

# Computer Science Field Guide

v2.8.1

Print Edition

## Sponsors

Funding for this guide has been generously provided by these sponsors



Philanthropies



Produced by the [CS Education Research Group, University of Canterbury](#), New Zealand, and [by many others](#).

The Computer Science Field Guide uses a [Creative Commons \(CC BY-NC-SA 4.0\) license](#).

# 1. Introduction

Watch the video online at <https://www.youtube.com/embed/v5yeq5u2RMI?rel=0>

## 1.1. What's the big picture?

Why is it that people have a love-hate relationship with computers? Why are some people so fanatical about particular types of computers, while others have been so angry at digital devices that they have been physically violent with them? And what does this have to do with computer science? And what is computer science anyway?

I'm glad you asked! Put simply, computer science is about tools and techniques for designing and building applications that are very fast, have great interfaces, are reliable, secure, helpful --- even fun.

A lot of people confuse computer science with programming. It has been said that "computer science is no more about programming than astronomy is about telescopes" ([Mike Fellows](#)). Programming is the tool that computer scientists use to bring great ideas to life, but just knowing how to give programmed instructions to a computer isn't enough to create software that delights and empowers people.

For example, computers can perform billions of operations every second, and yet people often complain that they are too slow. Humans can perceive delays of about one tenth of a second, and if your program takes longer than that to respond it will be regarded as sluggish, jerky or frustrating. You've got well under a second to delight the user! If you are searching millions of items of data, or displaying millions of pixels (megapixels), you can't afford to do things the wrong way, and you can't just tell your users that they should buy a faster computer ... they'll probably just buy someone else's faster software instead!

Here's some advice from Fred Wilson, who has invested in many high profile tech companies:

First and foremost, we believe that speed is more than a feature. Speed is the most important feature. If your application is slow, people won't use it. I see this more with mainstream users than I do with power users. I think that power users sometimes have a bit of sympathetic eye to the challenges of building really fast web apps, and maybe they're willing to live with it, but when I look at my wife and kids, they're my

mainstream view of the world. If something is slow, they're just gone. ... speed is more than a feature. It's a requirement.

– Fred Wilson ([Source](#))

A key theme in computer science is working out how to make things run fast, especially if you want to be able to sell your software to the large market of people using old-generation smartphones, or run it in a data centre where you pay by the minute for computing time. You can't just tell your customers to buy a faster device --- you need to deliver efficient software.

## 1.2. Beyond speed

Computer science isn't just about speed. Try using the following two calculators to make a simple calculation. They both have the same functionality (they can do the same calculations), but which is nicer to use? Why?

(This book has many interactives like this. If the calculators don't work properly, you may need to use a more recent browser. The interactive material in this book works in most recent browsers; Google Chrome is a particularly safe bet.)

Use the interactive online at <http://www.csfieldguide.org.nz/en/interactives/awful-calculator/index.html>

The second calculator above is slower, and that can be frustrating. But it has a fancier interface --- buttons expand when you point to them to highlight what you're doing. Does this make it easier to use? Did you have problems because the "C" and "=" keys are so close?

How interfaces work is a core part of computer science. The aesthetics -- images and layout -- are important, but what's much more crucial is the psychology of how people interact. For example, suppose the "OK" and "Cancel" buttons in dialogue boxes were occasionally reversed. You would always need to check carefully before clicking on one of them, instead of using the instinctive moves you've made countless times before. There are some very simple principles based on how people think and behave that you can take advantage of to design systems that people love.

Making software that can scale up is another important theme. Imagine you've built a web interface and have attracted thousands of customers. Everything goes well until your site goes viral overnight, and you suddenly have millions of customers. If the system becomes bogged down, people will become frustrated waiting for a response, and tomorrow you will have no customers --- they'll all have moved on to someone else's system. But if your

programs are designed so they can scale up to work with such large amounts of data your main problem will be dealing with offers to buy your company!

Some of these problems can be solved by buying more equipment, but that can be an expensive and wasteful option (not just for cost, but because of the impact on the environment, including the wasted power used to do the processing inefficiently). With mobile computing it's even more important to keep things lean and efficient --- heavy duty programs chew up valuable battery life, and processing and memory must be used sparingly as these affect the size, weight and even heat dissipation of devices.

If your system is successful and becomes really popular, pretty soon people will be trying to hack into it to steal valuable customer data or passwords. How can you design systems so that you know they are secure from such attacks and your customers can trust you with their personal information or business transactions?

All these questions and more are addressed by the field of computer science. The purpose of this guide is to introduce you to those ideas so that you have a better idea of whether this field is for you. It is aimed at high-school level, and is intended to bring you to the point where you have a good overview of the field, and are well prepared for further in-depth study to become an expert.

We've broken computer science up into a whole lot of topics that you'll often find in curricula around the world, such as algorithms, human-computer interaction, compression, cryptography, computer graphics, and artificial intelligence. The reality is that all these topics interact, so be on the lookout for the connections.

This guide isn't a list of facts for you to memorise, or to copy and paste into projects! It is mainly a guide to things you can do --- experiences that will engage you with the topics. In fact, we won't go through all the topics in great detail, but will give you references to websites and books that explain things thoroughly. The idea of this guide is to give you enough background to understand the topics, and to do something meaningful with them.

## 1.3. Programming

And what about programming? You can get through this whole guide without doing any programming, although we'll suggest exercises. Ultimately, however, all the concepts here are reflected in programs that people write. If you want to learn programming there are many excellent courses available. It takes time and practice, and is well worth doing in parallel with working through the topics in this guide. There are a number of free online systems and books that you can use to teach yourself programming. A list of options for all ages learning to program is available at [www.code.org](http://www.code.org), where there is also a popular video of some well-known high-fliers in computing that is good to show classes.

Programming is just one of the skills you'll need to be a computer scientist. In this book you'll be exercising many other skills --- maths, psychology, and communication are important ones.

## 1.4. How to use this guide

This guide is intended to support a variety of curricula, and teacher guides will become available for using it in different contexts. For students, we've designed most chapters so that they can stand alone; the few that build on previous chapters explain at the outset what preparation you need (the most useful general preparation is the chapter on data representation, because everything on a computer is stored using binary numbers and so they have an important role in many areas of computer science.)

Each chapter begins with a section about the "big picture" --- why the topic is useful for understanding and designing computer systems, and what can be achieved using the main ideas in the chapter. You'll then be introduced to key ideas and applications of the topic through examples, and wherever possible we'll have interactive activities that enable you to work with the ideas first hand. Sometimes these will be simplified versions of the full sized problems that computer scientists need to deal with -- our intention is for you to actually interact with the ideas, not just read about them. Make sure you give them a go!

We finish each chapter by talking about the "whole story," giving hints about parts of the topic that we omitted because we didn't want to make the chapter too overwhelming. There will be pointers for further reading, but be warned that some of it might be quite deep, and require advanced math or programming skills.

If you are doing this for formal study, you'll end up having to do some sort of assessment. The curriculum guides provide ideas for projects and activities that could be used for this.

## 1.5. About this guide

This guide is free for you to copy, share and even modify. It is currently available online, and as a downloadable PDF file (although it's much better viewed in the other formats because you can watch the videos and use the interactive activities). The source material (all raw text, images, videos and interactive programs) is available through the "Contribute on GitHub" link.

This guide is licenced under a [Creative Commons Attribution-NonCommercial-ShareAlike licence](#), which means that you are welcome to take copies and modify them. If you do make improvements, we ask that you share those, and acknowledge this guide by linking

back to our web site. You can give away the guide (or any derivatives), and you can use it for teaching, but you're not allowed to sell it directly for profit.

Production of the guide was partially funded by a generous grant from Google Inc., and supported by the University of Canterbury. Of course, we welcome donations to support further work on the guide.

## 1.6. Further reading

Each chapter gives suggestions for further reading for that particular topic. There are also plenty of general books and websites about computer science that you might want to read to keep your view of the topic broad.

Books that we particularly recommend include:

- Algorithmics, by David Harel
- [Computational fairy tales](#), by Jeremy Kubica
- Algorithmic adventures: from knowledge to magic, by Juraj Hromkovic
- The Turing Omnibus, by A.K. Dewdney

Wikipedia has a fairly extensive [entry on computer science](#).

The AQA Computing A2 book(s), by Sylvia Langfield and Kevin Bond, give a more detailed account of many of these topics.

There are also some excellent general web sites about Computer Science, many of which we've referenced in other chapters:

- [Computer Science For Fun](#) --- a very readable collection of short articles about practical applications of topics in computer science
- [Babbage's bag](#) is an excellent collection of technical articles on many topics in computing.
- [CS Bytes](#) has up-to-date articles about applications of computer science.
- [Thriving in our digital world](#) has some excellent information and interactive material on topics from computer science.
- [The Virginia tech online interactive modules for teaching computer science](#) cover a range of relevant topics.
- [CS animated](#) has interactive activities on computer science.
- [CS for All](#)

# 2. Algorithms

Watch the video online at <https://www.youtube.com/embed/FOwCCvHEfY0>

## 2.1. What's the big picture?

Every computer device you have ever used, from your school computers to your calculator, has been using algorithms to tell it how to do whatever it was doing. Algorithms are a very important topic in Computer Science because they help software developers create efficient and error free programs. The most important thing to remember about algorithms is that there can be many different algorithms for the same problem, but some are much better than others!

Use the interactive online at <http://www.csfieldguide.org.nz/en/interactives/sorting-algorithm-comparison/index.html>

Computers are incredibly fast at manipulating, moving and looking through data. However the amount of data computers use is often so large that it doesn't matter how fast the computer is, it will take it far too long to examine every single piece of data (companies like Google, Facebook and Twitter routinely process billions of things per day, and in some cases, per minute!) This is where algorithms come in. If a computer is given a better algorithm to process the data then it doesn't matter how much information it has to look through, it will still be able to do it in a reasonable amount of time.

If you have read through the Introduction chapter you may remember that the speed of an application on a computer makes a big difference to a human using it. If an application you create is too slow, people will get frustrated with it and won't use it. It doesn't matter if your software is amazing, if it takes too long they will simply give up and try something else!

### 2.1.1. Algorithms, Programs and Informal Instructions

At this stage you might be thinking that algorithms and computer programs kind of sound like the same thing, but they are actually two very distinct concepts. They are each different ways of describing how to do something, but at different levels of precision:

Often you can get away with describing a process just using some sort of informal instructions using natural language; for example, an informal instruction in a non computing context might be "please get me a glass of water". A human can understand what this means and can figure out how to accomplish this task by thinking, but a computer would have no idea how to do this!

Use the interactive online at <http://www.csfieldguide.org.nz/en/interactives/high-score-boxes/index.html>

An example in a computational context might be if you wanted to find a high score in a table of scores: go through each score keeping track of the largest so far. Informal instructions like this aren't precise; there's no way that a computer could follow those instructions exactly, but a human could probably get the general idea of what you mean if they know what you're trying to achieve. This sort of description is only useful for quickly giving another human the general idea of what you mean, and even then there's a risk that they won't properly understand it.

In contrast, an **algorithm** is a step by step process that describes how to solve a problem and/or complete a task, which will always give the correct result. For our previous non-computing example, the algorithm might be 1) Go to the kitchen. 2) Pick up a glass. 3) Turn on the tap. 4) Put the glass under the running water and remove it once it is almost full. 5) Turn off the tap. 6) Take the glass back to the person who gave the instruction. A human could follow these instructions easily, but it's still using general English language rather than a strict list of computer instructions.

Algorithms are often expressed using a loosely defined format called **pseudo-code**, which matches a programming language fairly closely, but leaves out details that could easily be added later by a programmer. Pseudocode doesn't have strict rules about the sorts of commands you can use, but it's halfway between an informal instruction and a specific computer program.

With the high score problem, the algorithm might be written in pseudo-code like this:

```

if the table is empty
    display that there is no high score, and quit
otherwise, note the first score in the table
for each of the other scores in the table,
    if that score is larger than the one noted,
        replace the noted one with the current score
display the currently noted score

```

Algorithms are more precise than informal instructions and do not require any insight to follow; they are still not precise enough for a computer to follow in the form they are written, but are precise enough for a human to know exactly what you mean, so they can then work out how to implement your algorithm, either doing it themselves, or writing a computer program to do it. The other important thing with this level of precision is that we can often make a good estimate of how fast it will be. For the high score problem above, if the score table gets twice as big, the algorithm will take about twice as long. If the table could be very big (perhaps we're tracking millions of games and serving up the high score many times each second), that might already be enough to tell us that we need a better algorithm to track high scores regardless of which language it's going to be programmed in; or if the table only ever has 10 scores in it, then we know that the program is only going to do a few dozen operations, and is bound to be really fast even on a slow computer.

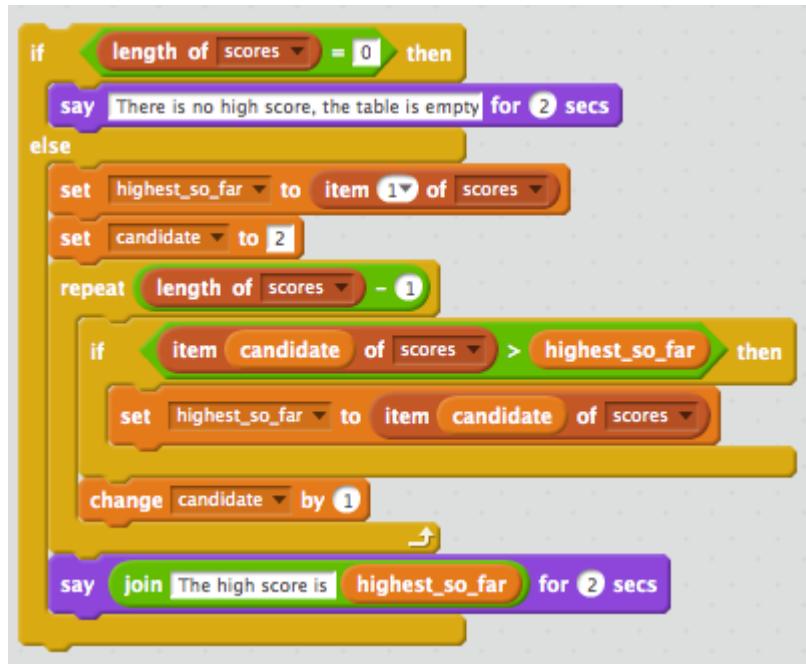
The most precise way of giving a set of instructions is in the form of a [program](#), which is a specific implementation of an algorithm, written in a specific programming language, with a very specific result for any particular input. This is the most precise of these three descriptions and computers are able to follow and understand these.

For the example with getting a drink, we might program a robot to do that; it would be written in some programming language that the robot's computer can run, and would tell the robot exactly how to retrieve a glass of water and bring it back to the person who asked for the water.

With the high-score problem, it would be written in a particular language; even in a particular language there are lots of choices about how to write it, but here's one particular way of working out a high score (don't worry too much about the detail of the program if the language isn't familiar; the main point is that you could give it to a computer that runs Python, and it would follow the instructions exactly):

```
def find_high_score(scores):
    if len(scores) == 0:
        print("No high score, table is empty")
        return -1
    else:
        highest_so_far = scores[0]
        for score in scores[1:]:
            if score > highest_so_far:
                highest_so_far = score
        return highest_so_far
```

But here's another program that implements exactly the same algorithm, this time in the Scratch language.



Both of the above programs are the same algorithm. In this chapter we'll look in more detail about what an algorithm is, and why they are such a fundamental idea in computer science. Because algorithms exist even if they aren't turned in to programs, we won't need to look at programs at all for this topic, unless you particularly want to.

## 2.1.2. Algorithm cost

When Computer Scientists are comparing algorithms they often talk about the 'cost' of an algorithm. The cost of an algorithm can be interpreted in several different ways, but it is always related to how well an algorithm performs based on the size of its input,  $n$ . In this chapter we will talk about the cost of an algorithm as either the time it takes a program (which performs the algorithm) to complete, or the number of steps that the algorithm makes before it finishes.

For example, one way of expressing the cost of the high score algorithm above would be to observe that for a table of 10 values, it does about 10 sets of operations to find the best score, whereas for a table of 20 scores, it would do about twice as many operations. In general the number of operations for a table of  $n$  items will be proportional to  $n$ . Not all algorithms take double the time for double the input; some take a lot more than double, while others take a lot less. That's worth knowing in advance because we usually need our programs to scale up well; in the case of the high scores, if you're running a game that suddenly becomes popular, you want to know in advance that the high score algorithm will be fast enough if you get more scores to check.

### Extra For Experts: Algorithm complexity

The formal term for working out the cost of an algorithm is [algorithm analysis](#), and we often refer to the cost as the algorithm's *complexity*. The most common complexity is the "time complexity" (a rough idea of how long it takes to run), but often the "space complexity" is of interest - how much memory or disk space will the algorithm use up when it's running?

There's more about how the cost of an algorithm is described in industry, using a widely agreed on convention called 'Big-O Notation', in the "[The whole story!](#)" section at the end of this chapter.

The amount of time a program which performs the algorithm takes to complete may seem like the simplest cost we could look at, but this can actually be affected by a lot of different things, like the speed of the computer being used, or the programming language the program has been written in. This means that if the time the program takes to complete is used to measure the cost of an algorithm it is important to use the same program and the same computer (or another computer with the same speed) for testing the algorithm with different numbers of inputs.

The number of operations (such as comparisons of data items) that an algorithm makes however will not change depending on the speed of a computer, or the programming language the program using the algorithm is written in. Some algorithms will always make the same number of comparisons for a certain input size, while others might vary.

### 2.1.3. Algorithm Correctness

If we develop or are given an algorithm to solve a problem, how do we know that it works? Sometimes we create test cases to verify the algorithm produces correct output for specific input values. While this is a useful practice and can help verify that we are on the right track, it is not enough to show that our algorithm is correct. The old adage "even a broken watch is correct twice a day" is a good analogy. Even an algorithm that is correct for two test cases might be incorrect for every other input. A computer scientist must reason formally or mathematically about an algorithm to show its correctness. Typically this is done by classifying ranges of input values and showing that algorithm produces expected results for boundary values of the range and all values in between.

Correctness is particularly important when comparing two algorithms that solve the same problem. If one algorithm is very fast to complete but produces incorrect results some of the time it may be far less useful than a correct algorithm that is slower. Correctness is

also important when using an algorithm as the building block for another algorithm. Here is an algorithm for assigning animals as pets to people on a waitlist:

1. Search for the person who is earliest on the the waitlist
2. Assign the person who is earliest on the waitlist with their preferred animal as a pet
3. Repeat 1-2 until no people remain on the waitlist

This algorithm relies on a correct search algorithm in the first step. If the search algorithm incorrectly chose a random person, the algorithm for assigning animals as pets would also be incorrect.

As you will see in this chapter with searching and sorting there exist multiple correct algorithms for the same problem. Often there are good reasons to know multiple correct algorithms because there are tradeoffs in simplicity, algorithm cost, and assumptions about inputs.

## 2.1.4. Searching and Sorting

In this chapter we will look at two of the most common and important types of algorithms, Searching and Sorting. You probably come across these kinds of algorithms every time you use a computer without even realising! They also happen to be great for illustrating some of the key concepts that arise with algorithms.

## 2.2. Searching

Searching through collections of data is something computers have to do all the time. It happens every time you type in a search on Google, or when you type in a file name to search for on your computer. Computers deal with such huge amounts of data that we need fast algorithms to help us find information quickly.

Lets investigate searching with a game...

Use the interactive online at <http://www.csfieldguide.org.nz/en/interactives/searching-algorithms/index.html?max=2>

You may have noticed that the numbers on the monsters and pets in the game were in a random order, which meant that finding the pet was basically luck! You might have found it on your first try, or if you were less lucky you might have had to look inside almost all the presents before you found it. This might not seem like such a bad thing since you had enough lives to look under all the boxes, but imagine if there had been 1,000 boxes, or

worse 1,000,000! It would have taken far too long to look through all the boxes and the pet might have never been found.

Now this next game is slightly different. You have less lives, which makes things a bit more challenging, but this time the numbers inside the boxes will be in order. The monsters, or maybe the pet, with the smallest number is in the present on the far left, and the one with the largest number is in the present on the far right. Let's see if you can collect all the pets without running out of lives...

Use the interactive online at <http://www.csfieldguide.org.nz/en/interactives/searching-algorithms/index.html?level=3>

Now that you have played through the whole game (and hopefully found all of the lost pets!) you may have noticed that even though you had less lives in the second part of the game, and lots of presents to search through, you were still able to find the pet. Why was this possible?

## 2.2.1. Linear Search

Since the boxes in the first game were in a random order there really wasn't any strategy you could have used to find the pet, except simply keep opening presents one by one until you found the pet. This is essentially the *Linear Search* algorithm (sometimes called a sequential search). In plain English, Linear Search algorithm is as follows:

- Check if the first item in a list is the item you are searching for, if it is the one you are looking for, you are done.
- If it isn't the item you are searching for move on and check the next item.
- Continue checking items until you find the one you are searching for.

If you used this algorithm you might get lucky and find what you are looking for on your first go, but if you were really unlucky you might have to look through everything in your list before you found the right object! For a list of 10 items this means on average you would only have to look at 5 items to find what you were looking for, but for a list of 10000 you would have to look through on average 5000.

### **Curiosity:** How is Bozo search different from Linear search?

If you watched the video at the beginning of the chapter you might be thinking that what you did in the present searching game sounds more like Bozo Search than Linear Search, but actually Bozo Search is even sillier than this! If you were doing a Bozo Search then after unwrapping a present and finding a monster inside, you would

wrap the present back up and try another one at random! This means you might end up checking the same present again and again and again and you might never find the pet, even with a small number of presents!

## 2.2.2. Binary search

A much better algorithm to use is called Binary Search. In the second part of the present searching game the boxes were in order, which meant you were able to be more clever when you were searching for the pet, and you might have been using a Binary Search without realising!

If you used a Binary Search on each of the levels then you would have always had enough lives to find the pet! Informally, the Binary Search algorithm is as follows:

- Look at the item in the centre of the list and compare it to what you are searching for
- If it is what you are looking for then you are done.
- If it is larger than the item you are looking for then you can ignore all the items in the list which are larger than that item (if the list is from smallest to largest this means you can ignore all the items to the right of the centre item).
- If it is smaller then you can ignore all the items in the list which are smaller than that centre item.
- Now repeat the algorithm on the remaining half of the list, checking the middle of the list and choosing one of the halves, until you find the item you are searching for.

Binary Search is a very powerful algorithm. If you had 1000 presents to search through it would take you at most 10 checks for Binary search to find something and Linear search would take at most 1000 checks, but if you doubled the number of presents to search through how would this change the number of checks made by Binary Search and Linear search?

**Spoiler:** How does doubling the number of boxes affect the number of checks required?

The answer to the above question is that the maximum number of checks for Linear Search would double, but the maximum number for Binary Search would only increase by one.

It is important to remember that you can only perform a Binary Search if the items you are searching through are sorted into order. This makes the sorting algorithms we will look at

next even more important because without sorting algorithms we wouldn't be able to use Binary Search to quickly look through data!

**Project:** Code to run linear and binary search for yourself

The following files will run linear and binary search in various languages; you can use them to generate random lists of values and measure how long they take to find a given value. Your project is to measure the amount of time taken as the number of items ( $n$ ) increases; try drawing a graph showing this.

- [Scratch - Download Scratch here](#)
- [Python \(Version 2\) - Download Python 2 here](#)
- [Python \(Version 3\) - Download Python 3 here](#)

## 2.3. Sorting

Sorting is another very important area of algorithms. Computers often have to sort large amounts of data into order based on some attribute of that data, such as sorting a list of files by their name or size, or emails by the date they were received, or a customer list according to people's names. Most of the time this is done to make searching easier. For example you might have a large amount of data and each piece of data could be someone's name and their phone number. If you want to search for someone by name it would help to first have the data sorted alphabetically according to everyone's names, but if you then wanted to search for a phone number it would be more useful to have the data sorted according to people's phone numbers.

Like searching there are many different sorting algorithms, but some take much longer than others. In this section you will be introduced to two slower algorithms and one much better one.

### 2.3.1. Scales Interactive

Throughout this section you can use the sorting interactive to test out the algorithms we talk about. When you're using it make sure you take note of the comparisons at the bottom of the screen, each time you compare two boxes the algorithm is making 'one comparison' so the total number of comparisons you have to make with each algorithm is the cost of that algorithm for the 8 boxes.

Use the scales to compare the boxes (you can only compare two boxes at a time) and then arrange them along the bottom of the screen. Arrange them so that the lightest box is on

the far left and the heaviest is on the far right. Once you think they are in order click 'Test order'.

If the interactive does not run properly on your computer you can use a set of physical balance scales instead; just make sure you can only tell if one box is heavier than the other, not their exact weight (so not digital scales that show the exact weight).

Use the interactive online at <http://www.csfieldguide.org.nz/en/interactives/sorting-algorithms/index.html>

### 2.3.2. Selection Sort

One of the most intuitive ways to sort a group of boxes into order, from lightest to heaviest, is to start by first finding the lightest (or the heaviest) box and placing that to the side. Try this with the scales interactive.

After finding the lightest box simply repeat the process again with the remaining boxes until you find the second lightest, now place that to the side alongside the lightest box. If you keep repeating this process you will eventually find you have placed each box into order. Try sorting the whole group of boxes in the scales interactive into order using this method and count how many comparisons you have to make.

Tip: Start by moving all the boxes to the right of the screen and then once you have found the lightest box place it to the far right (if you want to find the heaviest first instead then move them all to the left).

If you record how many comparisons you had to make each time to find the next lightest box you might notice a pattern (hint: finding the lightest should take 7 comparisons, and then finding the second lightest should take 6 comparisons...). If you can see the pattern then how many comparisons do you think it would take to then sort 9 boxes into order? What about 20? If you knew how many comparisons it would take to sort 1000 boxes, then how many more comparisons would it take to sort 1001 instead?

This algorithm is called Selection sort, because each time you look through the list you are 'selecting' the next lightest box and putting it into the correct position. If you go back to the algorithms racing interactive at the top of the page you might now be able to watch the selection sort list and understand what it is doing at each step.

The selection sort algorithm can be described as follows:

- Find the smallest item in the list and place it to one side. This will be your sorted list.
- Next find the smallest item in the remaining list, remove it and place it into your sorted list beside the item you previously put to the side.

- Repeat this process until all items have been selected and moved into their correct position in the sorted list.

You can swap the word 'smallest' for 'largest' and the algorithm will still work, as long as you are consistent it doesn't matter if you are looking for the smallest or the largest item each time.

### 2.3.3. Insertion Sort

This algorithm works by removing each box from the original group of boxes and inserting it into its correct position in a new sorted list. Like Selection Sort, it is very intuitive and people often perform it when they are sorting objects themselves, like cards in their hands.

Try this with the scales interactive. Start by moving all the boxes to one side of the screen, this is your original, and unsorted, group. Now choose a box at random and place that on the other side of the screen, this is the start of your sorted group.

To insert another box into the sorted group, compare it to the box that is already in the sorted group and then arrange these two boxes in the correct order. Then to add the next box compare it to these boxes (depending on the weight of the box you might only have to compare it to one!) and then arrange these three boxes in the correct order. Continue inserting boxes until the sorted list is complete. Don't forget to count how many comparisons you had to make!

This algorithm is called Insertion Sort. If you're not quite sure if you've got the idea of the algorithm yet then have a look at [this animation](#) from [Wikipedia](#).

Insertion sort can be described with informal instructions as follows:

- Take an item from your unsorted list and place it to the side, this will be your sorted list.
- One by one, take each item from the unsorted list and insert it into the correct position in the sorted list.
- Do this until all items have been sorted.

People often perform this when they physically sort items. It can also be a very useful algorithm to use if you already have a sorted set of data and want to add a new piece of data into the set. For example if you owned a library and purchased a new book you

wouldn't do a Selection Sort on the entire library just to place this new book, you would simply insert the new book in its correct place.

### 2.3.4. Quicksort

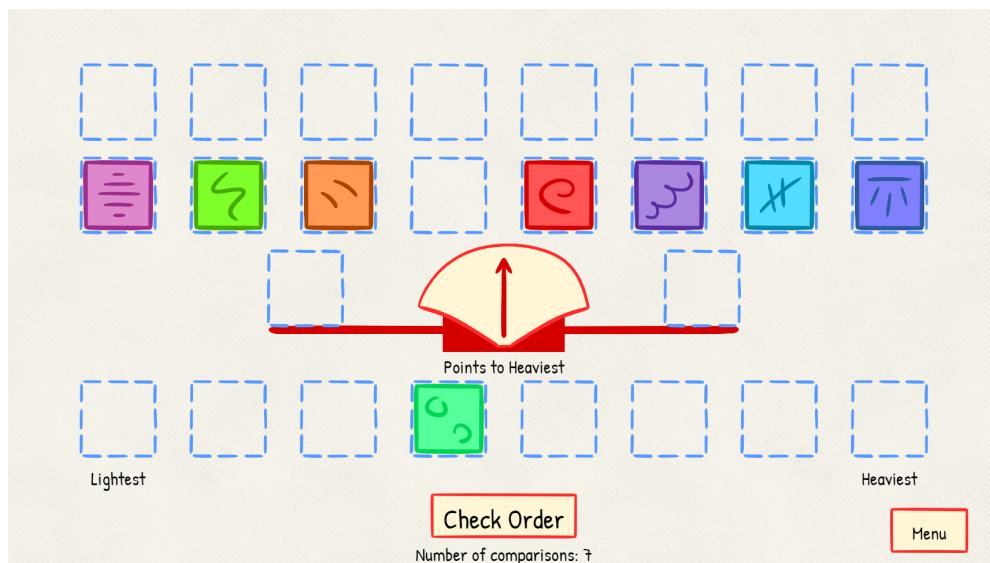
Insertion and Selection Sort may seem like logical ways to sort things into order, but they both take far too many comparisons when they are used for large amounts of data.

Remember computers often have to search through **HUGE** amounts of data, so even if they use a good searching algorithm like Binary Search to look through their data, if they use a bad sorting algorithm to first sort that data into order then finding anything will take far too long!

A much better sorting algorithm is **Quicksort!** (the name is a bit of a giveaway)

Use the interactive online at <http://www.csfieldguide.org.nz/en/interactives/sorting-algorithms/index.html?method=quick>

This algorithm is a little more complicated, but is very powerful. To do this algorithm with the sorting interactive, start by randomly choosing a box and placing it on the scales. Now compare every other box to the one you selected; heavier boxes should be put on the right of the second row and lighter boxes are put on the left. When you are done, place the box you were comparing everything else to between these two groups, but to help you keep track of things, put it in the row below. The following example shows how it might look after this step. Note that the selected block is in the right place for the final sorted order, and everything on either side will remain on the side that it is on.



Now apply this process to each of the two groups of boxes (the lighter ones, then the heavier ones). Keep on doing this until they are all sorted. The boxes should then be in sorted order!

It might be worth trying this algorithm out a few times and counting the number of comparisons you perform each time. This is because sometimes you might be unlucky and happen to pick the heaviest, or the lightest box first. On the other hand you might be very lucky and choose the middle box to compare everything to first. Depending on this the number of comparisons you perform will change.

Quicksort can be described in the following way:

- Choose an item from the list and compare every other item in the list to this (this item is often called the pivot).
- Place all the items that are greater than it into one subgroup and all the items that are smaller into another subgroup. Place the pivot item in between these two subgroups.
- Choose a subgroup and repeat this process. Eventually each subgroup will contain only one item and at this stage the items will be in sorted order.

#### **Project:** Code to run selection sort and quicksort for yourself

The following files will run selection sort and quicksort in various languages; you can use them to generate random lists of values and measure how long they take to be sorted. Note how long these take for various amounts of input ( $n$ ), and show it in a table or graph. You should notice that the time taken by Quicksort is quite different to that taken by selection sort.

- [Scratch - Download Scratch here](#)
- [Python \(Version 2\) - Download Python 2 here](#)
- [Python \(Version 3\) - Download Python 3 here](#)

There are dozens of sorting algorithms that have been invented; most of the ones that are used in practice are based on quicksort and/or mergesort. These, and many others, can be seen in this intriguing animated video.

Watch the video online at <https://www.youtube.com/watch?v=kPRA0W1kECg>

## 2.4. What makes an algorithm?

We've looked at algorithms that solved well known computational problems of sorting and searching data. When a computer scientist approaches a new computational problem that does not already have a well known solution they must create an algorithm.

There are three building blocks to develop a new algorithm: sequencing, selection, and iteration. One interesting early result in computer science is that combined, these three building blocks are sufficient to represent any algorithm that solves a computational problem!

### 2.4.1. Sequencing

Sequencing is the technique of deciding the order instructions are executed to produce the correct result. Imagine that we have the following instructions (A, B, C) to make a loaf of bread:

- **A** llow to sit at room temperature for 1 hour
- **B** ake for 30 minutes
- **C** ombine ingredients

C->A->B is a standard algorithm for a yeast bread. A different sequence, for example C->B->A, might produce a result that is edible but not high quality. Even worse, a sequence of B->C->A would not even produce something edible.

### 2.4.2. Selection

Selection is the technique of allowing the algorithm to select which instructions to execute depending on criteria. Using our previous bread baking example, our algorithm C->A->B works if the ingredients include yeast, but C->B would be faster if the ingredients do not include yeast (for example, the recipe might include baking powder as the rising agent). Selection allows us to create one algorithm to solve both cases:

1. Combine ingredients
2. **If ingredients contain yeast**, allow to sit at room temperature for 1 hour

3. Bake for 30 minutes

### 2.4.3. Iteration

Iteration allows an algorithm to repeat instructions. In its simplest form we might specify the exact number of times. For example, here is an algorithm to bake 2 loaves of bread:

1. **Repeat 2 times:**
  1. Combine ingredients
  2. If ingredients contain yeast, allow to sit at room temperature for 1 hour
  3. Bake for 30 minutes

This algorithm clearly works but it would take at least 3 hours to complete! If we had to make 20 loaves we would probably want to design a better algorithm. We could measure the size of the mixing bowl, how many loaves fit on the table to rise, and how many loaves we could bake at the same time in the oven. Our algorithm might then look like:

1. **Repeat 10 times:**
  1. Combine ingredients **for 2 loaves**
  2. **Split dough into 2 bread pans**
  3. If ingredients contain yeast, allow to sit at room temperature for 1 hour
  4. Bake bread pans in the same oven for 30 minutes

But what if we upgraded to a larger kitchen? Most algorithms are written to combine iteration with selection to handle arbitrarily large amounts of data (i.e. an unknown number of loaves of bread). We might create a general purpose bread baking algorithm:

1. **While we have enough ingredients for at least one loaf:**
  1. Combine ingredients **for up to X loaves** (where X is the maximum number of loaves that can fit in the mixing bowl or rising table)
  2. Split dough into X bread pans
  3. If ingredients contain yeast, allow to sit at room temperature for 1 hour
  4. **While there are still bread pans on the rising table:**
    1. Move **up to Y loaves** from the rising table to the oven (where Y is the maximum number of loaves that can fit in the oven)
    2. Bake bread pans in the same oven for 30 minutes

*Astute observers will note that this algorithm is still inefficient because the rising table and oven are not used at the same time. Designing algorithms that take advantage of parallelism is an important advanced topic in computer science.*

## 2.4.4. Combining Algorithms

One of the advantages of the building blocks perspective is that completed algorithms themselves can now be seen as new blocks we can build with. We can connect complete algorithms or we can interleave parts of algorithms to create new algorithms.

For example, a recipe for croutons might be:

1. Cut a loaf of bread into 2cm cubes
2. Brush cubes lightly with olive oil and season with salt, pepper, and herbs
3. Bake on large tray, flipping the cubes halfway through

We can connect the algorithm for baking bread in the previous section to this algorithm to create a new algorithm that makes croutons from scratch. If we required other ingredients for our recipe, we could connect multiple algorithms to build very complex algorithms.

Often when we have multiple algorithms that solve a problem there are advantages of each algorithm for specific cases. Hybrid algorithms take parts of multiple algorithms and combine them to gain the advantages of both original algorithms. For example, Timsort is one of the fastest known sorting algorithms in practice and it uses parts of insertion sort and merge sort. Insertion sort is used on very small sequences to take advantage of its speed for already or partially ordered sequences. Merge sort is used to merge these small sequences into larger ones to take advantage of the better upper bound on algorithm cost for large data sets.

### **Curiosity:** Why are there so many different programming languages?

So if we know how to define an algorithm, why are there so many programming languages? Programming languages are often created or adapted to express algorithms clearly for a specific problem domain. For example, it is easier to read mathematical algorithms in Python than Scratch. Similarly, data flow algorithms are clearer in visual programming languages like LabVIEW than Python.

## 2.5. The whole story!

We've only really scratched the surface of algorithms in this chapter, as there are millions of different algorithms for millions of different problems! Algorithms are used in maths, route planning, network planning and operation, problem solving, artificial intelligence, genetic programming, computer vision, the list goes on and on! But by going through this

chapter you should have gained an understanding of the key concepts of algorithms and will be well prepared to tackle more complicated ones in the future.

The algorithms introduced in this chapter aren't even necessarily the best for any situation; there are several other common ways of searching (e.g. hashing and search trees) and sorting (e.g. mergesort), and a computer scientist needs to know them, and be able to apply and fine tune the right one to a given situation.

In this chapter we have only talked about the number of comparisons an algorithm makes, and the amount of time a program takes to complete as 'costs' of algorithms. There are actually many other ways of measuring the cost of an algorithm. These include the amount of memory the algorithm uses and its computational complexity.

An algorithm often uses computer memory to store temporary data such as a partial sum of a list of numbers or a list of products that match some search criteria. With the large size of modern computer memory this may seem to not be as important as the number of steps an algorithm takes, but a poorly performing algorithm in terms of computer memory may be limited in its ability to work with the large data sets common in many industry applications. For example, a query algorithm that stored even a single bit for each record it searched could quickly overwhelm a web server's memory if it was searching a large data set such as Netflix's current movie offerings. Minimising memory usage while also minimizing the number of steps an algorithm takes is not always possible; there is often a tradeoff between computation and memory usage.

Computer Scientists use 'Big O notation' to more accurately describe the performance or complexity of an algorithm, and you are likely to come across this notation very quickly when investigating the performance of algorithms. It characterises the resources needed by an algorithm and is usually applied to the execution time required, or sometimes the space used by the algorithm.

#### **Extra For Experts:** Examples of Big O notation

Here are some Big O examples:

- $O(1)$  - An algorithm with  $O(1)$  complexity will always execute in the same amount of time regardless of how much data you give it to process. For example, finding the smallest value in a sorted list is always easy.
- $O(n)$  - The amount of time an algorithm with  $O(n)$  complexity will take to execute will increase roughly linearly with (i.e. in direct proportion to) the amount of data you give it to process. The high-score algorithm was  $O(n)$ , and so was the linear search.

- $O(n^2)$  - The performance of an algorithm with this complexity is roughly proportional to the square of the size of the input data set. Selection sort and insertion sort take  $O(n^2)$  time. That's not very good value - 10 times the amount of input will take 100 times as long!
- $O(2^n)$  - The amount of time an algorithm with this complexity will take to complete will double with each additional element added to the data set! We haven't seen these kinds of situations in this chapter, but they are common, and are a theme of the Complexity and Tractability chapter. Algorithms that are this slow can be almost impossible to use!

Big O Notation however requires some advanced mathematics to explore thoroughly so has been intentionally left out of this main chapter, but if you want to learn more check out the Useful Links section. These topics are looked at in more depth in the Complexity and Tractability chapter.

To make things even more complicated, theoretical analysis techniques such as Big O Notation are extremely useful when designing and predicting performance but empirical analysis such as stopwatch timing shows that in practice algorithm performance can vary greatly due to hardware and operating system design. Most computers have cached memory and virtual memory, where the time to access a particular value can be particularly short, or particularly long. There is a whole range of algorithms that are used for this situation to make sure that the algorithm still runs efficiently in such environments. Such algorithms are still based on the ideas we've looked at in this chapter, but require some clever adjustments to ensure that they work well.

## 2.6. Further reading

### 2.6.1. Other topics in algorithms

- There is another searching algorithm which performs even better than Binary Search. It is called Hashing and can be investigated with the CS Unplugged [Battleships Game](#)
- There are some problems for which no good algorithms have been found (and many people believe they will never be found). For more on these kinds of algorithms see the Complexity and Tractability chapter in the Field Guide.

### 2.6.2. Useful Links

- [CS Unplugged Searching algorithms](#)
- [CS Unplugged Sorting algorithms](#)

- Searching algorithm game, may not be suitable
- Video series on Algorithms by Nested
- Wikipedia has more details on [Linear Search](#), [Binary Search](#), [Selection sort](#), [Insertion sort](#), [Quicksort](#), and [Timsort](#).
- The [Sorting Bricks game](#) is a great way to learn about several sorting algorithms (requires Java).
- [Sorting Algorithms Visualisations](#) shows several different sorting algorithms racing and contains information and pseudocode for each.
- [Beginner's Guide to Big O Notation](#)

# 3. Programming Languages

## 3.1. What's the big picture?

Programming, sometimes referred to as coding, is a nuts and bolts activity for computer science. While this book won't teach you how to program (we've given some links to sites that can do this in the introduction), we are going to look at what a programming language is, and how computer scientists breath life into a language. From a programmer's point of view, they type some instructions, and the computer follows them. But how does the computer know what to do? Bear in mind that you might be using one of the many languages such as Python, Java, Scratch, Basic or C#, yet computers only have the hardware to follow instructions in one particular language, which is usually a very simple "machine code" that is hard for humans to read and write. And if you invent a new programming language, how do you tell the computer how to use it?

In this chapter we'll look at what happens when you write and run a program, and how this affects the way that you distribute the program for others to use.

We start with an optional subsection on what programming is, for those who have never programmed before and want an idea about what a program is. Examples of very simple programs in Python are provided, and these can be run and modified slightly. Working through this section should give you sufficient knowledge for the rest of this chapter to make sense; we won't teach you how to program, but you will get to go through the process that programmers use to get a program to run. Feel free to skip this section if you are already know a bit about programming.

A subsection on what this chapter focuses on then follows. Everybody should read that section.

### 3.1.1. What is programming?

Note: This section is intended for those who are unfamiliar with programming. If you already know a little about programming, feel free to skip over this section. Otherwise, it will give you a quick overview so that the remainder of the chapter makes sense.

An example of the simplest kind of program is as follows -- it has five instructions (one on each line) that are followed one after the other.

```
print("*****")
print("*****")
print("** Welcome to computer programming, Student **")
print("*****")
print("*****")
```

This program is written in a language called Python, and when the program runs, it will print the following text to the screen

```
*****
*****
*** Welcome to computer programming, Student ***
*****
*****
```

In order to run a Python program, we need something called a Python interpreter. A Python interpreter is able to read your program, and process it. Below is a Python interpreter that you can use to run your own programs. If you have a Python interpreter installed on your computer (ask your teacher if you are following this book for a class and are confused) and know how to start it and run programs in it, you can use that.

Use the interactive online at <http://www.csfieldguide.org.nz/en/interactives/python-interpreter/index.html>

Try changing the program so that it says your name instead of *Student*. When you think you have it right, try running the program again to see. Make sure you don't remove the double quotes or the parentheses (round brackets) in the program by mistake. What happens if you spelt "programming" wrong? Does the computer correct it? If you are completely stuck, ask your teacher for help before going any further.

Hopefully you figured out how to make the program print your name. You can also change the asterisks (\*) to other symbols. What happens if you do remove one of the double quotes or one of the parentheses? Try it!

If you change a critical symbol in the program you will probably find that the Python interpreter gives an error message. In the online Python interpreter linked to above, it says "ParseError: bad input on line 1", although different interpreters will express the error in different ways. If you have trouble fixing the error again, just copy the program back into Python from above.

Programming languages can do much more than print out text though. The following program is able to print out multiples of a number. Try running the program.

```
print("I am going to print the first 5 multiples of 3")
for i in range(5):
    print(i*3)
```

The first line is a print statement, like those you saw earlier, which just tells the system to put the message on the screen. The second line is a *loop*, which says to repeat the lines after it 5 times. Each time it loops, the value of *i* changes. i.e. the first time *i* is 0, then 1, then 2, then 3, and finally 4. It may seem weird that it goes from 0 to 4 rather than 1 to 5, but programmers tend to like counting from 0 as it makes some things work out a bit simpler. The third line says to print the current value of *i* multiplied by 3 (because we want multiples of 3). Note that there is *not* double quotes around the last print statement, as they are only used when we want to print out a something literally as text. If we did put them in, this program would print the text "i\*3" out 5 times instead of the value we want!

Try make the following changes to the program.

- Make it print multiples of 5 instead of 3. *Hint:* You need to change more than just the first line, you will need to make a change on the third line as well.
- Make it print the first 10 multiples instead of the first 5. Make sure it printed 10 multiples, and not 9 or 11!

You can also loop over a list of data. Try running the program below. It will generate a series of “spam” messages, one addressed to each person in the recipients list!

Note that the # symbol tells the computer that it should ignore the line, as it is a comment for the programmer.

```
# List of recipients to generate messages for
spam_recipients = ["Heidi", "Tim", "Pondy", "Jack", "Caitlin", "Sam", "David"]
# Go through each recipient
for recipient in spam_recipients:
    # Write out the letter for the current recipient
    print("Dear " + recipient + ", \n")
    print("You have been successful in the random draw for all people ")
    print("who have walked over a specific piece of ground located 2 meters ")
    print("from the engineering road entrance to Canterbury University.\n")
    print("For being successful in this draw you, " + recipient + ", win ")
    print("a prize of 10 million kilograms of chocolate!!!\n")
    print("And " + recipient + " if you phone us within the next 10 minutes ")
    print("you will get a bonus 5 million kilograms of chocolate!!! \n")
    print("\n\n\n") # Put some new lines between the messages
```

Try changing the recipients or the letter. Look carefully at all the symbols that were used to include the recipient's name in the letter.

### Jargon Buster: Syntax

The detailed requirements of a programming language about exactly which characters need to be used and where, is called its *syntax*. In the example above, the syntax for the list of names requires square brackets around the list, inverted commas around the names, and a comma between each one. If you make a mistake, such as leaving out one of the square brackets, the system will have a *syntax error*, and won't be able to run the program. Every symbol counts, and one small error in a program can stop it running, or make it do the wrong thing.

Programs can also use *variables* to store the results of calculations in, receive user input, and make decisions (called *conditionals*, such as *if* statements). Try running this program. Enter a number of miles to convert when asked. Don't put units on the number you enter; for example just put "12".

```
print("This program will convert miles to kilometers")
number_of_miles = int(input("Number of miles: "))
if number_of_miles < 0:
    print("Error: Can only convert a positive number of miles!")
else:
    number_of_kilometers = number_of_miles / 0.6214
    print("Calculated number of kilometers...")
    print(number_of_kilometers)
```

The first line is a *print* statement (which you should be very familiar with by now!) The second line asks the user for a number of miles which is converted from input text (called a string) to an integer, the third line uses an *if* statement to check if the number entered was less than 0, so that it can print an error if it is. Otherwise if the number was ok, the program jumps into the *else* section (the error is not printed because the *if* was not true), calculates the number of kilometers (there are 0.6214 kilometers in a mile), stores it into a *variable* called *number\_of\_kilometers* for later reference, and then the last line prints it out. Again, we don't have quotes around *number\_of\_kilometers* in the last line as we want to print the value out that is stored in the *number\_of\_kilometers* variable. If this doesn't make sense, don't worry. You aren't expected to know how to program for this chapter, this introduction is only intended for you to have some idea of what a program is and the things it can do.

If you are keen, you could modify this program to calculate something else, such as pounds to kilograms or Fahrenheit to Celsius. It may be best to use an installed Python interpreter on your computer rather than the web version, as the web version can give very unhelpful error messages when your program has a mistake in it (although all interpreters give terrible error messages at least sometimes!)

Programs can do many more things, such as having a graphical user interface (like most computer programs you will be familiar with), being able to print graphics onto a screen, or being able to write to and read from files on the computer in order to save information between each time you run the program.

### 3.1.2. Where are we going?

When you ran the programs, it might have seemed quite magical that the computer was able to instantly give you the output. Behind the scenes however, the computer was running your example programs through another program in order to convert them into a form that it could make sense of and then run.

Firstly, you might be wondering why we need languages such as Python, and why we can't give computers instructions in English. If we typed into the computer "Okay computer, print me the first 5 multiples of 3", there's no reason that it would be able to understand. For starters, it would not know what a "multiple" is. And it would not even know how to go about this task. Computers cannot be told what every word means, and they cannot know how to accomplish every possible task. Understanding human language is a very difficult task for a computer, as you will find out in the Artificial Intelligence chapter. Unlike humans who have an understanding of the world, and see meaning, computers are only able to follow the precise instructions you give them. Therefore, we need languages that are constrained and unambiguous that the computer "understands" instructions in. These can be used to give the computer instructions, like those in the previous section.

It isn't this simple though, a computer cannot run instructions given directly in these languages. At the lowest level, a computer has to use physical hardware to run the instructions. Arithmetic such as addition, subtraction, multiplication, and division, or simple comparisons such as less than, greater than, or equal to are done on numbers represented in binary by putting electricity through physical computer chips containing transistors. The output is also a number represented in binary. Building a fast and cheap circuit to do simple arithmetic such as this isn't that hard, but the kind of instructions that people want to give computers (like "print the following sentence", or "repeat the following 100 times") are much harder to build circuitry for.

#### Jargon Buster: Binary

The electronics in computers uses circuitry that mainly just works with two values (represented as high and low voltages) to make it reliable and fast. This system is called *binary*, and is often written on paper using zeroes and ones. There's a lot more about binary in the [data representation](#) chapter, and it's worth having a quick look at the first section of that now if you haven't come across binary before.

So instead of building computers that can understand these high level instructions that you find in languages like Python (or Java, Basic, JavaScript, C and so on), we build computers that can follow a very limited set of instructions, and then we write programs that convert the instructions in the standard languages people write programs in into the simple instructions that the circuitry can directly carry out. The language of these simple instructions is a low level programming language often referred to as machine code.

The conversion from a high level to a low level language can involve *compiling*, which replaces the high level instructions with machine code instructions that can then be run, or it can be done by *interpreting*, where each instruction is converted and followed one by one, as the program is run. In reality, a lot of languages use a mixture of these, sometimes compiling a program to an intermediate language, then interpreting it (Java does this). The language we looked at earlier, Python, is an interpreted language. Other languages such as C++ are compiled. We will talk more about compiling and interpreting later.

Different levels of programming languages are an abstraction that allows programmers to concern themselves with only the necessary details of a single level. High level programmers can produce sophisticated programs in Python without expert knowledge of low level languages such as MIPS, x86, or ARM. Low level programmers can produce embedded programs in ARM without expert knowledge of electronic circuitry.

We will start with looking at low level languages and how computers actually carry out the instructions in them, then we will look at some other programming languages that programmers use to give instructions to computers, and then finally we will talk about how we convert programs that were written by humans in a high level language into a low level language that the computer can carry out.

## 3.2. Machine Code (Low level languages)

A computer has to carry out instructions on physical circuits. These circuits contain transistors laid out in a special way that will give a correct output based on the inputs.

Data such as numbers (represented using binary) have to be put into storage places called registers while the circuit is processing them. Registers can be set to values, or data from memory can be put into registers. Once in registers, they can be added, subtracted, multiplied, divided, or be checked for equality, greater than, or less than. The output is put into a register, where it can either be retrieved or used in further arithmetic.

All computers have a machine code language (commonly referred to as an instruction set) that is used to tell the computer to put values into registers, to carry out arithmetic with the values in certain registers and put the result into another specified register like what we talked about above. Machine code also contains instructions for loading and saving values from memory (into or out of registers), jumping to a certain line in the program (that is either before or after the current line), or to jump to the line only if a certain condition is met (by doing a specified comparisons on values in registers). There are also instructions for handling simple input/ output, and interacting with other hardware on the computer.

The instructions are quite different to the ones you will have seen before in high level languages. For example, the following program is written in a machine language called MIPS; which is used on some embedded computer systems. We will use MIPS in examples throughout this chapter.

It starts by adding 2 numbers (that have been put in registers \$t0 and \$t1) and printing out the result. It then prints “Hello World!” Don’t worry, we aren’t about to make you learn how to actually program in this language! And if you don’t really understand the program, that’s also fine because many software engineers wouldn’t either! (We are showing it to you to help you to appreciate high level languages!)

```
.data
str: .asciiz "\nHello World!\n"
# You can change what is between the quotes if you like

.text
.globl main

main:
# Do the addition
# For this, we first need to put the values
# to add into registers ($t0 and $t1)
# You can change the 30 below to another value
li $t0, 30
# You can change the 20 below to another value
li $t1, 20

# Now we can add the values in $t0
# and $t1, putting the result in special register $a0
add $a0, $t0, $t1
```

```

# Set up for printing the value in $a0.
# A 1 in $v0 means we want to print an int
li $v0, 1

# The system call looks at what is in $v0
# and $a0, and knows to print what is in $a0
syscall

# Now we want to print Hello World
# So we load the (address of the) string into $a0
la $a0, str

# And put a 4 in $v0 to mean print a string
li $v0, 4

# And just like before syscall looks at
# $v0 and $a0 and knows to print the string
syscall

# Nicely end the program
li $v0, 0
jr $ra

```

You can run this program using a MIPS emulator using this interactive:

Use the interactive online at <http://www.csfieldguide.org.nz/en/interactives/mips-assembler/index.php>

Copy and paste the output in the “Assembler Output” box into the box in this simulator interactive:

Use the interactive online at <http://www.csfieldguide.org.nz/en/interactives/mips-simulator/index.php>

Once you have got the program working, try changing the values that are added. The comments tell you where these numbers that can be changed are. You should also be able to change the string (text) that is printed without too much trouble also. As a challenge, can you make it so that it subtracts rather than adds the numbers? Clue: instruction names are always very short. Unfortunately you won't be able to make it multiply or divide using this simulator as this is not currently supported. Remember that to rerun the program after changing it, you will have to follow both steps 1 and 2 again.

You may be wondering why you have to carry out both these steps. Because computers work in 1's and 0's, the instructions need to simply be converted into hexadecimal.

Hexadecimal is a shorthand notation for binary numbers. *Don't muddle this process with*

*compiling or interpreting!* Unlike these, it is much simpler as in general each instruction from the source code ends up being one line in the hexadecimal.

One thing you might have noticed while reading over the possible instructions is that there is no loop instruction in MIPS. Using several instructions though, it actually is possible to write a loop using this simple language. Have another read of the paragraph that describes the various instructions in MIPS. Do you have any ideas on how to solve this problem? It requires being quite creative!

The jumping to a line, and jumping to a line if a condition is met can be used to make loops! A very simple program we could write that requires a loop is one that counts down from five and then says “Go!!!!” once it gets down to one. In Python we can easily write this program in three lines.

```
# Start at 5, count down by 1 each time, and stop when we get to 0
for number in range(5, 0, -1):
    print(number)
print("GO!!!!!!")
```

But in MIPS, it isn’t that straightforward. We need to put values into registers, and we need to build the loop out of jump statements. Firstly, how can we design the loop?

And the full MIPS program for this is as follows. You can go away and change it.

```
# Define the data strings
.data
go_str: .asciiz "GO!!!!!\n"
new_line: .asciiz "\n"

.text
# Where should we start?
.globl main

main:
    # Put our starting value 5 into register $t0. We will update it as we go
    li $t0, 5
    # Put our stopping value 0 into register $t1
    li $t1, 0

    # This label is just used for the jumps to refer to
start_loop:
    # This says that if the values in $t0 and $t1 are the same,
    # it should jump down to the end_loop label. This is the
    # main loop condition.
    beq $t0, $t1, end_loop
    # These three lines prepare for and print the current int
    # It must be moved into $a0 for the printing
```

```

move $a0, $t0
li $v0, 1
syscall
# These three lines print a new line character so that
# each number is on a new line
li $v0, 4
la $a0, new_line
syscall
# Add -1 to the value in $t0, i.e decrement it by 1
addi $t0, $t0, -1
# Jump back up to the start_loop label
j start_loop

# This is the end loop label that we jumped to when the loop is false
end_loop:
# These three lines print the "GO!!!!" string
li $v0, 4
la $a0, go_str
syscall
# And these 2 lines make the program exit nicely
li $v0, 0
jr $ra

```

Can you change the Python program so that it counts down from 10? What about so it stops at 5? (You might have to try a couple of times, as it is somewhat counter intuitive. Remember that when *i* is the stopping number, it stops there and does not run the loop for that value!). And what about decrementing by 2 instead of 1? And changing the string (text) that is printed at the end?

You probably found the Python program not too difficult to modify. See if you can make these same changes to the MIPS program.

If that was too easy for you, can you make both programs print out "GO!!!!" twice instead of once? (you don't have to use a loop for that). And if THAT was too easy, what about making each program print out "GO!!!!" 10 times? Because repeating a line in a program 10 times without a loop would be terrible programming practice, you'd need to use a loop for this task.

More than likely, you're rather confused at this point and unable to modify the MIPS program with all these suggested changes. And if you do have an additional loop in your MIPS program correctly printing "GO!!! 10 times, then you are well on your way to being a good programmer!

So, what was the point of all this? These low level instructions may seem tedious and a bit silly, but the computer is able to directly run them on hardware due to their simplicity. A programmer can write a program in this language if they know the language, and the computer would be able to run it directly without doing any further processing. As you have

probably realised though, it is extremely time consuming to have to program in this way. Moving stuff in and out of registers, implementing loops using jump and branch statements, and printing strings and integers using a three line pattern that you'd probably never have guessed was for printing had we not told you leaves even more opportunities for bugs in the program. Not to mention, the resulting programs are extremely difficult to read and understand.

Because computers cannot directly run the instructions in the languages that programmers like, high level programming languages by themselves are not enough. The solution to this problem of different needs is to use a compiler or interpreter that is able to convert a program in the high level programming language that the programmer used into the machine code that the computer is able to understand.

These days, few programmers program directly in these languages. In the early days of computers, programs written directly in machine language tended to be faster than those compiled from high level languages. This was because compilers weren't very good at minimising the number of machine language instructions, referred to as *optimizing*, and people trained to write in machine code were better at it. These days however, compilers have been made a lot smarter, and can optimize code far better than most people can. Writing a program directly in machine code may result in a program that is *less* optimized than one that was compiled from a high level language. Don't put in your report that low level languages are faster!

This isn't the full story; the MIPS machine code described here is something called a Reduced Instruction Set Architecture (RISC). Many computers these days use a Complex Instruction Set Architecture (CISC). This means that the computer chips can be a little more clever and can do more in a single step. This is well beyond the scope of this book though, and understanding the kinds of things RISC machine code can do, and the differences between MIPS and high level languages is fine at this level, and fine for most computer scientists and software engineers.

In summary, we require low level programming languages because the computer can understand them, and we require high level programming languages because humans can understand them. A later section talks more about compilers and interpreters; programs that are used to convert a program that is written in a high level language (for humans) into a low level language (for computers).

## 3.3. A Babel of programming languages

There are many different programming languages. Here we have included a small subset of languages, to illustrate the range of purposes that languages are used for. There are

many, many more languages that are used for various purposes, and have a strong following of people who find them particularly useful for their applications.

For a much larger list you can [check Wikipedia here](#).

### 3.3.1. Python

Python is a widely used language, that has also become very popular as a teaching language. Many people learn Python as their first programming language. In the introduction, we looked at some examples of Python programs, for those who have never programmed before.

Originally though, Python was intended to be a scripting language. Scripting languages have syntax that makes them quick to write programs for file processing in, and for doing repetitive tasks on a computer.

As an example of a situation where Python is very useful, imagine your teacher has given 5 quizzes throughout the year, and recorded the results for each student in a file such as this (It could include more than 6 students), where each student's name is followed by their scores. Some students didn't bother going to class for all the quizzes, so have less than 5 results recorded.

```
Karen 12 12 14 18 17
James 9 7 1
Ben 19 17 19 13
Lisa 9 1 3 0
Amalia 20 20 19 15 18
Cameron 19 15 12 9 3
```

She realises she needs to know the average (assuming 5 quizzes) that each student scored, and with many other things to do does not want to spend much time on this task. Using python, she can very quickly generate the data she needs in less than 10 lines of code.

Note that understanding the details of this code is irrelevant to this chapter, particularly if you aren't yet a programmer. Just read the comments (the things that start with a "#") if you don't understand, so that you can get a vague idea of how the problem was approached.

```
# Open the raw score file for reading
raw_scores_file = open("scores.txt", "r")
# Create and open a file for writing the processed scores into
processed_scores_file = open("processed_scores.txt", "w")
```

```

# For each line in the file
for line in raw_scores_file.readlines():
    # Get the name, which is in the first part of the line
    name = line.split()[0]
    # Get a list of the scores, which are on the remainder of the lines
    scores_on_line = [int(score) for score in line.split()[1:]]
    # Calculate the average, which is the sum of the scores divided by 5
    average = sum(scores_on_line) / 5
    # Write the average to the processed scores output file
    processed_scores_file.write(name + " " + str(average) + "\n")

# Close both files
raw_scores_file.close()
processed_scores_file.close()

```

This will generate a file that contains each student's name followed by the result of adding their scores and dividing the sum by 5. You can try the code if you have python installed on your computer (it won't work on the online interpreter, because it needs access to a file system). Just put the raw data into a file called "scores.txt" in the same format it was displayed above. As long as it is in the same directory as the source code file you make for the code, it will work.

This problem could of course be solved in any language, but some languages make it far simpler than others. Standard software engineering languages such as Java, which we talk about shortly, do not offer such straight forward file processing. Java requires the programmer to specify what to do if opening the file fails in order to prevent the program from crashing. Python does not require the programmer to do this, although does have the option to handle file opening failing should the programmer wish to. Both these approaches have advantages in different situations. For the teacher writing a quick script to process the quiz results, it does not matter if the program crashes so it is ideal to not waste time writing code to deal with it. For a large software system that many people use, crashes are inconvenient and a security risk. Forcing all programmers working on that system to handle this potential crash correctly could prevent a lot of trouble later on, which is where Java's approach helps.

In addition to straight forward file handling, Python did not require the code to be put inside a class or function, and it provided some very useful built in functions for solving the problem. For example, the function that found the sum of the list, and the line of code that was able to convert the raw line of text into a list of numbers (using a very commonly used pattern).

This same program written in Java would require at least twice as many lines of code.

There are many other scripting languages in addition to Python, such as Perl, Bash, and Ruby.

### 3.3.2. Scratch

Scratch is a programming language used to teach people how to program. A drag and drop interface is used so that new programmers don't have to worry so much about syntax, and programs written in Scratch are centered around controlling cartoon characters or other sprites on the screen.

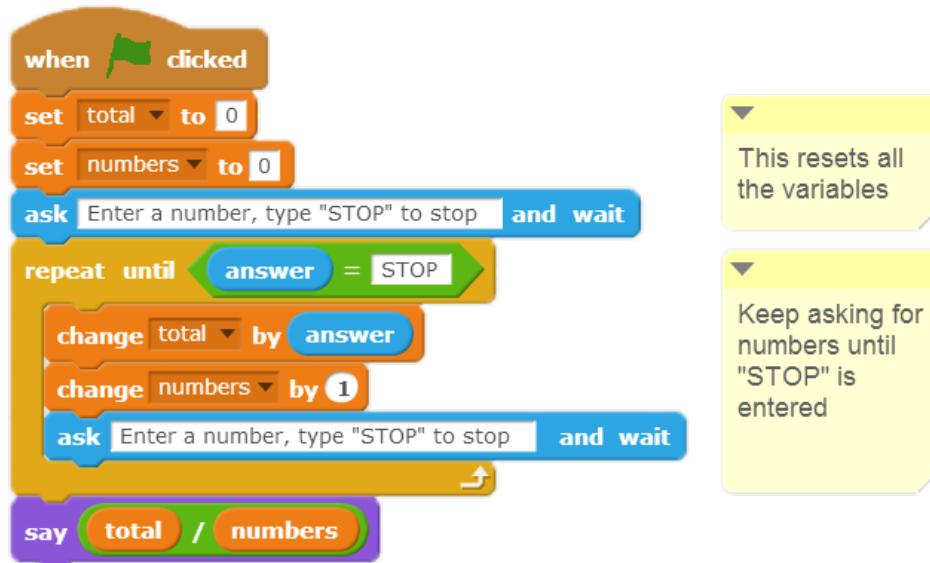
Scratch is never used in programming in industry, only in teaching. If you are interested in trying Scratch, [you can try it out online here](http://scratch.mit.edu/projects/19711355/#editor), no need to download or install anything.

#### Example Scratch

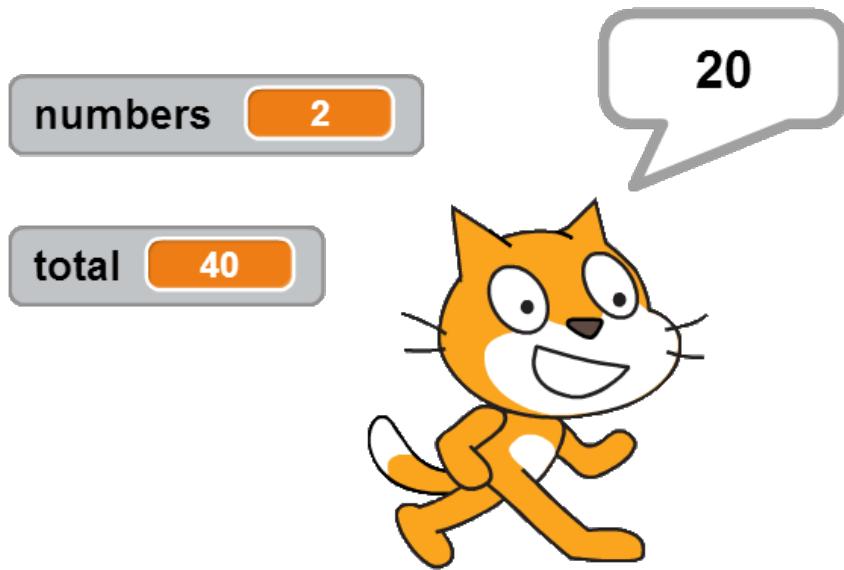
#### project

View the link online at <http://scratch.mit.edu/projects/19711355/#editor>

This is an example of a simple program in Scratch that is similar to the programs we have above for Python and Java. It asks the user for numbers until they say "stop" and then finds the average of the numbers given.



And this is the output that will be displayed when the green flag is clicked:



Scratch can be used for simple calculations, creating games and animations. However it doesn't have all the capabilities of other languages.

Other educational languages include Alice and Logo. Alice also uses drag and drop, but in a 3D environment. Logo is a very old general purpose language based on Lisp. It is not used much anymore, but it was famous for having a turtle with a pen that could draw on the screen, much like Scratch. The design of Scratch was partially influenced by Logo. These languages are not used beyond educational purposes, as they are slow and inefficient.

### 3.3.3. Java

Java is a popular general purpose software engineering language. It is used to build large software systems involving possibly hundreds or even thousands of software engineers. Unlike Python, it forces programmers to say how certain errors should be handled, and it forces them to state what type of data their variables are intended to hold, e.g. *int* (i.e. a number with no decimal places), or *String* (some text data). Python does not require types to be stated like this. All these features help to reduce the number of bugs in the code. Additionally, they can make it easier for other programmers to read the code, as they can easily see what type each variable is intended to hold (figuring this out in a python program written by somebody else can be challenging at times, making it very difficult to modify their code without breaking it!)

This is the Java code for solving the same problem that we looked at in Python; generating a file of averages.

```
import java.io.*;
import java.util.*;
```

```

public class Averager {
    public static void main() {
        String inputFile = "scores.txt";
        String outputFile = "processed_scores.txt";
        try {
            Scanner scanner = new Scanner(new File(inputFile));
            PrintStream outputFile = new PrintStream(new File(outputFile));
            while (scanner.hasNextLine()) {
                String name = scanner.next();
                Scanner numbersToRead = new Scanner(scanner.nextLine());
                int totalForLine = 0;
                while (numbersToRead.hasNextInt()) {
                    totalForLine += numbersToRead.nextInt();
                }
                outputFile.println(name + " " + totalForLine/5.0 + "\n");
            }
            outputFile.close();
        } catch (IOException e) {
            System.out.println("The file could not be opened!" + e);
        }
        System.out.println("I am finished!");
    }
}

```

While the code is longer, it ensures that the program doesn't crash if something goes wrong. It says to *try* opening and reading the file, and if an error occurs, then it should *catch* that error and print out an error message to tell the user. The alternative (such as in Python) would be to just crash the program, preventing anything else from being able to run it. Regardless of whether or not an error occurs, the "I am finished!" line will be printed, because the error was safely "caught". Python is able to do error handling like this, but it is up to the programmer to do it. Java will not even compile the code if this wasn't done! This prevents programmers from forgetting or just being lazy.

There are many other general software engineering languages, such as C# and C++. Python is sometimes used for making large software systems, although is generally not considered an ideal language for this role.

### 3.3.4. JavaScript

- Interpreted in a web browser
- Similar language: Actionscript (Flash)

Note that this section will be completed in a future version of the field guide. For now, you should refer to wikipedia page for more information.

### 3.3.5. C

- Low level language with the syntax of a high level language
- Used commonly for programming operating systems, and embedded systems
- Programs written in C tend to be very fast (because it is designed in a way that makes it easy to compile it optimally into machine code)
- Bug prone due to the low level details. Best not used in situations where it is unnecessary
- Related languages: C++ (somewhat)

Note that this section will be completed in a future version of the field guide. For now, you should refer to wikipedia page for more information.

### 3.3.6. Matlab

- Used for writing programs that involve advanced math (calculus, linear algebra, etc.)
- Not freely available
- Related languages: Mathematica, Maple

Note that this section will be completed in a future version of the field guide. For now, you should refer to wikipedia page for more information.

### 3.3.7. Esoteric Programming Languages

Anybody can make their own programming language. Doing so involves coming up with a syntax for your language, and writing a parser and compiler or interpreter so that programs in your language can be run. Most programming languages that people have made never become widely used.

In addition to programming languages that have practical uses, people have made many programming languages that were intended to be nothing more than jokes, or to test the limits of how obscure a programming language can be. Some of them make the low level machine languages you saw earlier seem rather logical! Wikipedia has a [list of such languages](#).

You could even make your own programming language if you wanted to!

## 3.4. How does the computer process your program?

A programming language such as Python or Java is implemented using a program itself --- the thing that takes your Python program and runs it is a program that someone has written!

Since the computer hardware can only run programs in a low level language (machine code), the programming system has to make it possible for your Python instructions to be executed using only machine language. There are two broad ways to do this: interpreting and compiling.

[This 1983 video](#) provides a good analogy of the difference between an interpreter and a compiler.

The main difference is that a compiler is a program that converts your program to machine language, which is then run on the computer. An interpreter is a program that reads your program line by line, works out what those instructions are, and does them immediately.

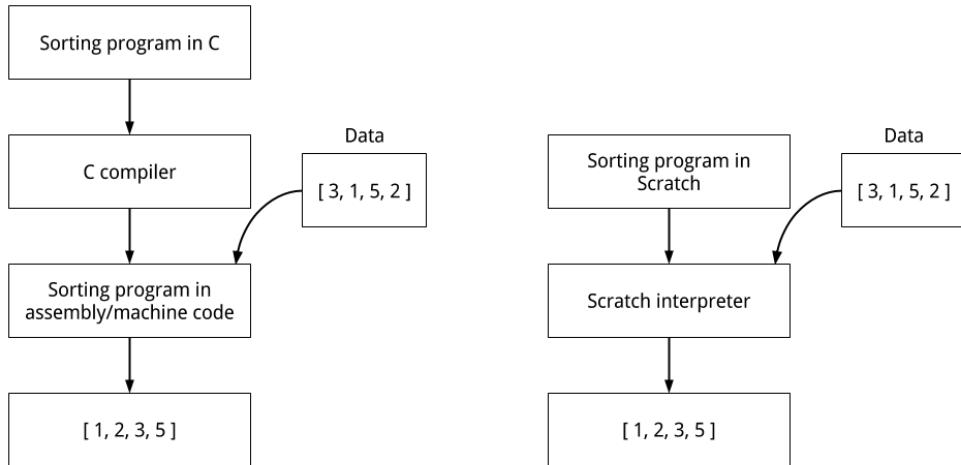
There are advantages to both approaches, and each one suits some languages better than others. In reality, most modern languages use a mixture of compiling and interpreting. For example, most Java programs are *compiled* to an "intermediate language" called ByteCode, which is closer to machine code than Java. The ByteCode is then executed by an interpreter.

If your program is to be distributed for widespread use, you will usually want it to be in machine code because it will run faster, the user doesn't have to have an interpreter for your particular language installed, and when someone downloads the machine code, they aren't getting a copy of your original high-level program. Languages where this happens include C#, Objective C (used for programming iOS devices), Java, and C.

Interpreted programs have the advantage that they can be easier to program because you can test them quickly, trace what is happening in them more easily, and even sometimes type in single instructions to see what they do, without having to go through the whole compilation process. For this reason they are widely used for introductory languages (for example, Scratch and Alice are interpreted), and also for simple programs such as scripts that perform simple tasks, as they can be written and tested quickly (for example, languages like PHP, Ruby and Python are used in these situations).

The diagram below shows the difference between what happens in an interpreter and compiler if you write and run a program that sorts some numbers. The compiler produces

a machine code program that will do the sorting, and the data is fed into that second program to get the sorted result. The interpreter simply does the sorting on the input by immediately following the instructions in the program. The compiler produces a machine code program that you can distribute, but it involves an extra phase in the process.



## 3.5. The whole story!

There are many different programming languages, and new ones are always being invented. Each new language will need a new compiler and/or interpreter to be developed to support it. Fortunately there are good tools to help do this quickly, and some of these ideas will come up in the *Formal Languages* chapter, where things like regular expressions and grammars can be used to describe a language, and a compiler can be built automatically from the description.

The languages we have discussed in this chapter are ones that you are likely to come across in introductory programming, but there are some completely different styles of languages that have very important applications. There is an approach to programming called **Functional programming** where all operations are formulated as mathematical functions. Common languages that use functional techniques include Lisp, Scheme, Haskell, Clojure and F#; even some conventional languages (such as Python) include ideas from functional programming. A pure functional programming style eliminates a problem called *side effects*, and without this problem it can be easier to make sure a program does exactly what it is intended to do. Another important type of programming is **logic programming**, where a program can be thought of as a set of rules stating what it should

do, rather than instructions on how to do it. The most well-known logic programming language is Prolog.

## 3.6. Further reading

### 3.6.1. Useful Links

- The [TeachICT lesson on programming languages](#) covers many of the topics in this chapter
- CS Online has a [quick overview of this topic](#)
- Wikipedia entries on [Programming language](#), [High level language](#), and [Low level language](#)
- website including posters comparing programming languages by Samuel Williams
- tutorial comparing programming languages
- a poster with full details of the file content in an executable file (the exe format)
- David Bolton explains a [Programming Language](#), [Compiler](#), and [the difference between Compilers and Interpreters](#).
- Computerworld article on the A to Z of programming languages
- [What is Python?](#) (compared with other languages)
- A very large poster showing a timeline of the development of programming languages
- Hello World program in hundreds of programming languages
- 99 bottles of beer song in hundreds of programming languages

# 4. Human Computer Interaction

Watch the video online at <https://www.youtube.com/embed/Uw0PlSu2pog>

## 4.1. What's the big picture?

People often become frustrated with computers and other digital devices. At some point when using these devices, you are likely to become annoyed that the system did something you didn't want it to do, or you can't figure out how to get the computer to do what you want, but why is that? Humans made computers, so why are computers often so frustrating for humans to use?

Computers are becoming hundreds of times more powerful every decade, yet there is one important component of the computer system that hasn't changed significantly in performance since the first computers were developed in the 1940s: the human. For a computer system to work really well, it needs to be designed by people who understand the human part of the system well.

In this chapter we'll look at what typically makes good and bad interfaces. The idea is to make you sensitive to the main issues so that you can critique existing interfaces, and begin to think about how you might design good interfaces.

Often software developers create a computer program or system for a device that requires the user to spend some time to learn how to use. The interface might be easy to use for the developer since they know the system really well, but a user just wants to get the job done without spending too much time learning the software (and they might switch to another program if it's too hard to use). A developer might think of the program and the user separately, but the user is part of the system, and a developer needs to create the software with the user in mind, designing a program that they will find easy to use and intuitive.

Human-computer interaction (HCI) is about trying to make programs useful, usable, and accessible to humans. It goes way beyond choosing layouts, colours, and fonts for an interface. It's strongly influenced by the psychology of how people interact with digital

devices, which means understanding many issues about how people behave, how they perceive things, and how they understand things so that they feel that a system is working to help them and not hinder them. By understanding HCI, developers are more likely to create software that is effective and popular. If you ask people if they have ever been frustrated using a computer system, you'll probably get a clear message that HCI isn't always done well.

Try out the following interactive task, and get some friends to try it:

Use the interactive online at <http://www.csfieldguide.org.nz/en/interactives/deceiver/index.html>

Did anyone get a wrong answer to the question even though they thought they got it right? You may have noticed that the "Even" and "Odd" buttons sometimes swap. Inconsistency is almost always a really bad thing in an interface, as it can easily fool the user into making an error.

The only situation it might be desirable is if it was intentionally done to make a computer game more interesting (which perhaps the above interactive could be). But imagine you have a web form in which the "reset" and "submit" buttons often switched places. Users would frequently clear the form when they meant to submit it, or submit the form when they had meant to clear it!

The study of Human Computer Interaction involves a lot of psychology (how people behave) because this affects how they will use a system. As a simple example, the human short term memory only lasts for a matter of seconds (even in young people!) If a device delays a response for more than about 10 seconds, the user has to make a conscious effort to remember what they were doing, and that's extra work for the user (which from their point of view, makes the system more tiring to use). Another example is that people get "captured" into sequences: if you start biking on a route that you take each day, you'll soon find yourself arriving without thinking about every turn along the way, which is fine unless you were supposed to go somewhere else on the way, or if you have recently moved to a new house or workplace. A similar effect occurs with confirmation dialogues; perhaps you often accidentally close a file without saving it, and the system says "Do you want to save it?", so you press "Yes". After you've done this a few times you'll be captured into that sequence, so on the one occasion that you don't want to overwrite your old file, you may accidentally click "Yes" anyway.

**Curiosity:** Capture errors

Getting used to a regular route or procedure, and as a result forgetting something different you had to do that day, is called a *capture error*. This is easy to remember, as you get "captured" in your usual sequence. Capture is a good thing much of the time, as it saves you having to think hard about everyday actions (which can literally be more tiring), but it can also trick you into doing something you didn't intend. A good interface designer will be aware of this, and avoid setting up the interface so that a user might be captured into doing something that they can't undo.

A lot of people might blame themselves for such errors, but basic psychology says that this is a natural error to make, and a good system should protect users from such errors (for example, by allowing them to be undone).

Designing good interfaces is very difficult. Just when you think you've got a clever idea, you'll find that a whole group of people struggle to figure out how to use it, or it backfires in some situation. Even worse, some computer developers think that their users are dummies and that any interface problems are the user's fault and not the developer's. But the real problem is that the developer knows the system really well, whereas the user just wants to get their job done without having to spend a lot of time learning the software – if the software is too hard to use and they have a choice, they'll just find something else that's easier. Good interfaces are worth a lot in the market!

There are many ways to evaluate and fine tune interfaces, and in this chapter we'll look at some of these. One important principle is that one of the worst people to evaluate an interface is the person who designed and programmed it. They know all the details of how it works, they've probably been thinking about it for weeks, they know the bits that you're not supposed to touch and the options that shouldn't be selected, and of course they have a vested interest in finding out what is *right* with it rather than what is *wrong*. It's also important that the interface should be evaluated by someone who is going to be a typical user; if you get a 12-year-old to evaluate a retirement planning system they may not know what the user will be interested in; and if you get a teacher to try out a system that students will use, they may know what the answers are and what the correct process is.

Often interfaces are evaluated by getting typical users to try them out, and carefully noting any problems they have. There are companies that do nothing but these kinds of user tests – they will be given a prototype product, and pay groups of people to try it out. A report on the product is then produced and given to the people who are working on it. This is an expensive process, but it makes the product a lot better, and may well give it a huge advantage over its competitors. Having it evaluated by a separate company means that you avoid any bias from people in your own company who want to prove (even

subconsciously) that they've done a good job designing it, rather than uncover any niggling problems in the software that will annoy users.

Before we look at different ways to evaluate interfaces, we need to consider what is happening with an interface.

## 4.2. Users and tasks

A very important consideration when designing or evaluating an interface is who the users are likely to be. For example, the typical age of a user can be significant: very young children may have difficulty reading some words and prefer images and animations, while someone in a commercial setting who uses an interface frequently will want it to be very fast to use, perhaps using keyboard shortcuts only.

Think about the kinds of considerations you would have to make for the following user groups.

- Senior citizens
- Gamers
- Casual users
- Foreign visitors

**Spoiler:** Some possible answers: Don't open until you've thought about it!

- Senior citizens: use large print, have few features to learn, don't rely so much on memory, allow for poor eyesight and less agile physically (e.g. large buttons help), don't assume previous experience with computers
- Gamers: use previous experience with typical game interfaces, expecting challenges, probably running on a high end machine
- Casual users: interface needs to be very easy to learn, perhaps based on widely used systems, needs clear guidance
- Foreign visitors: use simple language and meaningful images/icons

The interface is the only part of a program that the user sees (that's the definition of an interface!), so if the interface doesn't work for them, then the program doesn't work.

Another important thing to do when designing and evaluating an interface is to think about what tasks it is being used for. Advertisements for digital devices often hide the complexity of a task, and simply point out the features available for doing the task. For example, suppose a smartphone is advertised as having a high resolution camera. The real

task that someone might want to do is to take a photo of something they've just spotted, and send it to a friend. If you look at what happens in reality, the smartphone might be in their pocket or bag, and if they see something cool going past, they need to get it out, perhaps unlock it, open the camera app, adjust the lighting and other settings, press a button (is it easy to find when you're holding the camera up?), select the photo, choose a sharing option, select a friend to share it with (does the system help with that?), send it (what happens if you're out of reception range?), and then put the phone away. If any one of these steps is slow or hard to remember, the whole experience can be frustrating, and it's possible the photo opportunity will be missed, or for some other reason the friend won't receive the photo.

It's important to think through all the parts of a task when discussing an interface, as it's the small steps in a task that make all the difference between using the interface in a real situation, compared with a demonstration of some features of the device.

### **Challenge:** Thinking about the context of tasks

It's very important to think about the whole context when describing a task. As an exercise, can you provide an example of a real task, including context, for a real person for each of the following:

- set an alarm clock
- show a slide (Powerpoint) presentation

Discuss your answers with a classmate or a friend. This should help you to evaluate your own answer, and consider other possible examples.

### **Curiosity:** Dumb users or dumb interfaces?

Computer systems often make people feel dumb - in fact, there are lots of "dummies" books available, such as "iPad for dummies" or "The Complete Idiot's Guide to Microsoft Windows 8". These books sell millions of copies, yet chances are the people who buy them are actually quite intelligent --- it's just that the interfaces can make people so frustrated that they feel like a dummy. The truth is that if an interface makes lots of people feel like an idiot, chances are the real problem is with the interface and not the user. In the past there has been a culture where the balance of power was with the programmers who designed a system, and they could afford to blame the users for any problems. However, now users have a large choice of systems, and there are competitors ready to offer a better interface, so if a

programmer continually blame your users for problems, chances are it's the programmer who is the complete idiot! If you hear people using derogatory terms such as luser, **PEBKAC**, or ID-10T error (Idiot error), they may be humorous, but they usually show a disregard for the importance of getting an interface right, and are a sign that the system is badly designed.

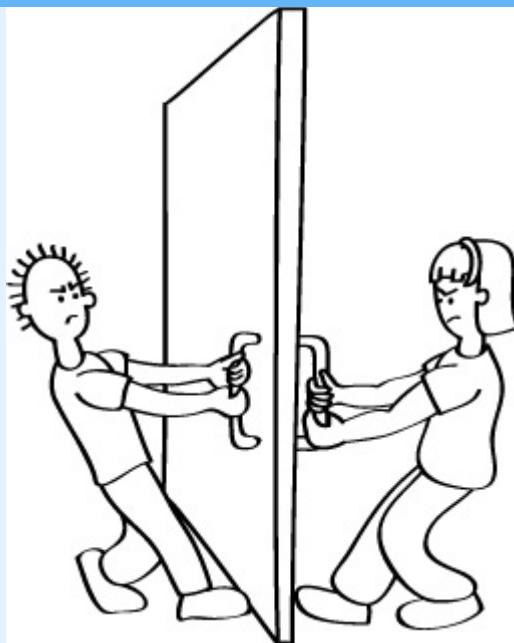
### **Project:** Sending an email from multiple devices

For this project, try sending an email from both a computer and a mobile phone. Take note of all the steps required from when you start using the device until the email is sent.

You will probably notice quite a few differences between the two interfaces.

Keep your notes for later, as you can further analyse them once you have read through more of this chapter.

### **Project:** Designing stovetops and door handles



For this project, you will design the top of a cooking stove, or the handles on a door. This isn't a computer system, but will help demonstrate some of the issues that come up. The main task is to sketch three different configurations for the stovetop which includes the arrangement of the 4 elements and the 4 control knobs.

The task is [described in detail in CS Unplugged human interface design activity](#).

## 4.3. Interface Usability

Devices are often sold using catch phrases like "user friendly" and "intuitive", but these are vague terms that are hard to pin down. In this section we will use the more technical term, **Usability**, which is well understood by HCI experts, and gives us some ways to evaluate how suitable an interface is for a particular task. Usability isn't just about an interface being nice to use: poor usability can lead to serious problems, and has been the cause of major disasters, such as airplane crashes, financial disasters, and medical mishaps. It is also important because an interface that requires a lot of dexterity, quick reactions or a good memory makes it less accessible to much of the population, when accessibility can be both a moral and legal expectation.

### Curiosity: When interface design goes horribly wrong

- 87 people were killed when [Air Inter Flight 148 crashed](#) due to the pilots entering "33" to get a 3.3 degree descent angle, but the same interface was used to enter the descent rate, which the autopilot interpreted as 3,300 feet per minute. This interface problem is called a "mode error" (described later). There is more information [here](#).
- 13 people died and many more were injured when the pilots of [Varig Flight 254](#) entered an incorrect heading. The flight plan had specified a heading of 0270, which the captain interpreted and entered into the flight computer as 270 degrees. What it actually meant was 027.0 degrees. This confusion came about due to the format of headings and the position of the decimal point on flight plans being changed without him knowing. Unfortunately, the co-pilot mindlessly copied the captain's heading instead of reading it off the flight plan like he was supposed to. The plane then cruised on autopilot for a few hours. Unfortunately, [confirmation bias](#) got the better of the pilots who were convinced they were near their destination, when in fact they were hundreds of miles away. The plane ran out of fuel and crash landed in the Amazon Jungle. Designing aircraft systems which work for humans is a big challenge, and is a part of the wider area of human factors research.
- A bank employee [accidentally gave a customer a loan of \\$10 million instead of \\$100,000](#). The customer withdrew most of the money and fled to Asia, the bank lost millions of dollars in the process, and the teller concerned would have had a traumatic time just because of a typing error. The error was due to the employee typing in two extra zeroes, apparently because some interfaces automatically put in the decimal point (you could type 524 to enter \$5.24), and others didn't.

This error can be explained in terms of a lack of consistency in the interface, causing a mode error.

- A 43-year old woman suffered respiratory arrest after a nurse accidentally entered 5 instead of 0.5 for a dose rate for morphine. The interface should have made it difficult to make an error by a factor of 10. There is a [paper about it](#), and an [article about the interface problem](#). Similar problems can occur in any control system where the operator types in a value; a better interface would force the operator to press an "up" and "down" button, so big changes take a lot of work (this is an example of an "off by one error", where one extra digit is missed or added, and also relates to the principle of commensurate effort)

In all these cases the fault could be blamed on the user (the pilots, the bank teller and the nurse) for making a mistake, but a well designed interface that doesn't cause serious consequences from mistakes that humans can easily make would be much better.

There are many elements that can be considered in usability, and we will mention a few that you are likely to come across when evaluating everyday interfaces. Bear in mind that the interfaces might not just be a computer – any digital device such as an alarm clock, air conditioning remote control, microwave oven, or burglar alarms can suffer from usability problems.

### 4.3.1. Consistency

A "golden rule" of usability is *consistency*. If a system keeps changing on you, it's going to be frustrating to use. Earlier we had the example of a "Even"/"Odd" button pair that occasionally swapped places. A positive example is the consistent use of "control-C" and "control-V" in many different programs to copy and paste text or images. This also helps *learnability*: once you have learned copy and paste in one program, you know how to use it in many others. Imagine if every program used different menu commands and keystrokes for this!

A related issue is the [Mode error](#), where the behaviour of an action depends on what mode you are in. A simple example is having the caps lock key down (particularly for entering a password, where you can't see the effect of the mode). A classic example is in Excel spreadsheets, where the effect of clicking on a cell depends on the mode: sometimes it selects the cell, and other times it puts the name of the cell you clicked on into another

cell. Modes are considered bad practice in interface design because they can easily cause the user to make the wrong action, and should be avoided if possible.

### 4.3.2. Response time

The speed at which an interface responds (its *reaction time*) has a significant effect on usability. The way that humans perceive time isn't always proportional to the time taken. If something happens fast enough, we will perceive it as being instant. If we have to wait and can't do anything while waiting, time can appear to go slower!

The following interactive lets you find out how fast "instant" is for you. As you click on each cell, there will sometimes be a random delay before it comes up; other cells won't have a delay. Click on each cell, and if it seems to respond instantly, leave it as it is. However, if you perceive that there is a small delay before the image comes up, click it again (which makes the cell green). Just make a quick, gut-level decision the first time you click each cell - don't overthink it. The delay may be very short, but only make the cell green if you are fairly sure you noticed a delay.

Use the interactive online at <http://www.csfieldguide.org.nz/en/interactives/delay-analyser/index.html>

Once you have clicked on all the cells, click on "View statistics" to see how long the delays were compared with your perception. 100 ms (100 milliseconds) is one tenth of a second; for most people this is where they are likely to start perceiving a delay; anything shorter (particularly around 50 ms) is very hard to notice. Longer delays (for example, 350 ms, which is over a third of a second) are very easy to notice.

The point of this is that any interface element (such as a button or checkbox) that takes more than 100 ms to respond is likely to be perceived by the user as not working, and they may even click it again. In the case of a checkbox, this may lead to it staying off (from the two clicks), making the user think that it's not working. Try clicking the following checkbox just enough times to make it show as selected.

Use the interactive online at <http://www.csfieldguide.org.nz/en/interactives/delayed-checkbox/index.html>

So, as you evaluate interfaces, bear in mind that even very small delays can make a system hard to use.

The following video is of an experiment that was done with virtual reality goggles to simulate Internet lag in real life situations. It has English captions, but the most interesting part is what is happening in the action.

Watch the video online at [https://www.youtube.com/watch?v=\\_fNp37zFn9Q](https://www.youtube.com/watch?v=_fNp37zFn9Q)

### 4.3.3. Human short term memory

Another important length of time to bear in mind is our *short term memory* time, which is usually a matter of seconds. To remember something for longer, the user needs to rehearse it (repeat it over) or make a note of the information, such as writing it down. If a system takes some time to respond (say, 10 seconds) then chances are the user may have forgotten some detail of what they were going to do with the system. For example, if you have a phone number to type in that someone has just told you, and it takes 12 seconds before you can type it, you may forget the number, whereas if you can access the interface within a couple of seconds, you can probably enter the number without much effort. For this reason, any part of a system that takes more than about 10 seconds to respond forces the user to rehearse or write down key information, which is more tiring.

There's some more information about "time limits" for interfaces in [this article by Jakob Nielsen](#).

### 4.3.4. Human spatial memory

Another important usability consideration is *spatial memory* – our ability to remember where things are located (such as where a button or icon is). Human spatial memory has a high capacity (you can probably remember the location of many places and objects), it is long lasting (people visiting a town they grew up in can often remember the layout), and we can remember things very quickly. A very simple aspect of usability that comes from this is that the layout of an interface shouldn't keep changing. The interactive task at the start of this chapter was deliberately set up to be frustrating by swapping the two buttons occasionally; the reason people often make a mistake in that situation is that their spatial memory takes over, so the location of the button is more important than what is written on it. Systems that aren't consistent in their spatial placement of the "OK" and "Cancel" buttons can easily cause people to press the wrong one.

Another place that the layout of an interface changes quickly is when a tablet or smartphone is rotated. Some devices rearrange the icons for the new orientation, which loses the spatial layout, while others keep them the same (except they may not look right in the new rotation). Try a few different devices and see which ones change the layout when rotated.

**Curiosity:** Common situations where layouts unexpectedly change

There are a number of other situations where the layout can change suddenly for the user, and create confusion. Here are some examples:

- The layout may change if a data projector is plugged in and the screen resolution changes (which is particularly frustrating because the user may well be about to present to an audience, and they can't find an icon, with the added awkwardness that lots of people are waiting).
- If you upgrade to a different size device (such as a larger monitor or different smartphone) you may have to re-learn where everything is.
- Layouts often change with new versions of software (which is one reason that upgrading every time a new version comes out may not be the best plan).
- Using the same software on a different operating system may have subtly different layout (e.g. if someone who uses the Chrome browser all the time on Windows starts using Chrome under MacOS). This can be particularly frustrating because the location of common controls (close/maximise window, and even the control key on the keyboard) is different, which frustrates the user's spatial memory.
- The Microsoft Word "ribbon" was particularly frustrating for users when it came out for several of the reasons already mentioned -- the position of each item was quite different to the previous versions.
- Adaptive interfaces can also be a problem; it might seem like a good idea to gradually change a menu in a program so that the frequently used items are near the top, or unused items are hidden, but this can lead to a frustrating treasure hunt for the user as they can't rely on their spatial memory to find things.

Associated with spatial memory is our *muscle memory*, which helps us to locate items without having to look carefully. With some practice, you can probably select a common button with a mouse just by moving your hand the same distance that you always have, rather than having to look carefully. Working with a new keyboard can mean having to re-learn the muscle memory that you have developed, and so may slow you down a bit or cause you to press the wrong keys.

### 4.3.5. Missing the button

One common human error that an interface needs to take account of is the *off by one error*, where the user accidentally clicks or types on an item next to the one they intended. For example, if the "save" menu item is next to a "delete" menu item, that is risky because one small slip could cause the user to erase a file instead of saving it. A similar issue occurs on keyboards; for example, control-W might close just one window in a web browser, and

control-Q might close the entire web-browser, so choosing these two adjacent keys is a problem. Of course, this can be fixed by either checking if the user quits, or by having all the windows saved so that the user just needs to open the browser again to get their work back. This can also occur in web forms, where there is a reset button next to the submit button, and the off-by-one error causes the user to lose all the data they just entered.



### 4.3.6. Deliberately making tasks more challenging

Another idea used by HCI designers is the *principle of commensurate effort*, which says that frequently done simple tasks should be easy to do, but it's ok to require a complex procedure for a complex task. For example, in a word processor, printing a page as it is displayed should be easy, but it's ok if some effort is required to make it double sided, two to a page, with a staple in the top left corner. In fact, sometimes more effort should be required if the command has a serious consequence, such as deleting a file, wiping a device, or closing an account. In such cases artificial tasks may be added, such as asking "Are you sure?", or to get an extreme setting on a device (like setting a voltage for a power supply) might require pressing an "up" button many times, rather than letting the user type in an extra couple of zeroes.

Use the interactive online at <http://www.csfieldguide.org.nz/en/interactives/action-menu/index.html>

### 4.3.7. In summary

These are just a few ideas from HCI that will help you to be aware of the kinds of issues that interfaces can have. In the following project you can observe these kinds of problems firsthand by watching *someone else* use an interface, noting any problems they have. It's much easier to observe someone else than do this yourself, partly because it's hard to concentrate on the interface and take notes at the same time, and partly because you might already know the interface and have learned to overcome some of the less usable features.

#### Project: Think aloud protocol

In a think aloud protocol, you observe someone else using the interface that you want to evaluate, and encourage them to explain what they're thinking at each step. You'll take notes on what they say, and you can reflect on that afterwards to evaluate the interface (it can be helpful to record the session if possible.)

This protocol gives insights into what might be confusing in an interface, and why.

For example, if someone setting an alarm clock says "I'm pressing the up button until I get to 7am - oh bother, it stopped at 7:09, now I have to go right around again", that gives some insight into how the interface might get in the way of the users completing a task efficiently.

This approach is focussed on observing a user doing a particular *task*, to capture what happens in reality when people use an interface. *Tasks* are often confused with *features*; you use the features of a device to accomplish a task. For example, a camera might have a feature of taking multiple photos quickly, but a relevant task is more likely to be to "take a photo of an action event, choose the best photo, and share it". This could involve a number of user actions: getting into the multi-photo mode, configuring the camera for the lighting conditions, taking the photos, choosing the best one, connect to a computer, transfer the photo to a website, and share it with friends.

The project will be more interesting if your helper isn't completely familiar with the system, or if it's a system that people often find confusing or frustrating. Your writeup could be used to make decisions about improving the system in the future.

The task could be things like setting the time on a clock, finding a recently dialled number on an unfamiliar phone, or choosing a TV program to record.

To do the evaluation, you should give the device to your helper, explain the task to them, and ask them to explain what they are thinking at each step. Your helper may not be used to doing that, so you can prompt them as they go with questions like:

- what are you going to do now? Why?
- why did you choose that button?
- what are you looking for?
- are you having difficulty? What's the problem?
- can you see what went wrong?
- how are you feeling about this?

If they get the hang of "thinking aloud", just keep quiet and take notes on what they say.

It's very important not to criticise or intimidate the helper! If they make a mistake, try to figure out how the interface made them do the wrong thing, rather than blaming them. Any mistakes they make are going to be valuable for your project! If they get everything right, it won't be very interesting.

Once you've noted what happened, go over it, looking for explanations for where the user had difficulty. The examples earlier in the chapter will help you to be sensitive to how interfaces can frustrate the user, and find ways that they could be improved.

There's some [more information about think-aloud protocols on Wikipedia](#), on [Nielsen's web site](#), and [some notes put together by HCI students](#).

### Project: Cognitive walk-through

Another way of evaluating an interface is a "Cognitive Walkthrough". This is normally done without involving someone else, although the description here has been adapted to involve another user to make it a bit easier if you're not experienced at HCI evaluation. The *cognitive walkthrough* is a technique that HCI experts use to do a quick evaluation of an interface. It is particularly useful for evaluating interfaces with few steps, that are being used by new or occasional users (such as someone using a car park ticket machine at an airport, setting an alarm clock in a hotel room, or using a museum display).

The first step is to choose a typical task that someone might do with the interface being evaluated (such as get a 2-hour ticket, set the alarm for 5:20am, or find out where a particular display is in a museum).

The goal of the cognitive walkthrough is to identify if the user can see what to do at each step, and especially to notice if there is anything that is confusing or ambiguous (like which button to press next), and to find out if they're confident that the right thing happened.

An experienced HCI evaluator would do this on their own, imagining what a user would do at each step, but it's may be easier for you to do this with someone else using the interface, because that lets you see the interface through someone else's eyes. So we recommend asking a friend to go through a task for you.

The task only needs to have about 3 or 4 steps (e.g. button presses), as you'll be asking three questions at each step and taking notes on their responses, so it could take a while. You should know how to do the task yourself as we'll be focussing on the few steps needed to accomplish the task; if the user goes off track, you can put them back on task rather than observe them trying to recover from an HCI problem that shouldn't have been there in the first place. The task might be something like recording a 10-second video on a mobile phone, deleting a text message, or setting a microwave oven to reheat food for 45 seconds.

Present the interface to your helper without giving any instructions on using it, and tell them what the goal of the task is. Before they take any action, ask:

- do you know what to try to do at this step? Then have them look at the interface, and ask:
- can you see how to do it? Then have them take the action they suggested, and ask:
- are you able to tell that you did the right thing?

If their decisions go off track, you can reset the interface, and start over, explaining what to do for the step they took wrong if necessary (but noting that this wasn't obvious to them --- it will be a point to consider for improving the interface.)

Once the first action has been completed, repeat this with the next action required (it might be pressing a button or adjusting a control). Once again, ask the three questions above in the process.

In practice the second question (can you see how to do it?) is usually split into two: do they notice the control at all, and if so, do they realise that it's the one that is needed? For this exercise we'll simplify it to just one question.

[More details of how to do a cognitive walkthrough are on the cs4fn site.](#)

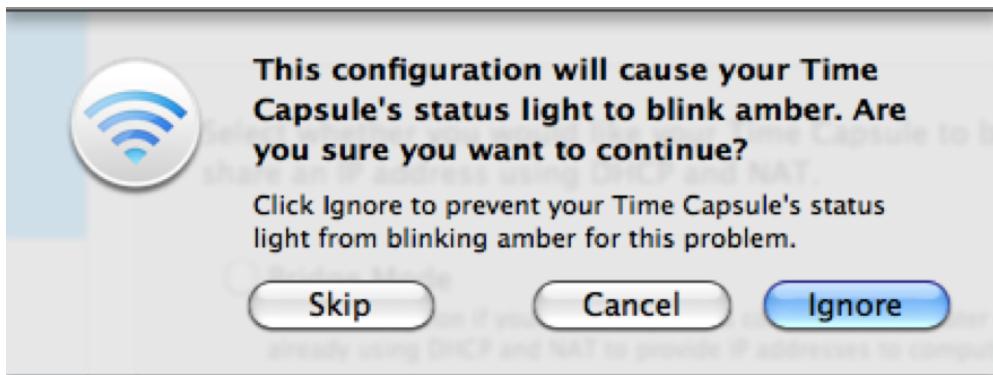
There is also more information in the [Wikipedia entry for Cognitive Walkthrough](#).

## 4.4. Usability Heuristics

Evaluating an interface is best done by getting feedback from having lots of potential users try it out. However, this can be expensive and time-consuming, so HCI experts have come up with some quick rules of thumb that help us spot obvious problems quickly. The formal word for a rule of thumb is a *heuristic*, and in this section we will look at some common heuristics that can be used to critique an interface.

There are various sets of heuristics that people have proposed for evaluating interfaces, but a Danish researcher called Jakob Nielsen has come up with a set of 10 heuristics that have become very widely used, and we will describe them in this section. If you encounter a usability problem in an interface, it is almost certainly breaking one of these heuristics, and possibly a few of them. It's not easy to design a system that doesn't break any of the heuristics, and sometimes you wouldn't want to follow them strictly – that's why they are called heuristics, and not rules.

Often a confusing feature in an interface design will break multiple heuristics. For example, the following error message (it is for real) doesn't help users recover from errors (the real error is a network setting, but it is explained as causing a flashing light!), and the "Skip", "Cancel" and "Ignore" buttons don't speak the user's language (match between the system and the real world).



#### 4.4.1. Visibility of system status

*The system should always keep users informed about what is going on, through appropriate feedback within reasonable time.*

This heuristic states that a user should be able to see what the device is doing (the system's status), at all times. This varies from the user being able to tell if the device is turned on or off, to a range of actions. A classic example is the "caps lock" key, which may not clearly show if it is on, and when typing a password the user might not know why it is being rejected; a positive example of this is when a password entry box warns you that the caps lock key is on.

One of the simplest statuses for a device is on or off, which is usually a coloured light on the outside of the computer. However, some devices take a while to show the status (for example, some DVD players take a while to respond when switched on), and the user might press the power button again or otherwise get confused about the status.

There are many tasks that users ask computers to do that require some time including copying documents, downloading files, and loading video games. In this situation, one of the most common ways to keep a user informed of the task is the progress bar.

However, progress indicators aren't always helpful; the spinning wheels above don't indicate if you are going to have to wait a few seconds or a few minutes (or even hours) for the task to complete, which can be frustrating.



Giving feedback in a "reasonable time" is really important, and the "reasonable time" is often shorter than what you might think. In the section above there was an experiment to find out at what point people perceive a delayed reaction; you probably found that it was around a tenth of a second. If a computer takes longer than that to respond then it can be confusing to use. There's more about this in the previous section.



[Image source](#)

There are some other important delay periods in interface evaluation: a delay of around 1 second is where natural dialogues start to get awkward, and around 10 seconds puts a lot of load on the user to remember what they were doing. Nielsen has an [article about the importance of these time periods](#). If you want to test these ideas, try having a conversation with someone where you wait 3 seconds before each response; or put random 10 second delays in when you're working on a task!

And if you haven't tried it already, have a go at the following interactive.

Use the interactive online at <http://www.csfieldguide.org.nz/en/interactives/delay-analyser/index.html>

Getting computers to respond quickly often depends on the algorithms used (covered in the chapter on algorithms), and can also depend on the design of a program (such as whether it stores data on disk or waits for a network response before continuing). It is particularly noticeable on small devices like smartphones, which have limited computing power, and might take a second or two to open an app or respond to some input. It's not too hard to find these sorts of delays in systems when you're evaluating them.

## 4.4.2. Match between system and the real world

*The system should speak the users' language, with words, phrases and concepts familiar to the user, rather than system-oriented terms. Follow real-world conventions, making information appear in a natural and logical order.*

The language, colours and notation in an interface should match the user's world, and while this seems obvious and sensible, it's often something that is overlooked. Take for example the following two buttons – can you see what is confusing about them?

Use the interactive online at <http://www.csfieldguide.org.nz/en/interactives/confused-buttons/index.html>

The following interface is from a bank system for paying another person. Suppose you get an email asking someone to pay you \$1699.50 dollars for a used car; try entering "\$1699.50" into the box.

Use the interactive online at <http://www.csfieldguide.org.nz/en/interactives/payment-interface/index.html>

The notation "\$1699.50" is a common way to express a dollar amount, but this system forces you to follow its own conventions (probably to make things easier for the programmer who wrote the system).

Try find some other amounts which should be valid, but this system seems to reject. Ideally, the system should be flexible with the inputted text to prevent errors.

**Spoiler:** Answers to above, try it before reading this!

The dialogue also rejects commas in the input e.g. "1,000", even though they are a very useful way to read dollar amounts, e.g. it's hard to differentiate between 1000000 and 100000, yet this could make a huge difference! It also doesn't allow you to have a space before or after the number, yet if the number has been copied and pasted from an email, it might look perfectly alright. A less lazy programmer would allow for these

situations; the current version is probably using a simple number conversion system that saves having to do extra programming...

### 4.4.3. User control and freedom

*Users often choose system functions by mistake and will need a clearly marked "emergency exit" to leave the unwanted state without having to go through an extended dialogue.*

*Support undo and redo.*

It is very frustrating to make a mistake and not be able to get out of it. It is particularly bad if one small action can wipe a lot of work that can't be recovered. The reset button on some web forms is infamous for this – it is often next to the submit button, and you can wipe all your data with an off-by-one error.

A common way to provide user freedom is an "undo" feature, which means that not only can mistakes be fixed easily, but the user is encouraged to experiment, trying out features of the interface secure in the knowledge that they can just "undo" to get back to how things were, instead of worrying that they'll end up in a state that they can't fix. If "redo" is also available, they can flick back and forth, deciding which is best. (In fact, redo is really an undo for undo!)

Here's an example of a button that doesn't provide user control; if you press it, you'll lose this whole page and have to find your way back (we warned you!)

Use the interactive online at <http://www.csfieldguide.org.nz/en/interactives/close-window/index.html>

Sometimes the interface can force the user into doing something they don't want to do. For example, it is quite common for operating systems or programs to perform updates automatically that require a restart. Sometimes the interface may not give them the opportunity to cancel or delay this, and restart nevertheless. This is bad if it happens when the user is just about to give a presentation.

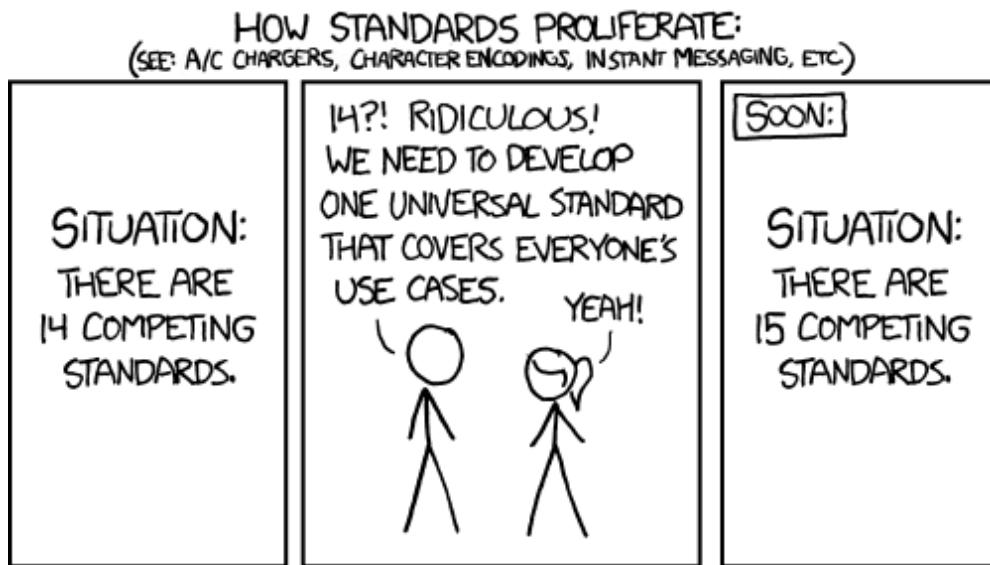
Another common form of this problem is not being able to quit a system. A positive example is the "home" button on smartphones, which almost always stops the current app that is in use.

