 30 \pmod{31}$, $3^{10} = 25 \pmod{31}$ and $3^6 = 16 \pmod{31}$. Since none of these is $1 \pmod{31}$, by Lemma 7.15 it follows that the order of 3 is 30 and hence $a = 3$ is primitive.

Chapter 8

Public key cryptography

Recall that the fundamental problem with symmetric key ciphers, like AES-256, is that they require both the sender and receiver of messages to know the secret key; so before Alice and Bob can start exchanging secret messages, they must exchange a secret message containing the key.

In AES-256, the secret key is a 256-bit binary number, or equivalently a number $0 < k < 2^{256}$. In particular, for symmetric key cryptography Alice and Bob need to somehow secretly agree on a large number k.

This key exchange is performed using what is known as *public key cryptography*, or *PKC*. In this chapter, we see three cryptographic schemes to do exactly this, all using the trapdoor functions from the previous chapter:

- RSA, using factoring;
- Diffie–Hellman, using discrete logarithms;
- ElGamal, using discrete logarithms.

All three are named after their creators.

Using the same circle of ideas, we'll also see how to send secret messages without using any keys at all, via the so-called *no-key protocol*.

8.1. Diffie–Hellman Key Exchange

Whitfield Diffie and Martin Hellman published the first paper on public key encryption in 1976. In their paper, they introduced the following algorithm, based on the discrete logarithm problem.

Alice and Bob want to share secret messages with each other. They want to generate a key to create ciphers. To use the AES-256 cipher, they need to find a 256-bit binary number k that:

- both Bob and Alice know,
- that nobody else knows,
- that doesn't require Bob and Alice to ever meet in person to exchange.

Diffie–Hellman solved this via the following algorithm.

Algorithm 8.1 (Diffie–Hellman key exchange)**.** *(1) Alice and Bob fix a (very) large*[a] *prime p and a primitive element $a \pmod p$. These are known to everybody.*

(2) Alice chooses a secret integer $0 < d_A < p-1$, and Bob chooses a secret integer $0 < d_B < p-1$. These are the private keys*.*

(3) Alice computes her 'public key'

$$e_A = a^{d_A} \pmod p$$

and publishes it. Similarly Bob computes and publishes

$$e_B = a^{d_B} \pmod p.$$

(4) Bob computes

$$k = e_A^{d_B} = a^{d_A \cdot d_B} \pmod p.$$

by taking the d_B-th power of Alice's public key e_A modulo p.

(5) Alice can also *compute k by raising Bob's public key e_B to d_A-th power modulo p.*

So without having to meet in person to exchange a key in secret, Alice and Bob have both computed the same number k, which can now be used as a key in a classical symmetric cipher system.

[a]Presently it is agreed that the prime should be at least 1000 bits in size, and ideally bigger.

Here's a table summarising all the parts in Diffie–Hellman, and precisely who knows what at each step. We see that at the end of it all, Alice and Bob both know the same number k, but that nobody else knows what k is.

In the table, the arrows show where the public information came from: e.g. if there's an arrow from Alice's column to the middle column, it means she computed that data and published it. If there's no arrow then the public data was mutually agreed.

Figure 8.1: Diffie–Hellman: who knows what?

Known only to Alice	**Known to everybody** *Including any attackers!*	**Known only to Bob**
	large prime p primitive element $a \pmod p$	
private key $0 < d_A < p-1$		private key $0 < d_B < p-1$
$\longrightarrow$	Alice's public key $e_A = a^{d_A} \pmod p$	
	Bob's public key $e_B = a^{d_B} \pmod p$	$\longleftarrow$
$k = e_B^{d_A} \pmod p$		$k = e_A^{d_B} \pmod p$

Attack 8.2 (Diffie–Hellman)**.** Now suppose an eavesdropper, Eve, is trying to listen to Alice and Bob's messages. To read their ciphertext, she needs to know k. The only information she has are the values a, p, e_A, and e_B. She needs to know d_A or d_B to compute k. In other words, she needs to find d_A such that

$$a^{d_A} \equiv e_A \, (\operatorname{mod} p),$$

i.e. she needs to solve the discrete logarithm problem – which we know is extremely difficult, and not possible to do in a reasonable timeframe. So Eve has been foiled!

Example 8.3.

- Alice and Bob fix $p = 101$ and $a = 2$.
- Alice picks the secret key $d_A = 7$, while Bob chooses $d_B = 21$.
- Alice sends $e_A = 2^7 \equiv 27 \pmod{101}$ and Bob sends $e_B = 2^{21} \equiv 89 \pmod{101}$.
- Alice computes
$$89^7 \equiv 63 \, (\operatorname{mod} 101).$$
and Bob computes
$$27^{21} \equiv 63 \, (\operatorname{mod} 101),$$

Both of them now know the key $k = 63 \, (\operatorname{mod} 101)$, and can use this to encrypt messages using a cipher.

Exercise. Let $p = 31$. Recall that $a = 3$ is a primitive element modulo 31.

Assume that Alice and Bob choose private keys $d_A = 5$ and $d_B = 11$. Find the corresponding public keys, e_A and e_B, compute the shared secret k and verify explicitly that both Alice and Bob computes the same k. You can use any method for the computation performed by Alice but use fast modular exponentiation for the computation performed by Bob. [1]

8.2. RSA

In 1978[2] Ron Rivest, Adi Shamir, and Leonard Adleman found how one could use the trapdoor function factorisation to create a public key cipher system. Their method, abbreviated RSA, is

[1]Solution: If $d_A = 5$ then $e_A = a^{d_A} = 3^5 \equiv 26 \pmod{31}$ and if $d_B = 11$ then

$$e_B = 3^{11} = 3^5 \cdot 3^6 = 3^5 \cdot (3^3)^2 = 26 \cdot (9)^2 = 26 \cdot 16 = 13 \pmod{31}$$

so the public keys are $e_A = 26$ and $e_B = 13$. The shared secret is

$$k = e_B^{d_A} = 13^5 \pmod{31} \text{ (by A.)}$$
$$k = e_A^{d_B} = 26^{11} \pmod{31} \text{ (by B.)}$$

We can easily compute $13^5 = 371293 \equiv 6 \pmod{31}$ but for 26^{11} we use fast modular exponentiation. Note that $11 = 1 + 2 + 0 \cdot 4 + 8 = (1101)_2$ so $n = 3$, $a_3 = 1$, $a_2 = 0$, $a_1 = 1$ and $a_0 = 1$ so:

i	3	2	1	0
k_i	1	0	1	1
b	26	$26^2 \equiv 25 \pmod{31}$	$26 \cdot 25^2 \equiv 6 \pmod{31}$	$26 \cdot 6^2 \equiv 6 \pmod{31}$

The shared key is thus $k = 6 \pmod{31}$.

[2]Once again, GCHQ got there first: Clifford Cocks, in top secret work that has now been declassified, discovered the method in 1973!

among the most used systems nowadays. Most secure internet connections will use RSA in order to exchange keys for a symmetric key cipher (for example, the AES scheme we mentioned in §6.4).

As always, we suppose Alice wants to send Bob a private message k (which could then be the cipher key for a cipher system). To do this using RSA:

– *only Bob* creates private/public keys;

– Alice uses the public key to encrypt her message;

– and then Bob uses the private key to decrypt it.

More precisely, this is done in the following steps.

Algorithm 8.4 (RSA). • *First, Bob creates a public and private key as follows:*

- *Bob chooses two large*[a] *distinct prime numbers p and q and computes $N = p \cdot q$.*
- *Bob chooses a large $d < (p-1)(q-1)$ which is coprime to*[b] $(p-1) \cdot (q-1)$.
- *Using the Euclidean algorithm, he computes an inverse e of d modulo $(p-1)\cdot(q-1)$, in other words finds e such that*

$$ed \equiv 1 \mod (p-1)(q-1).$$

- *Bob now has:*
  - *a* private key (p, q, d), *which he keeps to himself; and*
  - *a* public key (N, e), *which he publishes to everyone.*
- *Alice can now chooses her message $0 < k < N$, and encrypts it by computing*

$$c = k^e \pmod N.$$

  She sends c to Bob.
- *Bob, receiving c, uses his private key d to compute $c^d = k^{de} \pmod N$.*

[a]Nowadays, many internet applications use 2048 bit encryption, in which p and q have roughly 300 digits.
[b]This is of course $\varphi(N)$ for those who know Euler's φ-function.

