Computing Without Computers

A Gentle Introduction
to Computer Programming,
Data Structures and Algorithms

Version 0.15

Paul Curzon

February, 2014

Paul Curzon

computer

1 *A person who makes calculations; spec. a person employed for this in an observatory, etc*

The New Shorter Oxford English Dictionary*, OUP, 1993*

Contents

Preface

This book aims to be a *gentle* introduction to the main concepts of computer programming and the related subject of data structures and algorithms. Rather than focussing on particular programming languages that can appear alien and incomprehensible to beginners, it concentrates on the underlying concepts common to a whole range of programming languages. Whatever language you might be learning it should be of use if you are struggling to understand.

It is intended primarily for people with little background in the subject and for those for whom programming appears a little scary. The approach taken is that of understanding by analogy. The idea behind this approach was very clearly captured by Hideki Yukawa: the first Japanese winner of the Nobel Prize for Physics, here quoted from (Wilson 1999).

> *"Suppose there is something which a person cannot understand. He happens to notice the similarity of this something to some other thing which he understands quite well. By comparing them he may come to understand the thing which he could not understand up to that moment."*

He is discussing how scientists come to understand new areas at the frontiers of science. However, the words are just as applicable to those of us following behind and trying to understand things previously discovered by others.

Computer Science text books full of programming fragments can be hard to read. The details of particular languages can obscure the things that are common. It is the general concepts that matter most if a deep understanding of programming is to be obtained. Here I avoid discussing computer examples directly and instead explain the terminology and concepts using a variety of non-computing examples that should be familiar and understandable to all. By understanding how the concepts apply to everyday examples, I hope it will then be easier to follow the more technical details of a formal text book.

Of course analogy has to be treated with care. If pushed too far, the analogy breaks down and we can be left drawing wrong conclusions. By looking at each topic from a variety of different examples and looking at their commonality, I hope that this problem can be at least reduced.

People do not learn just by being told things or reading about them. The fact that I have read a booklet telling me how to juggle does not mean I can then pick up juggling balls and immediately juggle them without dropping them. I can only learn properly by lots of practice. We learn best by actually *doing.* This book also therefore contains lots of puzzles. If your aim in reading this book is to learn about programming you will help yourself achieve this if you actually try the puzzles rather than just reading them. If your aim is to learn how to program you will then need to actually go away and write programs. It is my hope that in reading this book before (or at the same time as) learning about programming more conventionally you will understand more deeply than otherwise.

Part 1: Programs

1. Out of Disorder (Introduction)

Disorder, horror, fear and mutiny shall here inhabit.
William Shakespeare, *Richard II* (1595) Act 4 scene 1, 1.139

Humans hate disorder. We try and organise our lives, our cities, our homes and even sometimes our desks and our bedrooms. Our clothes are hung in neat rows in wardrobes, our cities are connected by networks of roads, and are organised into streets with addresses and post codes that narrow down a house to one of a half dozen or so in the whole country. We put our books on to shelves. We stand in queues at supermarkets. Wherever there is disorder we try and put order.

There are many ways of organising things, and we naturally organise different things in different ways. The organisation we use depends on the things we are trying to organise and often more importantly, what we actually wish to do with them once they are organised. Addresses are organised into postal areas so that it is easier for letters to be delivered. A bookshop organises its novels on the shelves usually alphabetically by author. I organise mine on the bookshelf in my living room so that the books that appear most impressive are in the most visible positions. Stephen Hawkin's *A Brief History of Time* is therefore in the middle with books by John Steinbeck. This is because I am most interested in impressing visitors. Bookshops are most interested in selling books.

The way we organise the things we must manipulate to achieve a task can make a massive difference to how quickly, easily or successfully the task can be done. This can be illustrated by the game of *Spit Not So* (taken from Berlekamp *et al*, 1982). The game is played as follows by two players. Write the nine words: SPIT NOT SO FAT FOP AS IF IN PAN on separate pieces of cards. The cards are placed face up. Each player takes it in turns to take a card. To win you must get all the cards that have the same letter. For example if you had picked up the cards SPIT, IF and IN, you would have all the cards containing the letter I so would win. If all the cards are taken with no one having all of a particular letter then the game is a draw. A game might thus go:

PLAYER 1: SPIT
PLAYER 2: SO
PLAYER 1: FAT
PLAYER 2: NOT
PLAYER 1: FOP
PLAYER 2: IF
PLAYER 1: PAN

At this point Player 1 wins as they have the cards: SPIT, FOP and PAN – all the Ps.

Play a few games to get then idea, then turn the page to see how a suitable organisation will help you play better (or perhaps you can work it out for yourself)!

Before you start to play draw up the following grid, with the words that are on the cards written in the squares shown. The position of each word is important. They have been organised so that there is a single letter in common in each row, column and diagonal. All the F-words run along the bottom, for example.

NOT	IN	PAN
SO	SPIT	AS
FOP	IF	FAT

Keep this hidden from your opponent – otherwise they may see the secret. Play the game exactly as before, except, now each time you pick up a card, put a cross in the square that corresponds to that word. Put a nought in the squares corresponding to the cards your opponent takes. For example, the above game will end with your table looking like the following:

NOT **O**	IN	PAN **X**
SO **O**	SPIT **X**	AS
FOP **X**	IF **O**	FAT **X**

Have you spotted it yet? While your opponent is playing the tricky game of *Spit Not So*, you are just playing *Noughts and Crosses* – which everyone knows how to play well. Just by choosing to arrange the words into a grid with each word in a particular position, you have turned a game that is quite easy to make mistakes in to one that you are unlikely to lose. That is what choosing a suitable organisation, together with an appropriate set of rules to follow, can do for you!