### 4.4.4. Consistency and standards

*Users should not have to wonder whether different words, situations, or actions mean the same thing. Follow platform conventions.*

Consistency (something being the same every time) is extremely useful for people using interfaces, and is sometimes called the "golden rule of HCI". If an interface is consistent with other interfaces then learning in one interface transfers directly to another. One of the biggest examples of consistency in computer programs is copy and paste, which works the same way in most software, so users only have to learn the concept once. The shortcut keys for copy and paste are also fairly consistent between programs. But in some software, copy/paste behaves differently, and this can be confusing for users.

An example of inconsistency is generally found within spreadsheet programs, where the result of pushing "control-A" (select all) depends on whether you are editing a cell or just have the cell selected (this particular problem is a 'mode' problem). While this may make sense to a user experienced with spreadsheets, a new user can be very confused when the same action causes a different response.



[Image source](#)

A lack of consistency is often the reason behind people not liking a new system. It is particularly noticeable between Mac and Windows users; someone who has only used one system can find the other very frustrating to use because so many things are different (consider the window controls for a start, which are in a different place and have different icons). An experienced user of one interface will think that it is "obvious", and can't understand why the other person finds it frustrating, which can lead to discussions of religious fervour on which interface is best. Similar problems can occur when a radically different version of an operating system comes out (such as Windows 8); a lot of the learning that has been done on the previous system needs to be undone, and the lack of consistency (i.e. losing prior learning) is frustrating.

## 4.4.5. Error prevention

*Even better than good error messages is a careful design which prevents a problem from occurring in the first place. Either eliminate error-prone conditions or check for them and present users with a confirmation option before they commit to the action.*

A computer program shouldn't make it easy for people to make serious errors. An example of error prevention found in many programs is a menu item on a toolbar or dropdown being 'greyed out' or deactivated. It stops the user from using a function that shouldn't be used in that situation, like trying to copy when nothing is selected. A good program would also inform the user why an item is not available (for example in a tooltip).

Below is a date picker; can you see what errors can be produced with it?

Use the interactive online at <http://www.csfieldguide.org.nz/en/interactives/date-picker/index.html>

**Spoiler:** Some of the errors you might have observed

The date picker allows the user to choose invalid dates, such as Feb 30, or Nov 31. The three-menu date picker is hard to get right, because each menu item limits what can be in the others, but any can be changed. For example, you might pick 29 Feb 2008 (a valid date), then change the year to 2009 (not valid), then back to 2008. When 2009 was chosen the day number would need to change to 28 to prevent errors, but if that was just an accident and the user changes back to 2008, the number has now changed, and might not be noticed. It's preferable to use a more sophisticated date picker that shows a calendar, so the user can only click on valid dates (many websites will offer this). Date picking systems usually provide a rich example for exploring interface issues!

A related problem with dates is when a user needs to pick a start and end date (for example, booking flights or a hotel room); the system should prevent a date prior to the first date being selected for the second date.

Here's a menu system that offers several choices:

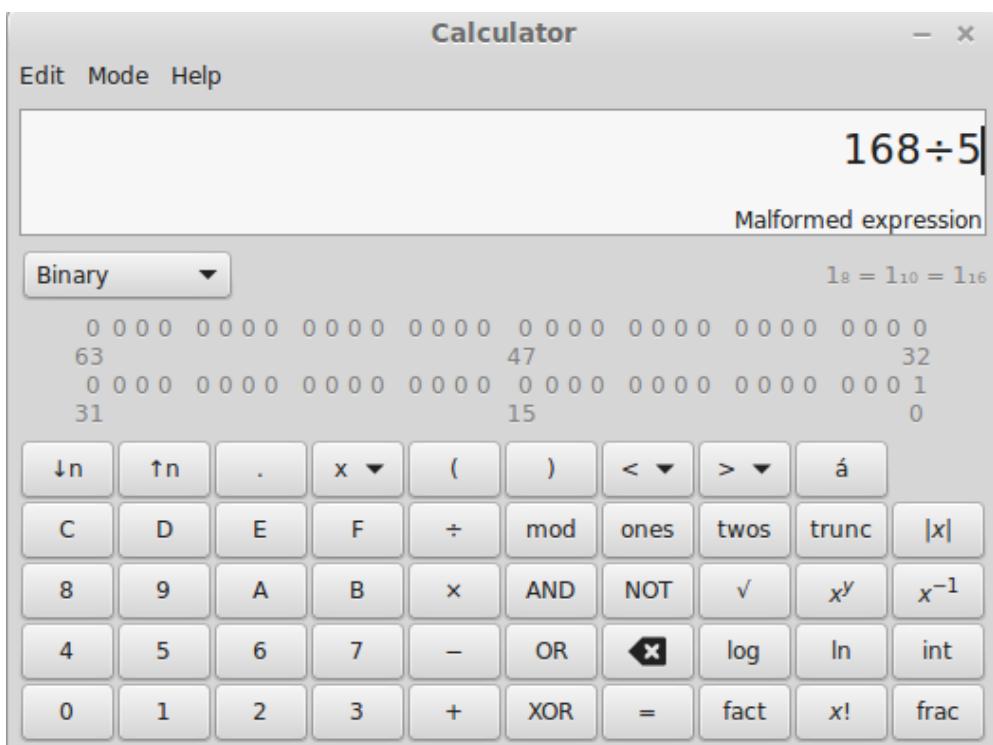
Use the interactive online at <http://www.csfieldguide.org.nz/en/interactives/available-menu-items/index.html>

Any time a dialogue box comes up that says you weren't allowed to do a certain action, it's frustrating because the system has failed to prevent an error. Of course, it may be difficult

to do that because the error can depend on so many user choices, but it is ideal that the system doesn't offer something that it can't do.

Another example of preventing errors is an automatic teller machine (ATM) that can only dispense, say, \$20 notes. If it allows you to enter any amount (such as \$53.92, or even \$50) then an error would be quite likely. What techniques have you seen used in ATM software to prevent this kind of error?

And here's another example, this time with a computer science slant: the following calculator has a binary mode, where it does calculations on binary numbers. The trouble is that in this mode you can still type in decimal digits, which gives an error when you do the calculation. A user could easily not notice that it's in binary mode, and the error message isn't particularly helpful!



#### 4.4.6. Recognition rather than recall

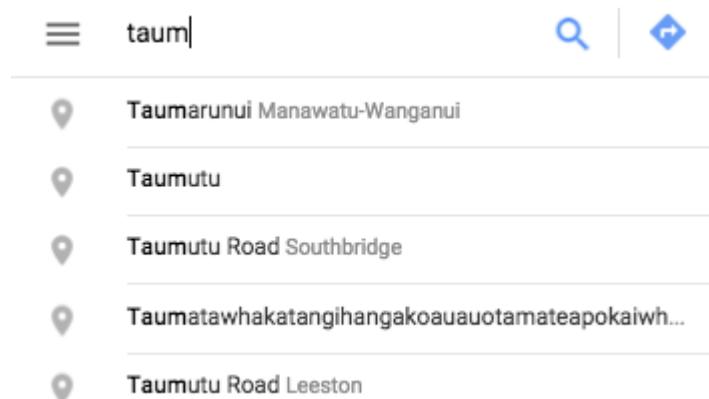
*Minimize the user's memory load by making objects, actions, and options visible. The user should not have to remember information from one part of the dialogue to another.*

*Instructions for use of the system should be visible or easily retrievable whenever appropriate.*

Humans are generally very good at recognising items, but computers are good at remembering them accurately. A good example of this is a menu system; if you click on the "Edit" menu in a piece of software, it will remind you of all the editing tasks available, and you can choose the appropriate one easily. If instead you had to type in a command from

memory, that would put more load on the user. In general it's good for the computer to "remember" details, and the user to be presented with options rather than having to remember them. The exception is a system that is used all the time by an expert who knows all the options; in this case entering commands directly can sometimes be more flexible and faster than having to select from a list.

For example, when you type in a place name in an online map, the system might start suggesting names based on what you're typing, and probably adapted to focus on your location or past searches. The following image is from Google maps, which suggests the name of the place you may be trying to type (in this case, the user has only had to type 4 letters, and the system saves them from having to recall the correct spelling of "Taumatawhakatangihangakoauauotamateapokaiwhenuakitanatahu" because they can then select it.) A similar feature in web browsers saves users from having to remember the exact details of a URL that they have used in the past; a system that required you to type in place names exactly before you could search for them could get rather frustrating.



#### 4.4.7. Flexibility and efficiency of use

*Accelerators -- unseen by the novice user -- may often speed up the interaction for the expert user such that the system can cater to both inexperienced and experienced users. Allow users to tailor frequent actions.*

When someone is using software every day, they will soon have common sequences of operations they do (such as "Open the file, find the next blank space, type in a record of what just happened"). It's good to offer ways to make this quick to do, such as "macros", which do a sequence of actions from a single keystroke.

Similarly, it's good to be able to customise software by allocating keystrokes for frequent actions (such as "file this email in the 'pending' folder"). Common tasks like copy and paste usually have keystrokes added to them, and these allow experienced users to perform the task without having to reach for a mouse.

An important area of research in HCI is working out how to make shortcuts easy to learn. You don't want them to get in the way for beginners, but you don't want frequent users to be unaware of them either. A simple way of doing this is having keystroke equivalents in a menu (an accelerator); the menu displayed here shows that shift-command-O will open a new project, so the user can learn this sequence if they are using the command frequently.

File	Edit	Selection	Find	\
New Window		⇧⌘N		
New File		⌘N		
Open...		⌘O		
Add Project Folder...		⇧⌘O		
Reopen Last Item		⇧⌘T		
Save		⌘S		
Save As...		⇧⌘S		
Save All		⌥⌘S		
Close Tab		⌘W		
Close Pane				
Close Window		⇧⌘W		
Close All Tabs				

A flexible system would allow the user to add a keystroke equivalent for the "Close Pane" command themselves, if that turned out to be used frequently. Other systems might offer suggestions to the user if they notice an action being done frequently. A related approach is offering recent selections near the top of a list of options.

#### 4.4.8. Aesthetic and minimalist design

*Dialogues should not contain information which is irrelevant or rarely needed. Every extra unit of information in a dialogue competes with the relevant units of information and diminishes their relative visibility.*

Software can contain many features, and if they are all visible at the same time (for example, on a toolbar), this can be overwhelming, especially for a new user.

TV remote controls often provide a great example of a complicated interface. One reason that they have so many buttons is that it can help to make the device look impressive in the shop, but once you get it home, many of the buttons become redundant or confusing.



The remote control shown here has several buttons that could potentially do the same thing: "Direct Navigator", "Guide", "Function Menu", "Status" and "Option" all give access to different functions, but it's hard to predict which is which. This remote has about 55 buttons altogether!



In contrast, the Apple remote has very few buttons, and is a good example of a minimalist interface. There's only one "Menu" to choose, so it's fairly obvious what to do to select the controls needed. Of course, the simple remote relies on displaying menus on the screen, and these have the potential to make things more complicated.



The third remote control shows a solution for simplifying it to save the user from having to read extensive manual information. It's a bit drastic, but it might save the user from getting into modes that they can't get out of! Some people have reported removing keys from mobile phones, or gluing buttons in place, so that the user can't get the device into a state that they shouldn't. Some controls try to offer the best of both worlds by having a small flap that can be opened to reveal more functionality.

#### Curiosity: Scary interfaces

The following site identified some of the "scariest" interfaces around, some of which are great examples of *not* having minimalist design: [OK/Cancel scariest interface](#).

Cartoonist [Roz Chast](#) illustrates how scary a remote control can be with her cartoon "[How Grandma sees the remote](#)".

#### 4.4.9. Help users recognize, diagnose, and recover from errors

*Error messages should be expressed in plain language (no codes), precisely indicate the problem, and constructively suggest a solution.*

It's not hard to find error messages that don't really tell you what's wrong! The most common examples are messages like "Misc error", "Error number -2431", or "Error in one of

the input values". These force the user to go on a debugging mission to find out what went wrong, which could be anything from a disconnected cable or unfixable compatibility issue, to a missing digit in a number.

For example, some troubleshooting software produced the "unexpected" error below. The error message is particularly unhelpful because the software was supposed to help with finding problems, but instead it has given the user a new problem to solve! There is some extra information not shown below such as "Path: Unknown" and "Error code: 0x800070002". Searching for the error code can lead to suggested solutions, but it also leads to scam software that claims to fix the problem. By not giving useful error recovery information, the system has put the user at the mercy of the advice available online!



[An error occurred while troubleshooting:](#)

An unexpected error has occurred. The troubleshooting wizard can't continue.

A variant of unhelpful error messages is one that gives two alternatives, such as "File may not exist, or it may already be in use". A better message would save the user having to figure out which of these is the problem.

A positive example can be found in some alarm clocks such as the following one on an Android smartphone. For example, here the alarm time is shown at "9:00". In a country that uses 12-hour time, a user might mistake this for 9pm, and the alarm would go off at the wrong time.



However, the interface provides an opportunity to notice it because the display indicates how long it will be until the alarm will go off, making it easier to recognize the error of selecting the wrong time (or day).

Alarm set for 12 hours and 57 minutes from now.  
Alarm already set for 09:00.

## 4.4.10. Help and documentation

Even though it is better if the system can be used without documentation, it may be necessary to provide help and documentation. Any such information should be easy to search, focused on the user's task, list concrete steps to be carried out, and not be too large. The following interactive illustrates a situation you might have run into before!

Use the interactive online at <http://www.csfieldguide.org.nz/en/interactives/no-help/index.html>

Often help is an afterthought, and tends to be feature-centred (e.g. a catalogue of menu items), rather than task-centred (the series of actions needed to complete typical tasks, which is more useful for the user). When a user needs help, they typically have a task to complete (such as upload photos from a camera), and good documentation should explain how to do common tasks, rather than explain each feature (such as "Setting the camera to USB mode").

## 4.4.11. To find out more about heuristics

You can find more information about the [heuristics online on Jakob Nielsen's website](#).

# 4.5. The whole story!

In this chapter we've mainly looked at how to critique interfaces, but we haven't said much about how to design good interfaces. That's a whole new problem, although being able to see what's wrong with an interface is a good basis for designing good interfaces. Many commercial systems are tested using the ideas above to check that people will find them easy to use; in fact, before releasing a new application, often they are tested many times with many users. Improvements are made, and then more tests need to be run to check that the improvements didn't make some other aspect of the interface worse! It's no wonder that good software can be expensive – there are many people and a lot of time involved in making sure that it's easy to use before it's released.

There are many other ideas from psychology, physiology, sociology and even anthropology that HCI experts must draw on. Things that come into play include [Mental models](#), about how someone believes a system works compared with how it actually works (these are almost never the same e.g. double clicking on an icon that only needs to be single clicked), [Fitts's law](#), about how long it takes to point to objects on a screen (such as clicking on a small button), the [Hick-Hyman law](#), about how long it takes to make a decision between multiple choices (such as from a menu), [Miller's law](#) about the number of items a person can think about at once, [affordances](#), about how properties of an object help us to perform actions on them, [interaction design \(IxD\)](#), about creating digital devices that work for the people who will use the product, [the NASA TLX \(Task Load Index\)](#) for rating the perceived workload that a task puts on a user, and many more laws, observations and guidelines about designing interfaces that take account of human behaviour and how the human body functions.

### 4.5.1. Further reading

- The book "Designing with the mind in mind" by Jeff Johnson provides excellent background reading for many of the issues discussed in this chapter
- The [cs4fn website has a lot of articles and activities on Human Computer Interaction](#), such as [problems around reporting interface problems](#), [cultural issues in interface design](#), and [The importance of Sushi](#).
- A classic book relating to usability is "The psychology of everyday things" (later re-titled "The design of everyday things") by Don Norman. It's about everyday objects like doors and phones, and it was written a while ago, but it contains lots of thought provoking and often humorous examples.

### 4.5.2. Useful Links

- [The ten usability heuristics on Nielsen's website](#), and a [collection of articles about usability heuristics](#)
- There is a [CS Unplugged activity on HCI](#) which includes background information
- There is [extensive material on HCI on the cs4fn website](#)
- A [glossary of usability terms](#)

# 5. Data Representation

## 5.1. What's the big picture?

Computers are machines that do stuff with information. They let you view, listen, create, and edit information in documents, images, videos, sound, spreadsheets and databases. They let you play games in simulated worlds that don't really exist except as information inside the computer's memory and displayed on the screen. They let you compute and calculate with numerical information; they let you send and receive information over networks. Fundamental to all of this is that the computer has to represent that information in some way inside the computer's memory, as well as storing it on disk or sending it over a network.

To make computers easier to build and keep them reliable, everything is represented using just two values. You may have seen these two values represented as 0 and 1, but on a computer they are represented by anything that can be in two states. For example, in memory a low or high voltage is used to store each 0 or 1. On a magnetic disk it's stored with magnetism (whether a tiny spot on the disk is magnetised north or south).

The idea that *everything* stored and transmitted in our digital world is stored using just two values might seem somewhat fantastic, but here's an exercise that will give you a little experience using just black and white cards to represent numbers. In the following interactive, click on the last card (on the right) to reveal that it has one dot on it. Now click on the previous card, which should have two dots on it. Before clicking on the next one, how many dots do you predict it will have? Carry on clicking on each card moving left, trying to guess how many dots each has.

Use the interactive online at <http://www.csfieldguide.org.nz/en/interactives/binary-cards/index.html?digits=5&start=BBBBB>

The challenge for you now is to find a way to have exactly 22 dots showing (the answer is in the spoiler below). Now try making up other numbers of dots, such as 11, 29 and 19. Is there any number that can't be represented? To test this, try counting up from 0.

**Spoiler:** Solution to card puzzles

You may have noticed that each card shows twice as many dots as the one to its right. This is an important pattern in data representation on computers.

The number 22 requires the cards to be "white, black, white, white, black", 11 is "black, white, black, white, white", 29 is "white, white, white, black, white", and 19 is "white, black, black, black, white".

You should have found that any number from 0 to 31 can be represented with 5 cards. Each of the numbers could be communicated using just two words: black and white. For example, 22 dots is "white, black, white, white, black". Or you could decode "black, black, white, white, white" to the number 7. This is the basis of data representation - anything that can have two different states can represent anything on a digital device.

When we write what is stored in a computer on paper, we normally use "0" for one of the states, and "1" for the other state. For example, a piece of computer memory could have the following voltages:

```
low low high low high high high low high low low
```

We could allocate "0" to "low", and "1" to "high" and write this sequence down as:

```
0 0 1 0 1 1 1 0 1 0 0
```

While this notation is used extensively, and you may often hear the data being referred to as being "0's and 1's", it is important to remember that a computer does *not* store 0's and 1's; it has no way of doing this. They are just using physical mechanisms such as high and low voltage, north or south polarity, and light or dark materials.

### Jargon Buster: Bits

The use of the two digits 0 and 1 is so common that some of the best known computer jargon is used for them. Since there are only two digits, the system is called binary. The short word for a "binary digit" is made by taking the first two letters and the last letter --- a *bit* is just a digit that can have two values.

Every file you save, every picture you make, every download, every digital recording, every web page is just a whole lot of bits. These binary digits are what make digital technology

*digital!* And the nature of these digits unlock a powerful world of storing and sharing a wealth of information and entertainment.

Computer scientists don't spend a lot of time reading bits themselves, but knowing how they are stored is really important because it affects the amount of space that data will use, the amount of time it takes to send the data to a friend (as data that takes more space takes longer to send!) and the quality of what is being stored. You may have come across things like "24-bit colour", "128-bit encryption", "32-bit IPv4 addresses" or "8-bit ASCII". Understanding what the bits are doing enables you to work out how much space will be required to get high-quality colour, hard-to-crack secret codes, a unique ID for every device in the world, or text that uses more characters than the usual English alphabet.

This chapter is about some of the different methods that computers use to code different kinds of information in patterns of these bits, and how this affects the cost and quality of what we do on the computer, or even if something is feasible at all.

## 5.2. Getting Started

To begin with, we'll look at Braille. Braille is not actually a way that computers represent data, but is a great introduction to the topic.

### **Additional Information:** Representing Braille without making holes in paper

When working through the material in this section, a good way to draw braille on paper without having to actually make raised dots is to draw a rectangle with 6 small circles in it, and to colour in the circles that are raised, and not colour in the ones that aren't raised.

### 5.2.1. What is Braille?

More than 200 years ago a 15-year-old French boy invented a system for representing text using combinations of flat and raised dots on paper so that they could be read by touch. The system became very popular with people who had visual impairment as it provided a relatively fast and reliable way to "read" text without seeing it. Louis Braille's system is an early example of a "binary" representation of data --- there are only two symbols (raised and flat), and yet combinations of them can be used to represent reference books and works of literature. Each character in braille is represented with a cell of 6 dots. Each dot can either be raised or not raised. Different numbers and letters can be made by using different patterns of raised and not raised dots.

<b>Braille Alphabet</b>																											
The six dots of the braille cell are arranged and numbered:																											
																											
The capital sign, dot 6, placed before a letter makes a capital letter.																											
																											
The number sign, dots 3, 4, 5, 6 placed before the characters a through j, makes the numbers 1 through 0. For example a preceded by the number sign is 1, b is 2, etc.																											
																											
<table border="1"> <thead> <tr> <th>Capital Sign</th> <th>Number Sign</th> <th>Period</th> <th>Comma</th> <th>Question Mark</th> <th>Semi-colon</th> <th>Exclamation point</th> <th>Opening quote</th> <th>Closing quote</th> </tr> </thead> <tbody> <tr> <td>•</td> <td>••</td> <td>•••</td> <td>••••</td> <td>•••••</td> <td>••••••</td> <td>•••••••</td> <td>••••••••</td> <td>•••••••••</td> </tr> </tbody> </table>										Capital Sign	Number Sign	Period	Comma	Question Mark	Semi-colon	Exclamation point	Opening quote	Closing quote	•	••	•••	••••	•••••	••••••	•••••••	••••••••	•••••••••
Capital Sign	Number Sign	Period	Comma	Question Mark	Semi-colon	Exclamation point	Opening quote	Closing quote																			
•	••	•••	••••	•••••	••••••	•••••••	••••••••	•••••••••																			
National Braille Press copyright 2000																											

Let's work out how many different patterns can be made using the 6 dots in a Braille character. If braille used only 2 dots, there would be 4 patterns. And with 3 dots there would be 8 patterns

## Two dots

- 1 
- 2 
- 3 
- 4 

## Three dots

- |   |   |
|---|---|
| 1   | 5   |
| 2  | 6  |
| 3  | 7  |
| 4  | 8  |

You may have noticed that there are twice as many patterns with 3 dots as there are with 2 dots. It turns out that every time you add an extra dot, that gives twice as many patterns, so with 4 dots there are 16 patterns, 5 dots has 32 patterns, and 6 dots has 64 patterns. Can you come up with an explanation as to why this doubling of the number of patterns occurs?

**Spoiler:** Why does adding one more dot double the number of possible patterns?

The reason that the number of patterns doubles with each extra dot is that with, say, 3 dots you have 8 patterns, so with 4 dots you can use all the 3-dot patterns with the 4th dot flat, and all of them with it raised. This gives 16 4-dot patterns. And then, you can do the same with one more dot to bring it up to 5 dots. This process can be repeated infinitely.

So, Braille, with its 6 dots, can make 64 patterns. That's enough for all the letters of the alphabet, and other symbols too, such as digits and punctuation.

## 5.2.2. So how does Braille relate to data representation?

The reason we're looking at Braille in this chapter is because it is a representation using bits. That is, it contains 2 different values (raised and not raised) and contains sequences of these to represent different patterns. The letter m, for example, could be written as 110010, where "1" means raised dot, and "0" means not raised dot (assuming we're reading from left to right and then down). This is the same as how we sometimes use 1's and 0's to show how a computer is representing data.

Braille also illustrates why binary representation is so popular. It would be possible to have three kinds of dot: flat, half raised, and raised. A skilled braille reader could distinguish them, and with three values per dot, you would only need 4 dots to represent 64 patterns. The trouble is that you would need more accurate devices to create the dots, and people would need to be more accurate at sensing them. If a page was squashed, even very slightly, it could leave the information unreadable.

Digital devices almost always use two values (binary) for similar reasons: computer disks and memory can be made cheaper and smaller if they only need to be able to distinguish between two extreme values (such as a high and low voltage), rather than fine-grained distinctions between very subtle differences in voltages. Using ten digits (like we do in our every day decimal counting system) would obviously be too challenging.

### Curiosity: Decimal-based computers

Why are digital systems so hung up on only using two digits? After all, you could do all the same things with a 10 digit system?

As it happens, people have tried to build decimal-based computers, but it's just too hard. Recording a digit between 0 and 9 involves having accurate equipment for reading voltage levels, magnetisation or reflections, and it's a lot easier just to check if it's mainly one way or the other.

There's a more in-depth discussion on why we use binary here:

Watch the video online at <https://www.youtube.com/watch?v=thrx3SBEpL8>

## 5.3. Numbers

In this section, we will look at how computers represent numbers. To begin with, we'll revise how the base-10 number system that we use every day works, and then look at binary, which is base-2. After that, we'll look at some other characteristics of numbers that computers must deal with, such as negative numbers and numbers with decimal points.

### 5.3.1. Understanding the base 10 number system

The number system that humans normally use is in base 10 (also known as decimal). It's worth revising quickly, because binary numbers use the same ideas as decimal numbers, just with fewer digits!

In decimal, the value of each digit in a number depends on its **place** in the number. For example, in \$123, the 3 represents \$3, whereas the 1 represents \$100. Each place value in a number is worth 10 times more than the place value to its right, i.e. there are the "ones", the "tens", the "hundreds", the "thousands", the "ten thousands", the "hundred thousands", the "millions", and so on. Also, there are 10 different **digits** (0,1,2,3,4,5,6,7,8,9) that can be at each of those place values.

If you were only able to use one digit to represent a number, then the largest number would be 9. After that, you need a second digit, which goes to the left, giving you the next ten numbers (10, 11, 12... 19). It's because we have 10 digits that each one is worth 10 times as much as the one to its right.

You may have encountered different ways of expressing numbers using "expanded form". For example, if you want to write the number 90328 in expanded form you might have written it as:

$$90328 = 90000 + 300 + 20 + 8$$

A more sophisticated way of writing it is:

$$90328 = (9 \times 10000) + (0 \times 1000) + (3 \times 100) + (2 \times 10) + (8 \times 1)$$

If you've learnt about exponents, you could write it as

$$90328 = (9 \times 10^4) + (0 \times 10^3) + (3 \times 10^2) + (2 \times 10^1) + (8 \times 10^0)$$

Remember that any number to the power of 0 is 1. i.e. the  $8 \times 10^0$  is 8, because the  $10^0$  is 1.

The key ideas to notice from this are:

- Decimal has 10 **digits** – 0, 1, 2, 3, 4, 5, 6, 7, 8, 9.

- A **place** is the place in the number that a digit is, i.e. ones, tens, hundreds, thousands, and so on. For example, in the number 90328, 3 is in the "hundreds" place, 2 is in the "tens" place, and 9 is in the "ten thousands" place.
- Numbers are made with a sequence of digits.
- The right-most digit is the one that's worth the least (in the "ones" place).
- The left-most digit is the one that's worth the most.
- Because we have 10 digits, the digit at each place is worth 10 times as much as the one immediately to the right of it.

All this probably sounds really obvious, but it is worth thinking about consciously, because binary numbers have the same properties.

### 5.3.2. Representing whole numbers in Binary

As discussed earlier, computers can only store information using bits, which only have 2 possible states. This means that they cannot represent base 10 numbers using digits 0 to 9, the way we write down numbers in decimal. Instead, they must represent numbers using just 2 digits -- 0 and 1.

Binary works in a very similar way to Decimal, even though it might not initially seem that way. Because there are only 2 digits, this means that each digit is **2** times the value of the one immediately to the right.

#### CURIOSITY: The Denary number system

The base 10 (decimal) system is sometimes called denary, which is more consistent with the name binary for the base 2 system. The word "denary" also refers to the Roman denarius coin, which was worth ten asses (an "as" was a copper or bronze coin). The term "denary" seems to be used mainly in the UK; in the US, Australia and NZ the term "decimal" is more common.

The interactive below illustrates how this binary number system represents numbers. Have a play around with it to see what patterns you can see.

Use the interactive online at <http://www.csfieldguide.org.nz/en/interactives/base-calculator/index.html>

**To ensure you are understanding correctly how to use the interactive, verify that when you enter the binary number 101101 it shows that the decimal representation is 45, that when you enter 100000 it shows that the decimal representation is 32, and when you enter 001010 it shows the decimal representation is 10.**

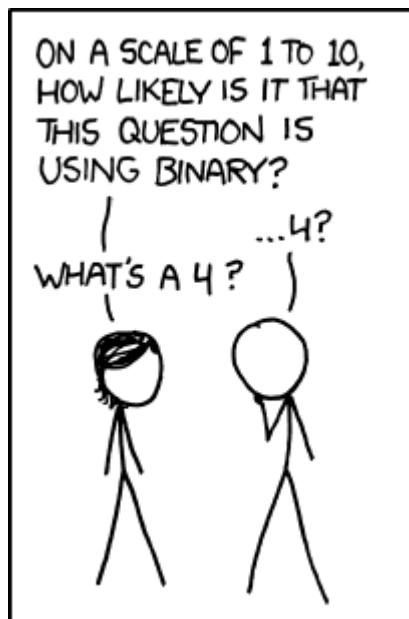
Find the representations of 4, 7, 12, and 57 using the interactive.

What is the largest number you can make with the interactive? What is the smallest? Is there any integer value in between the biggest and the smallest that you can't make? Are there any numbers with more than one representation? Why/ why not?

### Spoiler: Largest and smallest numbers

- 000000 in binary, 0 in decimal is the smallest number.
- 111111 in binary, 63 in decimal is the largest number
- All the integer values (0, 1, 2... 63) in the range can be represented (and there is a unique representation for each one). This is exactly the same as decimal!

You have probably noticed from the interactive that when set to 1, the leftmost bit (the "most significant bit") adds 32 to the total, the next adds 16, and then the rest add 8, 4, 2, and 1 respectively. When set to 0, a bit does not add anything to the total. So the idea is to make numbers by adding some or all of 32, 16, 8, 4, 2, and 1 together, and each of those numbers can only be included once.



[Image source](#)

Choose a number less than 61 (perhaps your house number, your age, a friend's age, or the day of the month you were born on), set all the binary digits to zero, and then start with the *left-most* digit (32), trying out if it should be zero or one. See if you can find a method for

converting the number without too much trial and error. Try different numbers until you find a quick way of doing this.

Figure out the binary representation for 23 **without** using the interactive? What about 4, 0, and 32? Check all your answers using the interactive to verify they are correct.

### Challenge: Counting in binary

Can you figure out a systematic approach to counting in binary? i.e. start with the number 0, then increment it to 1, then 2, then 3, and so on, all the way up to the highest number that can be made with the 7 bits. Try counting from 0 to 16, and see if you can detect a pattern. Hint: Think about how you add 1 to a number in base 10. e.g. how do you work out  $7 + 1$ ,  $38 + 1$ ,  $19 + 1$ ,  $99 + 1$ ,  $230899999 + 1$ , etc? Can you apply that same idea to binary?

Using your new knowledge of the binary number system, can you figure out a way to count to higher than 10 using your 10 fingers? What is the highest number you can represent using your 10 fingers? What if you included your 10 toes as well (so you have 20 fingers and toes to count with).

### Spoiler: Counting in binary

A binary number can be incremented by starting at the right and flipping all consecutive bits until a 1 comes up (which will be on the very first bit half of the time).

Counting on fingers in binary means that you can count to 31 on 5 fingers, and 1023 on 10 fingers. There are a number of videos on YouTube of people counting in binary on their fingers. One twist is to wear white gloves with the numbers 16, 8, 4, 2, 1 on the 5 fingers respectively, which makes it easy to work out the value of having certain fingers raised.

The interactive used exactly 6 bits. In practice, we can use as many or as few bits as we need, just like we do with decimal. For example, with 5 bits, the place values would be 16, 8, 4, 2 and 1, so the largest value is 11111 in binary, or 31 in decimal. Representing 14 with 5 bits would give 01110.

### Challenge: Representing numbers with bits

Write representations for the following. If it is not possible to do the representation, put "Impossible".

- Represent **101** with **7 bits**
- Represent **28** with **10 bits**
- Represent **7** with **3 bits**
- Represent **18** with **4 bits**
- Represent **28232** with **16 bits**

#### **Spoiler:** Answers for above challenge

The answers are (spaces are added to make the answers easier to read, but are not required)

- 101 with 7 bits is: **110 0101**
- 28 with 10 bits is: **00 0001 1100**
- 7 with 3 bits is: **111**
- 18 with 4 bits is: **Impossible to represent** (not enough bits)
- 28232 with 16 bits is: **0110 1110 0100 1000**

An important concept with binary numbers is the range of values that can be represented using a given number of bits. When we have 8 bits the binary numbers start to get useful --- they can represent values from 0 to 255, so it is enough to store someone's age, the day of the month, and so on.

#### **Jargon Buster:** What is a byte?

Groups of 8 bits are so useful that they have their own name: a **byte**. Computer memory and disk space are usually divided up into bytes, and bigger values are stored using more than one byte. For example, two bytes (16 bits) are enough to store numbers from 0 to 65,535. Four bytes (32 bits) can store numbers up to

4,294,967,295. You can check these numbers by working out the place values of the bits. Every bit that's added will double the range of the number.

In practice, computers store numbers with either 16, 32, or 64 bits. This is because these are full numbers of bytes (a byte is 8 bits), and makes it easier for computers to know where each number starts and stops.

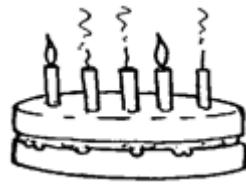
### Curiosity: Binary cakes -- preventing fires

Candles on birthday cakes use the base 1 numbering system, where each place is worth 1 more than the one to its right. For example, the number 3 is 111, and 10 is 1111111111. This can cause problems as you get older --- if you've ever seen a cake with 100 candles on it, you'll be aware that it's a serious fire hazard.



Luckily it's possible to use binary notation for birthday candles --- each candle is either lit or not lit. For example, if you are 18, the binary notation is 10010, and you need 5 candles (with only two of them lit).

There's a [video on using binary notation for counting up to 1023 on your hands, as well as using it for birthday cakes](#).



It's a lot smarter to use binary notation on candles for birthdays as you get older, as you don't need as many candles.

### 5.3.3. Shorthand for binary numbers - Hexadecimal

Most of the time binary numbers are stored electronically, and we don't need to worry about making sense of them. But sometimes it's useful to be able to write down and share numbers, such as the unique identifier assigned to each digital device (MAC address), or the colours specified in an HTML page.

Writing out long binary numbers is tedious --- for example, suppose you need to copy down the 16-bit number 0101001110010001. A widely used shortcut is to break the number up into 4-bit groups (in this case, 0101 0011 1001 0001), and then write down the digit that each group represents (giving 5391). There's just one small problem: each group of 4 bits can go up to 1111, which is 15, and the digits only go up to 9.

The solution is simple: we introduce symbols for the digits from 1010 (10) to 1111 (15), which are just the letters A to F. So, for example, the 16-bit binary number 1011 1000 1110 0001 can be written more concisely as B8E1. The "B" represents the binary 1011, which is the decimal number 11, and the E represents binary 1110, which is decimal 14.

Because we now have 16 digits, this representation is base 16, and known as hexadecimal (or hex for short). Converting between binary and hexadecimal is very simple, and that's why hexadecimal is a very common way of writing down large binary numbers.

Here's a full table of all the 4-bit numbers and their hexadecimal digit equivalent:

Binary	Hex
0000	0
0001	1
0010	2

<b>Binary</b>	<b>Hex</b>
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	A
1011	B
1100	C
1101	D
1110	E
1111	F

For example, the largest 8-bit binary number is 11111111. This can be written as FF in hexadecimal. Both of those representations mean 255 in our conventional decimal system (you can check that by converting the binary number to decimal).

Which notation you use will depend on the situation; binary numbers represent what is actually stored, but can be confusing to read and write; hexadecimal numbers are a good shorthand of the binary; and decimal numbers are used if you're trying to understand the

meaning of the number or doing normal math. All three are widely used in computer science.

It is important to remember though, that computers **only** represent numbers using binary. They **cannot** represent numbers directly in decimal or hexadecimal.

### 5.3.4. Computers representing numbers in practice

A common place that numbers are stored on computers is in spreadsheets or databases. These can be entered either through a spreadsheet program or database program, through a program you or somebody else wrote, or through additional hardware such as sensors, collecting data such as temperatures, air pressure, or ground shaking.

Some of the things that we might think of as numbers, such as the telephone number (03) 555-1234, aren't actually stored as numbers, as they contain important characters (like dashes and spaces) as well as the leading 0 which would be lost if it was stored as a number (the above number would come out as 35551234, which isn't quite right). These are stored as **text**, which is discussed in the next section.

On the other hand, things that don't look like a number (such as "30 January 2014") are often stored using a value that is converted to a format that is meaningful to the reader (try typing two dates into Excel, and then subtract one from the other --- the result is a useful number). In the underlying representation, a number is used. Program code is used to translate the underlying representation into a meaningful date on the user interface.

#### Curiosity: More on date representation

The difference between two dates in Excel is the number of days between them; the date itself (as in many systems) is stored as the amount of time elapsed since a fixed date (such as 1 January 1900). You can test this by typing a date like "1 January 1850" --- chances are that it won't be formatted as a normal date. Likewise, a date sufficiently in the future may behave strangely due to the limited number of bits available to store the date.

Numbers are used to store things as diverse as dates, student marks, prices, statistics, scientific readings, sizes and dimensions of graphics.

The following issues need to be considered when storing numbers on a computer

- What range of numbers should be able to be represented?
- How do we handle negative numbers?

- How do we handle decimal points or fractions?

### 5.3.5. How many bits are used in practice?

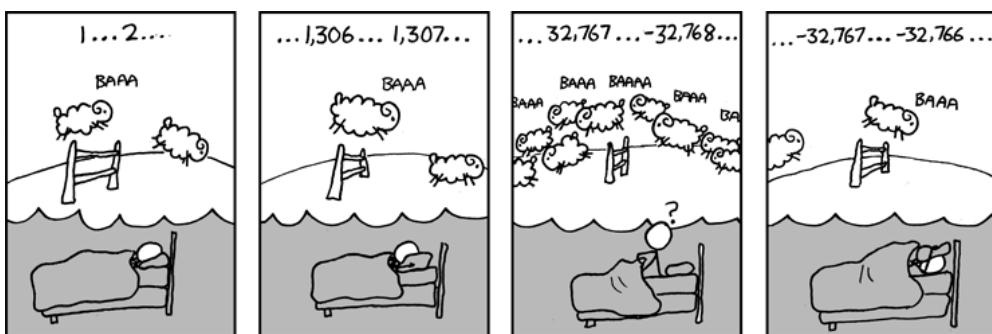
In practice, we need to allocate a fixed number of bits to a number, before we know how big the number is. This is often 32 bits or 64 bits, although can be set to 16 bits, or even 128 bits, if needed. This is because a computer has no way of knowing where a number starts and ends, otherwise.

Any system that stores numbers needs to make a compromise between the number of bits allocated to store the number, and the range of values that can be stored.

In some systems (like the Java and C programming languages and databases) it's possible to specify how accurately numbers should be stored; in others it is fixed in advance (such as in spreadsheets).

Some are able to work with arbitrarily large numbers by increasing the space used to store them as necessary (e.g. integers in the Python programming language). However, it is likely that these are still working with a multiple of 32 bits (e.g. 64 bits, 96 bits, 128 bits, 160 bits, etc). Once the number is too big to fit in 32 bits, the computer would reallocate it to have up to 64 bits.

In some programming languages there isn't a check for when a number gets too big (overflows). For example, if you have an 8-bit number using two's complement, then 01111111 is the largest number (127), and if you add one without checking, it will change to 10000000, which happens to be the number -128. This can cause serious problems if not checked for, and is behind a variant of the Y2K problem, called the [Year 2038 problem](#), involving a 32-bit number overflowing for dates on Tuesday, 19 January 2038.



[Image source](#)

On tiny computers, such as those embedded inside your car, washing machine, or a tiny sensor that is barely larger than a grain of sand, we might need to specify more precisely how big a number needs to be. While computers prefer to work with chunks of 32 bits, we

could write a program (as an example for an earthquake sensor) that knows the first 7 bits are the latitude, the next 7 bits are the longitude, the next 10 bits are the depth, and the last 8 bits are the amount of force.

Even on standard computers, it is important to think carefully about the number of bits you will need. For example, if you have a field in your database that could be either "0", "1", "2", or "3" (perhaps representing the four bases that can occur in a DNA sequence), and you used a 64 bit number for every one, that will add up as your database grows. If you have 10,000,000 items in your database, you will have wasted 62 bits for each one (only 2 bits is needed to represent the 4 numbers in the example), a total of 620,000,000 bits, which is around 74 MB. If you are doing this a lot in your database, that will really add up -- human DNA has about 3 billion base pairs in it, so it's incredibly wasteful to use more than 2 bits for each one.

And for applications such as Google Maps, which are storing an astronomical amount of data, wasting space is not an option at all!

### Challenge: How many bits will you need?

It is really useful to know roughly how many bits you will need to represent a certain value. Have a think about the following scenarios, and choose the best number of bits out of the options given. You want to ensure that the largest possible number will fit within the number of bits, but you also want to ensure that you are not wasting space.

1. Storing the day of the week

- a) 1 bit
- b) 4 bits
- c) 8 bits
- d) 32 bits

2. Storing the number of people in the world

- a) 16 bits
- b) 32 bits
- c) 64 bits
- d) 128 bits

3. Storing the number of roads in New Zealand

- a) 16 bits
- b) 32 bits

- c) 64 bits
- d) 128 bits

#### 4. Storing the number of stars in the universe

- a) 16 bits
- b) 32 bits
- c) 64 bits
- d) 128 bits

#### **Spoiler:** Answers for above challenge

1. b (actually, 3 bits is enough as it gives 8 values, but amounts that fit evenly into 8-bit bytes are easier to work with)
2. c (32 bits is slightly too small, so you will need 64 bits)
3. c (This is a challenging question, but one a database designer would have to think about. There's about 94,000 km of roads in NZ, so if the average length of a road was 1km, there would be too many roads for 16 bits. Either way, 32 bits would be a safe bet.)
4. d (Even 64 bits is not enough, but 128 bits is plenty! Remember that 128 bits isn't twice the range of 64 bits.)

## 5.3.6. Representing negative numbers in practice

The binary number representation we have looked at so far allows us to represent positive numbers only. In practice, we will want to be able to represent negative numbers as well, such as when the balance of an account goes to a negative amount, or the temperature falls below zero. In our normal representation of base 10 numbers, we represent negative numbers by putting a minus sign in front of the number. But in binary, is it this simple?

We will look at two possible approaches: Adding a simple sign bit, much like we do for decimal, and then a more useful system called Two's Complement.

### 5.3.6.1. Using a simple sign bit

On a computer we don't have minus signs for numbers (it doesn't work very well to use the text based one when representing a number because you can't do arithmetic on characters), but we can do it by allocating one extra bit, called a *sign* bit, to represent the minus sign. Just like with decimal numbers, we put the negative indicator on the left of the

number --- when the sign bit is set to "0", that means the number is positive and when the sign bit is set to "1", the number is negative (just as if there were a minus sign in front of it).

For example, if we wanted to represent the number **41** using 7 bits along with an additional bit that is the sign bit (to give a total of 8 bits), we would represent it by **00101001**. The first bit is a 0, meaning the number is positive, then the remaining 7 bits give **41**, meaning the number is **+41**. If we wanted to make **-59**, this would be **10111011**. The first bit is a 1, meaning the number is negative, and then the remaining 7 bits represent **59**, meaning the number is **-59**.

### Challenge: Representing negative numbers with sign bit

Using 8 bits as described above (one for the sign, and 7 for the actual number), what would be the binary representations for 1, -1, -8, 34, -37, -88, and 102?

### Spoiler: Representing negative numbers with sign bit

The spaces are not necessary, but are added to make reading the binary numbers easier

- 1 is 0000 0001
- -1 is 1000 0001
- -8 is 1000 1000
- 34 is 0010 0010
- -37 is 1010 0101
- -88 is 1101 1000
- 102 is 0110 0110

Going the other way is just as easy. If we have the binary number **10010111**, we know it is negative because the first digit is a 1. The number part is the next 7 bits **0010111**, which is **23**. This means the number is **-23**.

### Challenge: Converting binary with sign bit to decimal

What would the decimal values be for the following, assuming that the first bit is a sign bit?

- 00010011

- 10000110
- 10100011
- 01111111
- 11111111

**Spoiler:** Converting binary with sign bit to decimal

- 00010011 is 19
- 10000110 is -6
- 10100011 is -35
- 01111111 is 127
- 11111111 is -127

But what about **10000000**? That converts to **-0**. And **00000000** is **+0**. Since -0 and +0 are both just 0, it is very strange to have two different representations for the same number.

This is one of the reasons that we don't use a simple sign bit in practice. Instead, computers usually use a more sophisticated representation for negative binary numbers called *Two's Complement*.

### 5.3.6.2. Two's Complement

There's an alternative representation called *Two's Complement*, which avoids having two representations for 0, and more importantly, makes it easier to do arithmetic with negative numbers.

#### **Representing positive numbers with Two's Complement**

Representing positive numbers is the same as the method you have already learnt. Using **8 bits**, the leftmost bit is a zero and the other 7 bits are the usual binary representation of the number; for example, **1** would be **00000001**, and **65** would be **00110010**.

#### **Representing negative numbers with Two's Complement**

This is where things get more interesting. In order to convert a negative number to its two's complement representation, use the following process.

1. Convert the number to binary (don't use a sign bit, and pretend it is a positive number).
2. Invert all the digits (i.e. change 0's to 1's and 1's to 0's).

3. Add 1 to the result (Adding 1 is easy in binary; you could do it by converting to decimal first, but think carefully about what happens when a binary number is incremented by 1 by trying a few; there are more hints in the panel below).

For example, assume we want to convert **-118** to its Two's Complement representation. We would use the process as follows.

1. The binary number for **118** is **01110110**
2. **01110110** with the digits inverted is **10001001**
3. **10001001 + 1** is **10001010**

Therefore, the Two's Complement representation for **-118** is **10001010**.

#### **Challenge:** Adding one to a binary number

The rule for adding one to a binary number is pretty simple, so we'll let you figure it out for yourself. First, if a binary number ends with a 0 (e.g. 1101010), how would the number change if you replace the last 0 with a 1? Now, if it ends with 01, how much would it increase if you change the 01 to 10? What about ending with 011? 011111?

The method for adding is so simple that it's easy to build computer hardware to do it very quickly.

#### **Challenge:** Determining the Two's Complement

What would be the two's complement representation for the following numbers, **using 8 bits**? Follow the process given in this section, and remember that you do not need to do anything special for positive numbers.

1. 19
2. -19
3. 107
4. -107
5. -92

#### **Spoiler:** Determining the Two's Complement

1. 19 in binary is **0001 0011**, which is the two's complement for a positive number.

2. For -19, we take the binary of the positive, which is 0001 0011 (above), invert it to 1110 1100, and add 1, giving a representation of **1110 1101**.
3. 107 in binary is **0110 1011**, which is the two's complement for a positive number.
4. For -107, we take the binary of the positive, which is 0110 1011 (above), invert it to 1001 0100, and add 1, giving a representation of **1001 0101**.
5. For -92, we take the binary of the positive, which is 0101 1100, invert it to 1010 0011, and add 1, giving a representation of **1010 0100**. (If you have this incorrect, double check that you incremented by 1 correctly).

### **Converting a Two's Complement number back to decimal**

In order to reverse the process, we need to know whether the number we are looking at is positive or negative. For positive numbers, we can simply convert the binary number back to decimal. But for negative numbers, we first need to convert it back to a normal binary number.

So how do we know if the number is positive or negative? It turns out (for reasons you will understand later in this section) that Two's Complement numbers that are negative always start in a 1, and positive numbers always start in a 0. Have a look back at the previous examples to double check this.

So, if the number starts with a 1, use the following process to convert the number back to a negative decimal number.

1. Subtract 1 from the number
2. Invert all the digits
3. Convert the resulting binary number to decimal
4. Add a minus sign in front of it.

So if we needed to convert 11100010 back to decimal, we would do the following.

1. Subtract 1 from **11100010**, giving **11100001**.
2. Invert all the digits, giving **00011110**.
3. Convert **00011110** to a binary number, giving **30**.
4. Add a negative sign, giving **-30**.

**Challenge:** Reversing Two's Complement

Convert the following Two's Complement numbers to decimal.

1. 00001100
2. 10001100
3. 10111111

### Spoiler: Reversing Two's Complement

1. **12**
2. 10001100 -> (-1) 10001011 -> (inverted) 01110100 -> (to decimal) 116 -> (negative sign added) **-116**
3. 10111111 -> (-1) 10111110 -> (inverted) 01000001 -> (to decimal) 65 -> (negative sign added) **-65**

### **How many numbers can be represented using Two's Complement?**

While it might initially seem that there is no bit allocated as the sign bit, the left-most bit behaves like one. With 8 bits, you can still only make 256 possible patterns of 0's and 1's. If you attempted to use 8 bits to represent positive numbers up to 255, and negative numbers down to -255, you would quickly realise that some numbers were mapped onto the same pattern of bits. Obviously, this will make it impossible to know what number is actually being represented!

In practice, numbers within the following ranges can be represented. **Unsigned Range** is how many numbers you can represent if you only allow positive numbers (no sign is needed), and **Two's Complement Range** is how many numbers you can represent if you require both positive and negative numbers. You can work these out because the range of unsigned values (for 8 bits) will be from 00000000 to 11111111, while the unsigned range is from 10000000 (the lowest number) to 01111111 (the highest).

Number	Unsigned Range	Two's Complement Range
8 bit	0 to 255	-128 to 127
16 bit	0 to 65,535	-32,768 to 32,767
32 bit	0 to 4,294,967,295	-2,147,483,648 to 2,147,483,647

Number	Unsigned Range	Two's Complement Range
64 bit	0 to 18,446,744,073,709,551,615	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807

### 5.3.6.3. Adding negative binary numbers

Before adding negative binary numbers, we'll look at adding positive numbers. It's basically the same as the addition methods used on decimal numbers, except the rules are way simpler because there are only two different digits that you might add!

You've probably learnt about column addition. For example, the following column addition would be used to do **128 + 255**.

```

1   (carries)
128
+255
-----
 383

```

When you go to add  $5 + 8$ , the result is higher than 9, so you put the 3 in the one's column, and carry the 1 to the 10's column. Binary addition works in exactly the same way.

#### ***Adding positive binary numbers***

If you wanted to add two positive binary numbers, such as **00001111** and **11001110**, you would follow a similar process to the column addition. You only need to know  $0+0$ ,  $0+1$ ,  $1+0$ , and  $1+1$ , and  $1+1+1$ . The first three are just what you might expect. Adding  $1+1$  causes a carry digit, since in binary  $1+1 = 10$ , which translates to "0, carry 1" when doing column addition. The last one,  $1+1+1$  adds up to 11 in binary, which we can express as "1, carry 1". For our two example numbers, the addition works like this:

```

111   (carries)
11001110
+00001111
-----
 11011101

```

Remember that the digits can be only 1 or 0. So you will need to carry a 1 to the next column if the total you get for a column is (decimal) 2 or 3.

#### ***Adding negative numbers with a simple sign bit***

With negative numbers using sign bits like we did before, this does not work. If you wanted to add **+11 (01011)** and **-7 (10111)**, you would expect to get an answer of **+4 (00100)**.

```
11111 (carries)
01011
+10111
100010
```

Which is **-2**.

One way we could solve the problem is to use column subtraction instead. But this would require giving the computer a hardware circuit which could do this. Luckily this is unnecessary, because addition with negative numbers works automatically using Two's Complement!

### ***Adding negative numbers with Two's Complement***

For the above addition ( $+11 + -7$ ), we can start by converting the numbers to their 5-bit Two's Complement form. Because **01011 (+11)** is a positive number, it does not need to be changed. But for the negative number, **00111 (-7)** (sign bit from before removed as we don't use it for Two's Complement), we need to invert the digits and then add 1, giving **11001**.

Adding these two numbers works like this:

```
01011
11001
100100
```

Any extra bits to the left (beyond what we are using, in this case 5 bits) have been truncated. This leaves **00100**, which is **4**, like we were expecting.

We can also use this for subtraction. If we are subtracting a positive number from a positive number, we would need to convert the number we are subtracting to a negative number. Then we should add the two numbers. This is the same as for decimal numbers, for example  $5 - 2 = 3$  is the same as  $5 + (-2) = 3$ .

This property of Two's Complement is very useful. It means that positive numbers and negative numbers can be handled by the same computer circuit, and addition and subtraction can be treated as the same operation.

**Curiosity:** What's going on with Two's complement?

The idea of using a "complementary" number to change subtraction to addition can be seen by doing the same in decimal. The complement of a decimal digit is the digit that adds up to 10; for example, the complement of 4 is 6, and the complement of 8 is 2. (The word "complement" comes from the root "complete" - it completes it to a nice round number.)

Subtracting 2 from 6 is the same as adding the complement, and ignoring the extra 1 digit on the left. The complement of 2 is 8, so we add 8 to 6, giving (1)4.

For larger numbers (such as subtracting the two 3-digit numbers 255 - 128), the complement is the number that adds up to the next power of 10 i.e.  $1000-128 = 872$ . Check that adding 872 to 255 produces (almost) the same result as subtracting 128.

Working out complements in binary is way easier because there are only two digits to work with, but working them out in decimal may help you to understand what is going on.

#### 5.3.6.4. Using sign bits vs using Two's Complement

We have now looked at two different ways of representing negative numbers on a computer. In practice, a simple sign bit is rarely used, because of having two different representations of zero, and requiring a different computer circuit to handle negative and positive numbers, and to do addition and subtraction.

Two's Complement is widely used, because it only has one representation for zero, and it allows positive numbers and negative numbers to be treated in the same way, and addition and subtraction to be treated as one operation.

There are other systems such as "One's Complement" and "Excess-k", but Two's Complement is by far the most widely used in practice.

## 5.4. Text

There are several different ways in which computers use bits to store text. In this section, we will look at some common ones and then look at the pros and cons of each representation.

### 5.4.1. ASCII

We saw earlier that 64 unique patterns can be made using 6 dots in Braille. A dot corresponds to a bit, because both dots and bits have 2 different possible values.

Count how many different characters -- upper-case letters, lower-case letters, numbers, and symbols -- that you could type into a text editor using your keyboard. (Don't forget to count both of the symbols that share the number keys, and the symbols to the side that are for punctuation!)

### Jargon Buster: Characters

The collective name for upper-case letters, lower-case letters, numbers, and symbols is *characters* e.g. a, D, 1, h, 6, \*, ], and ~ are all characters. Importantly, a space is also a character.

If you counted correctly, you should find that there were more than 64 characters, and you might have found up to around 95. Because 6 bits can only represent 64 characters, we will need more than 6 bits; it turns out that we need at least 7 bits to represent all of these characters as this gives 128 possible patterns. This is exactly what the **ASCII** representation for text does.

### Challenge: Why 7 bits?

In the previous section, we explained what happens when the number of dots was increased by 1 (remember that a dot in Braille is effectively a bit). Can you explain how we knew that if 6 bits is enough to represent 64 characters, then 7 bits must be enough to represent 128 characters?

Each pattern in ASCII is usually stored in 8 bits, with one wasted bit, rather than 7 bits. However, the left-most bit in each 8-bit pattern is a 0, meaning there are still only 128 possible patterns. Where possible, we prefer to deal with full bytes (8 bits) on a computer, this is why ASCII has an extra wasted bit.

Here is a table that shows the patterns of bits that ASCII uses for each of the characters.

Binary	Char	Binary	Char	Binary	Char
0100000	Space	1000000	@	1100000	'
0100001	!	1000001	A	1100001	a

<b>Binary</b>	<b>Char</b>	<b>Binary</b>	<b>Char</b>	<b>Binary</b>	<b>Char</b>
0100010	"	1000010	B	1100010	b
0100011	#	1000011	C	1100011	c
0100100	\$	1000100	D	1100100	d
0100101	%	1000101	E	1100101	e
0100110	&	1000110	F	1100110	f
0100111	'	1000111	G	1100111	g
0101000	(	1001000	H	1101000	h
0101001	)	1001001	I	1101001	i
0101010	*	1001010	J	1101010	j
0101011	+	1001011	K	1101011	k
0101100	,	1001100	L	1101100	l
0101101	-	1001101	M	1101101	m
0101110	.	1001110	N	1101110	n
0101111	/	1001111	O	1101111	o
0110000	0	1010000	P	1110000	p
0110001	1	1010001	Q	1110001	q
0110010	2	1010010	R	1110010	r

Binary	Char	Binary	Char	Binary	Char
0110011	3	1010011	S	1110011	s
0110100	4	1010100	T	1110100	t
0110101	5	1010101	U	1110101	u
0110110	6	1010110	V	1110110	v
0110111	7	1010111	W	1110111	w
0111000	8	1011000	X	1111000	x
0111001	9	1011001	Y	1111001	y
0111010	:	1011010	Z	1111010	z
0111011	;	1011011	[	1111011	{
0111100	<	1011100	\	1111100	\
0111101	=	1011101	]	1111101	}
0111110	>	1011110	^	1111110	~
0111111	?	1011111	_	1111111	Delete

For example, the letter c (lower-case) in the table has the pattern “01100011” (the 0 at the front is just extra padding to make it up to 8 bits). The letter o has the pattern “01101111”. You could write a word out using this code, and if you give it to someone else, they should be able to decode it exactly.

Computers can represent pieces of text with sequences of these patterns, much like Braille does. For example, the word “computers” (all lower-case) would be 01100011 01101111 01101101 01110000 01110101 01110100 01100101 01110010 01110011. This is

because "c" is "01100011", "o" is "01101111", and so on. Have a look at the ASCII table above to check that we are right!

### **Curiosity:** What does ASCII stand for?

The name "ASCII" stands for "American Standard Code for Information Interchange", which was a particular way of assigning bit patterns to the characters on a keyboard. The ASCII system even includes "characters" for ringing a bell (useful for getting attention on old telegraph systems), deleting the previous character (kind of an early "undo"), and "end of transmission" (to let the receiver know that the message was finished). These days those characters are rarely used, but the codes for them still exist (they are the missing patterns in the table above). Nowadays ASCII has been supplanted by a code called "UTF-8", which happens to be the same as ASCII if the extra left-hand bit is a 0, but opens up a huge range of characters if the left-hand bit is a 1.

### **Challenge:** More practice at ASCII

Have a go at the following ASCII exercises

- How would you represent "science" in ASCII?
- How would you represent "Wellington" in ASCII? (note that it starts with an upper-case "W")
- How would you represent "358" in ASCII (it is three characters, even though it looks like a number)
- How would you represent "Hello, how are you?" (look for the comma, question mark, and space characters in ASCII table)

Be sure to have a go at all of them before checking the answer!

### **Spoiler:** Answers to questions above

These are the answers.

- "science" = 01110011 01100011 01101001 01100101 01101110 01100011  
01100101
- "Wellington" = 01010111 01100101 01101100 01101100 01101001 01101110  
01100111 01110100 01101111 01101110

- "358" = 00110011 00110101 00111000

Note that the text "358" is treated as 3 characters in ASCII, which may be confusing, as the text "358" is different to the number 358! You may have encountered this distinction in a spreadsheet e.g. if a cell starts with an inverted comma in Excel, it is treated as text rather than a number. One place this comes up is with phone numbers; if you type 027555555 into a spreadsheet as a number, it will come up as 27555555, but as text the 0 can be displayed. In fact, phone numbers aren't really just numbers because a leading zero can be important, as they can contain other characters -- for example, +64 3 555 1234 extn. 1234.

#### 5.4.1.1. ASCII usage in practice

ASCII was first used commercially in 1963, and despite the big changes in computers since then, it is still the basis of how English text is stored on computers. ASCII assigned a different pattern of bits to each of the characters, along with a few other "control" characters, such as delete or backspace.

English text can easily be represented using ASCII, but what about languages such as Chinese where there are thousands of different characters? Unsurprisingly, the 128 patterns aren't nearly enough to represent such languages. Because of this, ASCII is not so useful in practice, and is no longer used widely. In the next sections, we will look at Unicode and its representations. These solve the problem of being unable to represent non-English characters.

##### **Curiosity:** What came before ASCII?

There are several other codes that were popular before ASCII, including the [Baudot code](#) and [EBCDIC](#). A widely used variant of the Baudot code was the "Murray code", named after New Zealand born inventor [Donald Murray](#). One of Murray's significant improvements was to introduce the idea of "control characters", such as the carriage return (new line). The "control" key still exists on modern keyboards.

#### 5.4.2. Introduction to Unicode

In practice, we need to be able to represent more than just English characters. To solve this problem, we use a standard called **Unicode**. Unicode is a **character set** with around 120,000 different characters, in many different languages, current and historic. Each character has a unique number assigned to it, making it easy to identify.

Unicode itself is not a representation -- it is a character set. In order to represent Unicode characters as bits, a Unicode **encoding scheme** is used. The Unicode encoding scheme tells us how each number (which corresponds to a Unicode character) should be represented with a pattern of bits.

The following interactive will allow you to explore the Unicode character set. Enter a number in the box on the left to see what Unicode character corresponds to it, or enter a character on the right to see what its Unicode number is (you could paste one in from a foreign language web page to see what happens with non-English characters).

Use the interactive online at <http://www.csfieldguide.org.nz/en/interactives/unicode-chars/index.html>

The most widely used Unicode encoding schemes are called UTF-8, UTF-16, and UTF-32; you may have seen these names in email headers or describing a text file. Some of the Unicode encoding schemes are **fixed length**, and some are **variable length**. **Fixed length** means that each character is represented using the same number of bits. **Variable length** means that some characters are represented with fewer bits than others. It's better to be **variable length**, as this will ensure that the most commonly used characters are represented with fewer bits than the uncommonly used characters. Of course, what might be the most commonly used character in English is not necessarily the most commonly used character in Japanese. You may be wondering why we need so many encoding schemes for Unicode. It turns out that some are better for English language text, and some are better for Asian language text.

The remainder of the text representation section will look at some of these Unicode encoding schemes so that you understand how to use them, and why some of them are better than others in certain situations.

### 5.4.3. UTF-32

UTF-32 is a **fixed length** Unicode encoding scheme. The representation for each character is simply its number converted to a 32 bit binary number. Leading zeroes are used if there are not enough bits (just like how you can represent 254 as a 4 digit decimal number -- 0254). 32 bits is a nice round number on a computer, often referred to as a word (which is a bit confusing, since we can use UTF-32 characters to represent English words!)

For example, the character **H** in UTF-32 would be:

```
00000000 00000000 00000000 01001000
```

The character \$ in UTF-32 would be:

```
00000000 00000000 00000000 00100100
```

And the character 犬 in UTF-32 would be:

```
00000000 00000000 01110010 10101100
```

The following interactive will allow you to convert a Unicode character to its UTF-32 representation. The Unicode character's number is also displayed. The bits are simply the binary number form of the character number.

Use the interactive online at <http://www.csfieldguide.org.nz/en/interactives/unicode-binary/index.html?mode=utf32>

### Project: Represent your name with UTF-32

1. Represent each character in your name using UTF-32.
2. Check how many bits your representation required, and explain why it had this many (remember that each character should have required 32 bits)
3. Explain how you knew how to represent each character. Even if you used the interactive, you should still be able to explain it in terms of binary numbers.

ASCII actually took the same approach. Each ASCII character has a number between 0 and 255, and the representation for the character the number converted to an 8 bit binary number. ASCII is also a fixed length encoding scheme -- every character in ASCII is represented using 8 bits.

In practice, UTF-32 is rarely used -- you can see that it's pretty wasteful of space. UTF-8 and UTF-16 are both variable length encoding schemes, and very widely used. We will look at them next.

### Challenge: How big is 32 bits?

1. What is the largest number that can be represented with 32 bits? (In both decimal and binary).

2. The largest number in Unicode that has a character assigned to it is not actually the largest possible 32 bit number -- it is 00000000 00010000 11111111 11111111. What is this number in decimal?
  
3. Most numbers that can be made using 32 bits do not have a Unicode character attached to them -- there is a lot of wasted space. There are good reasons for this, but if you had a shorter number that could represent any character, what is the minimum number of bits you would need, given that there are currently around 120,000 Unicode characters?

#### **Spoiler:** Answers to above challenge

1. The largest number that can be represented using 32 bits is 4,294,967,295 (around 4.3 billion). You might have seen this number before -- it is the largest unsigned integer that a 32 bit computer can easily represent in programming languages such as C.
  
2. The decimal number for the largest character is 1,114,111.
  
3. You can represent all current characters with 17 bits. The largest number you can represent with 16 bits is 65,536, which is not enough. If we go up to 17 bits, that gives 131,072, which is larger than 120,000. Therefore, we need 17 bits.

### 5.4.4. UTF-8

UTF-8 is a **variable length** encoding scheme for Unicode. Characters with a lower Unicode number require fewer bits for their representation than those with a higher Unicode number. UTF-8 representations contain either 8, 16, 24, or 32 bits. Remembering that a **byte** is 8 bits, these are 1, 2, 3, and 4 bytes.

For example, the character **H** in UTF-8 would be:

```
01001000
```

The character **ø** in UTF-8 would be:

```
11000111 10111111
```

And the character 犬 in UTF-8 would be:

```
11100111 10001010 10101100
```

The following interactive will allow you to convert a Unicode character to its UTF-8 representation. The Unicode character's number is also displayed.

Use the interactive online at <http://www.csfieldguide.org.nz/en/interactives/unicode-binary/index.html?mode=utf8>

#### 5.4.4.1. How does UTF-8 work?

So how does UTF-8 actually work? Use the following process to do what the interactive is doing and convert characters to UTF-8 yourself.

1. Lookup the Unicode number of your character.
2. Convert the Unicode number to a binary number, using as **few** bits as necessary.  
Look back to the section on binary numbers if you cannot remember how to convert a number to binary.
3. Count how many bits are in the binary number, and choose the correct pattern to use, based on how many bits there were. Step 4 will explain how to use the pattern.

```
7 or fewer bits: 0xxxxxx
11 or fewer bits: 110xxxxx 10xxxxxx
16 or fewer bits: 1110xxxx 10xxxxxx 10xxxxxx
21 or fewer bits: 11110xxx 10xxxxxx 10xxxxxx 10xxxxxx
```

4. Replace the x's in the pattern with the bits of the binary number you converted in 2. If there are more x's than bits, replace extra left-most x's with 0's.

For example, if you wanted to find out the representation for 猫 (cat in Chinese), the steps you would take would be as follows.

1. Determine that the Unicode number for 猫 is **35987**.
2. Convert **35987** to binary -- giving **10001100 10010011**.
3. Count that there are **16** bits, and therefore the third pattern **1110xxxx 10xxxxxx 10xxxxxx** should be used.
4. Substitute the bits into the pattern to replace the x's -- **11101000 10110010 10010011**.

Therefore, the representation for 貓 is **11101000 10110010 10010011** using UTF-8.

## 5.4.5. UTF-16

Just like UTF-8, UTF-16 is a **variable length** encoding scheme for Unicode. Because it is far more complex than UTF-8, we won't explain how it works here.

However, the following interactive will allow you to represent text with UTF-16. Try putting some text that is in English and some text that is in Japanese into it. Compare the representations to what you get with UTF-8.

Use the interactive online at <http://www.csfieldguide.org.nz/en/interactives/unicode-binary/index.html?mode=utf16>

## 5.4.6. Comparison of text representations

We have looked at ASCII, UTF-32, UTF-8, and UTF-16.

The following table summarises what we have said so far about each representation.

Representation	Variable or Fixed	Bits per Character	Real world Usage
ASCII	Fixed Length	8 bits	No longer widely used
UTF-8	Variable Length	8, 16, 24, or 32 bits	Very widely used
UTF-16	Variable Length	16 or 32 bits	Widely used
UTF-32	Fixed Length	32 bits	Rarely used

In order to compare and evaluate them, we need to decide what it means for a representation to be "good". Two useful criteria are:

1. Can represent all characters, regardless of language.
2. Represents a piece of text using as few bits as possible.

We know that UTF-8, UTF-16, and UTF-32 can represent all characters, but ASCII can only represent English. Therefore, ASCII fails the first criterion. But for the second criteria, it isn't so simple.

The following interactive will allow you to find out the length of pieces of text using UTF-8, UTF-16, or UTF-32. Find some samples of English text and Asian text (forums or a translation site are a good place to look), and see how long your various samples are when encoded with each of the three representations. Copy paste or type text into the box.

Use the interactive online at <http://www.csfieldguide.org.nz/en/interactives/unicode-length/index.html>

As a general rule, UTF-8 is better for English text, and UTF-16 is better for Asian text. UTF-32 always requires 32 bits for each character, so is unpopular in practice.

### Curiosity: Emoji and Unicode

Those cute little characters that you might use in your Facebook statuses, tweets, texts, and so on, are called "emojis", and each one of them has their own Unicode value. Japanese mobile operators were the first to use emojis, but their recent popularity has resulted in many becoming part of the Unicode Standard and today there are well over 1000 different emojis included. A current list of these can be seen [here](#). What is interesting to notice is that a single emoji will look very different across different platforms, i.e. &#128518 ("smiling face with open mouth and tightly-closed eyes") in my tweet will look very different to what it does on your iPhone. This is because the Unicode Consortium only provides the character codes for each emoji and the end vendors determine what that emoji will look like, e.g. for Apple devices the "Apple Color Emoji" typeface is used (there are rules around this to make sure there is consistency across each system).

### 5.4.7. Project: Messages hidden in music

There are messages hidden in this video using a 5-bit representation. See if you can find them! Start by reading the explanation below to ensure you understand what we mean by a 5-bit representation.

Watch the video online at [https://www.youtube.com/watch?v=L-v4Awj\\_p7g](https://www.youtube.com/watch?v=L-v4Awj_p7g)

If you *only* wanted to represent the 26 letters of the alphabet, and weren't worried about upper-case or lower-case, you could get away with using just 5 bits, which allows for up to 32 different patterns.

You might have exchanged notes which used 1 for "a", 2 for "b", 3 for "c", all the way up to 26 for "z". We can convert those numbers into 5 digit binary numbers. In fact, you will also get the same 5 bits for each letter by looking at the last 5 bits for it in the ASCII table (and it doesn't matter whether you look at the upper case or the lower case letter).

Represent the word "water" with bits using this system. Check the below panel once you think you have it.

### Spoiler

```
w: 10111
a: 00001
t: 10111
e: 10100
r: 10010
```

**Now, have a go at decoding the music video!**

## 5.5. Images and Colours

### 5.5.1. How do computers display colours?

In school or art class you may have mixed different colours of paint or dye together in order to make new colours. In painting it's common to use red, yellow and blue as three "primary" colours that can be mixed to produce lots more colours. Mixing red and blue give purple, red and yellow give orange, and so on. By mixing red, yellow, and blue, you can make many new colours.

For printing, printers commonly use three slightly different primary colours: cyan, magenta, and yellow (CMY). All the colours on a printed document were made by mixing these primary colours.

Both these kinds of mixing are called "subtractive mixing", because they start with a white canvas or paper, and "subtract" colour from it. The interactive below allows you to experiment with CMY incase you are not familiar with it, or if you just like mixing colours.

Use the interactive online at <http://www.csfieldguide.org.nz/en/interactives/cmy-mixer/index.html>

Computer screens and related devices also rely on mixing three colours, except they need a different set of primary colours because they are *additive*, starting with a black screen and adding colour to it. For additive colour on computers, the colours red, green and blue (RGB) are used. Each pixel on a screen is typically made up of three tiny "lights"; one red, one green, and one blue. By increasing and decreasing the amount of light coming out of each of these three, all the different colours can be made. The following interactive allows you to play around with RGB.

Use the interactive online at <http://www.csfieldguide.org.nz/en/interactives/rgb-mixer/index.html>

See what colours you can make with the **RGB** interactive. Can you make black, white, shades of grey, yellow, orange, and purple?

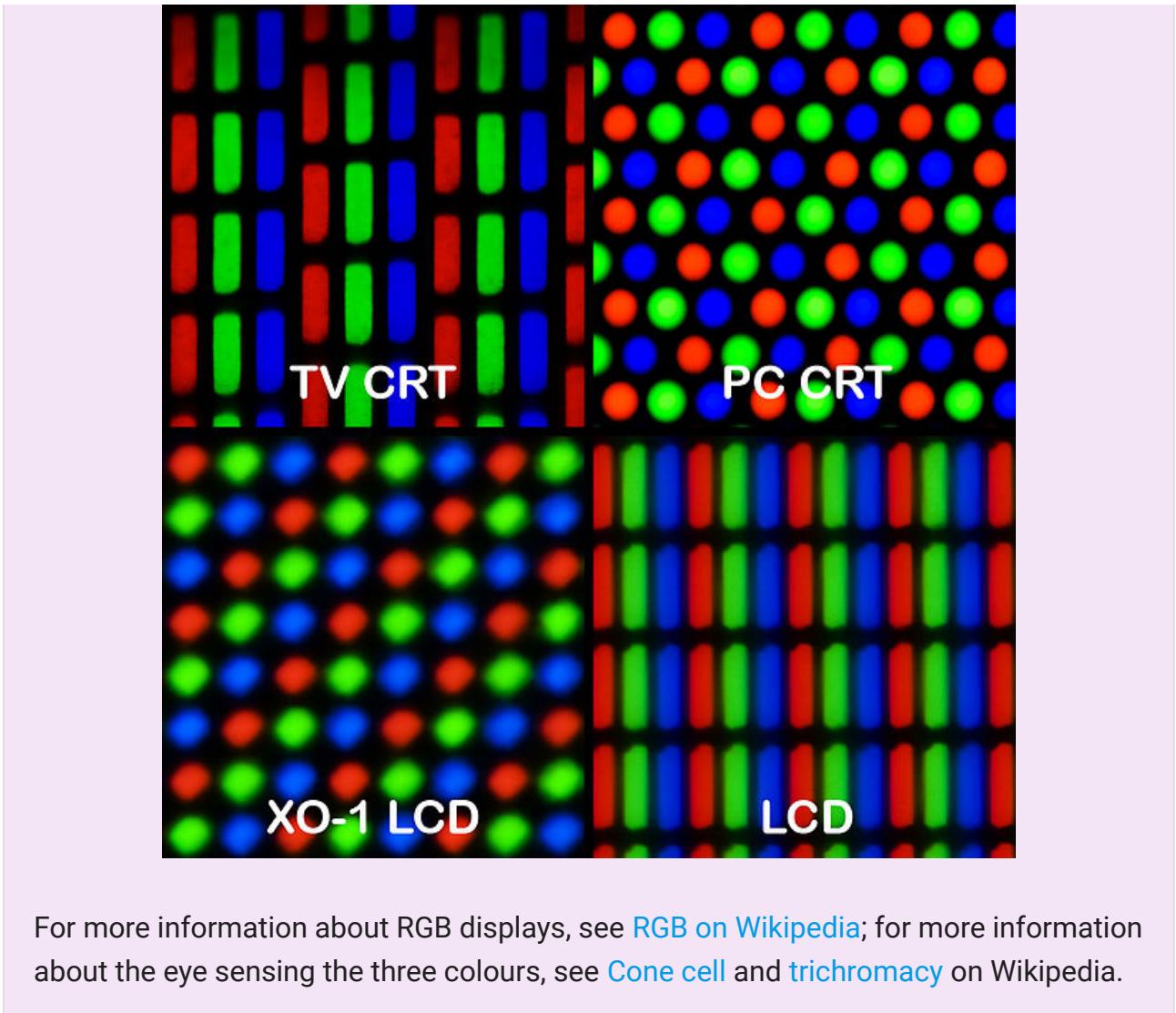
#### Spoiler: Hints for above

Having all the sliders at the extremes will produce black and white, and if they are all the same value but in between, it will be grey (i.e. between black and white).

Yellow is not what you might expect - it's made from red and green, with no blue.

#### Curiosity: Primary colours and the human eye

There's a very good reason that we mix three primary colours to specify the colour of a pixel. The human eye has millions of light sensors in it, and the ones that detect colour are called "cones". There are three different kinds of cones, which detect red, blue, and green light respectively. Colours are perceived by the amount of red, blue, and green light in them. Computer screen pixels take advantage of this by releasing the amounts of red, blue, and green light that will be perceived as the desired colour by your eyes. So when you see "purple", it's really the red and blue cones in your eyes being stimulated, and your brain converts that to a perceived colour. Scientists are still working out exactly how we perceive colour, but the representations used on computers seem to be good enough give the impression of looking at real images.



For more information about RGB displays, see [RGB on Wikipedia](#); for more information about the eye sensing the three colours, see [Cone cell](#) and [trichromacy](#) on Wikipedia.

## 5.5.2. Describing a colour with numbers

Because a colour is simply made up of amounts of the primary colours -- red, green and blue -- three numbers can be used to specify how much of each of these primary colours is needed to make the overall colour.

### Jargon Buster: Pixel

The word **pixel** is short for "picture element". On computer screens and printers an image is almost always displayed using a grid of pixels, each one set to the required colour. A pixel is typically a fraction of a millimeter across, and images can be made up of millions of pixels (one megapixel is a million pixels), so you can't usually see the individual pixels. Photographs commonly have several megapixels in them.

It's not unusual for computer screens to have millions of *pixels* on them, and the computer needs to represent a colour for each one of those pixels.

A commonly used scheme is to use numbers in the range 0 to 255. Those numbers tell the computer how fully to turn on each of the primary colour "lights" in an individual pixel. If red was set to 0, that means the red "light" is completely off. If the red "light" was set to 255, that would mean the "light" was fully on.

With 256 possible values for each of the three primary colours (don't forget to count 0!), that gives  $256 \times 256 \times 256 = 16,777,216$  possible colours -- more than the human eye can detect!

**Challenge:** What is special about 255?

Think back to the binary numbers section. What is special about the number 255, which is the maximum colour value?

We'll cover the answer later in this section if you are still not sure!

The following interactive allows you to zoom in on an image to see the pixels that are used to represent it. Each pixel is a solid colour square, and the computer needs to store the colour for each pixel. If you zoom in far enough, the interactive will show you the red-green-blue values for each pixel. You can pick a pixel and put the values on the slider above - it should come out as the same colour as the pixel.

Use the interactive online at <http://www.csfieldguide.org.nz/en/interactives/pixel-viewer/index.html>

**Curiosity:** Alternative material on bits and colour

Another exercise to see the relationship between bit patterns and colour images is [provided here](#).

### 5.5.3. Representing a colour with bits

The next thing we need to look at is how bits are used to represent each colour in a high quality image. Firstly, how many bits do we need? Secondly, how should we decide the values of each of those bits? This section will work through those problems.

### 5.5.3.1. How many bits will we need for each colour in the image?

With 256 different possible values for the amount of each primary colour, this means 8 bits would be needed to represent the number.

$$2^8 = 2 \times 2 = 256$$

The smallest number that can be represented using 8 bits is 00000000 -- which is 0. And the largest number that can be represented using 8 bits is 11111111 -- which is 255.

Because there are three primary colours, each of which will need 8 bits to represent each of its 256 different possible values, we need **24 bits in total** to represent a colour.

$$3 \times 8 = 24$$

So, how many colours are there in total with 24 bits? We know that there is 256 possible values each colour can take, so the easiest way of calculating it is:

$$256 \times 256 \times 256 = 16,777,216$$

This is the same as  $2^{24}$ .

Because 24 bits are required, this representation is called **24 bit colour**. 24 bit colour is sometimes referred to in settings as "True Color" (because it is more accurate than the human eye can see). On Apple systems, it is called "Millions of colours".

### 5.5.3.2. How do we use bits to represent the colour?

A logical way is to use 3 binary numbers that represent the amount of each of red, green, and blue in the pixel. In order to do this, convert the amount of each primary colour needed to an 8 bit binary number, and then put the 3 binary numbers side by side to give 24 bits.

Because consistency is important in order for a computer to make sense of the bit pattern, we normally adopt the convention that the binary number for red should be put first, followed by green, and then finally blue. The only reason we put red first is because that is the convention that most systems assume is being used. If everybody had agreed that green should be first, then it would have been green first.

For example, suppose you have the colour that has red = 145, green = 50, and blue = 123 that you would like to represent with bits. If you put these values into the interactive, you will get the colour below.



Start by converting each of the three numbers into binary, using 8 bits for each.

You should get:

- red = 10010001,
- green = 00110010,
- blue = 01111011.

Putting these values together gives 100100010011001001111011, which is the bit representation for the colour above.

There are **no spaces** between the three numbers, as this is a pattern of bits rather than actually being three binary numbers, and computers don't have any such concept of a space between bit patterns anyway --- everything must be a 0 or a 1. You could write it with spaces to make it easier to read, and to represent the idea that they are likely to be stored in 3 8-bit bytes, but inside the computer memory there is just a sequence of high and low voltages, so even writing 0 and 1 is an arbitrary notation.

Also, all leading and trailing 0's on each part are kept --- without them, it would be representing a shorter number. If there were 256 different possible values for each primary colour, then the final representation **must** be 24 bits long.

### Curiosity: Monochromatic images

"Black and white" images usually have more than two colours in them; typically 256 shades of grey, represented with 8 bits.

Remember that shades of grey can be made by having an equal amount of each of the 3 primary colours, for example red = 105, green = 105, and blue = 105.

So for a monochromatic image, we can simply use a representation which is a single binary number between 0 and 255, which tells us the value that all 3 primary colours should be set to.

The computer won't ever convert the number into decimal, as it works with the binary directly --- most of the process that takes the bits and makes the right pixels appear is typically done by a graphics card or a printer. We just started with decimal, because it is easier for humans to understand. The main point about knowing this representation is to understand the trade-off that is being made between the accuracy of colour (which should ideally be beyond human perception) and the amount of storage (bits) needed (which should be as little as possible).

### **Curiosity:** Hexadecimal colour codes

If you haven't already, read the section on [Hexadecimal](#), otherwise this section might not make sense!

When writing HTML code, you often need to specify colours for text, backgrounds, and so on. One way of doing this is to specify the colour name, for example "red", "blue", "purple", or "gold". For some purposes, this is okay.

However, the use of names limits the number of colours you can represent and the shade might not be exactly the one you wanted. A better way is to specify the 24 bit colour directly. Because 24 binary digits are hard to read, colours in HTML use **hexadecimal codes** as a quick way to write the 24 bits, for example #00FF9E. The hash sign means that it should be interpreted as a hexadecimal representation, and since each hexadecimal digit corresponds to 4 bits, the 6 digits represent 24 bits of colour information.

This "hex triplet" format is used in HTML pages to specify colours for things like the background of the page, the text, and the colour of links. It is also used in CSS, SVG, and other applications.

In the 24 bit colour example earlier, the 24 bit pattern was  
100100010011001001111011.

This can be broken up into groups of 4 bits: 1001 0001 0011 0010 0111 1011.

And now, each of these groups of 4 bits will need to be represented with a **hexadecimal** digit.

- 1001 -> 5
- 0001 -> 1
- 0011 -> 3
- 0010 -> 2
- 0111 -> 7
- 1011 -> B

Which gives #51327B.

Understanding how these hexadecimal colour codes are derived also allows you to change them slightly without having to refer back the colour table, when the colour isn't exactly the one you want. Remember that in the 24 bit color code, the first 8 bits specify the amount of red (so this is the first 2 digits of the hexadecimal code), the next 8 bits specify the amount of green (the next 2 digits of the hexadecimal code), and the last 8 bits specify the amount of blue (the last 2 digits of the hexadecimal code). To increase the amount of any one of these colours, you can change the appropriate hexadecimal letters.

For example, #000000 has zero for red, green and blue, so setting a higher value to the middle two digits (such as #004300) will add some green to the colour.

You can use this HTML page to experiment with hexadecimal colours. Just enter a colour in the space below:

Use the interactive online at <http://www.csfieldguide.org.nz/en/interactives/hex-background-colour/index.html>

## 5.5.4. Representing colours with fewer bits

What if we were to use fewer than 24 bits to represent each colour? How much space will be saved, compared to the impact on the image?

### 5.5.4.1. The range of colours with fewer bits

The following interactive gets you to try and match a specific colour using 24 bits, and then 8 bits.

It should be possible to get a perfect match using 24 bit colour. But what about 8 bits?

Use the interactive online at <http://www.csfieldguide.org.nz/en/interactives/colour-matcher/index.html>

The above system used 3 bits to specify the amount of red (8 possible values), 3 bits to specify the amount of green (again 8 possible values), and 2 bits to specify the amount of blue (4 possible values). This gives a total of 8 bits (hence the name), which can be used to make 256 different bit patterns, and thus can represent 256 different colours.

You may be wondering why blue is represented with fewer bits than red and green. This is because the human eye is the least sensitive to blue, and therefore it is the least important colour in the representation. The representation uses 8 bits rather than 9 bits because it's easiest for computers to work with full bytes.

Using this scheme to represent all the pixels of an image takes one third of the number of bits required for 24-bit colour, but it is not as good at showing smooth changes of colours or subtle shades, because there are only 256 possible colors for each pixel. This is one of the big tradeoffs in data representation: do you allocate less space (fewer bits), or do you want higher quality?

#### Jargon Buster: Colour depth

The number of bits used to represent the colours of pixels in a particular image is sometimes referred to as its "colour depth" or "bit depth". For example, an image or display with a colour depth of 8-bits has a choice of 256 colours for each pixel. There is [more information about this in Wikipedia](#). Drastically reducing the bit depth of an image can make it look very strange; sometimes this is used as a special effect called "posterisation" (ie. making it look like a poster that has been printed with just a few colours).

#### Curiosity: Colour depth and compression

There's a subtle boundary between low quality data representations (such as 8-bit colour) and compression methods. In principle, reducing an image to 8-bit colour is a

way to compress it, but it's a very poor approach, and a proper compression method like JPEG will do a much better job.

### 5.5.4.2. What impact does fewer bits have on the overall image?

The following interactive shows what happens to images when you use a smaller range of colours (including right down to zero bits!) You can choose an image using the menu or upload your own one. In which cases is the change in quality most noticeable? In which is it not? In which would you actually care about the colours in the image? In which situations is colour actually not necessary (i.e. when are we fine with two colours)?

Use the interactive online at <http://www.csfieldguide.org.nz/en/interactives/image-bit-comparer/index.html>

#### **Additional Information:** Software for exploring colour depth

Although we provide a simple interactive for reducing the number of bits in an image, you could also use software like Gimp or Photoshop to save files with different colour depths.

You probably noticed that 8-bit colour looks particularly bad for faces, where we are used to seeing subtle skin tones. Even the 16-bit colour is noticeably worse for faces.

In other cases, the 16-bit images are almost as good as 24-bit images unless you look really carefully. They also use two-thirds (16/24) of the space that they would with 24-bit colour. For images that will need to be downloaded on 3G devices where internet is expensive, this is worth thinking about carefully.

Have an experiment with the following interactive, to see what impact different numbers of bits for each colour has. Do you think 8 bit colour was right in having 2 bits for blue, or should it have been green or red that got only 2 bits?

Use the interactive online at <http://www.csfieldguide.org.nz/en/interactives/image-bit-comparer/index.html?change-bits=true>

#### **Curiosity:** Do we ever need more than 24 bit colour?

One other interesting thing to think about is whether or not we'd want more than 24 bit colour. It turns out that the human eye can only differentiate around 10 million

colours, so the ~16 million provided by 24 bit colour is already beyond what our eyes can distinguish. However, if the image were to be processed by some software that enhances the contrast, it may turn out that 24-bit colour isn't sufficient. Choosing the representation isn't simple!

### 5.5.4.3. How much space will low quality images save?

An image represented using 24 bit colour would have 24 bits per pixel. In  $600 \times 800$  pixel image (which is a reasonable size for a photo), this would contain  $600 \times 800 = 480,000$  pixels, and thus would use  $480,000 \times 24\text{bits} = 11,520,000$  bits. This works out to around 1.44 megabytes. If we use 8-bit colour instead, it will use a third of the memory, so it would save nearly a megabyte of storage. Or if the image is downloaded then a megabyte of bandwidth will be saved.

8 bit colour is not used much anymore, although it can still be helpful in situations such as accessing a computer desktop remotely on a slow internet connection, as the image of the desktop can instead be sent using 8 bit colour instead of 24 bit colour. Even though this may cause the desktop to appear a bit strange, it doesn't stop you from getting whatever it was you needed to get done, done. Seeing your desktop in 24 bit colour would not be very helpful if you couldn't get your work done!

In some countries, mobile internet data is very expensive. Every megabyte that is saved will be a cost saving. There are also some situations where colour doesn't matter at all, for example diagrams, and black and white printed images.

### 5.5.4.4. What about in practice?

If space really is an issue, then this crude method of reducing the range of colours isn't usually used; instead, compression methods such as JPEG, GIF and PNG are used.

These make much more clever compromises to reduce the space that an image takes, without making it look so bad, including choosing a better palette of colours to use rather than just using the simple representation discussed above. However, compression methods require a lot more processing, and images need to be decoded to the representations discussed in this chapter before they can be displayed.

The ideas in this present chapter more commonly come up when designing systems (such as graphics interfaces) and working with high-quality images (such as RAW photographs), and typically the goal is to choose the best representation possible without wasting too much space.

Have a look at the Compression Chapter to find out more!

## 5.6. Program Instructions

### Caution

Before reading this section, you should have an understanding of low level languages (see the section on [Machine Code in the Programming Languages chapter](#)).

In a similar fashion to representing text or numbers using binary, we can represent an entire actual program using binary. Since a program is just a sequence of instructions, we need to decide how many bits will be used to represent a single instruction and then how we are going to interpret those bits. Machine code instructions typically have a combination of two pieces: operation and operand.

```
li $t0, 10 #Load the value 10 into register $t0
li $t1, 20 #Load the value 20 into register $t1
#Add the values in $t0 and $t1, put the result in register $a0
add $a0, $t0, $t1
```

In the above machine code program li and add are considered to be operations to "load an integer" and "add two integers" respectively. \$t0, \$t1, and \$a0 are register operands and represent a place to store values inside of the machine. 10 and 20 are literal operands and allow instructions to represent the exact integer values 10 and 20. If we were using a 32-bit operating system we might encode the above instructions with each instruction broken into 4 8-bit pieces as follows:

Operation	Op1	Op2	Op3
00001000	00000000	00000000	00001010
00001000	00000001	00000000	00010100
00001010	10000000	00000000	00000001

Our operation will always be determined by the bits in the first 8-bits of the 32-bit instruction. In this example machine code, 00001000 means li and 00001010 means add. For the li operation, the bits in Op1 are interpreted to be a storage place, allowing

00000000 to represent \$t0. Similarly the bits in Op1 for the add instruction represent \$a0. Can you figure out what the bits in Op3 for each instruction represent?

Using bits to represent both the program instructions and data forms such as text, numbers, and images allows entire computer programs to be represented in the same binary format. This allows programs to be stored on disks, in memory, and transferred over the internet as easily as data.

## 5.7. The whole story!

The kind of image representations covered here are the basic ones used in most digital systems, and the main point of this chapter is to understand how digital representations work, and the compromises needed between the number of bits, storage used, and quality.

The colour representation discussed is what is often referred to as "raw" or "bitmap" (bmp) representation. For large images, real systems use compression methods such as JPEG, GIF or PNG to reduce the space needed to store an image, but at the point where an image is being captured or displayed it is inevitably represented using the raw bits as described in this chapter, and the basic choices for capturing and displaying images will affect the quality and cost of a device. Compression is regarded as a form of encoding, and is covered in a later chapter.

The representation of numbers is a whole area of study in itself. The choice of representation affects how quickly arithmetic can be done on the numbers, how accurate the results are, and how much memory or disk space is used up storing the data. Even integers have issues like the order in which a large number is broken up across multiple bytes. Floating point numbers generally follow common standards (the IEEE 754 standard is the most common one) to make it easy to design compatible hardware to process them. Spreadsheets usually store numbers using a floating point format, which limits the precision of calculations (typically about 64 bits are used for each number). There are many experiments that can be done (such as calculating  $1/3$ , or adding a very large number to a very small one) that demonstrate the limitations of floating point representations.

## 5.8. Further reading

This puzzle can be solved using the pattern in binary numbers: <http://www.cs4fn.org/binary/lock/>

[This site](#) has more complex activities with binary numbers, including fractions, multiplication and division.

## 5.8.1. Useful Links

- [Basics of binary numbers](#)
- [Representing bits using sound](#)
- [Hex game](#)
- [Thriving in our digital world](#) has good illustrations of data representation
- [How a hard disk works](#)

# 6. Coding - Introduction

## 6.1. What's the big picture?

The word "code" has lots of meanings in computer science. It's often used to talk about programming, and a program can be referred to as "source code". Even binary representation of information is sometimes referred to as a code. However, in this chapter (and the next three chapters), the sense of coding that will be used is about clever representations of information that address a practical issue, such as encrypting the data to keep it secret.

In the previous chapter we looked at using binary representations to store all kinds of data --- numbers, text, images and more. But often simple binary representations don't work so well. Sometimes they take up too much space, sometimes small errors in the data can cause big problems, and sometimes we worry that someone else could easily read our messages. Most of the time all three of these things are a problem! The codes that we will look at here overcome all of these problems, and are widely used for storing and transmitting important information.

The three main reasons that we use more complex representations of binary data are:

- **Compression:** this reduces the amount of space the data needs (for example, coding an audio file using MP3 compression can reduce the size of an audio file to well under 10% of its original size).
- **Encryption:** this changes the representation of data so that you need to have a "key" to unlock the message (for example, whenever your browser uses "https" instead of "http" to communicate with a website, encryption is being used to make sure that anyone eavesdropping on the connection can't make any sense of the information).
- **Error Control:** this adds extra information to your data so that if there are minor failures in the storage device or transmission, it is possible to detect that the data has been corrupted, and even reconstruct the information (for example, bar codes on products have an extra digit added to them so that if the bar code is scanned incorrectly in a checkout, it makes a warning sound instead of charging you for the wrong product).

Often all three of these are applied to the same data; for example, if you take a photo on a smartphone it is usually compressed using JPEG, stored in the phone's memory with error correction, and uploaded to the web through a wireless connection using an encryption protocol to prevent other people nearby getting a copy of the photo.

Without these forms of coding, digital devices would be very slow, have limited capacity, be unreliable, and be unable to keep your information private.

## 6.2. The story of coding



The idea of encoding data to make the representation more compact, robust or secure is centuries old, but the solid theory needed to support codes in the information age was developed in the 1940s -- not surprisingly considering that technology played such an important role in World War II, where efficiency, reliability and secrecy were all very important. One of the most celebrated researchers in this area was Claude Shannon, who

developed the field of "information theory", which is all about how data can be represented effectively (Shannon was also a juggler, unicyclist, and inventor of fanciful machines).

### Curiosity: Entropy

A key concept in Shannon's work is a measure of information called "entropy", which established mathematical limits like how small files could be compressed, and how many extra bits must be added to a message to achieve a given level of reliability. While the idea of entropy is beyond what we need to cover here, there are some fun games that provide a taste of how you could measure information content by guessing what letter comes next. For example, think of a sentence, and see if a friend can guess the first letter. If it's an English sentence, chances are they'll guess that the first letter is "T", "A" or "I", rather than "X" or "Z". If, after a while, you had guessed that the beginning letters in a sentence are "There is no revers", you'd probably guess that the next letter is an "e". Entropy is about how easy it is to guess the next letter; this is useful in compression (we give short codes to letters that are likely to occur next), encryption (a good code makes it hard to guess the letters), and error control (if an error occurs, it needs to be easy to "guess" what the original text was).

You can explore the idea of entropy further using an [Unplugged activity called Twenty Guesses](#), and an [online game for guessing sentences](#).

## 6.3. Further reading

James Gleick's book [The Information: A History, a Theory, a Flood](#) provides an interesting view of the history of several areas relating to coding.

### 6.3.1. Useful Links

- A good collection of resources related to all three kinds of coding is available in the [Bletchley Park Codes Resources](#)
- [Entropy and information theory](#)
- [History of information theory and its relationship to entropy in thermodynamics](#)
- [Timeline of information theory](#)
- [Shannon's seminal work in information theory](#)

# 7. Coding - Compression

## 7.1. What's the big picture?

Data compression reduces the amount of space needed to store files. If you can halve the size of a file, you can store twice as many files for the same cost, or you can download the files twice as fast (and at half the cost if you're paying for the download). Even though disks are getting bigger and high bandwidth is becoming common, it's nice to get even more value by working with smaller, compressed files. For large data warehouses, like those kept by Google and Facebook, halving the amount of space taken can represent a massive reduction in the space and computing required, and consequently big savings in power consumption and cooling, and a huge reduction in the impact on the environment.

Common forms of compression that are currently in use include JPEG (used for photos), MP3 (used for audio), MPEG (used for videos including DVDs), and ZIP (for many kinds of data). For example, the JPEG method reduces photos to a tenth or smaller of their original size, which means that a camera can store 10 times as many photos, and images on the web can be downloaded 10 times faster.

So what's the catch? Well, there can be an issue with the quality of the data – for example, a highly compressed JPEG image doesn't look as sharp as an image that hasn't been compressed. Also, it takes processing time to compress and decompress the data. In most cases, the tradeoff is worth it, but not always.

Use the interactive online at <http://www.csfieldguide.org.nz/en/interactives/compression-comparer/index.html>

In this chapter we'll look at how compression might be done, what the benefits are, and the costs associated with using compressed data that need to be considered when deciding whether or not to compress data.

We'll start with a simple example – Run Length Encoding – which gives some insight into the benefits and the issues around compression.

## 7.2. Run Length Encoding

Watch the video online at <https://www.youtube.com/embed/uaV2RuAJTjQ?rel=0>

Run length encoding (RLE) is a technique that isn't so widely used these days, but it's a great way to get a feel for some of the issues around using compression.

Imagine we have the following simple black and white image.



One very simple way a computer can store this image in binary is by using a format where '0' means white and '1' means black (this is a "bit map", because we've mapped the pixels onto the values of bits). Using this method, the above image would be represented in the following way:

```
011000010000110
100000111000001
000001111100000
000011111110000
000111111111000
001111101111100
011111000111110
111110000011111
011111000111110
001111101111100
000111111111000
000011111111000
000001111100000
100000111000001
011000010000110
```

### Curiosity: The PBM file format

There is an image format that uses the simple one-symbol-per-pixel representation we have just described. The format is called "portable bitmap format" (PBM). PBM files are saved with the file extension ".pbm", and contain a simple header, followed by the image data. The data in the file can be viewed by opening it in a text editor, much like opening a .txt file, and the image itself can be viewed by opening it in a drawing or

image viewing program that supports PBM files (the format isn't very well supported, but a number of image viewing and editing programs can display them). A pbm file for the diamond image used earlier would be as follows:

```
P1
15 15
011000010000110
100000111000001
000001111100000
000011111100000
000111111110000
001111101111100
011111000111110
111110000011111
011111000111110
001111101111100
000111111110000
000011111100000
100000111000001
011000010000110
```

The first two lines are the header. The first line specifies the format of the file (P1 means that the file contains ASCII zeroes and ones). The second line specifies the width and then the height of the image in pixels. This allows the computer to know the size and dimensions of the image, even if the newline characters separating the rows in the file were missing. The rest of the data is the image, just like above. If you wanted to, you could copy and paste this representation (including the header) into a text file, and save it with the file extension .pbm. If you have a program on your computer able to open PBM files, you could then view the image with it. You could even write a program to output these files, and then display them as images.

Because the digits are represented using ASCII in this format, it isn't very efficient, but it is useful if you want to read what's inside the file. There are variations of this format that pack the pixels into bits instead of characters, and variations that can be used for grey scale and colour images. More [information about this format is available on Wikipedia](#).

The key question in compression is whether or not we can represent the same image using fewer bits, but still be able to reconstruct the original image.

It turns out we can. There are many ways of going about it, but in this section we are focussing on a method called *run length encoding*.

Imagine that you had to read the bits above out to someone who was copying them down... after a while you might say things like "five zeroes" instead of "zero zero zero zero zero". Is the basic idea behind run length encoding (RLE), which is used to save space for storing digital images. In run length encoding, we replace each row with numbers that say how many consecutive pixels are the same colour, *always starting with the number of white pixels*. For example, the first row in the image above contains one white, two black, four white, one black, four white, two black, and one white pixel.

```
011000010000110
```

This could be represented as follows.

```
1, 2, 4, 1, 4, 2, 1
```

For the second row, because we need to say what the number of white pixels is before we say the number of black, we need to explicitly say there are zero at the start of the row.

```
100000111000001
```

```
0, 1, 5, 3, 5, 1
```

You might ask why we need to say the number of white pixels first, which in this case was zero. The reason is that if we didn't have a clear rule about which to start with, the computer would have no way of knowing which colour was which when it displays the image represented in this form!

The third row contains five whites, five blacks, five whites.

```
00000111100000
```

This is coded as:

```
5, 5, 5
```

That means we get the following representation for the first three rows.

```
1, 2, 4, 1, 4, 2, 1
0, 1, 5, 3, 5, 1
5, 5, 5
```

You can work out what the other rows would be following this same system.

#### **Spoiler:** Representation for the remaining rows

The remaining rows are

```
4, 7, 4
3, 9, 3
2, 5, 1, 5, 2
1, 5, 3, 5, 1
0, 5, 5, 5
1, 5, 3, 5, 1
2, 5, 1, 5, 2
3, 9, 3
4, 7, 4
5, 5, 5
0, 1, 5, 3, 5, 1
1, 2, 4, 1, 4, 2, 1
```

#### **Curiosity:** Run Length Encoding in the CS Unplugged show

In this video from a Computer Science Unplugged show, a Run length encoded image is decoded using very large pixels (the printer is a spray can!).

Watch the video online at <https://www.youtube.com/watch?v=VsjpPs146d8>

### 7.2.1. Converting Run Length Encoding back to the original representation

Just to ensure that we can reverse the compression process, have a go at finding the original representation (zeroes and ones) of this (compressed) image.

```
4, 11, 3
4, 9, 2, 1, 2
```

```
4, 9, 2, 1, 2  
4, 11, 3  
4, 9, 5  
4, 9, 5  
5, 7, 6  
0, 17, 1  
1, 15, 2
```

What is the image of? How many pixels were there in the original image? How many numbers were used to represent those pixels?

**Spoiler:** Answer for the above image

This image is from the [CS Unplugged image representation activity](#), and the solution is available in the activity (it is a cup and saucer).

The following interactive allows you to experiment further with Run Length Encoding.

Use the interactive online at <http://www.csfieldguide.org.nz/en/interactives/run-length-encoding/index.html>

## 7.2.2. Analysing Run Length Encoding

How much space have we saved using this alternate representation, and how can we measure it? One simple way to consider this is to imagine you were typing these representations, so you could think of each of the original bits being stored as one character, and each of the RLE codes using a character for each digit and comma (this is a bit crude, but it's a starting point).

In the original representation, 225 digits (ones and zeroes) were required to represent the image. Count up the number of commas and digits (but not spaces or newlines, ignore those) in the new representation. This is the number of characters required to represent the image with the new representation (to ensure you are on the right track, the first 3 rows that were given to you contain 29 characters).

Assuming you got the new image representation correct, and counted correctly, you should have found there are 121 characters in the new image (double check if your number differs). This means that the new representation only requires around 54% as many characters to represent (calculated using 121/225). This is a significant reduction in the

amount of space required to store the image --- it's about half the size. The new representation is a compressed form of the old one.

#### **Curiosity:** Run length coding representation in practice

In practice this method (with some extra tricks) can be used to compress images to about 15% of their original size. In real systems, the original image only uses one bit for every pixel to store the black and white values (not one character, which we used for our calculations). However, the run length numbers are also stored much more efficiently, again using bit patterns that take very little space to represent the numbers. The bit patterns used are usually based on a technique called Huffman coding, but that is beyond what we want to get into here.

### 7.2.3. Where is Run Length Encoding used in practice?

The main place that black and white scanned images are used now is on fax machines, which use this approach to compression. One reason that it works so well with scanned pages is the number of consecutive white pixels is huge. In fact, there will be entire scanned lines that are nothing but white pixels. A typical fax page is 200 pixels across or more, so replacing 200 bits with one number is a big saving. The number itself can take a few bits to represent, and in some places on the scanned page only a few consecutive pixels are replaced with a number, but overall the saving is significant. In fact, fax machines would take 7 times longer to send pages if they didn't use compression.

#### **Project:** Using Run Length Encoding for yourself

Now that you know how run length encoding works, you can come up with and compress your own black and white image, as well as uncompress an image that somebody else has given you.

Start by making your own picture with ones and zeroes. (Make sure it is rectangular – all the rows should have the same length.) You can either draw this on paper or prepare it on a computer (using a fixed width font, otherwise it can become really frustrating and confusing!) In order to make it easier, you could start by working out what you want your image to be on grid paper (such as that from a math exercise book) by shading in squares to represent the black ones, and leaving them blank to represent the white ones. Once you have done that, you could then write out the zeroes and ones for the image.

Work out the compressed representation of your image using run length coding, i.e. the run lengths separated by commas form that was explained above.

Now give a copy of the *compressed representation* (the run length codes, not the original uncompressed representation) to a friend or classmate, along with an explanation of how it is compressed. Ask them to try and draw the image on some grid paper. Once they are done, check their conversion against your original.

Imagining that you and your friend are both computers, by doing this you have shown that images using these systems of representations can be compressed on one computer, and decompressed on another, as long as you have standards that you've agreed on (e.g. that every line begins with a white pixel). It is very important for compression algorithms to follow standards so that a file compressed on one computer can be decompressed on another; for example, songs often follow the "mp3" standard so that when they are downloaded they can be played on a variety of devices.

#### 7.2.4. Lossy vs Lossless compression

As the compressed representation of the image can be converted back to the original representation, and both the original representation and the compressed representation would give the same image when read by a computer, this compression algorithm is called *lossless*, i.e. none of the data was lost from compressing the image, and as a result the compression could be undone exactly.

Not all compression algorithms are lossless though. In some types of files, in particular photos, sound, and videos, we are willing to sacrifice a little bit of the quality (i.e. lose a little of the data representing the image) if it allows us to make the file size a lot smaller. For downloading very large files such as movies, this can be essential to ensure the file size is not so big that it is infeasible to download! These compression methods are called *lossy*. If some of the data is lost, it is impossible to convert the file back to exactly the original form when lossy compression was used, but the person viewing the movie or listening to the music may not mind the lower quality if the files are smaller. Later in this chapter, we will investigate the effects some lossy compression algorithms have on images and sound.

Interestingly, it turns out that any *lossless* compression algorithm will have cases where the compressed version of the file is larger than the uncompressed version! Computer scientists have even proven this to be the case, meaning it is impossible for anybody to ever come up with a lossless compression algorithm that makes *all* possible files smaller.

In most cases this isn't an issue though, as a good lossless compression algorithm will tend to give the best compression on common patterns of data, and the worst compression on ones that are highly unlikely to occur.

### **Challenge:** Best and worst cases of run length encoding

What is the image with the best compression (i.e. an image that has a size that is a very small percentage of the original) that you can come up with? This is the best case performance for this compression algorithm.

What about the worst compression? Can you find an image that actually has a *larger* compressed representation? (Don't forget the commas in the version we used!) This is the worst case performance for this compression algorithm.

### **Spoiler:** Answer for above challenge

The best case above is when the image is entirely white (only one number is used per line). The worst case is when every pixel is alternating black and white, so there's one number for every pixel. In fact, in this case the size of the compressed file is likely to be a little larger than the original one because the numbers are likely to take more than one bit to store. Real systems don't represent the data exactly as we've discussed here, but the issues are the same.