Again, here is a table summarising who knows what, with arrows again describing the source of public data:

Figure 8.2: RSA: who knows what?

Known only to Alice	**Known to everybody** *Including any attackers!*	**Known only to Bob**
		2 large primes p and q, $d < (p-1)(q-1)$
	$N = pq$, $\leftarrow$	$\leftarrow$
	$e = d^{-1} \pmod{(p-1)(q-1)}$ $\leftarrow$	$\leftarrow$
message k		
$\longrightarrow$	$\longrightarrow$ $c = k^e \pmod N$	
		$k^{de} = c^d \pmod N$

The main point now is of course that Bob recovers Alice's message! More precisely, $k^{de} \equiv k \pmod N$, which is expressed in the following theorem.

Theorem 8.5 (Decrypting RSA)**.** *The decoded message $k^{de} = c^d \pmod N$ in RSA is equal to Alice's original plaintext message k.*

Proof. From the construction of the keys, we know that

$$d \cdot e = t \cdot (p-1)(q-1) + 1$$

for some integer t.

First we prove:

Claim. *We have $k^{de} \equiv k \pmod p$; in other words, p divides $k^{de} - k$.*

Proof. To see this, assume first that k is coprime to p. Then $k^{p-1} \equiv 1 \pmod p$ by Fermat's Little Theorem (Theorem 4). So we have

$$\begin{aligned} k^{de} \equiv k^{t(q-1)\cdot(p-1)+1} &\equiv k \cdot \left(k^{p-1}\right)^{t(q-1)} \\ &\equiv k \cdot (1)^{t(q-1)} \equiv k \pmod p \,. \end{aligned}$$

If p divides k, then p divides k^{de}, so that and we find $k^{de} \equiv k \equiv 0 \pmod p$. This proves the claim. □

By exactly the same arguments, $k^{de} \equiv k \pmod q$ as well. We deduce $k^{de} - k$ is divisible by both p and q. Since $p \neq q$, we deduce $k^{de} - k$ is divisible by $N = pq$, and hence that $k^{de} \equiv k \pmod N$.

Now we conclude, since both $k^{de} \pmod N$ and k are in the range $\{0, 1, ..., N-1\}$. □

Attack 8.6 (RSA)**.** Suppose Eve is eavesdropping again. She knows N and e, since these are public. If she could factor $N = pq$, then she could also compute $(p-1)(q-1)$, and then compute Bob's private key $d = e^{-1} \pmod{(p-1)(q-1)}$, since e is public. However, we already know that factoring is essentially impossible in practice! And, also:

> *At present, nobody knows of any way of breaking the RSA cipher that is easier than factoring N.*

It looks like as if attacking RSA and factoring numbers are equivalently hard, though this has not been proven to be the case[a].

[a]It has been proven for a related cipher called the Rabin cryptosystem (described by Michael Rabin in 1979).

As with Diffie–Hellman, RSA is a popular and secure cryptosystem, since:

- creating keys, encryption, and decryption requires only the use of primality testing, Euclid's algorithm and fast modular exponentiation, all of which are computationally easy and fast;
- all known attacks require factoring, which is computationally impossible.

Example 8.7 (A very simple example of RSA)**.** (1) Bob chooses the primes $p = 101$ and $q = 103$; so $N = 10403$.

(2) Bob computes $(p-1)\cdot(q-1) = 10200$. He chooses a private key coprime to this, for example $d = 19$.

(3) Bob computes the public key, finding the inverse of 19 using Euclid's algorithm. He finds $e = 6979$, i.e.

$$19 \cdot 6979 \equiv 1 \pmod{10200}.$$

(4) Bob publishes his public key

$$(N, e) = (10403, 6979).$$

(5) Alice can look this up and use it to send a message. Say the message is $k = 249$. She uses fast modular exponentiation to compute the ciphertext $c = k^e \equiv 4679 \pmod{N}$. She sends this to Bob.

(6) Having receiving c, Bob computes $b = c^d \pmod{N}$ with his secret key $d = 19$. He gets $4679^{19} \equiv 249 \pmod{N}$, recovering Alice's message.

Remark. There is another attack on RSA that Eve might try. She knows N and e. Suppose she chooses her own message m, and computes the ciphertext $c = m^e \pmod{N}$. They also know that $m = c^d \pmod{N}$, where d is Bob's private key. Given she knows m and c, if Eve can solve the discrete logarithm problem

$$m = c^d \pmod{N},$$

then she can obtain the private key d and crack the system. However, this is just as hard to solve as factoring N.

8.3. ElGamal

In 1985, Taher ElGamal described another public key cipher based on the discrete logarithm trapdoor function. Suppose Alice would like to send a secret message to Bob using the ElGamal cipher. This is done in the following steps:

Algorithm 8.8 (ElGamal)**. *Bob: generate keys***

- *First, Bob creates a public and private key as follows:*
  - *he chooses a large prime p and a primitive element $a \pmod p$ modulo p.*
  - *he chooses an integer (private key) $1 < d < p-1$.*
  - *Using fast modular exponentiation he computes $e = a^d \pmod p$.*
- *Bob now has the* private key (p, a, d) *and the* public key (p, a, e)*, which he publishes to Alice and everyone else.*

Alice: encrypt message

- *Alice now chooses a* random *integer $1 \leqslant r < p-1$.*
- *Alice encrypts a message $0 \leqslant k < p$ by computing*

$$\begin{aligned} c_1 &:= a^r \,(\mathrm{mod}\, p), \\ c_2 &:= k \cdot e^r \,(\mathrm{mod}\, p). \end{aligned}$$

- *Alice sends (c_1, c_2) to Bob.*

Bob: decrypt message

- *When receiving (c_1, c_2), Bob uses his secret key d to compute*[a]

$$c_2 \cdot c_1^{-d} \,(\mathrm{mod}\, p).$$

[a]Recall that $c^{-d} \pmod p$ can be computed either by first computing the inverse $c^{-1} \pmod p$ and then raise it to the d^{th} power, or by computing c^{p-1-d} and use Fermat's little theorem. Both are efficient.

Why does this work? Recall $e = a^d$; then we have

$$\begin{aligned} c_2 \cdot c_1^{-d} &= (ke^r) \cdot (a^r)^{-d} \\ &= ke^r \cdot (a^d)^{-r} \\ &= ke^r \cdot e^{-r} \\ &= k \,(\mathrm{mod}\, p), \end{aligned}$$

so $c_2 \cdot c_1^{-d} \,(\mathrm{mod}\, p)$ is equal to the original plaintext k.

Figure 8.3: ElGamal: who knows what?

Known only to Alice	**Known to everybody** *Including any attackers!*	**Known only to Bob**
	large prime p primitive element $a \pmod p$	
		private key d, $1 < d < p-1$
	$e = a^d \pmod p$ $\longleftarrow$	$\longleftarrow$
message $0 \leqslant k < p$ *random* integer $1 \leqslant r \leqslant p-1$		
$\longrightarrow$ $\longrightarrow$	$\longrightarrow$ $c_1 = a^r \pmod p$ $\longrightarrow$ $c_2 = k \cdot e^r \pmod p$	
		$k = c_2 \cdot c_1^{-d} \pmod p$

Example 8.9 (A simple example of ElGamal).

- Alice and Bob fix $p = 101$ and $a = 2$.
- Bob picks the secret key $d_A = 7$. He computes $e = 2^7 = 27 \pmod{101}$.
- Alice wants to send the message $53 \pmod{101}$. She chooses a random integer 17, and computes

$$c_1 = 2^{17} = 75 \pmod{101},$$
$$c_2 = 53 \cdot 27^{17} = 43 \pmod{101}.$$

- Alice publishes her message $(75, 43)$.
- Bob computes $43 \cdot 75^{-7} = 53 \pmod{101}$, recovering Alice's message.

Example 8.10 (Another simple example of ElGamal).