In Paris, after the French Revolution, addresses of buildings were organised by districts. This meant that many buildings were very difficult to locate. Napoleon personally solved the problem at a stroke by suggesting a sensible organisation of addresses. Odd numbers would be on one side of each street and even numbers would be on the other. To get over the problem of knowing which end of the street a number would be he decided that for streets running parallel to the river the numbers should increase in the same direction that the river flowed. For other streets the lowest numbers would be nearest the river (Cronin, 1994). By choosing a good organisation of numbering the problem disappeared. In America they have taken this a step further – all the cities are grids and are labelled with points of the compass.

The subject of **Data Structures and Algorithms** is part of the core of all computing subjects and especially computer programming. The subject is concerned with how information is organised (the data structure used) and the step by step instructions of how that information is manipulated (the algorithms used). Computers have only been in existence since the 1940s. However, the word computer was first used in the mid-17th century, over 300 years earlier. Its original meaning was of a *person* who performed calculations. Algorithms for various tasks have also been known much longer than the existence of computers. The word algorithm was first used in English around 1800 and derives from the name of someone who lived a thousand years ago in the 9th century: a Persian "computer" called *Abu Ja'far Mohammed ibn Musa al-Khowarizmi*. "Algorithm" is derived from his last name and a Latin word (he also gave his name to the word algebra).

In the Second World War, one of the challenges for each side was to try to crack the codes used by the opposition. This became urgent for the British as the German U-boat fleet was decimating British convoys in the Atlantic. The Admiralty desperately needed to know where the U-boats were, and this could only be achieved by intercepting their communications. In Britain the code-breaking work was done at Bletchley Park, where originally teams of people worked to decipher messages. Many did not understand the purpose of their work, as they only worked on small parts and did not see the bigger picture. They just did the calculations needed by blindly following the instructions they were given. The instructions being followed were devised by a team of Mathematicians led by Alan Turing: the original human computer programmers. They of course had to understand what was going on to devise the instructions. However, the mass of communications that were being intercepted meant that the teams of people could not work fast enough. Gradually machines were designed that could do some of the routine parts. These machines were the precursors to the modern computer (Hodges, 1985).

Machine-based computers thus just took over the things that human-based computers had previously done. Their speed and accuracy mean they can tackle bigger problems and do them faster, but the essence of what they do remains the same. Computer programs are just step-by-step instructions for doing a given task, written in computer programming languages. Human computers were also given programs to follow. The instructions were just given in a language the humans could follow. When you do a long multiplication you are doing exactly this, following step-by-step rules you wrote as a child. These step-by-step instructions are algorithms. An algorithm consists of the actions to do and the order to do them in. Algorithms occur all over the place – not just the instructions you learnt about how to do long multiplication, but also in the instructions of how to put together a DIY bookcase, and even a set of instructions of how to solve a Rubic's Cube are all algorithms. In fact there are a whole series of puzzles that involve devising algorithms to solve them of which a Rubic's cube is one of the most complex.

I am walking to the shops, when a car pulls up beside me. The driver winds down the window. "Excuse me, how do I get to Kirkcroft Lane?" I answer by giving an algorithm:

> "Go down the hill.
> Take the first major turning on the left.
> Go up the hill.
> Take the first turning on the left. (It is just after the Navigation pub).
> You are there."

What makes that an algorithm? It is a set of **instructions of how to** do something. More specifically, it is a sequence of actions to follow and the order they must be done. If they are attempted in the wrong order (for example going up the hill first instead of down) then the person will not end up in the right place. Computer programs are just sets of instructions to be followed, written in a way that makes it easy for a computer to follow them.

Not all lists of rules are algorithms. For example, go to any park and there is likely to be a notice board by the gate with a set of rules:

> Keep to the paths.
> Do not walk on the grass.
> No bicycles, skateboards or roller blades.
> The park closes at sunset.
> No ball games.

This is **not** an algorithm even though it is a list of rules. Neither is the list of rules on the door of the toilets in an airliner:

> No smoking
> No eating
> Do not use when the fasten seat belt sigh is lit.

The rules of algorithms tell you **how** to do something step by step. These rules tell you **what** you can and cannot do. Algorithms are about **how** not **what**. The above rules could also be read in any order without making any difference to the reader's understanding. In an algorithm the order matters. Other rules on an airliner do correspond to algorithms. For example, on the emergency exit door is the following:

> Emergency opening
> 1. Pull cover aside
> 2. Push lever to open position and release.
> 3. Push door outwards.

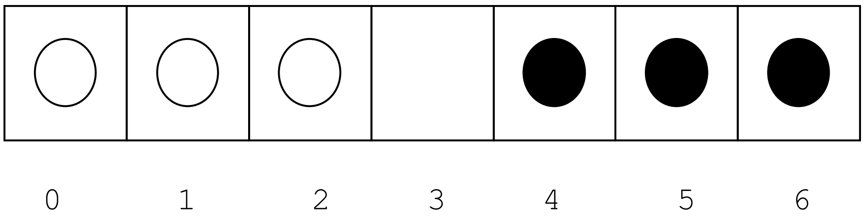
These instructions are about **how** to escape. In the event of an emergency I do not want a list of rules that just say what I am and am not allowed to do, I want some instructions that tell me how to get out of the plane in the quickest way. I want the instructions to be clear, unambiguous and be very, very precise. I want there to be no opportunity for me to be confused about what I am expected to do, or the order I am expected to do it. I want an algorithm. Informal algorithms found in everyday life often come numbered in this way. The numbered steps are there specifically to tell you the **order** to do things: order is very important in algorithms.