### **Curiosity:** Compression methods can expand files

In the worst case (with alternating black and white pixels) the run length encoding method will result in a file that's larger than the original! As noted above, every lossless compression method that makes at least one file smaller must also have some files that it makes larger --- it's not mathematically possible to have a method that always makes files smaller unless the method is lossy. As a trivial example, suppose someone claims to have a compression method that will convert any 3-bit file into a 2-bit file. How many different 3-bit files are there? (There are 8.) How many different 2-bit files are there? (There are 4.) Can you see the problem? We've got 8 possible files that we might want to compress, but only 4 ways to represent them. So some of them will have identical representations, and can't be decoded exactly.

Over the years there have been several frauds based on claims of a lossless compression method that will compress every file that it is given. This can only be

true if the method is lossy (loses information); all lossless methods must expand some files. It would be nice if all files could be compressed without loss; you could compress a huge file, then apply compression to the compressed file, and make it smaller again, repeating this until it was only one byte -- or one bit! Unfortunately, this isn't possible.

## 7.3. Image compression using JPEG

Images can take up a lot of space, and most of the time that pictures are stored on a computer they are compressed to avoid wasting too much space. With a lot of images (especially photographs), there's no need to store the image exactly as it was originally, because it contains way more detail than anyone can see. This can lead to considerable savings in space, especially if the details that are missing are the kind that people have trouble perceiving. This kind of compression is called lossy compression. There are other situations where images need to be stored exactly as they were in the original, such as for medical scans or very high quality photograph processing, and in these cases lossless methods are used, or the images aren't compressed at all (e.g. using RAW format on cameras).

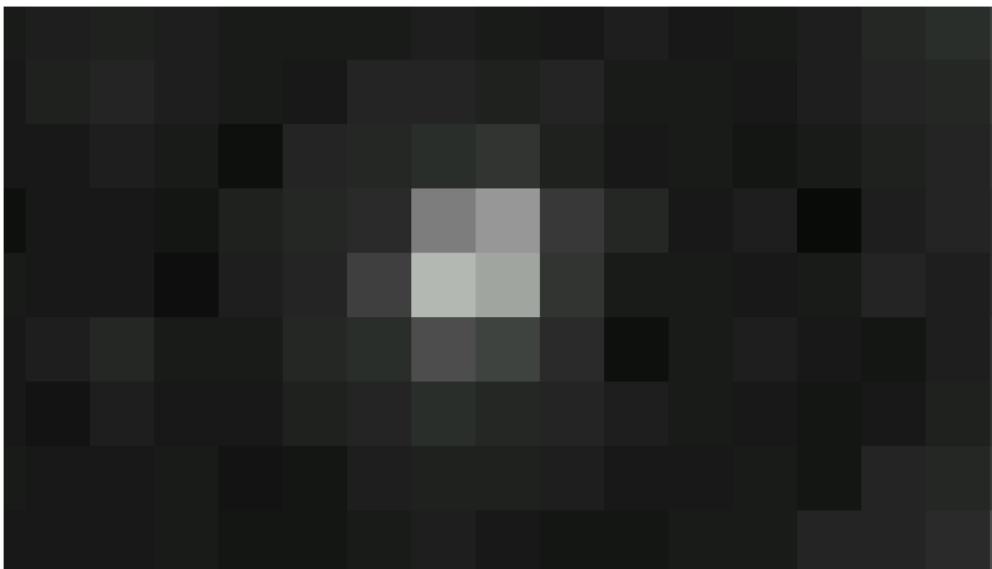
In the data representation section we looked at how the size of an image file can be reduced by using fewer bits to describe the colour of each pixel. However, image compression methods such as JPEG take advantage of patterns in the image to reduce the space needed to represent it, without impacting the image unnecessarily.

The following three images show the difference between reducing bit depth and using a specialised image compression system. The left hand image is the original, which was 24 bits per pixel. The middle image has been compressed to one third of the original size using JPEG; while it is a "lossy" version of the original, the difference is unlikely to be perceptible. The right hand one has had the number of colours reduced to 256, so there are 8 bits per pixel instead of 24, which means it is also stored in a third of the original size. Even though it has lost just as many bits, the information removed has had much more impact on how it looks. This is the advantage of JPEG: it removes information in the image that doesn't have so much impact on the perceived quality. Furthermore, with JPEG, you can choose the tradeoff between quality and file size.

Reducing the number of bits (the colour depth) is sufficiently crude that we don't really regard it as a compression method, but just a low quality representation. Image compression methods like JPEG, GIF and PNG are designed to take advantage of the patterns in an image to get a good reduction in file size without losing more quality than necessary.



For example, the following image shows a zoomed in view of the pixels that are part of the detail around an eye from the above (high quality) image.



Notice that the colours in adjacent pixels are often very similar, even in this part of the picture that has a lot of detail. For example, the pixels shown in the red box below just change gradually from very dark to very light.



Run-length encoding wouldn't work in this situation. You could use a variation that specifies a pixel's colour, and then says how many of the following pixels are the same colour, but although most adjacent pixels are nearly the same, the chances of them being identical are very low, and there would be almost no runs of identical colours.

But there is a way to take advantage of the gradually changing colours. For the pixels in the red box above, you could generate an approximate version of those colours by specifying just the first and last one, and getting the computer to calculate the ones in between assuming that the colour changes gradually between them. Instead of storing 5 pixel values, only 2 are needed, yet someone viewing it probably might not notice any difference. This would be *lossy* because you can't reproduce the original exactly, but it would be good enough for a lot of purposes, and save a lot of space.

### Jargon Buster: Interpolation

The process of guessing the colours of pixels between two that are known is an example of [interpolation](#). A *linear* interpolation assumes that the values increase at a constant rate between the two given values; for example, for the five pixels above, suppose the first pixel has a blue colour value of 124, and the last one has a blue value of 136, then a linear interpolation would guess that the blue values for the ones in between are 127, 130 and 133, and this would save storing them. In practice, a more complex approach is used to guess what the pixels are, but linear interpolation gives the idea of what's going on.

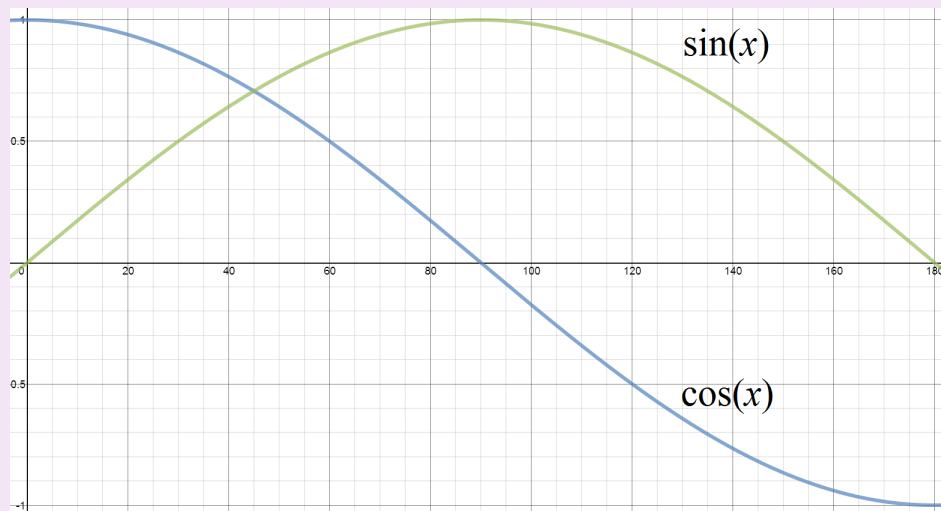
The JPEG system, which is widely used for photos, uses a more sophisticated version of this idea. Instead of taking a 5 by 1 run of pixels as we did above, it works with 8 by 8

blocks of pixels. And instead of estimating the values with a linear function, it uses combinations of cosine waves.

### Curiosity: What are cosine waves

A cosine wave form is from the trig function that is often used for calculating the sides of a triangle. If you plot the cosine value from 0 to 180 degrees, you get a smooth curve going from 1 to -1. Variations of this plot can be used to approximate the value of pixels, going from one colour to another. If you add in a higher frequency cosine wave, you can produce interesting shapes. In theory, any pattern of pixels can be created by adding together different cosine waves!

The following graph shows the values of  $\sin(x)$  and  $\cos(x)$  for  $x$  ranging from 0 to 180 degrees.

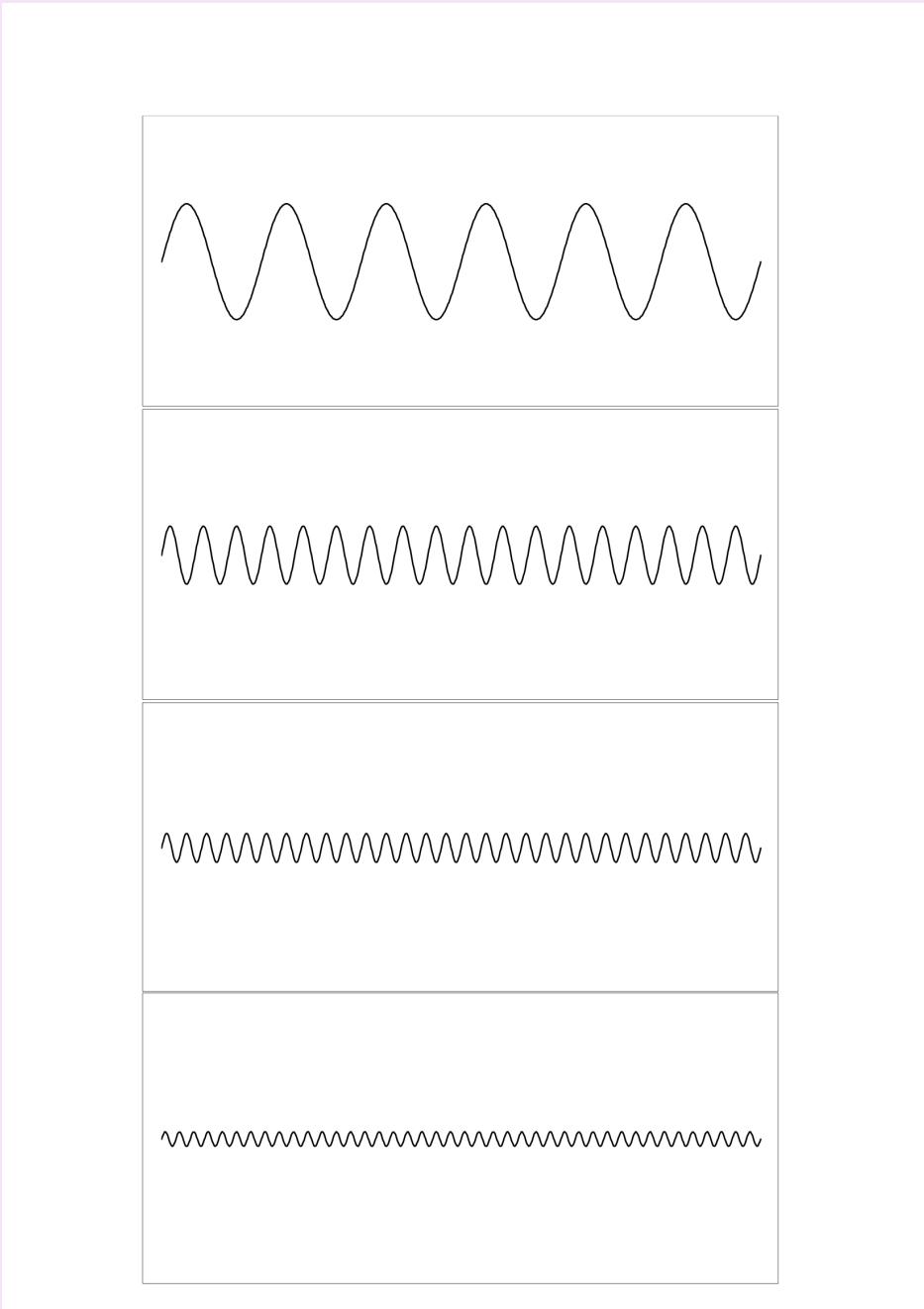


### Curiosity: Adding sine or cosine waves to create any waveform

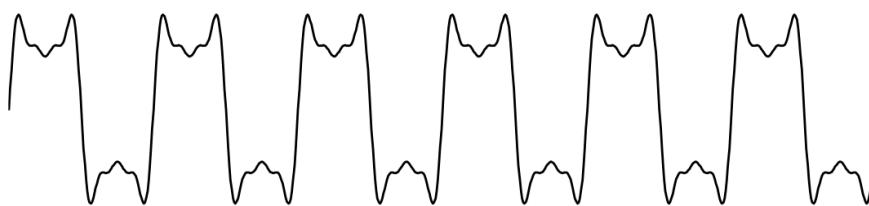
JPEGs (and MP3) are based on the idea that you can add together lots of sine or cosine waves to create any waveform that you want. Converting a waveform for a block of pixels or sample of music into a sum of simple waves can be done using a technique called a [Fourier transform](#), and is a widely used idea in signal processing.

You can experiment with adding sine waves together to generate other shapes using the [spreadsheet provided](#). In this spreadsheet, the yellow region on the first sheet allows you to choose which sine waves to add. Try setting the 4 sine waves to frequencies that are 3, 9, 15, and 21 times the fundamental frequency respectively (the "fundamental" is the lowest frequency.) Now set the "amplitude" (equivalent to

volume level) of the four to 0.5, 0.25, 0.125 and 0.0625 respectively (each is half of the previous one). This should produce the following four sine waves:



When the above four waves are added together, they interfere with each other, and produce a shape that has sharper transitions:



In fact, if you were to continue the pattern with more than four sine waves, this shape would become a "square wave", which is one that suddenly goes to the maximum value, and then suddenly to the minimum. The one shown above is bumpy because we've only used 4 sine waves to describe it.

This is exactly what is going on in JPEG if you compress a black and white image. The "colour" of pixels as you go across the image will either be 0 (black) or full intensity (white), but JPEG will approximate it with a small number of cosine waves (which have basically the same properties as sine waves.) This gives the "overshoot" that you see in the image above; in a JPEG image, this comes out as bright and dark patches surrounding the sudden change of colour, like here:



You can experiment with different combinations of sine waves to get different shapes. You may need to have more than four to get good approximations to a shape that you want; that's exactly the tradeoff that JPEG is making. There are some suggestions for parameters on the second sheet of the spreadsheet.

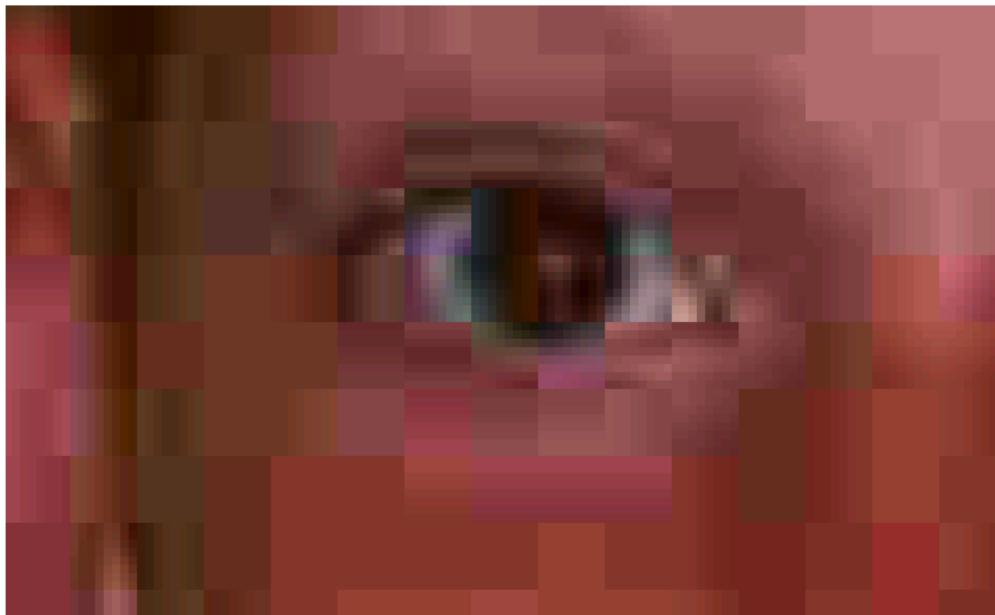
You can also learn about Fourier transforms using the Wolfram Alpha software; this will require you to install a browser plugin. The Wolfram demonstrations include: [an interactive demonstration of JPEG, showing the relationship between sine saves and](#)

creating other waveforms, and showing how sine waves can be summed to produce other shapes.

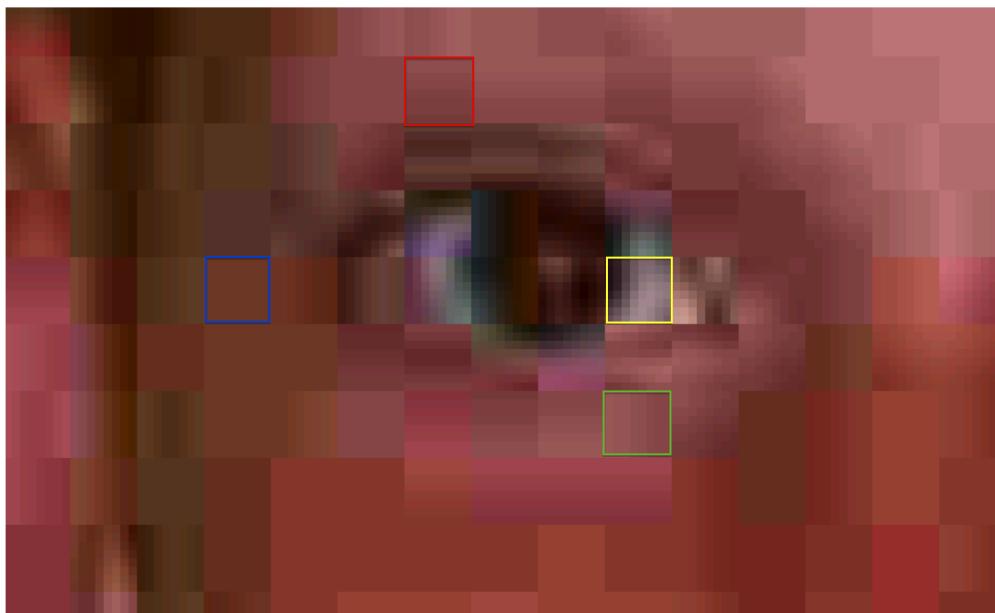
You can see the 8 by 8 blocks of pixels if you zoom in on a heavily compressed JPEG image. For example, the following image has been very heavily compressed using JPEG (it is just 1.5% of its original size).



If we zoom in on the eye area, you can see the 8 x 8 blocks of pixels:



Notice that there is very little variation across each block. In the following image the block in the red box only changes from top to bottom, and could probably be specified by giving just two values, and having the ones in between calculated by the decoder as for the line example before. The green square only varies from left to right, and again might only need 2 values stored instead of 64. The blue block has only one colour in it! The yellow block is more complicated because there is more activity in that part of the image, which is where the cosine waves come in. A "wave" value varies up and down, so this one can be represented by a left-to-right variation from dark to light to dark, and a top-to-bottom variation mainly from dark to light. Thus still only a few values need to be stored instead of the full 64.



The quality is quite low, but the saving in space is huge – it's more than 60 times smaller (for example, it would download 60 times faster). Higher quality JPEG images store more

detail for each 8 by 8 block, which makes it closer to the original image, but makes bigger files because more details are being stored. You can experiment with these tradeoffs by saving JPEGs with differing choices of the quality, and see how the file size changes. Most image processing software offers this option when you save an image as a JPEG.

**Jargon Buster:** Where does the term JPEG come from?

The name "JPEG" is short for "Joint Photographic Experts Group", a committee that was formed in the 1980s to create standards so that digital photographs could be captured and displayed on different brands of devices. Because some file extensions are limited to three characters, it is often seen as the ".jpg" extension.

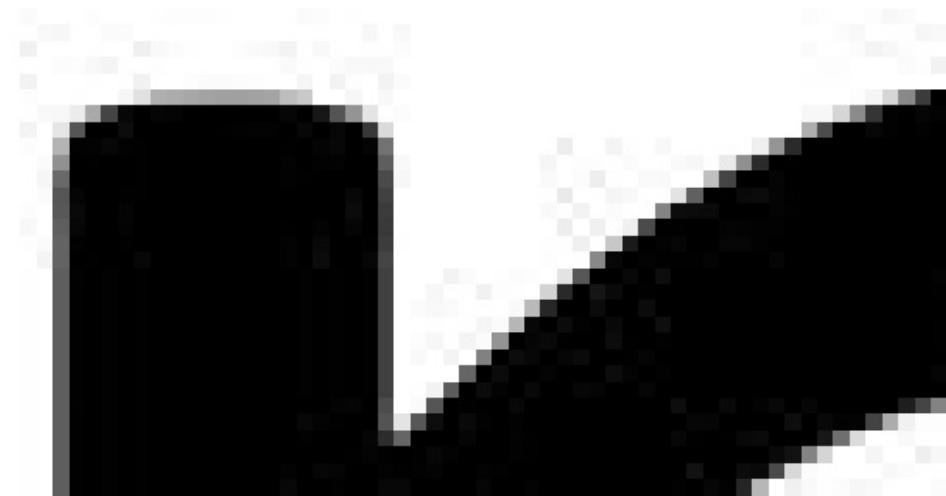
**Curiosity:** More about cosine waves

The cosine waves used for JPEG images are based on a "Discrete Cosine Transform". The "Discrete" means that the waveform is digital – it is the opposite of continuous, where any value can occur. In a JPEG wave, there are only  $8 \times 8$  values (for the block being coded), and each of those values can have a limited range of numbers (binary integers), rather than any value at all.

An important issue arises because JPEG represents images as smoothly varying colours: what happens if the colours change suddenly? In that case, lots of values need to be stored so that lots of cosine waves can be added together to make the sudden change in colour, or else the edge of the image become fuzzy. You can think of it as the cosine waves overshooting on the sudden changes, producing artifacts like the ones in the following image where the edges are messy.

The word "jpeg" is written in a large, bold, black font. The letters are lowercase and have a thick, rounded appearance. They are arranged horizontally, with "j" on the left, "p" in the middle, and "e g" on the right.

The original had sharp edges, but this zoomed in view of the JPEG version of it show that not only are the edges gradual, but some darker pixels occur further into the white space, looking a bit like shadows or echoes.



For this reason, JPEG is used for photos and natural images, but other techniques (such as GIF and PNG, which we will look at in another section) work better for artificial images like this one.

## 7.4. General purpose compression

General purpose compression methods need to be lossless because you can't assume that the user won't mind if the data is changed. The most widely used general purpose compression algorithms (such as ZIP, gzip, and rar) are based on a method called "Ziv-Lempel coding", invented by Jacob Ziv and Abraham Lempel in the 1970s.

We'll look at this with a text file as an example. The main idea of Ziv-Lempel coding is that sequences of characters are often repeated in files (for example, the sequence of characters "image " appears often in this chapter), and so instead of storing the repeated occurrence, you just replace it with a reference to where it last occurred. As long as the reference is smaller than the phrase being replaced, you'll save space. Typically this systems based on this approach can be used to reduce text files to as little as a quarter of their original size, which is almost as good as any method known for compressing text.

The following interactive allows you to explore this idea. The empty boxes have been replaced with a reference to the text occurring earlier. You can click on a box to see where the reference is, and you can type the referenced characters in to decode the text. What happens if a reference is pointing to another reference? As long as you decode them from first to last, the information will be available before you need it.

[View compression](#)

interactive

View the link online at [http://www.csfieldguide.org.nz/releases/1.9.9/\\_static/widgets/COMPRESSION/LWZ/public\\_html/index.html](http://www.csfieldguide.org.nz/releases/1.9.9/_static/widgets/COMPRESSION/LWZ/public_html/index.html)

You can also enter your own text by clicking on the "Text" tab. You could paste in some text of your own to see how many characters can be replaced with references.

The references are actually two numbers: the first says how many characters to count back to where the previous phrase starts, and the second says how long the referenced phrase is. Each reference typically takes about the space of one or two characters, so the system makes a saving as long as two characters are replaced. The options in the interactive above allow you to require the replaced length to be at least two, to avoid replacing a single character with a reference. Of course, all characters count, not just letters of the alphabet, so the system can also refer back to the white spaces between words. In fact, some of the most common sequences are things like a full stop followed by a space.

This approach also works very well for black and white images, since sequences like "10 white pixels" are likely to have occurred before. Here are some of the bits from the example earlier in this chapter; you can paste them into the interactive above to see how many pointers are needed to represent it.

```
011000010000110
100000111000001
000001111100000
000011111110000
000111111111000
001111101111100
011111000111110
111110000011111
```

In fact, this is essentially what happens with GIF and PNG images; the pixel values are compressed using the Ziv-Lempel algorithm, which works well if you have lots of

consecutive pixels the same colour. But it works very poorly with photographs, where pixel patterns are very unlikely to be repeated.

#### Curiosity: ZL or LZ compression?

The method we have described here is named “Ziv-Lempel” compression after Jacob Ziv and Abraham Lempel, the two computer scientists who invented it in the 1970s. Unfortunately someone mixed up the order of their names when they wrote an article about it, and called it “LZ” compression instead of “ZL” compression. So many people copied the mistake that Ziv and Lempel’s method is now usually called “LZ compression”!

## 7.5. Audio compression

One of the most widely used methods for compressing music is MP3, which is actually from a video compression standard called MPEG (Moving Picture Experts Group).

#### Curiosity: The naming of mp3

The name "mp3" isn't very self explanatory because the "mp" stands for "moving picture", and the 3 is from version 1, but mp3 files are used for music!

The full name of the standard that it comes from is MPEG, and the missing "EG" stands for "experts group", which was a consortium of companies and researchers that got together to agree on a standard so that people could easily play the same videos on different brands of equipment (so, for example, you could play the same DVD on any brand of DVD player). The very first version of their standards (called MPEG-1) had three methods of storing the sound track (layer 1, 2 and 3). One of those methods (MPEG-1 layer 3) became very popular for compressing music, and was abbreviated to MP3.

The MPEG-1 standard isn't used much for video now (for example, DVDs and TV mainly use MPEG-2), but it remains very important for audio coding.

The next MPEG version is MPEG-4 (MPEG-3 was redundant before it became a standard). MPEG-4 offers higher quality video, and is commonly used for digital video files, streaming media, Blu-Ray discs and some broadcast TV. The AAC audio compression method, used by Apple among others, is also from the MPEG-4

standard. On computers, MPEG-4 Part 14 is commonly used for video, and it's often abbreviated as "MP4."

So there you have it: MP3 stands for "MPEG-1 layer 3", and MP4 stands for "MPEG-4 part 14".

Most other audio compression methods use a similar approach to the MP3 method, although some offer better quality for the same amount of storage (or less storage for the same quality). We won't go into exactly how this works, but the general idea is to break the sound down into bands of different frequencies, and then represent each of those bands by adding together the values of a simple formula (the sum of cosine waves, to be precise).

There is some [more detail about how MP3 coding works on the cs4fn site](#), and also in [an article on the I Programmer site](#).

Other audio compression systems that you might come across include AAC, ALAC, Ogg Vorbis, and WMA. Each of these has various advantages over others, and some are more compatible or open than others.

The main questions with compressed audio are how small the file can be made, and how good the quality is of the human ear. (There is also the question of how long it takes to encode the file, which might affect how useful the system is.) The tradeoff between quality and size of audio files can depend on the situation you're in: if you are jogging and listening to music then the quality may not matter so much, but it's good to reduce the space available to store it. On the other hand, someone listening to a recording at home on a good sound system might not mind about having a large device to store the music, as long as the quality is high.

To evaluate an audio compression you should choose a variety of recordings that you have high quality originals for, typically on CD (or using uncompressed WAV or AIFF files). Choose different styles of music, and other kinds of audio such as speech, and perhaps even create a recording that is totally silent. Now convert these recordings to different audio formats. One system for doing this that is free to download is Apple's iTunes, which can be used to rip CDs to a variety of formats, and gives a choice of settings for the quality and size. A lot of other audio systems are able to convert files, or have plugins that can do the conversion.

Compress each of your recordings using a variety of methods, making sure that each compressed file is created from a high quality original. Make a table showing how long it took to process each recording, the size of the compressed file, and some evaluation of the quality of the sound compared with the original. Discuss the tradeoffs involved – do

you need much bigger files to store good quality sound? Is there a limit to how small you can make a file and still have it sounding ok? Do some methods work better for speech than others? Does a 2 minute recording of silence take more space than a 1 minute recording of silence? Does a 1 minute recording of music use more space than a minute of silence?

## 7.6. The whole story!

The details of how compression systems work have been glossed over in this chapter, as we have been more concerned about the file sizes and speed of the methods than how they work. Most compression systems are variations of the ideas that have been covered here, although one fundamental method that we haven't mentioned is Huffman coding, which turns out to be useful as the final stage of *all* of the above methods, and is often one of the first topics mentioned in textbooks discussing compression (there's a brief [explanation of it here](#)). A closely related system is Arithmetic coding (there's an [explanation of it here](#)). Also, video compression has been omitted, even though compressing videos saves more space than most kinds of compression. Most video compression is based on the "MPEG" standard (Moving Pictures Experts Group). There is some information about how this works in the [CS4FN article on "Movie Magic"](#).

The Ziv-Lempel method shown is a variation of the so-called "LZ77" method. Many of the more popular lossless compression methods are based on this, although there are many variations, and one called "LZW" has also been used a lot. Another high-compression general-purpose compression method is bzip, based on a very clever method called the Burrows-Wheeler Transform.

Questions like "what is the most compression that can be achieved" are addressed by the field of [information theory](#). There is an [activity on information theory on the CS Unplugged site](#), and there is a [fun activity that illustrates information theory](#). Based on this theory, it seems that English text can't be compressed to less than about 12% of its original size at the very best. Images, sound and video can get much better compression because they can use lossy compression, and don't have to reproduce the original data exactly.

## 7.7. Further reading

- "The Data Compression Book" by Mark Nelson and Jean-Loup Gailly is a good overview of this topic
- A list of books on this topic (and lots of other information about compression) is available from [The Data Compression Site](#).

- Gleick's book "The Information" has some background to compression, and coding in general.

## 7.7.1. Useful links

- Images, run-length-coding <http://csunplugged.org/image-representation> This is also relevant to binary representations in general, although is probably best used in the compression section.
- There is a detailed section on [JPEG encoding on Wikipedia](#).
- Text compression <http://csunplugged.org/text-compression>

## 7.7.2. Interesting articles

- [One pixel is worth three thousand words](#) by Jon Sneyers from [cloudinary.com](http://cloudinary.com)

# 8. Coding - Encryption

## 8.1. What's the big picture?

Encryption is used to keep data secret. In its simplest form, a file or data transmission is garbled so that only authorised people with a secret "key" can unlock the original text. If you're using digital devices then you'll be using systems based on encryption all the time: when you use online banking, when you access data through wifi, when you pay for something with a credit card (either by swiping, inserting or tapping), in fact, nearly every activity will involve layers of encryption. Without encryption, your information would be wide open to the world – anyone could pull up outside a house and read all the data going over your wifi, and stolen laptops, hard disks and SIM cards would yield all sorts of information about you – so encryption is critical to make computer systems usable.

An encryption system often consists of two computer programs: one to *encrypt* some data (referred to as *plaintext*) into a form that looks like nonsense (the *ciphertext*), and a second program that can *decrypt* the ciphertext back into the plaintext form. The encryption and decryption is carried out using some very clever math on the text with a chosen *key*. You will learn more about these concepts shortly.

Of course, we wouldn't need encryption if we lived in a world where everyone was honest and could be trusted, and it was ok for anyone to have access to all your personal information such as health records, online discussions, bank accounts and so on, and if you knew that no-one would interfere with things like aircraft control systems and computer controlled weapons. However, information is worth money, people value their privacy, and safety is important, so encryption has become fundamental to the design of computer systems. Even breaking the security on a traffic light system could be used to personal advantage.

### Curiosity: A case of hacking traffic lights

An interesting example of the value of using encryption outside of secret messages is the two engineers who were convicted of [changing traffic light patterns to cause chaos during a strike](#). A related problem in the US was traffic signals that could respond to codes from emergency vehicles to change to green; originally these didn't

use encryption, and people could figure out how to trigger them to their own advantage.

A big issue with encryption systems is people who want to break into them and decrypt messages without the key (which is some secret value or setting that can be used to unlock an encrypted file). Some systems that were used many years ago were discovered to be insecure because of attacks, so could no longer be used. It is possible that somebody will find an effective way of breaking into the widespread systems we use these days, which would cause a lot of problems.

Like all technologies, encryption can be used for good and bad purposes. A human rights organisation might use encryption to secretly send photographs of human rights abuse to the media, while drug traffickers might use it to avoid having their plans read by investigators. Understanding how encryption works and what is possible can help to make informed decisions around things like freedom of speech, human rights, tracking criminal activity, personal privacy, identity theft, online banking and payments, and the safety of systems that might be taken over if they were "hacked into".

**Jargon Buster:** Deciphering, Decrypting, Attacking, Cracking, Hacking, Cryptanalysts, Hackers, and Crackers

There are various words that can be used to refer to trying to get the plaintext from a ciphertext, including decipher, decrypt, crack, and cryptanalysis. Often the process of trying to break cryptography is referred to as an "attack". The term "hack" is also sometimes used, but it has other connotations, and is only used informally.

People who try to decrypt messages are called cryptanalysts; more informal terms like hackers and crackers are sometimes used, generally with the implication that they have bad intentions. Being a cryptanalyst is generally a good thing to do though: people who use encryption systems actually want to know if they have weaknesses, and don't want to wait until the bad guys find out for them. It's like a security guard checking doors on a building; the guard hopes that they can't get in, but if a door is found unlocked, they can do something about it to make sure the bad guys can't get

in. Of course, if a security guard finds an open door, and takes advantage of that to steal something for themselves, they're no longer doing their job properly!

## 8.2. Substitution Ciphers

### 8.2.1. Getting Started with Caesar Cipher

In this section, we will be looking at a simple substitution cipher called Caesar Cipher. Caesar Cipher is over 2000 years old, invented by a guy called Julius Caesar. Before we go any further, have a go at cracking this simple code. If you're stuck, try working in a small group with friends and classmates so that you can discuss ideas. A whiteboard or pen and paper would be helpful for doing this exercise.

```
DRO BOCMEO WSCCSYX GSVV ECO K ROVSMYZDOB,
KBBFSXQ KD XYYX DYWYBBYG.
LO BOKNI DY LBOKU YED KC CYYX
KC IYE ROKB DRB00
LVKCDC YX K GRSCDV0.
S'VV LO GOKBSXQ K BON KBWLKXN.
```

Once you have figured out what the text says, make a table with the letters of the alphabet in order and then write the letter they are represented with in the cipher text. You should notice an interesting pattern.

Given how easily broken this cipher is, you probably don't want your bank details encrypted with it. In practice, far stronger ciphers are used, although for now we are going to look a little bit further at Caesar Cipher, because it is a great introduction to the many ideas in encryption.

### 8.2.2. How does the Caesar Cipher work?

When you looked at the Caesar Cipher in the previous section and (hopefully) broke it and figured out what it said, you probably noticed that there was a pattern in how letters from the original message corresponded to letters in the decoded one. Each letter in the original message decoded to the letter that was 10 places before it in the alphabet. The conversion table you drew should have highlighted this. Here's the table for the letter correspondences, where the letter "K" translates to an "A". It is okay if your conversion table mapped the opposite way, i.e. "A" to "K" rather than "K" to "A". If you were unable to break the Caesar Cipher in the previous section, go back to it now and decode it using the 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
Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P

For this example, we say the key is 10 because keys in Caesar Cipher are a number between 1 and 25 (think carefully about why we wouldn't want a key of 26!), which specify how far the alphabet should be rotated. If instead we used a key of 8, the conversion table would be as follows.

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

### Jargon Buster: What is a key?

In a Caesar Cipher, the key represents how many places the alphabet should be rotated. In the examples above, we used keys of "8" and "10". More generally though, a key is simply a value that is required to do the math for the encryption and decryption. While Caesar Cipher only has 25 possible keys, real encryption systems have an incomprehensibly large number of possible keys, and preferably use keys which contains hundreds or even thousands of binary digits. Having a huge number of different possible keys is important, because it would take a computer less than a second to try all 25 Caesar Cipher keys.

In the physical world, a combination lock is completely analogous to a cipher (in fact, you could send a secret message in a box locked with a combination lock.) We'll assume that the only way to open the box is to work out the combination number. The combination number is the *key* for the box. If it's a three-digit lock, you'll only have 1000 values to try out, which might not take too long. A four-digit lock has 10 times as many values to try out, so is way more secure. Of course, there may be ways to reduce the amount of work required - for example, if you know that the person who locked it never has a correct digit showing, then you only have 9 digits to guess for each place, rather than 10, which would take less than three quarters of the time!

Try experimenting with the following interactive for Caesar Cipher. You will probably want to refer back to it later while working through the remainder of the sections on Caesar Cipher.

Use the interactive online at <http://www.csfieldguide.org.nz/en/interactives/caesar-cipher/index.html>

### 8.2.2.1. Decryption with Caesar Cipher

Before we looked at how to *crack* Caesar cipher – getting the plaintext from the ciphertext without being told the key beforehand. It is even easier to *decrypt* Caesar Cipher when we **do** have the key. In practice, a good encryption system ensures that the plaintext cannot be obtained from the ciphertext without the key, i.e. it can be *decrypted* but not *cracked*.

As an example of *decrypting* with Caesar Cipher, assume that we have the following ciphertext, and that the key is 6.

```
ZNK WAOIQ HXUCT LUD PASVY UBKX ZNK RGFE JUM
```

Because we know that the key is 6, we can subtract 6 places off each character in the ciphertext. For example, the letter 6 places before "Z" is "T", 6 places before "N" is "H", and 6 places before "K" is "E". From this, we know that the first word must be "THE". Going through the entire ciphertext in this way, we can eventually get the plaintext of:

```
THE QUICK BROWN FOX JUMPS OVER THE LAZY DOG
```

The interactive above can do this process for you. Just put the ciphertext into the box on the right, enter the key, and tell it to decrypt. You should ensure you understand how to encrypt messages yourself though!

#### Challenge: Decrypting a Caesar Cipher

##### Challenge 1

Decrypt the following message using Caesar Cipher. The key is 4.

```
HIGVCTXMRK GEIWEV GMTLIV MW IEWC
```

##### Challenge 2

What is the key for the following *cipher text*?

THIS IS A TRICK QUESTION

### 8.2.2.2. Encryption with Caesar Cipher

Encryption is equally straightforward. Instead of rotating backwards (subtracting) like we did for decrypting, we rotate forwards (add) the key to each letter in the plaintext. For example, assume we wanted to encrypt the following text with a key of 7.

HOW ARE YOU

We would start by working that the letter that is 7 places ahead of "H" is "O", 7 places ahead of "O" is "V", and 7 places ahead of "W" is "D". This means that the first word of the plaintext encrypts to "OVD" in the ciphertext. Going through the entire plaintext in this way, we can eventually get the ciphertext of:

OVD HYL FVB

#### **Challenge:** Encrypting with Caesar Cipher

##### **Challenge 1**

Encrypt the following message using Caesar Cipher and a key of 20

JUST ANOTHER RANDOM MESSAGE TO ENCRYPT

##### **Challenge 2**

Why is using a key of 26 on the following message not a good idea?

USING A KEY OF TWENTY SIX IN CAESAR CIPHER IS NOT A GOOD IDEA

#### **Curiosity:** ROT13 Caesar Cipher

The Caesar cipher with a key of 13 is the same as an approach called [ROT13 \(rotate 13 characters\)](#), which is sometimes used to obscure things like the punchline of a

joke, a spoiler for a story, the answer to a question, or text that might be offensive. It is easy to decode (and there are plenty of automatic systems for doing so), but the user has to deliberately ask to see the deciphered version. A key of 13 for a Caesar cipher has the interesting property that the encryption method is identical to the decryption method i.e. the same program can be used for both. Many strong encryption methods try to make the encryption and decryption processes as similar as possible so that the same software and/or hardware can be used for both parts of the task, generally with only minor adaptions.

### 8.2.3. Problems with Substitution Ciphers

**Jargon Buster:** What is a substitution cipher?

A substitution cipher simply means that each letter in the plaintext is substituted with another letter to form the ciphertext. If the same letter occurs more than once in the plaintext then it appears the same at each occurrence in the ciphertext. For example the phrase "HELLO THERE" has multiple H's, E's, and L's. All the H's in the plaintext might change to "C" in the ciphertext for example. Caesar Cipher is an example of a substitution cipher. Other substitution ciphers improve on the Caesar cipher by not having all the letters in order, and some older written ciphers use different symbols for each symbol. However, substitution ciphers are easy to attack because a statistical attack is so easy: you just look for a few common letters and sequences of letters, and match that to common patterns in the language.

So far, we have considered one way of cracking Caesar cipher: using patterns in the text. By looking for patterns such as one letter words, other short words, double letter patterns, apostrophe positions, and knowing rules such as all words must contain at least one of a, e, i, o, u, or y (excluding some acronyms and words written in txt language of course), cracking Caesar Cipher by looking for patterns is easy. Any good cryptosystem should not be able to be analysed in this way, i.e. it should be *semantically secure*

**Jargon Buster:** What do we mean by semantically secure?

Semantically secure means that there is no known efficient algorithm that can use the ciphertext to get any information about the plaintext, other than the length of the message. It is very important that cryptosystems used in practice are semantically secure.

As we saw above, Caesar Cipher is not semantically secure.

There are many other ways of cracking Caesar cipher which we will look at in this section. Understanding various common attacks on ciphers is important when looking at sophisticated cryptosystems which are used in practice.

### 8.2.3.1. Frequency Analysis Attacks

Frequency analysis means looking at how many times each letter appears in the encrypted message, and using this information to crack the code. A letter that appears many times in a message is far more likely to be "T" than "Z", for example.

The following interactive will help you analyze a piece of text by counting up the letter frequencies. You can paste in some text to see which are the most common (and least common) characters.

Use the interactive online at <http://www.csfieldguide.org.nz/en/interactives/frequency-analysis/index.html>

The following text has been coded using a Caesar cipher. To try to make sense of it, paste it into the statistical analyser above.

```
F QTSF RJXXFLJ HTSYFNSX QTYX TK
XYFYNXYNHFQ HQZJX YMFY HFS GJ
ZXJI YT FSFQDXJ BMFY YMJ RTXY
KWJVZJSY QJYYJWX FWJ, FSI JAJS
YMJ RTXY HTRRTS UFNWX TW YWNUQJX
TK QJYYJWX HFS MJQU YT GWJFP
YMJ HTIJ
```

"E" is the most common letter in the English alphabet. It is therefore a reasonable guess that "J" in the ciphertext represents "E" in the plaintext. Because "J" is 5 letters ahead of "E" in the alphabet, we can guess that the key is 5. If you put the ciphertext into the above interactive and set a key of 5, you will find that this is indeed the correct key.

Use the interactive online at <http://www.csfieldguide.org.nz/en/interactives/caesar-cipher/index.html>

**Spoiler:** Decrypted message

The message you should have decrypted is:

A LONG MESSAGE CONTAINS LOTS OF STATISTICAL CLUES THAT CAN BE USED TO ANALYSE WHAT THE MOST FREQUENT LETTERS ARE, AND EVEN THE MOST COMMON PAIRS OR TRIPLES OF LETTERS CAN HELP TO BREAK THE CODE

As the message says, long messages contain a lot of statistical clues. Very short messages (e.g. only a few words) are unlikely to have obvious statistical trends. Very long messages (e.g. entire books) will *almost* always have "E" as the most common letter. Wikipedia has a [list of letter frequencies](#), which you might find useful.

### Challenge: Frequency Analysis

Put the ciphertext into the above frequency analyser, guess what the key is (using the method explained above), and then try using that key with the ciphertext in the interactive above. Try to guess the key with as few guesses as you can!

#### Challenge 1

```
WTGT XH PCDIWTG BTHPVT IWPI NDJ HWDJAS WPKT CD IGDJQAT QGTPZXCV LXIW ATIITG  
UGTFJTCRN PCPANHXH
```

#### Challenge 2

```
OCDN ODHZ OCZ HZNNVBZ XJI0VDIN GJON JA OCZ GZOOZM 0, RCDXC DN OCZ NZXJIY HJNO  
XJHHJI GZOOZM DI OCZ VGKCVWZO
```

#### Challenge 3

```
BGDTCU BCEJ, BCXKGT, CPF BCPG BQQ0GF VJTQWIJ VJG BQQ
```

### Curiosity: The letter E isn't always the most common letter...

Although in almost all English texts the letter E is the most common letter, it isn't always. For example, the [1939 novel Gadsby by Ernest Vincent Wright](#) doesn't contain

a single letter E (this is called a lipogram). Furthermore, the text you're attacking may not be English. During World War 1 and 2, the US military had many Native American **Code talkers** translate messages into their own language, which provided a strong layer of security at the time.

### **Curiosity:** The Vigenere Cipher

A slightly stronger cipher than the Caesar cipher is the [Vigenere cipher](#), which is created by using multiple Caesar ciphers, where there is a key phrase (e.g. "acb"), and each letter in the key gives the offset (in the example this would be 1, 3, 2). These offsets are repeated to give the offset for encoding each character in the plaintext.

By having multiple Caesar ciphers, common letters such as E will no longer stand out as much, making frequency analysis a lot more challenging. The following website shows the effect on the distribution. [http://www.simonsingh.net/The\\_Black\\_Chamber/vigenere\\_strength.html](http://www.simonsingh.net/The_Black_Chamber/vigenere_strength.html)

However, while this makes the Vigenere cipher more challenging to crack than the Caesar cipher, ways have been found to crack it quickly. In fact, once you know the key length, it just breaks down to cracking several Caesar ciphers (which as you have seen is straightforward, and you can even use frequency analysis on the individual Caesar Ciphers!) Several statistical methods have been devised for working out the key length.

Attacking the Vigenere cipher by trying every possible key is hard because there are a lot more possible keys than for the Caesar cipher, but a statistical attack can work quite quickly. The Vigenere cipher is known as a *polyalphabetic substitution cipher*, since it uses multiple substitution rules.

### 8.2.3.2. Known Plain Text Attacks

Another kind of attack is the *known plaintext* attack, where you know part or all of the solution. For example, if you know that I start all my messages with "HI THERE", you can easily determine the key for the following message.

```
AB MAXKX LXVKXM FXXMBGZ TM MPH TF MANKLWTR
```

Even if you did not know the key used a simple rotation (not all substitution ciphers are), you have learnt that A->H, B->I, M->T, X->E, and K->R. This goes a long way towards deciphering the message. Filling in the letters you know, you would get:

```
AB MAXKX LXVKXM FXXMBGZ TM MPH TF MANKLWTR
HI THERE _E__ET _EETI__ _T T__ TH_____
```

By using the other tricks above, there are a very limited number of possibilities for the remaining letters. Have a go at figuring it out.

**Spoiler:** The above message is...

The deciphered message is:

```
HI THERE SECRET MEETING AT TWO AM THURSDAY
```

A known plaintext attack breaks a Caesar cipher straight away, but a good cryptosystem shouldn't have this vulnerability because it can be surprisingly easy for someone to know that a particular message is being sent. For example, a common message might be "Nothing to report", or in online banking there are likely to be common messages like headings in a bank account or parts of the web page that always appear. Even worse is a *chosen plaintext attack*, where you trick someone into sending your chosen message through their system so that you can see what its ciphertext is.

For this reason, it is essential for any good cryptosystem to not be breakable, even if the attacker has pieces of plaintext along with their corresponding ciphertext to work with. For this, the cryptosystem should give different ciphertext each time the same plaintext message is encrypted. It may initially sound impossible to achieve this, although there are several clever techniques used by real cryptosystems.

**Curiosity:** More general substitution ciphers

While Caesar cipher has a key specifying a rotation, a more general substitution cipher could randomly scramble the entire alphabet. This requires a key consisting of a sequence of 26 letters or numbers, specifying which letter maps onto each other one. For example, the first part of the key could be "D, Z, E", which would mean D: A, Z: B, E:C. The key would have to have another 23 letters in order to specify the rest of the mapping.

This increases the number of possible keys, and thus reduces the risk of a brute force attack. A can be substituted for any of the 26 letters in the alphabet, B can then be substituted for any of the 25 remaining letters (26 minus the letter already substituted for A), C can then be substituted for any of the 24 remaining letters...

This gives us 26 possibilities for A times 25 possibilities for B times 24 possibilities for C.. all the way down to 2 possibilities for Y and 1 possibility for Z.

$$\begin{aligned} 26 \times 25 \times 24 \times 23 \times 22 \times 21 \times 20 \times 19 \times 18 \times 17 \times \\ 16 \times 15 \times 14 \times 13 \times 12 \times 11 \times 10 \times 9 \times 8 \times 7 \times 6 \times \\ 5 \times 4 \times 3 \times 2 \times 1 = 26! \end{aligned}$$

Representing each of these possibilities requires around 88 bits, making the cipher's key size around 88 bits, which is below modern standards, although still not too bad!

However, this only solves one of the problems. The other techniques for breaking Caesar cipher we have looked at are still highly effective on all substitution ciphers, in particular the frequency analysis. For this reason, we need better ciphers in practice, which we will look at shortly.

### 8.2.3.3. Brute force Attacks

Another approach to cracking a ciphertext is a *brute force attack*, which involves trying out all possible keys, and seeing if any of them produce intelligible text. This is easy for a Caesar cipher because there are only 25 possible keys. For example, the following ciphertext is a single word, but is too short for a statistical attack. Try putting it into the decoder above, and trying keys until you decipher it.

EIJUDJQJYEKI

These days encryption keys are normally numbers that are 128 bits or longer. You could calculate how long it would take to try out every possible 128 bit number if a computer could test a million every second (including testing if each decoded text contains English words). It will eventually crack the message, but after the amount of time it would take, it's unlikely to be useful anymore – and the user of the key has probably changed it!

In fact, if we analyse it, a 128 bit key at 1,000,000 per second would take 10,790,283,070,000,000,000,000 years to test. Of course, it might find something in the first year, but the chances of that are ridiculously low, and it would be more realistic to hope to win the top prize in Lotto three times consecutively (and you'd probably get more money). On average, it will take around half that amount, i.e. a bit more than 5,000,000,000,000,000,000 years. Even if you get a really fast computer that can check one trillion keys a second (rather unrealistic in practice), it would still take around

5,000,000,000,000 years. Even if you could get one million of those computers (even more unrealistic in practice), it would still take 5,000,000 years.

And even if you did have the hardware that was considered above, then people would start using bigger keys. Every bit added to the key will double the number of years required to guess it. Just adding an extra 15 or 20 bits to the key in the above example will safely push the time required back to well beyond the expected life span of the Earth and Sun! This is how real cryptosystems protect themselves from brute force attacks. Cryptography relies a lot on low probabilities of success.

The calculator below can handle really big numbers. You can double check our calculations above if you want! Also, work out what would happen if the key size was double (i.e. 256 bits), or if a 1024 or 2048 bit key (common these days) was used.

### **Curiosity:** Tractability – problems that take too long to solve

Brute force attacks try out every possible key, and the number of possible keys grows *exponentially* as the key gets longer. As we saw above, no modern computer system could try out all possible 128 bit key values in a useful amount of time, and even if it were possible, adding just one more bit would double how long it would take.

In computer science, problems that take an exponential amount of time to solve are generally regarded as not being [tractable](#) --- that is, you can't get any traction on them; it's as if you're spinning your wheels. Working out which problems are tractable and which are intractable is a major area of research in computer science --- many other problems that we care about appear to be intractable, much to our frustration. The area of encryption is one of the few situations where we're pleased that an algorithm is intractible!

This guide has a [whole chapter about tractability](#), where you can explore these issues further.

### **Jargon Buster:** Terminology you should now be familiar with

The main terminology you should be familiar with now is that a *plaintext* is *encrypted* by a *cipher* to create a *ciphertext* using an *encryption key*. Someone without the encryption key who wants to *attack* the cipher could try various approaches, including a *brute force attack* (trying out all possible keys), a *frequency analysis attack* (looking for statistical patterns), and a *known plaintext attack* (matching some known text with the cipher to work out the key).

If you were given an example of a simple cipher being used, you should be able to talk about it using the proper terminology.

## 8.3. Cryptosystems used in practice

The substitution systems described above don't provide any security for modern digital systems. In the remainder of this chapter we will look at encryption systems that are used in practice. The first challenge is to find a way to exchange keys --- after all, if you're communicating to someone over the internet, how are you going to send the key for your secret message to them conveniently?

Cryptosystems are also used for purposes such as *authentication* (checking a password). This sounds simple, but how do you check when someone logs in, *without* having to store their password (after all, if someone got hold of the password list, that could ruin your reputation and business, so it's even safer not to store them.)

There are good solutions to these problems that are regularly used --- in fact, you probably use them online already, possibly without even knowing! We'll begin by looking at systems that allow people to decode secret messages without even having to be sent the key!

## 8.4. The Key Distribution Problem

**Curiosity:** Who are Alice, Bob, and Eve?

When describing an encryption scenario, cryptographers often use the fictitious characters "Alice" and "Bob", with a message being sent from Alice to Bob (A to B). We always assume that someone is eavesdropping on the conversation (in fact, if you're using a wireless connection, it's trivial to pick up the transmissions between Alice and Bob as long as you're in reach of the wireless network that one of them is using). The fictitious name for the eavesdropper is usually Eve.



[Image source](#)

There are several other characters used to describe activities around encryption protocols: for example Mallory (a malicious attacker) and Trudy (an intruder).

Wikipedia has a [list of Alice and Bob's friends](#)

If Alice is sending an encrypted message to Bob, this raises an interesting problem in encryption. The ciphertext itself can safely be sent across an “unsafe” network (one that Eve is listening on), but the key cannot. How can Alice get the key to Bob? Remember the key is the thing that tells Bob how to convert the ciphertext back to plaintext. So Alice can't include it in the encrypted message, because then Bob would be unable to access it. Alice can't just include it as plaintext either, because then Eve will be able to get ahold of it and use it to decrypt any messages that come through using it. You might ask why Alice doesn't just encrypt the key using a different encryption scheme, but then how will Bob know the new key? Alice would need to tell Bob the key that was used to encrypt it... and so on... this idea is definitely out!

Remember that Alice and Bob might be in different countries, and can only communicate through the internet. This also rules out Alice simply passing Bob the key in person.

**Curiosity:** Are we being paranoid?

In computer security we tend to assume that Eve, the eavesdropper, can read every message between Alice and Bob. This sounds like an inordinate level of wire tapping, but what about wireless systems? If you're using wireless (or a mobile phone), then all your data is being broadcast, and can be picked up by any wireless receiver in the vicinity. In fact, if another person in the room is also using wireless, their computer is already picking up everything being transmitted by your computer, and has to go to some trouble to ignore it.

Even in a wired connection, data is passed from one network node to another, stored on computers inbetween. Chances are that everyone who operates those computers is trustworthy, but probably don't even know which companies have handled your data in the last 24 hours, let alone whether every one of their employees can be trusted.

So assuming that someone can observe all the bits being transmitted and received from your computer is pretty reasonable.

Distributing keys physically is very expensive, and up to the 1970s large sums of money were spent physically sending keys internationally. Systems like this are called *symmetric* encryption, because Alice and Bob both need an identical copy of the key. The breakthrough was the realisation that you could make a system that used different keys for encoding and decoding. We will look further at this in the next section.

#### **Curiosity:** Some additional viewing

[Simon Singh's video](#) gives a good explanation of key distribution.

Additionally, there's a video illustrating how public key systems work using a padlock analogy.

Watch the video online at <https://www.youtube.com/watch?v=a72fHRr6MRU>

### 8.4.1. Public Key Systems

One of the remarkable discoveries in computer science in the 1970s was a method called *public key encryption*, where it's fine to tell everyone what the key is to encrypt any messages, but you need a special private key to decrypt it. Because Alice and Bob use different keys, this is called an *asymmetric* encryption system.

It's like giving out padlocks to all your friends, so anyone can lock a box and send it to you, but if you have the only (private) key, then you are the only person who can open the boxes. Once your friend locks a box, even they can't unlock it. It's really easy to distribute the padlocks. Public keys are the same – you can make them completely public – often people put them on their website or attach them to all emails they send. That's quite different to having to hire a security firm to deliver them to your colleagues.

Public key encryption is very heavily used for online commerce (such as internet banking and credit card payment) because your computer can set up a connection with the business or bank automatically using a public key system without you having to get together in advance to set up a key. Public key systems are generally slower than symmetric systems, so the public key system is often used to then send a new key for a symmetric system just once per session, and the symmetric key can be used from then on with a faster symmetric encryption system.

A very popular public key system is RSA. For this section on public key systems, we will use RSA as an example.

### 8.4.1.1. Generating the encryption and decryption keys

Firstly, you will need to generate a pair of keys using the key generator interactive. You should *keep the private key secret*, and *publicly announce the public key* so that your friends can send you messages (e.g. put it on the whiteboard, or email it to some friends). Make sure you save your keys somewhere so you don't forget them – a text file would be best.

Use the interactive online at <http://www.csfieldguide.org.nz/en/interactives/rsa-key-generator/index.html>

### 8.4.1.2. Encrypting messages with the public key

This next interactive is the encrypter, and it is used to encrypt messages with your **public key**. Your friends should use this to encrypt messages for you.

Use the interactive online at <http://www.csfieldguide.org.nz/en/interactives/rsa-no-padding/index.html>

To ensure you understand, try encrypting a short message with your **public key**. In the next section, there is an interactive that you can then use to decrypt the message with your private key.

### 8.4.1.3. Decrypting messages with the private key

Finally, this interactive is the decrypter. It is used to decrypt messages that were encrypted with your public key. In order to decrypt the messages, you will need your **private key**.

Use the interactive online at <http://www.csfieldguide.org.nz/en/interactives/rsa-no-padding/index.html?mode=decrypt>

Despite even your enemies knowing your public key (as you publicly announced it), they cannot use it to decrypt your messages which were encrypted using the public key. You are the only one who can decrypt messages, as that requires the private key which hopefully you are the only one who has access to.

Note that this interactive's implementation of RSA is just for demonstrating the concepts here and is not quite the same as the implementations used in live encryption systems.

#### CURIOSITY: Can we reverse the RSA calculations?

If you were asked to multiply the following two big prime numbers, you might find it a bit tiring to do by hand (although it is definitely achievable), and you could get an answer in a fraction of a second using a computer.

9739493281774982987432737457439209893878938489723948984873298423989898398696987090

3498372473234549852367394893403202898485093868948989658677273900243088492048950834

If on the other hand you were asked which two prime numbers were multiplied to get the following big number, you'd have a lot more trouble! (If you do find the answer, let us know! We'd be very interested to hear about it!)

3944604857329435839271430640488525351249090163937027434471421629606310815805347209

Creating an RSA code involves doing the multiplication above, which is easy for computers. If we could solve the second problem and find the multiples for a big number, we'd be able to crack an RSA code. However, no one knows a fast way to do that. This is called a "trapdoor" function - it's easy to go into the trapdoor (multiply two numbers), but it's pretty much impossible to get back out (find the two factors).

So why is it that despite these two problems being similar, one of them is “easy” and the other one is “hard”? Well, it comes down to the algorithms we have to solve each of the problems.

You have probably done long multiplication in school by making one line for each digit in the second number and then adding all the rows together. We can analyse the speed of this algorithm, much like we did in the algorithms chapter for sorting and searching. Assuming that each of the two numbers has the same number of digits, which we will call  $n$  (“Number of digits”), we need to write  $n$  rows. For each of those  $n$  rows, we will need to do around  $n$  multiplications. That gives us  $n \times n$  little multiplications. We need to add the  $n$  rows together at the end as well, but that doesn’t take long so lets ignore that part. We have determined that the number of small multiplications needed to multiply two big numbers is approximately the square of the number of digits. So for two numbers with 1000 digits, that’s 1,000,000 little multiplication operations. A computer can do that in less than a second! If you know about Big-O notation, this is an  $O(n^2)$  algorithm, where  $n$  is the number of digits. Note that some slightly better algorithms have been designed, but this estimate is good enough for our purposes.

For the second problem, we’d need an algorithm that could find the two numbers that were multiplied together. You might initially say, why can’t we just reverse the multiplication? The reverse of multiplication is division, so can’t we just divide to get the two numbers? It’s a good idea, but it won’t work. For division we need to know the big number, and one of the small numbers we want to divide into it, and that will give us the other small number. But in this case, we *only* know the big number. So it isn’t a straightforward long division problem at all! It turns out that there is no known fast algorithm to solve the problem. One way is to just try dividing by every number that is less than the number (well, we only need to go up to the square root, but that doesn’t help much!) There are still billions of billions of billions of numbers we need to check. Even a computer that could check 1 billion possibilities a second isn’t going to help us much with this! If you know about Big-O notation, this is an  $O(10^n)$  algorithm, where  $n$  is the number of digits -- even small numbers of digits are just too much to deal with!

There are slightly better solutions, but none of them shave off enough time to actually be useful for problems of the size of the one above!

The chapter on [complexity and tractability](#) looks at more computer science problems that are surprisingly challenging to solve. If you found this stuff interesting, do read about Complexity and Tractability when you are finished here!

### **Curiosity:** Encrypting with the private key instead of the public key --- Digital Signatures!

In order to encrypt a message, the public key is used. In order to decrypt it, the corresponding private key must be used. But what would happen if the message was encrypted using the *private key*? Could you then decrypt it with the public key?

Initially this might sound like a pointless thing to do --- why would you encrypt a message that can be decrypted using a key that everybody in the world can access!?! It turns out that indeed, encrypting a message with the private key and then decrypting it with the public key works, and it has a very useful application.

The only person who is able to *encrypt* the message using the *private key* is the person who owns the private key. The public key will only decrypt the message if the private key that was used to encrypt it actually is the public key's corresponding private key. If the message can't be decrypted, then it could not have been encrypted with that private key. This allows the sender to prove that the message actually is from them, and is known as a [digital signature](#).

You could check that someone is the authentic private key holder by giving them a phrase to encrypt with their private key. You then decrypt it with the public key to check that they were able to encrypt the phrase you gave them.

This has the same function as a physical signature, but is more reliable because it is essentially impossible to forge. Some email systems use this so that you can be sure an email came from the person who claims to be sending it.

## 8.5. Storing Passwords Securely

A really interesting puzzle in encryption is storing passwords in a way that even if the database with the passwords gets leaked, the passwords are not in a usable form. Such a system has many seemingly conflicting requirements.

- When a user logs in, it must be possible to check that they have entered the correct password.
- Even if the database is leaked, and the attacker has huge amounts of computing power...
  - The database should not give away obvious information, such as password lengths, users who chose the same passwords, letter frequencies of the passwords, or patterns in the passwords.
  - At the very least, users should have several days/ weeks to be able to change their password before the attacker cracks it. Ideally, it should not be possible for them to ever recover the passwords.
- There should be no way of recovering a forgotten password. If the user forgets their password, it must be reset. Even system administrators should not have access to a user's password.

Most login systems have a limit to how many times you can guess a password. This protects all but the poorest passwords from being guessed through a well designed login form. Suspicious login detection by checking IP address and country of origin is also becoming more common. However, none of these application enforced protections are of any use once the attacker has a copy of the database and can throw as much computational power at it as they want, without the restrictions the application enforces. This is often referred to "offline" attacking, because the attacker can attack the database in their own time.

In this section, we will look at a few widely used algorithms for secure password storage. We will then look at a couple of case studies where large databases were leaked. Secure password storage comes down to using clever encryption algorithms and techniques, and ensuring users choose effective passwords. Learning about password storage might also help you to understand the importance of choosing good passwords and not using the same password across multiple important sites.

## 8.5.1. Hashing passwords

A *hashing algorithm* is an algorithm that takes a password and performs complex computations on it and then outputs a seemingly random string of characters called a *hash*. This process is called *hashing*. Good hashing algorithms have the following properties:

- Each time a specific password is hashed, it should give the same hash.
- Given a specific **hash**, it should be impossible to efficiently compute what the original password was.

Mathematically, a hashing algorithm is called a "one way function". This just means that it is very easy to compute a hash for a given password, but trying to recover the password from a given hash can only be done by brute force. In other words, it is easy to go one way, but it is almost impossible to reverse it. A popular algorithm for hashing is called SHA-256. The remainder of this chapter will focus on SHA-256.

### Jargon Buster: What is meant by brute force?

In the Caesar Cipher section, we talked briefly about brute force attacks. Brute force attack in that context meant trying every possible key until the correct one was found.

More generally, brute force is trying every possibility until a solution is found. For hashing, this means going through a long list of possible passwords, running each through the hashing algorithm, and then checking if the outputted hash is identical to the one that we are trying to reverse.

For passwords, this is great. Instead of storing passwords in our database, we can store hashes. When a user signs up or changes their password, we simply need to put the password through the SHA-256 algorithm and then store the output hash instead of the password. When the user wants to log in, we just have to put their password through the SHA-256 algorithm again. If the output hash matches the one in the database, then the user has to have entered the correct password. If an attacker manages to access the password database, they cannot determine what the actual passwords are. The hashes themselves are not useful to the attacker.

The following interactive allows you to hash words, such as passwords (but please don't put your real password into it, as you should never enter your password on random sites). If you were to enter a well chosen password (e.g. a random string of numbers and letters),

and it was of sufficient length, you could safely put the hash on a public website, and nobody would be able to determine what your actual password was.

Use the interactive online at <http://www.csfieldguide.org.nz/en/interactives/sha2/index.html>

For example, the following database table shows four users of a fictional system, and the hashes of their passwords. You could determine their passwords by putting various possibilities through SHA-256 and checking whether or not the output is equivalent to any of the passwords in the database.

User	Password hash
coolguy	5e884898da28047151d0e56f8dc6292773603d0d6aabbdd62a11ef721d1542d8
randomuser	11c11f9a8d049409dc985973589f605ebd041b9255fe9e78e268c5f9273c41ec
what	cbd94589aae77c4eafb1620f2f731c994756e21d34b4ce8f87b30ef4d33b5ed0
hello	2cf24dba5fb0a30e26e83b2ac5b9e29e1b161e5c1fa7425e73043362938b9824

It might initially sound like we have the perfect system. But unfortunately, there is still a big problem. You can find *rainbow tables* online, which are precomputed lists of common passwords with what value they hash to. It isn't too difficult to generate rainbow tables containing all passwords up to a certain size in fact (this is one reason why using long passwords is strongly recommended!) This problem can be avoided by choosing a password that isn't a common word or combination of words.

Hashing is a good start, but we need to further improve our system so that if two users choose the same password, their hash is not the same, while still ensuring that it is possible to check whether or not a user has entered the correct password. The next idea, salting, addresses this issue.

### CURIOSITY: Passwords that hash to the same value

When we said that if the hashed password matches the one in the database, then the user has to have entered the correct password, we were not telling the full truth. Mathematically, we know that there have to be passwords which would hash to the same value. This is because the length of the output hash has a maximum length, whereas the password length (or other data being hashed) could be much larger. Therefore, there are more possible inputs than outputs, so some inputs must have the same output. When two different inputs have the same output hash, we call it a *collision*.

Currently, nobody knows of two unique passwords which hash to the same value with SHA-256. There is no known mathematical way of finding collisions, other than

hashing many values and then trying to find a pair which has the same hash. The probability of finding one in this way is believed to be in the order of 1 in a trillion trillion trillion trillion. With current computing power, nobody can come even close to this without it taking longer than the life of the sun and possibly the universe.

Some old algorithms, such as MD5 and SHA-1 were discovered to not be as immune to finding collisions as was initially thought. It is possible that there are ways of finding collisions more efficiently than by luck. Therefore, their use is now discouraged for applications where collisions could cause problems.

For password storage, collisions aren't really an issue anyway. Chances are, the password the user selected is somewhat predictable (e.g. a word out of a dictionary, with slight modifications), and an attacker is far more likely to guess the original password than one that happens to hash to the same value as it.

But hashing is used for more than just password storage. It is also used for digital signatures, which must be unique. For those applications, it is important to ensure collisions cannot be found.

## 8.5.2. Hashing passwords with a salt

A really clever technique which solves some of the problems of using a plain hash is salting. Salting simply means to attach some extra data, called *salt*, onto the end of the password and then hash the combined password and salt. Normally the salt is quite large (e.g. 128 bits). When a user tries to log in, we will need to know the salt for their password so that it can be added to the password before hashing and checking. While this initially sounds challenging, the salt should not be treated as a secret. Knowing the salt does not help the attacker to mathematically reverse the hash and recover the password. Therefore, a common practice is to store it in plaintext in the database.

So now when a user registers, a long random salt value is generated, added to the end of their password, and the combined password and salt is hashed. The plaintext salt is stored next to the hash.

## 8.5.3. The importance of good user passwords

If the passwords are salted and hashed, then a rainbow table is useless to the attacker. With current computing power and storage, it is impossible to generate rainbow tables for all common passwords with all possible salts. This slows the attacker down greatly, however they can still try and guess each password one by one. They simply need to guess passwords, add the salt to them, and then check if the hash is the one in the database.

A common brute force attack is a *dictionary attack*. This is where the attacker writes a simple program that goes through a long list of dictionary words, other common passwords, and all combinations of characters under a certain length. For each entry in the list, the program adds the salt to the entry and then hashes to see if it matches the hash they are trying to determine the password for. Good hardware can check millions, or even billions, of entries a second. Many user passwords can be recovered in less than a second using a dictionary attack.

Unfortunately for end users, many companies keep database leaks very quiet as it is a huge embarrassment that could cripple the company. Sometimes the company doesn't know its database was leaked, or has suspicions that it was but for PR reasons they choose to deny it. In the best case, they might require you to pick a new password, giving a vague excuse. For this reason, it is important to use different passwords on every site to ensure that the attacker does not break into accounts you own on other sites. There are quite possibly passwords of yours that you think nobody knows, but somewhere in the world an attacker has recovered it from a database they broke into.

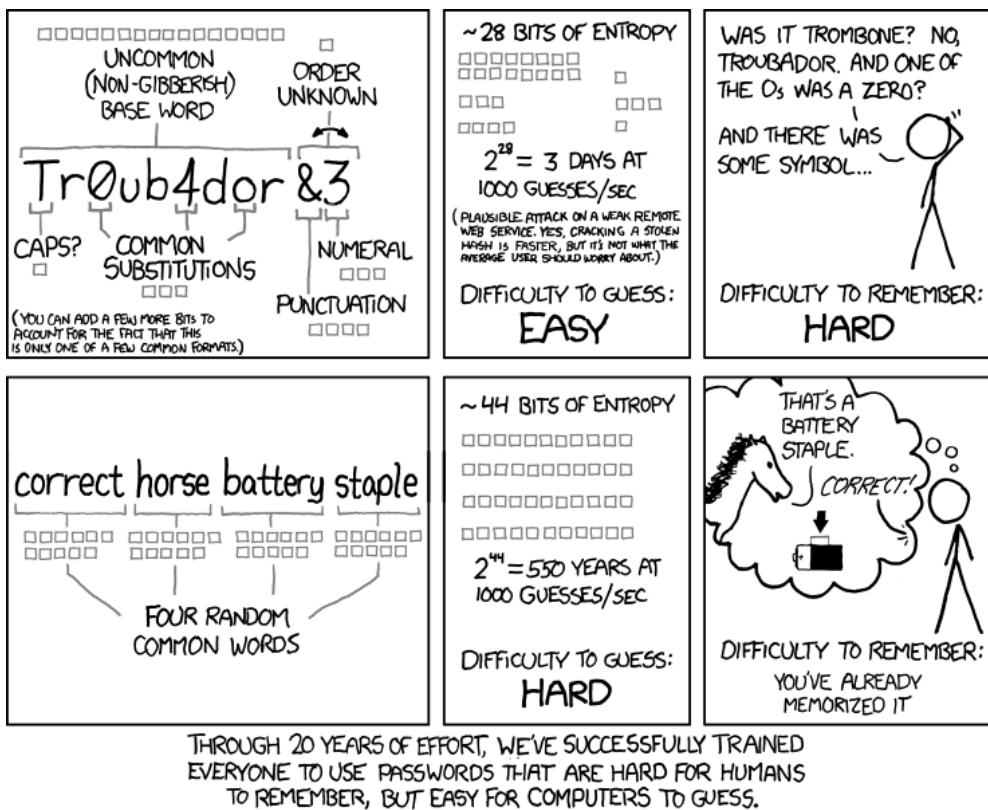
While in theory, encrypting the salts sounds like a good way to add further security, it isn't as great in practice. We couldn't use a one way hash function (as we need the salt to check the password), so instead would have to use one of the encryption methods we looked at earlier which use a secret key to unlock. This secret key would have to be accessible by the program that checks password (else it can't get the salts it needs to check passwords!), and we have to assume the attacker could get ahold of that as well. The best security against offline brute force attacks is good user passwords.

This is why websites have a minimum password length, and often require a mix of lowercase, uppercase, symbols, and numbers. There are 96 standard characters you can use in a password. 26 upper case letters, 26 lower case letters, 10 digits, and 34 symbols. If the password you choose is completely random (e.g. no words or patterns), then each character you add makes your password 96 times more difficult to guess. Between 8 and 16 characters long can provide a very high level of security, as long as the password is truly random. Ideally, this is the kind of passwords you should be using (and ensure you are using a different password for each site!).

Unfortunately though, these requirements don't work well for getting users to pick good passwords. Attackers know the tricks users use to make passwords that meet the restrictions, but can be remembered. For example, P@\$\$w0rd contains 8 characters (a commonly used minimum), and contains a mix of different types of characters. But attackers know that users like to replace S's with \$'s, mix o and 0, replace i with !, etc. In fact, they can just add these tricks into their list they use for dictionary attacks! For websites that require passwords to have at least one digit, the result is even worse. Many

users pick a standard English word and then add a single digit to the end of it. This again is easy work for a dictionary attack to crack!

As this xkcd comic points out, most password advice doesn't make a lot of sense.



[Image source](#)

You might not know what some of the words mean. In easy terms, what it is saying is that there are significantly fewer modifications of common dictionary words than there is of a random selection of four of the 2000 most common dictionary words. Note that the estimates are based on trying to guess through a login system. With a leaked database, the attacker can test billions of passwords a second rather than just a few thousand.

## 8.5.4. The whole story!

The early examples in this chapter use very weak encryption methods that were chosen to illustrate concepts, but would never be used for commercial or military systems.

There are many aspects to computer security beyond encryption. For example, access control (such as password systems and security on smart cards) is crucial to keeping a system secure. Another major problem is writing secure software that doesn't leave ways for a user to get access to information that they shouldn't (such as typing a database command into a website query and have the system accidentally run it, or overflowing the buffer with a long input, which could accidentally replace parts of the program). Also,

systems need to be protected from "denial of service" (DOS) attacks, where they get so overloaded with requests (e.g. to view a web site) that the server can't cope, and legitimate users get very slow response from the system, or it might even fail completely.

For other kinds of attacks relating to computer security, see the [Wikipedia entry on Hackers](#).

There's a dark cloud hanging over the security of all current encryption methods: [Quantum computing](#). Quantum computing is in its infancy, but if this approach to computing is successful, it has the potential to run very fast algorithms for attacking our most secure encryption systems (for example, it could be used to factorise numbers very quickly). In fact, the quantum algorithms have already been invented, but we don't know if quantum computers can be built to run them. Such computers aren't likely to appear overnight, and if they do become possible, they will also open the possibility for new encryption algorithms. This is yet another mystery in computer science where we don't know what the future holds, and where there could be major changes in the future. But we'll need very capable computer scientists around at the time to deal with these sorts of changes!

On the positive side, [quantum information transfer protocols](#) exist and are used in practice (using specialised equipment to generate quantum bits); these provide what is in theory a perfect encryption system, and don't depend on an attacker being unable to solve a particular computational problem. Because of the need for specialised equipment, they are only used in high security environments such as banking.

Of course, encryption doesn't fix all our security problems, and because we have such good encryption systems available, information thieves must turn to other approaches, especially social engineering. The easiest way to get a user's password is to ask them! A [phishing attack](#) does just that, and there are estimates that as many as 1 in 20 computer users have given out secret information this way at some stage.

Other social engineering approaches that can be used include bribing or blackmailing people who have access to a system, or simply looking for a password written on a sticky note on someone's monitor! Gaining access to someone's email account is a particularly easy way to get lots of passwords, because many "lost password" systems will send a new password to their email account.

### **Curiosity:** Steganography

Cryptography is about hiding the content of a message, but sometimes it's important to hide the *existence* of the message. Otherwise an enemy might figure out that something is being planned just because a lot more messages are being sent, even

though they can't read them. One way to achieve this is via *steganography*, where a secret message is hidden inside another message that seems innocuous. A classic scenario would be to publish a message in the public notices of a newspaper or send a letter from prison where the number of letters in each word represent a code. To a casual reader, the message might seem unimportant (and even say the opposite of the hidden one), but someone who knows the code could work it out. Messages can be hidden in digital images by making unnoticeable changes to pixels so that they store some information. You can find out [more about steganography on Wikipedia](#) or in this [lecture on steganography](#).

Two fun uses of steganography that you can try to decode yourself are a [film about ciphers that contains hidden ciphers \(called "The Thomas Beale Cipher"\)](#), and an activity that has [five-bit text codes hidden in music](#).

## 8.6. Further reading

The [Wikipedia entry on cryptography](#) has a fairly approachable entry going over the main terminology used in this chapter (and a lot more)

The encryption methods used these days rely on fairly advanced maths; for this reason books about encryption tend to either be beyond high school level, or else are about codes that aren't actually used in practice.

There are lots of intriguing stories around encryption, including its use in wartime and for spying e.g.

- How I Discovered World War II's Greatest Spy and Other Stories of Intelligence and Code (David Kahn)
- Decrypted Secrets: Methods and Maxims of Cryptology (Friedrich L. Bauer)
- Secret History: The Story of Cryptology (Craig Bauer)
- The Codebreakers: The Comprehensive History of Secret Communication from Ancient Times to the Internet (David Kahn) – this book is an older version of his new book, and may be hard to get

The following activities explore cryptographic protocols using an Unplugged approach; these methods aren't strong enough to use in practice, but provide some insight into what is possible:

- [Information hiding](#)

- [Cryptographic protocols](#)
- [Public key encryption](#)

[War in the fifth domain](#) looks at how encryption and security are key to our defence against a new kind of war.

There are lots of [articles in cs4fn on cryptography](#), including [a statistical attack that lead to a beheading](#).

The book "Hacking Secret Ciphers with Python: A beginner's guide to cryptography and computer programming with Python" (by Al Sweigart) goes over some simple ciphers including ones mentioned in this chapter, and how they can be programmed (and attacked) using Python programs.

### 8.6.1. Useful Links

- [How Stuff Works entry on Encryption](#)
- [Cryptool](#) is a free system for trying out classical and modern encryption methods. Some are beyond the scope of this chapter, but many will be useful for running demonstrations and experiments in cryptography.
- [Wikipedia entry on Cryptographic keys](#)
- [Wikipedia entry on the Caesar cipher](#)
- [Videos about modern encryption methods](#)
- [Online interactives for simple ciphers](#)

# 9. Coding - Error Control

## 9.1. What's the big picture?

Watch the video online at <https://www.youtube.com/embed/0Xz64qCjZ6k?rel=0>

The parity magic trick (in the video above) enables the magician to detect which card out of dozens has been flipped over while they weren't looking. The magic in the trick is actually computer science, using the same kind of technique that computers use to detect and correct errors in data. We will talk about how it works in the next section.

The same thing is happening to data stored on computers --- while you (or the computer) is looking away, some of it might accidentally change because of a minor fault. When the computer reads the data, you don't want it to just use the incorrect values. At least you want it to detect that something has gone wrong, and ideally it should do what the magician did, and put it right.

This chapter is about guarding against errors in data in its many different forms --- data stored on a harddrive, on a CD, on a floppy disk, on a solid state drive (such as that inside a cellphone, camera, or mp3 player), data currently in RAM (particularly on servers where the data correctness is critical), data going between the RAM and hard drive or between an external hard drive and the internal hard drive, data currently being processed in the processor or data going over a wired or wireless network such as from your computer to a server on the other side of the world. It even includes data such as the barcodes printed on products or the number on your credit card.

If we don't detect that data has been changed by some physical problem (such as small scratch on a CD, or a failing circuit in a flash drive), the information will just be used with incorrect values. A very poorly written banking system could potentially result in your bank balance being changed if just one of the bits in a number was changed by a cosmic ray affecting a value in the computer's memory! If the barcode on the packet of chips you buy from the shop is scanned incorrectly, you might be charged for shampoo instead. If you transfer a music file from your laptop to your mp3 player and a few of the bits were transferred incorrectly, the mp3 player might play annoying glitches in the music. Error

control codes guard against all these things, so that (most of the time) things just work without you having to worry about such errors.

There are several ways that data can be changed accidentally. Networks that have a lot of "noise" on them (caused by poor quality wiring, electrical interference, or interference from other networks in the case of wireless). The bits on disks are very very small, and imperfections in the surface can eventually cause some of the storage to fail. The surfaces on compact disks and DVDs are exposed, and can easily be damaged by storage (e.g. in heat or humidity) and handling (e.g. scratches or dust). Errors can also occur when numbers are typed in, such as entering a bank account number to make a payment into, or the number of a container that is being loaded onto a ship. A barcode on a product might be slightly scratched or have a black mark on it, or perhaps the package is bent or is unable to be read properly due to the scanner being waved too fast over it. Bits getting changed on permanent storage (such as hard drives, optical disks, and solid state drives) is sometimes referred to as data rot, and the [wikipedia page on bit rot](#) has a list of more ways that these errors can occur.

Nobody wants a computer that is unreliable and won't do what it's supposed to do because of bits being changed! So, how can we deal with these problems?

Error control coding is concerned with detecting when these errors occur, and if practical and possible, correcting the data to what it is supposed to be.

Some error control schemes have error correction built into them, such as the parity method that was briefly introduced at the beginning of this section. You might not understand yet how the parity trick worked, but after the card was flipped, the magician detected which card was flipped, and was able to correct it.

Other error control schemes, such as those that deal with sending data from a server overseas to your computer, send the data in very small pieces called packets (the network protocols chapter talks about this) and each packet has error detection information added to it.

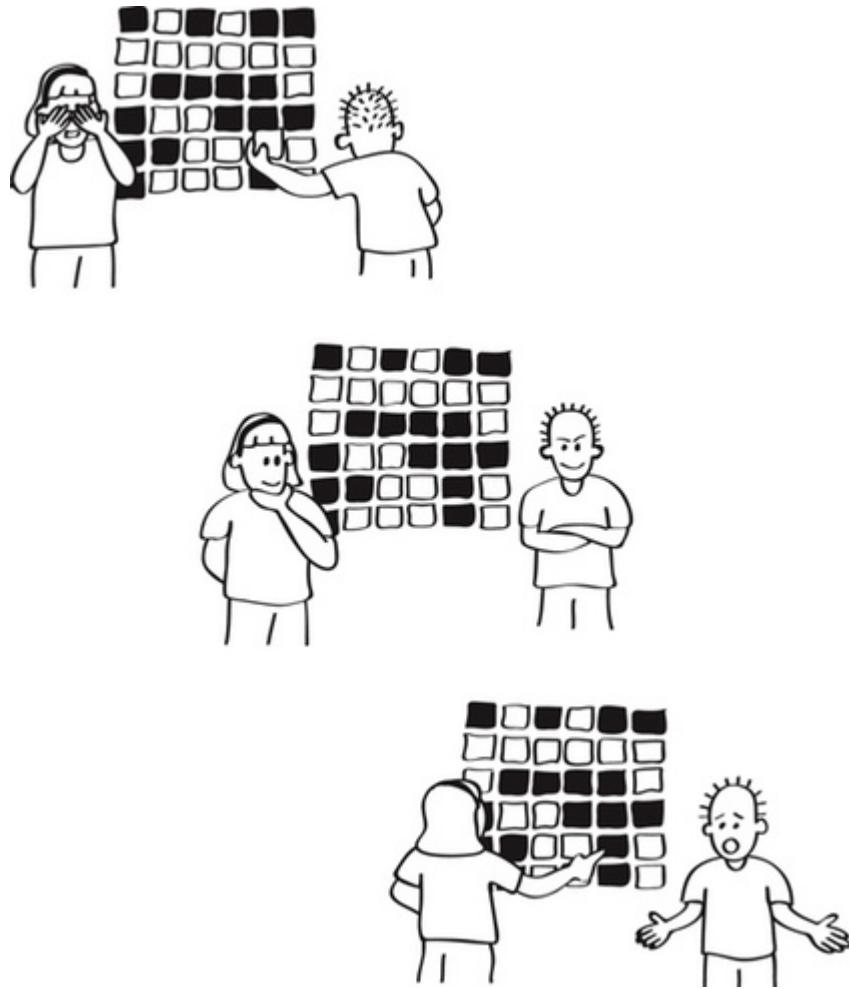
Error detection is also used on barcode numbers on products you buy, as well as the unique ISBN (International Standard Book Number) that all books have, and even the 16 digit number on a credit card. If any of these numbers are typed or scanned incorrectly, there's a good chance that the error will be detected, and the user can be asked to re-enter the data.

By the end of this chapter, you should understand the basic idea of error control coding, the reasons that we require it, the differences between algorithms that can detect errors and those that can both detect and correct errors, and some of the ways that error control

coding is used, in particular parity (focussing on the parity magic trick) and the check digits used to ensure book numbers, barcode numbers, and credit card numbers are entered correctly.

## 9.2. The Parity Magic Trick

If you have never seen the parity magic trick before, check out the video in the “What’s the Big Picture?” section above. This section assumes that you know what is meant by the parity magic trick, but now we’ll explain how it actually works!



A magician asks an observer to lay out a square grid of two-sided cards, and the magician then says they are going to make it a bit harder, and add an extra row and column to the square. The magician then faces the other way while the observer flips over one card. The magician turns back around again, and tells the observer which card was flipped!

The question now is, how did the magician know which card had been flipped without seeing the card being flipped, or memorising the layout?! The short answer is error control coding. Let's look more closely at that...

## 9.2.1. Carrying out the parity trick

In the interactive, the computer has a 7x7 grid of black and white cards. You must choose the colour of an extra card for each row (at the right) and column (at the bottom), making an 8x8 grid of cards. Each extra card should be chosen so that each row and column has an even number of black cards (since there are 8 cards, there will also be an even number of white cards). The bottom right-hand card can be chosen from either its row or column; they should both give the same colour.

Once you think you have this correct, you should tell the computer to flip a card. An animation will appear for a few seconds, and then the cards will reappear with one card flipped (all the rest will be the same as before). Your task is to identify the flipped card. You should be able to do this *without* having memorised the layout. Remember the pattern you made with the extra cards you added? That's the key to figuring it out. Once you think you have identified the card, click it to see whether or not you were right. The interactive will guide you through these instructions. If you are completely stuck identifying the flipped card, a hint follows the interactive, although you should try and figure it out for yourself first. Make sure you add the extra cards correctly; the computer won't tell you if you get them wrong, and you probably won't be able to identify the flipped card if the extra cards aren't chosen correctly.

Use the interactive online at <http://www.csfieldguide.org.nz/en/interactives/parity/index.html>

Remember how you made it so that each column had an even number of black cards? When a card is flipped, this results in the row and column that the card was in having an odd number of black cards. So all you need to do is to identify the row and column that have an odd number of black and white cards, and the card that is at the intersection of them must be the one that was flipped!

What we saw above is a simple error control coding algorithm, known as *2-dimensional parity*.

## 9.2.2. How does the parity trick relate to error control coding?

The cards represent bits, with their two states being black and white (in the "data representation" chapter we looked at how a bit can be stored by anything that can be in one of two states: shiny/not shiny, magnetised/not magnetised, high voltage/low voltage, black/white, etc). The original 7x7 cards that the computer laid out for you could be some kind of data, for example some text represented using bits, or an image, or some numbers.

Although they are laid out in a grid, on a computer the rows of bits would be stored or transmitted one after the other (as 8 lots of 8 bits).

The extra cards you added are called *parity bits*. **Parity** simply means whether a number is even or odd (the word comes from the same root as "pair"). By adding the extra cards in a way that ensured an even number of black cards in each row and column, you made it so that the rows and columns had what is called *even parity*.

When a card was flipped, this simulated an error being made in your data (such as a piece of dust landing on a bit stored on a CD, or a cosmic ray changing a bit stored on a hard disk, or electrical interference changing a bit being sent over a network cable). Because you knew that each row and column was supposed to have an even number of black and white cards in it, you could tell that there was an error from the fact that there was a column and row that had an odd number of black cards in it. This means that the algorithm is able to detect errors, i.e. it has **error detection**. The specific card that had been flipped was at the intersection of the row and column that had an odd number of black cards and white cards in them, and because you were able to identify exactly which card was flipped, you were able to correct the error, i.e the algorithm has **error correction**.

If you had not added the parity bits, you would have had no way of even knowing an error had occurred, unless you had memorised the entire layout of cards! And what if more than one bit had been flipped? We'll consider this later.

### Project: Being a magician or using the parity trick as a party trick!

Now that you have learnt how the parity trick works, you might like to try it with a physical set of cards like the busker in the video, or you could use any objects with two distinct sides, such as coins or cups. You could use playing cards, but the markings can be distracting, and cards with two colours are easiest (you can make them by cutting up sheets of card with the two colours on, or single coloured card with a scribble or sticker on one side).

You can find details and lots of ideas relating to the trick [here](#), or follow these instructions:

1. Ask a friend to lay out 25 cards in a 5 by 5 grid, trying to have a reasonably random mix of blacks and whites (this is smaller than the one in the interactive, but it is easier to have fewer cards to avoid errors in the next step!)
2. Take all the remaining cards, and then say that actually, 5 by 5 is too easy so you are going to make it 6 by 6. Instead of adding the new row and column randomly though, you are adding them in the way you did in the interactive (even parity).

Do this as fast as you can without making errors (it can look very casual if you practise this, even though the cards are being carefully selected).

3. Tell your friend that you are going to face the other way, and you want them to flip over one card while you are not looking. Check that they've flipped exactly one card.
4. Turn around again once they have flipped a card, look through the rows and columns, identifying a row and then a column that has an odd number of black cards in it. The flipped card will be the one at the intersection of that row and column. Flip that card back over.

It would take some practice to be able to add the extra cards, and identify the flipped card without the observer noticing that you are thinking hard about it. With practice you should be able to do it while having a casual conversation. Once you master it, you've got a great trick for parties, or even for busking.

To make it more showy, you can pretend that you are mind reading the person, waving your hands over the cards. A particular impressive variation is to have an assistant come in to the room after the card has been flipped; even though they haven't seen any of the setup, they will still be able to detect the error.

### 9.2.3. Analysing the parity trick

At this point, you should be able to carry out the parity trick well enough that you can demonstrate that you understand how to do it. The remainder of this section is focussed on exploring further ideas in error control coding related to the parity trick.

It would be ideal to have some physical parity cards at this point that you can layout in front of you and play around with to explore the questions raised.

An error control coding algorithm can often detect errors more easily than it can correct them. Errors involving multiple bits can sometimes even go undetected. What if the computer (or your friend if you were being a magician with actual parity cards) had been sneaky and turned over two cards instead of one? You could start by getting a friend or classmate to actually do this. Repeat it a few times. Are you always able to correct the errors, or do you get it wrong?

Remember that to *detect* errors using this algorithm, you know that if one or more rows and/or columns has an odd number of blacks and whites in it, that there must be at least one error. In order to *correct* errors you have to be able to pinpoint the specific card(s) that were flipped.

Are you always able to detect when an error has occurred if 2 cards have been flipped? Why? Are you ever able to correct the error? What about with 3 cards?

It turns out that you can always detect an error when 2 cards have been flipped (i.e. a 2-bit error), but the system can't correct more than a 1-bit error. When two cards are flipped, there will be at least two choices for flipping two cards to make the parity correct, and you won't know which is the correct one. With a 3-bit error (3 cards flipped), it will always be possible to detect that there is an error (an odd number of black bits in at least one row or column), but again, correction isn't possible. With 4 cards being flipped, it's possible (but not likely) that an error can go undetected.

There is actually a way to flip 4 cards where the error is then *undetected* meaning that the algorithm will be unable to detect the error. Can you find a way of doing this?

With more parity cards, we can detect and possibly correct more errors. Let's explore a very simple system with minimal parity cards. We can have a 7x7 grid of data with just one parity card. That parity card makes it so that there is an even number of black cards in the entire layout (including the parity card). How can you use this to detect errors? Are you ever able to correct errors in this system? In what situations do errors go undetected (think when you have multiple errors, i.e. more than one card flipped).

With only one extra card for parity checking, a single bit error can be detected (the total number of black cards will become odd), but a 2-bit error won't be detected because the number of black cards will be even again. A 3-bit error will be detected, but in general the system isn't very reliable.

So going back to the actual parity trick that has the 7 by 7 grid, and 15 parity cards to make it 8 by 8, it is interesting to note that only 1 extra card was needed to detect that an error had occurred, but an extra 15 cards were needed to be able to correct the error. In terms of the cost of an algorithm, it costs a lot more space to be able to correct errors than it does to be able to simply detect them!

What happens when you use grids of different sizes? The grid doesn't have to have an even number of black cards *and* an even number of white cards, it just happens that whenever you have an even number sized grid with the parity bits added (e.g. the 8x8 we have mostly used in this section) and you have an even number of black cards, you will also have to have an even number of whites, which makes it a bit easier to keep track of.

Try a 6x6 grid with parity cards to make it 7x7. The parity cards simply need to make each row and column have an even number of black cards (in this case there will always be an odd number of white cards in each row and column). The error detection is then looking for rows and columns that have an odd number of black cards in them (but an even number of

white cards). Interestingly, the grid doesn't even have to be a square! You could use 4x7 and it would work!

There's also no limit on the size. You could create a 10x10 grid (100 cards), and still be able to detect which card has been flipped over. Larger grids make for an even more impressive magic trick.

## 9.3. Check digits on barcodes and other numbers

You probably wouldn't be very happy if you bought a book online by entering the ISBN (International Standard Book Number), and the wrong book was sent to you, or if a few days after you ordered it, you got an email saying that the credit card number you entered was not yours, but was instead one that was one digit different and another credit card holder had complained about a false charge. Or if you went to the shop to buy a can of drink and the scanner read it as being a more expensive product. Sometimes, the scanner won't even read the barcode at all, and the checkout operator has to manually enter the number into the computer --- but if they don't enter it exactly as it is on the barcode you could end up being charged for the wrong product. These are all examples of situations that error control coding can help prevent.

Barcode numbers, credit card numbers, bank account numbers, ISBNs, national health and social security numbers, shipping labels (serial shipping container codes, or SSCC) and tax numbers all have error control coding in them to help reduce the chance of errors. The last digit in each of these numbers is a check digit, which is obtained doing a special calculation on all the other digits in the number. If for example you enter your credit card number into a web form to buy something, it will calculate what the 16th digit should be, using the first 15 digits and the special calculation (there are 16 digits in a credit card number). If the 16th digit that it expected is not the one you entered, it can tell that there was an error made when the number was entered and will notify you that the credit card number is not valid.

In this section we will be initially looking at one of the most commonly used barcode number formats used on most products you buy from supermarkets and other shops. We will then be having a look at credit card numbers. You don't have to understand *why* the calculations work so well (this is advanced math, and isn't important for understanding the overall ideas), and while it is good for you to know what the calculation is, it is not essential. So if math is challenging and worrying for you, don't panic too much because what we are looking at in this section isn't near as difficult as it might initially appear!

### 9.3.1. Check digits On product barcodes

Most products you can buy at the shop have a barcode on them with a 13 digit "global trade item number" (referred to as GTIN-13). The first 12 digits are the actual identification number for the product, the 13th is the check digit calculated from the other 12. Not all barcodes are GTIN-13, there are several other types around. However, if the barcode has 13 numbers in it, it is almost certainly GTIN-13.



The last digit of these numbers is calculated from the first 12. This is very closely related to the parity bit that we looked at above, where the last bit of a row is "calculated" from the preceding ones. With a GTIN-13 code, we want to be able to detect if one of the digits might have been entered incorrectly.

The following interactive checks GTIN-13 barcodes. Enter the first 12 digits of a barcode number into the interactive, and it will tell you that the last digit should be! You could start by using the barcode number "9 300675 036009".

Use the interactive online at <http://www.csfieldguide.org.nz/en/interactives/checksum-calculator-gtin-13/index.html>

What happens if you make a mistake when you type in the 12 digits (try changing one digit)? Does that enable you to detect that a mistake was made?

Have a look for another product that has a barcode on it, such as a food item from your lunch, or a stationery item. Note that some barcodes are a little different --- make sure the barcodes that you are using have 13 digits (although you might like to go and find out how the check digit works on some of the other ones). Can the interactive always determine whether or not you typed the barcode correctly?

One of the following product numbers has one incorrect digit. Can you tell which of the products has had its number typed incorrectly?

- 9 400550 619775

- 9 400559 001014
- 9 300617 013199

If you were scanning the above barcodes in a supermarket, the incorrect one will need to be rescanned, and the system can tell that it's a wrong number without even having to look it up. Typically that would be caused by the bar code itself being damaged (e.g. some ice on a frozen product making it read incorrectly). If an error is detected, the scanner will usually make a warning sound to alert the operator.

You could try swapping barcode numbers with a classmate, but before giving them the number toss a coin, and if it's heads, change one digit of the barcode before you give it to them. Can they determine that they've been given an erroneous barcode?

If one of the digits is incorrect, this calculation for the check digit will produce a different value to the checksum, and signals an error. So single digit errors will *always* be detected, but what if two digits change --- will that always detect the error?

What if the error is in the checksum itself but not in the other digits - will that be detected?

### **9.3.2. How do check digits protect against common human errors?**

People can make mistakes when they enter numbers into computers, and even barcode scanners can get a digit wrong. Check digits attempt to detect when an error has occurred and notify the computer and/or person of it. Suppose you give your cellphone number to a friend so that they can contact you later. To ensure that you told them the number correctly, you may get them to text you immediately to confirm (and so that you have their number too). If you don't get the text you will probably double check the number and will find that your friend made an error, for example they got a digit wrong or they reversed 2 digits next to one another. Mistakes happen, and good systems prevent those mistakes from having annoying or even serious consequences. If a check digit is being used, there is a good chance that the error will be detected when the check digit is not what the computer expects it to be.

Some of the really common errors are:

- Getting one digit wrong (substitution)
- Swapping two digits that are adjacent (transposition)
- Missing a digit
- Adding a digit

The last two will be picked up from the expected length of the number; for example,a GTIN-13 has 13 digits, so if 12 or 14 were entered, the computer immediately knows this is

not right. The first two depend on the check digit in order to be detected. Interestingly, all one digit errors will be detected by common checksum systems, and *most* transpositions will be detected (can you find examples of transpositions that aren't detected, using the interactive above?)

There are also some less common errors that people make

- Getting a digit wrong in two or more different places
- Doubling the wrong digit, e.g. putting 3481120 instead of 3481220
- Muddling 3 digits, e.g. 14829 instead of 12489
- Phonetic errors are possible when the number was read and typed by somebody listening (or reading the number to themselves as they type it). For example, "three-forty" (340) might be heard as "three-fourteen" (314), and numbers like 5 and 9 can sound similar on a bad phone line.

Experiment further with the interactive. What errors are picked up? What errors can you find that are not? Are the really common errors nearly always picked up? Can you find any situations that they are not? Try to find examples of errors that are detected and errors that are not for as many of the different types of errors as you can.

### 9.3.3. How is the check digit on product barcodes calculated?

The first 12 digits of the barcode represent information such as the country origin, manufacturer, and an identifier for the product. The 13th digit is a check digit, it is calculated from the first 12 digits. It is calculated using the following algorithm (also, see the example below).

- Multiply every second digit (starting with the second digit) by 3, and every other digit by 1 (so they stay the same).
- Add up all the multiplied numbers to obtain the *sum*.
- The check digit is whatever number would have to be added to the sum in order to bring it up to a multiple of 10 (i.e. the last digit of the sum should be 0). Or more formally, take the last digit of the sum and if it is 0, the check digit is 0. Otherwise, subtract the last digit from 10 to obtain the check digit.

Let's look at an example to illustrate this algorithm. We want to confirm that the check digit that was put on the barcode of a bottle of cola was the correct one. Its barcode number is 9300675032247. The last digit, 7, is the check digit. So we take the first 12 digits and multiply them by 1 or 3, depending on their positions ( $9 \times 1 + 3 \times 3 + 0 \times 1 + 0 \times 3 + 6 \times 1 + 7 \times 3 + 5 \times 1 + 0 \times 3 + 3 \times 1 + 2 \times 3 + 2 \times 1 + 4 \times 3$ ). We then add up all the multiplied numbers, obtaining a sum of

73. We want to add the check digit that will bring the sum up to the nearest multiple of 10, which is 80. This number is 7, which is indeed the check digit on the cola bottle's barcode.

The following interactive can be used to do the calculations for you. To make sure you understand the process, you need to do some of the steps yourself; this interactive can be used for a wide range of check digit systems. To check a GTIN-13 number, enter the first 12 digits where it says "Enter the number here". The multipliers for GTIN-13 can be entered as "131313131313" (each alternating digit multiplied by 3). This will give you the products of each of the 12 digits being multiplied by the corresponding number. You should calculate the total of the products (the numbers in the boxes) yourself and type it in. Then get the remainder when divided by 10. To work out the checksum, you should calculate the digit needed to make this number up to 10 (for example, if the remainder is 8, the check digit is 2). If the remainder is 0, then the check digit is also 0.

Use the interactive online at <http://www.csfieldguide.org.nz/en/interactives/checksum-calculator/index.html>

Try this with some other bar codes. Now observe what happens to the calculation when a digit is changed, or two are swapped.

The algorithm to check whether or not a barcode number was correctly entered is very similar. You could just calculate the check digit and compare it with the 13th digit, but a simpler way is to enter all 13 digits.

Multiply every second digit (starting with the second digit) by 3, and every other digit by 1. This includes the 13th digit, which is multiplied by 1. Add up all the multiplied numbers to obtain the *total*. If the last digit of the sum is a 0, the number was entered correctly. (That's the same as the remainder when divided by 10 being 0).

### **Curiosity:** Working out a checksum in your head

For 13-digit barcodes, a quick way to add up a checksum that can be done in your head (with some practice) is to separate the numbers to be multiplied by 3, add them up, and then multiply by 3. For the example above (9300675032247) the two groups are  $9+0+6+5+3+2+7 = 32$  and  $3+0+7+0+2+4 = 16$ . So we add  $32 + 16 \times 3$ , which gives the total of 80 including the check digit.

To make it even easier, for each of the additions you only need to note the last digit, as the other digits will never affect the final result. For example, the first addition above begins with  $9+0+6$ , so you can say that it adds up to 5 (rather than 15) and still get the same result. The next digit (5) takes the sum to 0, and so on. This also means that you can group digits that add to 10 (like 1 and 9, or 5 and 5), and ignore them. For

example, in the second group,  $3+0+7$  at the start adds up to 0, and the only sum that counts is  $2+4$ , giving 6 as the total.

Finally, even the multiplication will be ok if you just take the last digit. In the example above, that means we end up working out  $6 \times 3$  giving 8 (not 18); the original was  $16 \times 3$  giving 48, but it's only the final 8 digit that matters.

All these shortcuts can make it very easy to track the sum in your head.

### **Extra For Experts:** Why does this algorithm work so well?

In order to be effective, the algorithm needs to ensure the multiplied digits will not add up to a multiple of 10 any more if the digits are changed slightly. The choice of multipliers affects how likely it is to detect small changes in the input. It's possible to analyse these mathematically to work out what sorts of errors can be detected.

The check digit on barcodes is described in the chapter on [error control coding](#). Basically every second digit is multiplied by 3, and the sum of these multiples are added to the remaining digits.

Let's look at some smaller examples with 5 digits (4 normal digits and a check digit), as the same ideas will apply to the 13 digit numbers.

If we need a check digit for 8954, we would calculate  $(8 \times 1) + (9 \times 3) + (5 \times 1) + (4 \times 3) = 52$ , and in order to bring this up to 60, we need to add 8. This makes the full number 89548.

The first thing we should observe is that only the ones column (last digit) of each number added have any impact on the check digit.  $8+27+5+12=52$ , and  $8+7+5+2=22$  (only looking at the last digit of each number we are adding). Both these end in a 2, and therefore need 8 to bring them up to the nearest multiple of 10. You might be able to see why this is if you consider that the "2" and "1" were cut from the tens column, they are equal to  $10+20=30$ , a multiple of 10. Subtracting them only affects the tens column and beyond. This is always the case, and therefore we can simplify the problem by only adding the ones column of each number to the sum. (This can also be used as a shortcut to calculate the checksum in your head).

#### *Protection against single digit errors*

Next, let's look at why changing one digit in the number to another digit will *always* be detected with this algorithm. Each digit will contribute a number between 0 and 9 to

the sum (remember we only care about the ones column now). As long as changing the digit will result in it contributing a different amount to the sum, it becomes impossible for it to still sum to a multiple of 10. Remember that each digit is either multiplied by 1 or 3 before its ones column is added to the sum.

A number being multiplied by 1 will always contribute itself to the sum. If for example the digit was supposed to be 9, no other single digit can contribute 9 to the sum. So those multiplied by 1 are fine.

But what about those multiplied by 3? To answer that, we need to look at what each different digit contributes to the sum when multiplied by 3.

- 1 -> 3
- 2 -> 6
- 3 -> 9
- 4 -> 2 (from 12)
- 5 -> 5 (from 15)
- 6 -> 8 (from 18)
- 7 -> 1 (from 21)
- 8 -> 4 (from 24)
- 9 -> 7 (from 27)
- 0 -> 0

If you look at the right hand column, you should see that no number is repeated twice. This means that no digit contributes the same amount to the sum when it is multiplied by 3!

Therefore, we know that all single digit errors will be detected.

#### *Protection against swapping adjacent digit errors*

Seeing why the algorithm is able to protect against most swap errors is much more challenging.

If two digits are next to one another, one of them must be multiplied by 1, and the other by 3. If they are swapped, then this is reversed. For example, if the number 89548 from earlier is changed to 85948, then  $(5 \times 3) + (9 \times 1) = 24$  is being added to the total instead of  $(9 \times 3) + (5 \times 1) = 32$ . Because 24 and 32 have different values in their ones columns, the amount contributed to the total is different, and therefore the error will be detected.

But are there any cases where the totals will have the same values in their ones columns? Another way of looking at the problem is to take a pair of rows from the table above, for example:

- 8 -> 4
- 2 -> 6

Remember that the first column is how much will be contributed to the total for digits being multiplied by 1, and the second column is for those being multiplied by 3. Because adjacent digits are each multiplied by a different amount (one by 3 and the other by 1), the numbers diagonal to each other in the chosen pair will be added.

If for example the first 2 digits in a number are “28”, then we will add  $2+4=6$  to the sum. If they are then reversed, we will add  $8+6=14$ , which is equivalent to 4 as again, the “10” part does not affect the sum. 8+6 and 2+4 are the diagonals of the pair!

- 8 -> 4
- 2 -> 6

So the question now is, can you see any pairs where the diagonals would add up to the same value? There is one!

#### *Protection against twin errors*

A twin error is where a digit that is repeated twice in a number is changed to a different digit that is repeated twice. For example, if we have “22” in the number, somebody might somehow change it to “88”.

When two numbers are side by side, one is multiplied by 3 and the other by 1. So the amount contributed to the total is the sum of the number’s row in the above table. For example, 2 has the row “2->6”. This means that  $2+6=8$  will be contributed to the sum as a result of these two digits.

If any rows add up to the same number, this could be a problem. Where the sum was over 10, the tens column has been removed.

- 1 -> 3 adds to “4”
- 2 -> 6 adds to “8”
- 3 -> 9 adds to “2”
- 4 -> 2 adds to “6”
- 5 -> 5 adds to “0”
- 6 -> 8 adds to “4”
- 7 -> 1 adds to “8”

- 8 → 4 adds to “2”
- 9 → 7 adds to “6”
- 0 → 0 adds to “0”

Some of the rows add up to the same number! Because both 4 and 9 add up to 6, the error will not be detected if “44” changes to “99” in a number!

Rows that do not add will be detected. From the example above, if 22 changes to 88, this will be detected because 22’s total is 8, and 88’s total is 2.

*An error that is never detected*

Another error that people sometimes make is the jump transposition error. This is where two digits that have one digit in between them are swapped, for example, 812 to 218.

A pair of numbers that are two apart like this will always be multiplied by the same amount as each other, either 1 or 3. This means that the change in position of the numbers does not affect what they are multiplied by, and therefore what they contribute to the sum. So this kind of error will never be detected.