- Bob and Alice pick the prime $p = 31$ (which is public knowledge). They agree on the following code: represent

`a` $\longmapsto 0$	space $\longmapsto 26$
`b` $\longmapsto 1$	`'` $\longmapsto 27$
$\cdots$	`.` $\longmapsto 28$
`y` $\longmapsto 24$	`,` $\longmapsto 29$
`z` $\longmapsto 25$	`?` $\longmapsto 30$

- They agree on the primitive element $a = 3$ (note we can't use $a = 2$, which is not a primitive element since $2^5 = 32 \equiv 1 \pmod{31}$).
- Bob picks the private key $d = 17$ and publishes the public key $e = 3^{17} \equiv 22 \pmod{31}$.
- Alice would like to send the message

`God made beer because he loves us and wants us to be happy.`

She sends it letter by letter using ElGamal.

- She starts with ‘g’= 6, and picks $r = 19$. Since

$$c_1 = a^r = 3^{19} = 12 \pmod{31}, \qquad c_2 = m \cdot e^r = 6 \cdot 22^{19} = 4 \pmod{31},$$

she sends the pair (12,4).

- Bob can recover $c_2 \cdot c_1^{-d} = 12 \cdot 4^{-17} = 6 \pmod{31}$, which of course he decodes back to ‘g’.
- The full ciphertext reads as

(12, 4)	(21, 26)	(28, 27)	(23, 10)	(20, 26)	(12, 0)	(18, 11)	(1, 4)
(1, 26)	(7, 18)	(24, 21)	(4, 2)	(14, 15)	(7, 3)	(13, 17)	(17,22)
(7, 5)	(17, 0)	(1, 20)	(13, 27)	(6, 11)	(17, 19)	(11, 21)	(8, 8)
(15, 9)	(30, 20)	(21, 26)	(19, 17)	(29, 15)	(28, 7)	(29, 20)	(2, 18)
(4, 9)	(16, 22)	(6, 0)	(1, 13)	(3, 4)	(18, 23)	(28, 12)	(17, 0)
(14, 6)	(18, 18)	(1, 18)	(20, 15)	(13, 30)	(1, 18)	(30, 5)	(17, 27)
(6, 23)	(30, 5)	(6, 26)	(14, 9)	(13, 8)	(8, 14)	(25, 0)	(25, 13)
(1, 15)	(18, 26)	(11, 22)					

where Alice has used random r’s for each encryption.

Exercise. Why would the message be less secure if Alice used the same r for each transmission?[3]

Of course, with such a small p as in the above example, it is easy to break the cipher. There are already couples (e.g. (17, 0)) coming from the same letter/choice of r that repeat themselves in the ciphertext. Instead one should use very large primes.

Example 8.11. Here is how we should have done it better. Alice writes her message in base 31 as follows: given

letter	g	o	d		m	a	d	e		b	e	e	r	…	.
encoded	6	14	3	26	12	0	3	4	26	1	4	4	17	…	28,

define

$$k = \left[6\right] + \left[14 \cdot 31\right] + \left[3 \cdot 31^2\right] + \left[26 \cdot 31^3\right] + \cdots + \left[28 \cdot 31^{58}\right]$$
$$= 9082899822880427373708169102831642254331248084798202311264158629033633834114233285040279.$$

Then choose a prime of this size, in fact $p = k + 94$ is the next[4] larger prime. The integer 2 is a primitive element. Bob creates the e by picking

$$d = 3478403603391729284720989742523730885804084076876107176787956281742848502614696973565952$$

and raising 2 to the d^{th} power modulo p. Alice chooses

$$r = 3170574170754971085221124838496442777170289756228641999147846701231454687394088022441984$$

[3]Solution: in this particular example, if she fixed $r = 19$ for each transmission, then *every* instance of g is encoded as $(12, 4)$, and similarly every other letter is encoded the same each time. So this example of ElGamal would just be doing a monoalphabetic substitution cipher, and could be cracked using frequency analysis. (Of course, using $p = 31$ is far too small and this is not secure even with random r’s each time).

[4]Of course, this not a good choice, it should be an arbitrary prime that doesn’t have any relation to k!

and can then encrypt m to the pair

$$c_1 = 3509287120782112568854030089705565540810206268218667157170062432397477315065347837322468\,,$$

$$c_2 = 3812463595284459446104706089365986796987480570156136884508868431035733570987927364537703\,.$$

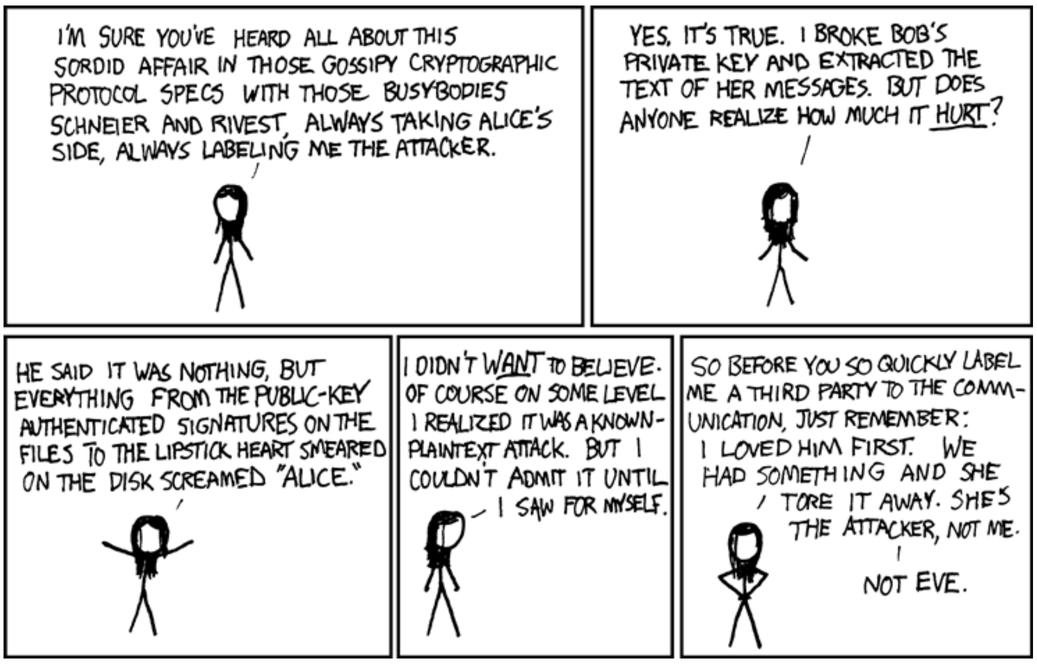
Now if Bob evaluates $c_2 \cdot c_1^{-d}$ he will recover the plaintext k. Writing it in base 31, he will recover the message

‘6, 14, 3, 26, 12, 0, 3, 4, . . . , 28’ = ‘`god made`$\cdots$.’

Attack 8.12 (ElGamal)**.** Frustrated at not cracking Alice and Bob’s messages when they used Diffie–Hellman and RSA, Eve tries to have a go at their ElGamal ciphertext. Alice’s encrypted messages (c_1, c_2) are assumed to be public, so if Eve has Bob’s secret key d, she can decrypt Alice’s messages.

Eve knows $e = a^d \pmod{p}$, so if she can solve the discrete logarithm problem, then she can obtain d and decrypt. However, as we know, this is computationally impossible on modern computers: so Eve is foiled again!

There does not seem any way of decrypting other than solving the discrete logarithm problem.


I'M SURE YOU'VE HEARD ALL ABOUT THIS SORDID AFFAIR IN THOSE GOSSIPY CRYPTOGRAPHIC PROTOCOL SPECS WITH THOSE BUSYBODIES SCHNEIER AND RIVEST, ALWAYS TAKING ALICE'S SIDE, ALWAYS LABELING ME THE ATTACKER.
YES, IT'S TRUE. I BROKE BOB'S PRIVATE KEY AND EXTRACTED THE TEXT OF HER MESSAGES. BUT DOES ANYONE REALIZE HOW MUCH IT HURT?
HE SAID IT WAS NOTHING, BUT EVERYTHING FROM THE PUBLIC-KEY AUTHENTICATED SIGNATURES ON THE FILES TO THE LIPSTICK HEART SMEARED ON THE DISK SCREAMED "ALICE."
I DIDN'T WANT TO BELIEVE. OF COURSE ON SOME LEVEL I REALIZED IT WAS A KNOWN-PLAINTEXT ATTACK. BUT I COULDN'T ADMIT IT UNTIL I SAW FOR MYSELF.
SO BEFORE YOU SO QUICKLY LABEL ME A THIRD PARTY TO THE COMMUNICATION, JUST REMEMBER: I LOVED HIM FIRST. WE HAD SOMETHING AND SHE TORE IT AWAY. SHE'S THE ATTACKER, NOT ME.
NOT EVE.