Frequently in the instructions we encounter in real life "how" rules are mixed up with "what" rules which can make it confusing working out whether some set of rules really are an algorithm or not. We will return to this later.

The individual steps in an algorithm consist of actions. They tell you to do something: "Pull", "Push", "Go", "Take", "Move". These are all verbs: doing words. The most basic doing words available to computers are to copy something from one place and store the copy somewhere else and to do calculations: addition and subtraction for example. However, by doing combinations of such simple actions we (or rather computers) can do much more interesting and useful things.

The instructions in our example above are more than just verbs, however. They tell you what thing, what object to apply the operation to. Pull what? – the "cover". Push what? – the "lever". These are nouns. As we shall see whether you think in terms of the verbs: the doing words; or the nouns: the things completely changes your view of what a program is and in doing so can make the program far easier to write.

The following puzzle is adapted from Kordemsky, 1975, puzzle 94. A playing board consists of 7 squares in a row numbered from 0 to 6. Initially, three white pieces are placed in positions 0,1 and 2. Three black pieces are positioned in positions 4, 5 and 6. Square 3 is empty. Pieces can move either by sliding into an adjacent empty square, or by jumping a single adjacent piece into the empty square beyond. The aim is swap the positions of the black and white pieces.



To solve this puzzle you must come up with an algorithm to do this (ie write down a sequence of instructions that if followed would result in the black and white pieces being swapped). The easiest way is just to write all the moves you make down as you make them (even moves that undo earlier mistakes). Use coins for pieces and experiment. For example, a possible first two moves might be:

1. Move the piece in square 2 to square 3.
2. Jump the piece in square 4 to square 2.

Given your algorithm anyone can now solve the puzzle, without doing any problem solving of their own – they just have to follow your instructions.

Now you should evaluate your solution to the problem. How many moves long is your algorithm? There are often very many different algorithms that can be used to solve a given problem, some faster than others. Is your algorithm the fastest? Try and improve on it – there is an efficient algorithm (one version of which is given below, though try and improve on your own first) for solving the puzzle in 15 moves. Perhaps you noticed some kinds of moves that were bad in the sense of not making progress. You should try and avoid getting into that kind of position. Other sequences of move were perhaps really good in that they made lots of progress - you want to try and engineer being in those positions.

A 15-step algorithm for solving the puzzle is as follows:

1. Move the piece in square 2 to square 3.
2. Jump the piece in square 4 to square 2.
3. Move the piece in square 5 to square 4.
4. Jump the piece in square 3 to square 5.
5. Jump the piece in square 1 to square 3.
6. Move the piece in square 0 to square 1.
7. Jump the piece in square 2 to square 0.
8. Jump the piece in square 4 to square 2.
9. Jump the piece in square 6 to square 4.

10. Move the piece in square 5 to square 6.
11. Jump the piece in square 3 to square 5.
12. Jump the piece in square 1 to square 3.
13. Move the piece in square 2 to square 1.
14. Jump the piece in square 4 to square 2.
15. Move the piece in square 3 to square 4.

We thus do not need to talk about computers or computer programs to explore the world of data structures and algorithms. We are all computers in the sense that we do many tasks by following step-by-step rules. The way such rules are written often mirrors the ways algorithms must be written so that they can be followed by a computer rather than by a human. We also organise both information and more solid things in ways that make it easier for us to do particular tasks. We even turn some of these things in to games.

The aim of this book is to introduce you to programming concepts and to a range of data structures and algorithms that are important in computer programming by relating them to equivalent things from everyday life. It is hoped that this will help you understand the underlying concepts and so make it easier for you to understand more technical issues to do with the algorithms, including evaluating their efficiency and turning them into computer programs. This book is thus intended to be used together with a more formal textbook on the subject. Read this booklet first and if you find it interesting you will find a normal textbook much more accessible.

2. The Language Instinct (Programming Languages)

The communication of the dead is tongued with fire
beyond the language of the living.
T.S. Eliot, Four Quartets 'Little Gidding', (1940)

Ambiguity
Algorithms are just instructions used to describe actions. In this booklet all the algorithms are described in English. However this can lead to problems. English (as is any other Human Language) is a very difficult language to be precise in, and the whole point of an algorithm is that it tells you precisely what to do. One of the problems is that human "natural" languages are **ambiguous**. By this we mean that a given sentence can mean several different things. My favourite is one pointed out by horror writer Stephen King. He quoted a passage from a thriller he had read:

"His eyes slid down the front of her dress".

As he pointed out, if he had written that in one of his horror books, it would have been describing something totally different to the lecherous thing that the thriller writer was trying to suggest. There are many other examples where a sentence can mean several different things. Sometimes the differences are subtle. For example
"He picked up the red pen"
can mean something slightly different depending on the context and where the emphasis is placed.
"*He* picked up the red pen"
might be emphasising it was him who did it not her.
"He *picked up* the red pen"
might be trying to suggest that he did not leave it alone.
"He picked up the *red* pen"
might be pointing out that he picked up the red pen, leaving the blue and black ones.
"He picked up the red *pen*"
might be pointing out he picked up the pen leaving the pencil behind. The point of these examples is that it is easy to misunderstand precisely what is being said in English.