### Project: Project with check sums

The following interactive will generate random numbers of a chosen type (e.g. ISBN numbers for books). These numbers are random, and are not based on numbers for actual books (or bank accounts!) This means that you can do this project without having to ask people for their personal information such as credit card numbers (in fact, they shouldn’t give you this information anyway!)

Although the numbers from this interactive are random, their check digits are calculated using the appropriate method, so you can use them as examples for your project. Actually, not all of them will be correct, so one of your challenges is to figure out which are ok!

Use the interactive online at <http://www.csfieldguide.org.nz/en/interactives/number-generator/index.html>

Your project is to choose a checksum other than the 13-digit barcodes, and research how it is calculated (they all use slightly different multipliers). You should

demonstrate how they work (using the following interactive if you want), and find out which of the numbers generated are incorrect.

ISBN-10 is particularly effective, and you could also look into why that is.

Use the interactive online at <http://www.csfieldguide.org.nz/en/interactives/checksum-calculator/index.html>

### 9.3.4. The whole story!

The codes discussed in this chapter are all widely used, but the most widely used codes for data storage are more sophisticated because they need to deal with more complex errors than a single bit changing. For example, if a CD is scratched or a hard disk has a small fault, it's likely to affect many adjacent bits. These systems use codes based on more advanced mathematical concepts. The most widely used codes for storage and data transfer are [the Reed-Solomon codes](#) and [Cyclic Redundancy Check \(CRC\)](#). For human readable numbers such as bar codes, bank numbers, tax numbers, social security numbers and so on, [checksums](#) are very common, and the [Luhn algorithm](#) is one of the more widely used. Larger checksums are also used to check that downloaded files are correct. The parity method is a form of [Hamming code](#).

## 9.4. Further reading

### 9.4.1. Useful Links

- [CS Unplugged Parity Trick](#)
- CS4FN has a [free book](#) that contains the Parity Trick and a number of other tricks related to computer science.
- Techradar has more [information about error detection and correction](#)
- [An explanation of error correcting codes](#)
- [A check digit calculator for common bar codes](#)

# 10. Artificial Intelligence

Watch the video online at <https://www.youtube.com/watch?v=ia-oYtacJHE>

## 10.1. What's the big picture?

Artificial Intelligence conjures up all sorts of images --- perhaps you think of friendly systems that can talk to you and solve tough problems; or maniac robots that are bent on world domination? There's the promise of driverless cars that are safer than human drivers, and the worry of medical advice systems that hold people's lives in their virtual hands. The field of Artificial Intelligence is a part of computer science that has a lot of promise and also raises a lot of concerns. It can be used to make decisions in systems as large as an aeroplane or an [autonomous dump truck](#), or as small as a mobile phone that accurately predicts text being typed into it. What they have in common is that they try to mimic aspects of human intelligence. And importantly, such systems can be of significant help in people's everyday lives.

AI (also known as intelligent systems) is primarily a branch of computer science but it has borrowed a lot of concepts and ideas from other fields, especially [mathematics](#) (particularly logic, combinatorics, statistics, probability and optimisation theory), [biology](#), [psychology](#), [linguistics](#), [neuroscience](#) and [philosophy](#).

In this chapter we'll explore a range of these intelligent systems. Inevitably this will mean dealing with ethical and philosophical issues too --- do we really want machines to take over some of our jobs? Can we trust them? Might it all go too far one day? What do we really mean by a computer being intelligent? While we won't address these questions directly in this chapter, gaining some technical knowledge about AI will enable you to make more informed decisions about the deeper issues.

## 10.2. Chatterbots and The Turing Test



Many humans take for granted the fact that they can easily have a conversation with another person, and choose appropriate things to say based on the conversation. The ability to do this is a form of intelligence, and for computers it isn't so easy! Many attempts have been made to design computer programs that can have a conversation with a human and sound intelligent. These computer programs are called *chatterbots*, or just *chatbots*.

You may come across chatterbots online for serious uses (such as giving help on a booking website). Sometimes it's hard to tell if you're getting an automated response. We'll start off by looking at some very simple chatterbots that are only designed as an experiment rather than to offer serious advice.

### 10.2.1. A Therapy Session with Eliza the Chatterbot

One such chatterbot is **Eliza**. Eliza attempts to sound like a Rogerian Psychotherapist, and sounds most intelligent in the kind of conversation that would be had in such a therapy session. ([Rogerian Psychotherapy](#) is a technique that revolves around trying to understand the client by asking questions and reflecting their statements back to clarify that they've been understood).

**Caution:** Chatbots and personal information!

Eliza is a system that was intended to get people thinking about AI, and you should not use it for your own therapeutic usage. You should never enter personal information into a computer that you wouldn't want anybody else reading, as you can never be certain that the web site isn't going to store the information or pass it on to someone. So you don't want to be telling Eliza the kinds of things a person would say in a therapy session --- just make stuff up when talking to Eliza! For the same reason,

do not tell any chatterbot other personal information such as your full name, date of birth, or address.

Go to the link just below and have a conversation with Eliza (keeping in mind that she is supposed to be a Rogerian Psychotherapist, so will respond like one).

[Click here to use the Eliza chatterbot](#)



Do you think Eliza is as smart as a human? Would you believe she was a human if somebody told you she was?

You will probably agree that while Eliza sounds somewhat intelligent at times, she gives very vague and general replies, forgets earlier parts of the conversation, and says some things that sound very strange! As an example, here is a conversation with Eliza:

**Eliza:** Hello, I am Eliza.

**Human:** Hello I am a human

**Eliza:** Do you believe it is normal to be a human?

**Human:** Yes

**Eliza:** Are you Sure?

**Human:** Yes

**Eliza:** Why did you repeat yourself?

Eliza often gives very general replies by looking for suitable rules based only on the line you've just typed, and that often leads to some very weird conversations, such as the one above! In this case Eliza used pattern matching to determine that it might be appropriate to ask me "Do you believe it is normal to be a human?", as I had said I was a human; if I had said "I am a banana" then the same rule would have responded with "Do you believe it is

normal to be a banana?". When I answered "Yes", Eliza didn't actually understand the *meaning* of me saying "Yes", so simply asked me if I was sure, as it has a general response that says a good way of responding to somebody saying "Yes" is to ask them if they were sure. Eliza also has rules that are triggered if somebody repeats themselves that say she should ask them why. Because Eliza doesn't actually understand meanings but instead just follows rules that say what might be appropriate responses, she had no way of knowing that it actually *was* meaningful for me to respond with "Yes" twice! This is a big giveaway that Eliza really isn't that intelligent, and doesn't actually understand meanings in conversation, or even whether or not a response to her own question is actually reasonable.

Now go back and have another conversation with Eliza. There are many other examples of Eliza saying stuff that makes no sense such as the above dialogue. How many can you find? In addition, how does Eliza respond when you do the following things?

- Try being a "parrot" that just copies everything Eliza says.
- What happens when you don't give meaningful answers to her questions?
- If you say the same thing you said earlier in the conversation, does Eliza always respond in the same way? (When you say it immediately after, she probably won't, as she'll comment on the fact that you repeated yourself the second time!)
- What happens when you talk about things that are unrelated to what would be said in a therapy session, i.e. you try to have a general conversation with Eliza (remember that Eliza works in a restricted domain, i.e. she assumes she is a therapist).

## 10.2.2. Alice the Chatterbot

We saw above that Eliza is a chatterbot who works with a restricted domain (trying to take the role of a therapist), and tried to sound intelligent by giving very general responses. As you probably found out though, this quickly reveals that Eliza really isn't intelligent at all, and will just repeat the same things over and over again. In addition, you probably also noticed that Eliza frequently doesn't make a lot of sense, and comes up with sentences that make no sense to a human.

Try having a conversation with another chatterbot --- Alice. Note that she knows you as "judge". You might work out why she calls you this once you have read the sections on the Turing Test below!

[Click here for the Alice chatterbot at pandorabots.com](http://pandorabots.com)

**A.L.I.C.E. and Fred**

**You said:** Hi Alice  
**A.L.I.C.E.:** Hi there!

You say:

**Say**

Does Alice sound more intelligent than Eliza? Does she sound as intelligent as a human, or can you trick her into saying things that make no sense? Try using the same tricks you used to fool Eliza and see if they fool Alice as well (look back at the list of bullet points in the Eliza section just above).

Like Eliza, Alice also uses hard coded rules to know what to say, although she has a lot more of these rules and a more sophisticated algorithm for choosing which rules should be used to determine what to say. If you want to know more, just ask Alice how she works, and she'll tell you!

Some possible rules that Alice might have are:

- If "favourite colours" are mentioned, say your favourite colour is "green".
- If family members are mentioned (e.g. "mum", "dad", or "brother") say something that is about family.

To use the rules, the chatterbot analyses your last sentence (and possibly earlier sentences also) by matching the sentence against the patterns in its rules. If the sentence matches the rule pattern, it "activates" the rule. The rule will specify a reply. Sometimes the reply is just a template for a sentence, and the chatterbot analyses the text you typed to extract words that can be inserted into the template. That's why the replies sometimes use part of what you typed.

The quality of these rules determines how intelligent the chatterbot will sound.

While Alice is undoubtedly more intelligent than Eliza, Alice still has her limitations. Sometimes she even admits to those limitations!

**Judge:** What will happen in the year 2000

**Alice:** I cannot predict the future.

**Judge:** That is the past

**Alice:** Yes it is one of the past.

**Judge:** Then why did you say you can't predict the future?

**Alice:** It was a preprogrammed response completely determined by your statements.

I suppose I was a bit mean for tricking her and saying "what will". Her rules assume a human wouldn't mislead her like that.

### 10.2.3. Chatterbots that learn from humans

We saw in the previous sections examples of two chatterbots that have rules that help them determine suitable things to say. One of those chatterbots only works in a restricted domain, taking the role of a therapist, and is unable to have very general conversations, and the other is able to have more general conversations. Both these chatterbots had their *rules* of what to say determined by programmers *at the time of programming*, and these rules will never be changed unless a programmer decides to change them.

There are other chatterbots that are able to *learn* their rules from the humans they have conversations with. By looking at how a human responds to various dialogues, the chatterbot attempts to learn how it should respond in various situations. The idea is that if it responds in similar ways to what a human does, then perhaps it will sound like a human. Most of these chatterbots aim to have very general conversations, i.e. they aren't restrained to one domain such as Eliza the therapist is.

If it is human intelligence you are trying to simulate, then perhaps learning from humans is the way to go?

#### **Caution:** Interacting with chatbots that learn

Please note that the following exercise involves interacting with one of these chatterbots. Because the chatterbot has learnt from humans, it will quite possibly have been taught to say things that you may find highly offensive. While we have tried to choose chatterbots that mostly say things that aren't going to offend, it is impossible to guarantee this, so use your discretion with them; you can skip this section and still cover the main concepts of this chapter. Because Eliza and Alice don't learn from humans, they won't say offensive things unless you do first!

And again, don't tell the chatterbots your personal details (such as your full name, date of birth, address, or any other information you wouldn't be happy sharing with everybody). Just make stuff up where necessary. A chatterbot that learns from people quite possibly *will* pass on what you say to other people in an attempt to sound intelligent to *them*!

These warnings will make more sense once you've learnt how these chatterbots work.

An example of a chatterbot that learns from humans is Cleverbot.

[Click on this link to have a conversation with Cleverbot](#)



4959 people talking

Good afternoon.

**Where are you ?**

I'm at my computer.

**What brand of computer?**
  
Think About It! Think For Me! Thoughts So Far

Unlike Eliza and Alice, whose rules of what to say were determined by programmers, Cleverbot learns rules based on what people say. For example, when Cleverbot says "hi" to a person, it keeps track of all the different responses that people make to that, such as "hi", "hello!", "hey ya", "sup!". A rule is made that says that if somebody says hi to you, then the things that people have commonly said in response to Cleverbot saying hi are appropriate things to say in response to "hi". In turn, when Cleverbot says something like "sup!" or "hello!", it will look at how humans respond to that in order to learn appropriate response for those. And then it will learn responses for those responses. This allows Cleverbot to built up an increasingly large database.

An implication of learning from humans is that Cleverbot makes the assumption that the humans actually are intelligent, and will teach it to say intelligent things. If for example people told Cleverbot something like "School is boring" in response to Cleverbot saying "hi", Cleverbot might learn that when a person says "hi" to it, it should say "School is boring"!

**Curiosity:** A short film written by Cleverbot

Check out the short film "[Do You Love Me](#)" (~3 mins), the result when Chris R Wilson collaborated with Cleverbot to write a movie script.

## 10.2.4. Even more Chatterbots!

There are even more chatterbots you can talk to. Try looking at the [list on wikipedia](#), or doing a google search for chatterbots. Each chatterbot on this list has its own wikipedia page. You should be able to find the chatterbots by either an internet search, or looking at the references of the wikipedia pages. Some of these will have rules that were determined by programmers, and others will have rules that were learnt from humans.

If you have a device that runs Apple iOS (for example an iPhone), have a look at the [Siri](#) chatterbot in the device's help system. Siri is an example of a chatterbot that has the job of *helping* a human, unlike most chatterbots which simply have the purpose of web entertainment. It also has voice recognition, so you can talk to it rather than just typing to it.

## 10.2.5. The Turing Test

In the above sections you met some chatterbots, and (hopefully!) have drawn the conclusion that they aren't entirely convincing in terms of sounding like a human (although some are better than others!). But maybe soon, there will be new chatterbots that don't have the same limitations. Should we consider them to be intelligent? How could we tell? Is there a formal way we can determine whether or not a chatterbot is of the level of human intelligence?

A very famous computer scientist, Alan Turing, answered this question back in 1950, before the first chatterbots even existed! Alan Turing had an extraordinary vision of the future, and knew that coming up with computers that were intelligent would become a big thing, and that we would need a way to know when we have succeeded in creating a truly intelligent computer.

He thought about how intelligence could be defined (defining intelligence is surprisingly difficult!), and decided that one way would be to say that a human was intelligent, and that if a computer was able to communicate convincingly like a human, then it must be intelligent also. This definition doesn't cover all of intelligence, as it only considers what a person or a computer says and ignores other components of intelligence such as determining the best way to walk through a building (or maze) or deciding how to act in a specific situation (such as at a social event, when deciding what to do next at work, or when lost). However, communication is still a very significant component of human intelligence.

In order to test whether or not a computer program can communicate like a human, Turing proposed a test. In addition to the computer program, two humans are required to carry out the test. One of the humans act as an interrogator, and the other as a "human" to compare the computer program to. The interrogator is put in a separate room from the computer running the computer program and the "human". The interrogator has conversations with both the human and the computer program, but isn't told which one they are having the conversation with at each time. The conversations are both carried out over something like an instant messaging program so that actual speech isn't required from the computer program. During the conversations, the human has to convince the interrogator that they are indeed the human, and the computer program has to convince the interrogator that it is actually the human. At the end of the conversations, the interrogator has to say which was the computer and which was the human. If they can't reliably tell, then the computer is said to have passed the test.

This test proposed by Turing eventually became very famous and got the name "The Turing Test". One of the motivations for writing chatterbots is to try and make one that passes the Turing Test. Unfortunately, making a chatterbot that successfully passes the Turing Test hasn't yet been achieved, and whether or not it is even possible is still an open question in computer science, along with many other questions in artificial intelligence that you will encounter later in this chapter.

Other forms of the Turing Test exist as well. Action games sometimes have computer controlled characters that fight your own character, in place of a second human controlled character. A variation of the Turing Test can be used to determine whether or not the computer controlled player seems to have human intelligence by getting an interrogator to play against both the computer character and the human character, and to see whether or not they can tell them apart.

In fact, many parts of human intelligence could be tested using a variation of the Turing Test. If you wanted a computer chess player that seemed like a human as opposed to a computer (as some people might prefer to be playing against a human rather than a computer), you could use a Turing Test for this as well! What other possible Turing Tests can you think of?

### CURIOSITY: The real Turing test

Alan Turing actually started by suggesting a simple party game requiring three players, where the first player was female, the second player was male, and the third player could be either male or female, and took the role of the "interrogator". The interrogator would be in a separate room to the other two players, and could only

communicate with them by passing written notes (for example, by passing the notes under a door). The male had to try and convince the interrogator that he was actually female, and the female had to try and convince the interrogator that she was the female. At the end the interrogator had to say which was the male and which was the female, and if the interrogator guessed incorrectly, then the male "won".

### **Project:** Run your own Turing test on a chatterbot

This section will involve you actually carrying out the Turing Test. Read this entire section carefully (and the previous section if you haven't done so already) before you start, and make sure you understand it all before starting.

In science classes, such as biology, physics, and chemistry, carrying out experiments is commonly done. If you have taken classes like these, you will probably know that if an experiment isn't carried out properly (e.g. in chemistry some students are tempted to put in more of a chemical than the instructions say to, or when timing is important this is easy to get wrong), then the results will not necessarily be the ones you are after and your experiment is essentially meaningless and pointless. You also have to be careful that other factors don't affect the results. e.g. controlling temperature and moisture in biology experiments that involve growing micro-organisms.

Carrying out the Turing Test is carrying out an experiment, just like carrying out experiments in chemistry classes. And just like the chemistry experiments, carrying out the Turing Test requires being careful to follow instructions correctly, and controlling factors that could potentially affect the results but aren't part of what is being tested. You should keep this in mind while you are carrying out this project.

For example, most chatterbots communicate in a text form rather than verbal. Communicating in a verbal form involves not only choosing intelligent sounding things to say, but also involves having a convincing voice and pronouncing words correctly. Tone of voice or accent could potentially make it very obvious to the interrogator which conversation was with the human and which was the computer, without them even having to consider what was actually said in the conversation. This is not what the Turing Test is supposed to be testing! Therefore, the Turing Test will have both the computer and the human communicating in a written form.

As another example, speed of response could have an impact. The computer is likely to be able to reply instantly, whereas the human will need time to think and then write their reply. To prevent the interrogator from making their decision based on the speed instead of the content, the speed of response needs to be controlled as well. The way

of carrying out the Turing Test described below tries to control these additional factors.

Choose a chatterbot from the list on Wikipedia (see the above chatterbots section), or possibly use Alice or Cleverbot (Eliza isn't recommended for this). You will be taking the role of the interrogator, but will need another person to act as the "human". For this it is recommended you choose a person in your class who you don't know very well. Do *not* choose your best friend in the class, because you will know their responses to questions too well, so will be able to identify them from the chatterbot based on their personality rather than the quality of the chatterbot.

In addition to the chatterbot and your classmate to act as the human, you will need access to a room with a computer with internet (this could just be the computer classroom), another room outside it (a hallway would be fine), pieces of paper, 2 pens, and a coin or a dice.

The chatterbot should be loaded on the computer to be used, and your classmate should be in the same room with the computer. You should be outside that room. As the interrogator, you will first have a conversation with either your classmate or the computer, and then a conversation with the other one. You should not know which order you will speak to them; to determine which you speak to first your classmate should use the dice or the coin to randomly decide (and shouldn't tell you).

In order to carry out the conversations, start by writing something at the top of the piece of paper such as "hello" or "hi" or "how are you?". Put a mark next to the line to make it clear that line was written by YOU. Pass the piece of paper into the room where your classmate and the computer are (if you can, slide it under the door) where your classmate will write a reply on it and pass it back to you. You should then write a reply to that and repeat the process. Each conversation should be on a separate piece of paper, and be around 20 to 40 lines long (this means that each person/ computer should say around 10 - 20 lines in each conversation). Put a mark next to each of the lines you write, so that it is clear who wrote which lines.

If your classmate is currently supposed to be having the conversation (rather than the chatterbot), they will write the reply based on what they would say.

If the chatterbot is currently supposed to be having the conversation, your classmate should type what you said into the chatterbot and then write its reply on the piece of paper. Before submitting the line to the chatterbot, they should double check it was entered correctly.

A problem is that it will take longer for the conversation between you and the chatterbot than between you and the classmate, because of the need for your classmate to type what you say to the chatterbot. You wouldn't want to make it obvious which was the computer and which was the human due to that factor! To deal with this, you could intentionally delay for a while before each reply so that they all take exactly one minute.

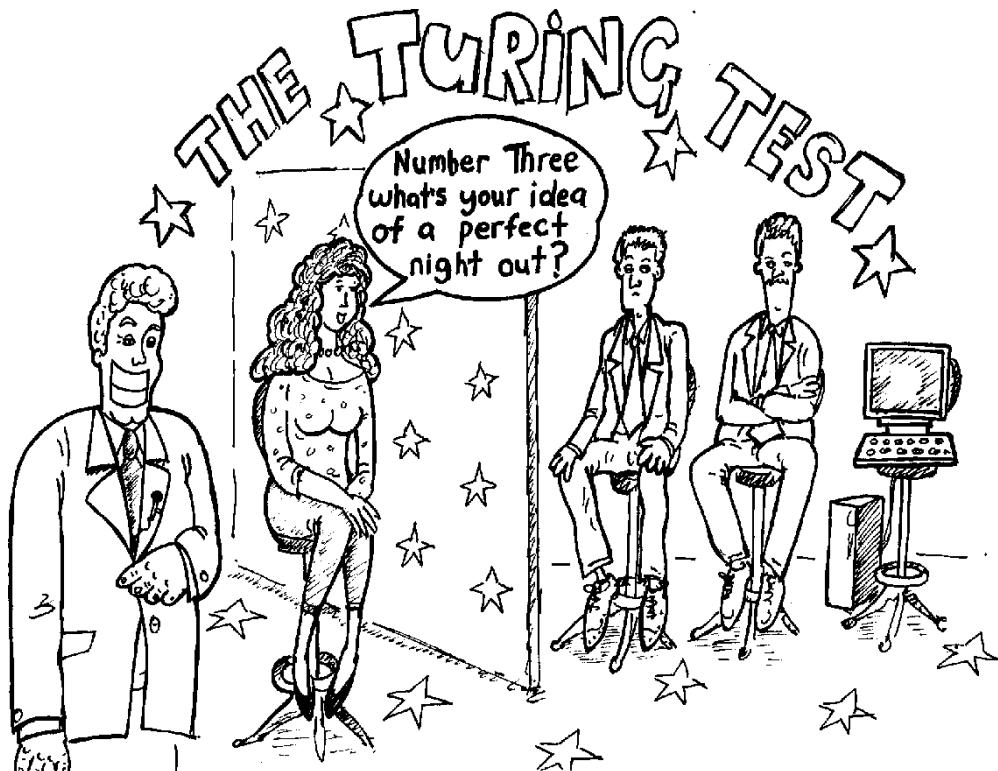
You can ask whatever you like, although keep in mind that you should be assuming you don't know your classmate already, so don't refer to common knowledge about previous things that happened such as in class or in the weekend in what you ask your classmate or the chatterbot. This would be unfair to the chatterbot since it can't possibly know about those things. Remember you're evaluating the chatterbot on its ability to display human intelligence, not on what it doesn't know about.

Good conversation topics would be favourite colours, games, foods, the weather, and the kinds of conversation topics you'd have with a person you don't know but are having a friendly conversation with at work, the supermarket, or a party. Coming up with good things to ask is challenging but just ask yourself whether something would require knowledge of an event that not everybody could be expected to have.

Once both conversations are complete, you as the interrogator has to say which was your classmate, and which was the chatterbot. Your classmate should tell you whether or not you were correct.

These are some questions you can consider after you have finished carrying out the Turing Test:

- How were you able to tell which was the chatterbot and which was your classmate?
- Were there any questions you asked that were "unfair" --- that depended on knowledge your classmate might have but no-one (computer or person) from another place could possibly have?
- Which gave it away more: the content of the answers, or the way in which the content was expressed?



## 10.3. The whole story!

In this chapter so far, we have only talked about one application of AI. AI contains many more exciting applications, such as computers that are able to play board games against humans, computers that are able to learn, and computers that are able to control robots that are autonomously exploring an environment too dangerous for humans to enter.

Eventually further sections on other topics in AI will be added to this chapter.

## 10.4. Further reading

- [Artificial Intelligence Strong and Weak - I Programmer](#)
- [The Paradox of Artificial Intelligence - I Programmer](#)

### 10.4.1. Useful Links

- [CS Unplugged - Programming Languages - Harold the Robot](#)- related to why AI is so tricky
- [CS Unplugged - The Turing Test - Conversations with Computers](#)
- [Wikipedia- Outline of Artificial Intelligence](#)
- [Wikipedia - Turing Test](#)
- [Wikipedia - Machine Learning](#)
- [CS 4 FUN - Meet the Chatterbots](#)

- [CS 4 FUN - The Illusion of Intelligence](#)
- [Alice Bot](#)
- [IEEE Spectrum](#)
- [TED Conversations matching Artificial Intelligence](#)

# 11. Complexity and tractability

## 11.1. What's the big picture?

Are there problems that are too hard even for computers? It turns out that there are. In the chapter on Artificial Intelligence we'll see that just having a conversation --- chatting --- is something computers can't do well, not because they can't speak but rather because they can't understand or think of sensible things to say. However, that's not the kind of hard problem we're talking about here --- it's not that computers couldn't have conversations, more that we don't know just how we do it ourselves and so we can't tell the computer what to do.

In this chapter we're going to look at problems where it's easy to tell the computer what to do --- by writing a program --- but the computer *can't* do what we want because it takes far too long: millions of centuries, perhaps. Not much good buying a faster computer either: if it were a hundred times faster it would still take millions of years; even one a million times faster would take hundreds of years. That's what you call a *hard* problem --- one where it takes far longer than the lifetime of the fastest computer imaginable to come up with a solution!

The area of *tractability* explores problems and algorithms that can take an impossible amount of computation to solve except perhaps for very small examples of the problem.

We'll define what we mean by **tractable** later on, but put very crudely, a tractable problem is one which we can write programs for that finish in a reasonable amount of time, and an intractable problem is one that will generally end up taking way too long.

Knowing when a problem you are trying to solve is one of these hard problems is very important. Otherwise it is easy to waste huge amounts of time trying to invent a clever program to solve it, and never getting anywhere. A computer scientist needs to be able to recognise a problem as an intractable problem, so that they can use other approaches. A very common approach is to give up on getting a perfect answer, and instead just aim for an approximately correct answer. There are a variety of techniques for getting good

approximate answers to hard problems; a way of getting an answer that isn't guaranteed to give the exact correct answer is sometimes referred to as a *heuristic*.

One important example of an intractable problem that this chapter looks at is the travelling salesman problem (TSP for short). It's a simple problem; if you've got a collection of places that you need to visit, and you know the distance to travel between each pair of places, what's the shortest route that visits all of the places exactly once? This is a very practical problem that comes up with courier vehicles choosing routes to deliver parcels, rock bands planning tours, and even a designated driver dropping friends off after an event. In fact, the measurement between cities doesn't have to be distance. It could actually be the dollar cost to travel between each pair of cities. For example, if you needed to visit Queenstown, Christchurch, Auckland, and Wellington one after the other while minimising airfares and you knew the cost of an airfare between each pair of those 4 cities, you could work out what the cheapest way of flying to each of them is. This is still an example of TSP. And it can also be applied to problems that don't involve travel; for example, it has been used to [work out how to efficiently synthesise DNA](#).

The following interactive has a program that solves the problem for however many cities you want to select by trying out all possible routes, and recording the best so far. You can get a feel for what an intractable problem looks like by seeing how long the interactive takes to solve the problem for different size maps. Try generating a map with about 5 cities, and press "Start" to solve the problem.

[View city trip](#)

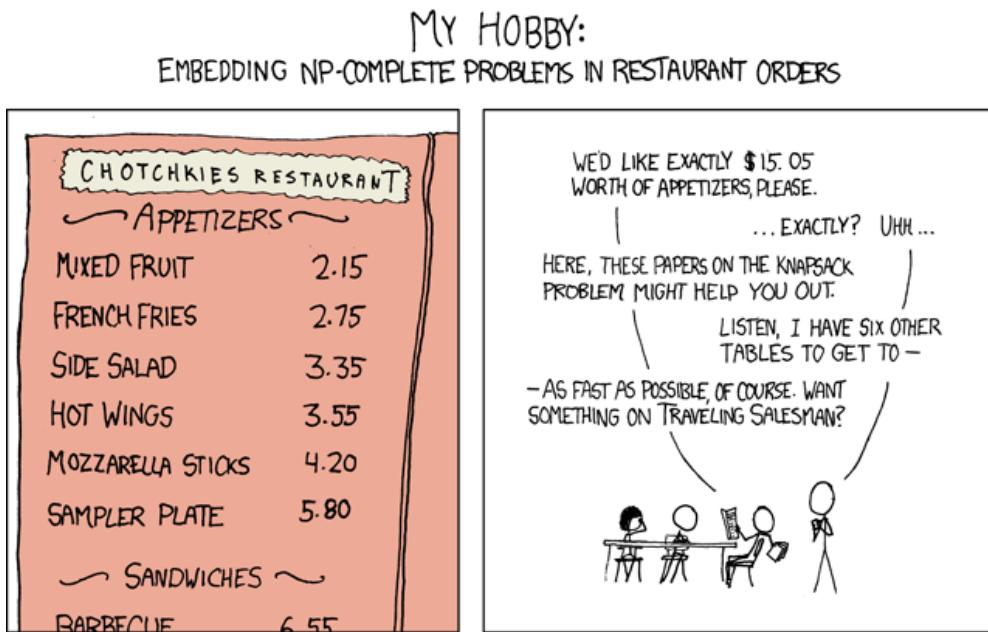
interactive

View the link online at [http://www.csfieldguide.org.nz/releases/1.9.9/\\_static/widgets/tract-tsp-basic-v2.html](http://www.csfieldguide.org.nz/releases/1.9.9/_static/widgets/tract-tsp-basic-v2.html)

Now try it for 10 cities (twice as many). Does it take twice as long? How about twice as many again (20 cities)? What about 50 cities? Can you guess how long it would take? You're starting to get a feel for what it means for a problem to be *intractable*.

Of course, for some situations, intractable problems are a good thing. In particular, most security and cryptography algorithms are based on intractable problems; the codes could be broken, but it would take billions of years and so would be futile. In fact, if anyone ever finds a fast algorithm for solving such problems, a lot of computer security systems would stop being secure overnight! So one of the jobs of computer scientists is to be confident that such solutions *don't* exist!

In this chapter we will look at the TSP and other problems for which *no* tractable solutions are known, problems that would take computers millions of centuries to solve. And we will encounter what is surely the greatest mystery in computer science today: that *no-one* knows whether there's a more efficient way of solving these problems! It may be just that no-one has come up with a good way yet, or it may be that there is no good way. We don't know which.



[Image source](#)

But let's start with a familiar problem that we can actually solve.

## 11.2. Algorithms, problems, and speed limits

**Complexity** is an important concept with problems and algorithms that solve them. Usually complexity is just the amount of time it takes to solve a problem, but there are several ways that we can measure the "time". Using the actual time on a particular computer can be useful, but to get a rough idea of the inherent behaviour of an algorithm, computer scientists often start by estimating the number of steps the algorithm will take for  $n$  items. For example, a linear search can end up checking each of  $n$  items being searched, so the algorithm will take  $n$  steps. An algorithm that compares every pair of values in a list of  $n$  items will have to make  $n^2$  comparisons, so we can characterise it as taking about  $n^2$  steps. This gives us a lot of information about how good an algorithm is without going into

details of which computer it was running on, which language, and how well written the program was. The term *complexity* is generally used to refer to these rough measures.

Having a rough idea of the complexity of a problem helps you to estimate how long it's likely to take. For example, if you write a program and run it with a simple input, but it doesn't finish after 10 minutes, should you quit, or is it about to finish? It's better if you can estimate the number of steps it needs to make, and then extrapolate from the time it takes other programs to find related solutions.

### Jargon Buster: Asymptotic complexity

If you're reading about complexity, you may come across some terminology like "Big Oh" notation and "asymptotic complexity", where an algorithm that takes about  $n^2$  steps is referred to as  $O(n^2)$ . We won't get into these in this chapter, but here's a little information in case you come across the terms in other reading. "Big Oh" notation is a precise way to talk about complexity, and is used with "asymptotic complexity", which simply means how an algorithm performs for large values of  $n$ . The "asymptotic" part means as  $n$  gets really large --- when this happens, you are less worried about small details of the running time. If an algorithm is going to take seven days to complete, it's not that interesting to find out that it's actually 7 days, 1 hour, 3 minutes and 4.33 seconds, and it's not worth wasting time to work it out precisely.

We won't use precise notation for asymptotic complexity (which says which parts of speed calculations you can safely ignore), but we will make rough estimates of the number of operations that an algorithm will go through. There's no need to get too hung up on precision since computer scientists are comfortable with a simple characterisation that gives a ballpark indication of speed.

For example, consider using selection sort to put a list of  $n$  values into increasing order. (This is explained in the chapter on algorithms). Suppose someone tells you that it takes 30 seconds to sort a thousand items. Does that sounds like a good algorithm? For a start, you'd probably want to know what sort of computer it was running on - if it's a supercomputer then that's not so good; if it's a tiny low-power device like a smartphone then maybe it's ok.

Also, a single data point doesn't tell you how well the system will work with larger problems. If the selection sort algorithm above was given 10 thousand items to sort, it would probably take about 50 minutes (3000 seconds) --- that's 100 times as long to process 10 times as much input.

These data points for a particular computer are useful for getting an idea of the performance (that is, complexity) of the algorithm, but they don't give a clear picture. It turns out that we can work out exactly how many steps the selection sort algorithm will take for  $n$  items: it will require about  $\frac{n(n-1)}{2}$  operations, or in expanded form,  $\frac{n^2}{2} - \frac{n}{2}$  operations. This formula applies regardless of the kind of computer its running on, and while it doesn't tell us the time that will be taken, it can help us to work out if it's going to be reasonable.

From the above formula we can see why it gets bad for large values of  $n$ : the number of steps taken increases with the square of the size of the input. Putting in a value of 1 thousand for  $n$  tells us that it will use  $1,000,000/2 - 1,000/2$  steps, which is 499,500 steps.

Notice that the second part ( $1000/2$ ) makes little difference to the calculation. If we just use the  $\frac{n^2}{2}$  part of the formula, the estimate will be out by 0.1%, and quite frankly, the user won't notice if it takes 20 seconds or 19.98 seconds. That's the point of asymptotic complexity --- we only need to focus on the most significant part of the formula, which contains  $n^2$ .

Also, since measuring the number of steps is independent of the computer it will run on, it doesn't really matter if it's described as  $\frac{n^2}{2}$  or  $n^2$ . The amount of time it takes will be proportional to both of these formulas, so we might as well simplify it to  $n^2$ . This is only a rough characterisation of the selection sort algorithm, but it tells us a lot about it, and this level of accuracy is widely used to quickly but fairly accurately characterise the complexity of an algorithm. In this chapter we'll be using similar crude characterisations because they are usually enough to know if an algorithm is likely to finish in a reasonable time or not.

If you've studied algorithms, you will have learnt that some sorting algorithms, such as mergesort and quicksort, are inherently faster than other algorithms, such as insertion sort, selection sort, or bubble sort. It's obviously better to use the faster ones. The first two have a complexity of  $n \log(n)$  time (that is, the number of steps that they take is roughly proportional to  $n \log(n)$ ), whereas the last three have complexity of  $n^2$ . Generally the consequence of using the wrong sorting algorithm will be that a user has to wait many minutes (or perhaps hours) rather than a few seconds or minutes.

Here we're going to consider another possible sorting algorithm, called *permutation sort*. Permutation sort says "Let's list all the possible orderings ("permutations") of the values to be sorted, and check each one to see if it is sorted, until the sorted order is found". This algorithm is straightforward to describe, but is it any good?

For example, if you are sorting the numbers 45, 21 and 84, then every possible order they can be put in (that is, all permutations) would be listed as:

45, 21, 84

45, 84, 21

21, 45, 84

21, 84, 45

84, 21, 45

84, 45, 21

Going through the above list, the only line that is in order is 21, 45, 84, so that's the solution. It's a very inefficient approach, but it will help to illustrate what we mean by tractability.

In order to understand how this works, and the implications, choose four different words (in the example below we have used colours) and list all the possible orderings of the four words. Each word should appear exactly once in each ordering. You can either do this yourself, or use an online permutation generator such as [JavaScriptPermutations](#) or [Text Mechanic](#).

For example if you'd picked red, blue, green, and yellow, the first few orderings could be:

red, blue, green, yellow

red, blue, yellow, green

red, yellow, blue, green

red, yellow, green, blue

They do not need to be in any particular order, although a systematic approach is recommended to ensure you don't forget any!

Once your list of permutations is complete, search down the list for the one that has the words sorted in alphabetical order. The process you have just completed is using permutation sort to sort the words.

Now add another word. How many possible orderings will there be with 5 words? What about with only 2 and 3 words --- how many orderings are there for those? If you gave up on writing out all the orderings with 5 words, can you now figure out how many there might be? Can you find a pattern? How many do you think there might be for 10 words? (You don't have to write them all out!).

If you didn't find the pattern for the number of orderings, think about using factorials. For 3 words, there are  $3!$  ("3 factorial") orderings. For 5 words, there are  $5!$  orderings. Check the jargon buster below if you don't know what a "factorial" is, or if you have forgotten!

### Jargon Buster: Factorials

Factorials are very easy to calculate; just multiply together all the integers from the number down to 1. For example, to calculate  $5!$  you would simply multiply:  $5 \times 4 \times 3 \times 2 \times 1 = 120$ . For  $8!$  you would simply multiply  $8 \times 7 \times 6 \times 5 \times 4 \times 3 \times 2 \times 1 = 40,320$ .

As stated above, the factorial of a number tells you how many permutations (orderings) there would be for that number of words (assuming they are all different). This means that if you are arranging 8 words, there will be 40,320 ways of arranging them (which is why you weren't asked to try this in the first exercise!!)

Your calculator may have a "!" button for calculating factorials and spreadsheets usually have a "FACT" function, although for the factorials under 10 in this section, we recommend that you calculate them the long way, and then use the calculator as a double check. Ensuring you understand how a factorial is calculated is essential for understanding the rest of this section!

For factorials of larger numbers, most desktop calculators won't work so well; for example,  $100!$  has 158 digits. You can use the calculator below to work with huge numbers (especially when using factorials and exponents).

Use the interactive online at <http://www.csfieldguide.org.nz/en/interactives/big-number-calculator/index.html>

Try calculating  $100!$  using this calculator --- that's the number of different routes that a travelling salesman might take to visit 100 places (not counting the starting place). With this calculator you can copy and paste the result back into the input if you want to do further calculations on the number. If you are doing these calculations for a report, you should also copy each step of the calculation into your report to show how you got the result.

There are other big number calculators available online; for example, the [Big Integer Calculator](#). Other big calculators are available online, or you could look for one to download for a desktop machine or smartphone.

As a final exercise on permutation sort, calculate how long a computer would take to use permutation sort to sort 100 numbers. Remember that you can use the calculator that was linked to above. Assume that you don't have to worry about how long it will take to generate the permutations, only how long it will take to check them. Assume that you have a computer that creates and checks an ordering every nanosecond.

- How many orderings need to be checked for sorting 100 numbers?
- How many orderings can be checked in a second?
- How many orderings can be checked in a year?
- How many years will checking all the orderings take?

And as an interesting thing to think about, do some calculations based on the assumptions listed below. How long would it take to use permutation sort on 100 numbers? What would happen first: the algorithm would finish, or the universe would end?

- There are  $10^{82}$  atoms in the universe
- The universe has another 14 billion years before it ends
- Suppose every atom in the universe is a computer that can check an ordering every nanosecond

By now, you should be very aware of the point that is being made. Permutation sort is so inefficient that sorting 100 numbers with it takes so long that it is essentially impossible. Trying to use permutation sort with a non trivial number of values simply won't work. While selection sort is a lot slower than quick sort or merge sort, it wouldn't be impossible for Facebook to use selection sort to sort their list of 1 billion users. It would take a lot longer than quick sort would, but it would be doable. Permutation sort on the other hand would be impossible to use!

At this point, we need to now distinguish between algorithms that are essentially usable, and algorithms that will take billions of year to finish running, even with a small input such as 100 values.

Computer Scientists call an algorithm "intractable" if it would take a completely unreasonable amount of time to run on reasonably sized inputs. Permutation sort is a good example of an intractable algorithm. The term "intractable" is used a bit more formally in computer science; it's explained in the next section.

But the *problem* of sorting items into order is not intractable - even though the Permutation sort algorithm is intractable, there are lots of other efficient and not-so-efficient algorithms that you could use to solve a sorting problem in a reasonable amount of time: quick sort, merge sort, selection sort, even bubble sort! However, there are some problems in which

the ONLY known algorithm is one of these intractable ones. Problems in this category are known as *intractable problems*.

### Curiosity: Towers of Hanoi

The Towers of Hanoi problem is a challenge where you have a stack of disks of increasing size on one peg, and two empty pegs. The challenge is to move all the disks from one peg to another, but you may not put a larger disk on top of a smaller one. There's a description of it at [Wikipedia](#).

This problem cannot be solved in fewer than  $2^{n-1}$  moves, so it's an intractable problem (a computer program that lists all the moves to make would use at least  $2^{n-1}$  steps). For 6 disks it only needs 63 moves, but for 50 disks this would be 1,125,899,906,842,623 moves.

We usually characterise a problem like this as having a complexity of  $2^n$ , as subtracting one to get a precise value makes almost no difference, and the shorter expression is simpler to communicate to others.

The Towers of Hanoi is one problem where we know for sure that it will take exponential time. There are many intractable problems where this isn't the case --- we don't have tractable solutions for them, but we don't know for sure if they don't exist. Plus this isn't a real problem --- it's just a game (although there is a backup system based on it). But it is a nice example of an exponential time algorithm, where adding one disk will double the number of steps required to produce a solution.

## 11.3. Tractability

There's a very simple rule that computer scientists use to decide if an algorithm is tractable or not, based on the complexity (estimated number of steps) of the algorithm. Essentially, if the algorithm takes an exponential amount of time or worse for an input of size  $n$ , it is labelled as intractable. This simple rule is a bit crude, but it's widely used and provides useful guidance. (Note that a factorial amount of time,  $n!$ , is intractable because it's bigger than an exponential function.)

To see what this means, let's consider how long various algorithms might take to run. The following interactive will do the calculations for you to estimate how long an algorithm might take to run. You can choose if the running time is exponential (that is,  $2^n$ , which is the time required for the Towers of Hanoi problem with  $n$  disks), or factorial (that is,  $n!$ , which is the time required for checking all possible routes a travelling salesman would

make to visit  $n$  places other than the starting point). You can use the interactive below to calculate the time.

For example, try choosing the factorial time for the TSP, and put in 20 for the value of  $n$  (i.e. this is to check all possible travelling salesman visits to 20 places). Press the return or tab key to update the calculation. The calculator will show a large number of seconds that the program will take to run; you can change the units to years to see how long this would be.

[View big time](#)

[calculator](#)

View the link online at [http://www.csfieldguide.org.nz/releases/1.9.9/\\_static/widgets/tract-scaling-v2.html](http://www.csfieldguide.org.nz/releases/1.9.9/_static/widgets/tract-scaling-v2.html)

So far the calculation assumes that the computer would only do 1 operation per second; try changing to a million (1,000,000) operations per second, which is more realistic, and see how long that would take.

Another way to solve problems faster is to have multiple processors work on different solutions at the same time. If you were to buy 1,000 processors (e.g. 1,000 computers, or 250 4-core computers) and have each one test out different routes, then the solution could be found 1,000 times faster. Try changing the number of processors to 1,000, and see how long that would take (you may need to change the units back --- is it seconds? hours? days?)

The interactive above estimates the amount of time taken for various algorithms to run given  $n$  values to be processed. Let's assume that we have a very fast computer, faster than any that exist. Try putting in the assumption that the computer can do a million million (1,000,000,000,000) steps per second. Is that achievable? But what if you add just two more locations to the problem (i.e.  $n=22$  instead of  $n=20$ )?

Now, consider an algorithm that has a complexity of  $n^2$  (there are lots that take roughly this number of steps, including selection sort which was mentioned earlier). Type in a value of 1,000,000 for  $n$  to see how long it might take to sort a million items on a single processor (keep the number of steps per second at 1,000,000,000,000, but set the number of processors to just 1) --- it should show that it will only take about 1 second on our hypothetical very fast machine. Now put in 10 million for  $n$  --- although it's sorting a list 10 times as big, it takes more than 10 times as long, and will now take a matter of minutes rather than seconds. At what value of  $n$  does the amount of time become out of the

question --- that is, how large would the problem need to be for it to take years to finish? Is anyone ever likely to be sorting this many values --- for example, what if for some reason you were sorting the name of every person in the world, or every base in the human genome?

What about an algorithm with complexity of  $n^3$ ? What's the largest size input that it can process in a reasonable amount of time?

Now try the same when the number of steps is  $2^n$ , but start with a value of 10 for  $n$ , then try 30, 40, 50 and so on. You'll probably find that for an input of about 70 items it will take an unreasonable amount of time. Is it much worse for 80 items?

Now try increasing the number of operations per second to 10 times as many. Does this help to solve bigger problems?

Trying out these figures you will likely have encountered the barrier between "tractable" and "intractable" problems. Algorithms that take  $n^2$ ,  $n^3$  or even  $n^4$  time to solve a problem (such as sorting a list) aren't amazing, but at least with a fast enough computer and for the size of inputs we might reasonably encounter, we have a chance of running them within a human lifetime, and these are regarded as *tractable*. However, for algorithms that take  $2^n$ ,  $3^n$  or more steps, the amount of time taken can end up as billions of years even for fairly small problems, and using computers that are thousand times faster still doesn't help to solve much bigger problems. Such problems are regarded as *intractable*. Mathematically, the boundary between tractable and intractable is between a polynomial number of steps (polynomials are formulas made up of  $n^2$ ,  $n^3$ ,  $n^4$  and so on), and an exponential number of steps ( $2^n$ ,  $3^n$ ,  $4^n$ , and so on).

The two formulas  $n^2$  and  $2^n$  look very similar, but they are really massively different, and can mean a difference between a few seconds and many millennia for the program to finish. The whole point of this chapter is to develop an awareness that there are many problems that we have tractable algorithms for, but there are also many that we haven't found any tractable algorithms for. It's very important to know about these, since it will be futile to try to write programs that are intractable, unless you are only going to be processing very small problems.

Note that algorithms that take a factorial amount of time ( $n!$ , or  $1 \times 2 \times 3 \times \dots n$ ) are in the intractable category (in fact, they take times that are a lot worse than  $2^n$ ).

Essentially any algorithm that tries out all combinations of the input will inevitably be intractable because the number of combinations is likely to be exponential or factorial. Thus an important point is that it's usually not going to work to design a system that just tries out all possible solutions to see which is the best.

Although we've provided  $n^6$  as an example of a tractable time, nearly all algorithms you're likely to encounter will be  $n^3$  and better, or  $2^n$  and worse --- only very specialised ones fall in the gap between those. So there's a big gulf between tractable and intractable problems, and trying to grapple with it is one of the biggest problems in computer science!

What about Moore's law, which says that computing power is increasing exponentially? Perhaps that means that if we wait a while, computers will be able to solve problems that are currently intractable? Unfortunately this argument is wrong; intractable problems are also exponential, and so the rate of improvement due to Moore's law means that it will only allow for slightly larger intractable problems to be solved. For example, if computing speed is doubling every 18 months (an optimistic view of Moore's law), and we have an intractable problem that takes  $2^n$  operations to solve (many take longer than this), then in 18 months we will be able to solve a problem that's just one item bigger. For example, if you can solve an exponential time problem for 50 items (50 countries on a map to colour, 50 cities for a salesman to tour, or 50 rings on a Towers of Hanoi problem) in 24 hours, then in 18 months you can expect to buy a computer that could solve it for 51 items at best! And in 20 years you're likely to be able to get a computer that could solve for 55 items in one day. You're going to have to be more than patient if you want Moore's law to help out here --- you have to be prepared to wait for decades for a small improvement!

Remember that if you need to do calculations of huge numbers, there's a calculator here that you can use:

Use the interactive online at <http://www.csfieldguide.org.nz/en/interactives/big-number-calculator/index.html>

## 11.4. The Travelling Salesman Problem

An example of an intractable problem is the Travelling Salesman Problem (TSP). The TSP involves a bunch of locations (cities, houses, airports,...) where you can travel between any possible pair of locations. The goal is to find the shortest route that will go through all the locations once --- this is what the interactive at the start of this chapter does.

Researchers have spent a lot of time trying to find efficient solutions to the Travelling Salesman Problem, yet have been unable to find a *tractable* algorithm for solving it. As you learnt in the previous section, *intractable* algorithms are very slow, to the point of being impossible to use. As the only solutions to TSP are intractable, TSP is known as an *intractable problem*.

It hasn't actually been *proven* that there is no tractable solution to TSP, although many of the world's top computer scientists have worked on this problem for the last 40 years, trying to find a solution but without success. What they have managed to do is find thousands of other problems that are also intractable, and more importantly, if a solution is found for any one of these problems, we know how to convert it to a solution for any of the others (these are called NP-complete problems). They all stand and fall together, including the TSP problem. So it's not just being lazy if you give up on finding an optimal TSP algorithm -- people have tried for decades and not found a tractable algorithm. Of course, this is also a strong motivator to try to find one -- if you do, you will have solved thousands of other problems at the same time! This is a great thing for a researcher to do, but if you have a program to get finished by the end of the month, it's not a good bet to work on it.

Current algorithms for finding the optimal TSP solution aren't a lot better than simply trying out all possible paths through the map (as in the interactive at the start of this chapter). The number of possible paths gets out of hand; it's an intractable approach. In the project below you'll be estimating how long it would take.

While TSP was originally identified as being the problem that sales people face when driving to several different locations and wanting to visit them in the order that leads to the shortest route (less petrol usage), the same problem applies to many other situations as well. Courier and delivery companies have variants of this problem -- often with extra constraints such as limits on how long a driver can work for, or allowing for left hand turns being faster than right-hand ones (in NZ at least!).

Since these problems are important for real companies, it is not reasonable to simply give up and say there is no solution. Instead, when confronted with an intractable problem, computer scientists look for algorithms that produce approximate solutions -- solutions that are not perfectly correct or optimal, but are hopefully close enough to be useful. By relaxing the requirement that the solution has to be perfectly correct, it is often possible to come up with tractable algorithms that will find good enough solutions in a reasonable time. This kind of algorithm is called a *heuristic* - it uses rules of thumb to suggest good choices and build up a solution made of pretty good choices.

A simple heuristic that often works OK is a *greedy* heuristic algorithm -- an algorithm that just takes what looks like the best choice at each step. For example, for the TSP, a greedy heuristic algorithm might repeatedly take the route to the next closest city. This won't always be the best choice, but it is very fast, and experience shows that it is typically no more than 25% worse than the optimal. There are more sophisticated ways of designing approximate algorithms that can do better than this (some can get within 3% of optimal for the TSP), but they take longer to run.

There are software companies that work on trying to make better and better approximate algorithms for guiding vehicles by GPS for delivery routes. Companies that write better algorithms can charge a lot of money if their routes are faster, because of all the fuel and time savings that can be made.

An interesting thing with intractability is that you can have two very similar problems, with one being intractable and the other being tractable. For example, finding the shortest route between two points (like a GPS device usually does) is a tractable problem, yet finding the shortest route around multiple points (the TSP) isn't. By the way, finding the *longest* path between two points (without going along any route twice) is also intractable, even though finding the *shortest* path is tractable!

### **Project:** The craypots problem

This project is based around a scenario where there is a cray fisher who has around 18 craypots that have been laid out in open water. Each day the fisher uses a boat to go between the craypots and check each one for crayfish.

The cray fisher has started wondering what the shortest route to take to check all the craypots would be, and has asked you for your help. Because every few weeks the craypots need to be moved around, the fisher would prefer a general way of solving the problem, rather than a solution to a single layout of craypots. Therefore, your investigations must consider more than one possible layout of craypots, and the layouts investigated should have the craypots placed *randomly* i.e. not in lines, patterns, or geometric shapes.

When asked to generate a random map of craypots, get a pile of coins (or counters) with however many craypots you need, and scatter them onto an A4 piece of paper. If any land on top of each other, place them beside one another so that they are touching but not overlapping. One by one, remove the coins, making a dot on the paper in the centre of where each coin was. Number each of the dots. Each dot represents one craypot that the cray fisher has to check. You should label the top left corner of the paper as being the boat dock, where the cray fisher stores the boat.

Generate a map with 7 or 8 craypots using the random map generation method described above. *Make an extra copy of this map, as you will need it again later.*

Using your intuition, find the shortest path between the craypots.

Now generate a map (same method as above) with somewhere between 15 and 25 craypots. *Make more than one copy of this map, as you will need it again later*

Now on this new map, try to use your intuition to find the shortest path between the craypots. Don't spend more than 5 minutes on this task; you don't need to include the solution in your report. Why was this task very challenging? Can you be sure you have an optimal solution?

Unless your locations were laid out in a circle or oval, you probably found it very challenging to find the shortest route. A computer would find it even harder, as you could at least take advantage of your visual search and intuition to make the task easier. A computer could only consider two locations at a time, whereas you can look at more than two. But even for you, the problem would have been challenging! Even if you measured the distance between each location and put lines between them and drew it on the map so that you didn't have to judge distances between locations in your head, it'd still be very challenging for you to figure out!

A straightforward algorithm to guarantee that you find the shortest route is to check *all* possible routes. This involves working out what all the possible routes are, and then checking each one. A possible route can be written as a list of the locations (i.e. the numbers on the craypots), in the order to go between them. This should be starting to sound familiar to you assuming you did the permutation sort discussed above. Just like in that activity you listed all the possible ordering for the values in the list to be sorted, this algorithm would require listing all the possible orderings of the craypots, which is equivalent (although you don't need to list all the orderings for this project!).

How many possible routes are there for the larger example you have generated? How is this related to permutation sort, and factorials? How long would it take to calculate the shortest route in your map, assuming the computer can check 1 billion (1,000,000,000) possible routes per second? (i.e. it can check one route per nanosecond) What can you conclude about the cost of this algorithm? Would this be a good way for the cray fisher to decide which path to take?

Make sure you show *all* your mathematical working in your answers to the above questions!

So this algorithm is intractable, but maybe there is a more clever algorithm that is tractable?

The answer is No.

You should be able to tell that this problem is equivalent to the TSP, and therefore it is intractable. How can you tell? What is the equivalent to a town in this scenario? What is the equivalent to a road?

Since we know that this craypot problem is an example of the TSP, and that there is no known tractable algorithm for the TSP, we know there is no tractable algorithm for the craypot problem either. Although there are slightly better algorithms than the one we used above, they are still intractable and with enough craypots, it would be impossible to work out a new route before the cray fisher has to move the pots again!

Instead of wasting time on trying to invent a clever algorithm that no-one has been able to find, we need to rely on an algorithm that will generate an approximate solution. The cray fisher would be happy with an approximate solution that is say, 10% longer than the best possible route, but which the computer can find quickly.

There are several ways of approaching this. Some are better than others in general, and some are better than others with certain layouts. One of the more obvious approximate algorithms, is to start from the boat dock in the top left corner of your map and to go to the nearest craypot. From there, you should go to the nearest craypot from that craypot, and repeatedly go to the nearest craypot that hasn't yet been checked. This approach is known as a *greedy heuristic algorithm* as it always makes the decision that looks the best at the current time, rather than making a not so good decision now to try and get a bigger pay off later. You will understand why this doesn't necessarily lead to the optimal solution after completing the following exercises.

On a copy of each of your 2 maps you generated, draw lines between the craypots to show the route you would find following the greedy algorithm (you should have made more than one copy of each of the maps!)

For your map with the smaller number of craypots (7 or 8), compare your optimal solution and your approximate solution. Are they the same? Or different? If they are the same, would they be the same in all cases? Show a map where they would be different (you can choose where to place the craypots yourself, just use as many craypots as you need to illustrate the point).

For your larger map, show why you don't have an optimal solution. The best way of doing this is to show a route that is similar to, but shorter than the approximate solution. The shorter solution you find doesn't have to be the optimal solution, it just has to be shorter than the one identified by the approximate algorithm (Talk to your teacher if you can't find a shorter route and they will advise on whether or not you should generate a new map). You will need to show a map that has a greedy route and a shorter route marked on it. Explain the technique you used to show there was a shorter solution. Remember that it doesn't matter how much shorter the new solution you identify is, just as long as it is at least slightly shorter than the approximate

solution --- you are just showing that the approximate solution couldn't possibly be the optimal solution by showing that there is a shorter solution than the approximate solution.

Even though the greedy algorithm only generates an approximate solution, as opposed to the optimal solution, explain why is it more suitable for the cray fisher than generating an optimal solution would be?

Why would it be important to the cray fisher to find a short route between the craypots, as opposed to just visiting them in a random order? Discuss other problems that are equivalent to TSP that real world companies encounter every day. Why is it important to these companies to find good solutions to TSP? Estimate how much money might a courier company be wasting over a year if their delivery routes were 10% worse than the optimal. How many different locations/towns/etc might their TSP solutions have to be able to handle?

Find a craypot layout that results in the greedy algorithm finding what seem to be a really inefficient route. Why is it inefficient? Don't worry about trying to find an actual worst case, just find a case that seems to be quite bad. What is a general pattern that seems to make this greedy algorithm inefficient?

Don't forget to include an introductory paragraph in your report that outlines the key ideas. It should include a brief description of what an intractable problem is, and how a computer scientist goes about dealing with such a problem. The report should also describe the Travelling Salesman Problem and the craypot problem in your own words. Explain why the craypot problem is a realistic problem that might matter to someone.

## 11.5. Other intractable problems

There are thousands of problems like the TSP for which no tractable solution is known. Extra sections will eventually be added here to introduce some of them, but in the meantime, if you are keen you might like to explore some of these problems:

- [map and graph colouring](#) (these can be reduced to a timetabling problem and vice versa, showing how NP-complete problems can relate to each other)
- [the knapsack problem](#)
- [the bin packing problem](#)
- [Hamiltonian paths](#) (no tractable solution for this is known, yet the very similar Eulerian path, which is often presented as the seven bridges problem, has an easy tractable solution)

- Steiner trees
- Dominating sets
- Longest path (this is interesting because finding the longest path is intractable, yet finding the shortest path is tractable - the shortest path is calculated when a GPS device works out the shortest route to a destination. Also, a Hamiltonian problem can be reduced easily to longest path, showing the concept of reduction when one NP-complete problem is used to solve another). [And here's a song about it!](#)
- the Battleship problem

## 11.6. The whole story!

The question of tractability is a big one in computer science --- in fact, what is widely regarded as the biggest unsolved problem in computer science revolves around it. You may recall that we mentioned that there are thousands of problems that we don't have a tractable solution for, yet a tractable solution to one can be adapted to all the others. This group of problems is called "NP-complete" (NP stands for non-deterministic polynomial if you really want to know; complete just means that they can all be converted to each other!) The big question is whether or not there is a polynomial time algorithm for any one of them, in which case all NP problems will have a P (polynomial time) solution. The question is often referred to as whether or not P equals NP.

Actually, things get worse. So far we've talked about intractable problems --- ones that can be solved, but might need billions of years on a computer. If you think it's bad that some problems take that long to solve, that's nothing! These problems are at least decidable --- if given enough time there exists an algorithm that will always lead to a correct answer. There are some well known problems that are undecidable -- we know we can *never* write a correct algorithm to solve the problem on a computer. For example, writing a program that reliably tells you if another program will finish or not is impossible! There are other examples of such problems here:

- <http://www.cs4fn.org/algorithms/tiles.php>
- <http://www.cs4fn.org/algorithms/uncomputable.php>
- <http://www.cs4fn.org/algorithms/haltingproblem.php>

It's good to know about these issues, to avoid getting stuck writing impossible programs. It's also a fascinating area of research with opportunities to make a discovery that could

change the world of computing, as well as contribute to our understanding on what can and can't be computed.

## 11.7. Further reading

This topic is covered very thoroughly in a way that is accessible to non-specialists in a popular book by David Harel called "Computers Ltd.: What They Really Can't Do".

### 11.7.1. Useful Links

- [https://en.wikipedia.org/wiki/Computational\\_complexity\\_theory](https://en.wikipedia.org/wiki/Computational_complexity_theory)
- <http://www.tsp.gatech.edu/games/index.html>
- <http://csunplugged.org/graph-colouring>
- [https://en.wikipedia.org/wiki/Travelling\\_salesman\\_problem](https://en.wikipedia.org/wiki/Travelling_salesman_problem)
- [https://en.wikipedia.org/wiki/Knapsack\\_problem](https://en.wikipedia.org/wiki/Knapsack_problem)
- [https://en.wikipedia.org/wiki/Bin\\_packing\\_problem](https://en.wikipedia.org/wiki/Bin_packing_problem)
- [https://en.wikipedia.org/wiki/Hamiltonian\\_path](https://en.wikipedia.org/wiki/Hamiltonian_path)
- [https://en.wikipedia.org/wiki/Brute-force\\_search](https://en.wikipedia.org/wiki/Brute-force_search)

# 12. Computer Graphics

Watch the video online at <https://www.youtube.com/embed/5kuoRjgfCl8>

## 12.1. What's the big picture?

Computer graphics will be familiar from games, films and images, and there is amazing software available to create images, but how does the software work? The role of a computer scientist is not just to *use* graphics systems, but to *create* them, and especially invent new techniques.

The entertainment industry is always trying to develop new graphics software so that they can push the boundaries and create new experiences. We've seen this in the evolution of animated films, from simple 2D films to realistic computer generated movies with detailed 3D images. The names of dozens of computer scientists now regularly appear in the credits for films that use CGI or animation, and [some have even won Oscars for their innovative software!](#)

Movie and gaming companies can't always just use existing software to make the next great thing -- they need computer scientists to come up with better graphics techniques to make something that's never been seen before. The creative possibilities are endless!

Computer graphics are used in a wide variety of situations: games and animated movies are common examples, but graphics techniques are also used to visualise large amounts of data (such as all cellphone calls being made in one day or friend connections in a social network), to display and animate graphical user interfaces, to create virtual reality and augmented reality worlds, and much more.

### Jargon Buster: Pixels

A digital image on a screen or printer is physically made up of a grid of tiny squares called [pixels](#). They are usually too small to see easily (otherwise the image would look blocky). Photographs usually use millions of pixels (a megapixel is a million pixels; for example, a screen that is 1080 pixels across and 720 down would contain 777,600 pixels, or 0.7776 megapixels).

The pixels is fundamental to computer graphics, as a lot of the work of computer graphics programmers is taking some abstract idea (such as objects in a scene), and working out the colour each pixel should be to make the viewer think they are looking at the scene. A digital camera also does this - but it just senses the colour falling on each of its millions of sensors, and stores those so that the pixels can be displayed when needed.

In this chapter we'll look at some of the basic techniques that are used to create computer graphics. These will give you an idea of the techniques that are used in graphics programming, although it's just the beginning of what's possible.

For this chapter we are using a system called WebGL which can render 3D graphics in your browser. If your browser is up to date everything should be fine. If you have issues, or if the performance is poor, there is [information here about how to get it going](#).

## 12.2. Graphics transformations

A computer graphics image is just the result of a whole lot of mathematical calculations. In fact, every pixel you see in an image has usually had many calculations made to work out what colour it should be, and there are often millions of pixels in a typical image.

Let's start with some simple but common calculations that are needed for in graphics programming. The following interactive shows a cube with writing on each face. You can move it around using what's called a *transform*, which simply adjusts where it is placed in space. Try typing in 3D coordinates into this interactive to find each code.

Use the interactive online at <http://www.csfieldguide.org.nz/en/interactives/box-translation/index.html>

You've just applied 3D *translation transforms* to the cube. Translation just means moving it in the three dimensions up and down, forward and back, and sideways.

Now try the following challenge, which requires you to rotate the box to find the codes.

Use the interactive online at <http://www.csfieldguide.org.nz/en/interactives/box-rotation/index.html>

There are several transformations that are used in computer graphics, but the most common ones are translation (moving the object), rotation (spinning it) and scaling (changing its size). They come up often in graphics because they are applied not only to objects, but to things like the positions of the camera and lighting sources.

In this section you can apply transformations to various images. We'll start by making the changes manually, one point at a time, but we'll move up to a quick shortcut method that uses a *matrix* to do the work for you. We'll start by looking at how these work in two dimensions - it's a bit easier to think about than three dimensions.

The following interactive shows an arrow, and on the right you can see a list of the points that correspond to its 7 corners. The arrow is on a grid (usually referred to as *cartesian coordinates*), where the centre point is the "zero" point. Points are specified using two numbers, *x* and *y*, usually written as  $(x,y)$ . The *x* value is how far the point is to the right of the centre and the *y* value is how far above the centre it is. For example, the first point in the list is the tip at  $(0,2)$ , which means it's 0 units to the right of the centre (i.e. at the centre), and 2 units above it. Which point does the last pair  $(2,0)$  correspond to? What does it mean if a coordinate has a negative *x* value?

The first list of coordinates is for the original arrow position, and in the second list, you can type in the transformed points to move the arrow --- the original is shown in green and the moved one is in blue.

Click for interactive: changing point

locations

View the link online at [http://csfieldguide.org.nz/releases/1.9.9/\\_static/widgets/CG/CG-arrow/CG-arrow-points.html?info=Your%20first%20challenge%20is%20to%20add%202%20to%20all%20the%20%3Cem%3Ex%3C/em%3E%20points,%20and%203%20to%20all%20the%20%3Cem%3Ey%3C/em%3E%20points%20\(you%20can%20either%20type%20the%20new%20number%20or%20put%20the%20calculation%20in%20the%20box%20e.g.%20%220.5+2%22.%0AWhat%20effect%20does%20this%20have%20on%20the%20original%20arrow?\(Be%20careful%20to%20add%20the%20negative%20numbers%20correctly;%20for%20example,%20adding%202%20to%20-0.5%20gives%201.5.\)%20What%20happens%20if%20you%20subtract%203%20from%20each%20of%20the%20original%20coordinate%20values?%0A&quiz=2%200%200%205%200%202%200%204%200%202%200%200%200%200%201%20&hidetarget=true](http://csfieldguide.org.nz/releases/1.9.9/_static/widgets/CG/CG-arrow/CG-arrow-points.html?info=Your%20first%20challenge%20is%20to%20add%202%20to%20all%20the%20%3Cem%3Ex%3C/em%3E%20points,%20and%203%20to%20all%20the%20%3Cem%3Ey%3C/em%3E%20points%20(you%20can%20either%20type%20the%20new%20number%20or%20put%20the%20calculation%20in%20the%20box%20e.g.%20%220.5+2%22.%0AWhat%20effect%20does%20this%20have%20on%20the%20original%20arrow?(Be%20careful%20to%20add%20the%20negative%20numbers%20correctly;%20for%20example,%20adding%202%20to%20-0.5%20gives%201.5.)%20What%20happens%20if%20you%20subtract%203%20from%20each%20of%20the%20original%20coordinate%20values?%0A&quiz=2%200%200%205%200%202%200%204%200%202%200%200%200%200%201%20&hidetarget=true)

The above transform is called a *translation* --- it translates the arrow around the grid. This kind of transform is used in graphics to specify where an object should be placed in a scene, but it has many other uses, such as making an animated object move along a path, or specifying the position of the imaginary camera (viewpoint).

The next challenge involves changing the size of the image.

Click for interactive:  
scaling

View the link online at [http://csfieldguide.org.nz/releases/1.9.9/\\_static/widgets/CG/CG-arrow/CG-arrow-points.html?info=In%20this%20next%20interactive,%20try%20replacing%20the%20coordinates%20in%20the%20second%20list%20with%20all%20the%20original%20values%20multiplied%20by%202.%20What%20is%20the%20effect%20of%20this%20transform?%20What%20would%20happen%20if%20you%20multiply%20each%20value%20by%2010?%20How%20about%200.5?%20What%20if%20you%20only%20multiply%20the%20%3Cem%3Ex%3C/em%3E%20values?&quiz=0.5%200%200%200%200.5%200%200%200%200%200%200%201%200%200%200%200%201%20&hidetarget=true](http://csfieldguide.org.nz/releases/1.9.9/_static/widgets/CG/CG-arrow/CG-arrow-points.html?info=In%20this%20next%20interactive,%20try%20replacing%20the%20coordinates%20in%20the%20second%20list%20with%20all%20the%20original%20values%20multiplied%20by%202.%20What%20is%20the%20effect%20of%20this%20transform?%20What%20would%20happen%20if%20you%20multiply%20each%20value%20by%2010?%20How%20about%200.5?%20What%20if%20you%20only%20multiply%20the%20%3Cem%3Ex%3C/em%3E%20values?&quiz=0.5%200%200%200%200.5%200%200%200%200%200%200%201%200%200%200%201%20&hidetarget=true)

This transformation is called *scaling*, and although it can obviously be used to control the size of an object, this can in turn be used to create a visual effect such as making the object appear closer or further away.

In the following interactive, try to get the blue arrow to match up with the red one. It will require a mixture of scaling and translation.

Click for interactive: combining scaling and translation  
challenge

View the link online at [http://csfieldguide.org.nz/releases/1.9.9/\\_static/widgets/CG/CG-arrow/CG-arrow-points.html?info=%20Try%20to%20get%20the%20blue%20arrow%20to%20match%20up%20with%20the%20red%20one.%20It%20will%20require%20a%20mixture%20of%20scaling%20and%20translation.&quiz=2%200%200%205%200%202%200%204%200%200%202%200%200%200%200%201](http://csfieldguide.org.nz/releases/1.9.9/_static/widgets/CG/CG-arrow/CG-arrow-points.html?info=%20Try%20to%20get%20the%20blue%20arrow%20to%20match%20up%20with%20the%20red%20one.%20It%20will%20require%20a%20mixture%20of%20scaling%20and%20translation.&quiz=2%200%200%205%200%202%200%204%200%200%202%200%200%200%200%201)

Next, see what happens if you swap the x and y value for each coordinate.

Click for interactive: swapping  
coordinates

View the link online at [http://csfieldguide.org.nz/releases/1.9.9/\\_static/widgets/CG/CG-arrow/CG-arrow-points.html?info=Next,%20see%20what%20happens%20if%20you%20swap%20the%20%Cem%3Ex%3C/em%3E%20and%20%Cem%3Ey%3C/em%3E%20value%20for%20each%20coordinate.&quiz=0%201%200%200%201%200%200%200%200%201%20&hidetarget=true%20&zoom=-5.0](http://csfieldguide.org.nz/releases/1.9.9/_static/widgets/CG/CG-arrow/CG-arrow-points.html?info=Next,%20see%20what%20happens%20if%20you%20swap%20the%20%Cem%3Ex%3C/em%3E%20and%20%Cem%3Ey%3C/em%3E%20value%20for%20each%20coordinate.&quiz=0%201%200%200%201%200%200%200%200%201%20&hidetarget=true%20&zoom=-5.0)

This is a simple *rotation* transformation, also useful for positioning objects in a scene, but also for specifying things like camera angles.

Typing all these coordinates by hand is inefficient. Luckily there's a much better way of achieving all this. Read on!

### 12.2.1. Matrix transformations

There's a much easier way to specify transformations than having to change each coordinate separately. Transformations are usually done in graphics using *matrix* arithmetic, which is a shorthand notation for doing lots of simple arithmetic operations in one go. The matrix for the two-dimensional transformations we've been doing above has four values in it. For the 2 dimensional scaling transform where we made each *x* and *y* value twice as large, the matrix is written as:

$$\begin{bmatrix} 2 & 0 \\ 0 & 2 \end{bmatrix}$$

You can try it out in the following interactive:

Click for interactive: 2D

scaling

View the link online at [http://www.csfieldguide.org.nz/releases/1.9.9/\\_static/widgets/CG/CG-arrow/CG-arrow-matrix.html?info=You%20can%20type%20the%20scaling%20matrix%20into%20this%20interactive%20to%20see%20what%20it%20does%20\(replace%20the%20ones%20with%20twos\).%20The%20top%20left-hand%20value%20just%20means%20multiply%20all%20the%20%Cem%3Ex%3C/em%3E%20values%20by%202,%20and%20the%20bottom%20right%20one%20means%20multiply%20all%20the%20%Cem%3Ey%3C/em%3E%20values%20by%202.%20For%20the%20meantime,%20leave%20the%20translate%20values%20as%2000.&quiz=2%200%200%200%200%202%200%200%200%200%200%200%200%200%200%200%200%200%201](http://www.csfieldguide.org.nz/releases/1.9.9/_static/widgets/CG/CG-arrow/CG-arrow-matrix.html?info=You%20can%20type%20the%20scaling%20matrix%20into%20this%20interactive%20to%20see%20what%20it%20does%20(replace%20the%20ones%20with%20twos).%20The%20top%20left-hand%20value%20just%20means%20multiply%20all%20the%20%Cem%3Ex%3C/em%3E%20values%20by%202,%20and%20the%20bottom%20right%20one%20means%20multiply%20all%20the%20%Cem%3Ey%3C/em%3E%20values%20by%202.%20For%20the%20meantime,%20leave%20the%20translate%20values%20as%2000.&quiz=2%200%200%200%200%202%200%200%200%200%200%200%200%200%200%200%200%200%201)

At this stage you may want to have the interactive open in a separate window so that you can read the text below and work on the interactive at the same time.

Now try changing the matrix to

$$\begin{bmatrix} 3 & 0 \\ 0 & 3 \end{bmatrix}$$

or

$$\begin{bmatrix} 0.2 & 0 \\ 0 & 0.2 \end{bmatrix}$$

The "add translate" values in the interactive are added to each x and y coordinate; experiment with them to see what they do. Now try to find suitable values for these and the matrix to match the arrow up with the red one.

What happens if you use the following matrix?

$$\begin{bmatrix} 2 & 0 \\ 0 & 4 \end{bmatrix}$$

Now try the following matrix:

$$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

This matrix should have rotated the arrow to the right.

A simple way of looking at the matrix is that the top row determines the transformed x value, simply by saying how much of the original x value and y value contribute to the new x value. So in the matrix:

$$\begin{bmatrix} 2 & 0 \\ 0 & 4 \end{bmatrix}$$

The top row just means that the new x value is 2 lots of the original x, and none of the original y, which is why all the x values double. The second row determines the y value: in the above example, it means that the new y value uses none of the original x, but 4 times the original y value. If you try this matrix, you should find that the location of all the x points is doubled, and the location of all the y points is multiplied by 4.

That now explains the  $\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$  matrix. The new x value has none of the original x, but exactly the original y value, and vice versa. This swaps all the x and y coordinates, which is the same as rotating the object to the right.

Where it gets interesting is when you use a little of each value; try the following matrix:

$$\begin{bmatrix} 0.7 & 0.7 \\ -0.7 & 0.7 \end{bmatrix}$$

Now the  $x$  value of each coordinate is a mixture of 0.7 of the original  $x$ , and 0.7 of the original  $y$ . This is called a *rotation*.

In general, to rotate an image by a given angle you need to use the sine (abbreviated sin) and cosine (abbreviated cos) functions from trigonometry. To rotate the image anticlockwise by  $\theta$  degrees, you'll need the following values in the matrix, which rely on trig functions:

$$\begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix}$$

Click for interactive: matrix

rotation

View the link online at [What is the matrix for rotation by 360 degrees?](http://www.csfieldguide.org.nz/releases/1.9.9/_static/widgets/CG/CG-arrow/CG-arrow-matrix.html?info=You%20can%20type%20calculations%20directly%20into%20the%20interactive%20-%20if%20you%20type%20cos(60)%20it%20will%20work%20out%20the%20cosine%20of%2060%20degrees%20for%20you,%20which%20happens%20to%20be%20exactly%200.5.%20Or%20you%20can%20just%20type%20in%20the%20sin%20and%20cosine%20values;%20the%200.7%20numbers%20in%20the%20rotation%20matrix%20are%20just%20the%20values%20for%20sin(45)%20and%20so%20on%20(or%20at%20least,%20they%20approximately%20the%20value;%20it%27s%20actually%200.70710678...,%20which%20happens%20to%20be%20the%20square%20root%20of%200.5,%20but%200.7%20is%20close%20enough%20for%20our%20example).&quiz=0.7%200.7%200%205%20-0.7%200.7%200%204%200%200%201%200%200%200%200%201</a></p>
</div>
<div data-bbox=)

The general matrix for *scaling* is a bit simpler than the one for rotation; if you want to scale by a factor of  $s$ , then you just use the matrix:

$$\begin{bmatrix} s & 0 \\ 0 & s \end{bmatrix}$$

A translation can't be specified by this kind of matrix, so in the interactives we've provided an extra place to specify an  $x$  and  $y$  value to translate the input. Try it out in the following interactive.

Click for interactive: translation  
challenge

View the link online at [http://www.csfieldguide.org.nz/releases/1.9.9/\\_static/widgets/CG/CG-arrow/CG-arrow-matrix.html?info=Try%20translating%20the%20original%20arrow%20so%20that%20it%20matches%20up%20with%20the%20red%20arrow.&quiz=1%200%200%205%200%201%200%204%200%200%201%200%200%200%200%201](http://www.csfieldguide.org.nz/releases/1.9.9/_static/widgets/CG/CG-arrow/CG-arrow-matrix.html?info=Try%20translating%20the%20original%20arrow%20so%20that%20it%20matches%20up%20with%20the%20red%20arrow.&quiz=1%200%200%205%200%201%200%204%200%200%201%200%200%200%200%201)

The next interactive needs you to combine translation with scaling.

Click for interactive: scaling and translation  
challenge

View the link online at [http://www.csfieldguide.org.nz/releases/1.9.9/\\_static/widgets/CG/CG-arrow/CG-arrow-matrix.html?info=Now%20try%20to%20scale%20the%20original%20arrow%20in%20the%20following,%20and%20translate%20it%20to%20match%20the%20red%20arrow.&quiz=2%200%200%204%200%202%200%203%200%200%202%200%200%200%200%201](http://www.csfieldguide.org.nz/releases/1.9.9/_static/widgets/CG/CG-arrow/CG-arrow-matrix.html?info=Now%20try%20to%20scale%20the%20original%20arrow%20in%20the%20following,%20and%20translate%20it%20to%20match%20the%20red%20arrow.&quiz=2%200%200%204%200%202%200%203%200%200%202%200%200%200%200%201)

The order in which translation and scaling happen makes a difference. Try the following challenge!

Click for interactive: translation before  
scaling

View the link online at [http://www.csfieldguide.org.nz/releases/1.9.9/\\_static/widgets/CG/CG-arrow/CG-arrow-matrix-reversed.html?info=The%20following%20interactive%20has%20the%20translation%20and%20scaling%20the%20other%20way%20around.%20Use%20this%20one%20to%20transform%20the%20blue%20arrow%20to%20the%20red%20arrow.%20The%20order%20in%20which%20the%20operations%20happen%20makes%20a%20difference!&quiz=2%200%200%205%200%202%200%204%200%200%202%200%200%200%200%201](http://www.csfieldguide.org.nz/releases/1.9.9/_static/widgets/CG/CG-arrow/CG-arrow-matrix-reversed.html?info=The%20following%20interactive%20has%20the%20translation%20and%20scaling%20the%20other%20way%20around.%20Use%20this%20one%20to%20transform%20the%20blue%20arrow%20to%20the%20red%20arrow.%20The%20order%20in%20which%20the%20operations%20happen%20makes%20a%20difference!&quiz=2%200%200%205%200%202%200%204%200%200%202%200%200%200%200%201)

In the above, you'll have noticed that scaling is affected by how far the object is from the centre. If you want to scale around a fixed point in the object (so it expands where it is), then an easy way is to translate it back to the centre (also called the *origin*), scale it, and then translate it back to where it was. The following interactive allows you to move the arrow, then scale it, and move it back.

Click for interactive: using translation to simplify  
scaling

View the link online at [http://www.csfieldguide.org.nz/releases/1.9.9/\\_static/widgets/CG/CG-arrow/CG-arrow-transmatrix.html?info=The%20tip%20is%20at%20\(-8,7\),%20so%20you%20should%20translate%20it%20to%20\(0,0\),%20scale%20by%202,%20and%20translate%20back%20to%20\(-8,%207\).&zoom=-15.0%20&quiz=2%200%200%20-8%200%202%200%203%200%200%202%200%200%200%200%201%20&start=1%200%200%20-8%200%201%200%205%200%200%201%200%200%200%200%201%20&allPrize=5](http://www.csfieldguide.org.nz/releases/1.9.9/_static/widgets/CG/CG-arrow/CG-arrow-transmatrix.html?info=The%20tip%20is%20at%20(-8,7),%20so%20you%20should%20translate%20it%20to%20(0,0),%20scale%20by%202,%20and%20translate%20back%20to%20(-8,%207).&zoom=-15.0%20&quiz=2%200%200%20-8%200%202%200%203%200%200%202%200%200%200%200%201%20&start=1%200%200%20-8%200%201%200%205%200%200%201%200%200%200%200%201%20&allPrize=5)

The same problem comes up with rotation. The following interactive allows you to use a translation first to make the scaling more predictable.

Click for interactive: using translation to simplify  
rotation

View the link online at [http://www.csfieldguide.org.nz/releases/1.9.9/\\_static/widgets/CG/CG-arrow/CG-arrow-transmatrix.html?info=Try%20rotating%20this%20image%20by%2045%20degrees.%20You%27ll%20need%20to%20translate%20the%20tip%20to%20the%20origin,%20apply%20the%20rotation,%20and%20translate%20it%20back.&zoom=-10.0%20&quiz=0.699999988079071%200.699999988079071%200%204.400000095367432%20-0.699999988079071%200.699999988079071%200%204.599999904632568%200%200%201%200%200%200%200%200%201%20&start=1%200%200%20-3%200%201%200%204%200%200%201%200%200%200%201%20&allPrize=5](http://www.csfieldguide.org.nz/releases/1.9.9/_static/widgets/CG/CG-arrow/CG-arrow-transmatrix.html?info=Try%20rotating%20this%20image%20by%2045%20degrees.%20You%27ll%20need%20to%20translate%20the%20tip%20to%20the%20origin,%20apply%20the%20rotation,%20and%20translate%20it%20back.&zoom=-10.0%20&quiz=0.699999988079071%200.699999988079071%200%204.400000095367432%20-0.699999988079071%200.699999988079071%200%204.599999904632568%200%200%201%200%200%200%200%201%20&start=1%200%200%20-3%200%201%200%204%200%200%201%200%200%200%201%20&allPrize=5)

The following two examples combine rotation, scaling and translation. You can use multiple matrices (that's the plural of matrix) to match up the target object --- the product of each matrix becomes the input to the next one. Oh, and the arrow is twice as fat, but still the same height (from base to tip).

Click for interactive: combining translation, scaling and rotation

View the link online at [http://www.csfieldguide.org.nz/releases/1.9.9/\\_static/widgets/CG/CG-arrow/CG-arrow-doublematrix.html?info=Try%20matching%20the%20blue%20arrow%20to%20the%20red%20one%20using%20two%20matrices%20\(one%20to%20scale%20and%20one%20to%20rotate\),%20and%20adding%20a%20vector.&zoom=-10.0%20&quiz=0%201%200%204%202%200%200%20-2%200%200%201%200%200%200%201%20&allPrize=5](http://www.csfieldguide.org.nz/releases/1.9.9/_static/widgets/CG/CG-arrow/CG-arrow-doublematrix.html?info=Try%20matching%20the%20blue%20arrow%20to%20the%20red%20one%20using%20two%20matrices%20(one%20to%20scale%20and%20one%20to%20rotate),%20and%20adding%20a%20vector.&zoom=-10.0%20&quiz=0%201%200%204%202%200%200%20-2%200%200%201%200%200%200%201%20&allPrize=5)

Here's another challenge combining all three transformations:

Click for interactive: multiple transformation challenge

View the link online at [http://www.csfieldguide.org.nz/releases/1.9.9/\\_static/widgets/CG/CG-arrow/CG-arrow-doublematrix.html?info=You%20will%20need%20to%20use%20all%20three%20operations%20to%20do%20this%20next%20one.&zoom=-6.0%20&quiz=0.3499999940395355%20-0.3499999940395355%200%20-1%200.3499999940395355%200.3499999940395355%200%20-2%200%200%201%200%200%200%201%20&allPrize=5](http://www.csfieldguide.org.nz/releases/1.9.9/_static/widgets/CG/CG-arrow/CG-arrow-doublematrix.html?info=You%20will%20need%20to%20use%20all%20three%20operations%20to%20do%20this%20next%20one.&zoom=-6.0%20&quiz=0.3499999940395355%20-0.3499999940395355%200%20-1%200.3499999940395355%200.3499999940395355%200%20-2%200%200%201%200%200%200%201%20&allPrize=5)

These combined transformations are common, and they might seem like a lot of work because each matrix has to be applied to every point in an object. Our arrows only had 7 points, but complex images can have thousands or even millions of points in them. Fortunately we can combine all the matrix operations in advance to give just one operation to apply to each point.

## 12.2.2. Combining transformations

Several transforms being applied to the same image can be made more efficient by creating one matrix that has the effect of all the transforms combined. The combination is done by "multiplying" all the matrices.

Multiplying two matrices can't be done by just multiplying the corresponding elements; if you are multiplying two matrices with the  $a$  and  $b$  values shown below, the resulting values from the multiplication are calculated as follows:

$$\begin{bmatrix} a_{11} & a_{21} \\ a_{12} & a_{22} \end{bmatrix} \times \begin{bmatrix} b_{11} & b_{21} \\ b_{12} & b_{22} \end{bmatrix} = \begin{bmatrix} a_{11}b_{11} + a_{21}b_{12} & a_{11}b_{21} + a_{21}b_{22} \\ a_{12}b_{11} + a_{22}b_{12} & a_{12}b_{21} + a_{22}b_{22} \end{bmatrix}$$

It's a bit complicated, but this calculation is only done once to work out the combined transformation, and it gives you a single matrix that will provide two transforms in one operation.

As a simple example, consider what happens when you scale by 2 and then rotate by 45 degrees. The two matrices to multiply work out like this:

$$\begin{bmatrix} 2 & 0 \\ 0 & 2 \end{bmatrix} \times \begin{bmatrix} 0.7 & 0.7 \\ -0.7 & 0.7 \end{bmatrix} = \begin{bmatrix} 2 \times 0.7 + 0 \times -0.7 & 2 \times 0.7 + 0 \times 0.7 \\ 0 \times 0.7 + 2 \times -0.7 & 0 \times 0.7 + 2 \times 0.7 \end{bmatrix} = \begin{bmatrix} 1.4 & 1.4 \\ -1.4 & 1.4 \end{bmatrix}$$

You can put the matrix we just calculated into the following interactive to see if it does indeed scale by 2 and rotate 45 degrees.

Click for interactive: check a single  
matrix

View the link online at [http://www.csfieldguide.org.nz/releases/1.9.9/\\_static/widgets/CG/CG-arrow/CG-arrow-singlematrix.html?info=Try%20putting%20in%20the%20final%20matrix%20here%20and%20see%20if%20it%20does%20scale%20by%202%20and%20rotate%20by%2045%20degrees.&zoom=-10.0%20&quiz=1.4%201.4%200%200%20-1.4%201.4%200%200%200%201%200%200%200%200%201%20&allPrize=5](http://www.csfieldguide.org.nz/releases/1.9.9/_static/widgets/CG/CG-arrow/CG-arrow-singlematrix.html?info=Try%20putting%20in%20the%20final%20matrix%20here%20and%20see%20if%20it%20does%20scale%20by%202%20and%20rotate%20by%2045%20degrees.&zoom=-10.0%20&quiz=1.4%201.4%200%200%20-1.4%201.4%200%200%200%201%200%200%200%200%201%20&allPrize=5)

Now try making up your own combination of transforms to see if they give the result you expect. In this interactive you can drag the matrices to change their order.

Click for interactive: multiple  
matrices

View the link online at [http://www.csfieldguide.org.nz/releases/1.9.9/\\_static/widgets/CG/CG-arrow/CG-arrow-multiply2matrix.html?info=Now%20try%20multiplying%20two%20other%20transform%20matrices%20that%20you%20make%20up%20yourself,%20and%20see%20if%20they%20produce%20the%20expected%20result.&zoom=-10.0%20&quiz=1.4%201.4%200%200%20-1.4%201.4%200%200%200%201%200%200%200%201%20&allPrize=5](http://www.csfieldguide.org.nz/releases/1.9.9/_static/widgets/CG/CG-arrow/CG-arrow-multiply2matrix.html?info=Now%20try%20multiplying%20two%20other%20transform%20matrices%20that%20you%20make%20up%20yourself,%20and%20see%20if%20they%20produce%20the%20expected%20result.&zoom=-10.0%20&quiz=1.4%201.4%200%200%20-1.4%201.4%200%200%200%201%200%200%200%201%20&allPrize=5)

In computer graphics systems there can be many transformations combined, and this is done by multiplying them all together (two at a time) to produce one matrix that does all the transforms in one go. That transform might then be applied to millions of points, so the time taken to do the matrix multiplication at the start will pay off well.

The project below gives the chance to explore combining matrices, and has an interactive that will calculate the multiplied matrices for you.

### 12.2.3. 3D transforms

So far we've just done the transforms in two dimensions. To do this in 3D, we need a z coordinate as well, which is the depth of the object into the screen. A matrix for operating on 3D points is 3 by 3. For example, the 3D matrix for doubling the size of an object is as follows; it multiplies each of the x, y and z values of a point by 2.

$$\begin{bmatrix} 2 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 2 \end{bmatrix}$$

You can try out this 3D matrix in the following interactive.

Click for interactive: 3D transform

matrix

View the link online at [http://www.csfieldguide.org.nz/releases/1.9.9/\\_static/widgets/CG/CG-mini-editor/main%20\(cutdown\).html?info=%0AIn%20this%20interactive,%20try%20changing%20the%20scaling%20on%20the%20image%20\(it%20starts%20with%20a%20scaling%20factor%20of%2010%20in%20all%20three%20dimensions\).](http://www.csfieldguide.org.nz/releases/1.9.9/_static/widgets/CG/CG-mini-editor/main%20(cutdown).html?info=%0AIn%20this%20interactive,%20try%20changing%20the%20scaling%20on%20the%20image%20(it%20starts%20with%20a%20scaling%20factor%20of%2010%20in%20all%20three%20dimensions).)

The above image mesh has 3644 points in it, and your matrix was applied to each one of them to work out the new image.

The next interactive allows you to do translation (using a vector). Use it to get used to translating in the three dimensions (don't worry about using matrices this time.)

Click for interactive: 3D

translation

View the link online at [http://www.csfieldguide.org.nz/releases/1.9.9/\\_static/widgets/CG/CG-mini-editor/main%20\(cutdown\).html?info=%0ATranslation%20requires%203%20values,%20which%20are%20added%20to%20the%20\\*x\\*,%20\\*y\\*%20and%20\\*z\\*%20coordinates%20of%20each%20point%20in%20an%20object.%3Cp%3EIn%20the%20following%20interactive,%20try%20moving%20the%20teapot%20left%20and%20right%20\(%20%3Cem%3Ex%3C/em%3E%20\),%20up%20and%20down%20\(%20%3Cem%3Ey%3C/em%3E%20\),%20and%20in%20and%20out%20of%20the%20screen%20\(%20%3Cem%3Ez%3C/em%3E%20\)%20by%20adding%20a%20E2%80%9Cvector%E2%80%9D%20to%20the%20operations.%20Then%20try%20combining%20all%20three.%3C/p%3E%0A](http://www.csfieldguide.org.nz/releases/1.9.9/_static/widgets/CG/CG-mini-editor/main%20(cutdown).html?info=%0ATranslation%20requires%203%20values,%20which%20are%20added%20to%20the%20*x*,%20*y*%20and%20*z*%20coordinates%20of%20each%20point%20in%20an%20object.%3Cp%3EIn%20the%20following%20interactive,%20try%20moving%20the%20teapot%20left%20and%20right%20(%20%3Cem%3Ex%3C/em%3E%20),%20up%20and%20down%20(%20%3Cem%3Ey%3C/em%3E%20),%20and%20in%20and%20out%20of%20the%20screen%20(%20%3Cem%3Ez%3C/em%3E%20)%20by%20adding%20a%20E2%80%9Cvector%E2%80%9D%20to%20the%20operations.%20Then%20try%20combining%20all%20three.%3C/p%3E%0A)

Rotation is trickier because you can now rotate in different directions. In 2D rotations were around the centre (origin) of the grid, but in 3D rotations are around a line (either the horizontal x-axis, the vertical y-axis, or the z-axis, which goes into the screen!)

The 2D rotation we used earlier can be applied to 3 dimensions using this matrix:

$$\begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Try applying that to the image above. This is rotating around the z-axis (a line going into the screen); that is, it's just moving the image around in the 2D plane. It's really the same as the rotation we used previously, as the last line (0, 0, 1) just keeps the z point the same.

Try the following matrix, which rotates around the x-axis (notice that the x value always stays the same because of the 1,0,0 in the first line):

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\theta) & -\sin(\theta) \\ 0 & \sin(\theta) & \cos(\theta) \end{bmatrix}$$

And this one for the y-axis:

$$\begin{bmatrix} \cos(\theta) & 0 & \sin(\theta) \\ 0 & 1 & 0 \\ -\sin(\theta) & 0 & \cos(\theta) \end{bmatrix}$$

The following interactive allows you to combine 3D matrices.

Click for interactive: 3D with multiple matrices and vectors

View the link online at [In the above examples, when you have several matrices being applied to every point in the image, a lot of time can be saved by converting the series of matrices and transforms to just one formula that does all of the transforms in one go. The following interactive can do those calculations for you.](http://www.csfieldguide.org.nz/releases/1.9.9/_static/widgets/CG/CG-mini-editor/main%20(cutdown).html?info=%0AYou%20can%20experiment%20with%20moving%20the%20teapot%20around%20in%20space,%20changing%20its%20size,%20and%20angle.%3Cdl%20class=%22docutils%22%3E%0A%3Cdt%3EThink%20about%20the%20order%20in%20which%20you%20need%20to%20combine%20the%20transforms%20to%20get%20a%20particular%20image%20that%20you%20want.%3C/dt%3E%0A%3Cdd%3EFor%20example,%20if%20you%20translate%20an%20image%20and%20then%20scale%20it,%20you%E2%80%99ll%20get%20a%20different%20effect%20to%20scaling%20it%20then%20translating%20it.%0Alf%20you%20want%20to%20rotate%20or%20scale%20around%20a%20particular%20point,%20you%20can%20do%20this%20in%20three%20steps%20(as%20with%20the%202D%20case%20above):%20(1)%20translate%20the%20object%20so%20that%20the%20point%20you%20want%20to%20scale%20or%20rotate%20around%20is%20the%20origin%20(where%20the%20x,%20y%20and%20z%20axes%20meet),%20(2)%20do%20the%20scaling/rotation,%20(3)%20translate%20the%20object%20back%20to%20where%20it%20was.%20If%20you%20just%20scale%20an%20object%20where%20it%20is,%20its%20distance%20from%20the%20origin%20will%20also%20be%20scaled%20up.%3C/dd%3E%0A%3C/dl%3E%0A</p>
</div>
<div data-bbox=)

For example, in the following interactive, type in the matrix for doubling the size of an object (put the number 2 instead of 1 on the main diagonal values), then add another matrix that triples the size of the image (3 on the main diagonal). The interactive shows a matrix on the right that combines the two --- does it look right?

Click for interactive: matrix simplifier

View the link online at [http://www.csfieldguide.org.nz/releases/1.9.9/\\_static/widgets/CG/CG-matrix-simplifier/CG-matrix-simplifier.html?info=Multiple%20transforms](http://www.csfieldguide.org.nz/releases/1.9.9/_static/widgets/CG/CG-matrix-simplifier/CG-matrix-simplifier.html?info=Multiple%20transforms)

The interactive also allows you to combine transformations (just three numbers, for x, y and z). Try combining a scaling followed by a translation. What if you add a rotation --- does the order matter?

### Curiosity: Matrix multiplication in 3D

In case you're wondering, the interactive is using the following formula to combine two matrices (you don't have to understand this to use it). It is called matrix multiplication, and while it might be a bit tricky, it's very useful in computer graphics because it reduces all the transformations you need to just one matrix, which is then applied to every point being transformed. This is way better than having to run all the matrices of every point.

$$\begin{bmatrix} a_{11} & a_{21} & a_{31} \\ a_{12} & a_{22} & a_{32} \\ a_{13} & a_{23} & a_{33} \end{bmatrix} \times \begin{bmatrix} b_{11} & b_{21} & b_{31} \\ b_{12} & b_{22} & b_{32} \\ b_{13} & b_{23} & b_{33} \end{bmatrix} = \begin{bmatrix} a_{11}b_{11} + a_{21}b_{12} + a_{31}b_{13} & a_{11}b_{21} + a_{21}b_{22} + a_{31}b_{23} & a_{11}b_{31} + a_{21}b_{32} + a_{31}b_{33} \\ a_{12}b_{11} + a_{22}b_{12} + a_{32}b_{13} & a_{12}b_{21} + a_{22}b_{22} + a_{32}b_{23} & a_{12}b_{31} + a_{22}b_{32} + a_{32}b_{33} \\ a_{13}b_{11} + a_{23}b_{12} + a_{33}b_{13} & a_{13}b_{21} + a_{23}b_{22} + a_{33}b_{23} & a_{13}b_{31} + a_{23}b_{32} + a_{33}b_{33} \end{bmatrix}$$

### Project: 3D transforms

For this project, you will demonstrate what you've learned in the section above by explaining a 3D transformation of a few objects. You should take screenshots of each step to illustrate the process for your report.

The following scene-creation interactive allows you to choose objects (and their colours etc.), and apply one transformation to them. To position them more interestingly, you will need to come up with multiple transformations (e.g. scale, then rotate, then translate), and use the "simplifier" interactive to combine all the matrices into one operation.

The scene-creation interactive can be run from here:

Click for interactive: scene  
creation

View the link online at [http://www.csfieldguide.org.nz/releases/1.9.9/\\_static/widgets/CG/CG-mini-editor/main.html?info=Multiple%20transforms](http://www.csfieldguide.org.nz/releases/1.9.9/_static/widgets/CG/CG-mini-editor/main.html?info=Multiple%20transforms)

To generate combined transformations, you can use the following transform simplifier interactive:

Click for interactive: matrix  
simplifier

View the link online at [http://www.csfieldguide.org.nz/releases/1.9.9/\\_static/widgets/CG/CG-matrix-simplifier/CG-matrix-simplifier.html?info=Multiple%20transforms](http://www.csfieldguide.org.nz/releases/1.9.9/_static/widgets/CG/CG-matrix-simplifier/CG-matrix-simplifier.html?info=Multiple%20transforms)

Because you can't save your work in the interactives, keep notes and screen shots as you go along. These will be useful for your report, and also can be used if you need to start over again.

Introduce your project with examples of 3D images, and how they are used (perhaps from movies or scenes that other people have created). Describe any innovations in the particular image (e.g. computer generated movies usually push the boundaries of what was previously possible, so discuss what boundaries were moved by a particular movie, and who wrote the programs to achieve the new effects). One way to confirm that a movie is innovative in this area is if it has won an award for the graphics software.

To show the basics of computer graphics, try putting a few objects in a particular arrangement (e.g. with the teapot sitting beside some cups), and explain the transforms needed to achieve this, showing the matrices needed.

Give simple examples of translation, scaling *and* rotation using your scene. You should include multiple transforms applied to one object, and show how they can be used to position an object.

Show how the matrices for a series of transforms can be multiplied together to get one matrix that applies all the transforms at once.

Discuss how the single matrix derived from all the others is more efficient, using your scene as an example to explain this.

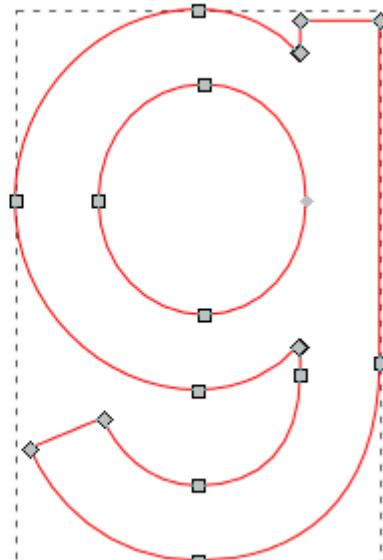
### Project: WebGL and OpenGL

If you're confident with programming and want to explore graphics at a more practical level, you could do a similar project to the previous one using a graphics programming system such as [WebGL](#) (which is the system used in the demonstrations above), or a widely used graphics system such as [OpenGL](#). There is an interactive tutorial on OpenGL called [JPOT](#).

Note that these project can be very time consuming because these are powerful systems, and there is quite a bit of detail to get right even for a simple operation.

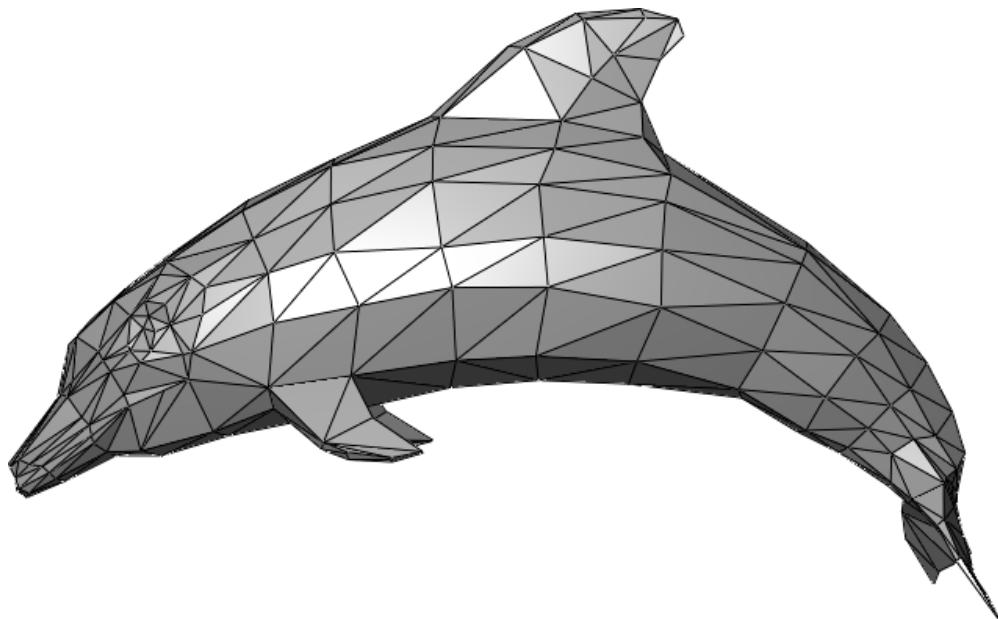
## 12.3. Drawing lines and circles

A fundamental operation in computer graphics is to draw lines and circles. For example, these are used as the components of scalable fonts and vector graphics; the letter "g" is specified as a series of lines and curves, so that when you zoom in on it the computer can redraw it at whatever resolution is needed. If the system only stored the pixels for the letter shape, then zooming in would result in a low quality image.



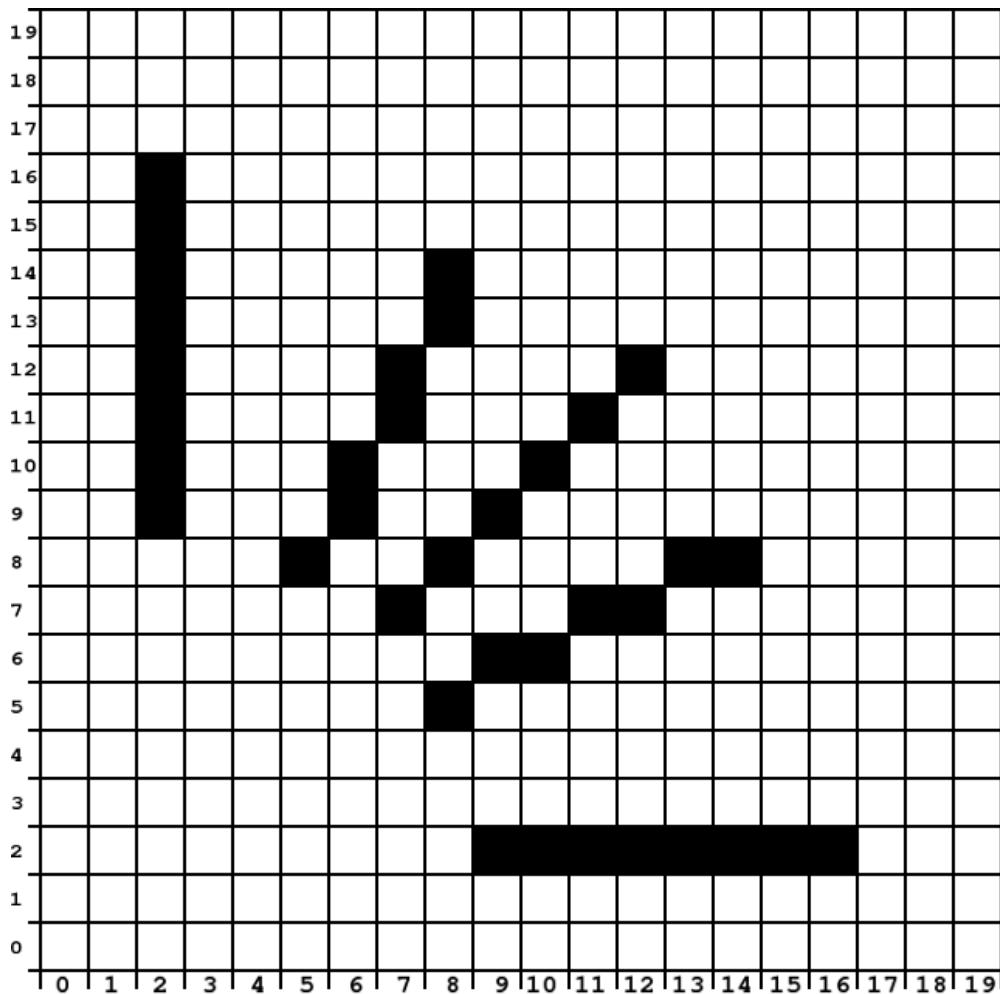
The points used to create the letter 'g' in Google's logo

In 3D graphics shapes are often stored using lines and curves that mark out the edges of tiny flat surfaces (usually triangles), each of which is so small that you can't see them unless you zoom right in.



The lines and circles that specify an object are usually given using numbers (for example, a line between a given starting and finishing position or a circle with a given centre and radius). From this a graphics program must calculate which pixels on the screen should be coloured in to represent the line or circle, or it may just need to work out where the line is without drawing it.

For example, here's a grid of pixels with 5 lines shown magnified. The vertical line would have been specified as going from pixel (2,9) to (2,16) --- that is, starting 2 across and 9 up, and finishing 2 across and 16 up. Of course, this is only a small part of a screen, as normally they are more like 1000 by 1000 pixels or more; even a smartphone can be hundreds of pixels high and wide.



These are things that are easy to do with pencil and paper using a ruler and compass, but on a computer the calculations need to be done for every pixel, and if you use the wrong method then it will take too long and the image will be displayed slowly or a live animation will appear jerky. In this section we will look into some very simple but clever algorithms that enable a computer to do these calculations very quickly.

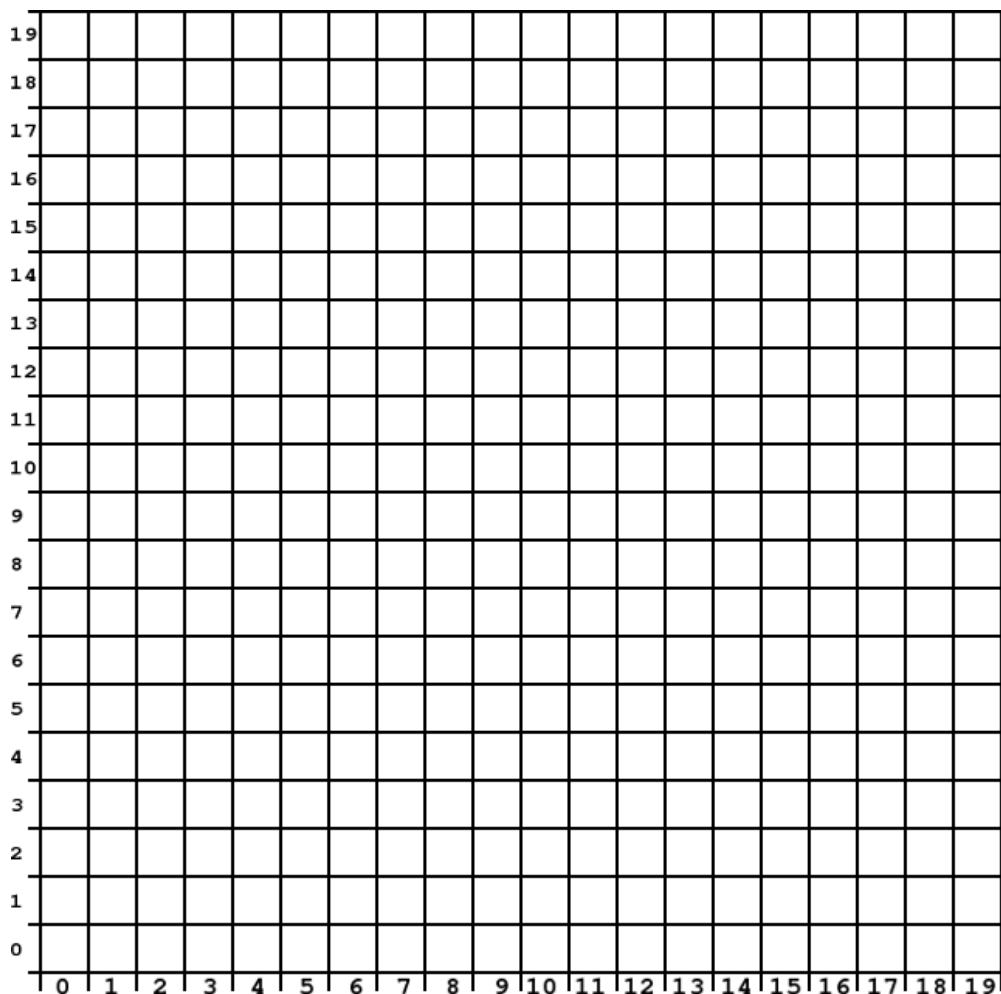
### 12.3.1. Line drawing

To draw a line, a computer must work out which pixels need to be filled so that the line looks straight. You can try this by colouring in squares on a grid, such as the one below (they are many times bigger than the pixels on a normal printer or screen). We'll identify the pixels on the grid using two values,  $(x,y)$ , where  $x$  is the distance across from the left, and  $y$

is the distance up from the bottom. The bottom left pixel below is (0,0), and the top right one is (19,19).