From `https://xkcd.com/177`

8.4. No-Key Protocol

This relatively simple encryption system was proposed by Adi Shamir around 1980. A version of it is the Massey–Omura cryptosystem, dating to 1982.

For motivation, suppose Alice would like to send a box containing some secret documents to Bob.

(1) Alice closes the box, puts a padlock on it and sends it to Bob.

(2) Bob can not open it, of course, but he can add a second padlock to the box and send it back to Alice.

(3) Alice can now remove her padlock and send the box once more to Bob. (She doesn't have to worry about security, because Bob's padlock is now guarding the documents).

(4) Bob can now remove his padlock and open the box.

The fascinating thing about this system is that Bob and Alice never needed to communicate their choice of keys to each other! This differs from the three systems we've seen already - in Diffie–Hellman, RSA and ElGamal one or both of Alice and Bob had to choose a private key, and then computed and published a public key e.

We can give a mathematical analogue, based on the discrete logarithm problem.

Algorithm 8.13 (No-key encryption)**.** ***Set-up:***

(1) Alice and Bob agree on a large prime p (the 'box').

(2) Alice chooses a 'padlock', i.e. an integer d_A coprime to $p-1$. Her 'key' is an inverse e_A of $d \pmod{p-1}$, so $d_A e_A \equiv 1 \pmod{p-1}$ (this is easy to compute using Euclid's algorithm).

(3) Bob also chooses a padlock and key d_B and e_B with $d_B e_B \equiv 1 \pmod{p-1}$.

Sending messages:

1. *Alice starts with the message k with $1 < k < p$, and adds her padlock by computing $c_1 = k^{d_A} \pmod p$. She sends c to Bob.*
2. *Bob cannot yet decrypt the message. Instead, he adds his padlock by computing*
$$c_2 = c_1^{d_B} = k^{d_A d_B} \pmod p$$
and sends c_2 back.
3. *Alice can now remove her padlock by computing*
$$c_3 = c_2^{e_A} = k^{d_A e_A d_B} \equiv k^{d_B} \pmod p$$
and sends c_3 back to Bob again.
4. *Bob can now finally recover the message by computing*
$$\begin{aligned} c_3^{e_B} &\equiv k^{d_B e_B} \\ &\equiv k \pmod p. \end{aligned}$$

In steps (3) and (4), we used Fermat's little theorem to show that

$$k^{d_A e_A} \equiv k^{d_B e_B} \equiv k^1 = k \pmod p.$$

Figure 8.4: No-key protocol: who knows what?

Known only to Alice	**Known to everybody** *Including any attackers!*	**Known only to Bob**
	large prime p	
d_A coprime to p $e_A = d_A^{-1} \pmod{p-1}$		d_B coprime to p $e_B = d_B^{-1} \pmod{p-1}$
message $1 < k < p$		
$\longrightarrow$	$c_1 = k^{d_A} \pmod{p}$	
	$c_2 = c_1^{d_B} \pmod{p}$	$\longleftarrow$
$\longrightarrow$	$c_3 = c_2^{e_A} \pmod{p}$	
		$k = c_3^{e_B} \pmod{p}$

Remarks.

- The security of this system relies on the difficulty of the discrete logarithm problem: Alice and Bob still have private keys d_A and d_B that can be cracked if one can solve discrete logarithms. If Eve has d_B, she can decrypt after intercepting Alice's second message.
- This algorithm does not need an exchange of public keys, but it does need twice as much data transmission as a system based on an exchange of keys.

Example 8.14. If Alice wants to send a secret message $k = 13$ to Bob.

- Alice and Bob fix the prime $p = 101$.
- Alice chooses a private key $d_A = 3$ and computes an inverse e_A using the Euclidean algorithm. Since $100 = 3 \cdot 33 + 1$ it follows that $1 = 100 - 3 \cdot 33 \Rightarrow 3^{-1} \equiv -33 \equiv 67 \pmod{100}$. Thus $e_A = 67$.
- Bob chooses the private key $d_B = 7$. To compute the inverse, he computes

$$\begin{tikzcd}
\mathbf{100} = \mathbf{7} \cdot 14 + \mathit{2} \arrow[d, squiggly] & \mathit{1} = \mathbf{7} - (3 \cdot (\mathbf{100} - 7 \cdot 14)) \;, \\
\mathbf{7} = \mathbf{2} \cdot 3 + \mathit{1} \arrow[r, squiggly] & \mathit{1} = \mathbf{7} - 3 \cdot \mathbf{2} \arrow[u, squiggly]
\end{tikzcd}$$

  and expanding we get $1 = 43 \cdot 7 - 3 \cdot 100$, whence $7^{-1} \equiv 43 \pmod{100}$. Hence $e_B = 43$.

We then compute the following steps:

(1) Alice computes $c = m^{d_A} = 13^3 \equiv 76 \pmod{101}$ and sends that to Bob.

(2) Bob computes $c' = c^{d_B} = 76^7 = 23 \pmod{101}$ and sends it back to Alice

(3) Alice now computes $c'' := (c')^{e_A} = 23^{67} \equiv 45 \pmod{101}$ and sends it over.

(4) Finally, Bob computes $m' := (c'')^{e_B} = 45^{43} \equiv 13$ and thus recovers the message.

Exercise 8.15. Assume that Alice wants to send a secret message $m = 42$ to Bob using the no-key protocol with prime $p = 47$ and their private keys d_A and d_B have inverses $e_A = 3$ and $e_B = 7$. Compute the private keys and describe each step of the communication in detail.

Solution 8.16. Alice and Bob have the private keys d_A and d_B satisfying

$$e_A d_A \equiv a_B d_B \equiv 1 \pmod{46}$$

so we need to invert 3 and 7 modulo 46. We can do this using the Euclidean algorithm:

$$46 = 3 \cdot 15 + 1$$

and hence $3 \cdot (-15) \equiv 1 \pmod{46}$ so $d_A \equiv 3^{-1} \equiv -15 \equiv 31$. Similarly

$$\begin{aligned} 46 &= 7 \cdot 6 + 4 \\ 7 &= 4 \cdot 1 + 3 \\ 4 &= 3 \cdot 1 + 1 \end{aligned}$$

so that $1 = 4 - 3 = 4 - (7 - 4) = 2 \cdot 4 - 7 = 2 \cdot (46 - 7 \cdot 6) - 7 = -13 \cdot 7 + 2 \cdot 46$ so $-13 \cdot 7 \equiv 1 \pmod{46}$. Hence $d_B \equiv -13 \equiv 33$.

We have following steps:

- Alice computes $c = m^{d_A} \pmod{47} \equiv 42^{31} \equiv 8 \pmod{47}$ and send this to Bob
- Bob now computes $c' = c^{d_B} \equiv 8^{33} \equiv 34 \pmod{47}$ and sends it back to Alice
- Alice now computes $c'' = c'^{e_A} \equiv 34^3 \equiv 12 \pmod{47}$ and sends it to Bob.
- Finally, Bob recovers the message by computing

$$m' = c''^{e_B} \equiv 12^7 \equiv 42 \pmod{47}.$$

(All exponentiations can be performed quickly using fast modular exponentiation).

Chapter 9

Defending from attacks

In the previous chapter, we saw a lot about our malicious eavesdropper Eve, who kept trying to intercept and decrypt Alice and Bob's secret messages. So far, she's been foiled every time, because she needed to either factor numbers or solve a discrete logarithm problem, which she's not able to do.

However, just because Eve can't read their messages doesn't mean she can't still cause havoc. So far, our emphasis has strongly been on **secrecy** of messages, ensuring that Eve cannot read private messages between Alice and Bob. Here are some other attacks Eve might try.