This fact has been known and exploited by lawyers for centuries: it is how many earn their living drafting laws and contracts to try to remove ambiguity. What do they do to minimise the problems? They resort to using very stylised language. They give definitions of the meaning words at the start. Essentially what they are doing is restricting the language to get rid of the ambiguity.

Despite all the trouble the first set of lawyers go to, to remove ambiguity, a whole other set of lawyers earn their money by arguing over the meaning of those same laws and contracts. Take insurance contracts for example. Frequently people discover they are not covered for something they thought they were. When I was a student I lived in a student residence and took out an insurance policy in case my possessions were stolen. When I took it out I asked the salesman if my records would be covered and was assured they would be. On reading the small print I noticed it seemed to suggest "collections" were not covered – and that there was an additional policy for DJs to cover such record collections. So was my record "collection" covered? That depends on what they meant by "collection". I had about a hundred LPs at the time, but they were not a "collection" in the sense that there were no rarities amongst them. In the end the insurance broker decided they would not be covered, so I cancelled the policy and found another one that did not mention "collections".

Whilst at University I also caught Glandular Fever. Just over a year later, whilst in Canada, I ended up in Hospital connected to drips, again with Glandular Fever. Every morning I had an accountant standing at the bottom of my bed asking how I was going to pay the thousand pound hospital bill. Luckily I had travel insurance but it contained a clause stating that I was "not covered for any illness I had suffered within a year of the claim". Was I covered? That depends on when the year ran from – from when I was first diagnosed as having Glandular Fever or from when I recovered? The accountant was not convinced the insurance company would pay so wanted me to pay before I was treated.

Both the above problems arise from the ambiguity in the meaning of statements. When claims go to court lawyers from one side will be arguing that the contract means one thing. Lawyers on the other side will be arguing that it means the opposite.

Many jokes rely totally on there being different meanings to the same words or phrase (especially the really cringe-worthy ones children love to tell).

> *Why did Cinderella not get selected for the Soccer team?*
> *Because she ran away from the Ball.*
>
> *A man enters a restaurant and says "do you serve Oysters here?"*
> *"Why certainly sir, we always serve Oysters first" replied the waiter.*
> *"There you see", said the man, pulling an Oyster out of his pocket and putting it on the chair, "You sit there and they will serve you first".*

Why are they supposed to be funny? The first relies on two meanings of the word "ball" and the second on subtly different uses of the word "serve". Telling a joke of this kind in a language designed for writing algorithms such as a programming language ought to be very difficult as such languages are designed so as not to have any ambiguity in meaning.

The rules of games can also suffer from being ambiguous. There are several games that when we play them we have to decide whose rules we are playing: my family read the rules as meaning one thing. My wife's family grew up believing they meant something else. Before we can play we must remove the ambiguity, as otherwise it will end in argument. We must decide which rules we are going to play: the "Sheffield rules" or the "London rules".

Recipes are similar to algorithms. My hope is that by following a recipe I will end up with something that looks like the picture in the recipe book. That rarely happens. This is in part due to my inability to follow instructions, but is also due to the vagueness of the instructions. "Stir until the sauce thickens": how thick? Is this thickened or not? "Fry until golden brown": was that golden brown or yellow? Oh it seems to be black now. "Add a tablespoon of sugar". Would that be a level tablespoon or a heaped one, and if heaped, how heaped? Again the ambiguity of English is a problem in giving precise instructions that will be interpreted by everyone in exactly the same way.

Programming Languages

When giving computers instructions they need us to be precise. We cannot use a language for which there is any ambiguity. Computers need to be told exactly what to do at every point. That is why programming languages were devised. There are many different programming languages in existence, just like the fact that there are many different human languages. As with human languages some are similar, as they have evolved from the same original language. Many European languages have evolved from Latin – so having learnt Italian, Spanish is relatively easy to learn. The computer languages, Java and C++ similarly have much in common, so having learnt one the other is relatively simple to learn. Others like Russian have a completely different history, so are much harder to learn. Similarly, having learnt Java would only be of limited help in learning the Prolog programming language, as it is from a completely different family.

Languages can exist as several dialects. The English and Americans both speak "English" but there are differences between the two. For example, "I went out wearing only my pants" would mean something different to an American and to a Brit. In England "pants" are underwear, whereas in America the word just means "trousers". Similarly there are programming languages that though essentially being the same have minor differences: with some things meaning slightly different things, or with extra words. For example, there are several variations of the Java language.

To avoid the problems this causes many languages are standardised. A committee declares exactly what is allowed in the language and precisely what each construct means. Such languages can change over time, but only in ways permitted by the committee. The French try and do the same thing with the French language. The *Academie Francaise* rules on additions, with the aim of keeping out words like "le Hotdog" and ensure there is only one French language. As I write a law has been passed in Poland making it illegal to use non-Polish words and phrases such as "sex shop" and "supermarket" to protect the language (Connolly 2000). Because a language has been standardised does not mean it is set in stone. It just means it only changes gradually in ways that have been vetted as sensible and in a way that can ensure everyone changes at the same time.

Most (European at least) languages are based on alphabets. For example, English uses an alphabet of 26 letters (A-Z). All English words are made up of letters from that alphabet. There is no reason that a language has to use that particular alphabet though. Indeed many languages do not. Russian for example uses a totally different set of letters (33 in all): a different alphabet. Some of the letters like A are familiar to English speakers. Others like Я and Ю are completely different. Greek also uses a different alphabet with letters like α, β, γ, δ. Computer languages generally use similar alphabets to English, but allow numbers and other symbols like '*' to be used as extra letters.