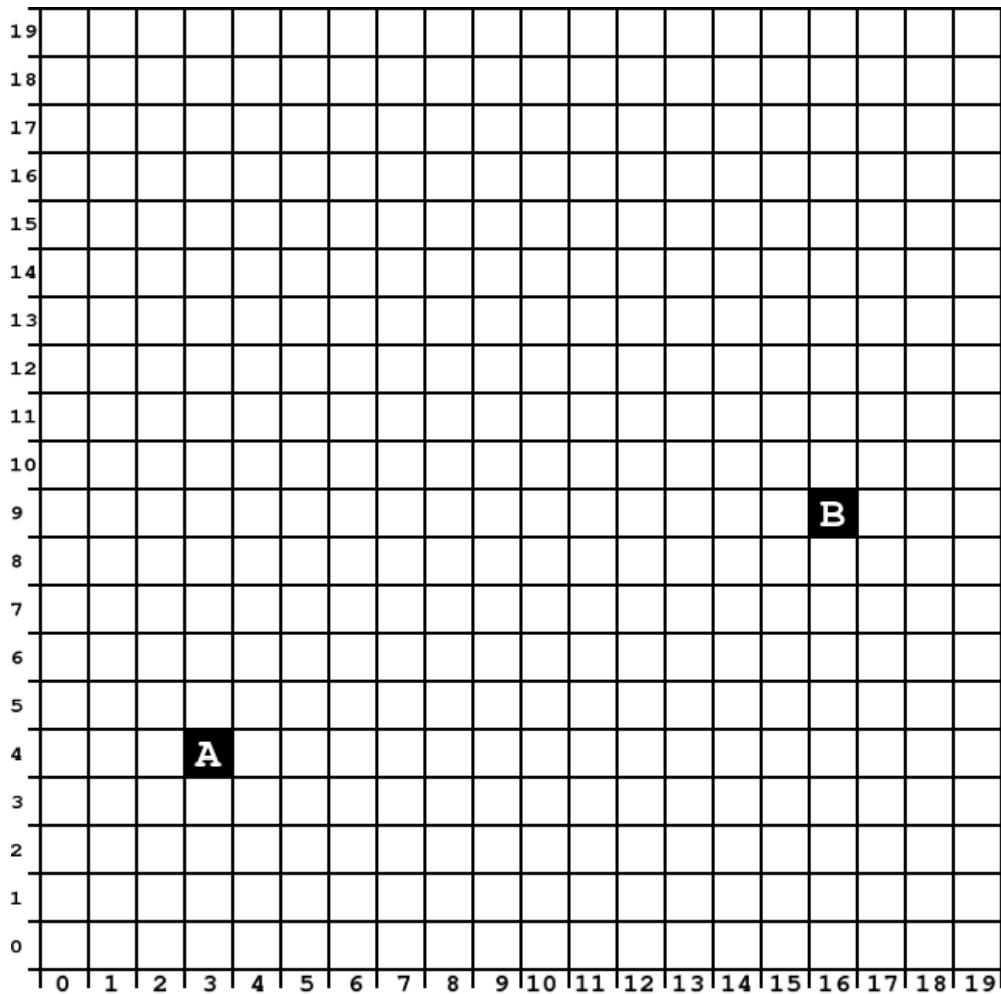
On the following grid, try to draw these straight lines by filling in pixels in the grid:

- from (2, 17) to (10, 17)
- from (18, 2) to (18, 14)
- from (1, 5) to (8, 12)



Drawing a horizontal, vertical or diagonal line like the ones above is easy; it's the ones at different angles that require some calculation.

Without using a ruler, can you draw a straight line from A to B on the following grid by colouring in pixels?



Once you have finished drawing your line, try checking it with a ruler. Place the ruler so that it goes from the centre of A to the centre of B. Does it cross all of the pixels that you have coloured?

### 12.3.2. Using a formula to draw a line

The mathematical formula for a line is  $y = mx + c$ . This gives you the  $y$  value for each  $x$  value across the screen, and you get to specify two things: the **slope** of the line, which is  $m$ , and where the line crosses the  $y$  axis, which is  $c$ . In other words, when you are  $x$  pixels across the screen with your line, the pixel to colour in would be  $(x, mx + c)$ .

For example, choosing  $m = 2$  and  $c = 3$  means that the line would go through the points  $(0,3)$ ,  $(1,5)$ ,  $(2,7)$ ,  $(3,9)$  and so on. This line goes up 2 pixels for every one across  $m = 2$ , and crosses the  $y$  axis 3 pixels up ( $c = 3$ ).

You should experiment with drawing graphs for various values of  $m$  and  $c$  (for example, start with  $c = 0$ , and try these three lines:  $m = 1$ ,  $m = 0.5$  and  $m = 0$ ) by putting in the values. What angle are these lines at?

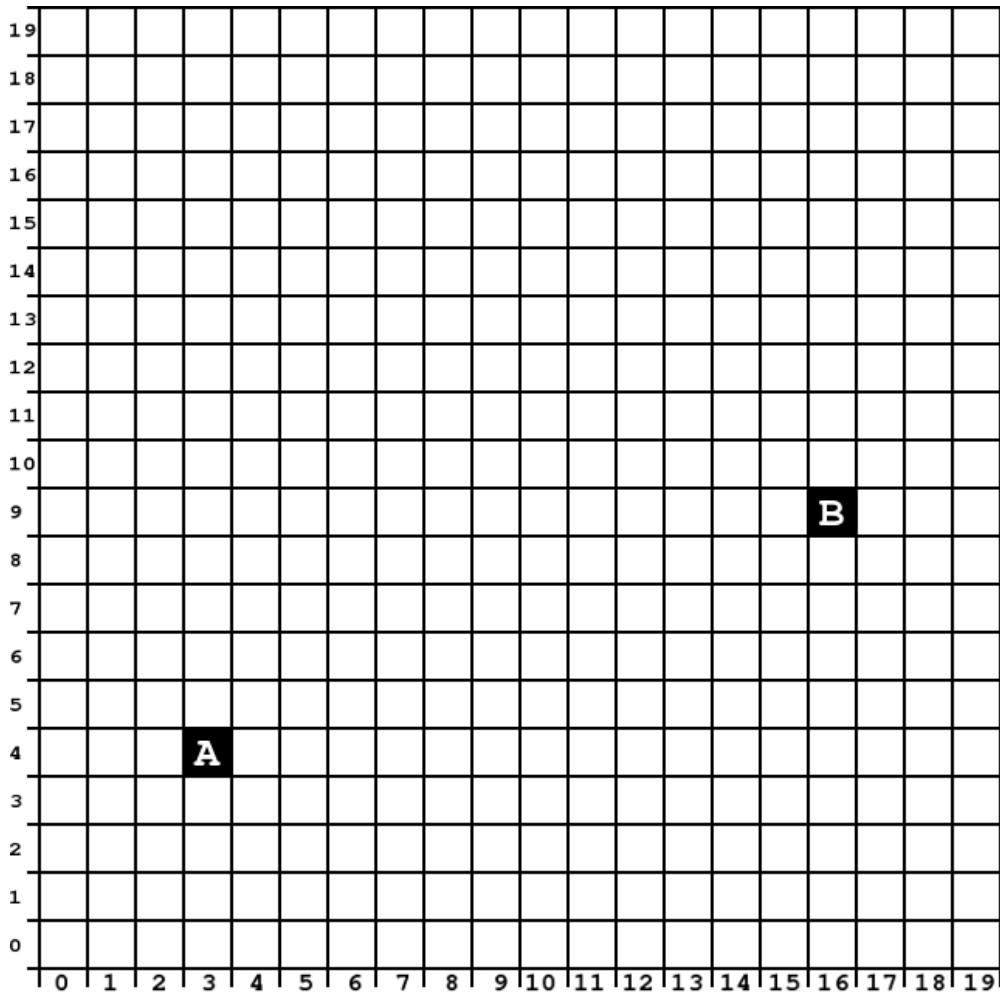
The  $mx + c$  formula can be used to work out which pixels should be coloured in for a line that goes between  $(x_1, y_1)$  and  $(x_2, y_2)$ . What are  $(x_1, y_1)$  and  $(x_2, y_2)$  for the points A and B on the grid below?

See if you can work out the  $m$  and  $b$  values for a line from A to B, or you can calculate them using the following formulas:

$$m = \frac{(y_2 - y_1)}{(x_2 - x_1)}$$

$$b = \frac{(y_1x_2 - y_2x_1)}{(x_2 - x_1)}$$

Now draw the same line as in the previous section (between A and B) using the formula  $y = mx + c$  to calculate  $y$  for each value of  $x$  from  $x_1$  to  $x_2$  (you will need to round  $y$  to the nearest integer to work out which pixel to colour in). If the formulas have been applied correctly, the  $y$  value should range from  $y_1$  to  $y_2$ .



Once you have completed the line, check it with a ruler. How does it compare to your first attempt?

Now consider the number of calculations that are needed to work out each point. It won't seem like many, but remember that a computer might be calculating hundreds of points on thousands of lines in a complicated image. Although this formula works fine, it's too slow to generate the complex graphics needed for good animations and games. In the next section we will explore a method that greatly speeds this up.

### 12.3.3. Bresenham's Line Algorithm

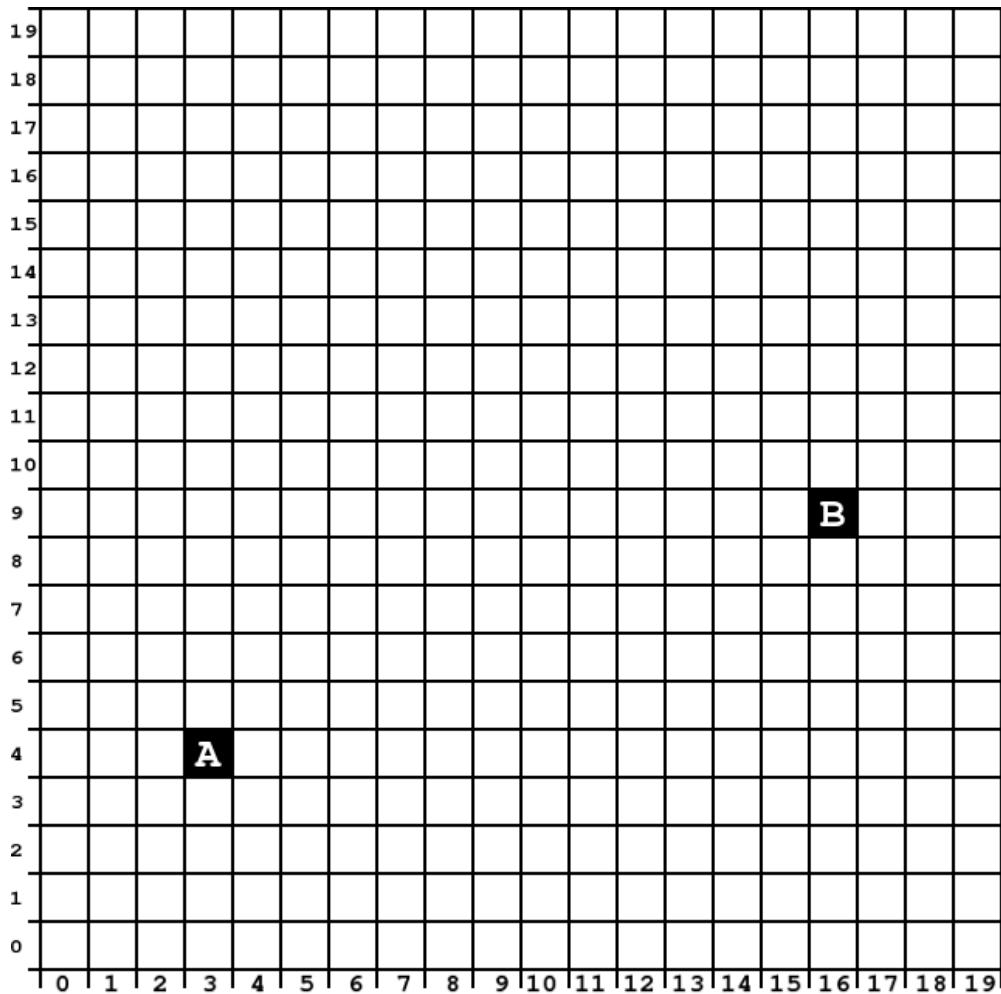
A faster way for a computer to calculate which pixels to colour in is to use Bresenham's Line Algorithm. It follows these simple rules. First, calculate these three values:

$$\begin{aligned} A &= 2 \times (y_2 - y_1) \\ B &= A - 2 \times (x_2 - x_1) \\ P &= A - (x_2 - x_1) \end{aligned}$$

To draw the line, fill the starting pixel, and then for every position along the x axis:

- if  $P$  is less than 0, draw the new pixel on the same line as the last pixel, and add  $A$  to  $P$ .
- if  $P$  was 0 or greater, draw the new pixel one line higher than the last pixel, and add  $B$  to  $P$ .
- repeat this decision until we reach the end of the line.

Without using a ruler, use Bresenham's Line Algorithm to draw a straight line from A to B:

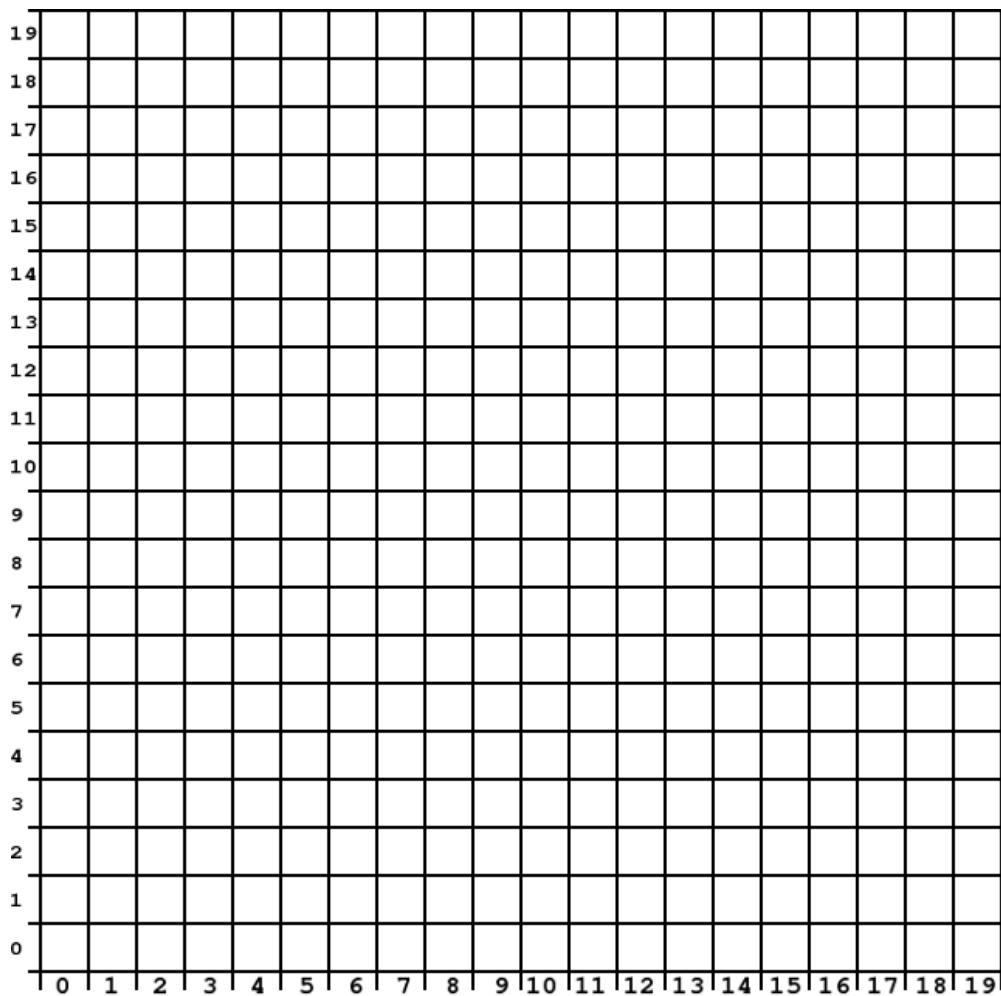


Once you have completed the line, check it with a ruler. How does it compare to the previous attempts?

#### 12.3.4. Lines at other angles

So far the version of Bresenham's line drawing algorithm that you have used only works for lines that have a gradient (slope) between 0 and 1 (that is, from horizontal to 45 degrees). To make this algorithm more general, so that it can be used to draw any line, some additional rules are needed:

- If a line is sloping downward instead of sloping upward, then when P is 0 or greater, draw the next column's pixel one row *below* the previous pixel, instead of above it.
- If the change in *y* value is greater than the change in *x* value (which means that the slope is more than 1), then the calculations for A, B, and the initial value for P will need to be changed. When calculating A, B, and the initial P, use X where you previously would have used Y, and vice versa. When drawing pixels, instead of going across every column in the X axis, go through every row in the Y axis, drawing one pixel per row.



In the grid above, choose two points of your own that are unique to you. Don't choose points that will give horizontal, vertical or diagonal lines!

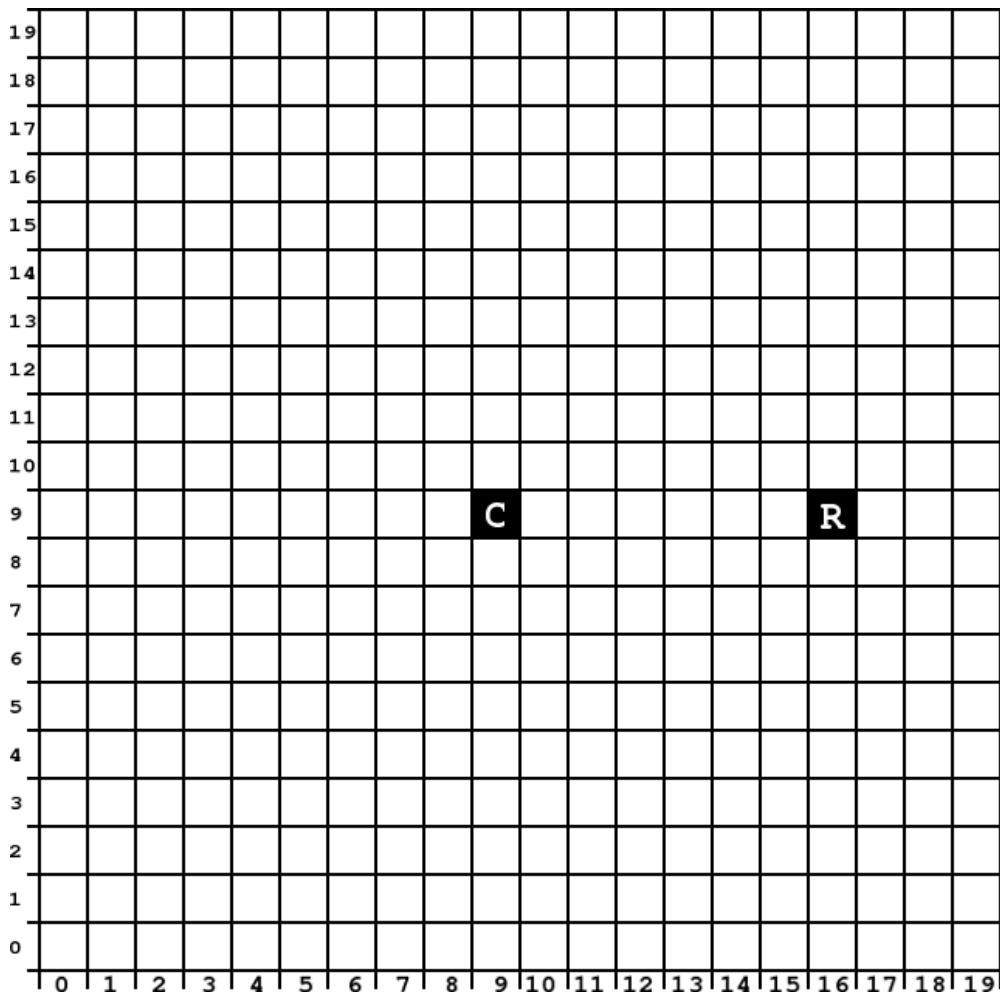
Now use Bresenham's algorithm to draw the line. Check that it gives the same points as you would have chosen using a ruler, or using the formula  $y = mx + b$ . How many arithmetic calculations (multiplications and additions) were needed for Bresenham's algorithm? How many would have been needed if you used the  $y = mx + b$  formula? Which is faster (bear in mind that adding is a lot faster than multiplying for most computers).

You could write a program or design a spreadsheet to do these calculations for you --- that's what graphics programmers have to do.

### 12.3.5. Circles

As well as straight lines, another common shape that computers often need to draw are circles. An algorithm similar to Bresenham's line drawing algorithm, called the Midpoint Circle Algorithm, has been developed for drawing a circle efficiently.

A circle is defined by a centre point, and a radius. Points on a circle are all the radius distance from the centre of the circle.



Try to draw a circle by hand by filling in pixels (without using a ruler or compass). Note how difficult it is to make the circle look round.

It is possible to draw the circle using a formula based on Pythagoras' theorem, but it requires calculating a square root for each pixel, which is very slow. The following algorithm is much faster, and only involves simple arithmetic so it runs quickly on a computer.

### 12.3.6. Bresenham's Midpoint Circle Algorithm

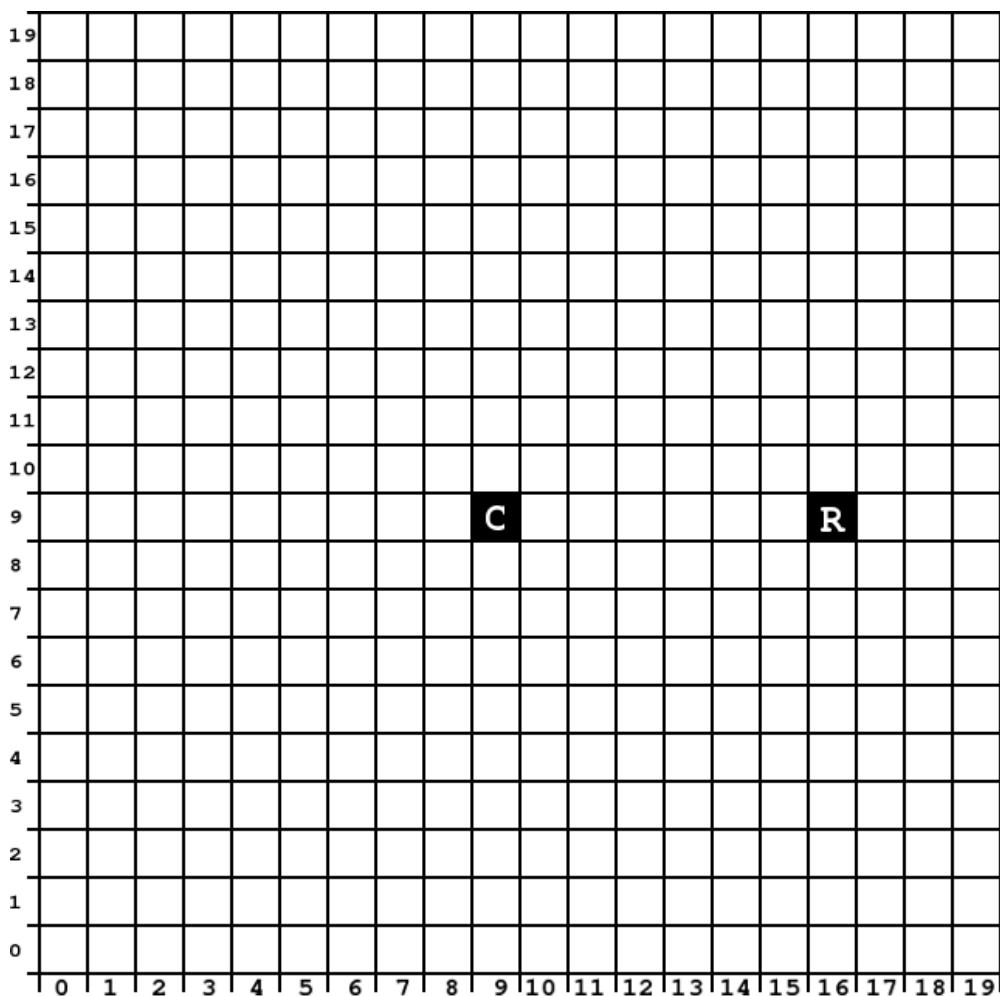
Here are the rules for the Midpoint Circle Algorithm for a circle around  $(c_x, c_y)$  with a radius of  $R$ :

$$\begin{aligned} E &= -R \\ X &= R \\ Y &= 0 \end{aligned}$$

Repeat the following rules in order until  $Y$  becomes greater than  $X$ :

- Fill the pixel at coordinate  $(c_x + X, c_y + Y)$
- Increase  $E$  by  $2 \times Y + 1$
- Increase  $Y$  by 1
- If  $E$  is greater than or equal to 0, subtract  $2 \times X - 1$  from  $E$ , and then subtract 1 from  $X$ .

Follow the rules to draw a circle on the grid, using  $(c_x, c_y)$  as the centre of the circle, and  $R$  the radius. Notice that it will only draw the start of the circle and then it stops because  $Y$  is greater than  $X$ !



When  $Y$  becomes greater than  $X$ , one eighth (an octant) of the circle is drawn. The remainder of the circle can be drawn by reflecting the octant that you already have (you can think of this as repeating the pattern of steps you just did in reverse). You should reflect pixels along the X and Y axis, so that the line of reflection crosses the middle of the centre pixel of the circle. Half of the circle is now drawn, the left and the right half. To add

the remainder of the circle, another line of reflection must be used. Can you work out which line of reflection is needed to complete the circle?

#### **Jargon Buster:** Quadrants and octants

A quadrant is a quarter of an area; the four quadrants that cover the whole area are marked off by a vertical and horizontal line that cross. An *octant* is one eighth of an area, and the 8 octants are marked off by 4 lines that intersect at one point (vertical, horizontal, and two diagonal lines).

To complete the circle, you need to reflect along the diagonal. The line of reflection should have a slope of 1 or -1, and should cross through the middle of the centre pixel of the circle.

While using a line of reflection on the octant is easier for a human to understand, a computer can draw all of the reflected points at the same time it draws a point in the first octant because when it is drawing pixel with an offset of  $(x,y)$  from the centre of the circle, it can also draw the pixels with offsets  $(x,-y)$ ,  $(-x,y)$ ,  $(-x,-y)$ ,  $(y,x)$ ,  $(y,-x)$ ,  $(-y,x)$  and  $(-y,-x)$ , which give all eight reflections of the original point!

By the way, this kind of algorithm can be adapted to draw ellipses, but it has to draw a whole quadrant because you don't have octant symmetry in an ellipse.

### 12.3.7. Practical applications

Computers need to draw lines, circles and ellipses for a wide variety of tasks, from game graphics to lines in an architect's drawing, and even a tiny circle for the dot on the top of the letter 'i' in a word processor. By combining line and circle drawing with techniques like 'filling' and 'antialiasing', computers can draw smooth, clear images that are resolution independent. When an image on a computer is described as an outline with fill colours it is called vector graphics --- these can be re-drawn at any resolution. This means that with a vector image, zooming in to the image will not cause the pixelation seen when zooming in to bitmap graphics, which only store the pixels and therefore make the pixels larger when you zoom in. However, with vector graphics the pixels are recalculated every time the image is redrawn, and that's why it's important to use a fast algorithm like the one above to draw the images.

Outline fonts are one of the most common uses for vector graphics as they allow the text size to be increased to very large sizes, with no loss of quality to the letter shapes.

Computer scientists have found fast algorithms for drawing other shapes too, which means that the image appears quickly, and graphics can display quickly on relatively slow hardware - for example, a smartphone needs to do these calculations all the time to display images, and reducing the amount of calculations can extend its battery life, as well as make it appear faster.

As usual, things aren't quite as simple as shown here. For example, consider a horizontal line that goes from (0,0) to (10,0), which has 11 pixels. Now compare it with a 45 degree line that goes from (0,0) to (10,10). It still has 11 pixels, but the line is longer (about 41% longer to be precise). This means that the line would appear thinner or fainter on a screen, and extra work needs to be done (mainly anti-aliasing) to make the line look ok. We've only just begun to explore how techniques in graphics are needed to quickly render high quality images.

### Project: Line and circle drawing

To compare Bresenham's method with using the equation of a line ( $y = mx + b$ ), choose your own start and end point of a line (of course, make sure it's at an interesting angle), and show the calculations that would be made by each method. Count up the number of additions, subtractions, multiplications and divisions that are made in each case to make the comparison. Note that addition and subtraction is usually a lot faster than multiplication and division on a computer.

You can estimate how long each operation takes on your computer by running a program that does thousands of each operation, and timing how long it takes for each. From this you can estimate the total time taken by each of the two methods. A good measurement for these is how many lines (of your chosen length) your computer could calculate per second.

If you're evaluating how fast circle drawing is, you can compare the number of addition and multiplication operations with those required by the Pythagoras formula that is a basis for the simple [equation of a circle](#) (for this calculation, the line from the centre of the circle to a particular pixel on the edge is the hypotenuse of a triangle, and the other two sides are a horizontal line from the centre, and a vertical line up to

the point that we're wanting to locate. You'll need to calculate the  $y$  value for each  $x$  value; the length of the hypotenuse is always equal to the radius).

## 12.4. The whole story!

We've only scraped the surface of the field of computer graphics. Computer scientists have developed algorithms for many areas of graphics, including:

- lighting (so that virtual lights can be added to the scene, which then creates shading and depth)
- texture mapping (to simulate realistic materials like grass, skin, wood, water, and so on),
- anti-aliasing (to avoid jaggie edges and strange artifacts when images are rendered digitally)
- projection (working out how to map the 3D objects in a scene onto a 2D screen that the viewer is using),
- hidden object removal (working out which parts of an image can't be seen by the viewer),
- photo-realistic rendering (making the image as realistic as possible), as well as deliberately un-realistic simulations, such as "painterly rendering" (making an image look like it was done with brush strokes), and
- efficiently simulating real-world phenomena like fire, waves, human movement, and so on.

The matrix multiplication system used in this chapter is a simplified version of the one used by production systems, which are based on [homogeneous coordinates](#). A homogeneous system uses a 4 by 4 matrix (instead of 3 by 3 that we were using for 3D). It has the advantage that all operations can be done by multiplication (in the 3 by 3 system that we used, you had to multiply for scaling and rotation, but add for translation), and it also makes some other graphics operations a lot easier. Graphics processing units (GPUs) in modern desktop computers are particularly good at processing homogeneous systems, which gives very fast graphics.

### **Curiosity:** Moebius strips and GPUs

Homogeneous coordinate systems, which are fundamental to modern computer graphics systems, were first introduced in 1827 by a German mathematician called [August Ferdinand Möbius](#). Möbius is perhaps better known for coming up with the [Möbius strip](#), which is a piece of paper with only one side!

Matrix operations are used for many things other than computer graphics, including computer vision, engineering simulations, and solving complex equations. Although GPUs were developed for computer graphics, they are often used as processors in their own right because they are so fast at such calculations.

The idea of homogeneous coordinates was developed 100 years before the first working computer existed, and it's almost 200 years later that Möbius's work is being used on millions of computers to render fast graphics. An [animation of a Möbius strip](#) therefore uses two of his ideas, bringing things full circle, so to speak.

## 12.5. Further reading

### 12.5.1. Useful Links

- [https://en.wikipedia.org/wiki/Computer\\_graphics](https://en.wikipedia.org/wiki/Computer_graphics)
- [https://en.wikipedia.org/wiki/Transformation\\_matrix](https://en.wikipedia.org/wiki/Transformation_matrix)
- [https://en.wikipedia.org/wiki/Bresenham's\\_line\\_algorithm](https://en.wikipedia.org/wiki/Bresenham's_line_algorithm)
- [https://en.wikipedia.org/wiki/Ray\\_trace](https://en.wikipedia.org/wiki/Ray_trace)
- <http://www.cosc.canterbury.ac.nz/mukundan/cogr/appcogr.html>
- <http://www.cosc.canterbury.ac.nz/mukundan/covn/appcovn.html>
- [http://www.povray.org/resources/links/3D\\_Tutorials/POV-Ray\\_Tutorials/](http://www.povray.org/resources/links/3D_Tutorials/POV-Ray_Tutorials/)

# 13. Computer Vision

Watch the video online at <https://www.youtube.com/embed/bE2u5trQAHM?rel=0>

## 13.1. What's the big picture?

When computers were first developed, the only way they could interact with the outside world was through the input that people wired or typed into them. Digital devices today often have cameras, microphones and other sensors through which programs can perceive the world we live in automatically. Processing images from a camera, and looking for interesting information in them, is what we call *computer vision*.

With increases in computer power, the decrease in the size of computers and progressively more advanced algorithms, computer vision has a growing range of applications. While it is commonly used in fields like healthcare, security and manufacturing, we are finding more and more uses for them in our everyday life, too.

For example, here is a sign written in Chinese:

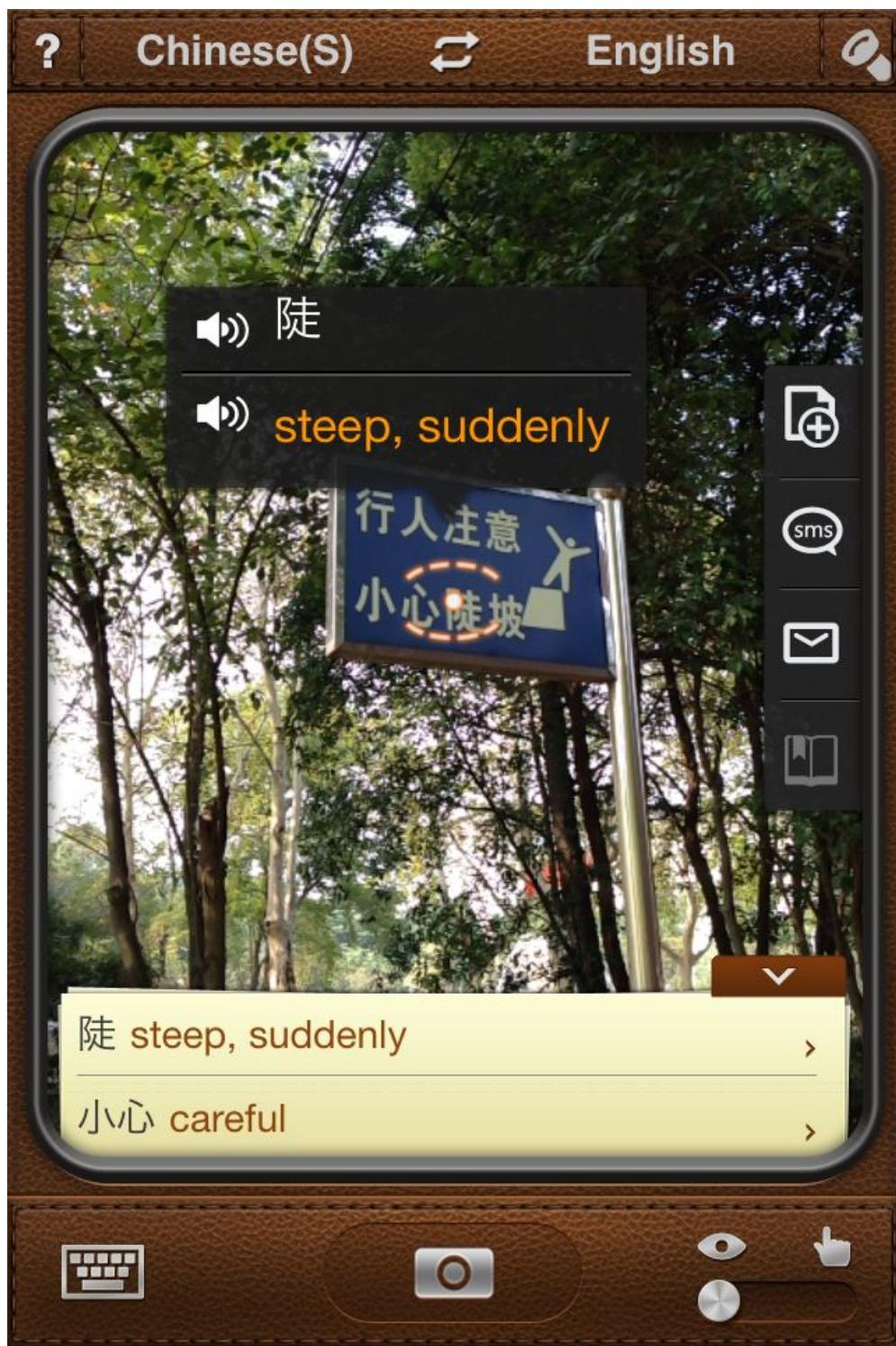


If you can't read the Chinese characters, there are apps available for smartphones that can help:



Having a small portable device that can "see" and translate characters makes a big difference for travellers. Note that the translation given is only for the second part of the phrase (the last two characters). The first part says "please don't", so it could be misleading if you think it's translating the whole phrase!

Recognising of Chinese characters may not work every time perfectly, though. Here is a warning sign:



My phone has been able to translate the “careful” and “steep” characters, but it hasn’t recognised the last character in the line. Why do you think that might be?

Giving users more information through computer vision is only one part of the story. Capturing information from the real world allows computers to assist us in other ways too.

In some places, computer vision is already being used to help car drivers to avoid collisions on the road, warning them when other cars are too close or there are other hazards on the road ahead. Combining computer vision with map software, people have now built cars that can drive to a destination without needing a human driver to steer them. A wheelchair guidance system can take advantage of vision to avoid bumping into doors, making it much easier to operate for someone with limited mobility.

## 13.2. Lights, Camera, Action!

Digital cameras and human eyes fulfill largely the same function: images come in through a lens and are focused onto a light sensitive surface, which converts them into electrical impulses that can be processed by the brain or a computer respectively. There are some differences, however.

**Human eyes** have a very sensitive area in the centre of their field of vision called the fovea. Objects that we are looking at directly are in sharp detail, while our peripheral vision is quite poor. We have separate sets of cone cells in the retina for sensing red, green and blue (RGB) light, but we also have special rod cells that are sensitive to light levels, allowing us to perceive a wide dynamic range of bright and dark colours. The retina has a blind spot (a place where all the nerves bundle together to send signals to the brain through the optic nerve), but most of the time we don't notice it because we have two eyes with overlapping fields of view, and we can move them around very quickly.

**Digital cameras** have uniform sensitivity to light across their whole field of vision. Light intensity and colour are picked up by RGB sensor elements on a silicon chip, but they aren't as good at capturing a wide range of light levels as our eyes are. Typically, a modern digital camera can automatically tune its exposure to either bright or dark scenes, but it might lose some detail (e.g. when it is tuned for dark exposure, any bright objects might just look like white blobs).

It is important to understand that neither a human eye nor a digital camera --- even a very expensive one --- can perfectly capture all of the information in the scene in front of it. Electronic engineers and computer scientists are constantly doing research to improve the quality of the images they capture, and the speed at which they can record and process them.

### Curiosity: Further reading

Further reading can be found at [Cambridge in Colour](#) and [Pixaq](#).

## 13.3. Noise

One challenge when using digital cameras is something called *noise*. That's when individual pixels in the image appear brighter or darker than they should be, due to interference in the electronic circuits inside the camera. It's more of a problem when light levels are dark, and the camera tries to boost the exposure of the image so that you can see more. You can see this if you take a digital photo in low light, and the camera uses a high ASA/ISO setting to capture as much light as possible. Because the sensor has been made very sensitive to light, it is also more sensitive to random interference, and gives photos a "grainy" effect.

Noise mainly appears as random changes to pixels. For example, the following image has "salt and pepper" noise.



Having noise in an image can make it harder to recognise what's in the image, so an important step in computer vision is reducing the effect of noise in an image. There are well-understood techniques for this, but they have to be careful that they don't discard useful information in the process. In each case, the technique has to make an educated guess about the image to predict which of the pixels that it sees are supposed to be there, and which aren't.

Since a camera image captures the levels of red, green and blue light separately for each pixel, a computer vision system can save a lot of processing time in some operations by

combining all three channels into a single “grayscale” image, which just represents light intensities for each pixel.

This helps to reduce the level of noise in the image. Can you tell why, and about how much less noise there might be? (As an experiment, you could take a photo in low light -- can you see small patches on it caused by noise? Now use photo editing software to change it to black and white --- does that reduce the effect of the noise?)

Rather than just considering the red, green and blue values of each pixel individually, most noise-reduction techniques look at other pixels in a region, to predict what the value in the middle of that neighbourhood ought to be.

A *mean filter* assumes that pixels nearby will be similar to each other, and takes the average (i.e. the *mean*) of all pixels within a square around the centre pixel. The wider the square, the more pixels there are to choose from, so a very wide mean filter tends to cause a lot of blurring, especially around areas of fine detail and edges where bright and dark pixels are next to each other.

A *median filter* takes a different approach. It collects all the same values that the mean filter does, but then sorts them and takes the middle (i.e. the *median*) value. This helps with the edges that the mean filter had problems with, as it will choose either a bright or a dark value (whichever is most common), but won't give you a value between the two. In a region where pixels are mostly the same value, a single bright or dark pixel will be ignored. However, numerically sorting all of the neighbouring pixels can be quite time-consuming!

A *Gaussian blur* is another common technique, which assumes that the closest pixels are going to be the most similar, and pixels that are farther away will be less similar. It works a lot like the mean filter above, but is statistically weighted according to a *normal distribution*

### 13.3.1. Activity: noise reduction filters

Open the [noise reduction filtering interactive using this link](#) and experiment with settings as below. You will need a webcam, and the interactive will ask you to allow access to it.

Mathematically, this process is applying a special kind of matrix called a *convolution kernel* to the value of each pixel in the source image, averaging it with the values of other pixels nearby and copying that average to each pixel in the new image. The average is weighted, so that the values of nearby pixels are given more importance than ones that are far away. The stronger the blur, the wider the convolution kernel has to be and the more calculations take place.

For this activity, investigate the different kinds of noise reduction filter and their settings (mask size, number of iterations) and determine:

- how well they cope with different kinds and levels of noise (you can set this in the interactive).
- how much time it takes to do the necessary processing (the interactive shows the number of frames per second that it can process)
- how they affect the quality of the underlying image (a variety of images + camera)

You can take screenshots of the image to show the effects in your writeup. You can discuss the tradeoffs that need to be made to reduce noise.

## 13.4. Face recognition

Recognising faces has become a widely used computer vision application. These days photo album systems like Picasa and Facebook can try to recognise who is in a photo using face recognition --- for example, the following photos were recognised in Picasa as being the same person, so to label the photos with people's names you only need to click one button rather than type each one in.



There are lots of other applications. Security systems such as customs at country borders use face recognition to identify people and match them with their passport. It can also be

useful for privacy --- Google Maps streetview identifies faces and blurs them. Digital cameras can find faces in a scene and use them to adjust the focus and lighting.

There is some information about [How facial recognition works](#) that you can read up as background, and some more information at [i-programmer.info](#).

There are some relevant [articles on the cs4fn website](#) that also provide some general material on computer vision.

### 13.4.1. Project: Recognising faces

First let's manually try some methods for recognising whether two photographs show the same person.

- Get about 3 photos each of 3 people
- Measure features on the faces such as distance between eyes, width of mouth, height of head etc. Calculate the ratios of some of these.
- Do photos of the same person show the same ratios? Do photos of different people show different ratios? Would these features be a reliable way to recognise two images as being the same person?
- Are there other features you could measure that might improve the accuracy?

You can evaluate the effectiveness of facial recognition in free software such as Google's Picasa or the Facebook photo tagging system, but uploading photos of a variety of people and seeing if it recognises photos of the same person. Are there any false negatives or positives? How much can you change your face when the photo is being taken to not have it match your face in the system? Does it recognise a person as being the same in photos taken several years apart? Does a baby photo match of a person get matched with them when they are five years old? When they are an adult? Why or why not does this work?

Try using [face recognition on this website](#) to see how well the Haar face recognition system can track a face in the image. What prevents it from tracking a face? Is it affected if you cover one eye or wear a hat? How much can the image change before it isn't recognised as a face? Is it possible to get it to incorrectly recognise something that isn't a face?

### 13.5. Edge detection

A useful technique in computer vision is *edge detection*, where the boundaries between objects are automatically identified. Having these boundaries makes it easy to *segment* the image (break it up into separate objects or areas), which can then be recognised separately.

For example, here's a photo where you might want to recognise individual objects:



And here's a version that has been processed by an edge detection algorithm:



Notice that the grain on the table above has affected the quality; some pre-processing to filter that would have helped!

You can experiment with edge-detection yourself with the [Canny edge detector on this website](#) (see the information about [Canny edge detection on Wikipedia](#)). This is a widely used algorithm in computer vision, developed in 1986 by John F. Canny.

### 13.5.1. Activity: Edge detection evaluation

With the canny edge detection website above, try putting different images in front of the camera and determine how good the algorithm is at detecting boundaries in the image. Capture images to put in your report as examples to illustrate your experiments with the detector.

- Can the Canny detector find all edges in the image? If there are some missing, why might this be?
- Are there any false edge detections? Why did the system think that they were edges?
- Does the lighting on the scene affect the quality of edge detection?
- Does the system find the boundary between two colours? How similar can the colours be and still have the edge detected?
- How fast can the system process the input? Does the nature of the image affect this?
- How well does the system deal with a page with text on it?

## 13.6. The whole story!

The field of computer vision is changing rapidly at the moment because camera technology has been improving quickly over the last couple of decades. Not only is the resolution of cameras increasing, but they are more sensitive for low light conditions, have less noise, can operate in infra-red (useful for detecting distances), and are getting very cheap so that it's reasonable to use multiple cameras, perhaps to give different angles or to get stereo vision.

Despite these recent changes, many of the fundamental ideas in computer vision have been around for a while; for example, the "k-means" segmentation algorithm was first described in 1967, and the first digital camera wasn't built until 1975 (it was a 100 by 100 pixel Kodak prototype).

(More material will be added to this chapter in the near future)

## 13.7. Further reading

- [https://en.wikipedia.org/wiki/Computer\\_vision](https://en.wikipedia.org/wiki/Computer_vision)
- <https://en.wikipedia.org/wiki/Mri>

- <http://www.cosc.canterbury.ac.nz/mukundan/cogr/applcogr.html>
- <http://www.cosc.canterbury.ac.nz/mukundan/covn/applcovn.html>

# 14. Formal Languages

Watch the video online at <https://www.youtube.com/embed/VnMGOSDkEx4>

## 14.1. What's the big picture?

If you've ever written a text-based program or typed a formula in a spreadsheet, chances are that at some stage the system has told you there's an error and won't even attempt to follow your instructions.

```
>>> x = (a+b)*c+d)
SyntaxError: invalid syntax
```

These "syntax errors" are annoying messages that programmers become excruciatingly familiar with ... it means that they didn't follow the rules somehow, even if it's just a tiny mistake. For example, suppose you intended to write:

```
x = (a+b)*(c+d)
```

but you accidentally left out one of the brackets:

```
x = (a+b)*c+d)
```

When you try to compile or run the program, the computer will tell you that there's an error. If it's really helpful, it might even suggest where the error is, but it won't run the program until you fix it.

This might seem annoying, but in fact by enforcing precision and attention to detail it helps pinpoint mistakes before they become bugs in the program that go undetected until someone using it complains that it's not working correctly.

Whenever you get errors like this, you're dealing with a *formal language*. Formal languages specify strict rules such as "all parentheses must be balanced", "all commands in the

program must be keywords selected from a small set", or "the date must contain three numbers separated by dashes".

Formal languages aren't just used for programming languages --- they're used anywhere the format of some input is tightly specified, such as typing an email address into a web form.

In all these cases, the commands that you have typed (whether in Python, Scratch, Snap!, C, Pascal, Basic, C#, HTML, or XML) are being read by a computer program. (That's right... Python is a program that reads in Python programs.) In fact, the compiler for a programming language is often written in its own language. Most C compilers are written in C --- which begs the question, who wrote the first C compiler (and what if it had bugs)?! Computer Scientists have discovered good ways to write programs that process other programs, and a key ingredient is that you have to specify what is allowed in a program very precisely. That's where formal languages come in.

Many of the concepts we'll look at in this chapter are used in a variety of other situations: checking input to a web page; analysing user interfaces; searching text, particularly with "wild cards" that can match any sequence of characters; creating logic circuits; specifying communication protocols; and designing embedded systems. Some advanced concepts in formal languages even used to explore limits of what can be computed.

Once you're familiar with the idea of formal languages, you'll possess a powerful tool for cutting complex systems down to size using an easily specified format.

`<DIV>Q: HOW DO YOU ANNOY A WEB DEVELOPER?</SPAN>`

[Image source](#)

## 14.2. Getting started

To give you a taste of what can be done, let's try searching for words that fit particular patterns. Suppose you're looking for words that contain the name "tim", type the word "tim" (or a few letters from your name), then press the "Filter words" button to find all words containing "tim".

Use the interactive online at <http://www.csfieldguide.org.nz/en/interactives/regular-expression-filter/index.html>

That's a pretty simple search (though the results may have surprised you!). But now we introduce the *wildcard* code, which in this case is "." --- this is a widely used convention in formal languages. This matches any character at all. So now you can do a search like

```
tim.b
```

and you will get any words that have both "tim" and "b" with a single character --- any character --- in between. Are there any words that match "tim..b"? "tim...b"? You can specify any number of occurrences of a symbol by putting a '\*' after it (again a widely used convention), so:

```
tim.*b
```

will match any words where "tim" is followed by "b", separated by any number of characters --- including none.

Try the following search. What kind of words does it find?

```
x.*y.*z
```

- Can you find words that contain your name, or your initials?
- What about words containing the letters from your name in the correct order?
- Are there any words that contain all the vowels in order (a, e, i, o, u)?

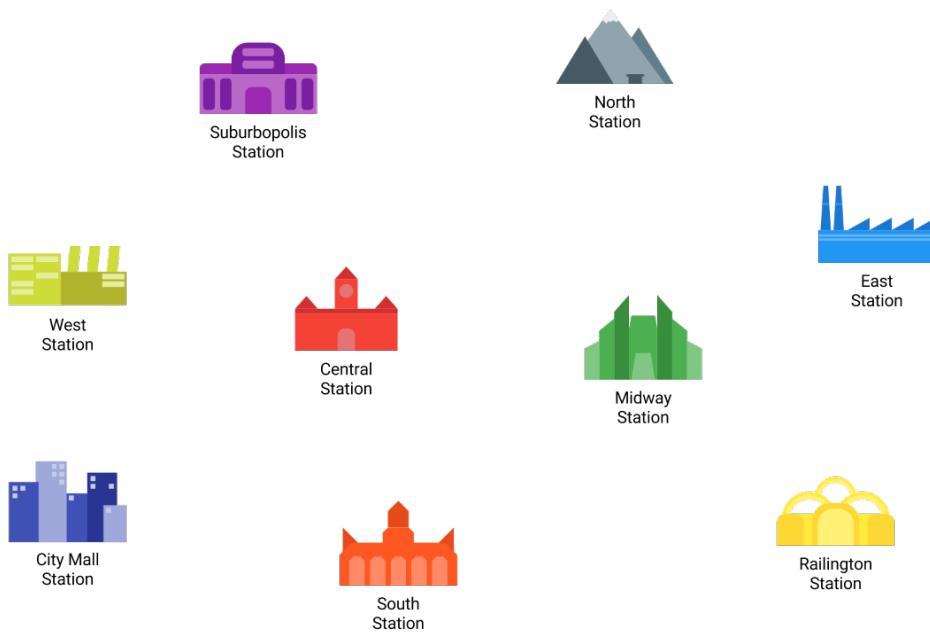
The code you've used above is a part of a formal language called a "regular expression". Computer programs that accept typed input use regular expressions for checking items like dates, credit card numbers and product codes. They're used extensively by programming language compilers and interpreters to make sense of the text that a programmer types in. We'll look at them in more detail in the section on [regular expressions](#).

Next we examine a simple system for reading input called a [finite state automaton](#), which --- as we'll find out later --- is closely related to [regular expressions](#). Later we'll explore the

idea of [grammars](#), another kind of formal language that can deal with more complicated forms of input.

## 14.3. Finite state automata

Here's a map of a commuter train system for the town of Trainsylvania. The trouble is, it doesn't show where the trains go --- all you know is that there are two trains from each station, the A-train and the B-train. The inhabitants of Trainsylvania don't seem to mind this --- it's quite fun choosing trains at each station, and after a while you usually find yourself arriving where you intended.



Click image to enlarge

You can travel around Trainsylvania yourself using the following interactive. You're starting at the City Mall station, and you need to find your way to Suburbopolis. At each station you can choose either the A-train or the B-train --- press the button to find out where it will take you. But, like the residents of Trainsylvania, you'll probably want to start drawing a map of the railway, because later you might be asked to find your way somewhere else. If you want a template to draw on, you can [print one out from here](#).

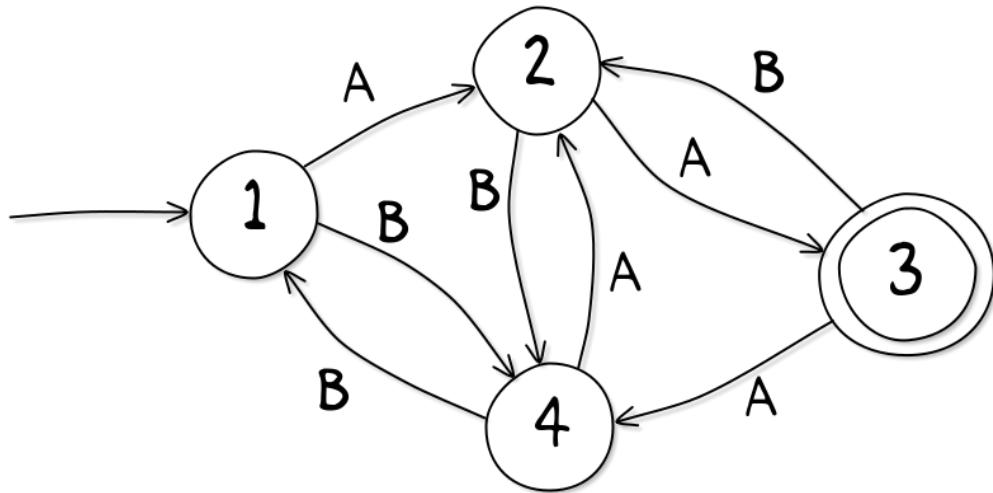
Use the interactive online at <http://www.csfieldguide.org.nz/en/interactives/trainsylvania/index.html>

Did you find a sequence of trains to get from City Mall to Suburbopolis? You can test it by typing the sequence of trains in the following interactive. For example, if you took the A-train, then the B-train, then an A-train, type in ABA.

Use the interactive online at <http://www.csfieldguide.org.nz/en/interactives/trainsylvania-planner/index.html>

Can you find a sequence that takes you from City Mall to Suburbopolis? Can you find another sequence, perhaps a longer one? Suppose you wanted to take a really long route ... can you find a sequence of 12 hops that would get you there? 20 hops?

Here's another map. It's for a different city, and the stations only have numbers, not names (but you can name them if you want).



Suppose you're starting at station 1, and need to get to station 3 (it has a double circle to show that's where you're headed.)

- What's the shortest way to get from station 1 to station 3?
- Where do you end up if you start at station 1 and take the trains ABAA?
- Where do you end up if you start at station 1 and take 20 train hops, always alternating A, B, A, B, A, B?
- Can you give an easy-to-describe sequence of 100 or more hops that will get you to station 3?

The map that we use here, with circles and arrows, is actually a powerful idea from computer science called a Finite State Automaton, or **FSA** for short. Being comfortable with such structures is a useful skill for computer scientists.

**Jargon Buster:** Finite State Automaton

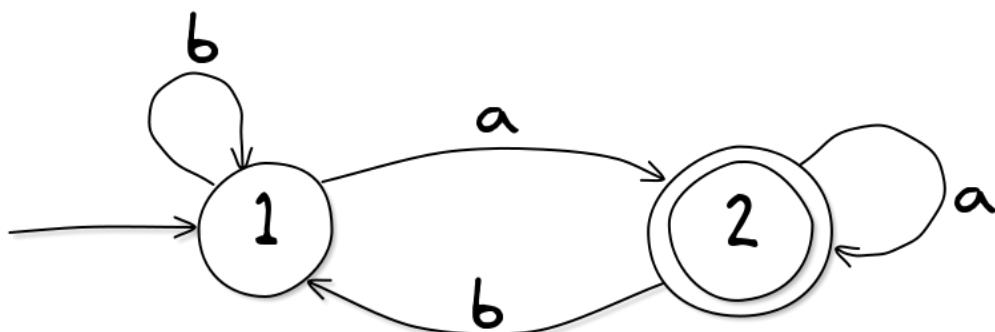
The name **finite state automaton** (FSA) might seem strange, but each word is quite simple. "Finite" just means that there is a limited number of states (such as train stations) in the map. The "state" is just as another name for the train stations we were using. "Automaton" is an old word meaning a machine that acts on its own, following simple rules (such as the cuckoo in a cuckoo clock).

Sometimes an FSA is called a **Finite State Machine** (FSM), or even just a "state machine". By the way, the plural of "Automaton" can be either "Automata" or "Automatons". People working with formal languages usually use Finite State Automata, but "FSAs" for short.

An FSA isn't all that useful for train maps, but the notation is used for many other purposes, from checking input to computer programs to controlling the behaviour of an interface. You may have come across it when you dial a telephone number and get a message saying "Press 1 for this ... Press 2 for that ... Press 3 to talk to a human operator." Your key presses are inputs to a finite state automaton at the other end of the phone line. The dialogue can be quite simple, or very complex. Sometimes you are taken round in circles because there is a peculiar loop in the finite-state automaton. If this occurs, it is an error in the design of the system — and it can be extremely frustrating for the caller!

Another example is the remote control for an air conditioning unit. It might have half a dozen main buttons, and pressing them changes the mode of operation (e.g. heating, cooling, automatic). To get to the mode you want you have to press just the right sequence, and if you press one too many buttons, it's like getting to the train station you wanted but accidentally hopping on one more train. It might be a long journey back, and you may end up exploring all sorts of modes to get there! If there's a manual for the controller, it may well contain a diagram that looks like a Finite State Automaton. If there isn't a manual, you may find yourself wanting to draw a map, just as for the trains above, so that you can understand it better.

The map that we used above uses a standard notation. Here's a smaller one:



Notice that this map has routes that go straight back to where they started! For example, if you start at 1 and take route "b", you immediately end up back at 1. This might seem pointless, but it can be quite useful. Each of the "train stations" is called a state, which is a general term that just represents where you are after some sequence of inputs or decisions. What it actually means depends on what the FSA is being used for. States could represent a mode of operation (like fast, medium, or slow when selecting a washing machine spin cycle), or the state of a lock or alarm (on, off, exit mode), or many other things. We'll see more examples soon.

One of the states has a double circle. By convention, this marks a "final" or "accepting" state, and if we end up there we've achieved some goal. There's also a "start" state --- that's the one with an arrow coming from nowhere. Usually the idea is to find a sequence of inputs that gets you from the start state to a final state. In the example above, the shortest input to get to state 2 is "a", but you can also get there with "aa", or "aba", or "baaaaa". People say that these inputs are "accepted" because they get you from the start state to the final state --- it doesn't have to be the shortest route.

What state would you end up in if the input was the letter "a" repeated 100 times?

Of course, not all inputs get you to state 2. For example, "aab" or even just "b" aren't accepted by this simple system. Can you characterise which inputs are accepted?

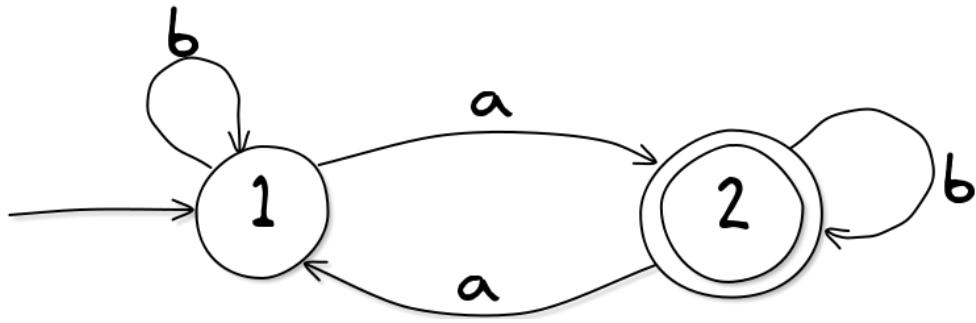
Here's an interactive that follows the rules of the FSA above. You can use it to test different inputs.

[View state interactive](#)

(1)

View the link online at [http://www.csfieldguide.org.nz/releases/1.9.9/\\_static/widgets/fsa-2state-v3.html?map=one](http://www.csfieldguide.org.nz/releases/1.9.9/_static/widgets/fsa-2state-v3.html?map=one)

Here's another FSA, which looks similar to the last one but behaves quite differently. You can test it in the interactive below.



Work out which of the following inputs it accepts. Remember to start in state 1 each time!

- "aaa"
- "abb"
- "aaaa"
- "bababab"
- "babababa"
- the letter "a" repeated 100 times
- the letter "a" repeated 1001 times
- the letter "b" a million times, then an "a", then another million of the letter "b"

Can you state a general rule for the input to be accepted?

[View state interactive](#)

(2)

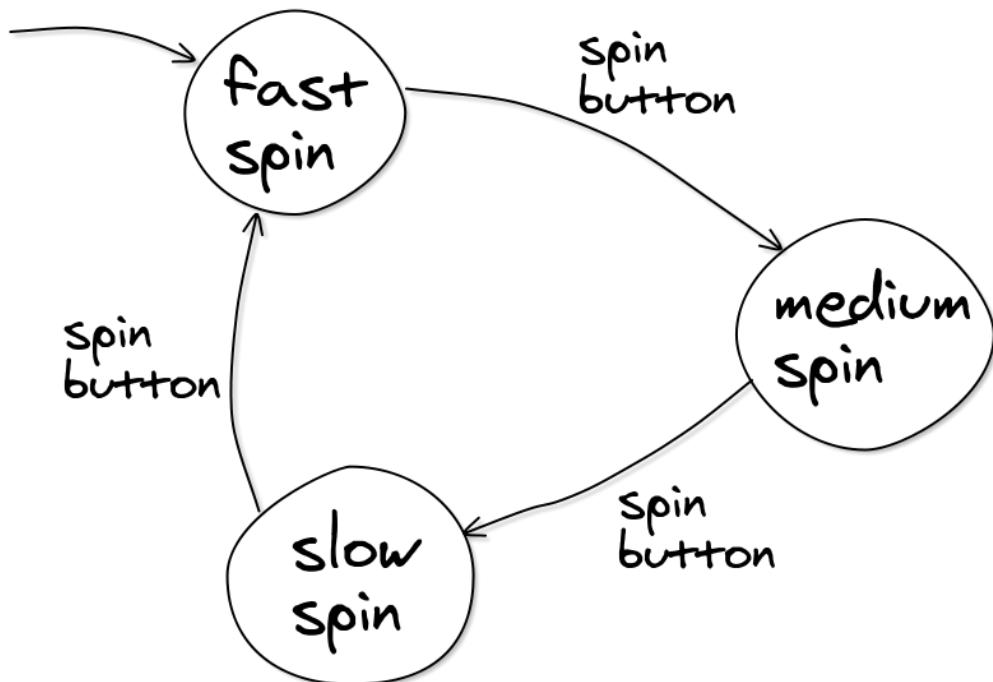
View the link online at [http://www.csfieldguide.org.nz/releases/1.9.9/\\_static/widgets/fsa-2state-v3.html?map=two](http://www.csfieldguide.org.nz/releases/1.9.9/_static/widgets/fsa-2state-v3.html?map=two)

To keep things precise, we'll define four further technical terms. One is the **alphabet**, which is just a list of all possible inputs that might happen. In the last couple of examples the alphabet has consisted of the two letters "a" and "b", but for an FSA that is processing text typed into a computer, the alphabet will have to include every letter on the keyboard.

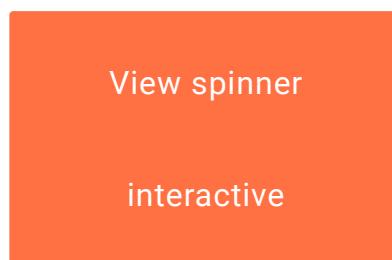
The connections between states are called **transitions**, since they are about changing state. The sequence of characters that we input into the FSA is often called a **string**, (it's just a string of letters), and the set of all strings that can be accepted by a particular FSA is called its **language**. For the FSA in the last example, its *language* includes the *strings* "a", "aaa", "bab", "ababab", and lots more, because these are accepted by it. However, it does not include the strings "bb" or "aa".

The language of many FSAs is big. In fact, the ones we've just looked at are infinite. You could go on all day listing patterns that they accept. There's no limit to the length of the strings they can accept.

That's good, because many real-life FSA's have to deal with "infinite" input. For example, the diagram below shows the FSA for the spin speed on a washing machine, where each press of the spin button changes the setting.

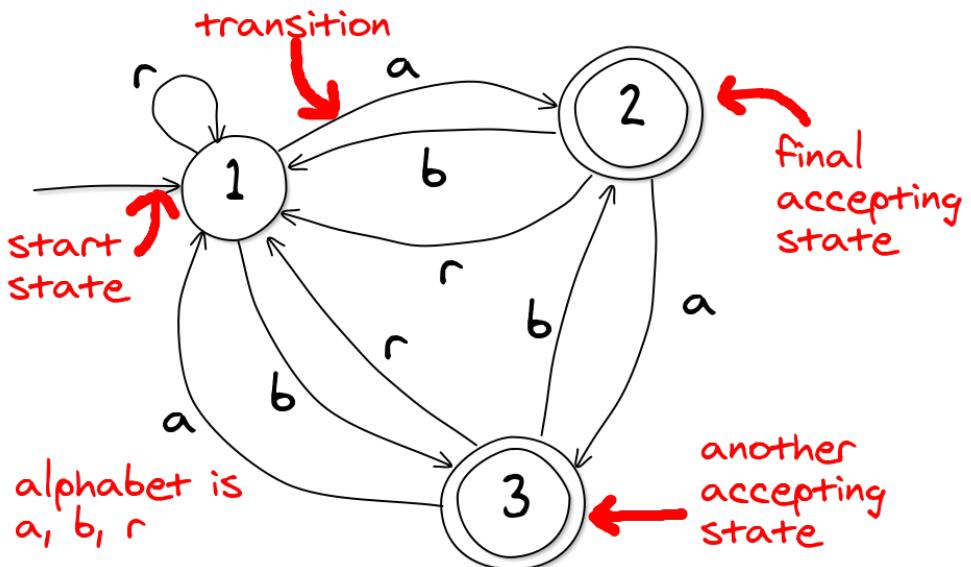


It would be frustrating if you could only change the spin setting 50 times, and then it stopped accepting input ever again. If you want, you could switch from fast to slow spin by pressing the spin button 3002 times. Or 2 times would do. Or 2 million times (try it if you're not convinced).



View the link online at [http://www.csfieldguide.org.nz/releases/1.9.9/\\_static/widgets/fsa-spin-graphic.html](http://www.csfieldguide.org.nz/releases/1.9.9/_static/widgets/fsa-spin-graphic.html)

The following diagram summarizes the terminology we have introduced. Notice that this FSA has two accepting states. You can have as many as you want, but only one start state.



For this FSA, the strings "aa" and "aabba" would be accepted, and "aaa" and "ar" wouldn't. By the way, notice that we often put inverted commas around strings to make it clear where they start and stop. Of course, the inverted commas aren't part of the strings. Notice that "r" always goes back to state 1 -- if it ever occurs in the input then it's like a reset.

Sometimes you'll see an FSA referred to as a Finite State Machine, or FSM, and there are other closely related systems with similar names. We'll mention some later in the chapter.

Now there's something we have to get out of the way before going further. If we're talking about which strings of inputs will get you into a particular state, and the system starts in that state, then the *empty string* --- that is, a string without any letters at all --- is one of the solutions! For example, here's a simple finite state automaton with just one input (button a) that represents a strange kind of light switch. The reset button isn't part of the FSA; it's just a way of letting you return to the starting state. See if you can figure out which patterns of input will turn the light on:

View light

interactive

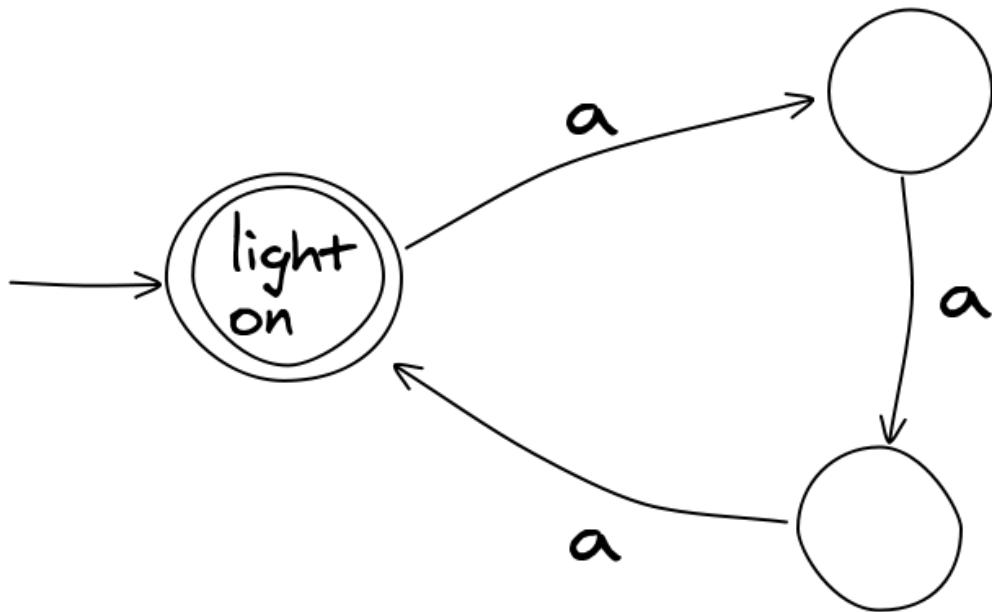
View the link online at [http://www.csfieldguide.org.nz/releases/1.9.9/\\_static/widgets/fsa-strangelight-v3.html](http://www.csfieldguide.org.nz/releases/1.9.9/_static/widgets/fsa-strangelight-v3.html)

Have you worked out which sequences of button presses turn on the light? Now think about the *shortest* sequence from a reset that can turn it on.

Since it's already on when it has been reset, the shortest sequence is zero button presses. It's hard to write that down (although you could use ""), so we have a symbol especially for it, which is the Greek letter epsilon:  $\epsilon$ . You'll come across  $\epsilon$  quite often with formal languages.

It can be a bit confusing. For example, the language (that is, the list of all accepted inputs) of the FSA above includes "aaa", "aaaaaa", and  $\epsilon$ . If you try telling someone that "nothing" will make the light come on that could be confusing --- it might mean that you could never turn the light on --- so it's handy being able to say that the *empty string* (or  $\epsilon$ ) will turn the light on. There are different kinds of "nothing", and we need to be precise about which one we mean!

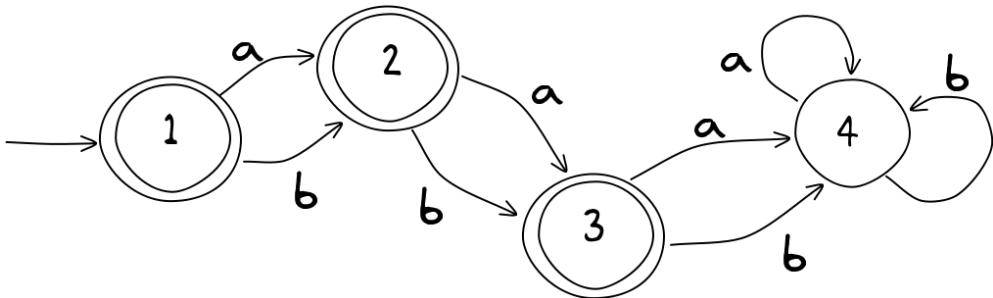
Here's the FSA for the strange light switch. You can tell that  $\epsilon$  is part of the language because the start state is also a final state (in fact, it's the only final state). Actually, the switch isn't all that strange --- data projectors often require two presses of the power button, to avoid accidentally turning them off.



An important part of the culture of computer science is always to consider extreme cases. One kind of extreme case is where there is no input at all: what if a program is given an empty file, or your database has zero entries in it? It's always important to make sure that these situations have been thought through. So it's not surprising that we have a symbol for the empty string. Just for variety, you'll occasionally find some people using the Greek letter lambda ( $\lambda$ ) instead of  $\epsilon$  to represent the empty string.

And by the way, the language of the three-state FSA above is infinitely large because it is the set of all strings that contain the letter "a" in multiples of 3, which is  $\{\epsilon, aaa, aaaaaa, aaaaaaaaa, \dots\}$ . That's pretty impressive for such a small machine.

While we're looking at extremes, here's another FSA to consider. It uses "a" and "b" as its alphabet.

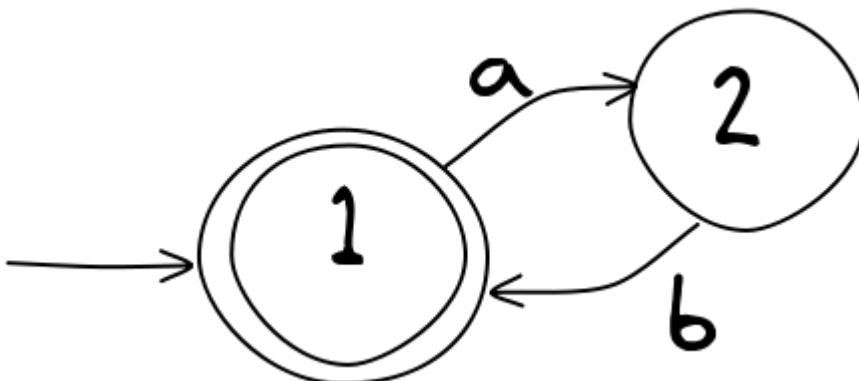


Will it accept the string "aaa"? Or "aba"? Or anything of 3 characters or more?

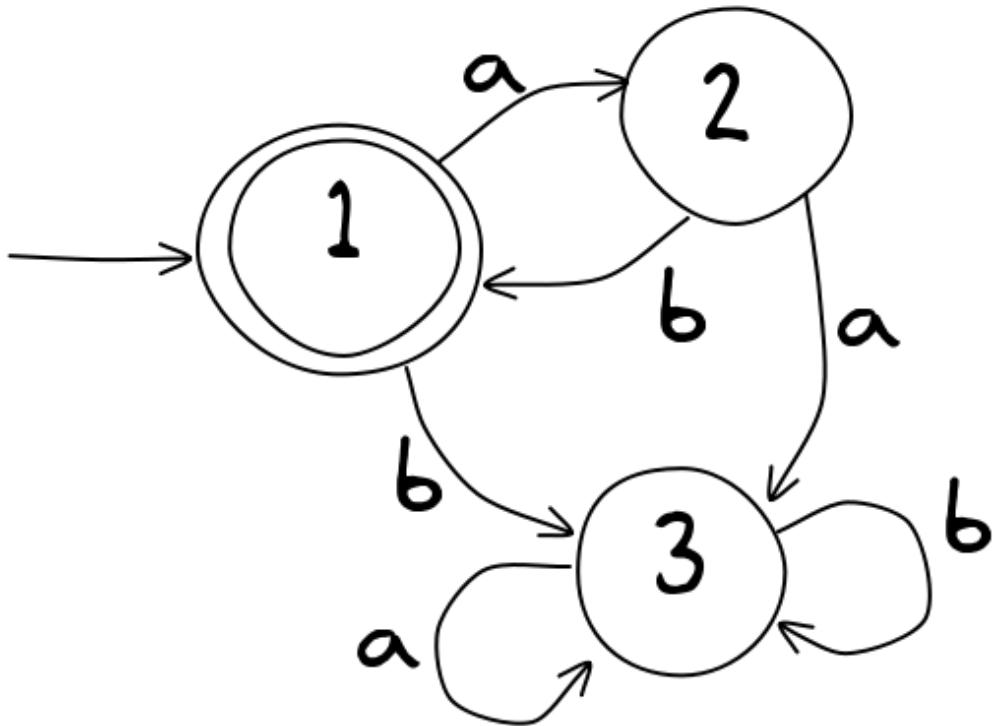
As soon as you get the third character you end up in state 4, which is called a *trap state* because you can't get out. If this was the map for the commuter train system we had at the start of this section it would cause problems, because eventually everyone would end up in the trap state, and you'd have serious overcrowding. But it can be useful in other situations --- especially if there's an error in the input, so no matter what else comes up, you don't want to go ahead.

For the example above, the language of the FSA is any mixture of "a"s and "b"s, but only two characters at most. Don't forget that the empty string is also accepted. It's a very small language; the only strings in it are:  $\{\epsilon, a, b, aa, ab, ba, bb\}$ .

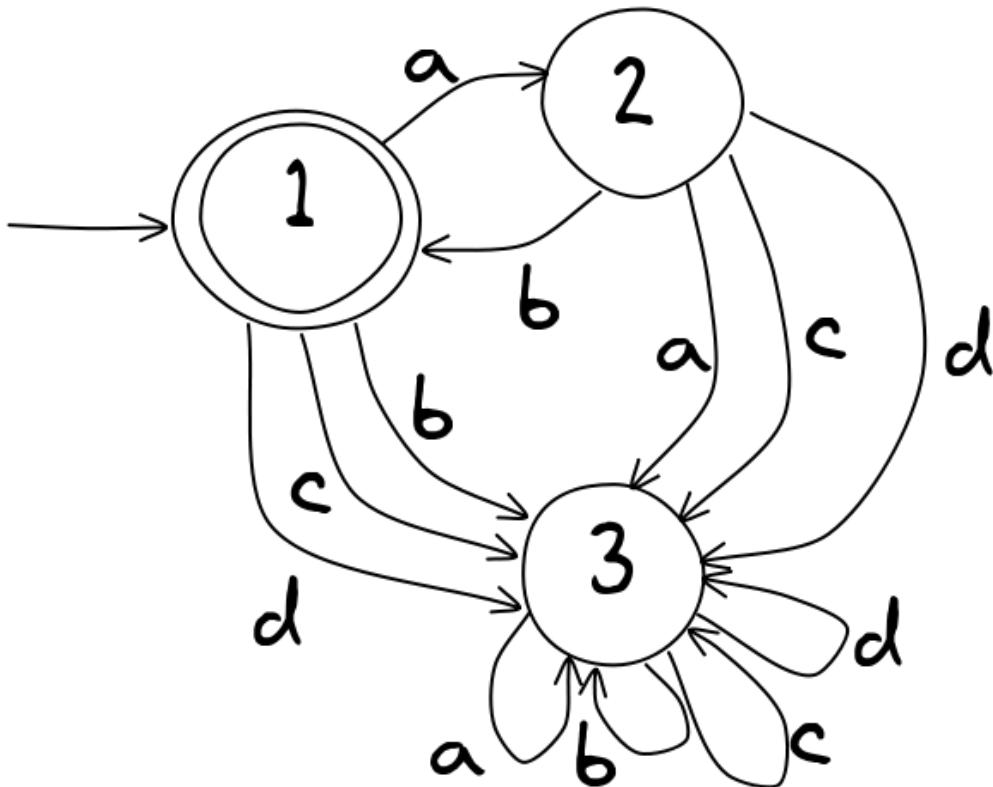
Here's another FSA to consider:



It's fairly clear what it will accept: strings like "ab", "abab", "abababababab", and, of course  $\epsilon$ . But there are some missing transitions: if you are in state 1 and get a "b" there's nowhere to go. If an input cannot be accepted, it will be rejected, as in this case. We could have put in a trap state to make this clear:



But things can get out of hand. What if there are more letters in the alphabet? We'd need something like this:



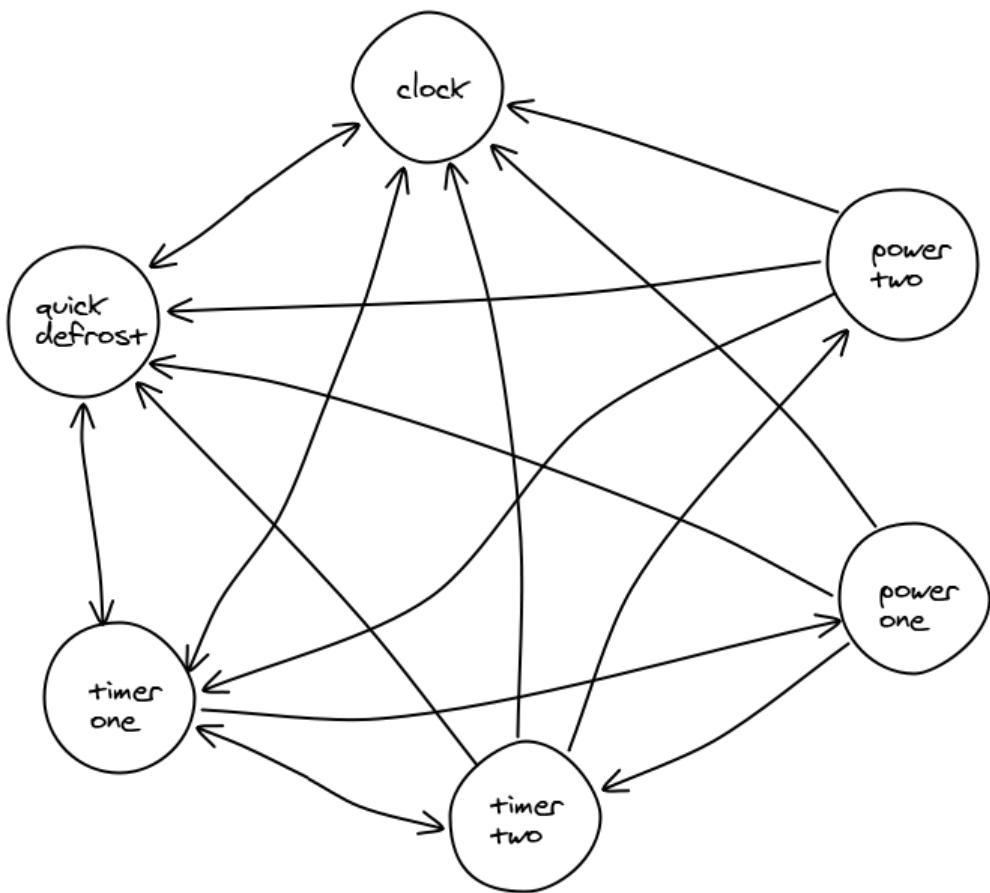
So, instead, we just say that any unspecified transition causes the input to be rejected (that is, it behaves as though it goes into a trap state). In other words, it's fine to use the simple version above, with just two transitions.

Now that we've got the terminology sorted out, let's explore some applications of this simple but powerful "machine" called the Finite State Automaton.

### 14.3.1. Who uses finite state automata?

Finite state automata are used a lot in the design of digital circuits (like the electronics in a hard drive) and embedded systems (such as burglar alarms or microwave ovens). Anything that has a few buttons on it and gets into different states when you press those buttons (such as alarm on/off, high/med/low power) is effectively a kind of FSA.

With such gadgets, FSAs can be used by designers to plan what will happen for every input in every situation, but they can also be used to analyse the interface of a device. If the FSA that describes a device is really complicated, it's a warning that the interface is likely to be hard to understand. For example, here's an FSA for a microwave oven. It reveals that, for example, you can't get from power2 to power1 without going through timer1. Restrictions like this will be very frustrating for a user. For example, if they try to set power1 it won't work until they've set timer1 first. Once you know this sequence it's easy, but the designer should think about whether it's necessary to force the user into that sort of sequence. These sorts of issues become clear when you look at the FSA. But we're straying into the area of Human-Computer Interaction! This isn't surprising because most areas of computer science end up relating to each other --- but this isn't a major application of FSAs, so let's get back to more common uses.



As we shall see in the next section, one of the most valuable uses of the FSA in computer science is for checking input to computers, whether it's a value typed into a dialogue box, a program given to a compiler, or some search text to be found in a large document. There are also data compression methods that use FSAs to capture patterns in the data being compressed, and variants of FSA are used to simulate large computer systems to see how best to configure it before spending money on actually building it.

### CURIOSITY: The largest FSA in the world

What's the biggest FSA in the world, one that lots of people use every day? It's the World-Wide Web. Each web page is like a state, and the links on that page are the transitions between them. Back in the year 2000 the web had a billion pages. In 2008 Google Inc. declared they had found a trillion different web page addresses. That's a lot. A book with a billion pages would be 50 km thick. With a trillion pages, its thickness would exceed the circumference of the earth.

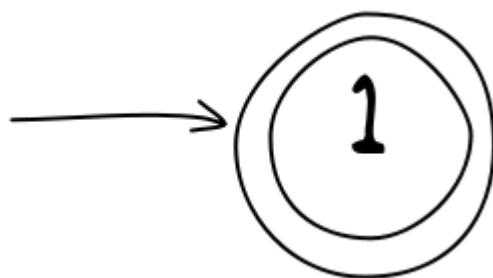
But the web is just a finite-state automaton. And in order to produce an index for you to use, search engine companies like Google have to examine all the pages to see what words they contain. They explore the web by following all the links, just as you

did in the train travelling exercise. Only, because it's called the "web," exploring is called "crawling" — like spiders do.

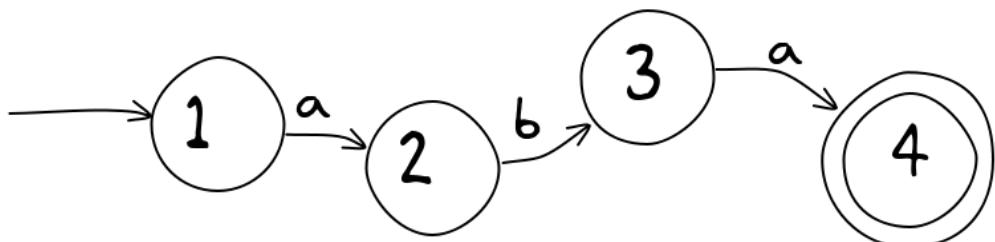
### 14.3.2. Activity: practice creating FSAs

This activity involves constructing and testing your own FSA, using free software that you can download yourself. Before doing that, we'll look at some general ways to create an FSA from a description. If you want to try out the examples here on a live FSA, read the next two sections about using Exorciser and JFLAP respectively, which allow you to enter FSAs and test them.

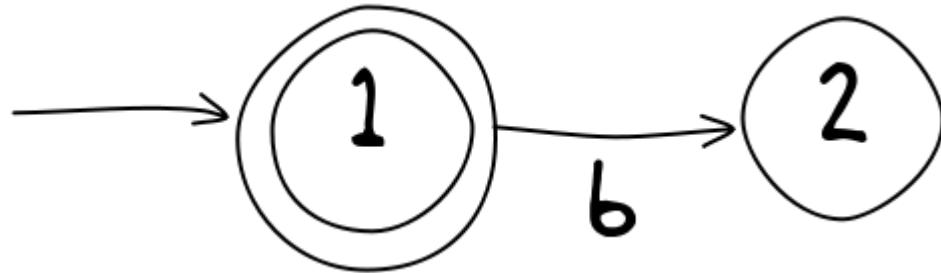
A good starting point is to think of the shortest string that is needed for a particular description. For example, suppose you need an FSA that accepts all strings that contain an even number of the letter "b". The shortest such string is  $\epsilon$ , which means that the starting state must also be a final state, so you can start by drawing this:



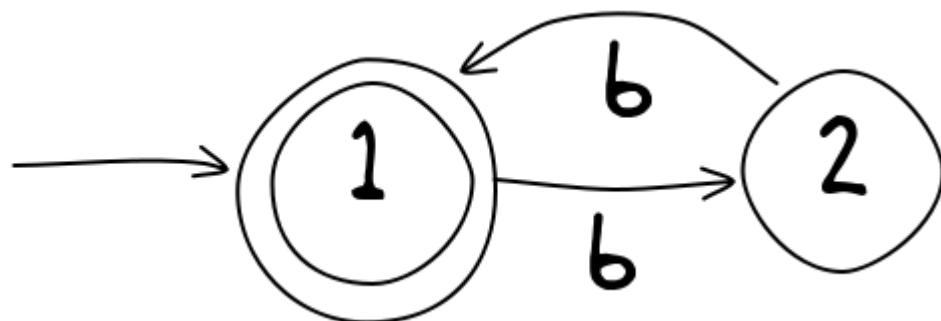
If instead you had to design an FSA where the shortest accepted string is "aba", you would need a sequence of 4 states like this:



Then you need to think what happens next. For example, if we are accepting strings with an even number of "b"s, a single "b" would have to take you from the start state to a non-accepting state:

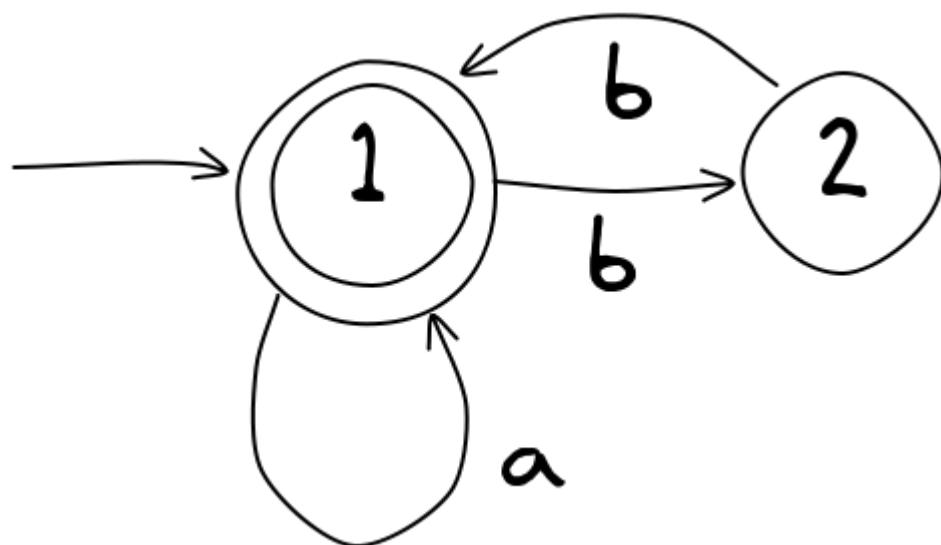


But another "b" would make an even number, so that's acceptable. And for any more input the result would be the same even if all the text to that point hadn't happened, so you can return to the start state:

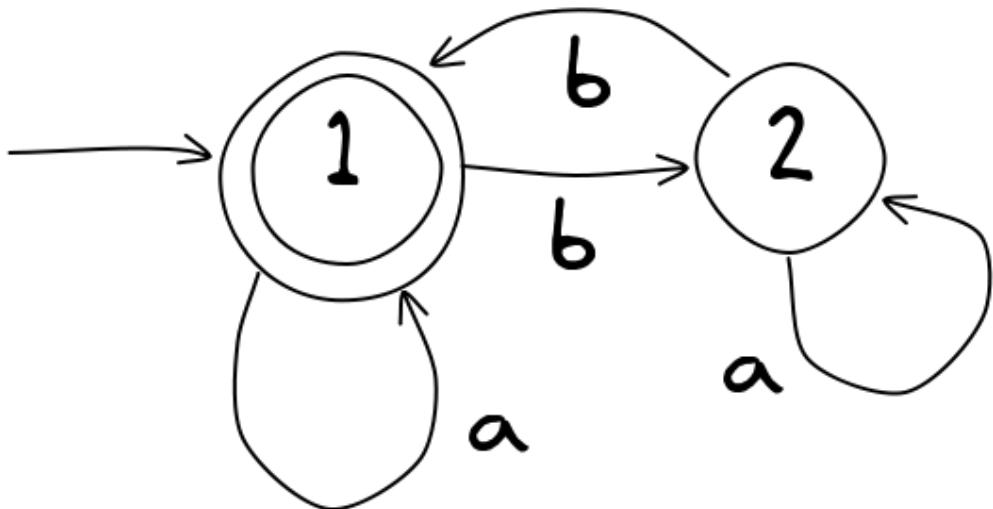


Usually you can find a "meaning" for a state. In this example, being in state 1 means that so far you've seen an even number of "b"s, and state 2 means that the number so far has been odd.

Now we need to think about missing transitions from each state. So far there's nothing for an "a" out of state 1. Thinking about state 1, an "a" doesn't affect the number of "b"s seen, and so we should remain in state 1:



The same applies to state 2:



Now every state has a transition for every input symbol, so the FSA is finished. You should now try some examples to check that an even number of "b"s always brings it to state 1.

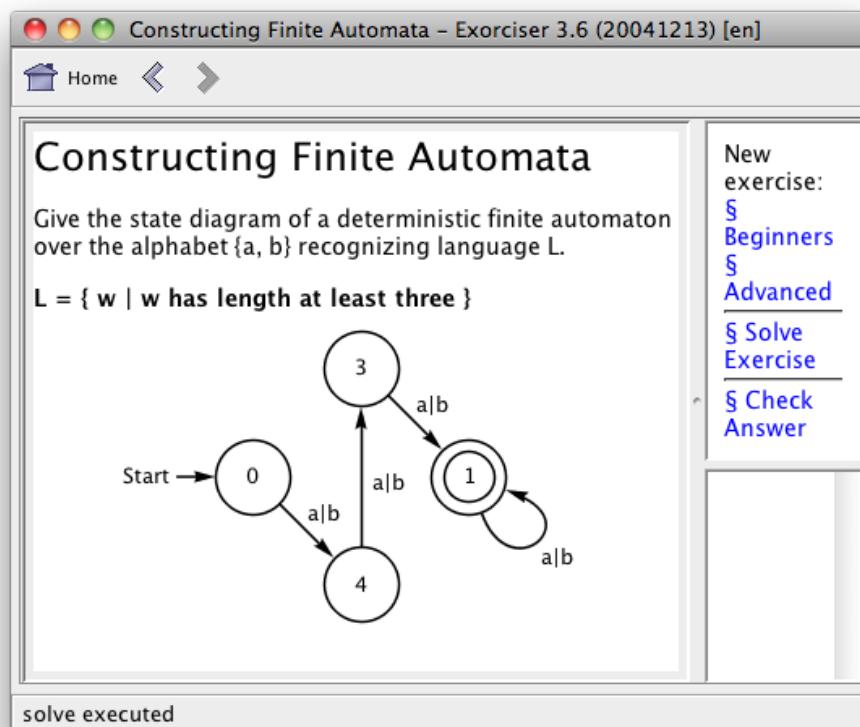
Get some practice doing this yourself! Here are instructions for two different programs that allow you to enter and test FSAs.

#### 14.3.2.1. Exorciser

This section shows how to use some educational software called "Exorciser". (The next section introduces an alternative called JFLAP which is a bit harder to use.) Exorciser has facilities for doing advanced exercises in formal languages; but here we'll use just the simplest ones.

Exorciser can be downloaded [here](#).

When you run it, select "Constructing Finite Automata" (the first menu item); click the "Beginners" link when you want a new exercise. The challenge in each FSA exercise is the part after the | in the braces (i.e., curly brackets). For example, in the diagram below you are being asked to draw an FSA that accepts an input string w if "w has length at least 3". You should draw and test your answer, although initially you may find it helpful to just click on "Solve exercise" to get a solution, and then follow strings around the solution to see how it works. That's what we did to make the diagram below.



To draw an FSA in the Exorciser system, right-click anywhere on the empty space and you'll get a menu of options for adding and deleting states, choosing the alphabet, and so on. To make a transition, drag from the outside circle of one state to another (or out and back to the state for a loop). You can right-click on states and transitions to change them. The notation "a|b" means that a transition will be taken on "a" or "b" (it's equivalent to two parallel transitions).

If your FSA doesn't solve their challenge, you'll get a hint in the form of a string that your FSA deals with incorrectly, so you can gradually fix it until it works. If you're stuck, click "Solve exercise". You can also track input as you type it: right-click to choose that option. See the [SwissEduc website](#) for more instructions.

Constructing Finite Automata – Exorciser 3.6 (20041213) [en]

Home

**Constructing Finite Automata**

Give the state diagram of a deterministic finite automaton over the alphabet {a, b} recognizing language L.

$L = \{ w \mid w \text{ begins with prefix "aaaa"} \}$

Input: aa

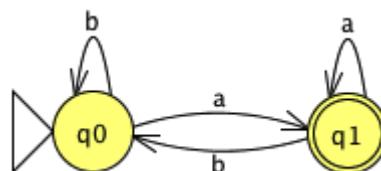
**Wrong answer**  
Your finite automaton fails to accept the language required. Check the word 'aaaa'.

The section after next gives some examples to try. If you're doing this for a report, keep copies of the automata and tests that you do. Right-click on the image for a "Save As" option, or else take screenshots of the images.

### 14.3.2.2. JFLAP

Another widely used system for experimenting with FSAs is a program called JFLAP (download it from <http://jflap.org>). You can use it as an alternative for Exorciser if necessary. You'll need to follow instructions carefully as it has many more features than you'll need, and it can be hard to get back to where you started.

Here's how to build an FSA using JFLAP. As an example, we'll use the following FSA:



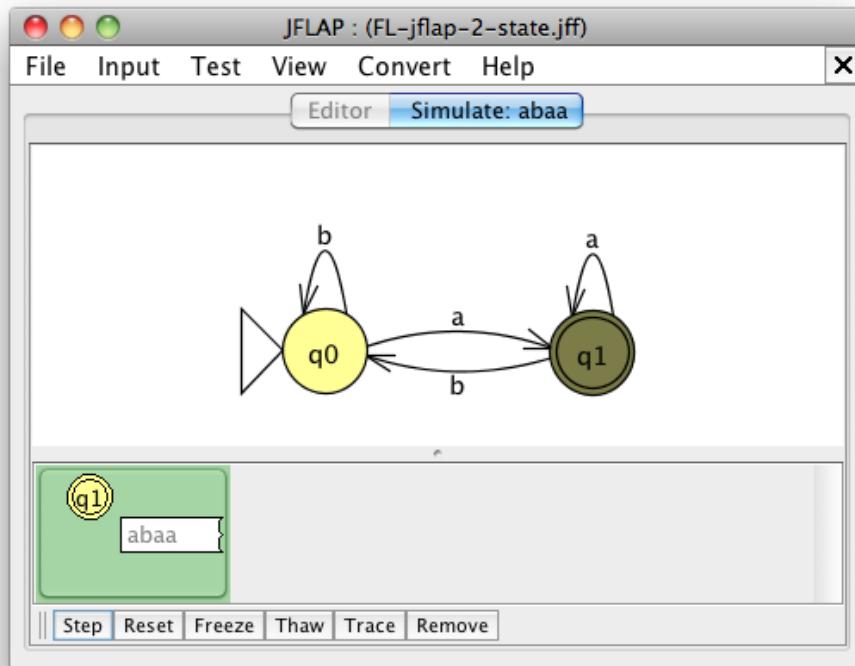
To build this, run JFLAP and:

- click on the "Finite Automaton" button in the control panel.
- In the Editor window, click on the picture of a state (with a little q in it), and then click in the window to create states.
- To move the states around, click on the arrow tool in the toolbar (leftmost icon). It doesn't matter where the states are, but you want them to be easy to view.

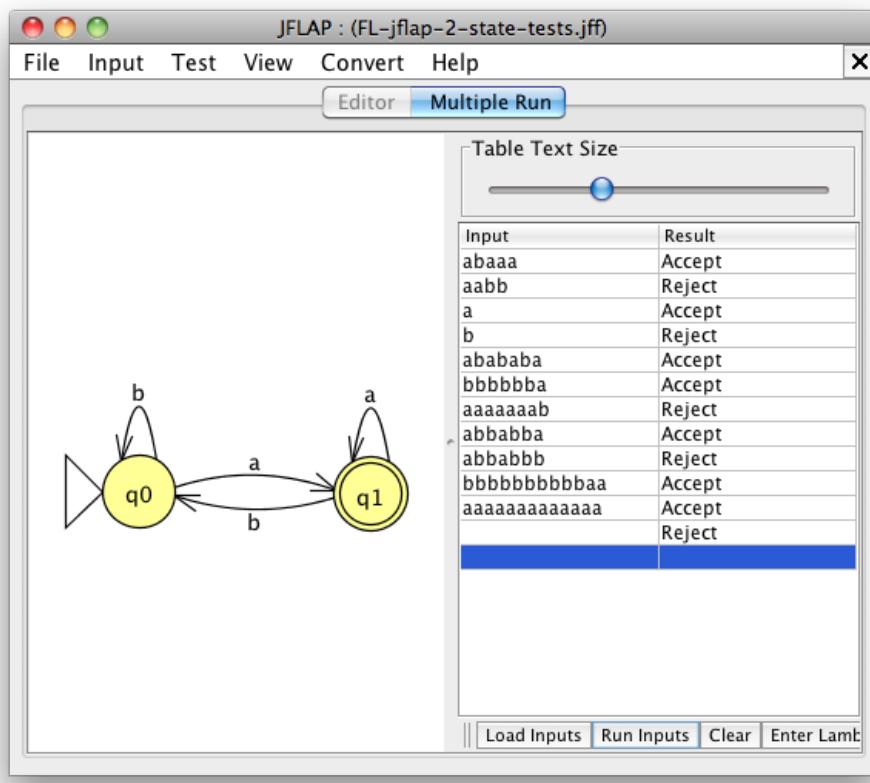
- To put a transition between two states, click on the transition tool (third icon), drag a line between two states, type the label for the transition ("a" or "b" for this exercise), and press return. (The system will offer the empty string ( $\lambda$ ) as a label, but please don't go there!)
- To make a transition loop back to a state, just click on the state with the transition tool.
- You can choose the start state by selecting the arrow tool (leftmost icon), right-clicking on the state, and selecting "Initial". Only one state can be the start state, but you can set more than one "Final" (accepting) state in the same way, by right-clicking on them.

If you need to change something, you can delete things with the delete tool (the skull icon). Alternatively, select the arrow tool and double-click on a transition label to edit it, or right-click on a state. You can drag states around using the arrow tool.

To watch your FSA process some input, use the "Input" menu (at the top), choose "Step with closure", type in a short string such as "abaa", and click "OK". Then at the bottom of the window you can trace the string one character at a time by pressing "Step", which highlights the current state as it steps through the string. If you step right through the string and end up in a final (accepting) state, the panel will come up green. To return to the Editor window, go to the "File" menu and select "Dismiss Tab".



You can run multiple tests in one go. From the "Input" menu choose "Multiple Run", and type your tests into the table, or load them from a text file.



You can even do tests with the empty string by leaving a blank line in the table, which you can do by pressing the "Enter Lambda" button.

There are some FSA examples in the next section. If you're doing this for a report, keep copies of the automata and tests that you do (JFLAP's "File" menu has a "Save Image As..." option for taking snapshots of your work; alternatively you can save an FSA that you've created in a file to open later).

### 14.3.2.3. Examples to try

Using Exorciser or JFLAP, construct an FSA that takes inputs made of the letters "a" and "b", and accepts the input if it meets one of the following requirements. You should build a separate FSA for each of these challenges.

- strings that start with the letter "a" (e.g. "aa", "abaaa", and "abbbb").
- strings that end with the letter "a" (e.g. "aa", "abaaa", and "bbbba").
- strings that have an even number of the letter "a" (e.g. "aa", "abaaa", "bbbb"; and don't forget the empty string  $\epsilon$ ).

- strings that have an odd number of the letter "a" (e.g. "a", "baaa", "bbbab", but not  $\epsilon$ ).
- strings where the number of "a"s in the input is a multiple of three (e.g. "aabaaaaa", "bababab").
- strings where every time an "a" appears in the input, it is followed by a "b" (e.g. "abb", "bbababbbbabab", "bab").
- strings that end with "ab"
- strings that start with "ab" and end with "ba", and only have "b" in the middle (e.g. "abba", "abbbbbba")

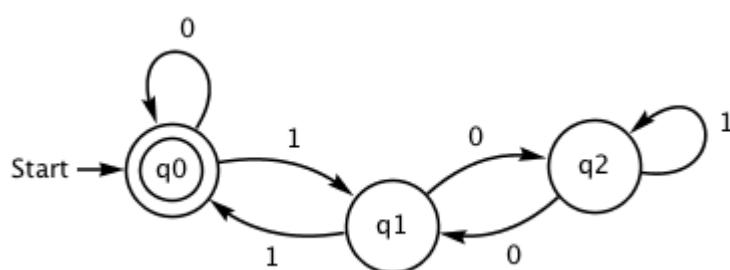
For the FSA(s) that you construct, check that they accept valid input, but also make sure they reject invalid input.

Here are some more sequences of characters that you can construct FSAs to detect. The input alphabet is more than just "a" and "b", but you don't need to put in a transition for every possible character in every state, because an FSA can automatically reject an input if it uses a character that you haven't given a transition for. Try doing two or three of these:

- the names for international standard paper sizes (A1 to A10, B1 to B10, and so on)
- a valid three-letter month name (Jan, Feb, Mar, etc.)
- a valid month number (1, 2, ... 12)
- a valid weekday name (Monday, Tuesday, ...)

A classic example of an FSA is an old-school vending machine that only takes a few kinds of coins. Suppose you have a machine that only takes 5 and 10 cent pieces, and you need to insert 30 cents to get it to work. The alphabet of the machine is the 5 and 10 cent coin, which we call F and T for short. For example, TTT would be putting in 3 ten cent coins, which would be accepted. TFTT would also be accepted, but TFFF wouldn't. Can you design an FSA that accepts the input when 30 cents or more is put into the machine? You can make up your own version for different denominations of coins and required total.

If you've worked with binary numbers, see if you can figure out what this FSA does. Try each binary number as input: 0, 1, 10, 11, 100, 101, 110, etc.



Can you work out what it means if the FSA finishes in state q1? State q2?

**Activity:** Find Finite State Automata in everyday use

There are lots of systems around that use FSAs. You could choose a system, explain how it can be represented with an FSA, and show examples of sequences of input that it deals with. Examples are:

- Board games. Simple board games are often just an FSA, where the next move is determined by some input (e.g. a number given by rolling dice), and the final state means that you have completed the game --- so the first person to the final state wins. Most games are too complex to draw a full FSA for, but a simple game like snakes and ladders could be used as an example. What are some sequences of dice throws that will get you to the end of the game? What are some sequences that don't?!
- Simple devices with a few buttons often have states that you can identify. For example, a remote control for a car alarm might have two buttons, and what happens to the car depends on the order in which you press them and the current state of the car (whether it is alarmed or not). For devices that automatically turn on or off after a period of time, you may have to include an input such as "waited for 30 seconds". Other devices to consider are digital watches (with states like "showing time", "showing date", "showing stopwatch", "stopwatch is running"), the power and eject buttons on a CD player, channel selection on a TV remote (just the numbers), setting a clock, storing presets on a car radio, and burglar alarm control panels.