- Impersonate Alice and/or Bob by putting herself into the middle of their conversation.
- Tamper with the messages Alice and Bob are sending, even though she cannot read them.
- Plot twist: *Eve is Alice!* ...and she's trying to defraud Bob. After their encrypted exchanges, Bob sends Alice/Eve money... but then Alice/Eve denies ever interacting with Bob.

In cryptography, we also seek to thwart all of these possible attacks. We aim to ensure:

- **Authenticity**. Bob wants to be sure that the messages he's receiving really did come from Alice.
- **Integrity**. Alice and Bob want to ensure that nobody has tampered with/altered their messages.
- **Non-repudiation**. Bob wants to be able to prove to third parties that he received messages definitely from Alice (if, for example, Alice decides to deny ever sending them).

In this chapter we describe some attacks of this nature on the cryptosystems we've been discussing, and some methods of defending against these attacks.

9.1. Man-in-the-Middle attack

If Eve can get between Alice and Bob, she can corrupt their messages, inserting herself into the middle of their communications. A silly real-world example:

Example. • Imagine Alice has a private message for Bob:

```
Hi!  Would you like to go for dinner sometime?
                                  - From Alice
```

- Eve tells her 'Oh, I'm going to see him later, I'll pass it on for you.' Alice seals her message in an envelope, and gives it to Eve to pass on.
- Eve opens the envelope, reads the message, and seals a *new* message in a new envelope:

```
Hi!  My friend Eve is really great, you should go for dinner with her.
                                                     - From Alice
```

- Eve passes *her own*, corrupted, message to Bob, and pretends it's from Alice.
- Bob seals a reply to Alice in an envelope:

```
Oh, thanks for the recommendation, I'll have dinner with Eve tonight.
                                                     - From Bob
```

- Again Eve reads it, and sends Alice a corrupted 'reply from Bob':

```
Sorry, I'd never go for dinner with you.
                                                     - From Bob
```

- Alice and Bob are none the wiser that they have not been communicating with each other. Meanwhile Eve has totally hijacked their conversation to get what she wanted.

This is called a *man-in-the-middle attack*. In the world of digital communication, Eve can do this much more discreetly.

Example 9.1. Consider the following situation where Alice wants to exchange a key with Bob using Diffie–Hellman, and Eve wants to hijack them.

- Alice and Bob both create a pair of private and public keys: (e_A, d_A) and (e_B, d_B). Eve also creates a public and private key pair (e_E, d_E), and closely monitors their communication.

Now, when Alice and Bob exchange 'their' public keys:

- Eve intercepts Alice's public key e_A before it reaches Bob, and instead sends Bob *her own* key e_E. Similarly, she intercepts Bob's public key e_B, and sends Alice her own key e_E.

$$\begin{tikzcd}
\text{Alice} \arrow[r, bend left, "e_A"] & \text{Bob} \arrow[l, bend left, "e_B"]
\end{tikzcd}$$

Alice & Bob's perception

$$\begin{tikzcd}
\text{Alice} \arrow[r, bend left, "e_A"] & \text{Eve} \arrow[l, bend left, "e_E"] \arrow[r, bend left, "e_E"] & \text{Bob} \arrow[l, bend left, "e_B"]
\end{tikzcd}$$

What's actually happening

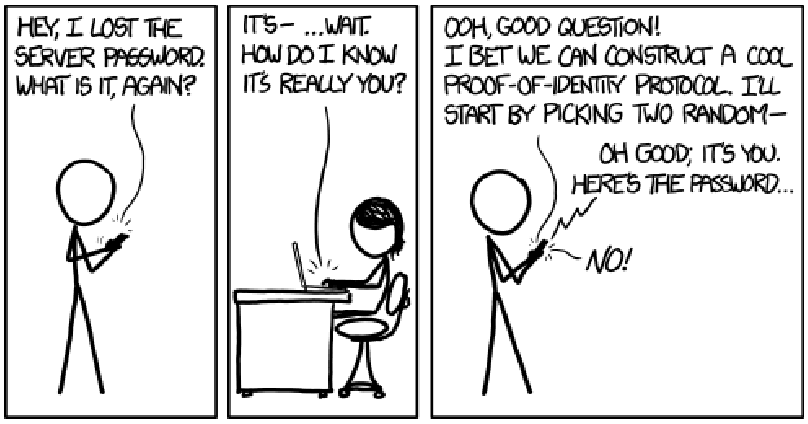
Eve continues the monitor/intercept the traffic between Alice and Bob.

- When Alice sends the message "Let's meet for dinner" she encrypts this (unknowingly) with *Eve's* public key. Thus *only Eve can decrypt the message* (as it requires Eve's private key).
- Eve can encrypt a different message "Meet Eve for dinner" using her private key and Bob's public key, and send it to Bob, who thinks it is coming from Alice.
- When Bob replies, similarly Eve can decrypt and send her own, seemingly encrypted, message.

All the protocols we have seen so far, and many more, are vulnerable to this type of attack!

9.2. A digital signature algorithm

How should we counter an attack like man-in-the-middle? We need to be sure that we can trust the source of the message, that is, that it is *really* coming from who we think it is.


HEY, I LOST THE SERVER PASSWORD. WHAT IS IT, AGAIN?
IT'S– ...WAIT. HOW DO I KNOW IT'S REALLY YOU?
OOH, GOOD QUESTION! I BET WE CAN CONSTRUCT A COOL PROOF-OF-IDENTITY PROTOCOL. I'LL START BY PICKING TWO RANDOM–
OH GOOD; IT'S YOU. HERE'S THE PASSWORD...
NO!

From `https://xkcd.com/1121`

In the real-world case Example 9.1, a simple thing Alice and Bob can do is *sign* their messages. If Eve wants to intercept and corrupt the messages, she needs to also fake their signatures.

To counter the digital version of Example 9.1, we can develop *digital signatures*; for example, Alice can add a digital signature to her public key, so that Bob can be sure that it really came from Alice, and not from Eve.

To that end, here is a standard signature algorithm due to ElGamal.

ElGamal signature scheme, attempt 1.

Registration. A *globally trusted third-party* (e.g. VeriSign) picks a large prime p and a primitive element $a \pmod p$.

Alice wants to register with VeriSign.

(1) Alice picks a private signing key $1 < d_A < p-1$.

(2) Alice computes her public signing key

$$e_A = a^{d_A} \pmod p.$$

(3) Alice publicly registers e_A at VeriSign.

There is now a publicly available *certificate*, at a trusted source, certifying that Alice's public signing key is e_A.

Signing procedure. Alice wants to sign an encrypted message c. (Note it's already been encrypted by some other method, with some other keys).

(1) Alice chooses a random integer u, coprime to $p-1$, and computes

$$r = a^u \bmod p.$$

(2) Next, using that u is invertible modulo $p-1$ and her private signing key d_A (from her registration with VeriSign), she computes

$$s = u^{-1} \cdot (c - d_A\, r) \pmod{p-1}.$$

(3) Alice's signature is (r, s). She sends Bob the triple (c, r, s).

Note that Alice has used her trusted private signing key, so for her given message c, *nobody else* can compute this value of s.

Verification. Bob receives the triple (c, r, s). He wants to verify this came from Alice.

(1) Bob goes to VeriSign and obtains Alice's trusted public signing key e_A.

(2) Bob computes the values

$$e_A^r \cdot r^s \pmod p$$

and

$$a^c \pmod p.$$

(3) If Alice genuinely produced the triple (c, r, s), then these two values must agree, as

$$\begin{aligned} e_A^r \cdot r^s &= (a^{d_A})^r \cdot (a^u)^{u^{-1}(c-d_A r)} \pmod p \\ &= a^{d_A r} \cdot a^{c-d_A r} = a^c \pmod p. \end{aligned}$$

In this case, the signature is valid.

Figure 9.1: ElGamal signatures: who knows what?