There are several different kinds of computer languages. We will consider two: **machine code** and **high-level languages**. Computers need instructions to be converted into machine-code languages if they are to be followed, as machine code is all computers really understand. These are very simple languages that use the alphabet that computer hardware works with. They have only two letters: 0 and 1. All words in machine code languages are therefore made from sequences of 0s and 1s. For example, 00000100 might be one word, whereas 10101010 might be another meaning something different. Of course not all sequences of 0s and 1s might be a word that means anything. This is just like the fact that *quelle* is not a word in English – though it is a word in French. Similarly the same word might mean different things in different languages. In human languages these are called *false friends.* A famous example of someone making this kind of mistake is when the then US President John F. Kennedy announced in a speech in West Berlin in 1963 "*Ich bin ein Berliner*" intending to say "*I am a Berliner*". Unfortunately in German *ein Berliner* is a doughnut not a person from Berlin, so he had actually said "*I am a doughnut*". The applause that resulted was not for the reason he thought! Different computer languages also often use the same words with very slightly different meanings. For example the word "for" in the language Pascal means something slightly different in some situations to the same word in the language Java.

Compilers
Humans find machine-code languages very hard to read and write. High-level languages aim to overcome this problem. They use more familiar alphabets and have words similar to those in English to help humans understand them. This however leaves us with a problem. If a computer can only follow instructions if they are written in a machine-code language of 0s and 1s, what use is a high level language to it. This is where compilers come in.

A programming language **compiler** is just a form of translator. Just like a human translator it takes a document written in one language and writes out a copy of it in another language. Different compilers can translate between different languages just as human translators often can only translate between two languages. A difference is that compilers are not capable of translating documents back again. It is as though someone could translate from French to English, but if given an English document would not be able to translate it to French. I am a little like that. I know enough French that I can get the general idea of what a French book is saying, but I do not remember French well enough to write French. Compilers take that to an extreme. They can translate perfectly from one language to the other, but not at all in the other direction. In particular, a compiler translates from a high-level language to the machine-code language of the computer that is to follow the instructions. The compiler reads the high level program and writes out a new version of it in 0's and 1's as a separate document. The computer then reads and follows the instructions in this document rather than the high level version originally written. Executing a newly written program is thus done in two stages. First it is compiled (ie translated). This need only be done once as you can then just keep using the translated version rather than needing to retranslate it every time you wish to use it. **Compile-time** refers to the point when this translation and error checking is done. Once compiled, the translated instructions can be followed: that point is referred to as **run-time**.

In the Discworld books by Terry Pratchet, the non-magical technology works by Imp power (Pratchett, 1983). For example, cameras are a box with an Imp with a paint box and easel. The Imp paints whatever it sees out of the camera when told to – the result is a Discworld photograph. Other technology works in a similar way. The only computer works in a different way, however – using ants and possibly due to being developed by Wizards has the whiff of magic about it. A Discworld computer that works using Imps would be possible and no doubt will eventually be invented on the Discworld. All it needs is a large number of imps each capable of doing specific simple tasks – following instructions given. Of course, humans could be used instead of Imps but then the box containing it would be too big to fit on a desk top. In fact a whole building would be needed. In essence this was the first meaning of the word computer as we dicussed in the introduction – people doing computation.

How would an Imp computer work in practice? A program written in some appropriate programming language would first be compiled. For an Imp computer, this would involve checking that the instructions made sense, translating them into Imp language and giving one instruction to each Imp. A special set of bilingual Imps could perhaps do this. Some mechanism would also be needed to ensure that the Imps knew when it was their turn to follow their instruction. Otherwise, the actions of the algorithm might be performed in the wrong order. Storage space (such as boxes) would be needed for them to store results of calculations. We will return to the design of this Imp computer in subsequent chapters as we discuss these issues in detail.