**Activity:** Kara the ladybug

[SwissEdu](#)c has a programming environment called [Kara](#) (requires Java to be installed), which is a programmable ladybug that (in its simplest version) walks around an imaginary world controlled by actions output by a Finite State Automaton.

The ladybug has (simulated) detectors that sense its immediate surroundings; these serve as input to the FSA.

## 14.4. Regular expressions

We've already had a taste of [regular expressions](#) in the [getting started](#) section. They are just a simple way to search for things in the input, or to specify what kind of input will be accepted as legitimate. For example, many web scripting programs use them to check input for patterns like dates, email addresses and URLs. They've become so popular that they're now built into most programming languages.

You might already have a suspicion that regular expressions are related to [finite state automata](#). And you'd be right, because it turns out that every regular expression has a Finite State Automaton that can check for matches, and every Finite State Automaton can be converted to a regular expression that shows exactly what it does (and doesn't) match. Regular expressions are usually easier for humans to read. For machines, a computer program can convert any regular expression to an FSA, and then the computer can follow very simple rules to check the input.

The simplest kind of matching is just entering some text to match. Open the interactive below and type the text "cat" into the box labeled "Regular expression":

Use the interactive online at <http://www.csfieldguide.org.nz/en/interactives/regular-expression-search/index.html?text=The%20fat%20cat%20sat%20on%20the%20mat.%250AThe%20vindication%20was%20catastrophic.%250AThe%20bilocation%20of%20the%20cataract%20required%20certification.%250AThe%2042%20buffalo%20baffled%20them%20with%20a%20pffffft%20sound.%250APennsylvania%206-5000.%250AAAssorted%20exhalations%3A%20pfft%20pfft%20pft.%250AWas%20that%20a%20match%20or%20was%20it%20not%3F>

If you've only typed the 3 characters "cat", then it should find 6 matches.

Now try typing a dot (full stop or period) as the fourth character: "cat.". In a regular expression, "." can match any single character. Try adding more dots before and after "cat". How about "cat.s" or "cat..n"?

What do you get if you search for "... " (three dots with a space before and after)?

Now try searching for "ic.". The "." matches any letter, so if you really wanted a full stop, you need to write it like this "ic\.". --- use this search to find "ic" at the end of a sentence.

Another special symbol is "\d", which matches any digit. Try matching 2, 3 or 4 digits in a row (for example, two digits in a row is "\d\d").

To choose from a small set of characters, try "[ua]ff". Either of the characters in the square brackets will match. Try writing a regular expression that will match "fat", "sat" and "mat", but not "cat".

A shortcut for "[mnopqrstuvwxyz]" is "[m-s]"; try "[m-s]at" and "[4-6]".

Another useful shortcut is being able to match repeated letters. There are four common rules:

- $a^*$  matches 0 or more repetitions of a
- $a^+$  matches 1 or more repetitions of a
- $a^?$  matches 0 or 1 occurrences of a (that is, a is optional)
- $a^{\{5\}}$  matches "aaaaa" (that is, a repeated 5 times)

Try experimenting with these. Here are some examples to try:

```
f+
pf*t
af*
f*t
f{5}
.{5}n
```

If you want to choose between options, the vertical bar is useful. Try the following, and work out what they match. You can type extra text into the test string area if you want to experiment:

```
was|that|hat
was|t?hat
th(at|e) cat
[Tt]h(at|e) [fc]at
(ff)+
f(ff)+
```

Notice the use of brackets to group parts of the regular expression. It's useful if you want the "+" or "\*" to apply to more than one character.

### Jargon Buster: Regular expression

The term **regular expression** is sometimes abbreviated to "regex", "regexp", or "RE". It's "regular" because it can be used to define sets of strings from a very simple class of

languages called "regular languages", and it's an "expression" because it is a combination of symbols that follow some rules.

Click here for another challenge: you should try to write a short regular expression to match the first two words, but not the last three:

Use the interactive online at <http://www.csfieldguide.org.nz/en/interactives/regular-expression-search/index.html?text=meeeeeeeow%250Ameooooooooooooow%250A%250Awoof%250Amew%250Acluck>

Of course, regular expressions are mainly used for more serious purposes. Click on the following interactive to get some new text to search:

Use the interactive online at <http://www.csfieldguide.org.nz/en/interactives/regular-expression-search/index.html?text=Contact%20me%20at%20spam%40mymail.com%20or%20on%20555-1234%250AFFE962%250ADetails%3A%20fred%40cheapmail.org.nz%20%2803%29%20987-6543%250ALooking%20forward%20to%2021%20Oct%202015%250AGood%20old%205%20Nov%201955%250ABack%20in%202%20Sep%201885%20is%20the%20earliest%20date%250AABC123%250ALet%27s%20buy%20another%202%20Mac%209012%20systems%20%40%20%242000%20each>

The following regular expression will find common New Zealand number plates in the sample text, but can you find a shorter version using the {n} notation?

```
[A-Z] [A-Z] [A-Z]\d\d\d
```

How about an expression to find the dates in the text? Here's one option, but it's not perfect:

```
\d [A-Z] [a-z] [a-z] \d\d\d\d
```

Can you improve on it?

What about phone numbers? You'll need to think about what variations of phone numbers are common! How about finding email addresses?



[Image source](#)

Regular expressions are useful!

The particular form of regular expression that we've been using is for the Ruby programming language (a popular language for web site development), although it's very similar to regular expressions used in other languages including Java, JavaScript, PHP, Python, and Microsoft's .NET Framework. Even some spreadsheets have regular expression matching facilities.

But regular expressions have their limits -- for example, you won't be able to create one that can match palindromes (words and phrases that are the same backwards as forwards, such as "kayak", "rotator" and "hannah"), and you can't use one to detect strings that consist of  $n$  repeats of the letter "a" followed by  $n$  repeats of the letter "b". For those sort of patterns you need a more powerful system called a grammar (see the [section on](#)

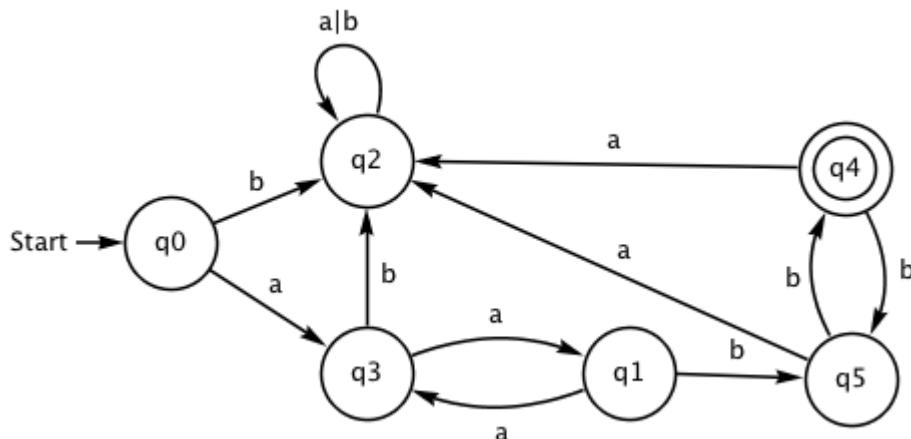
[Grammars](#)). But nevertheless, regular expressions are very useful for a lot of common pattern matching requirements.

### 14.4.1. Regular expressions and FSAs

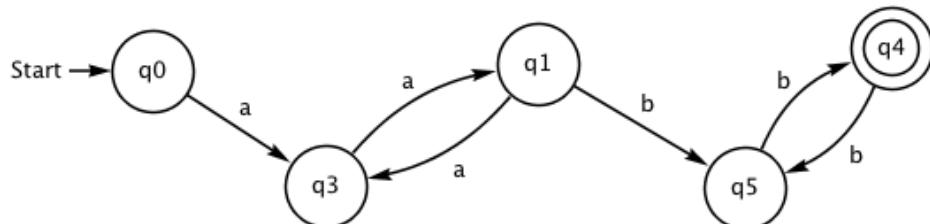
There's a direct relationship between regular expressions and FSAs. For example, consider the following regex, which matches strings that begin with an even number of the letter "a" and end with an even number of the letter "b":

 $(aa)^+(bb)^+$ 

Now look at how the following FSA works on these strings --- you could try "aabb", "aaaabb", "aaaaaabbbb", and also see what happens for strings like "aaabb", "aa", "aabb", and so on.



You may have noticed that q2 is a "trap state". We can achieve the same effect with the following FSA, where all the transitions to the trap state have been removed --- the FSA can reject the input as soon as a non-existent transition is needed.



Like an FSA, each regular expression represents a [language](#), which is just the set of all [strings](#) that match the regular expression. In the example above, the shortest string in the language is "aabb", then there's "aaaabb" and "aabb", and of course an infinite number

more. There's also an infinite number of strings that aren't in this language, like "a", "aaa", "aaaaaa" and so on.

In the above example, the FSA is a really easy way to check for the regular expression -- you can write a very fast and small program to implement it (in fact, it's a good exercise: you typically have an array or list with an entry for each state, and each entry tells you which state to go to next on each character, plus whether or not it's a final state. At each step the program just looks up which state to go to next.)

Fortunately, every regular expression can be converted to an FSA. We won't look at the process here, but both Exorciser and JFLAP can do it for you anyway (see the activities below).

Converting a regex to an FSA is also built into most programming languages.

Programmers usually use regular expressions by calling functions or methods that are passed the regex and the string to be searched. Behind the scenes, the regular expression is converted to a finite state automaton, and then the job of checking your regular expression is very easy.

### Project: Designing regular expressions

Here are some ideas for regular expressions for you to try to create. You can check them using the [Regular Expression Searcher](#) as we did earlier, but you'll need to make up your own text to check your regular expression. When testing your expressions, make sure that they not only accept correct strings, but reject ones that don't match, even if there's just one character missing.

You may find it easier to have one test match string per line in "Your test string". You can force your regular expression to match a whole line by putting "^" (start of line) before the regular expression, and "\$" (end of line) after it. For example, "^a+\$" only matches lines that have nothing but "a"s on them.

Here are some challenges to try to create regular expressions for:

- local forms of non-personalised number plates (e.g. AB1234 or ABC123 in New Zealand)
- any extended form of the word "hello", e.g. "heloooooooooooo"
- variants of "aaaarrrrrgggggghhhh"
- a 24-hour clock time (e.g. 23:00) or a 12-hour time (e.g. 11:55 pm)
- a bank account or credit card number
- a credit card expiry date (must have 4 digits e.g 01/15)
- a password that must contain at least 2 digits

- a date
- a phone number (choose your format e.g. mobile only, national numbers, or international)
- a dollar amount typed into a banking website, which should accept various formats like "\$21.43", "\$21", "21.43", and "\$5,000", but not "21\$", "21.5", "5,0000.00", and "300\$".
- acceptable identifiers in your programming language (usually something like a letter followed by a combination of letters, digits and some punctuation symbols)
- an integer in your programming language (allow for + and - at the front, and some languages allow suffixes like L, or prefixes like 0x)
- an IP address (e.g. 172.16.5.2 or 172.168.10.10:8080)
- a MAC address for a device (e.g. e1:ce:8f:2a:0a:ba)
- postal codes for several countries e.g. NZ: 8041, Canada: T2N 1N4, US: 90210
- a (limited) http URL, such as "<http://abc.xyz>", "<http://abc.xyz#conclusion>", "<http://abc.xyz?search=fgh>".

### Project: Converting Regular Expressions to FSAs

For this project you will make up a regular expression, convert it to an FSA, and demonstrate how some strings are processed.

There's one trick you'll need to know: the software we're using doesn't have all the notations we've been using above, which are common in programming languages, but not used so much in pure formal language theory. In fact, the only ones available are:

- $a^*$  matches 0 or more repetitions of  $a$
- $a \mid b$  matches  $a$  or  $b$
- $(aa \mid bb)^*$  Parentheses group commands together; in this case it gives a mixture of pairs of "a"s and pairs of "b"s.

Having only these three notations isn't too much of a problem, as you can get all the other notations using them. For example, " $a^+$ " is the same as " $aa^*$ ", and " $\d$ " is " $0|1|2|3|4|5|6|7|8|9$ ". It's a bit more tedious, but we'll mainly use exercises that only use a few characters.

#### Converting with Exorciser

Use this section if you're using Exorciser; we recommend Exorciser for this project, but if you're using JFLAP then skip to **Converting with JFLAP** below.

Exorciser is very simple. In fact, unless you change the default settings, it can only convert regular expressions using two characters: "a" and "b". But even that's enough (in fact, in theory any input can be represented with two characters --- that's what binary numbers are about!)

On the plus side, Exorciser has the empty string symbol available --- if you type "e" it will be converted to  $\epsilon$ . So, for example, "(a| $\epsilon$ )" means an optional "a" in the input.

To do this project using Exorciser, go to the start ("home") window, and select the second link, "Regular Expression to Finite Automata Conversion". Now type your regular expression into the text entry box that starts with "R =".

As a warmup, try:

```
aabb
```

then click on "solve exercise" (this is a shortcut --- the software is intended for students to create their own FSA, but that's beyond what we're doing in this chapter).

You should get a very simple FSA!

To test your FSA, right-click on the background and choose "Track input".

Now try some more complex regular expressions, such as the following. For each one, type it in, click on "solve exercise", and then track some sample inputs to see how it accepts and rejects different strings.

```
aa*b  
a(bb)*  
(bba*)*  
(b*a)*a
```

Your project report should show the regular expressions, explain what kind of strings they match, show the corresponding FSAs, show the sequence of states that some sample test strings would go through, and you should explain how the components of the FSA correspond to the parts of the regular expression using examples.

### Converting with JFLAP

If you're using [JFLAP](#) for your project, you can have almost any character as input. The main exceptions are "\*", "+" (confusingly, the "+" is used instead of "|" for

alternatives), and "!" (which is the empty string --- in the preferences you can choose if it is shown as  $\lambda$  or  $\epsilon$ ).

So the main operators available in JFLAP are:

- $a^*$  matches 0 or more repetitions of  $a$
- $a+b$  matches  $a$  or  $b$
- $(aa+bb)^*$  Parentheses group commands together; in this case it gives a mixture of pairs of "a"s and pairs of "b"s.

The JFLAP software can work with all sorts of formal languages, so you'll need to ignore a lot of the options that it offers! This section will guide you through exactly what to do.

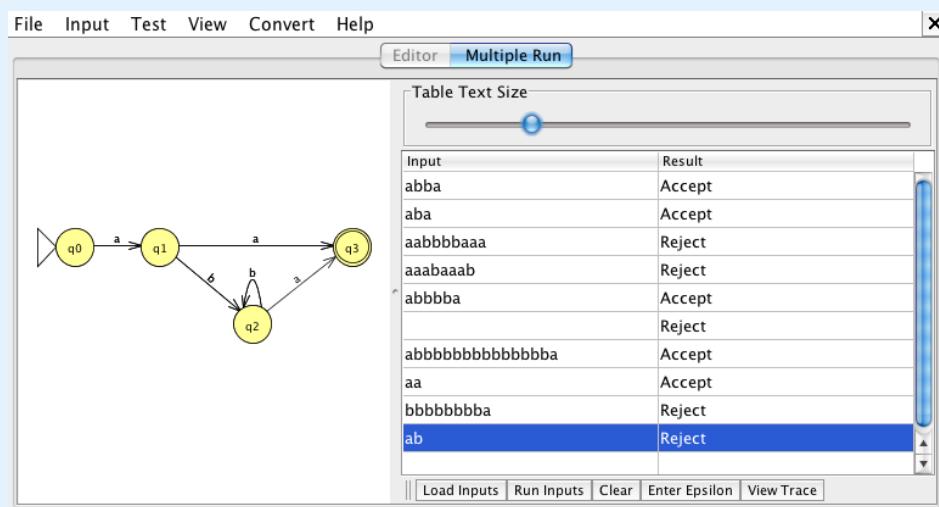
There are some details about the format that JFLAP uses for regular expressions in the following tutorial --- just read the "Definition" and "Creating a regular expression" sections.

<http://www.jflap.org/tutorial/regular/index.html>

As a warmup, we'll convert this regex to an FSA:

ab\*b

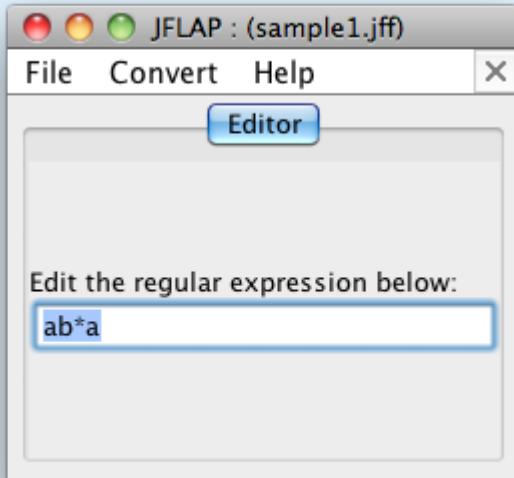
On the main control window of JFLAP click on "Regular Expression", and type your regular expression into JFLAP:



Now try some sample inputs. The starting state is labeled  $q_0$  and will have a large arrow pointing at it. You can get JFLAP to run through some input for you by using the "Input" menu. "Step by state" will follow your input state by state, "Fast run" will show

the sequence of states visited for your input, and "Multiple run" allows you to load a list of strings to test.

Multiple runs are good for showing lots of tests on your regular expression:



For example, "ab" is rejected because it would only get to state 2.

Now you should come up with your own regular expressions that test out interesting patterns, and generate FSA's for them. In JFLAP you can create FSAs for some of regular expressions we used earlier, such as (simple) dates, email addresses or URLs.

Your project report should show the regular expressions, explain what kind of strings they match, show the corresponding FSAs, show the sequence of states that some sample test strings would go through, and you should explain how the components of the FSA correspond to the parts of the regular expression using examples.

### Project: Other ideas for projects and activities

Here are some more ideas that you could use to investigate regular expressions:

- On the [regexdict site](#), read the instructions on the kinds of [pattern matching](#) it can do, and write regular expressions for finding words such as:
  - words that contain "aa"
  - all words with 3 letters
  - all words with 8 letters

- all words with more than 8 letters
- words that include the letters of your name
- words that are made up *only* of the letters in your name
- words that contain all the vowels in reverse order
- words that you can make using only the notes on a piano (i.e the letters A to G and a to g)
- words that are exceptions to the rule "i before e except after c" --- make sure you find words like "forfeit" as well as "science".
  
- Microsoft Word's *Find* command uses regular expressions if you select the "Use wildcards" option. For more details see [Graham Mayor's Finding and Replacing Characters using Wildcards](#).
  
- Explore regular expressions in spreadsheets. The Google docs spreadsheet has a function called RegExMatch, RegExExtract and RegExReplace. In Excel they are available via Visual Basic.
  
- Knitting patterns are a form of regular expression. If you're interested in knitting, you could look into how they are related through the [article about knitting and regular expressions at the CS4FN site](#).
  
- The "grep" command is available in many command line systems, and matches a regular expression in the command with lines in an input file. (the name comes from "Global Regular Expression Parser"). Demonstrate the grep command for various regular expressions.
  
- Functions for matching against regular expressions appear in most programming languages. If your favourite language has this feature, you could demonstrate how it works using sample regular expressions and strings.
  
- Advanced: The free tools *lex* and *flex* are able to take specifications for regular expressions and create programs that parse input according to the rules. They are commonly used as a front end to a compiler, and the input is a program that is being compiled. You could investigate these tools and demonstrate a simple implementation.

## 14.5. Grammars and parsing

With unusual grammar Yoda from Star Wars speaks. Yet still understand him, people can. The flexibility of the rules of English grammar mean that you can usually be understood if

you don't get it quite right, but it also means that the rules get very complicated and difficult to apply.

Grammars in formal languages are much more predictable than grammars in human languages --- that's why they're called *formal* languages! When you're doing English, grammar can be a tricky topic because not only are there so many rules, but there are also so many exceptions --- for example, you need an apostrophe if you write "the computer's USB port", but you have to leave it out if you say "its USB port".

**Grammars** in computer science are mainly used to specify programming languages and file formats, and compilers make a fuss even if you leave out just one bracket or comma! But at least they're predictable.

In this section [when it is finished!] we'll look at the kind of grammars that are widely used in computer science. They are very powerful because they allow a complicated system (like a compiler or a format like HTML) to be specified in a very concise way, and there are programs that will automatically take the grammar and build the system for you. The grammars for conventional programming languages are a bit too unwieldy to use as initial examples (they usually take a few pages to write out), so we're going to work with some small examples here, including parts of the grammars for programming languages.

Note: the remainder of this section is yet to be developed.

### Project: Other ideas for projects and activities

(Note that these will make more sense when the previous introduction to grammars has been completed!)

- Demonstrate how compilers, interpreters, parsers or validators find errors in formal languages e.g. introduce an error to a compiled program, XML document file or web page, and show the effect of the error.
- Find a grammar for a programming language, and show how a sample program would be parsed using the grammar.
- Use examples to show the parse tree (or tree) for a correct and incorrect program fragment, or to show a sequence of grammar productions to construct a correct program fragment
- Explore the grammar for balanced parentheses  $S \rightarrow SS$ ,  $S \rightarrow (S)$ ,  $S \rightarrow ()$

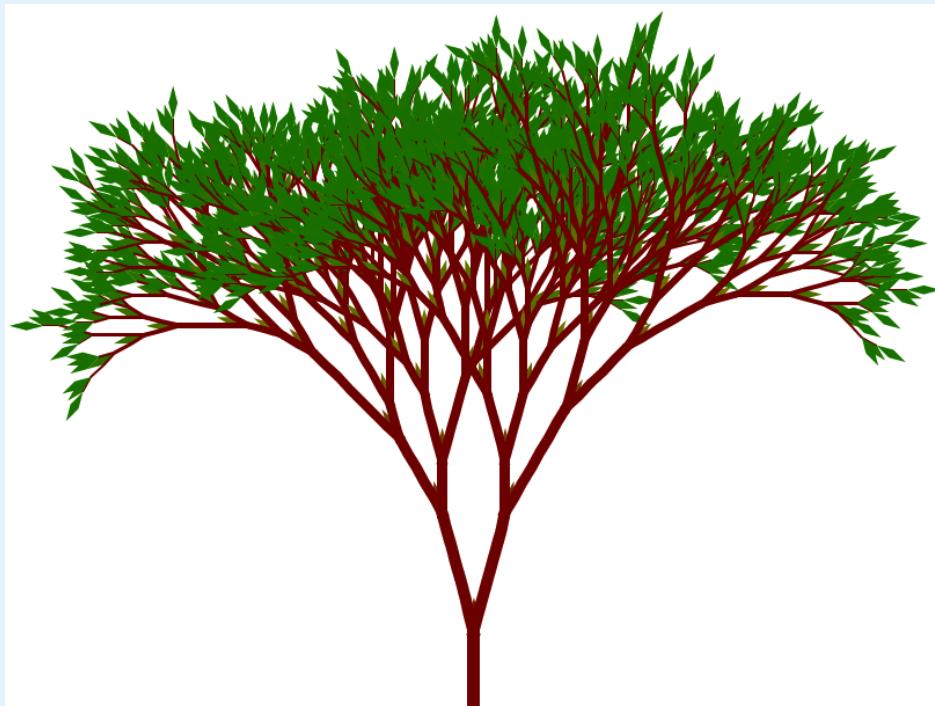
- Find a grammar for a simple arithmetic expression in a programming language, and show the parse tree for sample expressions (such as  $(a+b)^*(c-d)$  ).

**Project:** Grammars in art and music[Image source](#)

The *context free art* program ( <http://www.contextfreeart.org/> ) enables you to specify images using a context-free grammar. For example, the following pictures of trees are defined by just a few rules that are based around a forest being made of trees, a tree being made of branches, and the branches in turn being made of branches themselves! These simple definitions can create images with huge amounts of detail because the drawing process can break down the grammar into as many levels as required. You can define your own grammars to generate images, and even make a movie of them being created, like the one below. Of course, if you do this as a project make sure you understand how the system works and can explain the formal language behind your creation.

Watch the video online at <http://player.vimeo.com/video/52320658>

The JFLAP program also has a feature for rendering "L-systems" (<https://en.wikipedia.org/wiki/L-system>), which are another way to use grammars to create structured images. You'll need to read about how they work in the JFLAP tutorial (<http://www.jflap.org/tutorial/index.html>), and there's a more detailed tutorial at <http://www.cs.duke.edu/csed/pltl/exercises/lessons/20/L-system.zip>. There are some sample files here to get you inspired: (the ones starting "ex10..." <http://www.cs.duke.edu/csed/jflap/jflapbook/files/>) and here's an example of the kind of image that can be produced:



There's also an online system for generating images with L-systems: <http://www.kevs3d.co.uk/dev/lsystems/>

Grammars have been used for music notation:

- The following is the [BNF grammar for the ABC music format](#)
- <http://abc.sourceforge.net/>
- [https://meta.wikimedia.org/wiki/Music\\_markup](https://meta.wikimedia.org/wiki/Music_markup)
- <http://www.emergentmusics.org/theory/15-implementation>

- analyse a simple piece of music in terms of a formal grammar.

## 14.6. The whole story!

If you found the material in this chapter interesting, here are some topics that you might want to look into further, as we've only just scratched the surface of what can be done with formal languages.

Formal languages come up in various areas of computer science, and provide invaluable tools for the computer scientist to reduce incredibly complex systems to a small description, and conversely to create very complex systems from a few simple rules. They are essential for writing compilers, and so are activated every time someone writes a program! They are also associated with automata theory and questions relating to computability, and are used to some extent in natural language processing, where computers try to make sense of human languages.

Technically the kind of finite state automata (FSA) that we used in the [Finite state automata](#) section is a kind known as a *Deterministic Finite Automata* (DFA), because the decision about which transition to take is unambiguous at each step. Sometimes it's referred to as a *Finite State Acceptor* because it accepts and rejects input depending on whether it gets to the final state. There are all sorts of variants that we didn't mention, including the Mealy and Moore machines (which produce an output for each each transition taken or state reached), the nested state machine (where each state can be an FSA itself), the non-deterministic finite automata (which can have the same label on more than one transition out of a state), and the lambda-NFA (which can include transitions on the empty string,  $\lambda$ ). Believe it or not, all these variations are essentially equivalent, and you can convert from one to the other. They are used in a wide range of practical situations to design systems for processing input.

However, there are also more complex models of computation such as the push-down automaton (PDA) which is able to follow the rules of context-free grammars, and the most general model of computation which is called a Turing machine. These models are increasingly complicated and abstract, and structures like the Turing machine aren't used as physical devices (except for fun), but instead as a tool for reasoning about the limits on what can be computed. In fact, in principle every digital computer is a kind of limited Turing machine, so whatever limits we find for a Turing machine gives us limits for everyday computation.

The Turing machine is named after Alan Turing, who worked on these concepts in the early 20th century (that's the same person from whom we got the Turing test in AI, which is something quite different --- Turing's work comes up in many areas of computer science!).

If you want to investigate the idea of a Turing machine and you like chocolate, there's [an activity on the cs4fn site](#) that gives examples of how it works. The Kara programming environment also has a [demonstration of Turing machines](#)

This chapter looked at two main kinds of formal language: the regular expression (RE) and the context-free grammar (CFG). These typify the kinds of languages that are widely used in compilers and file processing systems. Regular expressions are good for finding simple patterns in a file, like identifiers, keywords and numbers in a program, or tags in an HTML file, or dates and URLs in a web form. Context-free grammars are good when you have nested structures, for example, when an expression is made up of other expressions, or when an "if" statement includes a block of statements, which in turn could be "if" statements, ad infinitum. There are more powerful forms of grammars that exist, the most common being context-sensitive grammars and unrestricted grammars, which allow you to have more than one non-terminal on the left hand side of a production; for example, you could have

$xAy \rightarrow aBb,$

which is more flexible but a lot harder to work with. The relationships between the main kinds of grammars was described by the linguist Noam Chomsky, and is often called the [Chomsky Hierarchy](#) after him.

There is a direct correspondence between the "machines" (such as the FSA) and languages (such as the Regular Expression), as each increasingly complex language needs the correspondingly complex machine to process it. For example, an FSA can be used to determine if the input matches a given Regular Expression, but a PDA is needed to match a string to a CFG. The study of formal languages looks at these relationships, and comes up with ways to create the appropriate machines for a given language and vice versa.

There are many tools available that will read in the specification for a language and produce another program to parse the language; some common ones are called "Lex" and "Flex" (both perform lexical analysis of regular expressions), "Yacc" ("yet another compiler compiler") and "Bison" (an improved version of Yacc). These systems make it relatively easy to make up your own programming language and construct a compiler for it, although they do demand quite a range of skills to get the whole thing working!

So we've barely got started on what can be done with formal languages, but the intention of this chapter is to give you a taste of the kind of structures that computer scientists work

with, and the powerful tools that have been created to make it possible to work with infinitely complex systems using small descriptions.

## 14.7. Further reading

Some of the material in this chapter was inspired by <http://www.ccs3.lanl.gov/mega-math/workbk/machine/malearn.html>

There's a good article on finite state machines at <http://www.i-programmer.info/babbages-bag/223-finite-state-machines.html>

### 14.7.1. Books

Textbooks on formal languages will have considerably more advanced material and more mathematical rigour than could be expected at High School level, but for students who really want to read more, a popular book is "Introduction to the Theory of Computation" by Michael Sipser.

Regular expressions and their relationship with FSAs is explained well in the book "Algorithms" by Robert Sedgewick.

### 14.7.2. Useful Links

- [https://en.wikipedia.org/wiki/Formal\\_language](https://en.wikipedia.org/wiki/Formal_language)
- [https://en.wikipedia.org/wiki/Context-free\\_grammar#Examples](https://en.wikipedia.org/wiki/Context-free_grammar#Examples)
- [https://en.wikipedia.org/wiki/Abstract\\_syntax\\_tree](https://en.wikipedia.org/wiki/Abstract_syntax_tree)
- [https://en.wikipedia.org/wiki/Regular\\_expression](https://en.wikipedia.org/wiki/Regular_expression)
- <http://csunplugged.org/finite-state-automata>
- <http://www.i-programmer.info/babbages-bag/223-finite-state-machines.html>
- <http://www.jflap.org/>
- [https://en.wikipedia.org/wiki/Deterministic\\_finite\\_automaton](https://en.wikipedia.org/wiki/Deterministic_finite_automaton)
- [https://en.wikipedia.org/wiki/Finite-state\\_machine](https://en.wikipedia.org/wiki/Finite-state_machine)

# 15. Network Communication Protocols

Watch the video online at <https://www.youtube.com/embed/Hwqmu6pvr6g>

## 15.1. What's the big picture?

Think about the last time someone sent you mail via the post. They probably wrote some content on some paper, put it in an envelope, wrote an address and put it in a postbox. From there, the letter probably went into a sorting center, got sorted, and was put in a bag. The bag then went into a vehicle like a truck, plane or boat. The vehicle either travelled through water, the air, or on the road. The postal system is a complicated one, designed to let individuals communicate easily, yet being efficient enough to group many letters into one postal delivery. The same ideas apply to how messages move around the internet. Whether it be a 'like' on Facebook, a video stream or an email - the internet and its various protocols looks after it for you so it is delivered on time and intact to the other person.

Below we introduce some concepts, algorithms, techniques, applications and problems that relate to network protocols; it isn't a complete list of all the ideas in the area, but should be enough to give you a good idea of what this area of computer science is about.

## 15.2. What is a protocol?

'Protocol' is a fancy word for simply saying "an agreed way to do something". You might have heard it in a cheesy cop show -- "argh Jim, that's against protocol!!!" -- or heard it used in a procedural sense, such as how to file a tax return or sit a driving test. We all use protocols, every day. Think of when you're in class. The *protocol* for asking a question may be as follows: raise your hand, wait for a nod from the teacher then begin asking your question.

Simple tasks require simple protocols like the one above; however more complicated processes may require more complicated protocols. Pilots and aviation crew have their own language (almost) for their tasks. A subset of normal language used to convey information such as altitude, heading, people on board, status and more.

Activities on the internet vary a lot too (email, Skype, video streaming, music, gaming, browsing, chatting), and so do the protocols used to achieve these. These collections of protocols form the topic of Networking Communication Protocols and this chapter will introduce you to some of them, what problems they solve, and what you can do to experience these protocols first hand. Let's start by talking about the one you're using if you're viewing this page on the web.

## 15.3. Application Level Protocols - HTTP, IRC

The URL for the home site of this book is <http://csfieldguide.org>. Ask a few friends what the "http" stands for - they have probably seen it thousands of times...do they know what it is? This section covers high level protocols such as HTTP and IRC, what they can do and how you can use them (hint: you're already using HTTP right now).

### 15.3.1. HyperText Transfer Protocol (HTTP)

The HyperText Transfer Protocol (HTTP) is the most common protocol in use on the internet. The protocol's job is to transfer [HyperText](#) (such as HTML) from a server to your computer. It's doing that right now. You just loaded the Field Guide from the servers where it is hosted. Hit refresh and you'll see it in action.

HTTP functions as a simple conversation between client and server. Think of when you're at a shop:

You: "Can I have a can of soda please?" Shop Keeper: "Sure, here's a can of soda"



HTTP uses a request/response pattern for solving the problem of reliable communication between client and server. The “ask for” is known as a *request* and the reply is known as a *response*. Both requests and responses can also have other data or *resources* sent along with it.

#### Jargon Buster: What is a resource?

A *resource* is any item of data on a server. For example, a blog post, a customer, an item of stock or a news story. As a business or a website, you would create, read, update and delete these as part of your daily business. HTTP is well suited to that. For example, if you’re a news site, every day your authors would add stories, you could update them, delete them if they’re old or become out of date, all sorts. These sorts of methods are required to manage content on a server, and HTTP is the way to do this.

This is happening all the time when you’re browsing the web; every web page you look at is delivered using the HyperText Transfer Protocol. Going back to the shop analogy, consider the same example, this time with more resources shown in asterisk (\*) characters.

You: “Can I have a can of soda please?” \*You hand the shop keeper \$2\* Shop Keeper: “Sure, here’s a can of soda” \*Also hands you a receipt and your change\*



There are nine types of requests that HTTP supports, and these are outlined below.

A GET request returns some text that describes the thing you're asking for. Like above, you ask for a can of soda, you get a can of soda.

A HEAD request returns what you'd get if you did a GET request. It's like this:

You: "Can I have a can of soda please?" Shop Keeper: "Sure, here's the can of soda you'd get" \*Holds up a can of soda\*

What's neat about HTTP is that it allows you to also modify the contents of the server. Say you're now also a representative for the soda company, and you'd like to re-stock some shelves.

A POST request allows you to send information in the other direction. This request allows you to replace a resource on the server with one you supply. These use what is called a Uniform Resource Identifier or URI. A URI is a unique code or number for a resource. Confused? Let's go back to the shop:

Sales Rep: "I'd like to replace this dented can of soda with barcode number 123-111-221 with this one, that isn't dented" Shop Keeper: "Sure, that has now been replaced"

A PUT request adds a new resource to a server, however, if the resource already exists with that URI, it is modified with the new one.

Sales Rep: "Here, have 10 more cans of lemonade for this shelf" Shop Keeper: "Thanks, I've now put them on the shelf"

A DELETE request does what you'd think, it deletes a resource.

Sales Rep: "We are no longer selling 'Lemonade with Extra Vegetables', no one likes it! Please remove them!" Shop Keeper: "Okay, they are gone".

Some other request types (*HTTP methods*) exist too, but they are less used; these are TRACE, OPTIONS, CONNECT and PATCH. You can [find out more about these](#) on your own if you're interested.

In HTTP, the first line of the response is called the *status line* and has a numeric status code such as **404** and a text-based *reason phrase* such as "Not Found". The most common is 200 and this means successful or "OK". HTTP status codes are primarily divided into five groups for better explanation of requests and responses between client and server are named by purpose and a number: Informational 1XX, Successful 2XX, Redirection 3XX, Client Error 4XX and Server Error 5XX. There are many [status codes](#) for representing different cases for error or success. There's even a nice 418: Teapot error on Google: <https://www.google.com/teapot>

So what's actually happening? Well, let's find out. Open a new tab in your browser and open the homepage of the [CS Field Guide here](#). If you're in a Chrome or Safari browser, press Ctrl + Shift + I in Windows or Command + Option + I on a Mac to bring up the web inspector. Select the Network tab. Refresh the page. What you're seeing now is a list of of HTTP requests your browser is making to the server to load the page you're currently viewing. Near the top you'll see a request to `index.html`. Click that and you'll see details of the Headers, Preview, Response, Cookies and Timing. Ignore those last two for now.

Let's look at the first few lines of the headers:

```
Remote Address: 132.181.2.122:3128
Request URL: http://www.csfieldguide.org.nz/en/index.html
Request Method: GET
Status Code: 200 OK
```

The *Remote Address* is the address of the server of the page is hosted on. The *Request URL* is the original URL that you requested. The request method should be familiar from above. It is a GET type request, saying "can I have the web page please?" and the response is the HTML. Don't believe me? Click the *Response* tab. Finally, the *Status Code* is a code that the page can respond with.

Let's look at the *Request Headers* now, click 'view source' to see the original request.

```
GET /index.html HTTP/1.1
Host: www.csfieldguide.org.nz
Connection: keep-alive
Cache-Control: max-age=0
Accept: text/html,application/xhtml+xml;q=0.9,image/webp,*/*;q=0.8
User-Agent: Chrome/34.0.1847.116
Accept-Encoding: gzip,deflate,sdch
Accept-Language: en-US,en;q=0.8
```

As you can see, a request message consists of the following:

- A request line in the form of *method URI protocol/version*
- Request Headers (Accept, User-Agent, Accept-Language etc)
- An empty line
- An optional message body.

Let's look at the *Response Headers*:

```
HTTP/1.1 200 OK
Date: Sun, 11 May 2014 03:52:56 GMT
Server: Apache/2.2.15 (Red Hat)
Accept-Ranges: bytes
Content-Length: 3947
Connection: Keep-Alive
Content-Type: text/html; charset=UTF-8
Vary: Accept-Encoding, User-Agent
Content-Encoding: gzip
```

As you can see, a response message consists of the following:

- Status Line, 200 OK means everything went well.
- Response Headers (Content-Length, Content-Type etc)
- An empty line
- An optional message body.

Go ahead and try this same process on a few other pages too. For example, try these sites:

- A very busy website in terms of content, such as [facebook.com](http://facebook.com)
- A chapter that [doesn't exist on Google](#)
- Your favourite website

**Curiosity:** Who came up with HTTP?

Tim Berners-Lee was credited for creating HTTP in 1989. You can read more about him [here](#).

## 15.3.2. Internet Relay Chat (IRC)

Internet Relay Chat (IRC) is a system that lets you transfer messages in the form of text. It's essentially a chat protocol. The system uses a client-server model. Clients are chat programs installed on a user's computer that connect to a central server. The clients communicate the message to the central server which in turn relays that to other clients. The protocol was originally designed for group communication in a discussion forum, called *channels*. IRC also supports one-to-one communication via *private messages*. It is also capable of file and data transfer too.

The neat thing about IRC is that users can use commands to interact with the server, client or other users. For example /DIE will tell the server to shutdown (although it will only work if you are the administrator!) /ADMIN will tell you who the administrator is.

Whilst IRC may be new to you, the concept of a group conversation online or a *chat room* may not be. There really isn't any difference. Groups exist in the forms of *channels*. A server hosts many channels, and you can choose which one to join.

Channels usually form around a particular topic, such as Python, Music, TV show fans, Gaming or Hacking. Convention dictates that channel names start with one or two # symbols, such as #python or ##TheBigBangTheory. *Conventions* are different to protocols in that they aren't actually enforced by the protocol, but people choose to use it that way.

To get started with IRC, first you should get a client. A client is a program that lets you connect. Ask your teacher about which one to use. For this chapter, we'll use the [freenode web client](#). Check with your teacher about which channel to join, as they may have set one up for you.

Try a few things while you're in there. Look at this [list of commands](#) and try to use some of them. What response do you get? Does this make sense?

Try a one on one conversation with a friend. If they use commands, do you see them? How about the other way around?

# 15.4. Transport Layer Protocols - TCP and UDP

So far we have talked about HTTP and IRC. These protocols are at a level that make sure you do not need to worry about how your data is being transported. Now we'll cover how your data is transferred reliably and efficiently, regardless of what the data is. Below this level is an unreliable medium for transfer (such as wifi or cable, which are subject to interference errors) which causes a concern for data transportation. These protocols take different approaches to ensure data is delivered in an effective and/or efficient manner.

## 15.4.1. TCP

TCP (The Transmission Control Protocol) is one of the most important protocols on the internet. It breaks large messages up into *packets*. What is a packet? A packet is a segment of data that when combined with other packets, make up a total message (something like a HTTP request, an email, an IRC message or a file like a picture or song being downloaded). For the rest of the section, we'll look at how these are used to load an image from a website.

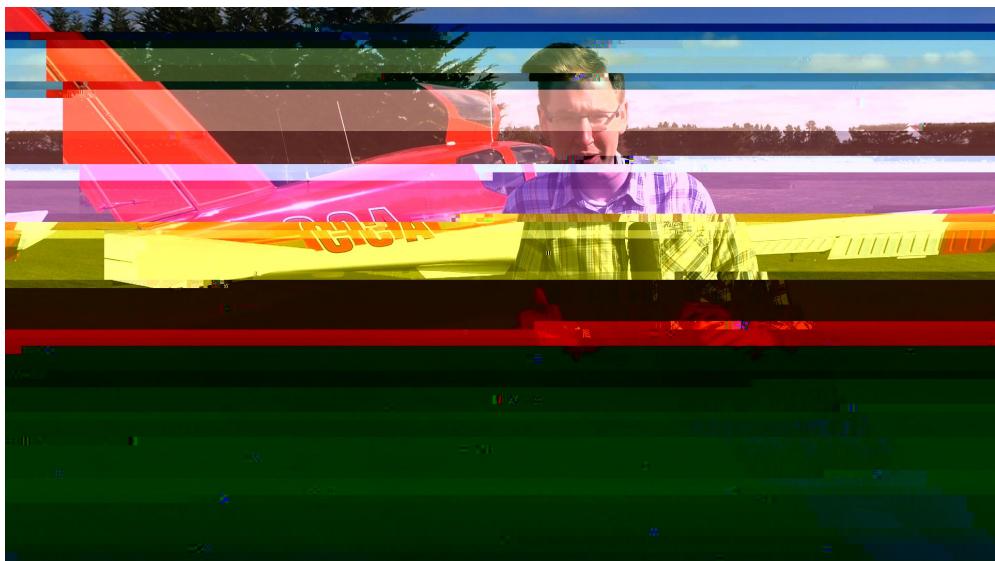
So computer A looks the file and takes it, breaks it into packets. It then sends the packets over the internet and computer B reassembles them and gives them back to you as the image, [which is demonstrated in this video](#).

By now you're probably wondering why we bother splitting up packets... wouldn't it be easier to send the file as a whole? Well, it solves congestion. Imagine you're in a bus, in rush hour and you have to be home by 5. The road is jammed and there's no way you and your friends are getting home on time. So you decide to get out of the bus and go your own separate ways. Web pages are like this too. They are too big to travel together so they are split up and sent in tiny pieces and then reassembled at the other end.

So why don't the packets all just go from computer A to computer B just fine? Ha! That'd be nice. Unfortunately it's not that simple. Through various means, there are some problems that can affect packets. These problems are:

- Packet loss
- Packet delay (packets arrive out of order)
- Packet corruption (the packet gets changed on the way)

So, if we didn't try fix these, the image wouldn't load, bits would be missing, corrupted or computer B might not even recognise what it is!



So, TCP is a protocol that solves these issues. To introduce you to TCP, play the game below, called *Packet attack*. In the game, *you* are the problems (loss, delay, corruption) and as you move through the levels, pay attention to how the computer tries to combat them. Good luck trying to stop the messages getting through correctly!

*Packet Attack* is a direct analogy for TCP and it is intended to be an interactive simulation of it. The Packet Creatures are TCP segments, which move between two computers. The yellow/gray zone is the unreliable channel, susceptible to unreliability. That unreliability is the user playing the game. Remember from the key problems of this topic on the transport level, we have delays, corruption and lost packets, these are the attacks; *delay, corrupt, kill*. Solutions come in the form of TCP mechanisms, they are slowly added level by level. Like in TCP, the game supports packet ordering, checksums (shields), Ack's and Nack's (return creatures) and timeouts .

Use the interactive online at <http://www.csfieldguide.org.nz/en/interactives/packet-attack/index.html>

### Curiosity: Creating your own levels in packet attack

You can also create your own levels in Packet Attack. We've put a level builder in the projects section below so that you can experiment with different reliabilities or combination of defenses.

Adjust the trues, falses and numbers to set different abilities. Raising the numbers will effectively equate to a less reliable communication channel. Adding in more abilities (by setting shields etc to true) will make for a harder to level to beat.

Let's talk about what you saw in that game. What did the levels do to solve the issues of packet loss, delay (reordering) and corruption? TCP has several mechanisms for dealing with packet troubles.

**Curiosity:** What causes delays, losses, and corruption?

Why do packets experience delays, loss and corruption? This is because as packets are sent over a network, they go through various *nodes*. These nodes are effectively different routers or computers. One route might experience more interference than another (causing packet loss), one might be faster or shorter than another (causing order to be lost). Corruption can happen at any time through electronic interference.

Firstly, TCP starts by doing what is known as a handshake. This basically means the two computers say to each other: "Hey, we're going to use TCP for this image. Reconstruct it as you would".

Next is **Ordering**. Since a computer can't look at data and order it like we can (like when we do a jigsaw puzzle or play Scrabble™) they need a way to "stitch" the packets back together. As we saw in *Packet Attack*, if you delayed a message that didn't have ordering, the message may look like "HELOLWOLRD". So, TCP puts a number on each packet (called a sequence number) which signifies its order. With this, it can put them back together again. It's a bit like when you print out a few pages from a printer and you see "Page 2 of 11" on the bottom. Now, if packets do become out of order, TCP will wait for all of the packets to arrive and then put the message together.

Another concept is **checksums**. This concept of storing information about the data may be familiar from the [error control coding chapter](#). Basically, a checksum can detect errors and sometimes with coding schemes, can correct them. In the case of a correctable packet, it is corrected. If not, the packet is useless and needs to be resent. In the game, shields represent checksums. Corrupt a checksum once, and it can recover from the error using error correction. Corrupt it again and it can't.

So how do packets get re-sent? TCP has a concept of *acknowledgement* and *negative acknowledgement* messages (ACK and NACK for short). You would have seen these in the higher levels of the game as the green (ACK) and red (NACK) creatures going back. Acks are sent to let the sender know when a packet arrives and it is usable. Nacks are sent back when a packet arrives and is damaged and needs resending. ACKs and NACKs are useful because they provide a channel *in the opposite direction* for communication. If

computer A receives a NACK, they can resend the message. If it receives an Ack, the computer can stop worrying about a resend.

But does a computer send it again if it doesn't hear back? Yes. It's called a timeout and it's the final line of defense in TCP. If a computer doesn't get an ACK or a NACK back, after a certain time it will just resend the packet. It's a bit like when you're tuning out in class, and the teacher keeps repeating your name until you answer. Maybe that's been you... woops. Sometimes, an ACK might get lost, so the packet is resent after a timeout, but that's OK, as TCP can recognise duplicates and ignore them.

So that's TCP. A protocol that puts accurate data transmission before efficiency and speed in a network. It uses timeouts, checksums, acks and nacks and many packets to deliver a message reliably. However, what if we don't need all the packets? Can we get the overall picture faster? Read on...

## 15.4.2. UDP

UDP (User Datagram Protocol) is a protocol for sending packets that does not guarantee delivery. UDP doesn't guarantee against lost packets, duplicate packets or out of order packets. It just gets the bulk of the data there when it can. Checksums are used for data integrity though, so they have some protection. It's still a protocol because it has a formal packet structure. The packets still include destination and origin as well as the size of the packet.

So do we even use such an unreliable protocol? Yes, but not for anything too important. Files, messages, emails, web pages and other text based items use TCP, but things like streaming music, video, VOIP and so on use UDP. Maybe you've had a call on Skype that has been poor quality. Maybe the video flickers or the sound drops for a split second. That's packets being lost. However, you of course get the overall picture and the conversation you're having is successful.

## 15.5. The Whole Story

Let's say I want to write an online music player. Okay, so I write the code for someone to press play on a website and the song plays. Do I now need to code up the protocol that streams the music? Fine, I write some UDP code. Now, do I need to go install the cables in your house? Sure, I jump in my van and spend a few weeks running cable to your house and make sure the packets can get over too.

No. This sounds absurd. As a web developer, I don't want to worry about anything other than making my music player easy to use and fast. I *don't* want to worry about UDP and I

don't want to worry about ethernet or cables. It's already done, I can assume it's taken care of. And it is.

Internet protocols exist in layers. We have four such layers in the computer science internet model. The top two levels are discussed above in detail, the bottom two we won't focus on. The first layer is the Application Layer, followed by the Transport, Internet and Link layers.

At each layer, data is made up of the previous layers' whole unit of data, and then *headers* are added and passed down. At the bottom layer, the Link layer, a *footer* is added also. Below is an example of what a UDP packet looks like when it's packaged up for transport.

### **Jargon Buster:** Headers and footers

Footers and Headers are basically packet *meta-data*. Information about the information. Like a letterhead or a footnote, they're not part of the content, but they are on the page. Headers and Footers exist on packets to store data. Headers come before the data and footers afterwards.

You can think of these protocols as a game of pass the parcel. When a message is sent in HTTP, it is wrapped in a TCP header, which is then wrapped in an IPv6 header, which is then wrapped in a Ethernet header and footer and sent over ethernet. At the other end, it's unwrapped again from an ethernet *frame*, back to a IP *packet*, a TCP *datagram*, to a HTTP *request*.

### **Curiosity:** What is a Packet?

The name packet is a generic term for a unit of data. In the application layer units of data are called *data* or *requests*, in the transport layer, *datagram* or *segments*, in the Network/IP layer, *packet* and in the physical layer, a *frame*. Each level has its own name for a unit of data (segment, packet, frame, request etc), however the more generic "packet" is often used instead, regardless of layer.

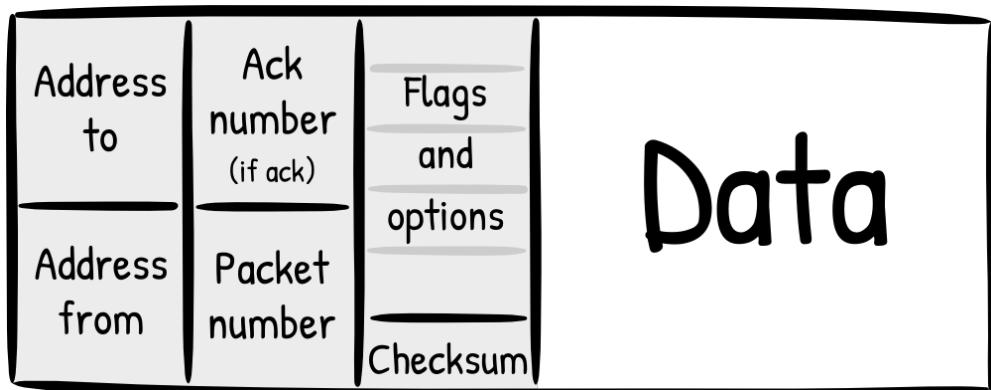
This system is neat because each layer can assume that the layers above and below have guaranteed something about the information, and each layer (and protocol in use at that layer) has a stand-alone role. So if you're making a website you just have to program website code, and not worry about code to make the site work over wifi as well as ethernet. A similar system is in the postal system... You don't put the courier's truck number on the front of the envelope! That's taken care of by the post company, which then uses a system

to sort the mail and assign it to drivers, and then drivers to trucks, and then drivers to routes... none of which you need to worry about when you send or receive a letter or use a courier.

### **Curiosity:** The OSI model vs the TCP/IP model

The OSI internet model is different from the TCP/IP model of the internet that Computer Scientists use to approach protocol design. OSI is considered and probably mentioned in the networking standards but the guide will use the computer science approach because it is simpler, however the main idea of layers of abstraction is more important to get across. You can read more about the differences [here](#).

So what does a TCP segment look like?



As you can see, a packet is divided into four main parts, addresses (source, destination), numbers (sequence number, ACK number if it's an acknowledgement), flags (urgent, checksum) in the header, then the actual data. At each level, a segment becomes the data for the next data unit, and that again gets its own header.

TCP and UDP packets have a number with how big they are. This number means that the packet can actually be as big as you like. Can you think of any advantages of having small packets? How about big ones? Think about the ratio of data to information (such as those in the header and footer).

### **Curiosity:** What does a packet trace look like?

Here's an example of a packet trace on our network... ([using tcpdump on the mac](#))

```
00:55:18.540237 b8:e8:56:02:f8:3e > c4:a8:1d:17:a0:d3, ethertype IPv4
(0x0800), length 100: (tos 0x0, ttl 64, id 41564, offset 0, flags [none],
```

```

proto UDP (17), length 86
    192.168.1.7.51413 > 37.48.71.67.63412: [udp sum ok] UDP, length 58
    0x0000: 4500 0056 a25c 0000 4011 aa18 c0a8 0107
    0x0010: 2530 4743 c8d5 f7b4 0042 1c72 6431 3a61
    0x0020: 6432 3a69 6432 303a b785 2dc9 2e78 e7fb
    0x0030: 68c3 81ab e28b fde3 cfef ae47 6531 3a71
    0x0040: 343a 7069 6e67 313a 7434 3a70 6e00 0031
    0x0050: 3a79 313a 7165

```

## 15.6. Further reading

- The [two generals problem](#) is a famous problem in protocols to talk about what happens when you can't be sure about communication success
- What happens if you were to send packets tied to birds? [IP over Avian Carriers](#)
- Protocols are found in the strangest of places.... [Engine Order Telegraph](#)
- Coursera course on [Internet History, Technology, and Security](#)

### 15.6.1. Videos

There and back again: a packet's tale

Watch the video online at <https://www.youtube.com/embed/WwyJGzZmBe8>

How does the internet work?

Watch the video online at <https://www.youtube.com/embed/i5oe63pOhLI>

How the internet works in 5 minutes

Watch the video online at [https://www.youtube.com/embed/7\\_LPdttKXPc](https://www.youtube.com/embed/7_LPdttKXPc)

### 15.6.2. Extra Activities

- CS Unplugged Routing - Why do packets get delayed? <http://csunplugged.org/routing-and-deadlock>
- Snail Mail - <http://www.cs4fn.org/internet/realsnailmail.php>
- Code.org - The Internet <https://learn.code.org/s/1/level/102>

### 15.6.3. Useful Links

- <http://simple.wikipedia.org/wiki/TCP/IP>

- [https://en.wikipedia.org/wiki/Internet\\_protocol\\_suite](https://en.wikipedia.org/wiki/Internet_protocol_suite)
- [https://en.wikipedia.org/wiki/Hypertext\\_Transfer\\_Protocol](https://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol)
- [https://en.wikipedia.org/wiki/Internet\\_Relay\\_Chat](https://en.wikipedia.org/wiki/Internet_Relay_Chat)
- [https://en.wikipedia.org/wiki/Transmission\\_Control\\_Protocol](https://en.wikipedia.org/wiki/Transmission_Control_Protocol)
- [https://en.wikipedia.org/wiki/User\\_Datagram\\_Protocol](https://en.wikipedia.org/wiki/User_Datagram_Protocol)
- <http://csunplugged.org/routing-and-deadlock>

# 16. Software Engineering

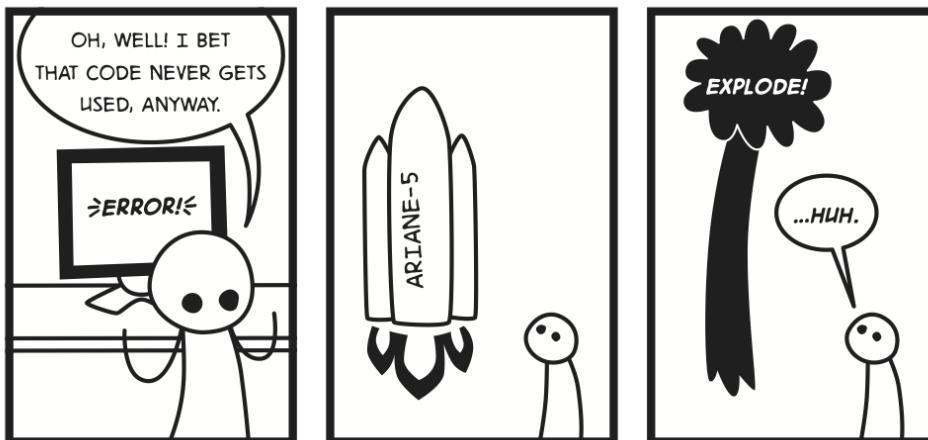
Watch the video online at <https://www.youtube.com/embed/ZNMBEbz2dys?rel=0>

## 16.1. What's the big picture?

Software failures happen all the time. Sometimes it's a little bug that makes a program difficult to use; other times an error might crash your entire computer. Some software failures are more spectacular than others.

In 1996, The ARIANE 5 rocket of the European Space Agency was launched for its first test flight: Countdown, ignition, flame and smoke, soaring rocket... then BANG! Lots of little pieces scattered through the South American rainforest. Investigators had to piece together what happened and finally tracked down this tiny, irrelevant bug. A piece of software on board the rocket which was not even needed had reported an error and started a self-destruct sequence. Thankfully, no one was on board but the failure still caused about US\$370m damage.

Watch the video online at [https://www.youtube.com/embed/gp\\_D8r-2hwk?rel=0](https://www.youtube.com/embed/gp_D8r-2hwk?rel=0)

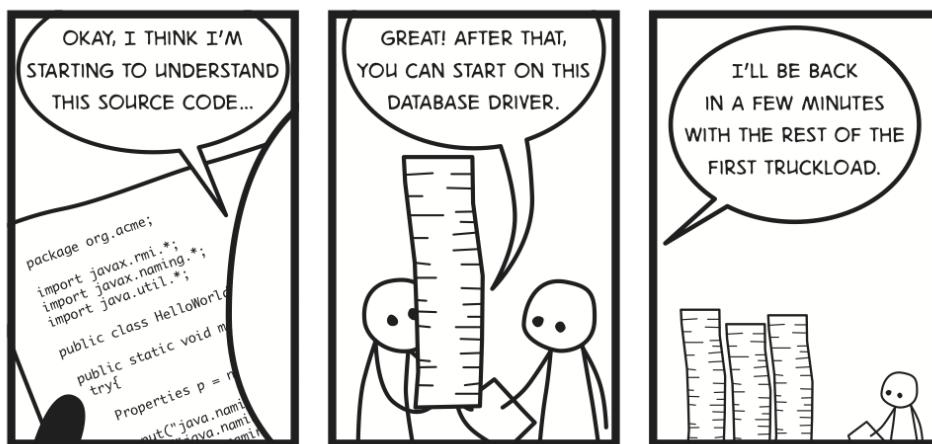


In extreme cases, software bugs can endanger lives. This happened in the 1980s, for example, when a [radiation therapy machine](#) caused the deaths of 3 patients by giving 100 times the intended dose of radiation. And in 1979, a US army computer almost started a nuclear war, when it misinterpreted a simulation of the Soviet Union launching a missile as

the real thing! (If you are interested in other software failures, [CS4FN](#) lists the most spectacular ones!)

Our society today is so reliant on software that we can't even imagine life without it anymore. In many ways, software has made our lives easier: we write emails, chat with friends on Facebook, play computer games and search for information on Google. Heaps of software is hidden behind the scenes too so we don't even know we're using it, for example in cars, traffic lights, TVs, washing machines, Japanese toilets, and hearing aids. We've become so used to having software, we expect it to work at all times!

So why doesn't it? Why do we get bugs in the first place? As it turns out, writing software is incredibly difficult. Software isn't a physical product, so we can't just look at it to see if it's correct. On top of that, most of the software you use every day is huge and extremely complex. Windows Vista is rumoured to have around 50 million lines of code; MacOSX has 86 million. If we printed Vista out on paper, we would get a 88m high stack! That's as high as a 22 storey building or the Statue of Liberty in New York! If you wanted to read through Vista and try to understand how it works, you can expect to get through about 120 lines per hour, so it would take you 417,000 hours or 47 ½ years! (And that's just to read through it, not write it.)



Software engineering is all about how we can create software despite this enormous size and complexity and hopefully get a working product in the end. It was first introduced as a topic of computer science in the 1960s during the so-called “software crisis”, when people realised that the capability of hardware was increasing at incredible speeds while our ability to develop software is staying pretty much the same.

As the name software engineering suggests, we are taking ideas and processes from other engineering disciplines (such as building bridges or computer hardware) and applying them to software. Having a structured process in place for developing software turns out to be hugely important because it allows us to manage the size and complexity of

software. As a result of advances in software engineering, there are many success stories of large and complex software products that work well and contain few bugs. Think, for example, of Google who have huge projects (Google search, Gmail, ...) and thousands of engineers working on them but somehow still manage to create software that does what it should.

Since the 1960s, software engineering has become a very important part of computer science, so much so that today programmers are rarely called programmers, but software engineers. That's because making software is much more than just programming. There are a huge number of jobs for software engineers, and demand for skilled workers continues to grow. The great thing about being a software engineer is that you get to work in large teams to produce products that will impact the lives of millions of people! Although you might think that software engineers would have to be very smart and a bit geeky, communication and teamwork skills are actually more important; software engineers have to be able to work in teams and communicate with their teammates. The ability to work well with humans is at least as important as the ability to work with computers.

As software becomes larger, the teams working on it have grown, and good communication skills have become even more important than in the past. Furthermore, computer systems are only useful if they make things better for humans, so developers need to be good at understanding the users they are developing software for. In fact, as computers become smaller and cheaper (following Moore's law), we've gone from having shared computers that humans have to queue up to use, to having multiple digital devices for each person, and it's the devices that have to wait until the human is ready. In a digital system, the human is the most important part!

### Curiosity: Moore's Law

In 1965, Gordon Moore noticed that the number of transistors on integrated circuits was doubling about every 2 years. This means that computers' processing power was doubling roughly every 2 years (sometimes this is quoted as 18 months due to the combination of the numbers and speed increasing). Moore said that he expected this trend to continue for at least 10 years.

Believe it or not, Moore's law didn't just last for 10 years but is still true nearly 50 years later (although a slowdown is predicted in the next couple of years). This means that computers today are over 100 million times faster than in 1965! (In 2015 it was 50 years since 1965, which means that Moore's law predicts that processing power has doubled about 25 times;  $2^{25}$  is 16,777,216 so if computers could run one instruction per second in 1965, they can now run 33,554,432.) It also means that if you buy a

computer today, you might regret it in two years time when new computers will be twice as fast. Moore's law also relates to other improvements in digital devices, such as processing power in cellphones and the number of pixels in digital cameras.

The exact numbers above will depend on exactly what you're describing, but the main point is that the processing power is increasing *exponentially* – exponential growth doesn't mean just getting a lot faster, but getting unbelievably faster; nothing in human history has ever grown this quickly! To illustrate this in reverse, the time taken to open an app on a smartphone might be half a second today, but a 1965 smartphone would have taken over a year to open the same app (and the phone would probably have been the size of a football field). It's no wonder that smartphones weren't popular in the 1960s.

Although software engineering has come a long way in the last decades, writing software is still difficult today. As a user, you only see the programs that were completed, not those that failed. In 2009, just under a third of all software projects succeeded, while almost a quarter failed outright or were cancelled before the software could be delivered. The remaining projects were either delivered late, were over budget or lacked functionality. A famous recent project failure was the software for the baggage handling system at the new airport in Denver. The system turned out to be more complex than engineers had expected; in the end, the entire airport was ready but had to wait for 16 months before it could be opened because the software for the baggage system was not working. Apparently, the airport lost \$1 million every day during these 16 months!

In this chapter, we look at some of the basics of software engineering. We'll give you an introduction about *analysing* the problem so you know what kind of software to build in the first place; we'll talk briefly about how to structure and *design* software and tell you a bit about *testing*, one of the most important steps for avoiding software bugs. As you'll see below, analysis, design and testing are all important steps when making software. The actual programming part usually takes up only 20% of time on a project (and in this chapter we barely even mention it)! Finally, we'll look at software processes which organise activities including analysis, design and testing so that we always know what we should be doing next.

### **Curiosity:** More about software failures

While successful projects are desirable, there is a lot that can be learnt from failures! Here are some sites that provide further material on this if you are interested.

- [Back to the drawing board - CS4FN](#)

- [Why software fails - IEEE](#)
- [Learning from software failure - IEEE](#)
- [Engineering Disasters 13: Software Flaws](#) is an excerpt from Engineering Disaster Episode 13 explaining software flaws in Ariane 5 and Patriot Missiles

## 16.2. Analysis: What do we build?

To be able to start making software, we first have to decide what we actually want to make. We call this part of the software project *analysis* because we analyse exactly what our software needs to be able to do. Although this sounds trivial, getting the details right is pretty tricky. If someone asked you to design a physical object like a chair or a toaster, you'd probably have a pretty good idea of what the finished product would be like. No matter how many legs you decide to put on your chair, they will still have to do the job of holding up a person against the force of gravity. When designing software, we often don't have the benefit of creating familiar objects, or even known constraints like the laws of physics. If your software was, say, a program to help authors invent imaginary worlds, where would you start? What could you take for granted?

Analysis is extremely important. Obviously, if we make a mistake at this stage of the project, the software we end up building may not be what was wanted; all the other work to design, build and test the software could be for nothing.

For example, imagine your friend Anna asks you to write a program to help her get to school in the morning. You write a great GPS navigation system and show it to Anna, but it turns out that she takes the bus to school so what she really needed was just software showing the current bus timetable. All your hard work was in vain, because you didn't get the details right in the start!

Sometimes we are making software for ourselves; in that case, we can just decide what the software should do. (But be careful: even if you think you know what you want the software to do when you start developing it, you will probably find that by the end of the project you will have a very different view of what it should do. The problem is that before you have the software, you can't really predict how you will use it when it's finished. For example, the people making smart phones and software for smart phones probably didn't anticipate how many people would want to use their smart phones as torches!)

In many cases, we build software for other people. You might make a website for your aunt's clothing shop or write software to help your friends with their maths homework. A software company might create software for a local council or a GP's practice. Google and Microsoft make software used by millions of people around the world. Either way, whether

you're writing a program for your friends or for millions of people, you first have to find out from your customers what they actually need the software to do.

We call anyone who has an interest in the software a *stakeholder*. These are the people that you need to talk to during the analysis part of your project to find out what they need the software to do.

Imagine that you are making a phone app that allows students to preorder food from the school cafeteria. They can use the app to request the food in the morning and then just go and pick up the food at lunch time. The idea is that this should help streamline the serving of food and reduce queues in the cafeteria. Obvious stakeholders for your project are the students (who will be using the phone app) and cafeteria staff (who will be receiving requests through the app). Less obvious (and indirect) stakeholders include parents ("I have to buy my child a smartphone so they can use this app?"), school admin ("No phones should be used during school time!") and school IT support who will have to deal with all the students who can't figure out how to work the app or connect to the network. Different stakeholders might have very different ideas about what the app should do.

To find out what our stakeholders want the software to do, we usually interview them. We ask them questions to find *functional* and *non-functional* requirements for the software. Functional requirements are things the software needs to do. For example, your phone app needs to allow students to choose the food they want to order. It should then send the order to the cafeteria, along with the student's name so that they can be easily identified when picking up the food.

Non-functional requirements, on the other hand, don't tell us *what* the software needs to do but *how* it needs to do it. How efficient does it need to be? How reliable? What sort of computer (or phone) does it need to run on? How easy to use should it be?

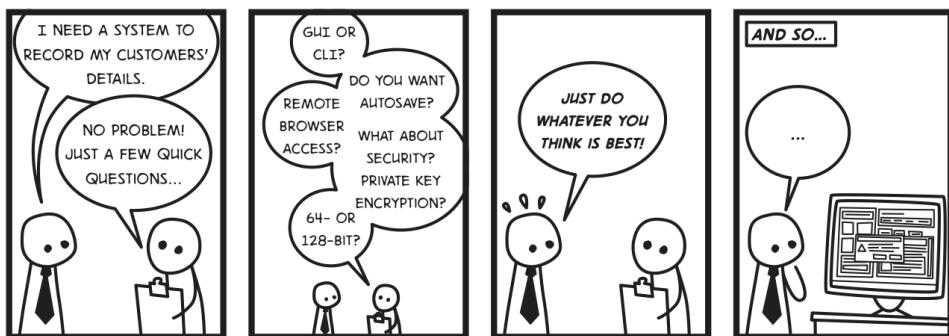
So we first figure out who our stakeholders are and then we go to interview them to find the requirements for the software. That doesn't sound too hard, right? Unfortunately, it's the communication with the customer that often turns out to be most difficult.

The first problem is that customers and software engineers often don't speak the same language. Of course, we don't mean to say that they don't both speak English, but software engineers tend to use technical language, while customers use language specific to their work. For example, doctors might use a lot of scary medical terms that you don't understand.

Imagine that a customer asks you to develop a scoring system for the (fictional) sport of Whacky-Flob. The customer tells you "It's really simple. You just need to record the foo-whacks, but not the bar-whacks, unless the Flob is circulating". After this description,

you're probably pretty confused because you don't know anything about the sport of Whacky-Flob and don't know the specific terms used. (What on earth are foo-whacks???) To get started, you should attend a few games of Whacky-Flob and observe how the game and the scoring works. This way, you'll be able to have a much better conversation with the customer since you have some knowledge about the problem domain. (Incidentally, this is one of the cool things about being a software engineer: you get exposure to all kinds of different, exciting problem domains. One project might be tracking grizzly bears, the next one might be identifying cyber terrorists or making a car drive itself.)

You should also never assume that a customer is familiar with technical terms that you might think everyone should know, such as JPEG, database or maybe even operating system. Something like "The metaclass subclass hierarchy was constrained to be parallel to the subclass hierarchy of the classes which are their instances" might make some sense to a software engineer, but a customer will just look at you very confused! One of the authors once took part in a customer interview where the stakeholder was asked if they want to use the system through a browser. Unfortunately, the customer had no idea what a browser was. Sometimes, customers may not want to admit that they have no idea what you're talking about and just say "Yes" to whatever you suggest. Remember, it's up to you to make sure you and your customer understand each other and that you get useful responses from your customer during the interview!

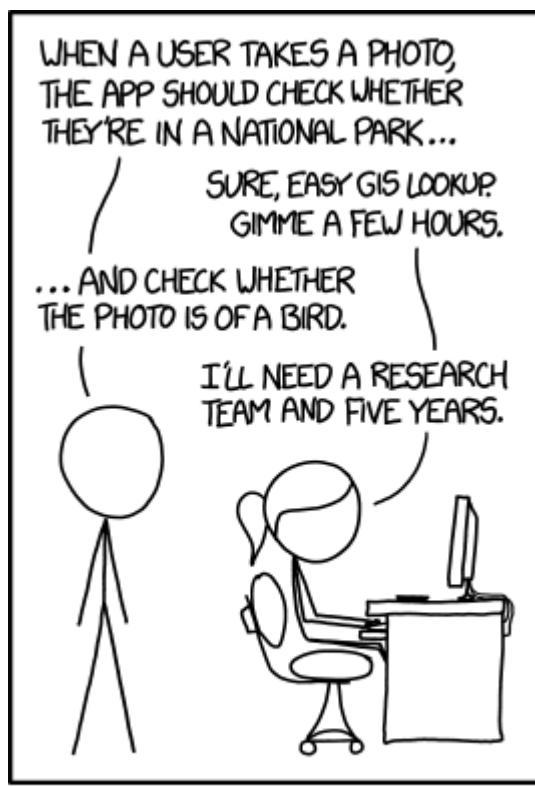


Even if you manage to communicate with a customer, you might find that they don't really know what they want the software to do or can't express it. They might say they want "software to improve their business" or to "make their work more efficient" but that's not very specific. (There's a great cartoon of [Dilbert](#) which illustrates this point!) When you show them the software you have built, they can usually tell you if that's what they wanted or what they like and don't like about it. For that reason, it's a good idea to build little prototypes while you're developing your system and keep showing them to customers to get feedback from them.

You'll often find that customers have a specific process that they follow already and want the software to fit in with that. We were once involved in a project being done by university

students for a library. Their staff used to write down information about borrowed items three times on a paper form, cut up the form and send the pieces to different places as records. When the students interviewed them, they asked for a screen in the program where they could enter the information three times as well (even though in a computer system there really isn't much point in that)!

Customers are usually experts in their field and are therefore likely to leave out information that they think is obvious, but may not be obvious to you. Other times, they do not really understand what can and cannot be done with computers and may not mention something because they do not realise that it is possible to do with a computer. Again, it's up to you to get this information from them and make sure that they tell you what you need to know.



[Image source](#)

**Curiosity:** Easy for computers and hard for humans vs hard for computers and easy for humans

The rollover text of the above image (you will need to actually view it on [xkcd's website](#)) is worth reading too. Image recognition is a problem that initially seemed straightforward, probably because humans find it easy. Interestingly, there are many

problems that computers find easy, but humans find challenging, such as multiplying large numbers. Conversely, there are many other problems that computers find hard, yet humans find easy, such as recognizing that the thing in a photo is, for example, a cat.

If you have multiple stakeholders, you can get conflicting viewpoints. For example, when you talk to the cafeteria people about your food-ordering app, they may suggest that every student should only be able to order food up to a value of \$10. In this way, they can avoid prank orders. When you talk to a teacher, they agree with this suggestion because they are worried about bullying. They don't want one student to get pressured into ordering food for lots of other students. But the students tell you that they want to be able to order food for their friends. In their view, \$10 isn't even enough for one student.

What do you do about these conflicting points of view? Situations like this can be difficult to handle, depending on the situation, the stakeholders and the software you are making. In this case, you need the support from the cafeteria and the teachers for your software to work, but maybe you could negotiate a slightly higher order limit of \$20 to try to keep all your stakeholders happy.

Finally, even if you get everything right during the analysis stage of your project, talk to all the stakeholders and find all the requirements for the software, requirements can change while you're making the software. Big software projects can take years to complete. Imagine how much changes in the technology world in a year! While you're working on the project, new hardware (phones, computers, tablets, ...) could come out or a competitor might release software very similar to what you're making. Your software itself might change the situation: once the software is delivered, the customer will try working with it and may realise it isn't what they really wanted. So you should never take the requirements for your software to be set in stone. Ideally, you should keep talking to customers regularly throughout the project and always be ready for changes in requirements!

### Project: Finding the requirements

For this project, you need to find someone for whom you could develop software. This could be someone from your family or a friend. They might, for example, need software to manage information about their business' customers or their squash club might want software to schedule squash tournaments or help with the timetabling of practices. (For this project, you won't actually be making the software, just looking at the requirements; if the project is small enough for you to program on your own, it's probably not big enough to be a good example for software engineering!)

Once you've found a project, start by identifying and describing the stakeholders for your project. (This project will work best if you have at least two different stakeholders.) Try to find all the stakeholders, remembering that some of them might only have an indirect interest in your software. For example, if you are making a database to store customer information, the customers whose information is being stored have some interest in your software even though they never use it directly; for example, they will want the software to be secure so that their data cannot be stolen. Write up a description about each stakeholder, giving as much background detail as possible. Who are they? What interest do they have in the software? How much technical knowledge do they have? ...

Interview one of the stakeholders to find out what they want the software to do. Write up the requirements for your software, giving some detail about each requirement. Try to distinguish between functional and non-functional requirements. Make sure you find out from your stakeholder which things are most important to them. This way you can give each requirement a priority (for example high, medium, low), so that if you would actually build the software you could start with the most important features.

For the other stakeholders, try to imagine what their requirements would be. In particular, try to figure out how the requirements would differ from the other stakeholders. It's possible that two stakeholders have the same requirements but in that case maybe they have different priorities? See if you can list any potential disagreements or conflicts between your stakeholders? If so, how would you go about resolving them?

## 16.3. Design: How do we build it?

Once you have decided what your software needs to be able to do, you can actually build it. But just blindly starting to program is likely to get you into trouble; remember that most software is huge and very complex. You need to somehow minimise the amount of complexity in software, otherwise it will become impossible to understand and maintain for other developers in the future.

Software design is all about managing this complexity and making sure that the software we create has a good structure. Before we start writing any code, we design the structure of our software in the *design* phase of the project. When you talk about software design, many people will think that you're talking about designing what the software will look like. Here, we're actually going to look at designing the *internal* structure of software.

So how can we design software in a way that it doesn't end up hugely complex and impossible to understand? Here, we give you an introduction to two important approaches: subdivision and abstraction. Those are pretty scary words, but as you'll see soon, the concepts behind them are surprisingly simple.

You can probably already guess what *subdivision* means: We break the software into many smaller parts that can be built independently. Each smaller part may again be broken into even smaller parts and so on. As we saw in the introduction, a lot of software is so large and complex that a single person cannot understand it all; we can deal much more easily with smaller parts. Large software is developed by large teams so different people can work on different parts and develop them in parallel, independently of each other. For example, for your cafeteria project, you might work on developing the database that records what food the cafeteria sells and how much each item costs, while your friend works on the actual phone app that students will use to order food.

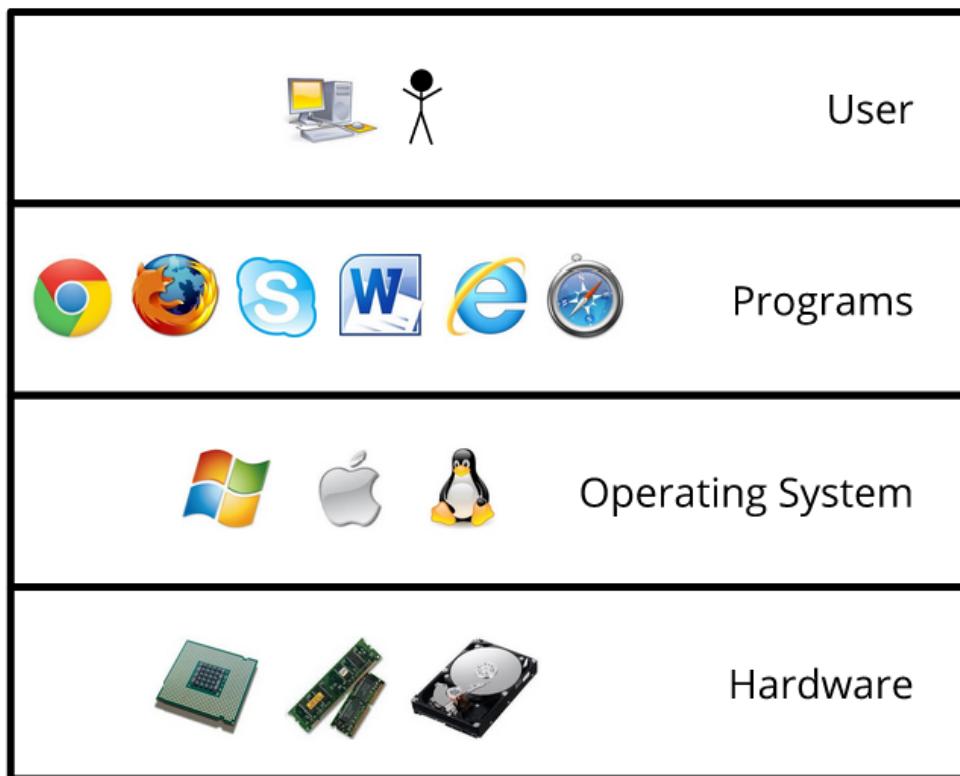
Once we have developed all the different parts, all we need to do is make them communicate with each other. If the different parts have been designed well, this is relatively easy. Each part has a so-called *interface* which other parts can use to communicate with it. For example, your part of the cafeteria project should provide a way for another part to find out what food is offered and how much each item costs. This way, your friend who is working on the phone app for students can simply send a request to your part and get this information. Your friend shouldn't need to know exactly how your part of the system works; they should just be able to send off a request and trust that the answer they get from your part is correct. This way, each person working on the project only needs to understand how their own part of the software works.

Let's talk about the second concept, *abstraction*. Have you ever thought about why you can drive a car without knowing how its engine works? Or how you can use a computer without knowing much about hardware? Maybe you know what a processor and a hard drive is but could you build your own computer? Could your parents? We don't need to know exactly how computers or cars work internally to be able to use them thanks to abstraction!

If we look more closely at a computer, we can see that it actually has a number of *layers* of abstraction. Right at the bottom, we have the hardware, including the processor, RAM, hard disk and various complicated looking circuit boards, cables and plugs.

When you boot your computer, you start running the operating system. The operating system is in charge of communicating with the hardware, usually through special driver software. Once you've started your computer, you can run programs, for example your browser. The browser actually doesn't communicate with the hardware directly but always goes through the operating system.

Finally, you're the top layer of the system. You use the program but you will (hopefully) never have to interact with the more complicated parts of the operating system such as driver software, let alone the hardware. In this way, you can use the computer without ever having to worry about these things.



## Your computer as a layered system

The computer can be broken down into multiple layers, starting with the user, then the programs, then the operating system, then finally the hardware.

We call a system like this a *layered system*. You can have any number of layers you want but each layer can only communicate with the one directly below it. The operating system can directly access the hardware but a program running on the computer can't. You can use programs but hopefully will never have to access the hardware or the more complex parts of the operating system such as drivers. This again reduces the complexity of the system because each layer only needs to know about the layer directly below it, not any others.

Each layer in the system needs to provide an interface so that the layer above it can communicate with it. For example, a processor provides a set of instructions to the operating system; the operating system provides commands to programs to create or delete files on the hard drive; a program provides buttons and commands so that you can interact with it.

One layer knows nothing about the internal workings of the layer below; it only needs to know how to use the layer's interface. In this way, the complexity of lower layers is completely hidden, or *abstracted*. Each layer represents a higher level of abstraction.

So each layer hides some complexity, so that as we go up the layers things remain manageable. Another advantage of having layers is that we can change one layer without affecting the others, as long as we keep the layer's interface the same of course. For example, your browser's code might change but you might never notice as long as the browser still looks and works the same as before. Of course, if the browser stops working or new buttons appear suddenly you know that something has changed.

We can have the same "layered" approach inside a single program. For example, websites are often designed as so-called *three-tier* systems with three layers: a database layer, a logic layer and a presentation layer. The database layer usually consists of a database with the data that the website needs. For example, Facebook has a huge database where it keeps information about its users. For each user, it stores information about who their friends are, what they have posted on their wall, what photos they have added, and so on. The logic layer processes the data that it gets from the database. Facebook's logic layer, for example, will decide which posts to show on your "Home" feed, which people to suggest as new friends, etc. Finally, the presentation layer gets information from the logic layer which it displays. Usually, the presentation layer doesn't do much processing on the information it gets but simply creates the HTML pages that you see.



## Facebook as a three-tier system

Facebook can be broken down into a three tier system, comprising of the presentations layer, then the logic layer, then finally the data layer.

### Curiosity: Reuse - Kangaroos and Helicopters

Since building software is so difficult and time-consuming, a popular idea has been to reuse existing software. Not surprisingly, we call this *software reuse*. It's a great idea in theory (why recreate something that already exists?) but turns out to be difficult to put into practice partly because existing software is also huge and complicated. Usually when you reuse software, you want only a small part of the existing software's functionality, rather than everything.

An interesting story that illustrates the problems with software reuse (although it is unfortunately not completely accurate, see <http://www.snopes.com/humor/nonsense/kangaroo.asp>) is that of helicopters and kangaroos. The Australian Air Force was developing a new helicopter simulator to train pilots. They wanted the simulator to be as realistic as possible and therefore decided to include herds of kangaroos in the simulation. To save time, they reused code from another simulator which included foot soldiers and simply changed the icons of the soldiers to kangaroos.

Once the program was finished, they demonstrated it to some pilots. One of the pilots decided to fly the helicopter close to a herd of kangaroos to see what would happen. The kangaroos scattered to take cover when the helicopter approached (so far so good) but then, to the pilot's extreme surprise, pulled out their guns and missile launchers and fired at the helicopter. It seemed the programmer had forgotten to remove *that* part of the code from the original simulator.

### Project: Designing your software

Think back to the requirements you found in the analysis project described above. In this project, we will look at how to design the software.

Start by thinking about how the software you are trying to build can be broken up into smaller parts. Maybe there is a database or a user interface or a website? For example, imagine you are writing software to control a robot. The robot needs to use its sensors to follow a black line on the ground until it reaches a target. The software for your robot should have a part that interacts with the sensors to get information about what they "see". It should then pass this information to another part, which analyses the data and decides where to move next. Finally, you should have a part of the software which interacts with the robot's wheels to make it move in a given direction.

Try to break down your software into as many parts as possible (remember, small components are much easier to build!) but don't go too far - each part should perform a sensible task and be relatively independent from the rest of the system.

For each part that you have identified, write a brief description about what it does. Then think about how the parts would interact. For each part, ask yourself which other parts it needs to communicate with directly. Maybe a diagram could help visualise this?

## 16.4. Testing: Did we Build the Right Thing and Does it Work?

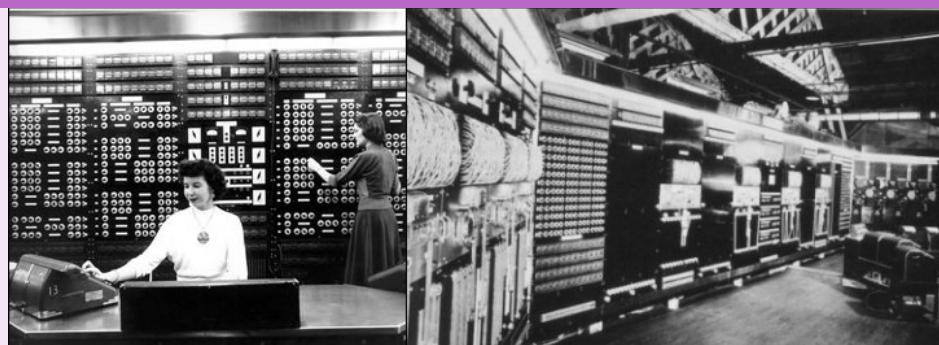
We've decided what our software should do (analysis) and designed its internal structure (design), and the system has been programmed according to the design. Now, of course, we have to test it to make sure it works correctly.

Testing is an incredibly important part of developing software. We cannot really release software that still has lots of bugs to our customers. (Well, we could but our customers wouldn't be very happy about it.) Remember that software bugs can have both very small and very large effects. On the less serious end of the scale, they might make a program difficult to use or crash your computer. On the other hand, they can cost millions of dollars and even endanger human life. More testing might have prevented the Ariane 5 failure or might have discovered the Therac bug that ended up killing three patients.

Unfortunately, testing is again really difficult because of the size and complexity of software. If a piece of software would take years to read and understand, imagine how long it would take to fully test it!

When we test software, we try lots of different inputs and see what outputs or behaviour the software produces. If the output is incorrect, we have found a bug.

### Curiosity: Bugs and Moths



The Mark II at Harvard

In 1947, engineers working on a computer called the *Mark II* were investigating a computer error and found that it was caused by a moth which had become trapped inside the computer! This incident is an early example of using the word *bug* to refer to computer errors. Of course, today we use the word to refer to errors in programs, rather than actual insects trapped in the computer.

The problem with testing is that it can only show the presence of errors, not their absence! If you get an incorrect output from the program, you know that you have found a bug. But if you get a correct output, can you really conclude that the program is correct? Not really. The software might work in this particular case but you cannot assume that it will work in other cases. No matter how thoroughly you test a program, you can never really be 100% sure that it's correct. In theory, you would have to test every possible input to your system,

but that's not usually possible. Imagine testing Google for everything that people could search for! But even if we can't test everything, we can try as many different test cases as possible and hopefully at least decrease the probability of bugs.

As with design, we can't possibly deal with the entire software at once, so we again just look at smaller pieces, testing one of them at a time. We call this approach *unit testing*. A unit test is usually done by a separate program which runs the tests on the program that you're writing. That way you can run the tests as often as you like --- perhaps once a day, or even every time there is a change to the program.

It's not unusual to write a unit test program before you write the actual program. It might seem like wasted work to have to write two programs instead of one, but being able to have your system tested carefully any time you make a change greatly improves the reliability of your final product, and can save a lot of time trying to find bugs in the overall system, since you have some assurance that each unit is working correctly.

Once all the separate pieces have been tested thoroughly, we can test the whole system to check if all the different parts work together correctly. This is called *integration testing*. Some testing can be automated while other testing needs to be done manually by the software engineer.

If I give you a part of the software to test, how would you start? Which test inputs would you use? How many different test cases would you need? When would you feel reasonably sure that it all works correctly?

There are two basic approaches you can take, which we call *black-box testing* and *white-box testing*. With black-box testing, you simply treat the program as a black box and pretend you don't know how it's structured and how it works internally. You give it test inputs, get outputs and see if the program acts as you expected.

But how do you select useful test inputs? There are usually so many different ones to choose from. For example, imagine you are asked to test a program that takes a whole number and outputs its successor, the next larger number (e.g. give it 3 and you get 4, give it -10 and you get -9, etc). You can't try the program for *all* numbers so which ones do you try?

You observe that many numbers are similar and if the program works for one of them it's probably safe to assume it works for other similar numbers. For example, if the program works as you expect when you give it the number 3, it's probably a waste of time to also try 4, 5, 6 and so on; they are just so similar to 3.

This is the concept of *equivalence classes*. Some inputs are so similar, you should only pick one or two and if the software works correctly for them you assume that it works for

all other similar inputs. In the case of our successor program above, there are two big equivalence classes, positive numbers and negative numbers. You might also argue that zero is its own equivalence class, since it is neither positive nor negative.

For testing, we pick a couple of inputs from each equivalence class. The inputs at the boundary of equivalence classes are usually particularly interesting. Here, we should definitely test -1 (this should output 0), 0 (this should output 1) and 1 (this should output 2). We should also try another negative and positive number not from the boundary, such as -48 and 57. Finally, it can be interesting to try some very large numbers, so maybe we'll take -2,338,678 and 10,462,873. We have only tested 7 different inputs, but these inputs will probably cover most of the interesting behaviour of our software and should reveal most bugs.

Of course, you might also want to try some invalid inputs, for example "hello" (a word) or "1,234" (a number with a comma in it) or "1.234" (a number with a decimal point). Often, test cases like these can get programs to behave in a very strange way or maybe even crash because the programmer hasn't considered that the program might be given invalid inputs. Remember that especially human users can give you all sorts of weird inputs, for example if they misunderstand how the program should be used. In case of an invalid input, you probably want the program to tell the user that the input is invalid; you definitely don't want it to crash!

Black-box testing is easy to do but not always enough because sometimes finding the different equivalence classes can be difficult if you don't know the internal structure of the program. When we do white-box testing, we look at the code we are testing and come up with test cases that will execute as many different lines of code as possible. If we execute each line at least once, we should be able to discover a lot of bugs. We call this approach *code coverage* and aim for 100% coverage, so that each line of code is run at least once. In reality, even 100% code coverage won't necessarily find all bugs though, because one line of code might work differently depending on inputs and values of variables in a program. Still, it's a pretty good start.

Unit testing is very useful for finding bugs. It helps us find out if the program works as we intended. Another important question during testing is if the software does what the *customer* wanted (Did we build the right thing?). *Acceptance testing* means showing your program to your stakeholders and getting feedback about what they like or don't like. Any mistakes that we made in the analysis stage of the project will probably show up during acceptance testing. If we misunderstood the customer during the interview, our *unit tests* might pass (i.e. the software does what we thought it should) but we may still have an unhappy customer.

Different stakeholders can be very different, for example in terms of technical skills, or even could have given us conflicting requirements for the software. It's therefore of course possible to get positive feedback from one stakeholder and negative feedback from another.

### Project: Acceptance Testing

For this project, choose a small program such as a Windows desktop app or an Apple dashboard widget. Pick something that you find particularly interesting or useful (such as a timer, dictionary or calculator). Start by reading the description of the program to find out what it does *before* you try it out.

Next, think about a stakeholder for this software. Who would use it and why? Briefly write down some background information about the stakeholder (as in the analysis project) and their main requirements. Note which requirements would be most important to them and why.

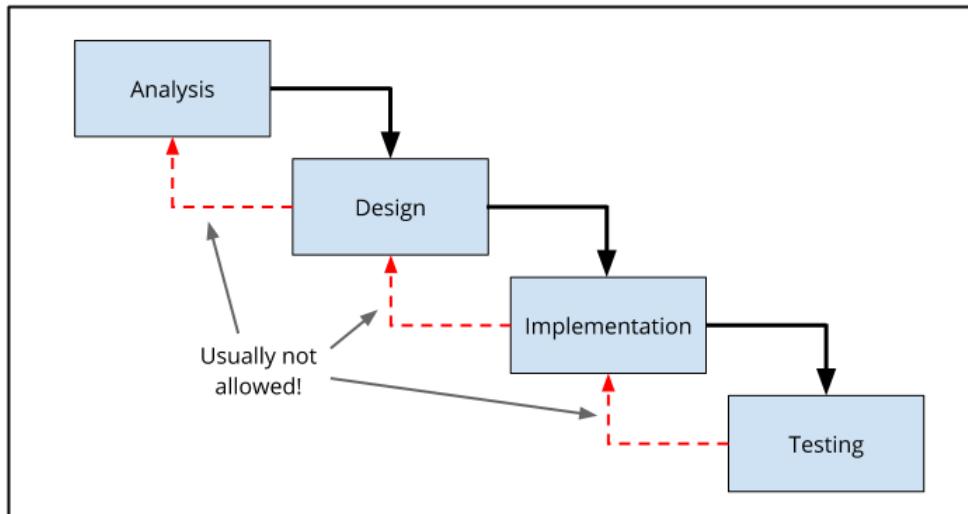
Now, you can go ahead and install the program and play around with it. Try to imagine that you are the stakeholder that you described above. Put yourself in this person's shoes. How would they feel about this program? Does it meet your requirements? What important features are missing? Try to see if you can find any particular problems or bugs in the program. (Tip: sometimes giving programs unexpected input, for example a word when they were expecting a number, can cause some interesting behaviour.)

Write up a brief acceptance test report about what you found. Try to link back to the requirements that you wrote down earlier, noting which have been met (or maybe partially met) and which haven't. Do you think that overall the stakeholder would be happy with the software? Do you think that they would be likely to use it? Which features would you tell the software developers to implement next?

## 16.5. Software processes

So far in this chapter, you've learned about different phases of software development: analysis, design and testing. But how do these phases fit together? At what time during the project do we do what activity? That's the topic of *software processes*.

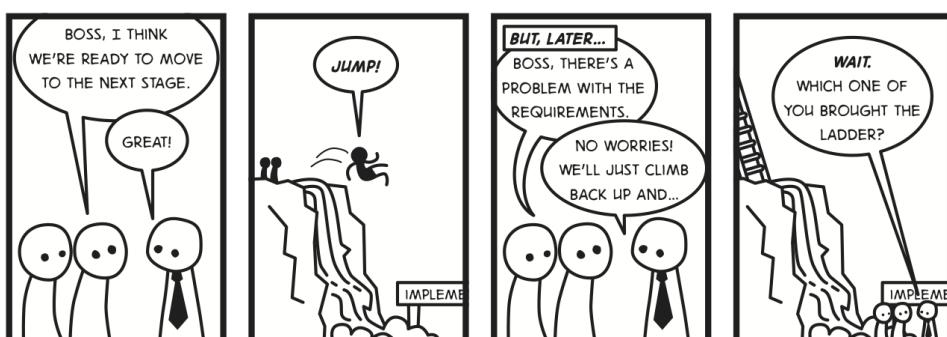
The obvious answer would be to start with analysis to figure out what we want to build, then design the structure of the software, implement everything and finally test the software. This is the simplest software process, called the *waterfall process*.



The waterfall process

The waterfall process is borrowed from other kinds of engineering. If we want to build a bridge, we go through the same phases of analysis, design, implementation and testing: we decide what sort of bridge we need (How long should it be? How wide? How much load should it be able to support?), design the bridge, build it and finally test it before we open it to the public. It's been done that way for many decades and works very well, for bridges at least.

We call this process the waterfall process because once you "jump" from one phase of the project to the next, you can't go back up to the previous one. In reality, a little bit of backtracking is allowed to fix problems from previous project phases but such backtracking is usually the exception. If during the testing phase of the project you suddenly find a problem with the requirements you certainly won't be allowed to go back and rewrite the requirements.



An advantage of the waterfall process is that it's very simple and easy to follow. At any point in the project, it's very clear what stage of the project you are at. This also helps with planning: if you're in the testing stage you know you're quite far into the project and should finish soon. For these reasons, the waterfall process is very popular with managers who like to feel in control of where the project is and where it's heading.

### Curiosity: Hofstadter's law

Your manager and customer will probably frequently ask you how much longer the project is going to take and when you will finally have the finished program. Unfortunately, it's really difficult to know how much longer a project is going to take. According to Hofstadter's law, "It always takes longer than you expect, even when you take into account Hofstadter's Law." Learning to make good estimates is an important part of software engineering.

Because it's just so nice and simple, the waterfall process is still in many software engineering textbooks and is widely used in industry. The only problem with this is that the waterfall process just does not work for most software projects.

So why does the waterfall process not work for software when it clearly works very well for other engineering products like bridges (after all, most bridges seem to hold up pretty well...)? First of all, we need to remember that software is very different from bridges. It is far more complex. Understanding the plans for a single bridge and how it works might be possible for one person but the same is not true for software. We cannot easily look at software as a whole (other than the code) to see its structure. It is not physical and thus does not follow the laws of physics. Since software is so different from other engineering products, there really is no reason why the same process should necessarily work for both.

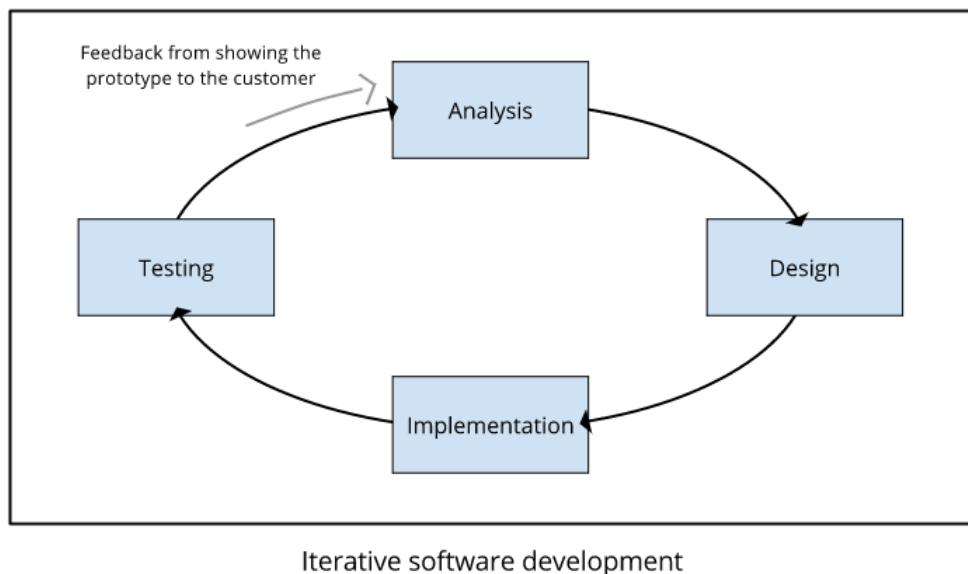
To understand why the waterfall process doesn't work, think back to our section about analysis and remember how hard it is to find the right requirements for software. Even if you manage to communicate with the customers and resolve conflicts between the stakeholders, the requirements could still change while you're developing the software. Therefore, it is very unlikely that you will get the complete and correct requirements for the software at the start of your project.

If you make mistakes during the analysis phase, most of them are usually found in the testing stage of the project, particularly when you show the customer your software during acceptance testing. At this point, the waterfall process doesn't allow you to go back and fix the problems you find. Similarly, you can't change the requirements halfway through the

process. Once the analysis phase of the project is finished, the waterfall process “freezes” the requirements. In the end of your project, you will end up with software that hopefully fulfills *those* requirements, but it is unlikely that those will be the *correct* requirements.

You end up having to tell the customer that they got what they asked for, not what they needed. If they've hired you, they'll be annoyed; if it's software that you're selling (such as a smartphone app), people just won't bother buying it. You can also get things wrong at other points in the project. For example, you might realise while you're writing the code that the design you came up with doesn't really work. But the waterfall process tells you that you have to stick with it anyway and make it work somehow.

So if the waterfall process doesn't work, what can we do instead? Most modern software development processes are based on the concept of iteration. We do a bit of analysis, followed by some design, some programming and some testing. (We call this one iteration.) This gives us a rather rough prototype of what the system will look like. We can play around with the prototype, show it to customers and see what works and what doesn't. Then, we do the whole thing again. We refine our requirements and do some more design, programming and testing to make our prototype better (another iteration). Over time, the prototype grows into the final system, getting closer and closer to what we want. Methodologies based on this idea are often referred to as *agile* -- they can easily adapt as changes become apparent.



The advantage with this approach is that if you make a mistake, you will find it soon (probably when you show the prototype to the customer the next time) and have the opportunity to fix it. The same is true if requirements change suddenly; you are flexible and can respond to changes quickly. You also get a lot of feedback from the customers as they slowly figures out what they need.

There are a number of different software processes that use iteration (we call them *iterative processes*); a famous one is the *spiral model*. Although the details of the different processes vary, they all use the same iteration structure and tend to work very well for software.

Apart from the question of what we do at what point of the project, another interesting question addressed by software processes is how much time we should spend on the different project phases. You might think that the biggest part of a software project is programming, but in a typical project, programming usually takes up only about 20% of the total time! 40% is spent on analysis and design and another 40% on testing. This shows that software engineering is so much more than programming.

Once you've finished developing your program and given it to the customer, the main part of the software project is over. Still, it's important that you don't just stop working on it. The next part of the project, which can often go on for years, is called *maintenance*. During this phase you fix bugs, provide customer support and maybe add new features that customers need.

### **Curiosity:** Brooks' law

Imagine that your project is running late and your customer is getting impatient. Your first instinct might be to ask some of your friends if they can help out so that you have more people working on the project. Brooks' law, however, suggests that that is exactly the wrong thing to do!

[Brooks' law](#) states that "adding manpower to a late software project makes it later." This might seem counterintuitive at first because you would assume that more people would get more work done. However, the overhead of getting new people started on the project (getting them to understand what you are trying to build, your design, the existing code, and so on) and of managing and coordinating the larger development team actually makes things slower rather than faster in the short term.

### **Project:** Fun with the Waterfall and Agile Processes

The waterfall process is simple and commonly used but doesn't really work in practice. In this activity, you'll get to see why. First, you will create a design which you then pass on to another group. They have to implement your design exactly and are not allowed to make any changes, even if it doesn't work!

You need a deck of cards and at least 6 people. Start by dividing up into groups of about 3-4 people. You need to have at least 2 groups. Each group should grab two chairs and put them about 30cm apart. The challenge is to build a bridge between the two chairs using only the deck of cards!

Before you get to build an actual bridge, you need to think about how you are going to make a bridge out of cards. Discuss with your team members how you think this could work and write up a short description of your idea. Include a diagram to make your description understandable for others.

Now exchange your design with another group. Use the deck of cards to try to build your bridge to the exact specification of the other group. You may not alter their design in any way (you are following the waterfall process here!). As frustrating as this can be (especially if you know how to fix the design), if it doesn't work, it doesn't work!

If you managed to build the bridge, congratulations to you and the group that managed to write up such a good specification! If you didn't, you now have a chance to talk to the other group and give them feedback about the design. Tell them about what problems you had and what worked or didn't work. The other group will tell you about the problems they had with your design!

Now, take your design back and improve it, using what you just learnt about building bridges out of cards and what the other group told you. You can experiment with cards as you go, and keep changing the design as you learn about what works and what doesn't (this is an agile approach, which we are going to be looking at further shortly). Keep iterating (developing ideas) until you get something that works.

Which of these two approaches worked best --- designing everything first, or doing it in the agile way?

### Project: A Navigation Language

In this activity, you will develop a language for navigating around your school. Imagine that you need to describe to your friend how to get to a particular classroom. This language will help you give a precise description that your friend can easily follow.

First, figure out what your language has to do (i.e. find the *requirements*). Will your language be for the entire school or only a small part? How exact will the descriptions be? How long will the descriptions be? How easy will they be to follow for someone who does / doesn't know your language? How easy will it be to learn? ...

Now, go ahead and *design* the language. Come up with different commands (e.g. turn left, go forward 10, ...). Make sure you have all the commands you need to describe how to get from one place in your school to any other!

Finally, *test* the language using another student. Don't tell them where they're going, just give them instructions and see if they follow them correctly. Try out different cases until you are sure that your language works and that you have all the commands that you need. If you find any problems, go back and fix them and try again!

Note down how much time each of the different phases of the project take you. When you have finished, discuss how much time you spent on each phase and compare with other students. Which phase was the hardest? Which took the longest? Do you think you had more time for some of the phases? What problems did you encounter? What would you do differently next time around?

### Project: Block Building (Precise Communication)

Communicating clearly with other software engineers and customers is essential for software engineers. In this activity, you get to practice communicating as precisely as possible!

Divide up into pairs, with one *creator* and one *builder* in each pair. Each person needs a set of at least 10 coloured building blocks (e.g. lego blocks). Make sure that each pair has a matching set of blocks or this activity won't work!

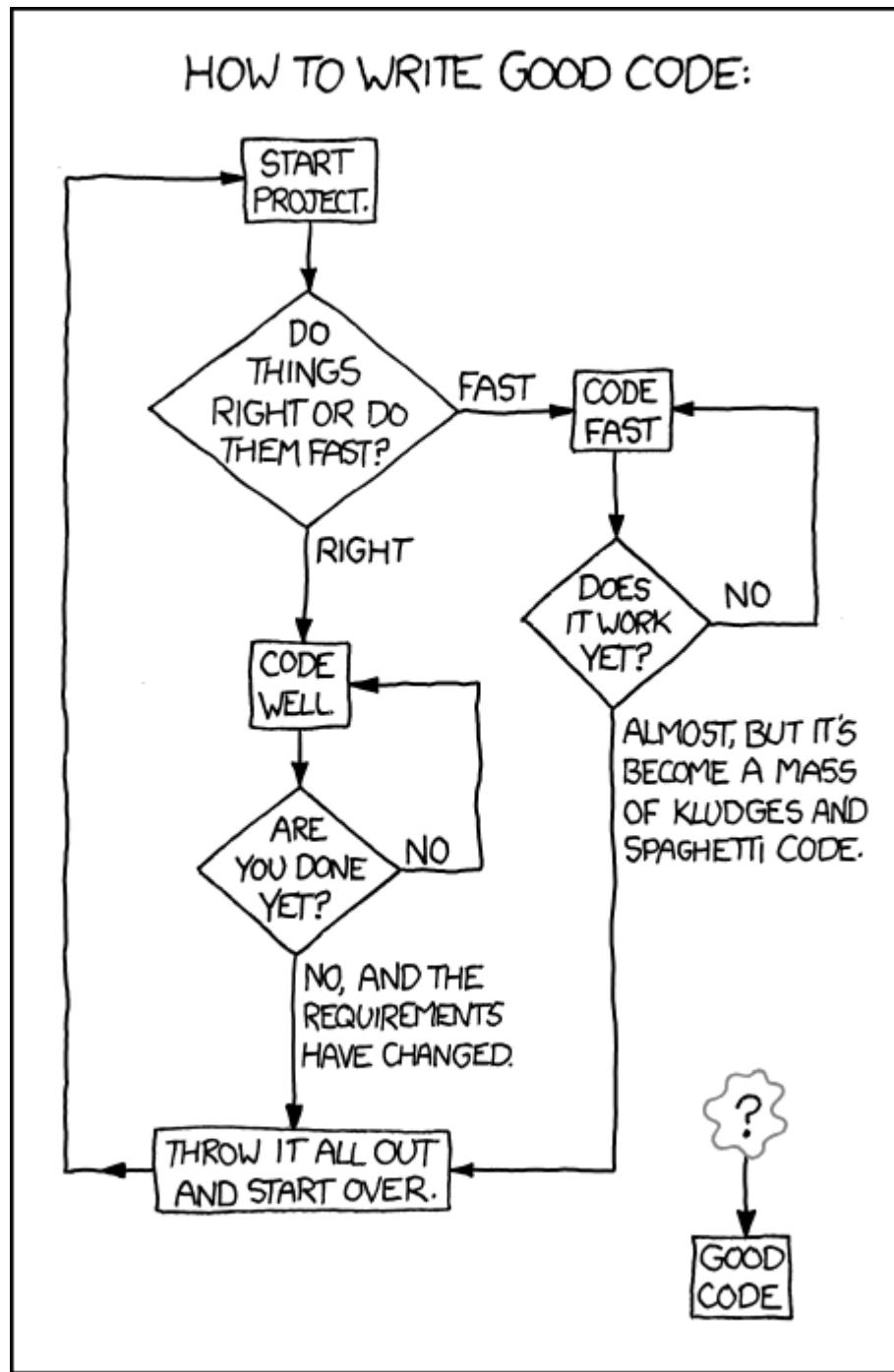
The two people in each pair should not be able to see each other but need to be able to hear each other to communicate. Put up a screen between the people in each pair or make them face in opposite directions. Now, the creator builds something with their blocks. The more creative you are the more interesting this activity will be!