Fixed for all messages, registered with trusted authority:

Known only to Alice	**Known to everybody** *Including any attackers!*	**Known only to Bob**
	large prime p	
	primitive element $a \pmod p$	
Alice's private signing key d_A		
	Alice's public signing key	
$\longrightarrow$	$\longrightarrow e_A = a^{d_A} \pmod p$	

Changing with every separate message:

Known only to Alice	**Known to everybody** *Including any attackers!*	**Known only to Bob**
	message c	
	(that Alice wants to sign)	
random integer $u < p$		
(coprime to $p-1$)		
$\longrightarrow$	$\longrightarrow r = a^u \pmod p$	
$\longrightarrow$	$\longrightarrow s = u^{-1}(c - d_A r) \pmod{p-1}$	
	whether signature is valid or not	

Example. Alice has registered with the trusted authority NottsCheck, who use the prime 1427 and the primitive element 5. Her trusted public signing key is $e_A = 209$.

Bob receives the message $(c, r, s) = (101, 1134, 607)$. He wants to check it came from Alice. He computes that

$$e_A^r \cdot r^s = 209^{1134} \cdot 1134^{607} = 749 \pmod{1427},$$

and also that

$$a^c = 5^{101} = 749 \pmod{1427}.$$

Bingo: they agree, so Bob has verified the signature and believes this likely came from Alice.

How did Alice produce this? Her private signing key was 76 (and you can check that $5^{76} = 209 \pmod{1427}$, so this does give the right public signing key, as lodged at NottsCheck). She already knew that the encrypted message she wanted to send was $c = 101$. She randomly picked the integer $u = 913$, and then computed

$$r = 5^{913} = 1134 \pmod{1427}, \qquad s = u^{-1}(c - d_A r) = 913^{-1}(101 - 76 \cdot 1134) = 607 \pmod{1426}.$$

This gives her the triple $(101, 1134, 607)$ as claimed.

Example. Bob receives another message from Alice. This time he receives the triple $(c, r, s) = (132, 983, 210)$. He runs the same check:

$$e_A^r \cdot r^s = 209^{983} \cdot 983^{210} = 999 \pmod{1427},$$

and

$$a^c = 5^{132} = 194 \pmod{1427}.$$

This time they don't agree, so Bob thinks something's up. He asks Alice if she sent him a message, and finds that she didn't: Eve has been at work again!

Remark. Note here that the trusted third-party is playing a *crucial* role. If you don't rely on a trusted third-party, then this is just as susceptible to man-in-the-middle attacks, because Eve can just create and send her own private/public signing keys, and sign her corrupted form of the message. To be safe, Bob has to go and find Alice's public signing key by himself, from a trusted source (and he absolutely must *not* just accept it from Alice, or Eve can easily hijack the whole process).

How do we trust the third-party? They themselves will be registered with another trusted source, and so on; but eventually there needs to be a 'base' level of trust. This is where *root certificates* come in. There are a small number of trusted root certificates from which all digital verification is derived.

This sort of process is going on whenever you access an `https://` website. The website should be registered with a trusted certificate authority, and when you access the website, your browser performs a 'handshake' with the server, verifying each other's trusted public keys to ensure the safe exchange of data.

Needless to say, if one of these most-trusted-of-all root certificates got hacked, then chaos can ensue, as hackers can start to impersonate others online whilst still being apparently trustworthy. There are instances of this happening; look up the 2011 hack of DigiNotar, for instance.

HEY, I JUST GOT HOME FROM THE PARTY
THE ONE WITH THE IRC FOLKS?
YEAH.
HOW WAS IT?
GOT TOO DRUNK. I SCREWED UP, BAD.
WHAT HAPPENED?
THERE WAS A GIRL. NO IDEA WHO SHE WAS. DON'T EVEN KNOW HER NAME. I WAS TOO DRUNK TO CARE.
AND WHAT, YOU SLEPT WITH HER?
NO.
I SIGNED HER PUBLIC KEY.
SHIT, MAN.

From `https://xkcd.com/364`

Exercise. In this scheme, even more than in our previous ElGamal encryption scheme, it is *absolutely essential* to choose a different integer u for every new signature.

Suppose you do not, and you pick the same u twice, for two different messages c and c', giving triples (c, r, s) and (c', r', s'). Show that from this data, you can directly compute Alice's private signing key d_A. (Note that you don't even have to attack d_A: you can just *write down what d_A is*).

Remark. It's natural to ask why signatures are necessary. Why can't Alice and Bob just create their Diffie–Hellman keys, and then register *these* with the trusted authority? This foils man-in-the-middle; as Eve can't create new public keys and pretend they came from Alice of Bob.

However, there are many moving parts in the sending of secure message. If Alice and Bob have fixed their public and private Diffie–Hellman keys, then this means their shared secret is *always the same*. If they are using this shared secret as the symmetric key in a classical cipher system, this means they always cipher with the same key. As we saw in Chapter 6, this is a big flaw, opening up frequency analysis.

In reality, you want to choose different cipher keys for every message. This means you have to choose different keys in your PKC encryption; and you don't want to certify every different key you use. So it makes more sense to introduce signatures and have a fixed key used only for signing. Since this key is never used in classical ciphering, it's not possible to attack with frequency analysis.

9.3. Existential forgery

What security do we get from this ElGamal signing algorithm? At the very least, we've greatly improved the security/reduced the risk of man-in-the-middle attacks:

(a) Suppose the message c is fixed. Since only Alice knows her private signing key, *only she* could have produced the signature (r, s) attached to the message c.

(b) Similarly, suppose the signature (r, s) is fixed. To produce the signature (r, s), Alice used both the encrypted message c, and her private signing key. If Eve tried to tamper with c, then the signature (r, s) *will not be valid any more.*

So our system works, right? If Bob receives a valid triple (c, r, s), he can safely assume it came from Alice.

...well no, not quite...

In point (a) above, we fixed c and declared that (r, s) could only come from Alice. In (b), we fixed (r, s) and declared that c could only have come from Alice. But Eve can easily forge entirely new triples (c, r, s) with Alice's valid signature. In particular, here's a forging attack on our first ElGamal signature scheme. (This attack was known to ElGamal even when he first introduced the scheme).

Attack 9.2 (Eve's 'existential forgery')**.** Pick an integer $w < p - 1$. Let:

- $r = e_A a^w \pmod{p}$,
- $s = -r = -e_A a^w \pmod{p-1}$,
- $c = sw \pmod{p-1}$.

Then (c, r, s) is a valid triple for the ElGamal signature scheme (attempt 1). To see this, we can compute:

$$\begin{aligned} e_A^r \cdot r^s &= e_A^r \cdot (e_A a^w)^{-r} \pmod{p} \\ &= e_A^{r-r} \cdot a^{-wr} \pmod{p} \\ &= a^{ws} = a^c \pmod{p}. \end{aligned}$$

In other words, Eve can produce forgeries, namely valid triples. The downside to her attack is that this is an *existential forgery*: she doesn't have good control over what c is. So she can convince Bob that her forged message is genuine, but it's difficult to make a forged message that says what she wants it to.

For example, suppose Eve has a malicious message c that she's desperate to forge from Alice. To send c with a valid forged signature using the above method, she needs to find an integer w such that

$$c = -w \cdot e_A \cdot a^w \ (\mathrm{mod}\, p-1),$$

that is, she needs to solve a discrete-logarithm-style problem. In the real-world, however, the mere possibility you could do this is considered a serious and expoitable weakness, and it cannot be tolerated in a secure system. Thus our algorithm requires more work.

The following example hopefully highlights that these forgeries really are rather weak.

Example. Alice has a message for Bob. She wrote it out in plaintext, then converted it to a binary using ASCII. This binary string is then converted to an actual binary *number* k_A. Alice wants to send this using RSA and sign it using an ElGamal signature. Remember from our earlier examples that she likes to use NottsCheck, who use $p = 1427$ and $a = 5$, and that her public signing key is $e_A = 209$.

We proceed as follows:

Bob validating and decrypting Alice's valid message.