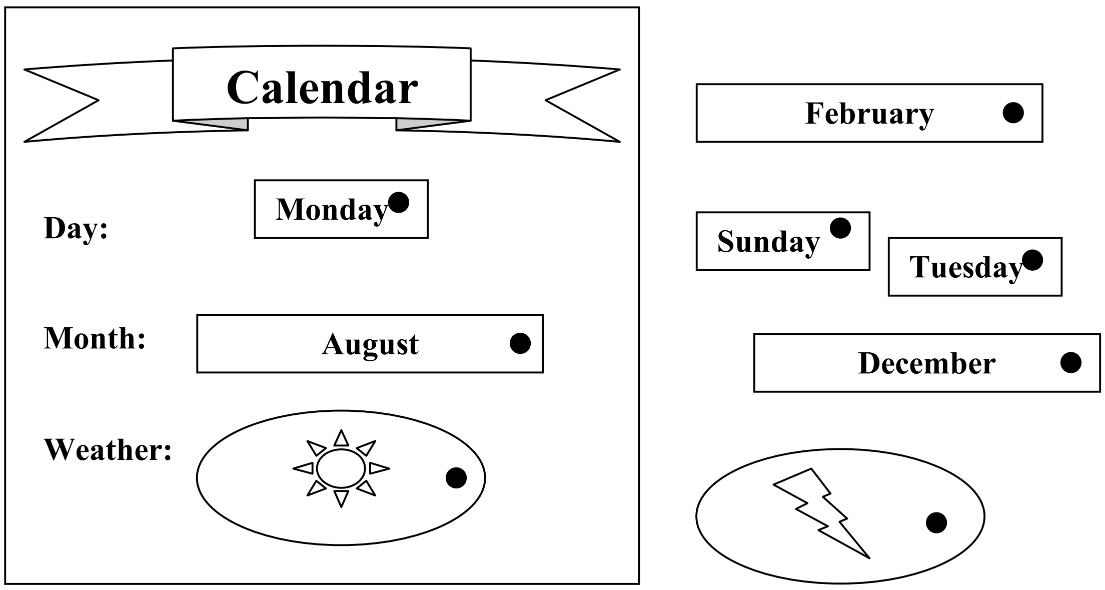
Compilers take a whole program and translate it completely before executing it. Suppose I had a set of directions written in Arabic which told me how to get to the market. As I cannot (yet) read Arabic I cannot follow the instructions unless I have them translated. With a compiler-approach I would give the instructions to someone who could read Arabic. They would give me back a copy of the instructions written in English. Armed with this new document I would be able to follow the instructions and get to the market. Some programming languages use **interpreters** rather than compilers. An interpreter effectively translates the program as it is executed: instructions are translated and immediately followed before moving to the next instruction. It is as though I took the person who could read Arabic with me to the market. They would translate the instructions one at a time. At each junction we came to they would translate the instruction and tell me which way to turn. Unlike with a compiler, with an interpreter I never receive a copy of the whole set of instructions in English. If on the following day I wanted to go to the market again using the same instructions I would need to take the translator with me again who would go through the same process all over again. With the compilation approach, I only do the translation once. I then have a copy of the instructions in a language I understand that I can use over and over again. Such a translated set of instructions is in computing terms **object code**. Similarly, once a program has been compiled, the object code can repeatedly be executed without any further need for the original program to be compiled again.

Syntax and Compile-time Errors

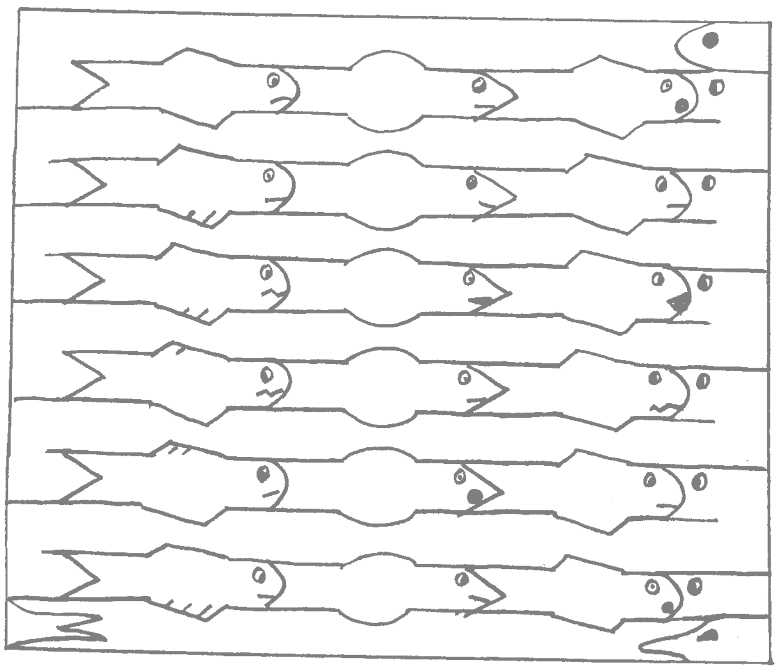
A computer language is defined by two things: its **syntax** and its **semantics**. By **syntax** we mean which combinations of letters are allowed as words (i.e. how to spell in the language) and what order those words can be put in to make meaningful sentences (i.e. how to write grammatically correct). By **semantics**, we mean what those words and sentences actually mean. As computer languages are languages for giving instructions in, the semantics of a computer language tells you what each word tells you to do. As we discussed above, for computers, it is important that it is always possible to tell exactly what each thing means with no ambiguity.

This leads to a second thing a compiler does. It looks out for mistakes. If I gave a human translator the “sentence”: “The cat sat on the matrs” to translate into French they would tell me that “matrs” is not a word in English so they cannot translate it. I made a mistake in the English that I need to correct before the sentence is translated. Similarly they would probably point out if I asked them to translate “The cats sat on there mats” that using “there” was grammatically incorrect in English. They are pointing out spelling and grammar errors. These are **syntax errors**: errors that mean what I have written is not English so cannot be translated until it has been corrected in the English version. Compilers will also point out syntax errors: spelling and grammar errors in programming languages and refuse to translate the program until it has been corrected. Students often complain that the compiler ought to correct such errors and carry on the translation rather than just giving up. Similarly one might expect the human translator above to be able to work out what I meant and do the translation. However, that could be dangerous. What if they mistakenly assumed I head meant the first sentence to be “The cat sat on the mat” and translated that. In fact I meant to write “The cat sat on the mattress”. Compilers cannot read the minds of the person who wrote the program. It is therefore safer for them to just point out the error rather than trying to correct it. Syntax errors are **compile-time errors**. They are found during the translation process.