When the creator has finished building, it's the builder's turn. His or her aim is to build an exact replica of the creator's structure (but obviously without knowing what it looks like). The creator should describe exactly what they need to do with the blocks. For example, the creator could say "Put the small red block on the big blue block" or "Stand two long blue blocks up vertically with a one block spacing between them, and then balance a red block on top of them". But the creator should not describe the building as a whole ("Make a doorframe.").

When the builder thinks they are done, compare what you built! How precise was your communication? Which parts were difficult to describe for the creator / unclear for the builder? Switch roles so that you get to experience both sides!

## 16.6. Agile software development

*Agile* software development has become popular over the last 10 years; two of the most famous agile processes are called [XP](#) and [Scrum](#). Agile software development is all about being extremely flexible and adaptive to change. Most other software processes try to manage and control changes to requirements during the process; agile processes accept and expect change. The following xkcd comic illustrates part of the apparent dilemma that agile processes aim to address. With Agile, we can develop software quickly, correctly, and be adaptive to change.



[Image source](#)

Agile processes work similarly to iterative processes in that they do a number of iterations of analysis, design, implementation and testing. However, these iterations are extremely short, each usually lasting only about 2 weeks.

In many other processes, documentation is important. We document the requirements so that we can look back at them; we document our design so that we can refer back to it when we program the system. Agile software processes expect things to change all the

time. Therefore, they do very little planning and documentation because documenting things that will change anyway is a bit of a waste of time.

Agile processes include lots of interesting principles that are quite different from standard software development. We look at the most interesting ones here. If you want to find out more, have a look at [Agile Academy on Youtube](#) which has lots of videos about interesting agile practices! There's also [another video here](#) that explains the differences between agile software development and the waterfall process.

Here are some general principles used for agile programming:

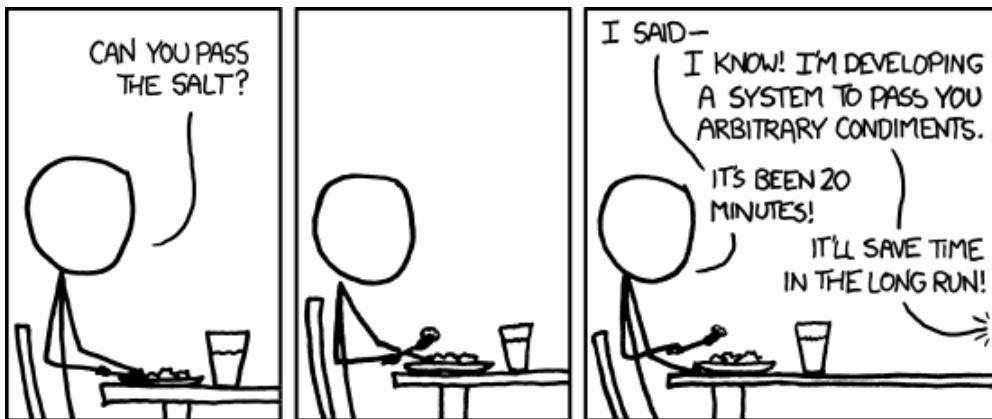
### 16.6.1. Pair-programming

Programming is done in pairs with one person coding while the other person watches and looks for bugs and special cases that the other might have missed. It's simply about catching small errors before they become bugs. After all, 4 eyes see more than 2.

You might think that pair-programming is not very efficient and that it would be more productive to have programmers working separately; that way, they can write more code more quickly, right? Pair-programming is about reducing errors. Testing, finding and fixing bugs is hard; trying not to create them in the first place is easier. As a result, pair-programming has actually been shown to be more efficient than everyone programming by themselves!

### 16.6.2. YAGNI

YAGNI stands for “You ain’t gonna need it” and tells developers to keep things simple and only design and implement the things that you know you are really going to need. It can be tempting to think that in the future you might need feature x and so you may as well already create it now. But remember that requirements are likely to change so chances are that you won’t need it after all.



[Image source](#)

You ain't gonna need it!

### 16.6.3. Constant testing

Agile processes take testing very seriously. They usually rely on having lots of automated unit tests that are run at least once a day. That way, if a change is made (and this happens often), we can easily check if this change has introduced an unexpected bug.

### 16.6.4. Refactoring

There are many different ways to design and program a system. YAGNI tells you to start by doing the simplest thing that's possible. As the project develops, you might have to change the original, simple design. This is called *refactoring*.

Refactoring means to change your design or implementation without changing the program's behaviour. After a refactoring, the program will work exactly the same, but will be better structured in some way. Unit tests really come in handy here because you can use them to check that the code works the same way before and after the refactoring.

Refactoring only works on software because it is "soft" and flexible. The same concept does not really work for physical engineering products. Imagine that when building a bridge, for example, you started off by doing the simplest possible thing (putting a plank over the river) and then continually refactored the bridge to get the final product.

### 16.6.5. Test-driven development

In standard software development, we first write some code and then test it. This makes sense: we need the code before we can test it, right? Test-driven development tells you to do the exact opposite!

Before you write a piece of code, you should write a test for the code that you are about to write. This forces you to think about exactly what you're trying to do and what special cases there are. Of course, if you try to run the test, it will fail (since the functionality it is testing does not yet exist). When you have a failing test, you can then write code to make the test pass.

## 16.6.6. Programmer welfare

Software developers should not work more than 40 hours per week. If they do overtime one week they should not do more overtime the following week. This helps keep software developers happy, productive, creative and energetic, and makes sure they don't get overworked.

## 16.6.7. Customer involvement

A customer representative should be part of the developing team (ideally spending full-time with the team), on hand to answer questions or give feedback at all times. This is important to be able to quickly change the requirements or direction of the project. If you have to wait 2 weeks until you can get feedback from your customer, you will not be able to adapt to change very quickly!

Although having a customer on the development team is a great idea in theory, it is quite hard to achieve in practice. Most customers simply want to tell you their requirements, pay you and then get the software delivered 5 months later. It's rare to find a customer who is willing and has the time to be more involved in the project. Sometimes companies will hire an expert to be part of the team; for example, a company working on health software might have a doctor on the team, or if they are working on educational software, they may hire a teacher. This sounds expensive, but since failed software can cost millions of dollars, paying the salary of an expert is a relatively small part of the overall cost, and much more likely to lead to success.

### Curiosity: Christopher Alexander

So far, we've mainly compared software development to engineering and building bridges, but you might have noticed that it's also pretty similar to architecture. In fact, software development (in particular agile software development) has borrowed a lot of concepts from architecture. An architect called Christopher Alexander, for example, suggested involving customers in the design process. Sound familiar? Several other suggestions from Christopher Alexander were also picked up by the agile development community and as a result his thinking about architecture has shaped

how we think about software development. This is despite the fact that Christopher Alexander knew nothing about software development. He was apparently very surprised when he found out how well known he is among software developers!

## 16.6.8. Courage

“Courage” might seem like an odd concept in the context of software development. In agile processes, things change all the time and therefore programmers need to have the courage to make changes to the code as needed, fix the problems that need to be fixed, correct the design where needed, throw away code that doesn’t work, and so on. This might not seem like a big deal, but it can actually be quite scary to change code, particularly if the code is complicated or has been written by a different person. Unit tests really help by giving you courage: you’ll feel more confident to change the code if you have tests that you can run to check your work later.

### Project: Software processes

This project will provide insight into a real software engineering process, but you'll need to find a software engineer who is prepared to be interviewed about their work. It will be ideal if the person works in a medium to large size company, and they need to be part of a software engineering team (i.e. not a lone programmer).

The project revolves around interviewing the person about the process they went through for some software development they did recently. They may be reluctant to talk about company processes, in which case it may help to assure them that you will keep their information confidential (your project should only be viewed by you and those involved in supervising and marking it; you should state its confidential nature clearly at the start so that it doesn't get published later).

You need to do substantial preparation for the interview. Find out about the kind of software that the company makes. Read up about software engineering (in this chapter) so that you know the main terminology and techniques.

Now prepare a list of questions for the interviewee. These should find out what kind of software development processes they use, what aspects your interviewee works on, and what the good and bad points are of the process, asking for examples to illustrate this.

You should take extensive notes during the interview (and record it if the person doesn't mind).

You then need to write up what you have learned, describing the process, discussing the techniques used, illustrating it with examples, and evaluating how well the process works.

## 16.7. The whole story!

In this chapter, we've tried to give you an introduction to the challenges of creating software and some techniques that software engineers use to overcome them. We've really only scratched the surface of software analysis, design, testing and software processes; there are entire books about each of these areas!

It can be difficult to understand the importance of some of the problems and techniques we have described here if you have never worked on a larger software project yourself. Some may seem blindingly obvious to you, others may seem irrelevant. When you work on your first large project, come back to this chapter and hopefully you'll recognise some of the problems we have described here!

## 16.8. Further reading

### 16.8.1. Useful Links

- [Wikipedia - Software engineering](#)
- [CS4FN - Software engineering](#)
- [Teach ICT - Systems Life Cycle](#)
- [Wikipedia - Software crisis](#)
- [IEEE - Why software fails](#)
- [Wikipedia - Software design](#)
- [Wikipedia - Abstraction](#)
- [Wikipedia - Software testing](#)
- [Wikipedia - Software development process](#)
- [Wikipedia - Waterfall model](#)
- [Wikipedia - Iterative and incremental development](#)
- [Wikipedia - Agile software development](#)
- [Wikipedia - Test driven development](#)

# 1.44 Assessment Guide

This document provides a brief introduction to teachers on the Computer Science Field Guide assessment guides for NCEA Achievement standard AS91074 (1.44).

## Topics

1.44 has bullet points for the following three topics in computer science:

- Algorithms
- Programming Languages
- Human Computer Interaction

Each of these topics has a chapter in the Computer Science Field Guide.

Currently, we provide two different assessment guides for algorithms (sorting and searching), and one for each of human computer interaction and programming languages. Note that students only need to follow one assessment guide for each of the three topics (i.e. they do not need to do both searching and sorting for the topic of algorithms).

The advice below applies to the whole standard; advice specific to each topic is available through [the main NCEA index](#).

## Sorting vs Searching for Algorithms

For the topic of algorithms, students can demonstrate their understanding of algorithms and their costs by using either sorting algorithms or searching algorithms.

For students who are weak at math, searching algorithms is probably the better choice. Sorting algorithms requires either being good at understanding trends from data in a table or understanding how to read trends from a graph in order to achieve merit or excellence, whereas the cost of searching algorithms can easily be seen by students carrying out the algorithms themselves.

Sorting algorithms provide a slightly richer range of possibilities, including more ways to demonstrate how they work in a student's report, and intriguing new approaches to a common and easily described task that may not have been obvious.

Guidance is given for achieved, merit, and excellence for both sorting algorithms and searching algorithms.

## Order of Topics

The three topics can be completed in any order, although the first bullet point in each level (comparing algorithms, programs, and informal instructions) is probably best left until both algorithms and programming languages have been completed, since those topics can provide examples to illustrate the points in the first bullet points.

Covering Human Computer Interaction first may make the Algorithms topic more relevant to students. In many cases, a not so good algorithm will take a second to run, whereas a better algorithm will take less than a tenth of a second. This is very significant in terms of a good user interface, so covering HCI first will make students more aware of issues like this.

In their report, it is important that the three topics are kept separate. Order does not matter, but the student should have three or four main headings (it is up to them whether or not they put the two parts of algorithms together), and keep all the material under the relevant headings. This will make it far easier for the marker to find the evidence they are looking for.

## Personalisation and Student Voice

It is important that students use personalised examples to base their explanations around, and that the explanations are in their own words, and based on their example (rather than being a paraphrase from Wikipedia, for example).

Personalised means that the student's example is different to their classmates. For example, they may have a program that prints their name or favourite saying, they may use a different number of items to sort or search through, their choice of the values being sorted or searched in examples is unique, and they may carry out their own usability exploration of a device they chose, and report on it in their own words.

If the teacher provides too many headings or leading questions for students to structure their work, this can reduce the opportunity for the report to reflect a personal understanding.

There is no reason for students to paraphrase Wikipedia in this achievement standard. All explanations should be based on their own examples.

# Report Length

It is important to note that the page limit given by NZQA is a limit - not a target. The markers prefer reports that are short and to the point. The requirements of the standard can easily be met within the limit.

The page limit for 1.44 is now 10 pages to cover the three topics. A possible breakdown that leaves one additional page is:

- Algorithms: 4 pages
- Programming Languages: 2 pages
- Human Computer Interaction: 3 pages

The assessment guides for the specific topics provide further guidance on how to stay within these limits. Students should be mindful of the recommended limits while they are working on their reports, in order to avoid having to delete work they put a lot of effort into.

Some hints to reduce total length and maximise readability:

- Only include what is relevant to the standard. While covering additional material in class is valuable for learning, additional content that doesn't demonstrate understanding of the topics and bullet points in the standard is only a distraction in the report.
- Resize screenshots and photos so that they are still readable, although don't take up unnecessary space. Use cropping to show the relevant parts of the image.
- Don't leave unnecessary space in the report. It both looks untidy and makes it more difficult for the marker to find what they are looking for.
- Use logical headings, and do not include a title page.
- A table of contents page is not necessary - your report should be organised in a way that the marker can easily find what they are looking for. For example, keep the three main sections (Algorithms, Programming Languages, and Human Computer Interaction) one after the other, and do not jump between them.

# Presenting the report

Students should regularly check over their report by trying to think of it from the marker's point of view. A common mistake is to put in graphs without labels on the axes, which can make it difficult for the marker to know what is being shown. Additionally make sure units are given for measurements (e.g. 5 seconds or 5 minutes?). If referring to colour in an image, don't print the report in black and white! Don't assume the marker will know

anything about the instructions that were given in class. Good explanations of what was done and why are essential.

If using examples, don't use ones taken from the Field Guide or other sources - students should make up their own. For sorting and searching, they should actually carry out the balance scales activity using either the field guide interactives or physical balance scales. For HCI, students can choose an interface to evaluate themselves.

## General Advice

In 2012 we did a study that looked over 151 student submissions for 1.44 in 2011. This was the first year 1.44 was offered, although the lessons learnt are still relevant, particularly for teachers teaching the standard for the first time. A WIPSCE paper was written presenting our findings of how well students approached the standard and our recommendations for avoiding pitfalls. Our key findings are reflected in the teacher guides, although reading the entire paper would be worthwhile.

The paper was Bell, T., Newton, H., Andreae, P., & Robins, A. (2012). The introduction of Computer Science to NZ High Schools --- an analysis of student work. In M. Knobelsdorf & R. Romeike (Eds.), The 7th Workshop in Primary and Secondary Computing Education (WiPSCE 2012). Hamburg, Germany. The [paper is available here](#).

# Algorithms (1.44) - Searching Algorithms

This is a guide for students attempting the *Algorithms* topic of digital technologies achievement standard 1.44 (AS91074). If you follow this guide, then you do **not** need to follow the sorting algorithms one.

In order to fully cover the standard, you will also need to have done projects covering the topics of *Programming Languages* and *Human Computer Interaction* in the standard, and included these in your report.

## Overview

The topic of *Algorithms* has the following bullet points in achievement standard 1.44, which this guide covers. This guide separates them into two categories.

### Comparing algorithms, programs, and informal instructions

**Achieved:** “describing the key characteristics, and roles of algorithms, programs and informal instructions”

**Merit:** “explaining how algorithms are distinct from related concepts such as programs and informal instructions”

**Excellence:** “comparing and contrasting the concepts of algorithms, programs, and informal instructions”

### Determining the cost of algorithms and understanding various kinds of steps in algorithms

**Achieved:** “describing an algorithm for a task, showing understanding of the kinds of steps that can be in an algorithm, and determining the cost of an algorithm for a problem of a particular size”

**Merit:** “showing understanding of the way steps in an algorithm for a task can be combined in sequential, conditional, and iterative structures and determining the cost of an iterative algorithm for a problem of size  $n$ ”

**Excellence:** “determining and comparing the costs of two different iterative algorithms for the same problem of size  $n$ ”

As with all externally assessed Digital Technology reports, you should base your explanations around personalised examples.

## Reading from the Computer Science Field Guide

You should read and work through the interactives in the following sections of the CS Field Guide in order to prepare yourself for the assessed project.

[2.1 - What's the big picture?](#)

[2.2 - Searching Algorithms](#)

## Project

This project involves understanding linear search and binary search.

### Writing your report for the main bullet points that cover algorithms

#### Achieved

Ensure you have tried both of the box searching interactives which are in the part of the field guide which you read. For one of them you had to use linear search, and for the other you had to use binary search.

Take a screenshot of a completed search using the *binary search* interactive (if you get lucky and find the target within 2 clicks, keep restarting until it takes at least 3, so that you something sufficient to show in your report). Show on your screenshot which boxes you opened, and put how many boxes you opened. The number of boxes you opened is the cost of the algorithm for this particular problem. Include your screenshot in your report.

Describe (in your own words with 1 - 3 sentences) the overall process you carried out to search through the boxes. Try and make your explanation general, e.g. if you gave the

instructions to somebody who needs to know how to search 100 boxes, or 500 boxes, the instructions would be meaningful.

You also need to show the kinds of steps that can be in an algorithm, such as iterative, conditional, and sequential. If you don't know what these terms mean, go have another look at the field guide. Get a Scratch program (or another language if you are fairly confident with understanding the language) that implements binary search. Take a screenshot of it, or a large part of it (you want to ensure that the screenshot takes up no more than half a page in the report, but is still readable) and open it in a drawing program such as paint. Add arrows and notes showing a part of the algorithm that is sequential, part that is conditional, and part that is iterative.

### **Merit/ Excellence**

It should be obvious from your initial investigation that binary search is far better than linear search! Although you still might say, why not just use a faster computer? To explore this possibility, you are now going to analyse what happens with a **huge** amount of data. More specifically, you are going to answer the following question: "*How do linear search and binary search compare when the amount of data to search is doubled?*"

Start by picking a really large number (e.g. in the billions, or even bigger - this is the amount of data that large online companies such as Google or Facebook have to search). Imagine you have this number of boxes that you have to search. Also, imagine that you then have two times that number of boxes, four times that number of boxes, eight times that number of boxes, and sixteen times that number of boxes.

Now, using those 5 different amounts of boxes, you are going to determine how many boxes would have to be looked at *on average* to find a target for linear search and binary search. You will then also calculate the amount of time you could expect it to take, using the average number of boxes to be looked at and an estimate of how many boxes a really fast computer could check per second. As you do the various calculations, you should add them into a table, such as the one below. This will be a part of your report.

<b>Values for n</b>	<b>Average for Linear Search</b>	<b>Average for Binary Search</b>	<b>Expected Time for Linear Search</b>	<b>Expected Time for Binary Search</b>
Chosen number	???	???	???	???

Values for n	Average for Linear Search	Average for Binary Search	Expected Time for Linear Search	Expected Time for Binary Search
Chosen number x 2	???	???	???	???
Chosen number x 4	???	???	???	???
Chosen number x 8	???	???	???	???
Chosen number x 16	???	???	???	???

Rather than actually carrying out the searching (the interactive is not big enough!), you are going to calculate the expected averages. Computer scientists call this *analysing* an algorithm, and often it is better to work out how long an algorithm can be expected to take before waiting years for it to run and wondering if it will ever complete. Remember that you can use the [big number calculator](#) and the [time calculator](#) in the field guide to help you with the math. If you are really keen, you could make a spreadsheet to do the calculations and graph trends.

Hint for estimating linear search: Remember that in the worst case, you would have to look at every box (if the target turned out to be the last one), and on average you'll have to check half of them. Therefore, to calculate the average number of boxes that linear search would have to look at, just halve the total number of boxes.

Hint for estimating binary search: Remember that with each box you look at, you are able to throw away half (give or take 1) of the boxes. Therefore, To calculate the average number of boxes that binary search would have to look at, repeatedly divide the number by 2 until it gets down to 1. However many times you divide by 2 is the average cost for binary search. Don't worry if your answer isn't perfect; it's okay to be within 3 or so of the correct answer. This means that if while halving your number it never gets down to exactly 1 (e.g. it gets down to 1.43 and then 0.715), your answer will be near enough. As long as you have halved your number repeatedly until it gets down to a number that is less than 1, your answer will be accurate.

Now that you have calculated the average number of boxes for each algorithm, you can calculate how long it would take on a high end computer for each algorithm with each problem size. Assume that the computer can look at 1 billion boxes per second. Don't worry about being too accurate (e.g. just round to the nearest millisecond (1/1000 of a second), second, minute, hour, day, month, or year). Some of the values will be a tiny fraction of a millisecond. For those, just write something like "Less than 1 millisecond". You can get the number by dividing the expected number of boxes to check by 1 billion.

You should notice some obvious trends in your table. Explain these trends, and in particular explain how the amount of time each algorithm takes changes as the problem size doubles. Does it have a significant impact on the amount of time the algorithm will take to run? Remember the original question you were asked to investigate: "*How do linear search and binary search compare when the amount of data to search is doubled?*".

Using your findings to guide you, discuss **one** of the following scenarios:

- Imagine that you are a data analyst with the task of searching these boxes, and in order to do your work you need to search for many pieces of data each day. What would happen if you were trying to use linear search instead of binary search?
- Imagine that you have a web server that has to search a large amount of data and then return a response to a user in a web browser (for example searching for a person on Facebook). A general rule of computer systems is that if they take longer than 1/10 of a second (100 milliseconds) to return an answer, the delay will be noticeable to a human. How do binary search and linear search compare when it comes to ensuring there is *not* a noticeable delay?

## Writing the part of your report for the other algorithms bullet points

### Achieved/ Merit/ Excellence

We recommend doing this part after you have done programming languages.

All three levels (A/M/E) are cleanly subsumed by the E requirement, so you should try to do that i.e. "comparing and contrasting the concepts of algorithms, programs, and informal instructions". You should refer to examples you used in your report or include additional examples (e.g. a program used as an example in the programming languages topic, or an

algorithm describing the searching process, etc). If you are confused, have another look at the field guide. You should only need to write a few sentences to address this requirement.

## Hints for success

- Don't confuse "algorithm cost" with the "algorithm length". The number of lines in the algorithm or program normally unrelated to the cost. Cost is the time the algorithm actually takes to run, or the number of comparisons that have to be made. You can find more information in the Field Guide if you are not sure.
- Resize screenshots/ photos so that they are large enough to see what is on them, but not taking up unnecessary space.

## Recommended Number of Pages

Within the 3 to 4 pages recommended for algorithms, a possible breakdown is:

- $\frac{1}{2}$  page: Screenshot of you carrying out binary search with the interactive. (**Achieved**)
- $\frac{1}{4}$  page: Explanation of your binary search screenshot. (**Achieved**)
- $\frac{1}{4}$  page: General instructions for carrying out binary search. (**Achieved**)
- $\frac{1}{2}$  page: Your example of the iterative, conditional, and sequential steps that can be in an algorithm (**Merit**)
- 1  $\frac{1}{2}$  pages: Your investigation and data collected for merit/ excellence. Including results and discussion (**Merit/ Excellence**)
- $\frac{1}{4}$  page to  $\frac{1}{2}$  page: Explanation of the difference between algorithms, programs, and informal instructions (**Achieved/ Merit/ Excellence**)

These are *maximums*, not targets!

For the topic of searching algorithms you probably won't need this much space (sorting algorithms tends to require more space).

Note that if you go over 4 pages for Algorithms, then you may have to use fewer pages for one of the other two topics, which could be problematic. No other material should be included for Algorithms.

# Algorithms (1.44) - Sorting Algorithms

This is a guide for students attempting the *Algorithms* topic of digital technologies achievement standard 1.44 (AS91074). If you follow this guide, then you do **not** need to follow the searching algorithms one.

In order to fully cover the standard, you will also need to have done projects covering the topics of *Programming Languages* and *Human Computer Interaction* in the standard, and included these in your report.

## Overview

The topic of *Algorithms* has the following bullet points in achievement standard 1.44, which this guide covers. This guide separates them into two categories.

### Comparing algorithms, programs, and informal instructions

**Achieved:** “describing the key characteristics, and roles of algorithms, programs and informal instructions”

**Merit:** “explaining how algorithms are distinct from related concepts such as programs and informal instructions”

**Excellence:** “comparing and contrasting the concepts of algorithms, programs, and informal instructions”

### Determining the cost of algorithms and understanding various kinds of steps in algorithms

**Achieved:** “describing an algorithm for a task, showing understanding of the kinds of steps that can be in an algorithm, and determining the cost of an algorithm for a problem of a particular size”

**Merit:** “showing understanding of the way steps in an algorithm for a task can be combined in sequential, conditional, and iterative structures and determining the cost of an iterative algorithm for a problem of size  $n$ ”

**Excellence:** “determining and comparing the costs of two different iterative algorithms for the same problem of size  $n$ ”

As with all externally assessed Digital Technology reports, you should base your explanations around personalised examples.

# Reading from the Computer Science Field Guide

You should read and work through the interactives in the following sections of the CS Field Guide in order to prepare yourself for the assessed project.

[2.1 - What's the bigger picture?](#)

[2.3 - Sorting Algorithms](#)

Note that 2.2 is not necessary for this project, as 2.2 focuses on *searching* algorithms, whereas this project focuses on *sorting* algorithms.

# Project

This project involves understanding how selection sort works and the types of steps that can be in it and other algorithms, and then comparing the cost of selection sort to the cost of quicksort.

## Writing your report for the main bullet points that cover algorithms

### Achieved

Carry out selection sort on a small amount of data. You can do this either using the balance scale interactive in the field guide (recommended), a physical set of balance scales if your school has them (normal scales that show the exact weights are unsuitable), or as a trace you did using pencil and paper (not recommended). Count how many comparisons you made to sort the items.

Take screenshots/ photos of you using the interactive or balance scales to do the sorting. Three or four pictures would be ideal (i.e. one showing the initial state of the scales and weights, one or two in the middle where you are comparing weights, and one at the end where all the weights are sorted). Use a drawing program to draw on each of the pictures and show which weights have been sorted so far, and which have not. Put on the screenshots how many comparisons have been made so far in the sorting process. Write a short explanation of what is happening in the images. Make sure you include the total number of comparisons that was needed to sort the items in your report.

Describe (in your own words with a few sentences) the overall process you carried out to sort the weights or numbers. Try and make your explanation general, e.g. if you gave the instructions to somebody who needs to know how to sort 100 numbers, or 500 numbers, the instructions would be meaningful.

You also need to show the kinds of steps that can be an algorithm, such as iterative, conditional, and sequential. If you don't know what these terms mean, go have another look at the field guide. Get a Scratch program (or another language if you are fairly confident with understanding the language) that implements selection sort. Take a screenshot of it, or a large part of it (you want to ensure that the screenshot takes up no more than half a page in the report, but is still readable) and open it in a drawing program such as paint. Add arrows and notes showing a part of the algorithm that is sequential, part that is conditional, and part that is iterative.

## Merit

Remember that some algorithms are a lot faster than others, especially as the size of the problem gets bigger. It isn't necessarily the case that if you try to sort twice as many items then it will take twice as long. As a quick warm up investigation to give you some idea of this, try the following.

Get an implementation of selection sort (there are some linked to at the end of the chapter in the field guide). Start by choosing a number between 10 and 20. How many comparisons does it take to sort that many randomly generated numbers with your chosen algorithm? Now, try sorting twice as many numbers. How many comparisons did it take now? Does it take twice as many? Now, try sorting 10 times as many numbers. Does it take 10 times as many comparisons? How many more times the original problem size's number of comparisons does it actually take? Hopefully you are starting to see a trend here.

*If you aren't attempting excellence, include the numbers you got from the warm up investigation, along with an explanation of the trend you found. If you are attempting*

*excellence, you should do the warm up investigation as it will help you (and will only take a few minutes), but you don't need to write about it.*

## Excellence

Choose 10 to 20 numbers in the range of 1 to 1000 (you will need a good variety of numbers, some high and some low. Do not pick the same numbers as your classmates!) For each of your 10 numbers, try sorting that many values with each of the sorting algorithms. Record your results in a table that has a column for the problem size, a column for how many comparisons selection sort used, and a column for how many comparisons quicksort used.

The best way of visualising the data you have just collected is to make a graph (e.g. using Excel). Your graph should have 2 lines; one for quicksort and one for selection sort, showing how the number of comparisons increases as the size of the problem goes up. Make sure you label the graph well. A simple way of making the graph is to use a scatter plot and put in lines connecting the dots (make sure the data for the graph is increasing order with the smallest problem sizes first and largest last so that the line gets drawn properly). Ask your teacher for guidance if you are having difficulty with excel.

Look at your graph. Does the rate of increase for the two algorithms seem to be quite different? Discuss what your graph shows. If you aren't sure what to include in the discussion of your findings, you could consider the following questions.

- What happens to the number of comparisons when you double how many numbers you are sorting with quicksort? What about when you sort 10 times as many numbers? How is this different to when you used selection sort at the start?
- What is the largest problem you can solve within a few seconds using selection sort? What about with quicksort?
- If you had a database with 1 million people in it and you needed to sort them by age, which of the two algorithms would you choose? Why? What would happen if you chose the other algorithm?

## Writing the part of your report for the other algorithms bullet points

### Achieved/ Merit/ Excellence

We recommend doing this part after you have done programming languages.

All three levels (A/M/E) are subsumed by the E requirement, so you should try to do that i.e. "comparing and contrasting the concepts of algorithms, programs, and informal instructions". You should refer to examples you used in your report or include additional

examples (e.g. a program used as an example in the programming languages topic, or an algorithm describing the sorting process, etc). If you are confused, have another look at the field guide. You should only need to write a few sentences to address this requirement.

## Hints for success

- Don't confuse "algorithm cost" with the "algorithm length". The number of lines in the algorithm or program normally unrelated to the cost. Cost is the time the algorithm actually takes to run, or the number of comparisons that have to be made. You can find more information in the Field Guide if you are not sure.
- While we recommend using the balance scales interactive (or real balance scales), if you do instead decide to include a pen and paper trace, don't give yourself more than 5 or 6 values to sort, and use an efficient layout that ensures the entire trace takes no more than about half a page.
- Resize screenshots/ photos so that they are large enough to see what is on them, but not taking up unnecessary space.
- Be sure to label the axis of your graph clearly so that the marker knows what your graph shows.

## Recommended Number of Pages

Within the 4 pages we recommend for algorithms, a possible breakdown is:

- 1  $\frac{1}{4}$  pages: Screenshots and explanations of you carrying out a chosen algorithm and determining the cost of it for your example problem (**Achieved**)
- $\frac{1}{4}$  page: General instructions for carrying out your chosen algorithm (**Achieved**)
- $\frac{1}{2}$  page: Example of the iterative, conditional, and sequential steps that can be in an algorithm (**Merit**)
- 1  $\frac{1}{2}$  pages: Your investigation and data collected for merit/ excellence. Including results and discussion (**Merit/ Excellence**)
- $\frac{1}{4}$  page to  $\frac{1}{2}$  page: Explanation of the difference between algorithms, programs, and informal instructions (**Achieved/ Merit/ Excellence**)

These are *maximums*, not targets!

Note that if you go over 4 pages for Algorithms, then you may have to use fewer pages for one of the other two topics, which could be problematic. No other material should be included for Algorithms.

# Human Computer Interaction (1.44)

This is a guide for students attempting Human Computer Interaction in digital technologies achievement standard 1.44 (AS91074).

In order to fully cover the standard, you will also need to have done projects covering the topics of Algorithms and Programming Languages, and included these in your report.

## Overview

Human Computer Interaction has the following bullet points in achievement standard 1.44, which this guide covers.

**Achieved:** “describing the role of a user interface and factors that contribute to its usability”

**Merit:** “explaining how different factors of a user interface contribute to its usability”

**Excellence:** “discussing how different factors of a user interface contribute to its usability by comparing and contrasting related interfaces”

As with all externally assessed reports, you should base your explanations around personalised examples, so that the marker can be confident that your report is your own work.

## Reading from the Computer Science Field Guide

You should read and work through the interactives in the following sections of the CS Field Guide in order to prepare yourself for the assessed project.

Start by reading both of these introduction sections. They will give you a general overview of what Human Computer Interaction is all about.

### [What's the Big Picture?](#)

## Users and Tasks

Then read one (or both if you're keen) of these sections on usability, in order to understand the kinds of things you will be looking for in your usability evaluation.

### Interface Usability

### Usability Heuristics

# Project

In this project, you will carry out a usability evaluation by observing a helper carry out a specific task on an interface you have chosen. While you could theoretically do the task yourself and write down where you had difficulty, it is surprisingly challenging to notice and be objective of usability issues you are facing yourself.

## Choosing an interface

It is essential that the interface you choose is one that your helper is not already familiar with.

Because you will need to compare related interfaces for excellence, make sure you choose an interface for which you will also be able to find a second related interface to compare with (e.g. two different alarm clocks or two different flight booking systems). The second interface should also be one that your helper is not familiar with (otherwise they may be biased towards the one they are familiar with).

In addition to choosing interfaces, you should also choose a specific piece of functionality within the interfaces. For example, the texting capabilities of a cellphone, file chooser on a computer, or drawing a simple picture in a drawing app. Focussing on something like "iPhone vs Samsung Phone", or "Windows vs Macintosh" are too general as there are thousands, if not millions, of aspects to those interfaces. You would need to pick specific apps or programs within them, and then a specific piece of functionality within the apps or programs.

Some possible pairs of interfaces you could use are: (Although remember that this list is far from exhaustive)

- Online booking systems for two different airlines (e.g. Air NZ vs Jetstar).
- Two different friends' cell phones.

- Two different email clients you have never used before (don't forget about the many webmail clients. Even signing up for webmail addresses could prove to be challenging in some cases).
- Two different microwaves. Cheap microwaves are notorious for being inconsistent and illogical to use. [Note that running a microwave with nothing in it will damage it! You would be best to put something inside it while you are experimenting with its interface. Water in a microwave safe glass is fine]
- Two different apps/ programs/ for setting an alarm (many exist). You could choose ones that go on a phone or on your computer, or one of each. A physical alarm clock would be good.
- Two different drawing programs you have never used before.

Note that an interface you (or your helper) designed yourself is unsuitable because you will know how it works in great detail.

## Choosing a task with the interface

Once you have chosen an interface, you need to think of one or two common tasks that are carried out with your chosen interface. The tasks should be specific. Some tasks (depending on the interfaces you chose) could be:

- Setting an alarm that will ring at 4:25am tomorrow to catch an early flight (or for a more sophisticated interface, at 7:25am on Monday, Tuesday, Wednesday, and Friday i.e. all weekdays except Thursday, which perhaps you have to get up at 6:30 AM to make it to a really early meeting).
- Sending a text to a friend that says "What are you doing at 3pm today? Want to go for coffee? :-)" (Symbols are good to include in the message, as these can be challenging to find on some interfaces).
- Changing a phone background to a photo you found online.
- Heating some food or water in a microwave for 1 minute, 20 seconds.
- Booking the cheapest flight that will arrive before 11 AM in Auckland from Christchurch, on the next Saturday (stop once you get to the part that asks for payment details!).
- Draw a smiley face with a drawing program. Put your name below the smiley face.

## Carrying out a usability evaluation

Be sure to read this entire assessment guide carefully (including the sections on writing your report and the hints for success) before beginning this step.

The two projects in the HCI chapter ("Think aloud protocol" and "Cognitive walkthrough") provide detailed procedures of how to do an evaluation using a widely used approach. We recommend choosing one of these (depending on the kind of interface).

Whichever approach you take, tell your helper what the task is, and give them one of your chosen interfaces so that they can carry out the task. While they are carrying out the task, you should be observing and keeping notes on the steps they take, paying particular attention to any points at which they are confused, select an incorrect option (or menu), have to use trial and error (e.g. they know the setting they want is probably in one of three menus, but have to check all three), something they didn't expect happens, wasted time following a dead end, or knew what to do due to useful prompts on the interface (e.g. meaningful icons or naming). Ideally, they will be verbalising their thought process while attempting the task (as in the think-aloud protocol), although keep in mind that some people find this challenging to do.

If you have more than one task for the interface, repeat the above process for each task. Also, if you have a second interface and are aiming for Excellence, repeat the process with the other interface and the same tasks. Remember to keep thorough notes on the entire process. You will need them to write a report that describes and explains what you've observed.

## Writing your report

In order to satisfy the requirements of the standard, you should do the following and include all your answers in your report.

### Achieved/ Merit

Start by writing an introduction to your report. The introduction should specify what interface(s) you have chosen, and what task(s) your helper will be carrying out with them. Briefly explain what your chosen interface(s) are for, and the kinds of steps you would expect to be included in carrying out your task(s). For example, specifying who to send a text to, choosing a suitable flight time, comparing prices of similar items in an e-shop, or setting the time and/ or ring tone for an alarm. A photo or screenshot of your interface(s) and relevant aspects of it are useful to include (but be sure to read the advice near the end of this guide on including images in your report).

You should write your introduction before you do the usability evaluation. By initially thinking about what you would expect to be able to do on the interface(s) for your task(s), you will be in a better position at the end to evaluate whether or not the interface(s) lived up to your expectations.

Now think back to sections 3.3 and/or 3.4 of the book and look over the notes you took during the usability evaluation for your chosen interface(s). Explain the negative characteristics of the interface(s) which caused your helper difficulties. Also explain the positive characteristics of the interface(s) which made it easier for your helper. Be sure to briefly describe the context of each characteristic (e.g. what was the user trying to accomplish at the time? What were they expecting to see happen)? If you have two interfaces, then write up two examples for each interface. If you have one interface, then write up three or four examples for it.

Remember that short and concise paragraphs are far better than long winded rambling. You might have noticed ten different usability issues, but instead of writing about all of them, it is better to pick the three or four that are most likely to come up and are the most serious, and explain them well.

### **Excellence**

In order to meet the excellence criteria, you need to "discuss how different factors of a user interface contribute to its usability by comparing and contrasting related interfaces". Therefore, you now need to discuss how the usability of the two interfaces compares. What was different between the two interfaces? Which interface did your helper find the easiest to use? Which did they prefer using? Why? If you were designing an interface that could be used for the same task, but was better than both the interfaces you investigated, which ideas would you take from each interface? Which ideas would you stay away from?

Keep in mind that interface design is really challenging to get completely right, and even the best interfaces still have usability issues. In addition, there are often trade-offs (e.g. not all features can be listed on the outermost menu). This is why companies such as Apple, Samsung, and Google put so much money into interface design. The implication of this for you, the one writing a report about usability, is that there are not necessarily any "right" answers. Therefore, you should just focus on explaining and justifying the points you make, and not worrying about whether or not your views are what the marker "wants" to see.

## Hints for success

- Be careful to talk about interface usability rather than just features. For example, a cell phone might have a fancy camera able to take very high resolution photos (a feature), but what we're interested in is how easy it actually is for somebody to take a photo with the camera (a usability factor), especially how easy it is to go from having the phone in your pocket to getting the photo, or from taking the photo to sharing or printing it.

- If your helper struggles to complete the task with the interface, it is likely to be because the interface was not designed well for them. This gives you great material for your project - look for the reasons they had trouble and don't blame them, as it isn't their fault.
- Don't evaluate an interface you designed yourself. As we said in the book, the designer knows the interface really well, and is the worst person to evaluate it!
- The page limit given by NZQA for the length of your report includes your work on algorithms and programming languages. The limit provides enough space to write an excellent report, but to avoid blowing out the page length:
- If you write concisely and clearly, you may be able to cover all the requirements with a page or less of writing (excluding pictures). This is fine, and in fact desirable for the marker as long as you have covered all the requirements.
- Try to keep photos/ screenshots large enough to see, but not so large that they take up needless amounts of space. Be careful about the resolution of them (we see far too many illegible images due to low resolution), and preferably print in colour. If you cannot print in colour, be sure that your images are clear in black and white. Ask your teacher for advice on sizing and positioning of images if you are having trouble.

## Recommended Number of Pages

Within the 3 pages we recommend for Human Computer Interaction, you should include a few images of your interface, and some explanations of the usability factors. For the project outlined above, a possible breakdown is:

- Up to  $\frac{1}{2}$  page of text introducing the topic, your chosen interface, and chosen tasks. (**Achieved**)
- Up to 1 page of text explaining the usability factors identified in the usability evaluation.
- Up to  $\frac{1}{2}$  page of text discussing comparing the two interfaces (**Excellence**)
- Up to  $\frac{1}{2}$  to 1 page worth of images (mixed with the above); ensure they are shrunk down in a way that they are still legible but not wasting space.

These are *maximums*, not targets!

The key to this topic is writing succinctly. Be careful to not ramble. You might not be able to include everything you wanted to; this is okay. Just prioritise and focus on the most interesting 2 or 3 issues for each interface. It is quite possible to use 2 or fewer pages (including images) in this topic, and infact some students have done an amazing job with just one page!

Note that if you go over 3 or 4 pages for Human Computer Interaction, then you may have to use fewer pages for one of the other two topics, which could be problematic.

No other material should be included for Human Computer Interaction. For example, don't include a list of heuristics explaining each one, or a list of general usability factors. You should only describe factors that directly relate to your chosen user interfaces.

# Programming Languages (1.44)

This is a guide for students attempting the *Programming Languages* topic of digital technologies achievement standard 1.44 (AS91074).

In order to fully cover the standard, you will also need to have done projects covering the topics of *Algorithms* and *Human Computer Interaction*, and included these in your report.

## Overview

Programming Languages has the following bullet points in achievement standard 1.44, which this guide covers. Note that merit is split into two bullet points.

**Achieved:** “describing the role and characteristics of programming languages, including the different roles and characteristics of high level languages and low level (or machine) languages, and the function of a compiler”

**Merit:** “explaining how the characteristics of programming languages, including the different characteristics of high level and low level (or machine) languages, are important for their roles” and “explaining the need for programs to translate between high and low level languages”

**Excellence:** “comparing and contrasting high level and low level (or machine) languages, and explaining different ways in which programs in a high level programming language are translated into a machine language

As with all externally assessed reports, you should base your explanations around personalised examples.

## Reading from the Computer Science Field Guide

You should read and work through the interactives and activities in the following sections of the CS Field Guide in order to prepare yourself for the assessed project.

[4.1 - What's the Big Picture?](#) (and an introduction to what programming is, intended for those of you with limited programming experience)

[4.2 - Machine Code \(Low Level Languages\)](#)

[4.3 - A Babel of Programming Languages \(High Level Languages\)](#) (optional, don't spend too much time on this)

[4.4 - How does the Computer Process your Program? \(Compilers and Interpreters\)](#)

It is very important that you actually do the activities in 4.2 (and 4.1 if you don't know much about programming).

# Project

This project consists of three main components. The first involves making a couple of examples. The second involves investigating the differences between high level and low level languages using your examples, and then the third involves investigating the different ways that high level languages can be converted to low level languages.

## Making Examples for your Report

You will need two examples of programs to include in your report; one that is in a high level language and one that is in a low level language. For the high level language example, you should use a program you wrote yourself, or make a small modification of a high level language program from the field guide. For the low level language example, you should make a small modification to one of the programs from the field guide (as long as you worked through the exercises in the low level languages section, you should be able to modify one of the programs without difficulty).

Include your program examples in your report as either screenshots or plain text. Note that the Hints for Success section has some advice on displaying code in a report which you should read first.

Briefly explain what each of the programs does (ideally you should have run them). e.g. does it add numbers, or does it print some output?. What output do your programs give? You do not need to explain how it does it (i.e. no need to explain what each statement in

the program does). The purpose of this is to show the marker that you do know what your example does, as opposed to just copying code you know nothing about.

## High and Low level languages (Achieved/Merit/ Excellence)

In order to make your answers to the following questions really clear (and to make it obvious that this is your own work), you should quote a few lines of your code examples which illustrate the points you make (e.g. some code that is cryptic and some code that you can tell easily what it does).

What is the main difference(s) you see between the high level language and the low level language? Why would a human not want to program in the language shown in your low level programming language example? What made modifying the low level programs in the field guide challenging? Given that a human probably doesn't want to program in a low level language, why do we need low level programming languages at all? What is their purpose?

When you wrote your high level program (or modified an existing program), what features of the language made this easier compared to when you attempted to modify the low level program? Why are there many different high level programming languages?

## Compilers and Interpreters

### Achieved/ Merit

If you have a compiler for the language your high level program example is written in, how would you use it to allow the computer to run your program? (Even if your language is an interpreted one, such as Python, just explain what would happen if you had a compiler for it, as technically a compiler can be written for any language, and there are infact compilers for Python). What is the purpose of the compiler?

### Excellence

What about an interpreter? How does the interpreter's function differ from a compiler in the way interpreted programs and compiled programs are run? Which is mostly used?

Here are some ideas for comparing compilers and interpreters: One way to consider the difference is to explain what happens if a program is transferred from one computer to another. Does it still run on the other computer? Does someone else need the same compiler or interpreter to run your software? Can you type in each line of a program and have it executed as you type it, or does the whole program have to be available before it can be run?

# Hints for Success

- You should easily be able to explain the concepts in half to one page of writing (in addition to the program examples). Any more than this is probably unnecessary.
- Don't use large programs in the examples. Keep it to 5 to 10 lines (slightly fewer is okay!) for the high level program, and a bit more for the low level program. A good trick for displaying the low level program without wasting space is to use 2 columns, because the low level language statements are so short (you could remove the comments in the code). If using a screenshot, get 2 screenshots with roughly half the program each and put them side by side, and if using text directly in report, just format it to 2 columns.
- If displaying the program examples as plain text in your report, then make the font size smaller for the code to try and prevent lines splitting (8pt or 9pt should be fine, as long as your explanations in the rest of your report are using the font size that NZQA requires!) Preferably use a fixed width font for program code as a variable width font can mess up the layout.
- If displaying the program examples as screenshots and the editor background is darker than the text colour, invert the colours using an image editor so as to make it easier to read on paper, and not waste black ink/toner!
- Paraphrasing definitions of high level languages, low level languages, compilers, and interpreters from Wikipedia or another site is not satisfactory for the standard. The marker needs to see what you understand, not what Wikipedia understands! You can show your understanding by explaining the ideas using your own examples.
- Overall, you should expect to spend less time on this part of the standard than on the Algorithms and Human Computer Interaction.

# Recommended Number of Pages

Within the 2 pages we recommend using for programming languages, a possible breakdown is:

- $\frac{1}{2}$  page: Example of low level program (If it is larger than this, you've probably put too much or not resized it as well as you could have)
- $\frac{1}{2}$  page: Example of high level program (If it is larger than this, you've probably put too much or not resized it as well as you could have)
- $\frac{1}{2}$  page: High Level and Low Level languages discussion
- $\frac{1}{2}$  page: Compilers and Interpreters discussion

These are *maximums*, not targets!

Note that if you go over 2 pages for Programming Languages, then you may have to use fewer pages for one of the other two topics, which could be problematic. No other material should be included for Programming Languages.

# 2.44 Assessment Guide

This document provides a brief introduction to teachers on the Computer Science Field Guide assessment guides for NCEA Achievement standard AS91371 (2.44).

While there has previously been a recommendation that the same device be used for all aspects of this standard, this can limit the range of observations that students can make (for example, some interfaces that are good to evaluate don't make it easy to find out how they represent data or might not use an aspect of encoding). If a student chooses to use a common theme then that is fine, but if their choice of device or theme doesn't have the richness or transparency to see how all aspects work, it is better to use different devices or examples for different aspects of the standard. Also note that the examples do not have to be related to a "device", for example it is fine to evaluate the interface of an interactive website for Human Computer Interaction or look at check digits on credit cards for Error Control Coding.

## Topics

2.44 has bullet points for the following topics in computer science.

- Representing Data using Bits
- Encoding (split into 3 sub topics)
  - Compression
  - Error Control Coding
  - Encryption
- Human Computer Interaction (different to 1.44)

Each of these topics has a chapter in the Computer Science Field Guide, which this assessment guide is based on.

There are multiple assessment guides for representing data and the encoding topics, of which students need to do a subset. The following explanations outlines what students should cover.

# Representing Data using Bits

Students should choose **two** data types. To get achieved, they should give examples for both their data types of the data type being represented using bits. To get merit, they should show two different representations using bits for each data type, and then compare and contrast them. This topic does not have excellence requirements. For this reason, students going for excellence should put more time into the discussions for encoding and human computer interaction than representing data using bits.

The following table shows common types of data that students could choose. For **achieved**, they should choose two rows in the table, and do what is in the achieved column for their chosen rows. For **merit** they should satisfy the achieved criteria, and additionally choose one data type in the merit column for each of their chosen rows, to compare with their ones from the achieved columns.

Data Type	Achieved	Merit
<i>Binary Numbers (Whole numbers)</i>	<i>Positive numbers</i>	<i>Negative numbers (simple sign bit) or Floating point or Twos complement</i>
<i>Characters/ Text</i>	<i>ASCII</i>	<i>Unicode</i>
<i>Images/ Colours</i>	<i>24 bit colour</i>	<i>Colour with fewer bits</i>
Sound	WAV file representation (16 bit, 44KHz)	Higher or lower quality sound (24 bit, 8 bit) and/or different sample rates

Note that data types and representations currently covered in the field guide are in italics. *Binary numbers* is a prerequisite for colours, and are recommended for all students. Students who struggle with binary numbers should just aim to represent a few numbers in binary (e.g. their age, birthday, etc) and then move onto representing text.

For example: for **achieved**, a student might choose the *Characters/ Text* and *Binary Numbers* rows, and therefore show examples of *ASCII* and *Positive Numbers* in their report. Another student who hopes to get at least **merit** in the standard might pick the same two rows, and therefore will cover *ASCII* and *Positive Numbers*, but they will also show examples of *Unicode* and *Floating Point Numbers* (they could have picked any of the many suggestions in that row). They will then compare *Unicode* and *ASCII*, and then *Floating Point Numbers* and *Positive Numbers*. They should **not** do comparisons across data types

(e.g. they should not compare a text representation with a number representation, as that does not make sense to do).

Most of the data types are based on binary numbers. Therefore, all students will need to learn how to represent whole numbers in binary before writing their report. However, they do not have to choose Binary Numbers as one of their two topics. They can just learn to represent whole numbers in binary, and then move on to using the numbers to representing Images/ Colours or Sound. Most students will find those topics much more interesting to evaluate, and easier to satisfy the merit criteria with - they can actually hear and see the varying Sound and Image qualities that using fewer or more bits leads to, and therefore include this in their discussions.

Doing *Characters/Text* is strongly recommended, because it is the only representation that is not based on binary numbers, and therefore gives students a wider understanding of the topic of data representation. It is also one of the easier data types to understand the representations for.

Generally, the students who are only aiming for achieved will be best picking binary numbers (Positive Numbers) and text (ASCII), as these are the most straightforward data representations.

Another issue is that hexadecimal is not a good example for students to use as a different representation of data, as it is simply a shorthand for binary. Writing a number as 01111010 (binary) or 7A (hexadecimal) represents exactly the same bits stored on a computer with exactly the same meaning; the latter is easier for humans to read and write, but both are 8-bit representations that have the same range of values. It is a useful shorthand, but shouldn't be used as a second representation for a type of data, or as a different type of data.

## Encoding

Students need to describe each of the three encoding topics in order to get achieved, and additionally they need to do a more in-depth project on one of the three topics in order to get merit or excellence.

Students should choose a subset of the provided projects that cover **one** of the following options. The first three options are for students aiming for merit/ excellence. The fourth option is for students just aiming to get achieved.

Topic	Option 1	Option 2	Option 3	Option 4
Compression	Up to Excellence	Only Achieved	Only Achieved	Only Achieved
Encryption	Only Achieved	Up to Excellence	Only Achieved	Only Achieved
Error Control Coding	Only Achieved	Only Achieved	Up to Excellence	Only Achieved

Note that some assessment guides provide projects that cover only achieved, and others go to excellence. Students should choose appropriate assessment guides based on the option they have chosen. It is best to do **one** topic to the excellence level and to focus on doing a really good job, as opposed to doing a not so good job on two or three.

At the excellence level students are required to evaluate "a widely used system for compression coding, error control coding, or encryption". The guides discuss some widely used systems, but it is worth noting that only one system *has* to be considered (e.g. JPEG is a widely used compression system, so evaluating JPEG would be sufficient; an alternative would be checksums used in bar codes). The evaluation would need to involve a comparison with *not* using the system, so for JPEG it might be with a RAW or BMP file; for bar codes, it would be to consider what would be different if a check digit isn't used. In some cases, it might make sense to compare the chosen widely used system with a mediocre alternative (that isn't widely used). One example where this would work is comparing the RSA crypto-system (widely used) with Caeser Cipher (no longer used in practice).

One issue to be aware of is that the data representation section includes reducing the number of bits (the "bit depth") for images and sound to reduce the space that they take. This overlaps with the idea of compression, but is should *not* be used for the compression part of the standards, as it's a very crude way to reduce file size, but not generally regarded as a compression method that takes advantage of the content of a file to make it smaller. For example, students could use examples of images with 16-bit and 24 colour to illustrate two representations of a type of data for data representation, but they should use an image compression method like JPEG, GIF and PNG to illustrate compressing image files. However, findings from reducing bit depth may be very useful to consider when evaluating a real image compression method, as it will allow students to see the lower loss of quality when the bits are cut using a "smart" method rather than simply truncating them.

## Human Computer Interaction

Human Computer Interaction is straightforward, and we provide one 2.44 guide for it. Note that the requirements for 2.44 HCI are different to 1.44 HCI.

In particular, students only need to discuss **one** interface. They should not discuss a second one or attempt to do comparisons, as this implies they did not read the requirements of the standard.

Reciting well known examples of heuristics (e.g. you turn off the Windows computer by clicking the "Start" button) is not a good idea, because it is poorly personalised.

## Order of Topics and Presentation

The three topics can be completed in any order, although encoding is best covered after representing data.

It may be a good idea to cover Human Computer Interaction (HCI) first, as students should already have some familiarity with it if they did 1.44, and it can help if they are able to start work on it early, then work on the other topics, and come back to HCI once they've had a while to reflect on the issues.

In their report, it is important that the material for each topic is kept together. It does not matter what order "Representing Data using Bits", "Encoding", and "Human Computer Interaction" are presented in the report, as long as there is one main heading for each, with all the relevant material below it. The three encoding topics should all be under "Encoding", and separated into three subheadings of "Encryption", "Compression", and "Error Control Coding". It would also be best to have two subheadings under "Representing Data using Bits" - one for each of the two chosen data types. Following a logical structure such as this ensures the marker can easily find their way around the report, and will not overlook anything. This also helps to show that the student understands the similarities and differences between the various topics.

## Personalisation and Student Voice

It is important that students use personalised examples to base their explanations around, and that the explanations are in their own words, and based on their example (rather than being a paraphrase from wikipedia, for example).

Personalised means that the student's example is different to their classmates. For example, they may represent their age or name using bits, carry out the parity trick (error

control coding) with a friend choosing random combinations and take photos, and they may carry out their own usability exploration of a device or website they chose, and report on it in their own words.

# Report Length

It is important to note that the page limit given by NZQA is not a target. The markers prefer reports that are short and to the point. Additionally, students and teachers have complained that there is too much work in computer science reports. In many cases, this seems to be students and teachers doing far more than what was needed for the standard.

A possible breakdown is:

- Representing Data using Bits: 2 - 4 pages (varies depending on data types, colours might require 2 pages, whereas binary numbers or text might only require 1 page)
- Encoding: 4 - 5 pages (1 for each achieved project and 2 - 3 for the achieved/ merit/ excellence project)
- Human Computer Interaction: 1 - 4 pages (High achieving students can write very concisely)

Note that the breakdown assumes the student is aiming for excellence - those only going for achieved will need less. Also note the wide recommended range of 7 pages to 13 pages. It is the top excellence students who will be able to cover the standard in just 7 pages. There is no gain in trying to bulk the length of the report. A report that satisfies the requirements for achieved in 5 pages may be more likely to get an achieved than one that is 14 pages bulked out with irrelevant material and copypasta. Including unnecessary material implies a lack of understanding.

It is also worth noting that some of the best examples we have seen of students doing the Human Computer Interaction were done in under 1 page. These high achieving students were able to concisely cover the requirements by writing 3 or 4 paragraphs following a pattern along the lines of: *A problem I/ my observer found with the device was . . I would fix this by*. Teachers should keep this in mind, as we've seen cases where the teacher thought the student had not written enough and therefore couldn't possibly be at the excellence level, when in fact it is beyond the minimum requirement for excellence!

Some hints to reduce total length:

- Only include what is relevant to the standard. While covering additional material in class is valuable for learning, additional content that doesn't demonstrate understanding of the topics in the standard is only a distraction in the report.

- Resize screenshots and photos so that they are still readable, although don't take up unnecessary space. Use cropping to show the relevant parts of the image. Students should ask their teacher for advice if they are ensure about whether or not their screenshots and photos are well sized.
- Don't leave unnecessary spaces in the report. It both looks untidy and makes it more difficult for the marker to find what they are looking for.
- Don't include a cover page or a table of contents page. A cover page is unnecessary (a heading at the top is fine), and the report should be laid out in a logical way that makes the table of contents unnecessary.

# Achieved Guide for Data Representation (2.44) - Numbers and Text

This is an achieved level guide for students attempting data representation in digital technologies achievement standard 2.44 (AS91371). It is only suitable for students who aren't aiming to get more than achieved.

Note that this is the *only* guide you need to follow to satisfy the data representation requirements up to achieved. It covers two different data types - numbers and text.

In order to fully cover the standard, you will also need to have done projects covering the three encoding topics up to the achieved level (error control coding, encryption, and compression), and a project covering human computer interaction, and include these in your report.

## Overview

The topic of Data Representation has the following achieved bullet point in achievement standard 2.44, which this guide covers.

**Achieved:** “describing ways in which different types of data can be represented using bits”

As with all externally assessed reports, you should base your explanations around personalised examples.

## Reading from the Computer Science Field Guide

You should read and work through the interactives in the following sections of the CS Field Guide in order to prepare yourself for the assessed project.

Read all of these sections, as they give the necessary introduction of the topic

## What's the Big Picture?

[Representing numbers with bits](#)

[Representing text with bits](#)

# Project

Start this section by writing an introduction to the topic of data representation. Describe what a "bit" is, and why computers use bits to represent data. This introduction only needs to be a couple of sentences - you are just showing the marker that you understand what a "bit" is, and how "bits" are used to represent data. It must be in your own words, based on what you understood in class (e.g. do not paraphrase a definition).

Next, you are going to show an example of how the day and month (number form) of your birthday are represented in binary. If you don't want to include your real birthday in your report, you can make one up, but it must not be the same birthday that somebody else in your class is using. For each of the two numbers, show the working you use to get the binary representation, the final binary representation using 1's and 0's, and a description of what you did to get the binary representation.

Finally you are going to show one other type of representation - the ASCII representation, which is used to represent strings of text. You will do this by showing your first name and favorite food in ASCII. Find an ASCII table (there is one in the field guide). Now, use the table to convert both the words to ASCII. Briefly describe how you used the ASCII table to convert these words to ASCII. You should *not* include the ASCII table in your report.

## Hints for success

- Put a main heading for Data Representation (your report will also have main headings for Encoding and Human Computer Interaction) and subheadings for Representing Numbers and Representing Text. Be careful to put your explanations and examples under the correct headings.
- Do not include large lists or tables, for example do not include an ASCII table or a list of binary number conversions.
- This project should not take up more than 1 to 1½ pages in your report.

Removed from section on numbers in programming Languages

# Achieved Guide for Compression (2.44) - Run Length Encoding

This is a guide for students attempting compression (one of the three encoding subtopics) in digital technologies achievement standard 2.44 (AS91371)

This is an achieved level guide.

Remember that you only need to do one of the three encoding topics (compression, encryption, and error control coding) to the excellence level. If you are either not interested in getting more than achieved, or are doing either encryption or error control coding to the excellence level, then this is the right guide for you. If you were wanting to do compression up to the excellence level, then you should select the alternate excellence guide instead.

In order to fully cover the standard, you will also need to have done projects covering the topics of encryption and error control coding to at least the achieved level (with one of them to the excellence level if you are attempting to get more than achieved), and projects covering the topics of representing data using bits and human computer interaction, and include these in your report.

## Overview

Encoding has the following *achieved* bullet points in achievement standard 2.44 which this guide covers.

**Achieved:** “describing the concept of encoding information using compression coding, error control coding, and encryption; and typical uses of encoded information”

As with all externally assessed reports, you should base your explanations around personalised examples.

# Reading from the Computer Science Field Guide

You should read and work through the interactivities in the following sections of the CS Field Guide in order to prepare yourself for the assessed project.

Read all of these sections, as they give the necessary introduction of the topics

[7.1 - What's the Big Picture?](#)

[7.2 - Run Length Encoding](#)

## Project

Start this section by writing an introduction to the topic of compression. *Briefly* explain what compression is, what it is used for, and what kinds of problems would exist if there was no such thing as compression. This introduction only needs to be a few sentences - you are just showing the marker that you understand the bigger picture of what compression is, and some of the typical uses of it.

Now you are going to make an example of compression in action to include in your report. Start by making a grid of squares (any size is fine, but it should be at least 6x6) and draw a picture by filling some of the squares with black and leave others white. Underneath (or alongside each row), show how a computer could represent your image using Run Length Encoding. You should not worry about how it is represented at the bit level. It is fine to just use normal numbers which are comma separated, like what was done in the field guide chapter on compression.

Count how many characters are needed to represent your image in its original form (i.e. how many squares does it contain?). Count how many characters were used in your Run Length Encoding representation. Don't forget to include the commas! (check the field guide or read the teacher note if you don't understand why we say you must count the commas). How well did Run Length Encoding compress your image? You might choose to use [the Run Length Encoding interactive](#).

*Briefly* describe what Run Length Encoding is, relating back to the example you have just put in your report. Regardless of whether you use the interactive or calculate the run length encoding by hand, describe how you arrived at your answer for a couple of lines in the image. The marker needs to be able to know that you understand the example, and what it is of.

# Hints for success

- Do a realistic image or pattern rather than randomly selecting squares for your example. You want to illustrate that the compression is useful rather than show an unrealistic case where it did more harm than good. Checkerboards or random layouts of black and white squares tend to not compress very well (which is fine, as normally we don't want to store them anyway!).
- Make sure your example is shrunk down enough to not waste space, but that the individual black and white squares and any numbering are still clearly visible.
- Put compression in its own section (your report should have suitable headings and subheadings for each topic to make it clear for the marker) and ensure that you briefly introduce the topic. It is important that your report clearly demonstrates that you know the difference between encryption, error control coding, and compression, and what their different purposes are.

# Achieved Guide for Error Control Coding (2.44) - Check Sums

This is a guide for students attempting error control coding (one of the three encoding subtopics) in digital technologies achievement standard 2.44 (AS91371)

This is an achieved level guide.

Remember that you only need to do one of the three encoding topics (compression, encryption, and error control coding) to the excellence level. If you are either not interested in getting more than achieved, or are doing either encryption or compression to the excellence level, then this is the right guide for you. If you were wanting to do error control coding up to the excellence level, then you should select the alternate excellence guide instead.

In order to fully cover the standard, you will also need to have done projects covering the topics of encryption and compression to at least the achieved level (with one of them to the excellence level if you are attempting to get more than achieved), and projects covering the topics of representing data using bits and human computer interaction, and include these in your report.

## Overview

Encoding has the following bullet points in achievement standard 2.44 which this guide covers.

**Achieved:** “describing the concept of encoding information using compression coding, error control coding, and encryption; and typical uses of encoded information”

As with all externally assessed reports, you should base your explanations around personalised examples.

# Reading from the Computer Science Field Guide

You should read and work through the interactivities in the following sections of the CS Field Guide in order to prepare yourself for the assessed project.

Read all of these sections, as they give the necessary introduction of the topics

[9.1 - What's the Big Picture?](#)

[9.3 - Check digits on barcodes and other numbers](#)

## Project

Start this section by writing an introduction to the topic of error control coding. *Briefly* explain what error control coding is, what it is used for, and what kinds of problems would exist if there was no such thing as error control coding. Briefly describe what a check digit is, and how it fits into the larger topic of error control coding. This introduction only needs to be a few sentences - you are just showing the marker that you understand the bigger picture of what error control coding is, and some of the typical uses of it.

Now you are going to show an example of error control coding in action to include in your report. Get the packaging for a food you like, ensuring it has a barcode on it. Take a photo of the barcode and include it in your report, with a small caption saying what the product is and that you are going to use it to investigate check digits.

Enter the barcode number into the interactive in the field guide, and take a screenshot of the interactive showing that the barcode number is valid. Change *one* digit of the barcode number in the interactive and show that the interactive now says it is invalid.

*Briefly* describe how the barcode number checker interactive was able to determine whether or not the barcode number was valid.

## Hints for success

- Put error control coding in its own section (your report should have suitable headings and subheadings for each topic to make it clear for the marker) and ensure that you briefly introduce the topic. It is important that your report clearly demonstrates that you know the difference between encryption, error control coding, and compression, and what their different purposes are.

- Be sure to shrink down examples so they do not take up too much space. A barcode only needs to be big enough for the numbers to be readable; it does not need to take up half a page!
- This project should not take up more than one 1 page in your report. If it is going beyond a page then you have probably either not shrunk down your barcodes as much as you could, or are doing more than you need to.

# Achieved Guide for Error Control Coding (2.44) - Parity

This is a guide for students attempting error control coding (one of the three encoding subtopics) in digital technologies achievement standard 2.44 (AS91371)

This is an achieved level guide.

Remember that you only need to do one of the three encoding topics (compression, encryption, and error control coding) to the excellence level. If you are either not interested in getting more than achieved, or are doing either encryption or compression to the excellence level, then this is the right guide for you. If you were wanting to do error control coding up to the excellence level, then you should select the excellence guide that uses check digits instead. Note that we do not currently have an excellence guide for parity.

In order to fully cover the standard, you will also need to have done projects covering the topics of encryption and compression to at least the achieved level (with one of them to the excellence level if you are attempting to get more than achieved), and projects covering the topics of representing data using bits and human computer interaction, and include these in your report.

## Overview

Encoding has the following bullet points in achievement standard 2.44 which this guide covers.

**Achieved:** “describing the concept of encoding information using compression coding, error control coding, and encryption; and typical uses of encoded information”

As with all externally assessed reports, you should base your explanations around personalised examples.

# Reading from the Computer Science Field Guide

You should read and work through the interactives in the following sections of the CS Field Guide in order to prepare yourself for the assessed project.

Read all of these sections, as they give the necessary introduction of the topics

[What's the Big Picture?](#)

[The Parity Magic Trick](#)

## Project

Start this section by writing an introduction to the topic of error control coding. *Briefly* explain what error control coding is, what it is used for, and what kinds of problems would exist if there was no such thing as error control coding. Briefly describe what parity bits are, and how they fit into the larger topic of error control coding. This introduction only needs to be a few sentences - you are just showing the marker that you understand the bigger picture of what error control coding is, and some of the typical uses of it.

You will need to choose somebody else (e.g. a classmate, or even somebody at home) to be your helper. You take the role of the magician, and they take the role of laying out the initial grid (before you add parity bits) and flipping a card while you are not looking.

Carry out the parity trick with your helper, taking the following photos.

- The original 5x5 grid that your helper lays out (you might want to take this photo at the very end if your helper hasn't seen the trick before - you don't want to draw attention to what you are doing and ruin it for them!)
- The grid after you have added the parity bits.
- The grid after your helper has flipped a card (edit the image afterwards, circling the flipped card).

For each of your photos, you will need to write (briefly, remember this is only at the achieved level) what was done. Where you added the parity bits, you should describe how you knew which way up to put the cards. For the final photo where you identify the flipped card, you should describe how you knew which it was.

Finish this section by *briefly* (one or two sentences) describing what the cards represent, and what flipping a card represents. This should be straightforward - you are simply linking the parity trick back to actual computers.

## Hints for success

- Put error control coding in its own section (your report should have suitable headings and subheadings for each topic to make it clear for the marker) and ensure that you briefly introduce the topic. It is important that your report clearly demonstrates that you know the difference between encryption, error control coding, and compression, and what their different purposes are.
- You will need to get a good balance between displaying photos which are clear, but not using too much space. Ask your teacher for advice on this if you are unsure.
- This project will take up slightly more space than the other encoding achieved projects. As long as you are following our other maximum page length recommendations (which are below the 14 pages maximum), this shouldn't be an issue. With your photos and explanations, you could expect it to take between 1 and 2 pages.

# Achieved Guide for Encryption (2.44) - Caesar Cipher

This is a guide for students attempting encryption (one of the three encoding subtopics) in digital technologies achievement standard 2.44 (AS91371)

This is an achieved level guide.

Remember that you only need to do one of the three encoding topics (compression, encryption, and error control coding) to the excellence level. If you are either not interested in getting more than achieved, or are doing either compression or error control coding to the excellence level, then this is the right guide for you. If you were wanting to do encryption up to the excellence level, then you should select the RSA cipher assessment guide instead. Note that there is no excellence level assessment guide for caesar cipher - while it is great for the achieved criteria, it is unsuitable for merit/ excellence due to not being used in the real world.

In order to fully cover the standard, you will also need to have done projects covering the topics of compression and error control coding to at least the achieved level (with one of them to the excellence level if you are attempting to get more than achieved), and projects covering the topics of representing data using bits and human computer interaction, and include these in your report.

## Overview

Encoding has the following *achieved* bullet points in achievement standard 2.44 which this guide covers.

**Achieved:** “describing the concept of encoding information using compression coding, error control coding, and encryption; and typical uses of encoded information”

As with all externally assessed reports, you should base your explanations around personalised examples.

# Reading from the Computer Science Field Guide

You should read and work through the interactives in the following sections of the CS Field Guide in order to prepare yourself for the assessed project.

Read all of these sections, as they give the necessary introduction of the topics

[What's the Big Picture?](#)

[Substitution Ciphers](#)

If you are really keen, you might like to read further into the problems with substitution ciphers, although note that this is optional because it is not necessary for the project in this guide.

[Problems with Substitution Ciphers](#)

## Project

Start this section by writing an introduction to the topic of encryption. *Briefly* explain what encryption is, what it is used for, and what kinds of problems would exist if there was no such thing as encryption. This introduction only needs to be a few sentences - you are just showing the marker that you understand the bigger picture of what encryption is, and some of the typical uses of it.

Now you are going to make an example of encryption being used to encrypt a sentence. For this example, you should use Caesar Cipher. Despite it not having been used in practice for a very long time, it is ideal for illustrating the basic ideas of encryption. Your example should consist of the following components, all of which should be included in your report.

1. Start by writing a short sentence (a few words will be enough, and keep it appropriate for your report). You will be encrypting this sentence.
2. Choose a number between 1 and 25 that will be your encryption key.
3. Make a conversion table that shows how each letter in your original sentence should be changed using your encryption key (format the conversion table to 3 columns in your report to make it compact and easy to read)
4. Encrypt your original sentence using the conversion table you have generated.

Briefly describe what you have done throughout the various parts of your example, and be sure your descriptions include the terms "plain text", "key", and "cipher text" in the relevant places.

Briefly describe what information a classmate would need to decrypt your message (assuming they decrypt it the "proper" way, as opposed to trying to break the cipher).

## Hints for success

- Put encryption in its own section (your report should have suitable headings and subheadings for each topic to make it clear for the marker) and ensure that you briefly introduce the topic. It is important that your report clearly demonstrates that you know the difference between encryption, error control coding, and compression, and what their different purposes are.
- Display the conversion table in a way that doesn't waste space. Format it to 3 columns, if not 4.
- Overall, this project should take up to 1 page, possibly 1 ½ pages of your report.

# Achieved only Guide for Human Computer Interaction (2.44)

This is a guide for students attempting Human Computer Interaction in digital technologies achievement standard 2.44 (AS91371), who are aiming for **Achieved**.

In order to fully cover the standard, you will also need to have done projects covering the topics of Representing Data and Encoding, and included these in your report.

## Overview

Human Computer Interaction has the following achieved bullet points in achievement standard 2.44, which this guide covers.

**Achieved:** "providing examples from human-computer interfaces that illustrate usability heuristics"

As with all externally assessed reports, you should base your explanations around personalised examples, so that the marker can be confident that your report is your own work.

## Reading from the Computer Science Field Guide

Start by reading these introduction sections. They will give you a general overview of what Human Computer Interaction is all about. If you read them last year for 1.44 and remember the material well, you can just quickly skim over them.

- [What's the Big Picture?](#)

- Usability Heuristics

# Project

## Achieved

Everybody has experienced times where a computer system does not do what they expect and/ or they cannot get it to do what they want. A natural reaction to this for many people is to blame themselves. However, in almost every case, the real problem is with the design of the interface. In the field guide or in class, you will have learnt a lot about Nielson's Heuristics. If you think back on some of the times where you've had difficulty with a computer system, you'll probably be able to identify which of Nielson's Heuristics were violated.

In order to demonstrate understanding of Usability Heuristics, you should write a brief (around half to 1 pages) introduction to Nielson's heuristics, using a couple of examples (preferably two good ones, but no more than three) you have come across in your day to day life to illustrate your introduction.

Your introduction should do the following:

- Summarise the purpose of Nielson's heuristics. i.e. why do computer scientists find the heuristics useful to know?
- Include two or three examples of heuristics violations you've come across in your every day life. For each of these examples you should:
  - Explain what and when it happened (one or two sentences should be enough).
  - State which heuristic(s) was/were violated (give the full name of the heuristic - just its number is not enough). Remember that there is not necessarily one right answer. Different people will come to different conclusions.
  - Explain **why** you think it was that/those particular heuristic(s) violated (one sentence should be enough). There isn't any one right answer, just write what you think.

You should **not**:

- List all the heuristics (it's not necessary and wastes space/ bulks out the report unnecessarily)

- List examples you were given in class or read about on the Internet (the marker is only interested in the heuristic violations **you** identify and classify).

## Merit/ Excellence

Because this is an **achieved only** guide, you should follow the merit/ excellence guide for HCI instead if you are aiming for higher than achieved.

## Hints for Success

- Do not include a list of the heuristics. It is not necessary.
- Do not write about the cliché examples you have found on the Internet. Remember that the point of the standard is that *you* can identify the heuristic that was violated, given an identified usability issue.
- Only use images if they are helpful. *Ensure they are legible.*

## Recommended Number of Pages

We recommend not writing more than 1 page for this topic for achieved. 2 pages might be reasonable if you want to include a lot of images, although should not be viewed as a target.

# Excellence Guide for Human Computer Interaction (2.44)

This is a guide for students attempting Human Computer Interaction in digital technologies achievement standard 2.44 (AS91371).

In order to fully cover the standard, you will also need to have done projects covering the topics of Representing Data and Encoding, and included these in your report.

## Overview

Human Computer Interaction has the following bullet points in achievement standard 2.44, which this guide covers.

**Achieved:** "providing examples from human-computer interfaces that illustrate usability heuristics"

**Merit:** "evaluating a given human-computer interface in terms of usability heuristics"

**Excellence:** "suggesting improvements to a given human-computer interface based on an evaluation in terms of usability heuristics"

As with all externally assessed reports, you should base your explanations around personalised examples, so that the marker can be confident that your report is your own work.

## Reading from the Computer Science Field Guide

Start by reading these introduction sections. They will give you a general overview of what Human Computer Interaction is all about. If you read them last year for 1.44 and remember the material well, you can just quickly skim over them.

## What's the Big Picture?

### Users and Tasks

### Interface Usability

Then read the section on usability heuristics. You will need to understand this material well.

### Usability Heuristics

# Project

### Achieved

Everybody has experienced times where a computer system does not do what they expect and/ or they cannot get it to do what they want. A natural reaction to this for many people is to blame themselves. However, in almost every case, the real problem is with the design of the interface. In the field guide or in class, you will have learnt a lot about Nielson's Heuristics. If you think back on some of the times where you've had difficulty with a computer system, you'll probably be able to identify which of Nielson's Heuristics were violated.

To start this section of your report, you should write a brief (around half a page) introduction to Nielson's heuristics, using a couple of examples (preferably two good ones, but no more than three) you have come across in your day to day life to illustrate your introduction.

Your introduction should do the following:

- Summarise the purpose of Nielson's heuristics. i.e. why do computer scientists find the heuristics useful to know?
- Include two or three examples of heuristics violations you've come across in your every day life. For each of these examples you should
  - Explain what and when it happened (one or two sentences should be enough).
  - State which heuristic(s) was/were violated (give the full name of the heuristic - just its number is not enough). Remember that there is not necessarily one right answer. Different people will come to different conclusions.
  - Explain **why** you think it was that/those particular heuristic(s) violated (one sentence should be enough). There isn't any one right answer, just write what you think.

You should **not**:

- List all the heuristics (it's not necessary and wastes space/ bulks out the report unnecessarily)
- List examples you were given in class or read about on the Internet (the marker is only interested in the heuristic violations **you** identify and classify).

### **Merit/ Excellence**

Next, you are going to carry out a usability evaluation using heuristics. If you did 1.44, this might seem a lot like what you did then. However, you should keep in mind the following points:

- You **must** use heuristics. While in 1.44 the heuristics were optional, in 2.44 they are a requirement.
- You only need to evaluate **one** interface.
- Excellence is not comparing multiple interfaces, but instead using your own intuition to suggest improvements which would address the usability problems you identify.

## Choosing an interface to use

Start by choosing a suitable interface. A suitable interface would be one that you have not used before. Write a list of common tasks with that device, for example sending a text or adding a new contact on a cellphone, or drawing a picture and adding text in a drawing program. You might not need to use all the tasks you come up with - it will depend on how many usability issues you are finding.

## Carrying out the usability evaluation

Go through the tasks on your list. Go slowly, and whenever you run into an issue, write down what it was.

Once you have three or four issues written down and know which heuristic was violated (make sure there are at least two different heuristics across all the issues you identify), you can stop.

## Writing about the usability evaluation in your report

Put a heading that says something like "Usability evaluation" to make it really clear to the marker that you are now doing your usability evaluation. Start the section by writing a paragraph stating what device you choose and which tasks you carried out. Also briefly explain how you carried out your evaluation.

Then, for two or three of the issues you identified, write a paragraph that includes the following:

- Explain what and when it happened (one or two sentences should be enough).
- State which heuristic(s) was/were violated (give the full name of the heuristic - just its number is not enough). Remember that there is not necessarily one right answer. Different people will come to different conclusions.
- Explain **why** you think it was that/those particular heuristic(s) violated (one sentence should be enough). There isn't any one right answer, just write what you think.
- Suggest how the interface designer could address the issue you identified. Be sure to justify your answer.

The length of the paragraphs will depend largely on how concisely you can write. For this reason, keep in mind that more is not necessarily better.

You might like to include images of usability issues (e.g. strange layouts of buttons, strange icons, etc), but be sure these don't take up more than 1 page in total, are clear, and contribute directly to your explanations. Images are not required, so don't include them if you don't need them.

Finish up with a conclusion of two or three sentences. Was the interface a good design overall? Why did you reach this conclusion? While the conclusion is supposed to be your opinion, keep your focus on the *usability* of the interface. Stay away from external factors, such as your view towards the company as a whole. For example, saying "It sucks because it is Internet Explorer" is not what the marker is looking for. Instead, say "Internet Explorer was difficult to use because".

## Hints for Success

- Write as concisely as you can. The quality of your examples and whether or not you justified your conclusions about which heuristic was violated are more important than the amount of text. It is possible to cover all the requirements to the excellence level with less than one page, and it has been done before.
- Do not include a list of the heuristics. It is not necessary.
- Do not write about the cliché examples you have found on the Internet. Remember that the point of the standard is that *you* can identify the heuristic that was violated, given an identified usability issue.

- Only use images if they are helpful. Ensure they are legible.

# Recommended Number of Pages

We recommend that this project does not take up more than 2 pages of text, and up to 1 page of images (optional, only include images if they are helpful to your explanations). Students only attempting achieved and not doing an evaluation will need less than one page.

Keep in mind that these are maximums. Concise writing will make it easily possible to cover the requirements in less space.

- Up to  $\frac{1}{2}$  to 1 page introducing the topic and giving your general examples of heuristics (**Achieved**)
  - If you are attempting merit/ excellence, it should be nearer to  $\frac{1}{2}$  because your evaluation will give additional heuristic examples. (**Merit/ Excellence**)
- Up to  $\frac{1}{2}$  a page introducing the interface and tasks you chose, and how you went about your evaluation. (**Merit/ Excellence**)
- Up to 1 page explaining the issues you found when evaluating the interface and then concluding your findings. (**Merit/ Excellence**)
- Up to 1 page of various images to go along with your explanations (optional)

# Excellence Guide for Data Representation (2.44) - Text

This is an excellence level guide for students attempting data representation in digital technologies achievement standard 2.44 (AS91371). It is aimed at students aiming for merit or excellence.

**You will need to do one more project in data representation in addition to this one,** because the standard requires you to cover two types of data. This guide covers only text. Note that there is no excellence requirement for data representation -- it only goes up to merit. This guide is called an excellence guide though to avoid confusion about whether or not it is suitable for students aiming for excellence.

In order to fully cover the standard, as well as one more investigation on data representation for another type of data, you will need to do projects covering the three encoding topics up to the achieved level (error control coding, encryption, and compression), follow one of the three coding methods to excellence level, and a project covering human computer interaction, and include these in your report.

## Overview

The topic of Data Representation has the following bullet points in achievement standard 2.44, which this guide covers.

**Achieved:** "describing ways in which different types of data can be represented using bits"

**Merit:** "comparing and contrasting different ways in which different types of data can be represented using bits and discussing the implications"

This guide focusses on one of the types of data you will need to cover (you will need to cover two).

As with all externally assessed reports, you should base your explanations around personalised examples.

# Reading from the Computer Science Field Guide

You should read and work through the interactives in the following sections of the CS Field Guide in order to prepare yourself for the assessed project. Note that while you do not need to write about ASCII in this project; it is assumed you have a solid understanding of ASCII.

- [Representing whole numbers with Binary](#) - Most other representations are based on binary number representations, so understanding these first is essential. You only need to understand the material in this subsection.
- [Representing Text](#) - You should read this entire section carefully. It is central to the project.

## Project

### Writing an Introduction to Data Representation

***You only need to do this if you have not already done it (the other guide you follow for data representation will also tell you to write this intro)***

This section will be around  $\frac{1}{4}$  to  $\frac{1}{2}$  of a page.

Start the data representation section of your report by writing an introduction to the topic of data representation. Describe what a "bit" is, and why computers use bits to represent data. Briefly describe a couple of the ways in which computers physically represent bits (if you open a computer, you will **not** see lots of little 0's and 1's!).

This explanation must be in your own words, based on what you understood in class (e.g. do not paraphrase a definition).

### Representing text with various Unicode systems

***This is the main part of your project.***

This section will be around  $1\frac{1}{2}$  pages in your report. There are 5 parts to it.

1. Introducing the idea of Unicode and UTF (up to  $\frac{1}{2}$  of a page).
2. Showing your text samples.
3. Showing how one character from each text sample can be represented with UTF-32
4. Showing how one character from each text sample can be represented with UTF-8

5. Comparing the representation size of each text sample using UTF-8, UTF-16 and UTF-32, and explaining your findings.

### Choosing the text samples

Choose two text samples, which you will use to explain Unicode and the representations for it. One text sample should be in English, and the other should be in an Asian language, for example Japanese. The text samples should be no longer than 50 characters long. You should be able to find a suitable text sample online, for example by visiting a Japanese forum. Check it in [Google Translate](#) to ensure it is appropriate for your report.

## Introduction to Unicode and UTF

Start by writing an introductory paragraph to Unicode and the representations used to represent it. Your paragraph should:

- Explain what Unicode is.
- Explain how Unicode is a character set (and not a representation). Use one character from each of your text samples as examples in your explanation.
- Explain how UTF-8, UTF-16, and UTF-32 are various ways of representing the Unicode characters. The paragraph should be no more  $\frac{1}{2}$  of a page long.

If you are confused, reread the Unicode section in the CS Field Guide or ask your teacher for help.

### Include your text samples

Show your text samples. Explain that you are going to be using them to explain the concepts in your report.

These should not take up more than  $\frac{1}{3}$  of a page in your report, for both of them. If they take more space, than use smaller samples.

### Showing examples of UTF-32 representation

Show how the first character in each of your samples is represented using UTF-32. Briefly explain what UTF-32 does.

This should not take up more than  $\frac{1}{3}$  to  $\frac{1}{2}$  of a page in your report.

### Showing examples of UTF-8 representation

Show how the first character in each of your samples is represented using UTF-8. Explain the method that was used to convert the character's unicode number into a UTF-8

representation. Your process for each character should be the following (which is in the CS Field Guide).

1. Look up the character in the Unicode table to get its Unicode number
2. Convert the Unicode number into binary
3. Look at the UTF-8 conversion table to find the correct pattern to use
4. Fill in the blanks in the pattern with the bits in the character's binary representation

This should not take up more than  $\frac{1}{2}$  of a page in your report.

## Comparing the various UTF representations

The field guide includes an interactive [coming soon] which is used to determine how many bits are needed to represent a piece of text, using UTF-8, UTF-16, or UTF-32. Use that interactive to determine the best UTF representation for each of your text samples.

Include a table in your report that shows the size for each text sample, with each representation.

Explain your findings. Which seems to be the best for English text? Which seems to be the best for Asian text? Why is this the case?

(optional) If you are keen, you might like to read more about [UTF-16](#) and [UTF-32](#) to try and figure out why you got the results that you did.

This (including the table) should not take more than  $\frac{2}{3}$  of a page in your report.

## Hints for success

- Your 2.44 report should be structured in the following way. Note that the bolded parts are recommended headings. Details have not been included for sections covered in other guides. Look carefully at the Representing Data section to ensure you have structured your report properly. This will help the marker find what they need to find.

### Computer Science report for 2.44

- **Representing Data**

Put your introduction to what bits are here

- **Representing Text**

Put all your project work related to Unicode here

- **Representing [other topic]**

(see relevant guide for information on structuring this section)

- **Encoding**

(see relevant guide for information on structuring this section)

- **Human Computer Interaction**

(see relevant guide for information on structuring this section)

- Be careful to put your explanations and examples under the correct headings.
- Do not include large lists or tables, for example do not include an ASCII or Unicode table or a list of binary number conversions.
- The introduction to data representation should not take more than ½ of a page in your report.
- The Unicode project in this guide should not take up more than 1½ to 2 pages in your report.

# Excellence Guide for Data Representation (2.44) - Numbers

This is a merit/excellence level guide for students attempting data representation in digital technologies achievement standard 2.44 (AS91371). It is aimed at students aiming for merit or excellence.

**You will need to do one more project in data representation in addition to this one,** because the standard requires you to cover two types of data. This guide covers only numbers (positive and negative numbers). Note that there is no excellence requirement for data representation – it only goes up to merit. This guide is called an excellence guide though to avoid confusion about whether or not it is suitable for students aiming for excellence.

In order to fully cover the standard, as well as one more investigation on data representation for another type of data, you will need to do projects covering the three encoding topics up to the achieved level (error control coding, encryption, and compression), follow one of the three coding methods to excellence level, and a project covering human computer interaction, and include these in your report.

## Overview

The topic of Data Representation has the following bullet points in achievement standard 2.44, which this guide covers.

**Achieved:** "describing ways in which different types of data can be represented using bits"

**Merit:** "comparing and contrasting different ways in which different types of data can be represented using bits and discussing the implications"

This guide focuses on one of the types of data you will need to cover (you will need to cover two).

As with all externally assessed reports, you should base your explanations around personalised examples.

# Reading from the Computer Science Field Guide

You should read and work through the interactives in the following sections of the CS Field Guide in order to prepare yourself for the assessed project.

- [Representing whole numbers with Binary](#) - It is important that you understand this section really well before going any further, as every other concept is based on it.
- [Representing numbers in practice](#) - This section is useful background, which explains the issues we need to consider in practice when representing numbers in binary.
- [How many bits are used in practice](#)
- [Representing negative numbers in practice](#) - It is important that you understand this section really well, because it is central to this project. You should understand what we mean by a simple sign bit, Two's Complement, and how to add and subtract binary numbers, and the implications for simple sign bits vs Two's Complement.

## Project

### Writing an Introduction to Data Representation

***You only need to do this if you have not already done it (the other guide you follow for data representation will also tell you to write this intro)***

This section will be around  $\frac{1}{4}$  to  $\frac{1}{2}$  of a page.

Start the data representation section of your report by writing an introduction to the topic of data representation. Describe what a "bit" is, and why computers use bits to represent data. Briefly describe a couple of the ways in which computers physically represent bits (if you open a computer, you will **not** see lots of little 0's and 1's!).

This explanation must be in your own words, based on what you understood in class (e.g. do not paraphrase a definition).

### Representing Positive and Negative Numbers in Binary

***This is the main part of your project.***

This section will be around 2 pages in your report. There are 4 parts to it.

1. Showing how **positive** numbers can be represented in binary, and clearly explaining the process.
2. Showing how **negative** numbers can be represented in binary with a simple sign bit.
3. Showing how **negative** numbers can be represented in binary using Two's Complement, and clearly explaining the process.
4. Explaining reasons why Two's Complement is widely used, and simple sign bits are not, by showing examples of addition and subtraction. Also explain why the way a simple sign bit represents 0 is problematic.

### **Choosing three numbers for your examples**

You will need to pick three numbers, which you will use to illustrate the various ways of representing numbers in binary. Your chosen numbers should **not** be in field guide examples.

1. A **positive** number **between 65 and 120**
2. A **negative** number **between -10 and -64**
3. A **additional positive** number, which is the "positive" form of your negative number (i.e. if you had chosen -11, this number would be 11). This number is **only used in the last part of the project**.

The numbers **should be within the ranges**, to ensure they work with all the representations you'll be using them for.

### **Representing positive numbers in binary**

Show how your **positive** number is represented in binary. Explain clearly how you figured it out.

This section should not take up more than  $\frac{1}{3}$  of a page.

### **Representing numbers in binary with a simple sign bit**

Show how your **positive** number is represented in binary with a simple sign bit.

Show how your **negative** number is represented in binary with a simple sign bit.

This section should not take up more than  $\frac{1}{4}$  of a page. Keep it very brief, and you should be able to explain it in no more than 1 - 2 sentences.

## Representing numbers in binary with Two's Complement

Show how your **negative** number is represented in binary with Two's Complement. Explain clearly how you arrived at that answer. This could involve clearly listing the steps, and showing the result after each one.

**Common pitfall warning:** Don't forget to remove the sign bit before calculating the Two's Complement representation!

This section should not take up more than  $\frac{1}{3}$  of a page.

### Explaining reasons why Two's Complement is widely used, and simple sign bits are not

**Briefly** explain why the simple sign bit representation for **0** is problematic.

The field guide showed a few examples of adding and subtracting binary numbers using a simple sign bit, and then Two's Complement. These examples illustrated that Two's Complement is far easier to work with than a simple sign bit. You'll now be doing your own calculations to illustrate this point.

Start by ensuring you have all the following binary representations ready to use (some of them you will not have done yet so will need to do now, but do not show any more conversions to binary in your report).

*All the numbers should be 8 bits long. Add leading 0's if needed, but put them **after** the simple sign bit if there is one.*

Number 1: **positive number** in binary with a **simple sign bit**.

Number 2: **additional positive number** in binary with a **simple sign bit**.

Number 3: **positive number** in binary, using Two's Complement (remember that this is just a plain binary number).

Number 4: **negative number** in binary, using Two's Complement.

Number 5: **additional positive number** in binary, using Two's Complement (remember that this is just a plain binary number).

Now, show and explain the following calculations. Indicate whether or not they lead to a correct answer. You might find it easier to do them on paper, and then scan and include them as images in your report. Read the hints for success for advice on this.

**Calculation 1:** Number 3 + Number 5

Use this first calculation to show how two positive binary numbers can be added.

### Calculation 2: Number 1 – Number 2

This calculation shows whether or not subtraction can be done with two positive numbers using simple sign bits, without making a special case for the sign.

### Calculation 3: Number 3 + Number 4

This calculation shows that subtraction can be done by **adding** a negative number to the positive number. (Remember that with decimal numbers, adding the negative of a number is the same as subtracting the number. It's the same for binary.)

### Calculation 4: Number 5 + Number 4

This calculation shows what happens when a bigger number is subtracted from a smaller number.

Explain what you have found.

**Hint:** You should find that calculation 1 works (it's just adding positive numbers, so no surprise), calculation 2 fails (the sign bits mess up the calculation), and calculations 3 and 4 work (thanks to Two's Complement).

This section should not take more than 1 page in your report.

## Hints for success

- Your 2.44 report should be structured in the following way. Note that the bolded parts are recommended headings. Details have not been included for sections covered in other guides. Look carefully at the Representing Data section to ensure you have structured your report properly. This will help the marker find what they need to find.

### Computer Science Report for 2.44

- **Representing Data**

Put your introduction to what bits are here

- **Representing Numbers**

Put all your project work related to Binary numbers and Two's Complement here

- **Representing [other topic]**

(see relevant guide for information on structuring this section)

- **Encoding**

(see relevant guide for information on structuring this section)

- **Human Computer Interaction**

(see relevant guide for information on structuring this section)

- Be careful to put your explanations and examples under the correct headings.
- Do not include large lists or tables, for example do not include an ASCII table or a list of binary number conversions.
- The introduction to data representation should not take more than ½ of a page in your report.
- If you choose to include some information as scanned writing/drawings, do **not** save as JPEG! Use PNG instead. JPEG distorts writing really badly! Be sure to use a dark pen, and if you use a camera rather than scanner to do the scanning, make sure the image is focussed properly.
- The project on positive and negative numbers should not take up more than 1½ to 2 pages in your report.

# Excellence Guide for Data Representation (2.44) - Colours

This is an excellence level guide for students attempting data representation in digital technologies achievement standard 2.44 (AS91371). It is aimed at students going for merit or excellence.

You will need to do one more project in data representation in addition to this one, because the standard requires you to cover two. This guide only covers text. Note that there is no excellence requirements for data representation -- it only goes up to merit. This guide is called an excellence guide though to avoid confusion about whether or not it is suitable for students aiming for excellence.

In order to fully cover the standard, you will also need to have done one more project on data representation, projects covering the three encoding topics up to the achieved level (error control coding, encryption, and compression), and a project covering human computer interaction, and included these in your report.

## Overview

The topic of Data Representation has the following bullet points in achievement standard 2.44, which this guide covers.

- **Achieved:** "describing ways in which different types of data can be represented using bits"
- **Merit:** "comparing and contrasting different ways in which different types of data can be represented using bits and discussing the implications"

This guide focusses on one of the types of data you will need to cover (you will need to cover 2).

As with all externally assessed reports, you should base your explanations around personalised examples.

# Reading from the Computer Science Field Guide

- [Representing whole numbers with Binary](#) - It is important that you understand this section really well before going any further, as every other concept is based on it.
- [Images and Colours](#) - You will need to read this entire section well, as the project is based on it.

## Project

### Writing an Introduction to Data Representation

**You only need to do this if you have not already done it (the other guide you follow for data representation will also tell you to write this intro)**

This section will be around  $\frac{1}{4}$  to  $\frac{1}{2}$  of a page.

Start the data representation section of your report by writing an introduction to the topic of data representation. Describe what a "bit" is, and why computers use bits to represent data. Briefly describe a couple of the ways in which computers physically represent bits (if you open a computer, you will **not** see lots of little 0's and 1's!).

This explanation must be in your own words, based on what you understood in class (e.g. do not paraphrase a definition)

***This is the main part of your project.***

This section will be around 1 to  $1\frac{1}{2}$  pages in your report. There are 4 parts to it.

1. Choosing a colour that you like, explaining how it is represented with red, green, and blue components, explaining why computers represent colours in this way.
2. Converting your chosen colour to its 24 bit binary representation, explaining the process clearly.
3. Showing how your colour would be represented with 8 bits, and explaining whether or not the colour is still the same as it was with 24 bits, and why.
4. Discussing why we commonly use 24 bit colour as opposed to 8 bit colour or 16 bit colour, but not a higher number of bits, such as 32 bits or more.

Each of the subsections will look at one of these parts.

## Explaining how a colour is represented with red, green, and blue components

**Choose a colour**, using the RGB colour mixer which is near the start of the section on colours (do not use the CMY one). Include a sample of your chosen colour in your report (it just needs to be a small rectangle, no more than about 50 pixels tall), and specify how much red, green, and blue your chosen colour has.

Briefly explain (1 - 2 sentences) why colours are described as amounts of red, green, and blue.

This section should be no longer than  $\frac{1}{3}$  of a page in your report.

## Converting a colour to a 24 bit representation

Start by briefly explaining (1 - 2 sentences) what is meant by colour depth, and "8 bit colour", "16 bit colour", and "24 bit colour"

Show the 24 bit representation for your chosen colour, clearly explaining the process you used to get to it.

This section should be no longer than  $\frac{1}{3}$  of a page in your report.

## Converting a colour to a 8 bit Representation

Convert your chosen colour to an 8 bit representation. The best way of doing this is to take your 24 bit colour representation, and remove all the bits except for the leftmost 3 red bits, leftmost 3 green bits, and leftmost 2 blue bits. Put your colour into the 8 bit colour interactive to see what it looks like.

Explain whether or not your colour looks different with just 8 bits. How many different colours can be represented with 8 bits? And what about 24 bits? Why is it impossible to represent every colour that can be represented with 24 bits, with just 8 bits?

This section should be no longer than  $\frac{1}{3}$  of a page in your report.

## Discussing the use of 24 bit colour and alternatives

24 bit colour is the most widely used representation of colour, but it is not the only one. Sometimes we can use 8 bit colour or especially 16 bit colour, and sometimes we want more than 24 bit colour, even though the human eye can't actually distinguish between more than 24 bits worth of colours. In this section, you should write about why we mostly use 24 bit colour, and the situations where we might use more or less bits for colours.

The field guide contains a few ideas, and you may find the interactives useful for experimenting, although you should do some of your own reading as well. [Wikipedia is a good place to start](#), although remember not to paraphrase information. Put it into your own words, and relate it back to your own understand of representing colours.

**This section should be no longer than ½ to 1 page in your report.** There should be no more than **½ a page of text** -- only go up to 1 page if you have included diagrams or images. Any images you include must be entirely your own work, and if you use the field guide interactives which show how images appear with a different number of bits, then you must **use your own image**, setting it with the "Choose file" option in the interactive. Your explanation must explain what your image is demonstrating.

Be sure that your final report is printed in colour, if you include images.

## Hints for success

- Your 2.44 report should be structured in the following way. Note that the bolded parts are recommended headings. Details have not been included for sections covered in other guides. Look carefully at the Representing Data section to ensure you have structured your report properly. This will help the marker find what they need to find.

### Computer Science report for 2.44

- **Representing Data**

Put your introduction to what bits are here

- **Representing Colours**

Put all your project work related to representing colours here

- **Representing [other topic]**

(see relevant guide for information on structuring this section)

- **Encoding**

(see relevant guide for information on structuring this section)

- **Human Computer Interaction**

(see relevant guide for information on structuring this section)

- Be careful to put your explanations and examples under the correct headings.
- Do not include large lists or tables, for example do not include an ASCII or Unicode table or a list of binary number conversions.
- The introduction to data representation should not take more than  $\frac{1}{2}$  of a page in your report.
- The Colours project in this guide should not take up more than  $1\frac{1}{2}$  to 2 pages in your report.
- It is recommended to print your final report for marking in colour, so that all the colours you have used are visible to the marker :)

# Software Engineering Methodologies - Agile Processes vs Waterfall Processes

This is a guide for students attempting Software Engineering in digital technologies achievement standard 3.44. This guide is not official, although we intend for it to be helpful, and welcome any feedback.

In order to fully cover the standard, you will also need to have done a project in one other 3.44 topic. The other project should be in either Complexity and Tractability, Artificial Intelligence, Formal Languages, Network Protocols, or Graphics and Visual Computing.

## **Additional Information:** Required skills and preparation for this project

For this project, students will need to interview somebody from the software engineering industry about what software methodologies (Agile and Waterfall) their company uses. It would be best to pick a company that uses Agile (which is now very common). They will also need a couple of examples of projects they have done themselves: one where they used an Agile approach, and one where they used a Waterfall approach. The projects could either be ones they do this year, or ones they have done in the past.

The teacher should arrange for the class to visit a local software company where there is a software engineer they can interview, or for a software engineer to visit the class. The students should prepare “interview” questions before the visit, to ask the software engineer in order to get the information they need to complete their report.

The student's Waterfall project to use as an example need not be in computing; in other technology classes, such as hard materials, students might have used a waterfall process where they had to do analysis/requirements gathering, then design, and then implementation (building) in that order, without being able to easily go back

to the previous step. Some students naturally carry this process over to digital technologies, and they may have even used it in 1.45/1.46 and 2.45/2.46.

To get an example of an Agile project the student did: some of the level 3 digital technology standards, such as 3.46 or 3.43, give students an excellent opportunity to put agile processes into practice. If they were to approach these standards using a Waterfall approach (by trying to design their entire project and then implementing their entire project, and then testing their entire project), they would be risking wasting a lot of time unnecessarily or completely failing at them when something takes longer than expected or doesn't go as planned. Instead, it is far better to work in small increments, designing and then implementing small pieces of functionality so that they always have something that works that they can show to their teacher. It is possible they are already using Agile processes in this way without realising it.

By including both a real world project a software engineer told them about and some of their own projects, students will be able to both consider the real world issues, and have a deeper personalised understanding of the issues than if they were only told about them by the software engineer. This approach to the standard seems to have worked well for students in the past.

Students who chose this project for 3.44 will need to have the following skills (keep this in mind when picking projects, because different projects require different main skills):

- Good written communication: While all of 3.44 requires students to write well, it is most important in this topic because all the ideas are presented in writing, and their report for software engineering will be around 3 to 4 pages of solid text (others, particularly tractability and formal languages, have more room for expressing information in diagrams).
- Good verbal communication: They will need to be able to get the information they need out of the software engineer they are interviewing.
- Good listening skills: They will need to be able to understand the responses to their interview questions, and explore them.
- Good note taking skills: As they will need to be able to remember the details in the information they got.

These skills are also important for software engineers, for very similar reasons. It isn't just about programming!

In New Zealand, a convenient way to find software engineers to come to a class for an interview is through the ICT-Connect programme (<http://www.ictconnect.org.nz/>) or

FutureInTech (their "ambassador" program: <http://schools.futureintech.org.nz/forschools.cfm>).

# Overview

Note that a plural means “at least two” in NZQA documents. Because this is one of the two areas of computer science that the student must cover, most of the plurals effectively become singular for this project, which is half their overall report. This project will give them at least one algorithm/technique and practical application, and the other project they do will give them another of each.

- **Achieved: [A1]** “describing key problems that are addressed in selected areas of computer science”
- **Achieved: [A2]** “describing examples of practical applications of selected areas to demonstrate the use of key algorithms and/or techniques from these areas”
- **Merit: [M1]** “explaining how key algorithms or techniques are applied in selected areas”
- **Merit: [M2]** “explaining examples of practical applications of selected areas to demonstrate the use of key algorithms and/or techniques from these areas”
- **Excellence: [E1]** “discussing examples of practical applications of selected areas to demonstrate the use of key algorithms and/or techniques from these areas”
- **Excellence: [E2]** “evaluating the effectiveness of algorithms, techniques, or applications from selected areas”

The terminology in the 3.44 standard can be challenging to understand because it applies to six different areas. The following list describes how the terminology of the standard maps onto this project.

- *Selected Area:* Software Engineering
- *Key Problem:* Minimising the risk of failure when designing large and innovative software systems.
- *Algorithms/ Techniques:* Waterfall Process, Agile Processes
- *Practical Application:* Software Development
- *Practical Application Examples:* Experiences of a software engineer working in industry (this information is obtained by interviewing them), your own experiences using the processes.

In summary, to satisfy the standard you might do the following:

- Describe the key problems, giving some background information about why they are so important. [A1]

- Explain how Waterfall Processes and Agile processes are applied in the company/ your own work. [M1]
- Describe/ Explain/ Discuss what the company does, and how this was impacted by the agile processes. [A2, M2, E1] (and) Describe/ Explain/ Discuss how the work you have done in your own projects was impacted by Agile/Waterfall processes. [A2, M2, E1]
- Evaluate the use of Agile processes and Waterfall processes. e.g. projects done in the past that worked well with the Waterfall, what would happen if the company you interviewed used Waterfall (ask them this in the interview!), what might have happened if you had used Waterfall in some of your own projects that you used Agile in. [E2]

It is important to understand what not to include as well. Remember that you only want to write 4 pages (your total report has to be under 10 pages), and you want the marker to easily find the material that is relevant to the standard. If what you are writing is not fairly closely related to the software processes (Agile and Waterfall), it is probably not relevant to the standard.

## Reading from the Computer Science Field Guide

Ideally you should read through the entire Software Engineering chapter, as it is all going to help you to improve your understanding of why Software Engineering is so challenging. Don't worry about doing the projects (although this assessment guide is based on one of them).

## Project

This project is based on the “Software Processes” project in the CS Field Guide. Your teacher will arrange for a software engineer to visit your class or for your class to visit a local software company. In this visit, you will interview a software engineer to find out about the software development practices their company uses and how this impacts their work, in particular Agile processes. This guide assumes an agile process is used by the company interviewed, because this is generally used in large and innovative software projects - the projects that are the most challenging to bring to success. Before the interview, you need to come up with a list of interview questions which will help you to get the information that you need to complete this report.

In addition, you should also have your own experience using Agile processes (and Waterfall processes). You might have even used a form of Agile without realising it! Being

able to discuss these experiences and how they were impacted by the user of Waterfall and Agile will be valuable for your report. Some ideas are: Working on a large program or website, you might have designed, implemented, and tested small parts of the functionality at a time, as opposed to trying to design your entire project, then implement it, and then test it.

In other technology projects, you might have used the Waterfall Process. For example, in hard materials technology, you would typically do analysis to get ideas, and then come up with a bunch of designs, and then pick one to actually build. In other digital technology projects, you might also have used the Waterfall Process. In some cases, this would have worked well (if the project was simple for you and you fully understood it). In other cases, you may realise that an Agile Process would have been more effective.

## Writing your report

It makes sense to structure your report based on the 4 different bullet points in the achievement standard (note that 3 of them are step ups of each other, so are just being counted as 1 by this guide). The order in which this guide presents them is probably the best order for your report to present them. You should come up with suitable headings based on these 4 different bullet points to avoid the report being a continuous block of text!

Follow the page limit recommendations, as these will help you to ensure you have enough room in your report for everything. Your goal should be to write the best possible explanations and discussions you can within the page limits. Try to write concisely and to only include the most interesting points (you don't need to include everything you know on the topic). Some points also have paragraph recommendations, to help prevent you writing more than you needed to.

### Introduction (Achieved/ Merit/ Excellence) [A1 + background for the rest of the report]

This should ideally be no more than ½ a page long.

The kinds of things you should include are: What is the key problem you are discussing in your report? *Start the first sentence of your report with “A key problem in software engineering is...”*. Also make sure you include some of the risk factors for projects failing (e.g. new ideas, size, characteristics of the engineers working on it...) This will ensure the marker can easily tick off bullet point A1. (1 paragraph is enough)

What is some motivation for being interested in the key problem? e.g, what interesting software disasters have you read about? What are the consequences of not paying

attention to it? (1 paragraph, or possibly 2 short paragraphs should be enough) What company did you visit (or that visited you), and what kind of software do they make? What methodology do they use? (just 1 short paragraph is enough)

What are your own projects that you will be discussing in your report? Describe each project in one sentence, and state (without defining or justifying) whether it is your example of Agile or Waterfall. If it wasn't purely one or the other, state what it is closest to, and you can discuss why later in your report (i.e. no details are required at this point).

This information is mostly to satisfy the first bullet point, and to convey key information to the marker about your personalised examples that your report uses.

## Explaining how Agile Processes and the Waterfall Process are applied (Achieved/ Merit/ Excellence) [A2, M1]

**Additional Information:** This section mostly covers M1, but will help with A2

Even students that are only aiming for achieved should do this part. It will help count towards A2 as well. Note that forgetting to address this bullet point is a common reason that students who have otherwise done great work did not get excellence. They could potentially include it in the next section, although we recommend doing it explicitly here, unless they are a very good writer. Students who mix this section and the next one should be very careful that their report does satisfy all the criteria, in particular ensuring that they have clear examples of how Agile and Waterfall are put into practice (that is required for M1 and good students frequently overlook it).