- Bob chooses $p = 10968523783746901$ and $q = 43142746595714191$. He chooses $d_B = 17$, and computes his RSA public key

$$N = 473212242131256750429190077972091,$$
$$e = 389704199402211396967698575244353.$$

- Alice then computes her encrypted message to be

$$c_A = k_A^e = 411834726986260043865421686266120 \ (\mathrm{mod}\, N).$$

- She now signs it using the ElGamal signature scheme, and sends the triple[a]

$$(c_A, r, s) = (411834726986260043865421686266120, 421, 528).$$

- Bob verifies that

$$e_A^r \cdot r^s = a^{c_A} = 1010 \ (\mathrm{mod}\, 1427),$$

  so the signature is valid. He proceeds to decrypt with his private RSA key, computing that

$$k_A = c_A^d = c_A^{17} = 114784820031265 \ (\mathrm{mod}\, N).$$

- He converts this to binary, obtaining

```
011010000110010101101100011011000110111100100001
```

  (noting that the number of digits should be a multiple of 8, so I've padded the front with `0` to ensure this). Converting this back to text via ASCII gives...

```
hello!
```

  ...a message that it *vitally* important Alice kept top secret.

[a]Here she chose the random integer $u = 813$.

Eve attempting and failing to send a specific malicious message.

Now Eve wants a slice of the action. She wants to send Bob the message `ihateyou`, and pretend that really it was Alice who sent it. Via ASCII, in binary this is

```
0110100101101000011000010111010001100101011110010110111101110101,
```

which corresponds to $k_E = 7595427924106899317$. She can encrypt this using Bob's public RSA key to obtain

$$c_E = k_E^e = 109319202871340767823440561030363 \pmod{N}.$$

She can send this to Bob as it is... But there's no signature, so Bob will rightly be suspicious of it. She can try and forge Alice's signature, but to do this she needs to choose a value of w with

$$109319202871340767823440561030363 = -w \cdot e_A \cdot a^w = w \cdot 209 \cdot 5^w \pmod{1426}.$$

This looks rather hard, even for this small value of p (and certainly it seems impossible for cryptographic p).

Eve sending a valid forgery that ends up being meaningless.

Frustrated, and wanting to cause as much havoc as possible, Eve gives up on sending her specific message and instead forges a signature on *some* message. She picks $w = 627$, which via the existential forgery algorithm yields the triple

$$\begin{aligned} r = a^w \cdot e_A = 5^{627} \cdot 209 = 591 \pmod{1427} \\ s = -r = 835 \pmod{1426}, \\ c = ws = 627 \cdot 835 = 203 \pmod{1426}. \end{aligned}$$

She now has a valid signed triple (c, r, s). She can modify c by any multiple of 1426 and still have a valid triple, but it doesn't really help her find a useful malicious message. At random she settles on

$$\begin{aligned} c &= 203 + 319202871340767823440\mathbf{5} \times 1426 \\ &= 4551832945319349162261733, \end{aligned}$$

and sends the triple (c, r, s) to Bob.

Bob receives this, and verifies the signature. He finds

$$a^c = e_A^r \cdot r^s = 950 \pmod{1427},$$

so is happy this is valid and moves on to decrypting. He computes

$$\begin{aligned} k &= 45518329453193491622617 33^{17} \\ &= 250469233794958537961678558188448 \pmod{N}, \end{aligned}$$

and converts this to binary. Then he uses ASCII to convert back to text. He finds the decrypted message

```
Y]¾<w°K¯ëÖ³Ó
```

Oof. Bob's pretty sure this isn't a valid message – it's more like the obvious spam emails you get promising 'You've been chosen for this *unbelieveable* prize!', and Bob just ignores it. This is nothing like the hurt that Eve was intending to cause with her man-in-the-middle attack.

9.4. Hash functions

As ElGamal observed in his original paper, one can remove even the possibility of existential forgeries by introducing *hash functions*.

The main problem with the digital signature scheme above was that if you pick r and s cleverly, then you can determine c, because s depends on c itself: theoretically you could recover c from r and s by computing

$$c = us + d_A r \pmod{p-1}.$$

This weakness was enough that Eve could produce her existential forgeries, even without knowing the specific values of u and d_A that Alice is using. We want to kill this dependence.

Definition 9.3. An n-bit *hash function* is a function

$$H : \{\text{all strings of text}\} \to V_n,$$

that takes text of arbitrary lengths, and outputs a binary string of length n.

Note here that the source of H is infinite, whilst the target is finite. This means there will be 'collisions', i.e. there will be different strings m, m' with $H(m) = H(m')$.

Example. The most simple hash function is the 1-bit 'check-sum' function

$$H_{\text{cs}} : \{\text{finite binary strings}\} \to \{0, 1\}$$

given by summing all the digits (mod 2). We already saw this kind of idea way back in §1.2, where we used to as an error-detection scheme. In particular, you could consider the paper tape code as the process

- Alice starts with the message $m \in V_7$.
- Alice sends the pair $[m, H_{\text{cs}}(m)]$.
- Bob receives the pair $[R, S]$.
- Bob checks the pair is valid by computing $H_{\text{cs}}(R)$ and checking it is equal to S.

In cryptography, we can do something similar. Suppose Alice has a message m for Bob. She can choose a large hash function H, and send him the pair $(m, H(m))$. If Bob receives the pair (R, S), he can check it was valid by comparing $H(R)$ and S; they should be equal.

To be truly useful, the hash function H needs to satisfy some basic properties:

(1) **You must not be able to invert** H in any way; given $H(m)$, you should not be able to find m, or indeed any other m' such that $H(m') = m$.

In other words, cryptography hash functions should be one-way functions. *To achieve this, we need to take n relatively large, e.g. $n = 256$.*

(2) **Small changes in** m should correspond to **completely different values of** $H(m)$. You should not be able to modify m and $H(m)$ in the same consistent manner.

To achieve this, our function H needs to be mathematically very complicated.

(3) Given m, we should be able to **compute** $H(m)$ **efficiently** (i.e. in polynomial-time).

Thus our function needs to be built from efficient mathematical operations.

In practice, hash functions are built in a similar way to AES-256: lots of repeated efficient steps that are not invertible, scrambling the string.

To write down $H(m)$, it's usually more convenient to convert the image binary string to hexadecimal (using the numbers 0-9 and the letters `a–f`), giving it a more compressed form; since there are $16 = 2^4$ hexadecimal digits, a hexadecimal string of of length r corresponds to a binary string of length $4r$.

Here's an example of the 128-bit hash function MD5 in practice:

$$m = \texttt{I know that the human being and the fish can coexist.}$$
$$H(m) = \texttt{ce5ee19205f9ea24f94f1037c83e3ca7}$$

Changing just one letter completely changes the output:

$$m' = \texttt{I knew that the human being and the fish can coexist.}$$
$$H(m') = a01c29091ee0c3d06d5c86d7e0895ade$$

Remark 9.4. When you create a password P for a website, it should never be stored as plaintext (unless the website is very badly designed). Instead, the database should store the hashed version $H(P)$. When you try and log in, you enter P; the server computes $H(P)$, and compares it with the database entry. If they agree, it logs you in.

This way, if a hacker obtains access to the database, they only get the hashed value $H(P)$, never your actual password. Because hash functions cannot be inverted, they have no way to obtain your plaintext password.

So, all major websites know about this, right? This is security 101. You'd never find a large social media website doing something as stupid as storing passwords in plaintext, right? Right?...

`https://www.theverge.com/2019/4/18/18485599/facebook-instagram-passwords-plain-text-millions-users`

...oh.

Incidentally, this is a major reason why you should use different passwords for different websites. If I use the same password for every website, and just *one* of those websites – let's say some obscure online shop you used once back in 2014 – stored your email and password in plaintext, then didn't update their security and got hacked, then your plaintext password and email are available to anybody who purchases that hack. They can try it on any website they like. Lessons from this:

- Use a password manager and generate new long random passwords for every account you ever make. Even the ones that seem unimportant.
- Turn on 2-factor authentication at every opportunity. Then even if your plaintext password is hacked, you still have some defences.

Remark 9.5. Here's a curiosity. Since there are infinitely many possible strings, and only finitely many possible values of H, there will always be collisions; that is, there will likely be an infinite number of strings P' of text such that $H(P') = H(P)$. Any of these, if inputted alongside your username, will give a valid password. In practice, it's extremely likely that your actual password P will be by far the shortest of these 'passwords', but this is another consideration when designing hash functions; it would be disastrous if you chose the secure 20-digit password `1kJ4123$H2%ahs90HaD!`, and it so happened that

$$H(\texttt{1kJ4123\$H2\%ahs90HaD!}) = H(\texttt{123}),$$

in which case – through no fault of your own – you could unlock your account with the password `123`. Oops.