The way the compiler treats a program is a little like a special kind of jigsaw puzzle. The program is split into what are known as tokens – the separate pieces of the puzzle. Some pieces have identical shapes and can be interchanged. Some simple toddler jigsaws are like that – for example each piece is the carriage of a train containing a different letter of the alphabet. They fit together in any order, except the engine has to be on the front as it only fits to a piece behind. A more complicated example is a “Jigsaw Calendar”. Imagine one of the wooden jigsaws for young babies – the sort that have pieces with handles that fit into slots on a wooden board. A Jigsaw calendar is similar except that several different pieces fit into each slot. One day holds the day of the week, another the month and another perhaps for the weather – where pictures of the sun might fit. However the pieces are such that only day pieces fit into the day slot, only month pieces will fit in the month slot and only weather pieces in the weather slot. The pieces are the tokens. Many different versions of the calender can be created using the different pieces, but they all are syntactically correct. The shapes physically enforce a syntax. Checking for syntax errors involves checking that the pieces really do fit.



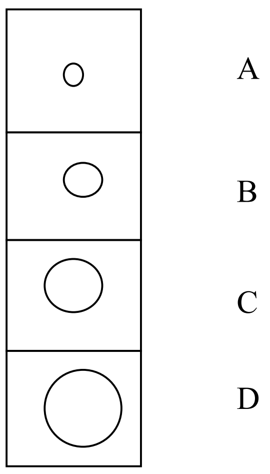
Here is another more complex jigsaw puzzle made of fishy shapes. Each of the pieces is a fish or an eel. Most of the fish pieces are the same shape. The middle fish each have round fins and are identical to each other in shape. The pointed-fin fish are also identical to each other in shape. Each fish has a slightly different expression. There are thus many ways to create a jigsaw that fits. However, only one combination is the one below.



Such jigsaws are such that as long as the pieces fit, you get a sensible picture (though not necessarily the one on the box, or one that corresponds to today's date in the case of the calendar). Syntax errors are like the situation when someone tries to force a piece into a position it does not fit. Even if you have used all the pieces or filled all the gaps, if any of the pieces do not properly fit then you have solved the jigsaw puzzle. If you have written a program and the tokens have been put in places they do not fit, then you do not have a real program. The situation when all the pieces do fit, but you have a different picture to that on the box is like a run time error. You cannot detect it by looking at the way the pieces and whether they fit. It is after all a perfectly good picture. It is only wrong with respect to the picture you were trying to create. Unless you know what that is you cannot say the jigsaw is wrong. Similarly with a program, if the tokens do fit, then it is a program that does something. However, the compiler cannot tell whether it does what it is supposed to do because it does not know what that is.

What are the tokens for a language (human or computer)? They are the different kinds of words. In human languages like English, the tokens are things like nouns (dog, rock, hat, etc) and verbs (eat, run, etc). In English nouns and verbs fit into different positions in a sentence – they have different "shapes". Get the words in places where they fit and you get a sentence, though one that might be nonsense. In computer languages, the tokens are not nouns and verbs but "identifiers", "keywords" etc. We will look at the different kinds of tokens in later chapters. For now we will use a puzzle to give you the idea.

Consider the following watery puzzle (variations of which we will return to in a later chapter). You have four counters representing bubbles of different sizes placed on a board in the order shown with the smallest bubble at the top and the largest at the bottom.



The aim of the puzzle is to get the counters so that the one with the biggest bubble is at the top, that with the next biggest bubble comes next and so on. This must be done in as few steps as possible. The bigger the bubble, the nearer to the top (the surface) that it bubbles. However this can only be done be swapping pairs of adjacent bubbles at a time (which counts as a turn). For example, on the first turn you might swap, the counters on squares A and B. The second turn might be to swap C and D and so on. The hard (well not that hard!) part of the puzzle is that you only have 6 turns.

Now lets add a twist to the puzzle. Not only must you solve the puzzle, but you must write down the set of instructions that will be followed – you must write down an algorithm that always solves the puzzle in 6 turns. Moves are written in the form

____ is swapped with ____ .

In the blanks you must write letters A, B, C or D. For example, the first two moves of the algorithm might be:

A is swapped with B.
B is swapped with C.