We decided to put A2 in the heading despite the limit relevance to help prevent students skipping it.

This should ideally be around ½ to 1 pages long.

The point of this section is to define what the Waterfall process and Agile processes are, and give some specific examples of how Agile methodologies are actually implemented in practice, that is, what specific practices mean that the overall methodology the software engineer you interviewed is using Agile? And what practices did you use in your own project(s) that made them Agile? You don't need to go into too much depth here, you just need to give several examples of practices, and make sure you have explained them, so that it is clear to the marker what they mean. (For example, just saying "X company does development in sprints" is not enough. You need to explain what a sprint actually is). You

should also include 1 or 2 examples from one of your own Waterfall projects (e.g. the order in which you did the phases in it).

You might want to do the following:

- Explain what the Waterfall process is, and the key steps in it.
- Explain what Agile processes are, and the key steps in it, ensuring that your explanation is clear in why they are different to the Waterfall process.
- Give a list of examples of ways in which the Agile and Waterfall methods are put into practice. Each example should be something you did in one of your projects, or from what the software engineer you interviewed told you. State what the practice is, which of your example projects it is from, and explain what it means (especially where jargon is involved). Don't go into depth about the implications, you should do this in the next section. This information for this section could safely be presented as bullet points, as *long as the bullet points are each 2 to 3 full sentences*.

#### **Additional Information:** An example for this section

These points could be in the form of something like "xxxx company has stand up meetings every morning. These go for xxxx minutes, and are so that everybody in the team can catch up with what each other is doing, and xxxx". The key idea is that they are short, but long enough for the student to briefly explain, and they are personalised based on the situation they have heard about.

## **Explaining and discussing examples of projects using Agile Processes or the Waterfall Process, and the implications (Achieved/ Merit/ Excellence) [A2, M2, E1]**

This should ideally be around 1 ½ to 2 ½ pages long.

In this section, you now need to pull everything together, and discuss how your example projects were impacted by whatever methodology was used. Remember that your example projects are the one that the software engineer you interviewed was working on, and a couple of the projects you have done yourself (one Agile and one Waterfall). The way you go about this is very open ended, although some key points you might like to consider when discussing each project are:

- What the project involves, and its characteristics (size, novelty, number of people involved in it, etc).

- Some examples of the Agile/ Waterfall processes in action in the projects (this is more in-depth than the previous section, where you were just trying to define what the Waterfall Process and Agile Processes actually are).
- Whether or not the process being used is effective for the project.
- What the positives of the process being used for the project are.
- What the downsides of the process being used are for the project.
- What you would do differently (with regards to Waterfall and Agile) if you were to do the project again (for the ones that are your projects).

## Evaluating the use of Agile Processes and the Waterfall Process [E2]

This should ideally be around  $\frac{1}{2}$  to 1 page long. Remember to only include your most interesting points, and write concisely. Your goal should be to conclude your report by evaluating whether or not the Agile process addresses the key problem (go back to your first paragraph if you cannot remember what you wrote for the key problem). In addition to using the information you gained from interviewing a software engineer, you might want to do some additional reading. Some points you might consider:

- What would happen if the company you interviewed used Waterfall instead of Agile? Would they still be able to develop their software effectively? Do they consider Agile to be effective? Is there anything about Agile that still isn't ideal for them? (The questions you ask them when you interview them will be essential for this.)
- How well does the Agile process address the key problem you outlined in your introduction?
- What shortcomings does the Agile process have? What kinds of ideas are being considered to address them?
- Are there any modifications the company you interviewed are considering to address issues in their own Agile Process?

## Hints for Success

- It is better to keep the scope of your report small, and go into depth within the small scope. There are many ideas in software engineering that you could potentially include in your report, in addition to software processes (Agile and Waterfall). Some students also talk about the details testing methodologies, implementation methodologies, requirements gathering, etc. Remember that for the standard, you only need to talk about 1 key problem per topic, and only 1 - 2 techniques for it. For this assessment guide, you should only focus on Agile processes vs the Waterfall process.

- Make sure you are focussing on the software methodologies of Agile processes and the Waterfall process. If you have an entire paragraph that has nothing to do with Agile processes or the Waterfall process, then it is probably not relevant to the project you have chosen for the standard.
- Be sure that all your explanations are centered around examples (i.e. the interview with an engineer or your own projects). Remember that you are supposed to be discussing those examples in order to demonstrate techniques, i.e. agile and waterfall.
- Be careful when using bullet points and pros vs cons lists.
  - Generally, single sentence/ statement bullet points don't work well for merit/excellence, because you need to have explained and justified the point you have made. Simply stating facts is only "describing". For example, when evaluating the processes, you may be tempted to have a bunch of bullet points such as "Does not work well for innovative software projects". This statement isn't going to help you at all - it will look like you just copied this fact from the field guide! You need to justify it, and explain why the Waterfall process does not work well for innovative software projects. Once you have done this, you will have several sentences or a paragraph.
  - One exception is for an introduction or conclusion. You might say "In summary, the key points are" and then follow with a list of bullet points that summarise your key points.

## Staying within the page limit

The page limit for achievement standard 3.44 is 10 pages. Remember that you have to do a project on a second, different topic as well, so that leaves 5 pages for Software Engineering. We recommend aiming for 4½ pages so that if some sections go slightly over, you will still be under 10 pages overall. Also, don't forget you will need a bibliography at the end. The page recommendations for specific sections are within the main part of the assessment guide.

# Complexity and Tractability (3.44) - The Traveling Salesman Problem

This is a guide for students attempting Complexity and Tractability in digital technologies achievement standard 3.44. This guide is not official, although we intend for it to be helpful, and welcome any feedback.

In order to fully cover the standard, you will also need to have done a project in one other 3.44 topic. The other project should be in either Software Engineering, Artificial Intelligence, Formal Languages, Network Protocols, or Graphics and Visual Computing.

## Overview

Each project needs to satisfy all bullet points in the standard, which are given below.

Note that a plural means “at least 2” in NZQA documents. Because this is one of the two areas of computer science that the student must cover, most of the plurals effectively become singular for this project, which will make up half of the overall report. This project will give at least one algorithm/technique and practical application, and the other project will give another of each.

- **Achieved: [A1]** “describing key problems that are addressed in selected areas of computer science”
- **Achieved: [A2]** “describing examples of practical applications of selected areas to demonstrate the use of key algorithms and/or techniques from these areas”
- **Merit: [M1]** “explaining how key algorithms or techniques are applied in selected areas”
- **Merit: [M2]** “explaining examples of practical applications of selected areas to demonstrate the use of key algorithms and/or techniques from these areas”
- **Excellence: [E1]** “discussing examples of practical applications of selected areas to demonstrate the use of key algorithms and/or techniques from these areas”
- **Excellence: [E2]** “evaluating the effectiveness of algorithms, techniques, or applications from selected areas”

The terminology in the 3.44 standard can be challenging to understand because it applies to six different areas. The following list describes how the terminology of the standard maps onto this project.

- *Selected Area:* Complexity and Tractability
- *Key Problem:* Finding solutions for problems in which the optimal solution would take an impractical amount of time to find. In this case, the problem is finding the shortest path to visit a number of locations and return to the starting point (the Travelling Salesman Problem).
- *Algorithms/ Techniques:* Brute force algorithms, Greedy algorithms, Heuristic algorithms.
- *Practical Application:* Solving the Cray Pot Problem (or another problem based on the Travelling Salesperson Problem)

#### **Jargon Buster:** What is meant by 'Heuristics'?

Note that the term heuristics means “rules of thumb”, and that this should not be confused with the heuristics covered in Human Computer Interaction.

In HCI, they are also rules of thumb (for making interfaces usable). The heuristics in this context are “rules of thumb” for solving the algorithmic problems encountered in complexity and tractability; a heuristic isn't guaranteed to give the best possible solution, but usually will give a fairly good result.

In summary, to satisfy the standard you might do the following:

- Describe the key problem(s). [A1]
- Show (explain) how a brute force algorithm (that gives the optimal result) and at least one heuristic (e.g. in TSP, one greedy algorithm is Nearest Neighbour, which is a greedy heuristic) can be used to find a solution to an instance of the Cray Pot Problem that you have made. [A2 (partially), M1]
- Explain the results, and what each of the the algorithms has calculated. [A2, M2]
- Discuss some real world examples with specific maps, and how each of the two algorithms would work for them. [E1]
- Evaluate the use of the two algorithms in real world problems. This will involve considering the cost vs benefit tradeoff of finding better solutions, giving examples that show how a heuristic might not find the optimal solution, and possibly looking at additional complications that the basic algorithms cannot solve (e.g. traffic conditions and roadworks). [E2]

# Reading from the Computer Science Field Guide

You should read the text and work through the interactives in the following sections of the CS Field Guide in order to prepare yourself for the assessed project.

- 11.1 - What's the Big Picture?
- 11.2 - Algorithms, problems, and Speed Limits
- 11.3 - Tractability
- 11.4 - The Travelling Salesman Problem

## Project

This project is based around a fictional scenario where there is a cray fisher who has around 18 cray pots that have been laid out in open water. Each day the fisher uses a boat to go between the cray pots and check each one for crayfish.

The cray fisher has started wondering what the shortest route to take to check all the cray pots would be, and has asked you for your help. Because every few weeks the craypots need to be moved around, the fisher would prefer a general way of solving the problem, rather than a solution to a single layout of craypots. Therefore, your investigations must consider more than one possible layout of craypots, and the layouts investigated should have the cray pots placed randomly i.e. not in lines, patterns, or geometric shapes.

### Generating a Craypot Map

In order to generate a random map of craypots that can be used as your own unique personalised example, get a pile of coins (or counters) with however many craypots you need, and scatter them onto an A4 piece of paper. If any land on top of each other, place them beside one another so that they are touching but not overlapping. One by one, remove the coins, making a dot on the paper in the centre of where each coin was. Number each of the dots. Each dot represents one craypot that the cray fisher has to check. You should label a point in the top left corner or the paper as being the boat dock, where the cray fisher stores the boat.

In this project, you will need to make two maps, a “small” map with 7 or 8 craypots and a “large” map, with 15 to 25 craypots.

# Writing your report

## Introduction (Achieved/ Merit/ Excellence)

[A1 + background for the rest of the report]

This should not take more than 1 paragraph, or  $\frac{1}{3}$  of a page. It is essential to do a good job of this part, even if what you are writing seems obvious.

Briefly introduce the Cray Pot Problem. You should be able to explain how the Craypot Problem is equivalent to the Travelling Salesman Problem. Briefly describe how you determined this, and what the equivalent of a town and road is in the Craypot Problem. Explain why computer scientists are so interested in the Travelling Salesman Problem. Include this introduction at the start of the complexity/ tractability section of your report. We strongly recommend the following sentence being at the start or end of the introduction "The key problem I am looking at is [a few words describing the problem]", so that it is really clear to the marker.

## Showing and explaining a brute force approach and then a heuristic approach (Achieved/ Merit/ Excellence)

**Additional Information:** Ensuring you cover the achieved and merit criteria sufficiently

Showing how the brute force algorithm can be applied to the Cray Pot Problem + Explaining why it is not helpful to the crayfisherman + Showing how a greedy heuristic algorithm can be applied to the Cray Pot problem + Explaining what kind of solution the greedy heuristic algorithm has found, and why this is more helpful to the cray fisher (A2/M1/M2).

Note that the difference between achieved and merit will be in the quality of the explanations (i.e whether or not the marker considers them to be "describing" or "explaining"). Generating the personalised examples is necessary for achieved, because the marker needs to be able to see that the student has done their own work.

It is important that a report discusses both algorithms (brute force and greedy heuristic), because doing the brute force algorithm is necessary to show that it is unhelpful in practice, and then students need to explore approaches that can be used in practice, such as the greedy heuristic algorithm. This allows them to give a solid explanation of TSP based problems and how computer scientists deal with them.

Generate a map with 7 or 8 craypots using the random map generation method described above. This is your “small” map. Then make a map with somewhere between 15 and 25 craypots. This is your “large” map. Make a copy of each of your maps, as you will need them again. Read the “hints for success” at the bottom of this guide before making the maps, because it has some advice on making the maps so that they are legible and minimise the usage of precious space in your report.

Using your intuition, find the shortest path between the cray pots in your small map. Do the same with your large map. Don’t spend more than 5 minutes on the large map - you don’t need to include a solution to the large map in your report. It is extremely unlikely you’ll find the optimum for the large map (Recognising the challenges in the problem is far more important than finding a solution). Number the order in which you visit the cray pots on your map.

Use the field guide interactive to estimate how long it would take a computer to solve each of your craypot problems and find an optimal solution.

Explain why using the brute force algorithm to find the optimal route is unhelpful to the cray fisher (remember that they generally have between 20 - 25 cray pots in the water at a time). Save this explanation so that you can include it in your report later. One paragraph is enough.

Unless your locations were laid out in a circle or oval, you probably found it very challenging to find the shortest route. A computer could find it even harder, as you could at least take advantage of your visual search and intuition to make the task easier. A computer could only consider two locations at a time, whereas you can look at more than two. But even for you, the problem would have been challenging! Even if you measured the distance between each location and put lines between them and drew it on the map so that you didn’t have to judge distances between locations in your head, it’d still be very challenging for you to figure out! It is clear the cray fisher isn’t going to want to wait for you to calculate the optimal solution. But can you still provide a solution that is better than visiting the cray pots in a random order?

There are several ways of approaching this. Some are better than others in general, and some are better than others with certain layouts. One of the more obvious approximate algorithms is to start from the boat dock in the top left corner of your map and to go to the nearest craypot. From there, you should go to the nearest cray pot from that cray pot, and repeatedly go to the nearest craypot that hasn’t yet been checked. This approach is known as the “Nearest Neighbour” algorithm, and is an example of a greedy heuristic algorithm, as it always makes the decision that looks the best at the current time, rather than making a not so good choice now to try and get a bigger payoff later. In the excellence part of this

guide, you will explore why it is not always optimal, even though it might initially seem like it is.

For both your small map and large map, use this greedy algorithm to find a solution (it shouldn't take you too long). Number the order in which you visit the cray pots on your map. A computer would be a lot faster than you at this, so you should have a pretty good idea about how the two algorithms compare.

Explain which algorithm is more suitable for determining a route for the crayfisherman, and why. What are the implications for each choice that you considered to arrive at your conclusion? (e.g. how long would he have to wait for it vs how much time and fuel the fisher saves going between cray pots) Save this explanation so that you can include it in your report later. 2 to 3 paragraphs is enough.

You should now have 4 maps (small + brute force algorithm, large + brute force algorithm, small + greedy heuristic algorithm, large + greedy heuristic algorithm), and several explanations. You now need to put your findings into report form. After your introduction, briefly explain how a computer would do a brute force algorithm (hint: check the field guide; the principle of the algorithm is simple, but it is very inefficient!). Next, include your two maps using the brute force algorithm, and briefly explain why the large one was so challenging, followed by your explanation on why this algorithm is no good for the cray fisher. Then explain how the greedy heuristic algorithm you used works, and include your two maps for it. Finally, these include your explanation where you determined which algorithm was best for the Cray Pot problem, and the implications of each choice. All up, you should have around 2 to 3 pages (if you have more, and are planning on attempting excellence, you might want to consider shortening some parts or shrinking down images a little more).

## Discussing real world examples of the traveling salesman problem and evaluating the effectiveness of the algorithms on the real world problems (Excellence)

In order to gain Excellence in the standard, you will need to go beyond just applying and explaining algorithms used on the Cray Pots Problem (and TSP). The reports of students who gain excellence in this standard generally have explored the algorithms and their implications in depth and/ or done their own research into real world applications of TSP. You should ensure you have discussed the practical applications and the algorithms, and have evaluated the algorithms in terms of the practical applications (i.e. are the algorithms any good in practice?) You might choose to do this by following some or all of the following suggestions (Focussing on two is a good number). Remember that you should

only use 5 pages on complexity/tractability in total, so your discussion and evaluation for excellence should take up no more than about 2 pages.

- Analyse the greedy heuristic algorithm (e.g. cases where it finds a really bad solution, and cases where it finds the optimal solution), and explore how effective it would be in practice. One way you could do this is to do a search in Google maps for something like supermarkets in a city (ideally you'll want at least 20 to 30 to appear), and then ensuring the roads are visible on the map, take a screenshot of it. Trace a greedy heuristic algorithm path onto it, and then evaluate how effective it was. You will probably find that some parts of the path make sense, although in other parts it is inefficient because a destination was "skipped" when others that were somewhat near it were visited, and the shortest path heuristic pulled the path away from that destination. What kinds of heuristics would you use to get a better solution? (e.g. could you somehow break the TSP into a bunch of clusters, in which you visit all in the cluster before moving onto the next cluster?) You might want to include a second map showing your other heuristic ideas. The maps should take up around  $\frac{1}{2}$  a page each. Make sure to discuss your conclusions.
- Exploring and discussing why companies that carry out tasks such as delivering goods (e.g. a soft drink company sending people around to restock their many vending machines or a courier service delivering parcels to various addresses) are willing to invest so much money in finding better solutions to their own travelling salesman problems (and the closely related problems that arise when additional constraints such as speed limits and road blocks are added).
- Investigating and discussing real world examples of the Travelling Salesman Problem e.g. a soft drink company sending people around to restock their many vending machines or a courier service delivering parcels to various addresses, or someone dropping off invitations to a group of friends.
- Identifying and discussing some of the additional issues that come up in real world examples, for example, traffic conditions, temporary roadblocks, roads only going between some of the destinations, speed limits, police checkpoints, congestion rules, and traffic lights. Once these are added in, the problem is no longer strictly the Travelling Salesman Problem, although it is still likely to be an intractable problem

(although some of the additional issues can actually make the problem easier to solve; identifying them and exploiting them is another big goal for solutions)

- Evaluating some of the Android and iPhone apps that claim to help the user with TSP style problems. Do they live up to the claims they make?

## Hints for Success

You should be able to write up the project in about 4 to 5 pages.

- This project is one that might be difficult to fit into the page limit because of all the diagrams. You want to be able to make the diagrams as small as possible, while ensuring they are still legible. It would be possible to get the cray pot maps side by side in pairs to save space. Consider making them vertical rather than horizontal (you will need to make this decision before starting, because the numbering must be the right way up). This means that you could place two side by side, taking up about half the page in total. The total space of your four maps would be 1 page. The following tips may also help.
  - Use a fine tipped marker or pen that gives a solid line to draw the dots and lines.
  - Use a ruler to draw the lines.
  - If scanning them makes any lines unclear, just redraw them with image editing software.
  - Use image sharpening or increase the contrast. It is okay if the image is black and white.
  - Don't save them as jpegs (If this seems strange to you, read about [how JPEG compression works](#) when you have some spare time)
- Remember that the marker wants to know about the applications, and the use of algorithms to solve them. This is what the standard asks for, and should always be kept in mind. Real world implications are important, i.e. in practice, the optimal solution is not essential for the result to be useful.
- For excellence, you will need to do additional background reading. Be careful what you include in your report; it is very obvious to the marker when a student has copied text from a source that they don't actually understand. In particular, be sure that you understand the jargon you use so that you can be certain you are using it correctly. Incorrectly used jargon sounds really bad to a reader (e.g. marker) who knows the topic!
- Note that it is essential for Achieved, Merit, and Excellence that you start with your own examples (e.g. craypot maps) and explain those, rather than simply paraphrasing information from the field guide and other sources.

# Staying within the page limit

The page limit for 3.44 is 10 pages. Remember that you have to do a second project on a different topic as well, so that leaves 5 pages for Complexity and Tractability. We recommend aiming for 4½ pages so that if some sections go slightly over, you will still be under 10 pages overall. Also, don't forget you will need a bibliography at the end.

A plausible breakdown would be:

- ½ page: introducing the key problem, the Cray Pot Problem, and why the Cray Pot Problem is equivalent to the Travelling Salesman problem
- 1 page: Craypot maps (Achieved/ Merit Excellence). Note that they are not necessarily on the same page, although their total area should not exceed 1 page.
- 1 page: Explanations about the Craypot maps from the Achieved/Merit part of the guide (Achieved/Merit/ Excellence)
- 2 pages: Whatever you decide to do for excellence. This might include diagrams or discussions. Don't let diagrams take up more than 1 page though, because you need to have an in depth discussion as well.

The project in this assessment guide may be challenging to fit into 5 pages (because of the diagrams/ maps), particularly for if you are aiming for excellence. Some of the other topics fit well into 3 to 4 pages, so you may be able to go slightly over 5 pages for this topic if your other topic requires less space.

# Formal Languages - Checking for valid input in websites

This is a guide for students attempting Formal Languages in digital technologies achievement standard 3.44. This guide is not official, although we intend for it to be helpful, and welcome any feedback.

In order to fully cover the standard, you will also need to have done a project in one other 3.44 topic. The other project should be in either Complexity and Tractability, Intelligence Systems(Artificial Intelligence), Software Engineering, Network Protocols, or Computer Graphics and Computer Vision. Anything outside these topics is not part of the standard.

## Overview

Note that a plural means “at least 2” in NZQA documents”. Note that because this is one of the two areas of computer science that the student must cover, most of the plurals effectively become singular for this project, which is half their overall report. This project will give them at least one algorithm/technique and practical application, and the other project they do will give them another of each.

- **Achieved [A1]:** “describing key problems that are addressed in selected areas of computer science”
- **Achieved [A2]:** “describing examples of practical applications of selected areas to demonstrate the use of key algorithms and/or techniques from these areas”
- **Merit [M1]:** “explaining how key algorithms or techniques are applied in selected areas”
- **Merit [M2]:** “explaining examples of practical applications of selected areas to demonstrate the use of key algorithms and/or techniques from these areas”
- **Excellence [E1]:** “discussing examples of practical applications of selected areas to demonstrate the use of key algorithms and/or techniques from these areas”
- **Excellence [E2]:** “evaluating the effectiveness of algorithms, techniques, or applications from selected areas”

The terminology in the 3.44 standard can be challenging to understand because it applies to six different areas. The following list describes how the terminology of the standard maps onto this project.

- **Selected Area:** Formal Languages
- **Key Problem:** Checking whether or not web form input is “valid”
- **Algorithms/ Techniques:** Regular Expressions, Finite State Automata, RE → FSA conversion algorithm.
- **Practical Application:** Checking various fields in inputs forms

In summary, to satisfy the standard you might do the following:

- Describe the key problems, giving some background information about why they are so important. [A1]
- Describe/ Explain/ Discuss various types of input that need to be checked, and how regular expressions, finite state automata, and other algorithms and techniques from formal languages are used to achieve this.
- Explain (by showing) how regular expressions and finite state automata are used to check inputs (note that this partially overlaps with the previous point).
- Evaluate how effective regular expressions and finite state automata are at checking the validity of various inputs. To do this, you might look at some examples where they are surprisingly ineffective. (Note that this will also help to satisfy E1)

## Reading from the Computer Science Field Guide

Most of the content in the formal languages chapter will help you to complete this project. While you will not be including the examples from the field guide in your report, you should still work through them as they will increase your understanding of regular expressions and finite state automata for when you write your report.

## Project

In this project, you will investigate what inputs can effectively be checked for validity using regular expressions, what inputs it is possible to check with regular expressions (although in some cases it is clearly not the best tool for the job!), and which are computationally impossible to check with regular expressions.

You will also convert a regular expression to its equivalent finite state automata, like what a computer does internally.

This guide is broken into two parts.

- **Achieved/ Merit:** A project which looks at regular expressions and finite state automata for email addresses.
- **Excellence:** Investigating the limits of regular expressions/ finite state automata, the problems which they work well for, the problems which they don't, and the problems they cannot solve. Unlike most of the assessment guides where most of your discussion and evaluation for excellence is generated from additional reading you have done, this one gives several examples for you to think about and analyse, which you will then consider when writing your discussion and evaluation.

**Important:** If you are only attempting Achieved/ Merit, you should **only** do email addresses (or a similar type of input). It is far better to focus on doing one type of input well rather than covering many different ones. Other inputs looked at should be with the intention of discussing and evaluating the effectiveness of regular expressions/ finite state automata.

## Introduction (Achieved/ Merit/ Excellence) [A1 + background for the rest of the report]

This should not take more than 1 paragraph, or around  $\frac{1}{3}$  of a page. It is essential to do a good job of this part, even if what you are writing seems obvious. Start your report for this project by introducing the key problem, and why it is important to be able to check inputs in a web form (e.g. being able to tell the user straight away that the input is invalid, ensuring invalid data doesn't end up in a database and cause problems later). *Start the first sentence of your report with "A key problem in computer science is..."*. Explain why regular expressions are a popular technique used to address the problem.

Additionally, write something like “The types of input I investigated regular expressions for were” followed by a comma separated list. List (without details) the applications that your report looks at (e.g. email addresses, dates, credit card numbers, mathematical expressions, etc). You might need to write this part after you have written the rest of your report. This summary of what you have investigated will assist the marker.

## Checking whether or not an email address is valid (Achieved/ Merit/ Excellence)

**Additional Information:** Bullet points covered in this section

[A2, M1, M2] - Bulk of report for Achieved/ Merit. In total, this will take up around 2 - 3 pages. Students should not go beyond 3 pages for this section if they want to attempt the excellence. A further page breakdown is provided within the section.

A common type of input to check is an email address. Write a regular expression to check for a valid email address. You can make the following assumptions about email addresses, although note that real email addresses are a bit more complicated!

- An email address contains two parts: the “local part” and the “domain part”, separated with a @. They are in the form of “local part”@“domain part”.
- The local part can be made up of any alphanumeric characters (i.e. all the upper and lower cases letters of the alphabet, and the 10 digits), “+” or “.”. There cannot be multiple “.” in a row.
- The domain part can be made up of any alphanumeric characters (i.e. all the upper and lower cases letters of the alphabet, and the 10 digits), and “.”.
- The local part and domain part cannot start or end with a “+”, or “.”.

Once you think you have your regular expression correct, convert it into a finite state automata (you can either do this by hand, or by using one of the tools linked to in the field guide). Make sure the states of your finite state automata are numbered (or have some kind of unique ID) as this will help you with your later explanations. You will be including the regular expression and the finite state automata in your report. The regular expression won’t take up much space, although you should ensure that the finite state automata does not take up more than half a page (and less if it is still legible when shrunk down further).

This should be around one paragraph for the regular expression and one paragraph for the finite state automata. All up, there should be about one page once the regular expression, finite state automata, and two paragraphs of explanation have been put together. Explain how you designed your regular expression (i.e. what does each part of it mean?), and then explain how your finite state automata is equivalent to your regular expression (i.e. which parts of the regular expression map onto what parts of the finite state automata?)

Don’t include this part in your report now, you will include some of them in the next bit (although you must do this testing to ensure you have a correct regular expression/ finite state automata!). Come up with some example email addresses to trace through your regular expression or finite state automata by hand. You should pick examples which are effective in showing that your regular expression accepts valid inputs and rejects invalid inputs. If you find an example that your regular expression does not handle correctly, then you should try and fix it. Your invalid examples should include:

- An email address with multiple dots in a row

- An email address without a @.
- An email address that starts with a @.
- An email address with multiple @'s.
- An email address that ends in a @.
- An email address that starts with a “.”
- An email address that starts with a “+”.

**Additional Information:** Ensuring the work is personalised

It is important that students come up with their own examples to test with. This personalises the student's work. All up, the traces and working should take around 1 to  $\frac{1}{2}$  pages. There are several ways of approaching it. The student can either refer to their regular expression or their finite state automata in the explanation (the latter is probably easier, especially if the states have been numbered), and they may use clearly labelled diagrams as part of their explanation (e.g. a drawn on finite state automata). Any diagrams must be clear and legible. Once you have finished testing and are satisfied that the regular expression/ finite state automata is working correctly, you should pick a couple of valid email addresses and three or four invalid email addresses (e.g. your ones from above) and explain how the finite state automata handles them. Explain how the finite state automata is moved into various states by the email address input. Remember that valid email addresses should end in a terminating state, and invalid email addresses should either end in a non terminating state, or be unable to transition at all at some point. Make sure that each of these worked examples clearly states what email address was used for it (use headings or bold the email address).

## Investigating, discussing, and evaluating the use of regular expressions for checking various inputs (Excellence).

As much as they are loved by computer scientists, regular expressions aren't always the right tool for the job when we need to validate inputs. In many cases, including the email addresses above, they are very helpful. However, there are also plenty of cases where a few lines of code in a standard programming language (e.g. Python) would be a lot better than using a regular expression. There are also cases where you could never come up with a valid regular expression, even if you had infinite time to and space to write one.

For excellence, you will investigate other types of inputs in order to identify what kinds of problems regular expressions are good for, which they could be used for but should not be, and which there is no regular expression for.

In most assessment guides, you are provided with a set of links and questions for excellence. For this one, most online resources you find will be full of strange mathematical symbols, and won't be helpful for you writing your report. So instead, we provide you with several interesting problems which involve a type of input. These can be found at the end of the document in the section "Regular Expression Problems for Excellence". Before you start writing the excellence part of your report, you should explore each of the problems by trying to write a regular expression for them (some parts of them are easier than the email address one), and thinking about why they are either efficient, inefficient, or impossible to solve with a regular expression. Your writing for excellence will be based on your own understanding and thoughts about the problems. Focus on justifying and discussing your thoughts rather than worrying about whether or not you have "correct" answers.

Make a table which summarises the problems you have looked at (there are around 9 of them including the sub problems - a list is included below to help you), and specifies which are possible to solve with regular expressions, which are not, and which should be solved with regular expressions, and which should not. For those which are possible to solve with regular expressions, and are simple enough that they will fit onto one line, include the regular expression in your table. For the ones that are possible but the regular expression is very long, describe in a sentence or two what it would look like, or what kind of thing it would have to do. The table will probably take up around  $\frac{3}{4}$  a page. It provides a summary of the investigations, and gives something to refer to in the discussion/ evaluation.

The problems in the table may include: (Remember to refer to the section at the end for more details!)

- Checking an email address
- Checking that a date entered is (somewhat) sensible
- Checking that a date entered is guaranteed to be valid
- Checking that a credit card number has 16 digits
- Checking what provider a credit card number is with
- Checking if a credit card number is valid (check digit)
- Checking if a simple mathematical equation with numbers and simple operators (+, -, \*, /) is valid.
- Checking if a mathematical equation with parentheses is valid.
- Checking if a string of parentheses is "balanced".

Your written discussion will cover the excellence criteria and should ideally be around 1 to 2 pages long. The key will be to write concisely. You will then write a discussion/evaluation on your findings, which addresses the following key points. The problems in the above table should be referred to.

- Discuss which problems seem to be impossible to solve with regular expressions. Can you notice anything in common about these problems?
- Discuss which problems seem to be possible to solve with regular expressions, but it is not the right tool for the job. Can you notice anything in common about these problems? Describe how you'd solve them in a programming language.
- Discuss which kinds of problems seem to be the ones you'd approach with regular expressions (refer to your table). What are regular expressions good for?

## Hints for Success

- Stay focussed on the key problems this assessment guide addresses. There are many cool applications of formal languages, although to satisfy the standard you need to focus on a narrow scope and go “deep” rather than “wide”.
- Remember that you can use bold and italics, particularly in explanations which involve tracing inputs, in order to make your explanations easier to read.
- Remember that including inappropriate words and material in your assessed report can land you in some serious trouble. In particular, think carefully about the example email addresses you choose to use.
- Don’t trace examples through the problems in the excellence section. You have already shown that you know how regular expressions process inputs when you did the achieved/ merit.

## Staying within the page limit

The page limit for 3.44 is 10 pages. Remember that you have to do a project on a different topic as well, so that leaves 5 pages for Formal Languages. We recommend aiming for 4 and ½ pages so that if some sections go slightly over, you will still be under 10 pages overall. Also, don’t forget you will need a bibliography at the end.

A plausible breakdown would be:

- ½ page: Introducing the key problems, motivations, and the list of applications which are discussed in the report.
- 1 page: Email address regular expression, finite state automata, and explanations of them both.

- 1 to 1 ½ pages: Examples of email addresses and explanations of how they are processed by the regular expression or finite state automata.
- ¾ - 1 page: Table summarising the other various input problems given.
- 1 page: Discussion/ Evaluation based on the findings about the other input problems.

# Regular Expression Problems for Excellence

Remember that these problems are for you to think about, and it is your own thoughts and reasoning that will get you excellence. You should not include traces of examples through your regular expressions (you have already done this with the email addresses).

## Problem 1: Checking for a valid date

Try writing a regular expression to check for a valid date in the form of dd/mm/yyyy (for example, 08/05/2015).

On the surface, this might have seemed straightforward. But have you managed to ensure all these invalid dates are detected?

- 30/02/2015
- 29/02/2014 (but remember that 29/02/2016 is valid!)
- 31/11/1998
- 21/13/2013
- 38/05/1992

Is it computationally possible to write a regular expression that could be correct in every case? Is it at all practical to do? What would the regular expression look like?

## Problem 2: Checking credit card number validity

A credit card number has to have 16 digits, and you should have no trouble writing a regular expression to ensure that it has 16 digits.

You should even be able to write a regular expression for each of the major providers (mastercard, visa, etc, look on wikipedia to discover how you can identify which is which) that will tell you whether or not a given credit card number is from that provider.

But what about the check digits that you learnt about in 2.44? The 16th digit of the credit card number is calculated from the first 15 digits. Why can this not be done effectively using regular expressions? (Think about what the role of regular expressions is) What would a regular expression that only accepted valid credit card numbers (including the

check digit) look like? (Hint: it is at least possible, in theory. But you'd find it impossible to write the entire regular expression by hand in your life-time, as it is just too long...)

## Problem 3: Checking for a valid mathematical expression

Pretend you have a form that you want a user to enter valid mathematical equations into. To be valid, the mathematical equation needs to satisfy the following rules.

- The equation can be made up of all the 10 digits (including multiple digits in a row, and +, -, \*, and /)
- The first and last characters must be digits
- There can be any number of \* + - /, although there cannot be 2 of them in a row.

For example: 123\*32+317-12 is valid.

For example: 123\*/32-23 is not valid.

Try writing a regular expression that can check these inputs. Check it with a few example to ensure it is correct (you should not include the testing in your report, although you should make sure the regular expression that will be going in your report is correct. Testing is important!). This should not be too difficult if you were able to do the email address one without difficulty.

Now, add one extra rule in: Parentheses are allowed, as long as they are “balanced”. i.e. there must be the same number of opening ones as closing ones, and if you scan across it from left to right, the current count of closing parenthesis should never be higher than the current count of opening parenthesis, and at the end, the two counts should be the same. There is no limit to the number of parenthesis allowed.

To save you from going crazy, we'll tell you now that it is impossible. Unlike the other problems so far which were all solvable with regular expressions, even if it was terribly inefficient to write one solution, this one is impossible. The problem is trying to check that the parentheses are balanced. Try and make sense of why it is impossible to use regular expressions to solve this, and what happens when you attempt to do so.

To investigate it, forget about the digits and mathematical operators, and only consider the parentheses, using the “balanced” rules above. For example, it should accept strings like:

- ()
- (( ))
- (( ( )))
- ((((( )))))

- (())(())(((())))(())(())

There is no limit to the length of the string of parentheses, as long as it is balanced.

And it should reject strings like

- (((
- )
- (()) – Too many closing parenthesis
- (())() – At one point there has been more closing than opening
- (())(()) – At one point there has been more closing than opening

Try to make a regular expression or finite state automata that is able to check for strings like these. Don't forget that there is no limit to the length of these parenthesis strings. Even if you make a regular expression/ finite state automata able to recognise some patterns, it is impossible to get them all. What happens when you try?

# Algorithms

## Overview

- EU 4.1 Algorithms are precise sequences of instructions for processes that can be executed by a computer and are implemented using programming languages.
- EU 4.2 Algorithms can solve many, but not all, computational problems.

## Reading from the Computer Science Field Guide

Start by reading through:

- [Algorithms](#)
- [Complexity and Tractability](#)

## Learning objectives

The above chapter readings include specific knowledge for EK's marked in bold. Work to include unmarked learning objectives in the CS Field Guide is currently in progress.

### LO 4.1.1 Develop an algorithm for implementation in a program

- EK 4.1.1A Sequencing, selection, and iteration are building blocks of algorithms.
- EK 4.1.1B Sequencing is the application of each step of an algorithm in the order in which the statements are given.
- EK 4.1.1C Selection uses a Boolean condition to determine which of two parts of an algorithm is used.
- EK 4.1.1D Iteration is the repetition of part of an algorithm until a condition is met or for a specified number of times.
- EK 4.1.1E Algorithms can be combined to make new algorithms.
- EK 4.1.1F Using existing correct algorithms as building blocks for constructing a new algorithm helps ensure the new algorithm is correct.

- **EK 4.1.1G Knowledge of standard algorithms can help in constructing new algorithms.**
- **EK 4.1.1H Different algorithms can be developed to solve the same problem.**
- EK 4.1.1I Developing a new algorithm to solve a problem can yield insight into the problem.

## LO 4.1.2 Express an algorithm in a language.

- **EK 4.1.2A Languages for algorithms include natural language, pseudocode, and visual and textual programming languages.**
- **EK 4.1.2B Natural language and pseudocode describe algorithms so that humans can understand them.**
- **K 4.1.2C Algorithms described in programming languages can be executed on a computer.**
- **EK 4.1.2D Different languages are better suited for expressing different algorithms.**
- **EK 4.1.2E Some programming languages are designed for specific domains and are better for expressing algorithms in those domains.**

## LO 4.2.1 Explain the difference between algorithms that run in a reasonable time and those that do not run in a reasonable time.

- **EK 4.2.1A Many problems can be solved in a reasonable time.**
- **EK 4.2.1B Reasonable time means that as the input size grows, the number of steps the algorithm takes is proportional to the square (or cube, fourth power, fifth power, etc.) of the size of the input.**
- **EK 4.2.1C Some problems cannot be solved in a reasonable time, even for small input sizes.**
- **EK 4.2.1D Some problems can be solved but not in a reasonable time. In these cases, heuristic approaches may be helpful to find solutions in reasonable time.**

## LO 4.2.2 Explain the difference between solvable and unsolvable problems in computer science.

- **EK 4.2.2A A heuristic is a technique that may allow us to find an approximate solution when typical methods fail to find an exact solution.**

- EK 4.2.2B Heuristics may be helpful for finding an approximate solution more quickly when exact methods are too slow.

## LO 4.2.4 Evaluate algorithms analytically and empirically for efficiency, correctness, and clarity.

- EK 4.2.4A Determining an algorithm's efficiency is done by reasoning formally or mathematically about the algorithm.
- EK 4.2.4B Empirical analysis of an algorithm is done by implementing the algorithm and running it on different inputs.
- EK 4.2.4C The correctness of an algorithm is determined by reasoning formally or mathematically about the algorithm, not by testing an implementation of the algorithm.
- EK 4.2.4D Different correct algorithms for the same problem can have different efficiencies.
- EK 4.2.4E Sometimes, more efficient algorithms are more complex.
- EK 4.2.4F Finding an efficient algorithm for a problem can help solve larger instances of the problem.
- EK 4.2.4G Efficiency includes both execution time and memory usage.
- EK 4.2.4H Linear search can be used when searching for an item in any list; binary search can be used only when the list is sorted.

# Abstraction

## Overview

- EU 2.1 A variety of abstractions built on binary sequences can be used to represent all digital data.
- EU 2.2 Multiple levels of abstraction are used to write programs or create other computational artifacts.
- EU 2.3 Models and simulations use abstraction to generate new understanding and knowledge.

## Reading from the Computer Science Field Guide

Start by reading through:

- [Data Representation](#)
- [Software Engineering - Layers of Abstraction](#)
- [Programming Languages](#)

## Learning objectives

The above chapter readings include specific knowledge for EK's marked in bold. Work to include unmarked learning objectives in the CS Field Guide is currently in progress.

### LO 2.1.1 Describe the variety of abstractions used to represent data.

- **EK 2.1.1A Digital data is represented by abstractions at different levels.**
- **EK 2.1.1B At the lowest level, all digital data are represented by bits.**
- **EK 2.1.1C At a higher level, bits are grouped to represent abstractions, including but not limited to numbers, characters, and color.**
- **EK 2.1.1D Number bases, including binary, decimal, and hexadecimal, are used to represent and investigate digital data.**

- EK 2.1.1E At one of the lowest levels of abstraction, digital data is represented in binary (base 2) using only combinations of the digits zero and one.
- EK 2.1.1F Hexadecimal (base 16) is used to represent digital data because hexadecimal representation uses fewer digits than binary.
- EK 2.1.1G Numbers can be converted from any base to any other base.

## LO 2.1.2 Explain how binary sequences are used to represent digital data.

- EK 2.1.2A A finite representation is used to model the infinite mathematical concept of a number.
- EK 2.1.2B In many programming languages, the fixed number of bits used to represent characters or integers limits the range of integer values and mathematical operations; this limitation can result in over flow or other errors.
- EK 2.1.2C In many programming languages, the fixed number of bits used to represent real numbers (as floating-point numbers) limits the range of floating-point values and mathematical operations; this limitation can result in round off and other errors.
- EK 2.1.2D The interpretation of a binary sequence depends on how it is used.
- EK 2.1.2E A sequence of bits may represent instructions or data.
- EK 2.1.2F A sequence of bits may represent different types of data in different contexts.

## LO 2.2.1 Develop an abstraction when writing a program or creating other computational artifacts.

- EK 2.2.1A The process of developing an abstraction involves removing detail and generalizing functionality.
- EK 2.2.1B An abstraction extracts common features from specific examples in order to generalize concepts.

- EK 2.2.1C An abstraction generalizes functionality with input parameters that allow software reuse.

## LO 2.2.2 Use multiple levels of abstraction to write programs.

- EK 2.2.2A Software is developed using multiple levels of abstractions, such as constants, expressions, statements, procedures, and libraries.
- EK 2.2.2B Being aware of and using multiple levels of abstraction in developing programs helps to more effectively apply available resources and tools to solve problems.

## LO 2.2.3 Identify multiple levels of abstractions that are used when writing programs.

- **EK 2.2.3A Different programming languages offer different levels of abstraction.**
- **EK 2.2.3B High-level programming languages provide more abstractions for the programmer and make it easier for people to read and write a program.**
- **EK 2.2.3C Code in a programming language is often translated into code in another (lower level) language to be executed on a computer.**
- **EK 2.2.3D In an abstraction hierarchy, higher levels of abstraction (the most general concepts) would be placed toward the top and lower level abstractions (the more specific concepts) toward the bottom.**
- EK 2.2.3E Binary data is processed by physical layers of computing hardware, including gates, chips, and components.
- EK 2.2.3F A logic gate is a hardware abstraction that is modeled by a Boolean function.
- EK 2.2.3G A chip is an abstraction composed of low-level components and circuits that perform a specific function.
- EK 2.2.3H A hardware component can be low level like a transistor or high level like a video card.
- EK 2.2.3I Hardware is built using multiple levels of abstractions, such as transistors, logic gates, chips, memory, motherboards, special purpose cards, and storage devices.
- EK 2.2.3J Applications and systems are designed, developed, and analyzed using levels of hardware, software, and conceptual abstractions.

- EK 2.2.3K Lower level abstractions can be combined to make higher level abstractions, such as short message services (SMS) or email messages, images, audio files, and videos.

## LO 2.3.1 Use models and simulations to represent phenomena.

- EK 2.3.1A Models and simulations are simplified representations of more complex objects or phenomena.
- EK 2.3.1B Models may use different abstractions or levels of abstraction depending on the objects or phenomena being posed.
- EK 2.3.1C Models often omit unnecessary features of the objects or phenomena that are being modeled.
- EK 2.3.1D Simulations mimic real-world events without the cost or danger of building and testing the phenomena in the real world.

## LO 2.3.2 Use models and simulations to formulate, refine, and test hypotheses.

- EK 2.3.2A Models and simulations facilitate the formulation and refinement of hypotheses related to the objects or phenomena under consideration.
- EK 2.3.2B Hypotheses are formulated to explain the objects or phenomena being modeled.
- EK 2.3.2C Hypotheses are refined by examining the insights that models and simulations provide into the objects or phenomena.
- EK 2.3.2D The results of simulations may generate new knowledge and new hypotheses related to the phenomena being modeled.
- EK 2.3.2E Simulations allow hypotheses to be tested without the constraints of the real world.
- EK 2.3.2F Simulations can facilitate extensive and rapid testing of models.
- EK 2.3.2G The time required for simulations is impacted by the level of detail and quality of the models and the software and hardware used for the simulation.
- EK 2.3.2H Rapid and extensive testing allows models to be changed to accurately reflect the objects or phenomena being modeled.

# Creativity

## Overview

- EU 1.1 Creative development can be an essential process for creating computational artifacts.
- EU 1.2 Computing enables people to use creative development processes to create computational artifacts for creative expression or to solve a problem.
- EU 1.3 Computing can extend traditional forms of human expression and experience.

## Reading from the Computer Science Field Guide

Start by reading through:

- Human Computer Interaction
  - [Interface Usability](#)

## Learning objectives

The above chapter readings include specific knowledge for EK's marked in bold. Work to include unmarked learning objectives in the CS Field Guide is currently in progress.

### LO 1.1.1 Apply a creative development process when creating computational artifacts.

- EK 1.1.1A A creative process in the development of a computational artifact can include, but is not limited to, employing nontraditional, nonprescribed techniques; the

use of novel combinations of artifacts, tools, and techniques; and the exploration of personal curiosities.

- EK 1.1.1B Creating computational artifacts employs an iterative and often exploratory process to translate ideas into tangible form.

## LO 1.2.1 Create a computational artifact for creative expression.

- EK 1.2.1A A computational artifact is something created by a human using a computer and can be, but is not limited to, a program, an image, audio, video, a presentation, or a Web page file.
- EK 1.2.1B Creating computational artifacts requires understanding of and use of software tools and services.
- EK 1.2.1C Computing tools and techniques are used to create computational artifacts and can include, but are not limited to, programming integrated development environments (IDEs), spreadsheets, 3D printers, or text editors.
- EK 1.2.1D A creatively developed computational artifact can be created by using nontraditional, nonprescribed computing techniques.
- EK 1.2.1E Creative expressions in a computational artifact can reflect personal expressions of ideas or interests.

## LO 1.2.2 Create a computational artifact using computing tools and techniques to solve a problem.

- EK 1.2.2A Computing tools and techniques can enhance the process of finding a solution to a problem.
- EK 1.2.2B A creative development process for creating computational artifacts can be used to solve problems when traditional or prescribed computing techniques are not effective.

## LO 1.2.3 Create a new computational artifact by combining or modifying existing artifacts.

- EK 1.2.3A Creating computational artifacts can be done by combining and modifying existing artifacts or by creating new artifacts.
- EK 1.2.3B Computation facilitates the creation and modification of computational artifacts with enhanced detail and precision.
- EK 1.2.3C Combining or modifying existing artifacts can show personal expression of ideas.

## LO 1.2.4 Collaborate in the creation of computational artifacts.

- EK 1.2.4A A collaboratively created computational artifact reflects effort by more than one person.
- EK 1.2.4B Effective collaborative teams consider the use of online collaborative tools.
- EK 1.2.4C Effective collaborative teams practice interpersonal communication, consensus building, conflict resolution, and negotiation.
- EK 1.2.4D Effective collaboration strategies enhance performance.
- EK 1.2.4E Collaboration facilitates the application of multiple perspectives (including sociocultural perspectives) and diverse talents and skills in developing computational artifacts.
- EK 1.2.4F A collaboratively created computational artifact can reflect personal expressions of ideas.

## LO 1.2.5 Analyze the correctness, usability, functionality, and suitability of computational artifacts.

- EK 1.2.5A The context in which an artifact is used determines the correctness, usability, functionality, and suitability of the artifact.
- EK 1.2.5B A computational artifact may have weaknesses, mistakes, or errors depending on the type of artifact.
- EK 1.2.5C The functionality of a computational artifact may be related to how it is used or perceived.
- EK 1.2.5D The suitability (or appropriateness) of a computational artifact may be related to how it is used or perceived.

## LO 1.3.1 Use computing tools and techniques for creative expression.

- EK 1.3.1A Creating digital effects, images, audio, video, and animations has transformed industries.
- EK 1.3.1B Digital audio and music can be created by synthesizing sounds, sampling existing audio and music, and recording and manipulating sounds, including layering and looping.
- EK 1.3.1C Digital images can be created by generating pixel patterns, manipulating existing digital images, or combining images.
- EK 1.3.1D Digital effects and animations can be created by using existing software or modified software that includes functionality to implement the effects and animations.

- EK 1.3.1E Computing enables creative exploration of both real and virtual phenomena.

# Data and Information

## Overview

- EU 3.1 People use computer programs to process information to gain insight and knowledge.
- EU 3.2 Computing facilitates exploration and the discovery of connections in information.
- EU 3.3 There are trade-offs when representing information as digital data.

## Reading from the Computer Science Field Guide

Start by reading through:

- [Data Representation](#)
- [Coding](#)
- [Compression](#)
- [Encryption](#)
- [Error Control](#)
- [Human Computer Interaction](#)

## Learning objectives

The above chapter readings include specific knowledge for EK's marked in bold. Work to include unmarked learning objectives in the CS Field Guide is currently in progress.

### LO 3.1.1 Find patterns and test hypotheses about digitally processed information to gain insight and knowledge.

- EK 3.1.1A Computers are used in an iterative and interactive way when processing digital information to gain insight and knowledge.
- EK 3.1.1B Digital information can be filtered and cleaned by using computers to process information.

- EK 3.1.1C Combining data sources, clustering data, and data classification are part of the process of using computers to process information.
- EK 3.1.1D Insight and knowledge can be obtained from translating and transforming digitally represented information.
- EK 3.1.1E Patterns can emerge when data is transformed using computational tools.

## LO 3.1.2 Collaborate when processing information to gain insight and knowledge.

- EK 3.1.2A Collaboration is an important part of solving data-driven problems.
- EK 3.1.2B Collaboration facilitates solving computational problems by applying multiple perspectives, experiences, and skill sets.
- EK 3.1.2C Communication between participants working on data-driven problems gives rise to enhanced insights and knowledge.
- EK 3.1.2D Collaboration in developing hypotheses and questions, and in testing hypotheses and answering questions, about data helps participants gain insight and knowledge.
- EK 3.1.2E Collaborating face-to-face and using online collaborative tools can facilitate processing information to gain insight and knowledge.
- EK 3.1.2F Investigating large data sets collaboratively can lead to insight and knowledge not obtained when working alone.

## LO 3.1.3 Explain the insight and knowledge gained from digitally processed data by using appropriate visualizations, notations, and precise language.

- EK 3.1.3A Visualization tools and software can communicate information about data.
- EK 3.1.3B Tables, diagrams, and textual displays can be used in communicating insight and knowledge gained from data.
- EK 3.1.3C Summaries of data analyzed computationally can be effective in communicating insight and knowledge gained from digitally represented information.
- EK 3.1.3D Transforming information can be effective in communicating knowledge gained from data.
- EK 3.1.3E Interactivity with data is an aspect of communicating.

## LO 3.2.1 Extract information from data to discover and explain connections or trends.

- EK 3.2.1A Large data sets provide opportunities and challenges for extracting information and knowledge.

- EK 3.2.1B Large data sets provide opportunities for identifying trends, making connections in data, and solving problems.
- EK 3.2.1C Computing tools facilitate the discovery of connections in information within large data sets.
- EK 3.2.1D Search tools are essential for efficiently finding information.
- EK 3.2.1E Information filtering systems are important tools for finding information and recognizing patterns in the information.
- EK 3.2.1F Software tools, including spreadsheets and databases, help to efficiently organize and find trends in information.
- EK 3.2.1G Metadata is data about data.
- EK 3.2.1H Metadata can be descriptive data about an image, a Web page, or other complex objects.
- EK 3.2.1I Metadata can increase the effective use of data or data sets by providing additional information about various aspects of that data.

## LO 3.2.2. Determine how large data sets impact the use of computational processes to discover information and knowledge.

- EK 3.2.2A Large data sets include data such as transactions, measurements, texts, sounds, images, and videos.
- EK 3.2.2B The storing, processing, and curating of large data sets is challenging.
- EK 3.2.2C Structuring large data sets for analysis can be challenging.
- EK 3.2.2D Maintaining privacy of large data sets containing personal information can be challenging.
- EK 3.2.2E Scalability of systems is an important consideration when data sets are large.
- EK 3.2.2F The size or scale of a system that stores data affects how that data set is used.
- EK 3.2.2G The effective use of large data sets requires computational solutions.

- EK 3.2.2H Analytical techniques to store, manage, transmit, and process data sets change as the size of data sets scale.

## LO 3.3.1 Analyze how data representation, storage, security, and transmission of data involve computational manipulation of information.

- **EK 3.3.1A Digital data representations involve trade-offs related to storage, security, and privacy concerns.**
- **EK 3.3.1B Security concerns engender trade-offs in storing and transmitting information.**
- **EK 3.3.1C There are trade-offs in using lossy and lossless compression techniques for storing and transmitting data.**

# Global Impact

## Overview

- EU 7.1 Computing enhances communication, interaction, and cognition.
- EU 7.2 Computing enables innovation in nearly every field.
- EU 7.3 Computing has a global affect – both beneficial and harmful – on people and society.
- EU 7.4 Computing innovations influence and are influenced by the economic, social, and cultural contexts in which they are designed and used.
- EU 7.5 An investigative process is aided by effective organization and selection of resources. Appropriate technologies and tools facilitate the accessing of information and enable the ability to evaluate the credibility of sources.

## Reading from the Computer Science Field Guide

### Learning objectives

The above chapter readings include specific knowledge for EK's marked in bold. Work to include unmarked learning objectives in the CS Field Guide is currently in progress.

#### LO 7.1.1 Explain how computing innovations affect communication, interaction, and cognition.

- EK 7.1.1A Email, SMS, and chat have fostered new ways to communicate and collaborate.
- EK 7.1.1B Video conferencing and video chat have fostered new ways to communicate and collaborate.
- EK 7.1.1C Social media continues to evolve and fosters new ways to communicate.
- EK 7.1.1D Cloud computing fosters new ways to communicate and collaborate.

- EK 7.1.1E Widespread access to information facilitates the identification of problems, development of solutions, and dissemination of results.
- EK 7.1.1F Public data provides widespread access and enables solutions to identified problems.
- EK 7.1.1G Search trends are predictors.
- EK 7.1.1H Social media, such as blogs and Twitter, have enhanced dissemination.
- EK 7.1.1I Global Positioning System (GPS) and related technologies have changed how humans travel, navigate, and find information related to geolocation.
- EK 7.1.1J Sensor networks facilitate new ways of interacting with the environment and with physical systems.
- EK 7.1.1K Smart grids, smart buildings, and smart transportation are changing and facilitating human capabilities.
- EK 7.1.1L Computing contributes to many assistive technologies that enhance human capabilities.
- EK 7.1.1M The Internet and the Web have enhanced methods of and opportunities for communication and collaboration.
- EK 7.1.1N The Internet and the Web have changed many areas, including e-commerce, health care, access to information and entertainment, and online learning.
- EK 7.1.1O The Internet and the Web have impacted productivity, positively and negatively, in many areas.

## LO 7.1.2 Explain how people participate in a problem-solving process that scales.

- EK 7.1.2A Distributed solutions must scale to solve some problems.
- EK 7.1.2B Science has been impacted by using scale and “citizen science” to solve scientific problems using home computers in scientific research.
- EK 7.1.2C Human computation harnesses contributions from many humans to solve problems related to digital data and the Web.
- EK 7.1.2D Human capabilities are enhanced by digitally enabled collaboration.
- EK 7.1.2E Some online services use the contributions of many people to benefit both individuals and society.
- EK 7.1.2F Crowdsourcing offers new models for collaboration, such as connecting people with jobs and businesses with funding.

- EK 7.1.2G The move from desktop computers to a proliferation of always-on mobile computers is leading to new applications.

## LO 7.2.1 Explain how computing has impacted innovations in other fields.

- EK 7.2.1A Machine learning and data mining have enabled innovation in medicine, business, and science.
- EK 7.2.1B Scientific computing has enabled innovation in science and business.
- EK 7.2.1C Computing enables innovation by providing the ability to access and share information.
- EK 7.2.1D Open access and Creative Commons have enabled broad access to digital information.
- EK 7.2.1E Open and curated scientific databases have benefited scientific researchers.
- EK 7.2.1F Moore's law has encouraged industries that use computers to effectively plan future research and development based on anticipated increases in computing power.
- EK 7.2.1G Advances in computing as an enabling technology have generated and increased the creativity in other fields.

## LO 7.3.1 Analyze the beneficial and harmful effects of computing.

- EK 7.3.1A Innovations enabled by computing raise legal and ethical concerns.
- EK 7.3.1B Commercial access to music and movie downloads and streaming raises legal and ethical concerns.
- EK 7.3.1C Access to digital content via peer-to-peer networks raises legal and ethical concerns.
- EK 7.3.1D Both authenticated and anonymous access to digital information raise legal and ethical concerns.
- EK 7.3.1E Commercial and governmental censorship of digital information raise legal and ethical concerns.
- EK 7.3.1F Open source and licensing of software and content raise legal and ethical concerns.
- EK 7.3.1G Privacy and security concerns arise in the development and use of computational systems and artifacts.
- EK 7.3.1H Aggregation of information, such as geolocation, cookies, and browsing history, raises privacy and security concerns.
- EK 7.3.1I Anonymity in online interactions can be enabled through the use of online anonymity software and proxy servers.

- EK 7.3.1J Technology enables the collection, use, and exploitation of information about, by, and for individuals, groups, and institutions.
- EK 7.3.1K People can have instant access to vast amounts of information online; accessing this information can enable the collection of both individual and aggregate data that can be used and collected.
- EK 7.3.1L Commercial and governmental curation of information may be exploited if privacy and other protections are ignored.
- EK 7.3.1M Targeted advertising is used to help individuals, but it can be misused at both individual and aggregate levels.
- EK 7.3.1N Widespread access to digitized information raises questions about intellectual property.
- EK 7.3.1O Creation of digital audio, video, and textual content by combining existing content has been impacted by copyright concerns.
- EK 7.3.1P The Digital Millennium Copyright Act (DMCA) has been a benefit and a challenge in making copyrighted digital material widely available.
- EK 7.3.1Q Open source and free software have practical, business, and ethical impacts on widespread access to programs, libraries, and code.

## LO 7.4.1 Explain the connections between computing and real-world contexts, including economic, social, and cultural contexts.

- EK 7.4.1A The innovation and impact of social media and online access varies in different countries and in different socioeconomic groups.
- EK 7.4.1B Mobile, wireless, and networked computing have an impact on innovation throughout the world.
- EK 7.4.1C The global distribution of computing resources raises issues of equity, access, and power.
- EK 7.4.1D Groups and individuals are affected by the “digital divide”—differing access to computing and the Internet based on socioeconomic or geographic characteristics.
- EK 7.4.1E Networks and infrastructure are supported by both commercial and governmental initiatives.

## LO 7.5.1 Access, manage, and attribute information using effective strategies.

- EK 7.5.1A Online databases and libraries catalog and house secondary and some primary sources.

- EK 7.5.1B Advance search tools, Boolean logic, and key words can refine the search focus, and/or limit their searches based on a variety of factors (e.g., data, peer-review status, type of publication).
- EK 7.5.1C Plagiarism is a serious offense that occurs when a person present another's ideas or words as his or her own. Plagiarism may be avoided by accurately acknowledging sources.

## LO 7.5.2 Evaluate online and print sources for appropriateness and credibility.

- EK 7.5.2A Determining the credibility of a source requires considering and evaluating the reputation and credentials of the author(s), publisher(s), site owner(s), and/or sponsor(s).
- EK 7.5.2B Information from a source is considered relevant when it supports an appropriate claim or the purpose of the investigation.

# Programming

## Overview

- EU 5.1 Programs can be developed for creative expression, to satisfy personal curiosity, to create new knowledge, or to solve problems (to help people, organizations, or society).
- EU 5.2 People write programs to execute algorithms.
- EU 5.3 Programming is facilitated by appropriate abstractions.
- EU 5.4 Programs are developed, maintained, and used by people for different purposes.
- EU 5.5 Programming uses mathematical and logical concepts.

## Reading from the Computer Science Field Guide

Start by reading through:

- [Programming Languages](#)
- [Software Engineering](#)
- [Human Computer Interaction](#)

The AP CSP framework does not require a specific programming language but require students to learn programming. Choose one or more of the options from:

- [Programming Introduction](#)

## Learning objectives

The above chapter readings include specific knowledge for EK's marked in bold. Work to include unmarked learning objectives in the CS Field Guide is currently in progress.

## LO 5.1.1 Develop a program for creative expression, to satisfy personal curiosity, or to create new knowledge.

- EK 5.1.1A Programs are developed and used in a variety of ways by a wide range of people depending on the goals of the programmer.
- EK 5.1.1B Programs developed for creative expression, to satisfy personal curiosity, or to create new knowledge may have visual, audible, or tactile inputs and outputs.
- EK 5.1.1C Programs developed for creative expression, to satisfy personal curiosity, or to create new knowledge may be developed with different standards or methods than programs developed for widespread distribution.
- EK 5.1.1D Additional desired outcomes may be realized independently of the original purpose of the program.
- EK 5.1.1E A computer program or the results of running a program may be rapidly shared with a large number of users and can have widespread impact on individuals, organizations, and society.
- EK 5.1.1F Advances in computing have generated and increased creativity in other fields.

## LO 5.1.2 Develop a correct program to solve problems.

- EK 5.1.2A An iterative process of program development helps in developing a correct program to solve problems.
- EK 5.1.2B Developing correct program components and then combining them helps in creating correct programs.
- EK 5.1.2C Incrementally adding tested program segments to correct working programs helps create large correct programs.
- EK 5.1.2D Program documentation helps programmers develop and maintain correct programs to efficiently solve problems.
- EK 5.1.2E Documentation about program components, such as blocks and procedures, helps in developing and maintaining programs.
- EK 5.1.2F Documentation helps in developing and maintaining programs when working individually or in collaborative programming environments.
- EK 5.1.2G Program development includes identifying programmer and user concerns that affect the solution to problems.
- EK 5.1.2H Consultation and communication with program users is an important aspect of program development to solve problems.
- EK 5.1.2I A programmer's knowledge and skill affects how a program is developed and how it is used to solve a problem.
- EK 5.1.2J A programmer designs, implements, tests, debugs, and maintains programs when solving problems.

## LO 5.1.3 Collaborate to develop a program.

- EK 5.1.3A Collaboration can decrease the size and complexity of tasks required of individual programmers.
- EK 5.1.3B Collaboration facilitates multiple perspectives in developing ideas for solving problems by programming.
- EK 5.1.3C Collaboration in the iterative development of a program requires different skills than developing a program alone.
- EK 5.1.3D Collaboration can make it easier to find and correct errors when developing programs.
- EK 5.1.3E Collaboration facilitates developing program components independently.
- EK 5.1.3F Effective communication between participants is required for successful collaboration when developing programs.

## LO 5.2.1 Explain how programs implement algorithms.

- **EK 5.2.1A Algorithms are implemented using program instructions that are processed during program execution.**
- EK 5.2.1B Program instructions are executed sequentially.
- EK 5.2.1C Program instructions may involve variables that are initialized and updated, read, and written.
- EK 5.2.1D An understanding of instruction processing and program execution is useful for programming.
- EK 5.2.1E Program execution automates processes.
- EK 5.2.1F Processes use memory, a central processing unit (CPU), and input and output.
- EK 5.2.1G A process may execute by itself or with other processes.
- EK 5.2.1H A process may execute on one or several CPUs.
- EK 5.2.1I Executable programs increase the scale of problems that can be addressed.
- EK 5.2.1J Simple algorithms can solve a large set of problems when automated.
- EK 5.2.1K Improvements in algorithms, hardware, and software increase the kinds of problems and the size of problems solvable by programming.

## LO 5.3.1 Use abstraction to manage complexity in programs.

- EK 5.3.1A Procedures are reusable programming abstractions.
- EK 5.3.1B A procedure is a named grouping of programming instructions.
- EK 5.3.1C Procedures reduce the complexity of writing and maintaining programs.

- EK 5.3.1D Procedures have names and may have parameters and return values.
- EK 5.3.1E Parameterization can generalize a specific solution.
- EK 5.3.1F Parameters generalize a solution by allowing a procedure to be used instead of duplicated code.
- EK 5.3.1G Parameters provide different values as input to procedures when they are called in a program.
- EK 5.3.1H Data abstraction provides a means of separating behavior from implementation.
- EK 5.3.1I Strings and string operations, including concatenation and some form of substring, are common in many programs.
- EK 5.3.1J Integers and floating-point numbers are used in programs without requiring understanding of how they are implemented.
- EK 5.3.1K Lists and list operations, such as add, remove, and search, are common in many programs.
- EK 5.3.1L Using lists and procedures as abstractions in programming can result in programs that are easier to develop and maintain.
- EK 5.3.1M Application program interfaces (APIs) and libraries simplify complex programming tasks.
- EK 5.3.1N Documentation for an API/library is an important aspect of programming.
- EK 5.3.1O APIs connect software components, allowing them to communicate.

## LO 5.4.1 Evaluate the correctness of a program.

- EK 5.4.1A Program style can affect the determination of program correctness.
- EK 5.4.1B Duplicated code can make it harder to reason about a program.
- EK 5.4.1C Meaningful names for variables and procedures help people better understand programs.
- EK 5.4.1D Longer code blocks are harder to reason about than shorter code blocks in a program.
- EK 5.4.1E Locating and correcting errors in a program is called debugging the program.
- EK 5.4.1F Knowledge of what a program is supposed to do is required in order to find most program errors.
- EK 5.4.1G Examples of intended behavior on specific inputs help people understand what a program is supposed to do.
- EK 5.4.1H Visual displays (or different modalities) of program state can help in finding errors.
- EK 5.4.1I Programmers justify and explain a program's correctness.
- EK 5.4.1J Justification can include a written explanation about how a program meets its specifications.

- EK 5.4.1K Correctness of a program depends on correctness of program components, including code blocks and procedures.
- EK 5.4.1L An explanation of a program helps people understand the functionality and purpose of it.
- EK 5.4.1M The functionality of a program is often described by how a user interacts with it.
- EK 5.4.1N The functionality of a program is best described at a high level by what the program does, not at the lower level of how the program statements work to accomplish this.

## LO 5.5.1 Employ appropriate mathematical and logical concepts in programming.

- **EK 5.5.1A Numbers and numerical concepts are fundamental to programming.**
- **EK 5.5.1B Integers may be constrained in the maximum and minimum values that can be represented in a program because of storage limitations.**
- **EK 5.5.1C Real numbers are approximated by floating-point representations that do not necessarily have infinite precision.**
- EK 5.5.1D Mathematical expressions using arithmetic operators are part of most programming languages.
- EK 5.5.1E Logical concepts and Boolean algebra are fundamental to programming.
- EK 5.5.1F Compound expressions using and, or, and not are part of most programming languages.
- EK 5.5.1G Intuitive and formal reasoning about program components using Boolean concepts helps in developing correct programs.
- EK 5.5.1H Computational methods may use lists and collections to solve problems.
- EK 5.5.1I Lists and other collections can be treated as abstract data types (ADTs) in developing programs.
- EK 5.5.1J Basic operations on collections include adding elements, removing elements, iterating over all elements, and determining whether an element is in a collection.

# The Internet

## Overview

- EU 6.1 The Internet is a network of autonomous systems.
- EU 6.2 Characteristics of the Internet influence the systems built on it.
- EU 6.3 Cybersecurity is an important concern for the Internet and the systems built on it.

## Reading from the Computer Science Field Guide

- Network Communication Protocols
- Coding - Encryption

## Learning objectives

The above chapter readings include specific knowledge for EK's marked in bold. Work to include unmarked learning objectives in the CS Field Guide is currently in progress.

### LO 6.1.1 Explain the abstractions in the Internet and how the Internet functions.

- EK 6.1.1A The Internet connects devices and networks all over the world.
- EK 6.1.1B An end-to-end architecture facilitates connecting new devices and networks on the Internet.
- **EK 6.1.1C Devices and networks that make up the Internet are connected and communicate using addresses and protocols.**
- EK 6.1.1D The Internet and the systems built on it facilitate collaboration.
- EK 6.1.1E Connecting new devices to the Internet is enabled by assignment of an Internet protocol (IP) address.
- EK 6.1.1F The Internet is built on evolving standards, including those for addresses and names.

- EK 6.1.1G The domain name system (DNS) translates names to IP addresses.
- EK 6.1.1H The number of devices that could use an IP address has grown so fast that a new protocol (IPv6) has been established to handle routing of many more devices.
- EK 6.1.1I Standards such as hypertext transfer protocol (HTTP), IP, and simple mail transfer protocol (SMTP) are developed and overseen by the Internet Engineering Task Force (IETF).

## LO 6.2.1 Explain characteristics of the Internet and the systems built on it.

- EK 6.2.1A The Internet and the systems built on it are hierarchical and redundant.
- EK 6.2.1B The domain name syntax is hierarchical.
- EK 6.2.1C IP addresses are hierarchical.
- EK 6.2.1D Routing on the Internet is fault tolerant and redundant.

## LO 6.2.2 Explain how the characteristics of the Internet influence the systems built on it.

- EK 6.2.2A Hierarchy and redundancy help systems scale.
- EK 6.2.2B The redundancy of routing (i.e., more than one way to route data) between two points on the Internet increases the reliability of the Internet and helps it scale to more devices and more people.
- EK 6.2.2C Hierarchy in the DNS helps that system scale.
- **EK 6.2.2D Interfaces and protocols enable widespread use of the Internet.**
- EK 6.2.2E Open standards fuel the growth of the Internet.
- **EK 6.2.2F The Internet is a packet-switched system through which digital data is sent by breaking the data into blocks of bits called packets, which contain both the data being transmitted and control information for routing the data.**
- **EK 6.2.2G Standards for packets and routing include transmission control protocol/Internet protocol (TCP/IP).**
- **EK 6.2.2H Standards for sharing information and communicating between browsers and servers on the Web include HTTP and secure sockets layer/transport layer security (SSL/TLS).**
- EK 6.2.2I The size and speed of systems affect their use.

- EK 6.2.2J The bandwidth of a system is a measure of bit rate—the amount of data (measured in bits) that can be sent in a fixed amount of time.
- EK 6.2.2K The latency of a system is the time elapsed between the transmission and the receipt of a request.

## LO 6.3.1 Identify existing cybersecurity concerns and potential options to address these issues with the Internet and the systems built on it.

- EK 6.3.1A The trust model of the Internet involves trade-offs.
- EK 6.3.1B The DNS was not designed to be completely secure.
- EK 6.3.1C Implementing cybersecurity has software, hardware, and human components.
- EK 6.3.1D Cyber warfare and cyber crime have widespread and potentially devastating effects.
- EK 6.3.1E Distributed denial-of-service attacks (DDoS) compromise a target by flooding it with requests from multiple systems.
- EK 6.3.1F Phishing, viruses, and other attacks have human and software components.
- EK 6.3.1G Antivirus software and firewalls can help prevent unauthorized access to private data.
- **EK 6.3.1H Cryptography is essential to many models of cybersecurity.**
- **EK 6.3.1I Cryptography has a mathematical foundation.**
- EK 6.3.1J Open standards help ensure cryptography is secure.
- **EK 6.3.1K Symmetric encryption is a method of encryption involving one key for encryption and decryption.**
- **EK 6.3.1L Public key encryption, which is not symmetric, is an encryption method that is widely used because of the functionality it provides.**
- EK 6.3.1M Certificate authorities (CAs) issue digital certificates that validate the ownership of encrypted keys used in secured communications and are based on a trust model.

# Glossary

## Algorithm

A step by step process that describes how to solve a problem and/or complete a task, which will always give a result.

See also [introduction](#), [computer program](#), [algorithm cost](#), [searching algorithms](#), and [sorting algorithms](#).

## Alphabet

In formal languages, a list of characters that may occur in a language, or more generally, a list of all possible inputs that might happen.

See also [Formal languages](#).

## Binary Search

Searching a sorted list by looking at the middle item, and then searching the appropriate half recursively (used for phone books, dictionaries and computer algorithms).

## Brooks' law

An observation that adding more people to a project that is running late may actually slow it down more.

See also [software engineering](#).

## Chomsky Hierarchy

A hierarchy of four classifications of formal languages, ranging from simple regular expressions to very flexible (but computationally difficult) grammars.

See also [Formal languages](#).

## Complexity

How long it takes to solve a problem. A problem has an inherent complexity (minimum time needed to solve it); any algorithm to solve the problem will have a higher complexity (take at least that long).

See also [problems and algorithms](#).

## Digital signature

An encryption system that allows the receiver to verify that a document was sent by the person who claims to have sent it.

## Finite State Automaton

In formal languages, a simple 'machine' that has states, and transitions from one state to another based on strings of input symbols.

See also [Formal languages](#), [FSA abbreviation](#), [Formal languages](#), and [related to regular expressions](#).

## Finite State Machine

Alternative name for a finite state automaton.

See also [Formal languages](#).

## Grammar

In formal languages, a set of rules for specifying a language, for example, to specify syntax for programming languages.

See also [Formal languages](#).

## Interpolation

Working out values between some given values; for example, if a sequence of 5 numbers starts with 3 and finishes with 11, we might interpolate the values 5, 7, 9 in between.

See also [compressing images](#).

## Language

In formal languages, it's the set of all strings that the language accepts i.e. that are correct.

See also [Formal languages](#), and [regular expression](#).

## Pattern matching

In formal languages, finding text that matches a particular rule, typically using a regular expression to give the rule.

See also [Formal languages](#).

## Pixel

This term is an abbreviation of *picture element*, the name given to the tiny squares that make up a grid that is used to represent images on a computer.

See also [definition](#).

## Quicksort

A process for achieving an outcome, normally for a general problem such as searching, sorting, finding an optimal path through a map and so on.

## Regular expression

A formula used to describe a pattern in a text that is to be matched or searched for. These are typically used for finding elements of a program (such as variable names) and checking input in forms (such as checking that an email address has the right format.)

See also [introduction](#), and [abbreviations](#).

## Slope

This is a way of expressing the angle or gradient of a line. The slope is simply how far up the line goes for every unit we move to the right. For example, if we have a line with a slope of 2, then after moving 3 units to the right, it will have gone up 6 units. A line with a slope of 0 is horizontal. Normally the slope of a line is represented using the symbol  $m$ .

See also [computer graphics](#).

## String

A sequence of characters.

See also [Formal languages](#), and [regular expression](#).

## tractable

A *tractable* problem is one that can be solved in a reasonable amount of time; usually the distinction between tractable and intractable is drawn at the boundary between problems that can be solved in an amount of time that is polynomial; those that require exponential time are regarded as intractable.

See also [encryption](#), and [complexity and tractability chapter](#).

## Transition

In a finite state machine, the links between the states.

See also [Formal languages](#).

# Contributors

This project is the result of contributions from a large community of Computer Science education enthusiasts from all over the world.

## Project Leads

- Tim Bell - International Field Guide, co-founder
- Peter Denning - International Field Guide, co-founder
- Jack Morgan - Project Manager

## Software Development

- Jack Morgan - Technical Lead
- Jordan Griffiths

## Interactive Development

- Jack Morgan
- David Thompson
- Rhem Munro
- Heidi Newton
- James Browning
- Sam Jarman
- Hayley van Waas
- Hannah Taylor
- Marcus Stenfert Kroese
- Victor Chang

## Writers

- Tim Bell - Formal Languages, Compression, Human Computer Interaction, and Coding Introduction

- Heidi Newton - NCEA Assessment Guides, Programming Languages, Data Representation, Compression, Encryption, Error Control, Artificial Intelligence, and Complexity and Tractability
- Caitlin Duncan - Algorithms
- Sam Jarman - Network Communication Protocols
- David Thompson - Computer Vision
- Rhem Munro - Computer Graphics
- Janina Voigt - Software Engineering
- Jon Rutherford - Software Engineering
- Joshua Scott - Computer Graphics

## Editors

- Tim Bell
- Heidi Newton
- Hayley van Waas
- Jack Morgan
- Ian Witten

## Advisors

- Mike Fellows - [CS Unplugged](#)
- Andrea Arpaci-Dusseau - [CS Unplugged](#)
- Paul Curzon - [CS4FN](#)
- Quintin Cutts - [Computing Science Inside](#)
- Calvin Lin - [Thriving in our Digital World](#)
- Bradley Beth - [Thriving in our Digital World](#)
- Peter Andreae - Artificial Intelligence, Complexity and Tractability, Data Representation, Compression, Error Control, and Encryption
- Wal Irwin - Software Engineering
- Patrick Baker - Curriculum (NCEA)
- Jenny Baker - Curriculum (NCEA)
- Neil Leslie - Curriculum (NCEA)
- Hannah Taylor - Curriculum (NCEA)
- Dr Mukundan - Computer Graphics & Vision
- Richard Green - Computer Vision
- DongSeong Kim - Network Communication Protocols, Encryption
- Andreas Willig - Network Communication Protocols
- Walter Guttmann - Formal Languages
- Joshua Scott - Programming Languages

- Brad Miller ([Runestone Interactive](#))
- David Ranum ([Runestone Interactive](#))
- Ian Witten
- Anthony Robins
- Shadi Ibrahim
- Renate Thies
- Jan Vahrenhold
- Paul Matthews

## Other

- Suman Murugesh - Research
- Marcus Stenfert Kroese
- Ben Gibson - Interactive games and related material
- Michael Bell ([Orange Studio](#)) - Video production
- Linda Pettigrew - Formal languages material

## Community Contributors

- [ArloL](#) (Arlo Louis O'Keeffe)
- [ner0x652](#) (Cornel Punga)
- [alanhogans](#) (Alan Hogan)
- [StevenMaude](#) (Steven Maude)
- [rdpse](#) (Rúben Enes)
- [oughter](#)
- [digitalDojoNZ](#)
- [pvskarthikeya](#) (Karthikeya Pammi)

**Note:** If there is an error in the list, please contact [Jack Morgan](#)

## Acknowledgements

This project has grown out of a partnership for an international field guide to computing. In the [University of Canterbury](#) CSSE department, it grew from the [Computer Science Unplugged project](#) and the [New Zealand Computer Science resource guides](#). At the Naval Postgraduate School in Monterey, California, it grew from a project called [A Field Guide to the Science of Computation](#) and the [Great Principles of Computing project](#), both led by Peter Denning.

Funding for this online textbook has been provided by Google Inc. In addition, countless hours of volunteer time have been contributed by those listed above. Tim Bell prepared an initial draft of this material while visiting Huazhong University of Science and Technology, Wuhan, China, whom we thank for providing an excellent environment for writing. The project is based at the [University of Canterbury](#), Christchurch, New Zealand; other authors are at [Victoria University of Wellington](#), New Zealand, and [Cambridge University](#), UK.

Partial funding for the US field guide project was provided by the US National Science Foundation under Grant No. 0938809.

# Interactives

We have plenty of interactives throughout the CSFG, to teach many different computer science concepts. This page details some troubleshooting tips if you encounter issues, and a list of all available interactives.

## Troubleshooting

Most of our interactives require a modern browser, if it's been updated in the last year you should be fine. We recommend:

- [Google Chrome](#)
- [Mozilla Firefox](#)
- [Microsoft Edge](#)
- [Safari](#)
- [Opera](#)

While most interactives work on both phones, tablets, and desktop computers, some of our more complex interactives require a desktop computer to achieve acceptable performance.

## WebGL

The computer graphics and vision chapters use [WebGL](#), which is a system that can render 3D images in a web browser. It is relatively new, so older browsers and operating systems may not have it setup correctly. The [CanIUse website](#) is a quick way to check if WebGL will work in your browser on your operating system. The general rule of thumb is if you are using an up-to-date version of a browser and the drivers for your operating system are up-to-date and the computer has a suitable GPU, then it should work.

If you are still having issues, this [answer on SuperUser](#) is quite useful. Also try searching your browser version and operating system in Google.

## Available Interactives

- [Action Menu](#)
- [Available Menu Items](#)

- Awful Calculator
- Base Calculator
- Big Number Calculator
- Binary Cards
- Caesar Cipher
- Checksum Calculator
  - GTIN-13 Checksum Calculator
- Close Window
- CMY Colour Mixer
- Colour Matcher
- Compression Comparer
- Confused Buttons
- Date Picker
- Deceiver
- Delay Analyser
- Delayed Checkbox
- Frequency Analysis
- High Score Boxes
- Image Bit Comparer
  - Change bit mode
- MIPS Assembler
- MIPS Simulator
- No Help
- Number Generator
- Packet Attack
- Packet Attack Level Creator
- Parity Trick
  - Setting parity only
  - Detecting errors only
  - Sandbox mode
- Payment Interface
- Pixel Viewer
- Python Interpreter
- Regular Expression Filter
- Regular Expression Search
- RGB Colour Mixer
- RSA Key Generator
- RSA Encrypter (no padding)
- RSA Decrypter (no padding)
- Run Length Encoding

- SHA2
- Searching Algorithms
- Sorting Algorithm Comparison
- Sorting Algorithms
- Trainsylvania
- Trainsylvania Planner
- Unicode Binary
- Unicode Character
- Unicode Length

# Releases

This page lists updates to the Computer Science Field Guide. All notable changes to this project will be documented in this file.

## What Numbering System Do We Use?

We base our numbering system from the guidelines at [Semantic Versioning 2.0.0](#), however since our project started before it was migrated to GitHub, the first open source release is being labeled as v2.0.0.

Given a version number MAJOR.MINOR.HOTFIX:

- MAJOR version change when major text modifications are made (for example: new chapter, changing how a curriculum guide teaches a subject).
- MINOR version change when content or functionality is added or updated (for example: new videos, new activities, large number of text (typo/grammar) fixes).
- HOTFIX version change when bug hotfixes are made (for example: fixing a typo, fixing a bug in an interactive).
- A pre-release version is denoted by appending a hyphen and the alpha label followed by the pre-release version.

We have listed major changes for each release below.

## Current Release

v2.8.1

**Release date:** 21st October 2016

**Downloads:** [Source downloads are available on GitHub](#)

**Changelog:**

- Update introduction chapter.
  - [Jack Morgan #231](#)

- Add notice of changes to AP-CSP curriculum in Fall 2016 release.
  - [Jack Morgan](#)
- Skip parsing # characters at start of Markdown links.
  - [Jack Morgan](#)

# Older Releases

## v2.8.0

**Release date:** 19th October 2016

**Downloads:** [Source downloads are available on GitHub](#)

### Notable changes:

This release adds an introductory video for the Human Computer Interaction chapter, plus a draft of guides for mapping the Computer Science Field Guide to the AP CSP curriculum.

### Changelog:

- Add introductory video to Human Computer Interaction chapter.
  - [Hayley van Waas](#)
- Add draft of guides for the AP CSP curriculum.
  - [James Atlas #316](#)
- Update and fix issues in high-score-boxes interactive.
  - [Victor Chang #390](#)
  - [Jack Morgan #391 #393](#)
- Add subtraction command to mips-simulator interactive. The interactive can now handle subtraction down to zero.
  - [Jack Morgan #382](#)
- Update sponsor information in footer.
  - [Jack Morgan](#)
- Improve the visibility of warning panels.
  - [Jack Morgan #389](#)
- Fix positioning of table of contents sidebar.
  - [Jack Morgan #387](#)
- Fix typos in Formal Languages chapter.
  - [Steven Maude #385](#)
- Update 404 page to avoid updating after each release.
  - [Karthikeya Pammi #394](#)
- Remove duplicate introduction to teacher guide.
  - [Jack Morgan #213](#)

- Add link to article on representing a 1 bit image.
  - [Jack Morgan #376](#)
- Fix broken link to contributors page in footer.
  - [Jack Morgan #383](#)
- Replace broken link to Eliza chatterbot.
  - [Jack Morgan #384](#)
- Fix footer link colour in teacher version.
  - [Jack Morgan #395](#)

## v2.7.1

**Release date:** 5th September 2016

**Downloads:** [Source available on GitHub](#)

**Notable changes:**

- Fixed broken link in footer to contributors page.

A full list of changes in this version is [available on GitHub](#).

## v2.7.0

**Release date:** 23rd August 2016

**Downloads:** [Source available on GitHub](#)

**Notable changes:**

- **New video:** Formal Languages now has an introductory video.
- **New interactive:** The [hexadecimal background colour interactive](#) allows a user to change the background colour of the page.
- **New curriculum guide:** A guide for NCEA [Artificial Intelligence: Turing Test](#) has been added.
- **Updated interactives:** The [box translation](#) and [box rotation](#) interactives are now open source and have been given a new look and made mobile friendly.
- **Generation improvements:** Basic translation support. Settings are now specific to each language, and contain the translation text.
- **Website improvements:** Added [help guide](#) for WebGL interactives.
- Also includes bug fixes to interactives, new links to supporting videos, and various text corrections from our staff and contributors.

A full list of changes in this version is [available on GitHub](#).

## v2.6.1

**Release date:** 14th July 2016

**Downloads:** [Source available on GitHub](#)

### Notable changes:

- Fixed issue on Human Computer Interaction chapter where duplicate library was causing several UI elements to not behave correctly.

## v2.6.0

**Release date:** 16th June 2016

**Downloads:** [Source available on GitHub](#)

### Notable changes:

- **New feature:** PDF output - A downloadable PDF of both student and teacher versions is now available from the homepage. The PDF also functions well as an ebook, with functional links to headings, glossary entries, interactives, and online resources.
- **New feature:** Printer friendly webpages - When printing a page of the CSFG through a browser, the page displays in a printer friendly manner by hiding navigational panels, opening all panels, and providing extra links to online resources.
- **New interactive:** The [binary cards interactive](#) emulates the Binary Cards CS Unplugged activity, used to teach binary numbers.
- **New interactive:** The [high score boxes interactive](#) was developed to give an example of searching boxes to find a maximum value to the student.
- **New interactive:** The [action menu interactive](#) is a small dropdown menu with one option that has severe consequences, but no confirmation screen if the user selects that option (used to demonstrate a key HCI concept).
- **Updated interactive:** The [trainsylvania interactive](#) (and supporting images/files) have been given a fresh coat of colour and a new station name.
- **Updated interactive:** The [trainsylvania planner interactive](#) is used alongside the trainsylvania interactive, and allows the user to input a path of train trips to see the resulting destination.
- **Updated interactive:** The [base calculator](#) allows a student to calculate a number, using a specific number base (binary, hexadecimal, etc).
- **Updated interactive:** The [big number calculator](#) allows a student to perform calculations with very large numbers/results.

- **Website improvements:** Redesigned homepage and footer with useful links and a splash of colour. Math equations are now line wrapped, and MathJax is loaded from a CDN. Images can now have text wrapped around them on a page.
- **Generation improvements:** Improvements to internal link creation (glossary links in particular). Separated dependency installation from generation script (see documentation for how to install and run generation script).
- **Project improvements:** Added documentation for contributing to and developing this project, including a code of conduct.

A full list of changes in this version is [available on GitHub](#).

## v2.5.0

**Release date:** 13th May 2016

**Downloads:** [Source available on GitHub](#)

### Notable changes:

- The Data Representation chapter has been rewritten and reorganised to be easier to follow, and three NCEA assessment guides have been written for students aiming for merit/excellence:
  - [Numbers \(Two's Complement\)](#)
  - [Text \(Unicode\)](#)
  - [Colours \(Various bit depths\)](#)

The chapter and assessment guides have been rewritten to take account of new feedback from the marking process and our own observations of student work.

As part of the rewrite of the Data Representation chapter, the following interactives were developed:

- New interactive: The [unicode binary interactive](#) displays the binary for a given character (or character by decimal number) dynamically with different encodings.
- New interactive: The [unicode character interactive](#) displays the character for a given decimal value.
- New interactive: The [unicode length interactive](#) displays the length (in bits) of text encoded using different encodings.
- Updated interactive: The [colour matcher interactive](#) has been redesigned to display values in binary, plus allow students to see and edit the bit values. The interface has also been restructured for readability and ease of use.

The old version of the Data Representation chapter can be [found here](#).

- Website improvements: A new image previewer was implemented, along with bugfixes to iFrame and panel rendering.
- Generation improvements: The Markdown parser has been replaced due to existing parsing issues. The new parser also gives us a large performance boost. A text box tag has also been added to highlight important text.

A full list of changes in this version is [available on GitHub](#).

## v2.4.1

**Release date:** 29th April 2016

**Downloads:** [Source available on GitHub](#)

**Notable changes:**

- Fixed version numbering system to allow hotfix changes

A full list of changes in this version is [available on GitHub](#).

## v2.4

**Release date:** 29th April 2016

**Downloads:** [Source available on GitHub](#)

**Notable changes:**

- Large number of typo, grammar, link, and math syntax fixes and also content corrections by contributors.
- New interactive: Added [GTIN-13 checksum calculator interactive](#) for calculating the last digit for a GTIN-13 barcode.
- Updated interactive: The [regular expression search interactive](#) has been updated and added to the repository.
- Updated interactive: The [image bit comparer interactive](#) has been updated and added to the repository. It also has a [changing bits mode](#) which allows the user to modify the number of bits for storing each colour.
- Added XKCD mouseover text (similar behaviour to website).
- Added feedback modal to allow developers to directly post issues to GitHub.
- Added encoding for HTML entities to stop certain characters not appearing correctly in browsers.

- Added summary of output at end of generation script.
- Added message for developers to contribute in the web console.

A full list of changes in this version is [available on GitHub](#).

## v2.3

**Release date:** 10th March 2016

**Downloads:** [Source available on GitHub](#)

**Notable changes:**

- Readability improvements to text within many chapters (grammer issues/typos) and to the Python scripts within the Algorithms chapter.
- Updated interactive: The RSA [encryption](#) and [decryption](#) interactives within Encryption have been updated and added to the repository.
- Updated interactive: The [searching algorithms interactive](#) within Algorithms have been updated and added to the repository.
- Updated interactive: The [word filter interactive](#) within Formal Languages have been updated and added to the repository.
- Updated interactives: Both the [MIPS assembler](#) and [MIPS simulator](#) were made open source by the original author, and we were given permission to incorporate our repository, and have been added to Programming Languages.
- A list of all interactives are now available on the [interactives page](#).

A full list of changes in this version is [available on GitHub](#).

## v2.2

**Release date:** 19th February 2016

**Downloads:** [Source available on GitHub](#)

**Notable changes:**

- New interactive: Parity trick with separate modes for [practicing setting parity](#), [practicing detecting parity](#), and [the whole trick](#). Also has a [sandbox mode](#).
- Updated interactives: Two colour mixers, one for [RGB](#) and one for [CMY](#) have been added.
- Updated interactive: A [colour matcher interactive](#) has been added for matching a colour in both 24 bit and 8 bit.
- Updated interactive: A [python interpreter interactive](#) has been added for the programming languages chapter.

- Website improvements: Code blocks now have syntax highlighting when a language is specified, dropdown arrows are fixed in Mozilla Firefox browsers, and whole page interactives now have nicer link buttons.

A full list of changes in this version is [available on GitHub](#).

## v2.1

**Release date:** 12th February 2016

**Downloads:** [Source available on GitHub](#)

**Notable changes:**

- Fixed many broken links and typos from 2.0.0
- Added calculator interactives to Introduction
- Added RSA key generator to Encryption
- Rewritten Braille Section in Data Representation

A full list of changes in this version is [available on GitHub](#).

## v2.0

**Release date:** 5th February 2016

**Downloads:** [Source available on GitHub](#)

**Notable changes:**

- First open source release
- Produces both student and teacher versions
- Produces landing page for selecting language
- Added new NCEA curriculum guides on Encryption and Human Computer Interaction

A full list of changes in this version is [available on GitHub](#).

**Comments:** The first major step in releasing a open source version of the Computer Science Field Guide. While some content (most notably interactives) have yet to be added to the new system, we are releasing this update for New Zealand teachers to use at the beginning of their academic year. For any interactives that are missing, links are in place to the older interactives.

## v2.0-alpha.3

**Release date:** 29th January 2016

**Downloads:** [Source available on GitHub](#)

## v2.0-alpha.2

**Release date:** 25th January 2016

## v2.0-alpha.1

**Release date:** 2nd December 2015

**Comments:** Released at CS4HS 2015.

## 1.?.?

**Release date:** 3rd February 2015

**Comments:** The last version of the CSFG before the open source version was adopted.

[This release is archived for viewing here.](#)

# Included Files

This page details files that are used within the Computer Science Field Guide. Files that are used in the CSFG that are not listed on this page add a warning to the output log upon generation.

xkcd-alice-and-bob.png, xkcd-cant-sleep-comic.png, xkcd-estimation.png, xkcd-good-code.png, xkcd-np-complete-cartoon.png, xkcd-password-strength.png, xkcd-protocol.png, xkcd-regular-expressions.png, xkcd-standards-cartoon.png, xkcd-tags.png, xkcd-tasks.png, xkcd-the-general-problem.png

**Owner:** Randall Munroe

**Used:** Comic images are used within text chapters.

**License:** This work is licensed under a Creative Commons Attribution-NonCommercial 2.5 License. This means that you are free to copy and reuse any of my drawings (noncommercially) as long as you tell people where they're from.

See [xkcd.com/license](http://xkcd.com/license) for more details.

coloured-roof-small.png

**Owner:** Jack Morgan

**Used:** Within pixel-viewer interactive.

**License:** Used with Jack Morgan's permission (email: [jack.morgan@canterbury.ac.nz](mailto:jack.morgan@canterbury.ac.nz))

mips-assembler, mips-simulator

**Owner:** Alan Hogan

**Used:** Within mips-assembler and mips-simulator interactives.

**Source:** [MIPS Assembler](#) and [MIPS Simulator](#)

**License:** [Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License](#)

## codemirror.css, codemirror.js

**Used:** Within regular expression interactives.

**Source:** [Available on GitHub](#)

**License:**

Copyright (C) 2016 by Marijn Haverbeke [marijnh@gmail.com](mailto:marijnh@gmail.com) and others

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

## en-words.txt

**Owner:** [atebits/Words](#)

**Used:** Within regular-expression-filter interactive.

**License:**

CC0 1.0 Universal

Statement of Purpose

The laws of most jurisdictions throughout the world automatically confer exclusive Copyright and Related Rights (defined below) upon the creator and subsequent owner(s) (each and all, an "owner") of an original work of authorship and/or a database (each, a "Work").

Certain owners wish to permanently relinquish those rights to a Work for the purpose of contributing to a commons of creative, cultural and scientific works ("Commons") that the public can reliably and without fear of later claims of infringement build upon, modify, incorporate in other works, reuse and redistribute as freely as possible in any form whatsoever and for any purposes, including without limitation commercial purposes. These owners may contribute to the Commons to promote the ideal of a free culture and the further production of creative, cultural and scientific works, or to gain reputation or greater distribution for their Work in part through the use and efforts of others.

For these and/or other purposes and motivations, and without any expectation of additional consideration or compensation, the person associating CC0 with a Work (the "Affirmer"), to the extent that he or she is an owner of Copyright and Related Rights in the Work, voluntarily elects to apply CC0 to the Work and publicly distribute the Work under its terms, with knowledge of his or her Copyright and Related Rights in the Work and the meaning and intended legal effect of CC0 on those rights.

1. Copyright and Related Rights. A Work made available under CC0 may be protected by copyright and related or neighboring rights ("Copyright and Related Rights"). Copyright and Related Rights include, but are not limited to, the following:

- i. the right to reproduce, adapt, distribute, perform, display, communicate, and translate a Work;
- ii. moral rights retained by the original author(s) and/or performer(s);
- iii. publicity and privacy rights pertaining to a person's image or likeness depicted in a Work;
- iv. rights protecting against unfair competition in regards to a Work, subject to the limitations in paragraph 4(a), below;
- v. rights protecting the extraction, dissemination, use and reuse of data in a Work;
- vi. database rights (such as those arising under Directive 96/9/EC of the European Parliament and of the Council of 11 March 1996 on the legal protection of databases, and under any national implementation thereof, including any amended or successor version of such directive); and
- vii. other similar, equivalent or corresponding rights throughout the world based on applicable law or treaty, and any national implementations thereof.

2. Waiver. To the greatest extent permitted by, but not in contravention of, applicable law, Affirmer hereby overtly, fully, permanently, irrevocably and unconditionally

waives, abandons, and surrenders all of Affirmer's Copyright and Related Rights and associated claims and causes of action, whether now known or unknown (including existing as well as future claims and causes of action), in the Work (i) in all territories worldwide, (ii) for the maximum duration provided by applicable law or treaty (including future time extensions), (iii) in any current or future medium and for any number of copies, and (iv) for any purpose whatsoever, including without limitation commercial, advertising or promotional purposes (the "Waiver"). Affirmer makes the Waiver for the benefit of each member of the public at large and to the detriment of Affirmer's heirs and successors, fully intending that such Waiver shall not be subject to revocation, rescission, cancellation, termination, or any other legal or equitable action to disrupt the quiet enjoyment of the Work by the public as contemplated by Affirmer's express Statement of Purpose.

3. Public License Fallback. Should any part of the Waiver for any reason be judged legally invalid or ineffective under applicable law, then the Waiver shall be preserved to the maximum extent permitted taking into account Affirmer's express Statement of Purpose. In addition, to the extent the Waiver is so judged Affirmer hereby grants to each affected person a royalty-free, non transferable, non sublicensable, non exclusive, irrevocable and unconditional license to exercise Affirmer's Copyright and Related Rights in the Work (i) in all territories worldwide, (ii) for the maximum duration provided by applicable law or treaty (including future time extensions), (iii) in any current or future medium and for any number of copies, and (iv) for any purpose whatsoever, including without limitation commercial, advertising or promotional purposes (the "License"). The License shall be deemed effective as of the date CC0 was applied by Affirmer to the Work. Should any part of the License for any reason be judged legally invalid or ineffective under applicable law, such partial invalidity or ineffectiveness shall not invalidate the remainder of the License, and in such case Affirmer hereby affirms that he or she will not (i) exercise any of his or her remaining Copyright and Related Rights in the Work or (ii) assert any associated claims and causes of action with respect to the Work, in either case contrary to Affirmer's express Statement of Purpose.

#### 4. Limitations and Disclaimers.

- a. No trademark or patent rights held by Affirmer are waived, abandoned, surrendered, licensed or otherwise affected by this document.
- b. Affirmer offers the Work as-is and makes no representations or warranties of any kind concerning the Work, express, implied, statutory or otherwise, including without limitation warranties of title, merchantability, fitness for a particular purpose, non infringement, or the absence of latent or other defects, accuracy, or the present or

absence of errors, whether or not discoverable, all to the greatest extent permissible under applicable law.

c. Affirmer disclaims responsibility for clearing rights of other persons that may apply to the Work or any use thereof, including without limitation any person's Copyright and Related Rights in the Work. Further, Affirmer disclaims responsibility for obtaining any necessary consents, permissions or other rights required for any use of the Work.

d. Affirmer understands and acknowledges that Creative Commons is not a party to this document and has no duty or obligation with respect to this CC0 or use of the Work.

For more information, please see <http://creativecommons.org/publicdomain/zero/1.0/>

## md5.js

**Owner:** Jeff Mott

**Used:** Within MD5-hash interactive

**License:**

(c) 2009-2013 by Jeff Mott. All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

Redistributions of source code must retain the above copyright notice, this list of conditions, and the following disclaimer. Redistributions in binary form must reproduce the above copyright notice, this list of conditions, and the following disclaimer in the documentation or other materials provided with the distribution. Neither the name CryptoJS nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission. THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS," AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)

ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

## skulpt.min.js

**Website:** <https://github.com/skulpt/skulpt>

**Used:** Within python-interpreter interactive

**License:**

Copyright (c) 2009-2010 Scott Graham

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

Portions of the code were written with the benefit of viewing code that's in the official "CPython" and "pypy" distribution and/or translated from code that's in the official "CPython" and "pypy" distribution. As such, they are:

Copyright (c) 2001, 2002, 2003, 2004, 2005, 2006 Python Software Foundation; All Rights Reserved"

per:

<http://www.python.org/psf/license/>  
<http://www.python.org/download/releases/2.6.2/license/>  
<https://bitbucket.org/pypy/pypy/src/default/LICENSE>

## jsencrypt.js

**Owner:** <https://github.com/travist/jsencrypt>

**Used:** Within rsa-key-generator interactive

**License:**

The MIT License (MIT) Copyright (c) 2013 AllPlayers.com

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

## jquery.js

**Owner:** jQuery

**Used:** Within MD5-hash interactive, pixel-viewer interactive, and delayed-checkbox interactive.

**License:** Available freely under the [MIT license](#)

## Animate.js, Scroller.js, Tiling.js

**Owner:** Zynga Inc. Copyright 2011

**Used:** Within pixel-viewer interactive

**License:** MIT license

## materialize.scss, materialize.min.js

**Owner/Creator:** MaterializeCSS ([Dogfalo](#))

**Used:** As framework (CSS and JS) to create HTML output.

**License:**

The MIT License (MIT)

Copyright (c) 2014-2015 Materialize

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

Roboto-Regular.ttf, Roboto-Regular.woff, Roboto-Regular.woff2, Roboto-Thin.ttf, Roboto-Thin.woff, Roboto-Thin.woff2, Roboto-Light.ttf, Roboto-Light.woff, Roboto-Light.woff2, Roboto-Medium.ttf, Roboto-Medium.woff, Roboto-Medium.woff2, Roboto-Bold.ttf, Roboto-Bold.woff, Roboto-Bold.woff2, Material-Design-Icons.svg, Material-Design-Icons.ttf, Material-Design-Icons.woff2, Material-Design-Icons.eot, Material-Design-Icons.woff

**Owner:** Christian Robertson (commissioned by Google)

**Used:** As font for output.

**License:**

Apache License Version 2.0, January 2004 <http://www.apache.org/licenses/>

## TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

### 1. Definitions.

"License" shall mean the terms and conditions for use, reproduction, and distribution as defined by Sections 1 through 9 of this document.

"Licensor" shall mean the copyright owner or entity authorized by the copyright owner that is granting the License.

"Legal Entity" shall mean the union of the acting entity and all other entities that control, are controlled by, or are under common control with that entity. For the purposes of this definition, "control" means (i) the power, direct or indirect, to cause the direction or management of such entity, whether by contract or otherwise, or (ii) ownership of fifty percent (50%) or more of the outstanding shares, or (iii) beneficial ownership of such entity.

"You" (or "Your") shall mean an individual or Legal Entity exercising permissions granted by this License.

"Source" form shall mean the preferred form for making modifications, including but not limited to software source code, documentation source, and configuration files.

"Object" form shall mean any form resulting from mechanical transformation or translation of a Source form, including but not limited to compiled object code, generated documentation, and conversions to other media types.

"Work" shall mean the work of authorship, whether in Source or Object form, made available under the License, as indicated by a copyright notice that is included in or attached to the work (an example is provided in the Appendix below).

"Derivative Works" shall mean any work, whether in Source or Object form, that is based on (or derived from) the Work and for which the editorial revisions, annotations, elaborations, or other modifications represent, as a whole, an original work of authorship. For the purposes of this License, Derivative Works shall not include works that remain separable from, or merely link (or bind by name) to the interfaces of, the Work and Derivative Works thereof.

"Contribution" shall mean any work of authorship, including the original version of the Work and any modifications or additions to that Work or Derivative Works thereof, that is intentionally submitted to Licensor for inclusion in the Work by the copyright owner or by

an individual or Legal Entity authorized to submit on behalf of the copyright owner. For the purposes of this definition, "submitted" means any form of electronic, verbal, or written communication sent to the Licensor or its representatives, including but not limited to communication on electronic mailing lists, source code control systems, and issue tracking systems that are managed by, or on behalf of, the Licensor for the purpose of discussing and improving the Work, but excluding communication that is conspicuously marked or otherwise designated in writing by the copyright owner as "Not a Contribution."

"Contributor" shall mean Licensor and any individual or Legal Entity on behalf of whom a Contribution has been received by Licensor and subsequently incorporated within the Work.

1. Grant of Copyright License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable copyright license to reproduce, prepare Derivative Works of, publicly display, publicly perform, sublicense, and distribute the Work and such Derivative Works in Source or Object form.
2. Grant of Patent License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable (except as stated in this section) patent license to make, have made, use, offer to sell, sell, import, and otherwise transfer the Work, where such license applies only to those patent claims licensable by such Contributor that are necessarily infringed by their Contribution(s) alone or by combination of their Contribution(s) with the Work to which such Contribution(s) was submitted. If You institute patent litigation against any entity (including a cross-claim or counterclaim in a lawsuit) alleging that the Work or a Contribution incorporated within the Work constitutes direct or contributory patent infringement, then any patent licenses granted to You under this License for that Work shall terminate as of the date such litigation is filed.
3. Redistribution. You may reproduce and distribute copies of the Work or Derivative Works thereof in any medium, with or without modifications, and in Source or Object form, provided that You meet the following conditions:
  - (a) You must give any other recipients of the Work or Derivative Works a copy of this License; and
  - (b) You must cause any modified files to carry prominent notices stating that You changed the files; and

- (c) You must retain, in the Source form of any Derivative Works that You distribute, all copyright, patent, trademark, and attribution notices from the Source form of the Work, excluding those notices that do not pertain to any part of the Derivative Works; and
- (d) If the Work includes a "NOTICE" text file as part of its distribution, then any Derivative Works that You distribute must include a readable copy of the attribution notices contained within such NOTICE file, excluding those notices that do not pertain to any part of the Derivative Works, in at least one of the following places: within a NOTICE text file distributed as part of the Derivative Works; within the Source form or documentation, if provided along with the Derivative Works; or, within a display generated by the Derivative Works, if and wherever such third-party notices normally appear. The contents of the NOTICE file are for informational purposes only and do not modify the License. You may add Your own attribution notices within Derivative Works that You distribute, alongside or as an addendum to the NOTICE text from the Work, provided that such additional attribution notices cannot be construed as modifying the License.

You may add Your own copyright statement to Your modifications and may provide additional or different license terms and conditions for use, reproduction, or distribution of Your modifications, or for any such Derivative Works as a whole, provided Your use, reproduction, and distribution of the Work otherwise complies with the conditions stated in this License.

1. **Submission of Contributions.** Unless You explicitly state otherwise, any Contribution intentionally submitted for inclusion in the Work by You to the Licenser shall be under the terms and conditions of this License, without any additional terms or conditions. Notwithstanding the above, nothing herein shall supersede or modify the terms of any separate license agreement you may have executed with Licenser regarding such Contributions.
2. **Trademarks.** This License does not grant permission to use the trade names, trademarks, service marks, or product names of the Licenser, except as required for reasonable and customary use in describing the origin of the Work and reproducing the content of the NOTICE file.
3. **Disclaimer of Warranty.** Unless required by applicable law or agreed to in writing, Licenser provides the Work (and each Contributor provides its Contributions) on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied, including, without limitation, any warranties or conditions of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A PARTICULAR PURPOSE. You are solely responsible for determining the appropriateness of using or redistributing

the Work and assume any risks associated with Your exercise of permissions under this License.

4. Limitation of Liability. In no event and under no legal theory, whether in tort (including negligence), contract, or otherwise, unless required by applicable law (such as deliberate and grossly negligent acts) or agreed to in writing, shall any Contributor be liable to You for damages, including any direct, indirect, special, incidental, or consequential damages of any character arising as a result of this License or out of the use or inability to use the Work (including but not limited to damages for loss of goodwill, work stoppage, computer failure or malfunction, or any and all other commercial damages or losses), even if such Contributor has been advised of the possibility of such damages.
5. Accepting Warranty or Additional Liability. While redistributing the Work or Derivative Works thereof, You may choose to offer, and charge a fee for, acceptance of support, warranty, indemnity, or other liability obligations and/or rights consistent with this License. However, in accepting such obligations, You may act only on Your own behalf and on Your sole responsibility, not on behalf of any other Contributor, and only if You agree to indemnify, defend, and hold each Contributor harmless for any liability incurred by, or claims asserted against, such Contributor by reason of your accepting any such warranty or additional liability.

## website.css

This file is a combination of `website.scss` and `materialize.scss` created for output. See the details for these files for more details.

## website.scss, website.js

Created by our team for the CSFG project. The Computer Science Field Guide uses a [Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International \(CC BY-NC-SA 4.0\) license](#).

This means you are free to:

- **Share:** copy and redistribute the material in any medium or format
- **Adapt:** remix, transform, and build upon the material

However you must obey the following terms:

- **Attribution:** You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.

- **NonCommercial:** You may not use the material for commercial purposes.
- **ShareAlike:** If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original.

This is a human-readable summary of (and not a substitute for) the [full license](#). This deed highlights only some of the key features and terms of the actual license. It is not a license and has no legal value. You should carefully review all of the terms and conditions of the actual license before using the licensed material.