9.5. A better digital signature algorithm

We now indicate how hash functions can be used to kill Eve's existential forgery attacks on ElGamal signatures. This improvement was suggested in ElGamal's original paper *A public key cryptosystem and a signature scheme based on discrete logarithms* (1985), in which he also proposed the ElGamal PKC system from §8.3.

In the first attempt, recall that we picked r, and then defined s so that

$$c = us + d_A r \pmod{p-1}.$$

In this way, theoretically c could be recovered from r and s. For our second attempt, we use a hash function instead; having picked r, we define s so that

$$H(c) = us + d_A r \pmod{p-1}.$$

Now if we have r and s, we can theoretically recover $H(c)$. But from $H(c)$, we cannot recover c (by the properties of hash functions). So Eve's clever trick to forge signatures now falls apart: if she picks r and s, she needs to invert a hash function to obtain c; if she picks c, she needs Alice's private signing key to construct a valid signature (r, s).

In full, the algorithm proceeds as follows:

ElGamal signature scheme, attempt 2.

Registration. Alice chooses her private key d_A and registers her private key e_A exactly as in attempt 1 (see page 144).

Signing procedure. Alice wants to sign an encrypted message c.

(1) Alice chooses a random integer u, coprime to $p-1$, and computes

$$r = a^u \bmod p.$$

(2) **Alice uses a hash function to compute $H(c)$.**

(3) Next, using that u is invertible modulo $p-1$ and her private signing key d_A (from her registration with VeriSign), she computes

$$s = u^{-1} \cdot (H(c) - d_A\, r) \pmod{p-1}.$$

(4) Alice's signature is (r, s). She sends Bob the triple (c, r, s).

Verification. Bob receives the triple (c, r, s). He wants to verify this came from Alice.

(1) Bob goes to VeriSign and obtains Alice's trusted public signing key e_A.

(2) Bob computes the values

$$e_A^r \cdot r^s \pmod{p}$$

and

$$a^{H(c)} \pmod{p}.$$

(3) If Alice genuinely produced the triple (c, r, s), then these two values must agree, by the same argument as before. In this case, the signature is valid.

This signing algorithm now checks off all three of the other aims of cryptography listed at the start of this chapter.

- **Authentication**. Since only Alice knows her private signing key, only she could have produced the signature (r, s) attached to the message c. So if the signature is valid, Bob knows the message genuinely came from Alice.

- **Integrity**. To produce the signature (r, s), Alice used both the encrypted message c, and her private signing key. If Eve tried to tamper with c, then the signature (r, s) *will not be valid any more*. So she also needs to tamper with the signature – but to produce a valid signature, she needs to know Alice's private signing key. So unless she can break the discrete logarithm problem, *Eve can't tamper with messages.*

- **Non-repudiation**. Since the verification step only requires public information, *anybody* can verify that the triple (c, r, s) came from Alice, even if they can't decrypt what c actually says. Thus Bob can prove to to others that Alice sent the message, should she try and deny it; and he doesn't even have to reveal what was in the message. (In particular, this kind of thing is going on with call log histories; authorities can prove there *was* a message, even though they can't read it).

This system is considered totally secure, although others are more commonly used.

Bonus: Homomorphic attack

To understand the necessity of signatures for **integrity** of messages, we describe one more form of attack that can be carried out during Man-in-the-Middle: a **homomorphic attack**. For certain cryptosystems, this allows an attacker to alter messages even without knowing what they say.

A cipher is said to be *homomorphic* if its encryption function **Enc** and decryption function **Dec** satisfy the following property:

$$\mathbf{Dec}(\mathbf{Enc}(m_1) \cdot \mathbf{Enc}(m_2)) = m_1 \cdot m_2. \tag{9.1}$$

If a cipher is homomorphic then an eavesdropper can change the message the recipient receives in a predictable way if they can modify the encrypted text. One way to protect against this type of attack is to use a signature.

Example 9.6 (Homomorphic attack on ElGamal)**.** ElGamal is clearly susceptible to this type of attack. Say I want to send $m = 123456789$. I then send a certain (c_1, c_2). Now Eve intercepts the transmission. Although she can not read it, she can still alter it before sending it forward.

- If she replaces c_2 by $1000c_2$ then the receiver, will compute

$$(1000 \cdot c_2) \cdot c_1^{-1} \equiv 1000 \cdot m = 123456789000.$$

- By just replacing c_2 we essentially used $k = 0$ but (9.1) holds for any value of k.

Example 9.7 (Homomorphic attack on RSA)**.** RSA is also susceptible to this type of attack: assume I again want to send the same $m = 123456789$ which is now represented by a certain c. If Eve now replaces c by $1000^e \cdot c$ the receiver will decrypt the message to

$$1000^{ed} \cdot c^d \equiv 1000 \cdot m \equiv 123456789000 \pmod N$$

which is again 1000 times the original message.

In particular, these cryptosystems would be very bad methods of sending your bank balance – unless you secure the integrity of your message with a digital signature.

9.6. The major flaw with ‘perfect’ cryptosystems

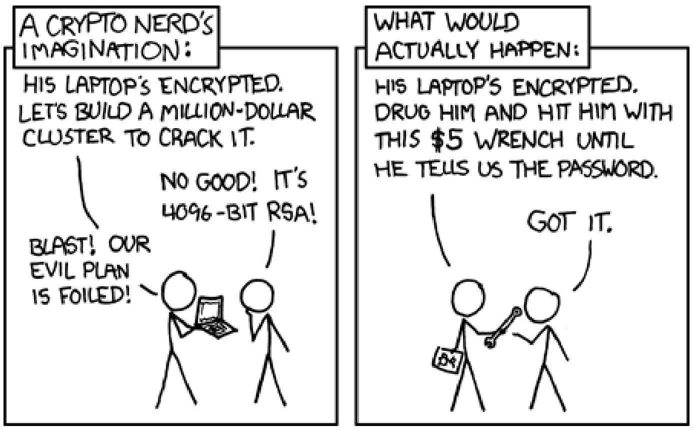
No matter how *theoretically* perfect we make our cryptographic systems, there is always one huge flaw...

...they are being implemented and used by humans, who can and do make mistakes.

Whilst most encryption schemes if used correctly are *mathematically secure*, the phrase ‘if used correctly’ is doing a lot of heavy lifting. A system is only as secure as the vigilence of those implementing and using it. For example:

- Attackers can break short (< 13) or non-random passwords. Use a strong master password and a password manager and 2-factor authentication if possible (and definitely don’t reuse passwords).

- Most computer systems can be hacked with physical access.
- If someone *really* wants to hack a system, they usually can, by exploiting some vulnerability in the implementation, or by...
- *Social engineering*: hacking using human vulnerabilities; for example, spear-phishing emails ('Dear [X], Your log-in details have expired: please go to this link and enter your username and password...')
- Hacking in bulk: companies can get lazy with security, meaning a hacker can obtain access to huge banks of data. A particularly egregious example is storing *plaintext* (non-hashed) usernames and passwords in a database. Any hacker that gets access to this can use passwords from data breaches and try them on all possible servers/accounts. *If you reuse a password, then one data breach then breaches all accounts with the same password!*
- Malware, trojans, worms, botnets etc. collect data.


A CRYPTO NERD'S IMAGINATION:
HIS LAPTOP'S ENCRYPTED. LET'S BUILD A MILLION-DOLLAR CLUSTER TO CRACK IT.
NO GOOD! IT'S 4096-BIT RSA!
BLAST! OUR EVIL PLAN IS FOILED!
WHAT WOULD ACTUALLY HAPPEN:
HIS LAPTOP'S ENCRYPTED. DRUG HIM AND HIT HIM WITH THIS $5 WRENCH UNTIL HE TELLS US THE PASSWORD.
GOT IT.

From `https://xkcd.com/538`

Don't be the low-hanging fruit for hackers! It's probably safest to assume that the only *real* security for your data is that nobody wants to hack you. And, if you happen to be a prominent politician or celebrity... then get your security in order!

Index