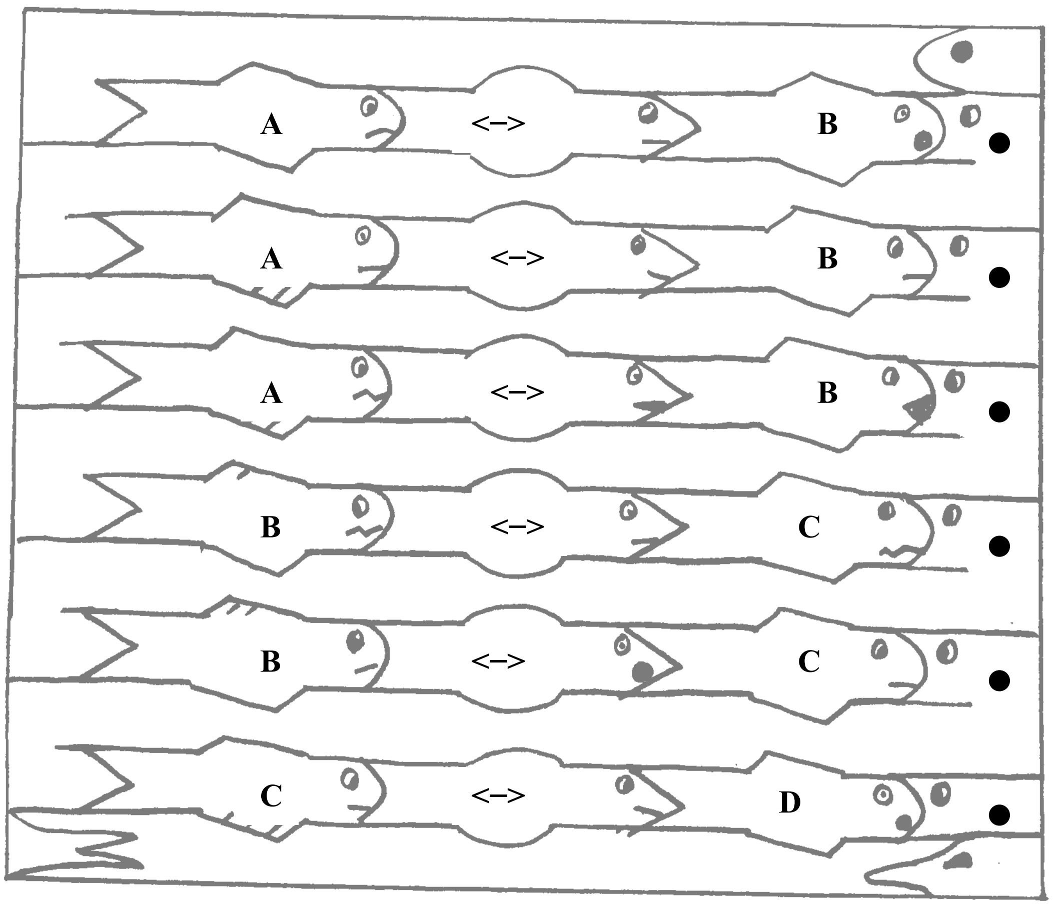
By "is swapped with" in an instruction we mean exchange the positions of the bubbles on the two counters on those squares – but only if they are in the wrong order – otherwise leave them alone.

What we have created is a language for writing algorithms. A program in this language consists of 6 "swap" statements. Our language has a syntax – we have said what each statement must look like – what words (ie tokens) can be used. The possible tokens are "is swapped with", "." and the letters labelling the squares on the board: "A", "B", "C" and "D". We have also specified the positions they can appear in: similar to the way the grammar of English specifies where verbs and nouns can appear. Notice that our language is not English though it looks a little like it. We are much more restrictive in what we allow to be written. In programming languages, symbols are sometimes used to mean operations. For example, in the above example the language might use <–> as an abbreviation for "is swapped with". The two instructions given above would be written is follows in this new language.

A <–> B.
B <–> C.

Now the tokens are the letters A, B, C and D and the symbols "." and "<–>".

The following may help you understand what we meant earlier that program tokens are like pieces in a jigsaw. Below are the pieces of a jigsaw similar to the one we looked at above. The shapes are the same but we have associated a different token with each shape. The board positions appear on identical shapes. The puzzle now is to construct a jigsaw that gives instructions that solve the original puzzle. You cannot change the pieces or what is on them, just move them around. They must of course only be put in positions that they fit. Any finished version of the jigsaw will be free of syntax errors in our language. It will therefore give a sensible sequence of instructions that can be followed in that it will tell you to swap the counters on two squares of the board. However, it may still contain errors that are not syntax errors: following the instructions could still do the wrong thing leaving us with the bubbles of the original puzzle in the wrong places. The version of the jigsaw below is like this – it is syntactically correct – the pieces fit, but following the instructions as in the jigsaw would not always solve the bubble puzzle. Try and rearrange the jigsaw so that the instructions it gives do always solve the puzzle.



Notice that the shapes of our pieces do not completely guarantee that the instructions follow the rules of the puzzle. In particular, it does not guarantee that the squares to be swapped are adjacent. We would need more complicated pieces (i.e. more complicated syntax rules) to prevent this. As it stands such errors cannot be detected just by checking that the pieces fit. With changes to the language they could be made into syntax errors. This kind of situation occurs in programming languages too – problems that can only be detected in one language when the program is run can be detected at compile-time in another language. We look at these other kinds of errors in the next section.

Semantic and Run-time Errors
Syntax errors are thus situations where what is written is incorrect use of the language concerned. What is written is just not English, or it is just not Java, or whatever. There are other kinds of errors that can be made when writing in a language: what is written is perfectly good English, for example. It just does not mean what the person who wrote it meant. Kennedy could have been trying to say "I am a doughnut", when he wrote "Ich bin ein Berliner" so someone checking his speech for spelling and grammar errors would not point out the problem. This is a **semantic error**. The error is in the meaning of the sentence and not in its spelling or grammar. The sentence "I was born on the 30th February" similarly contains a semantic error. The spelling and grammar is fine. It is perfectly good English. However when you consider its meaning, it makes no sense as there is no 30th February. It is a meaningless sentence.

For algorithms and computer programs, semantic errors are errors where the instructions are perfectly good instructions but where following them results in the wrong or unintended thing happening. Suppose that on being asked the directions to Kirkcroft Lane as in the previous chapter I gave the following instructions:

> “Go down the hill.
> **Turn left at the Co-op.**
> Go up the hill.
> Take the first turning on the left. (It is just after the Navigation pub).
> You are there.”

These instructions would seem perfectly sensible to the driver as they were writing them down. Suppose, however, that there was no Co-op (I had forgotten – it was knocked down last year and replaced by a car park.) The driver would keep driving, until eventually they realised they were lost again. At that point they would give up trying to follow my instructions. In the worst case the instructions might lead them, if followed to drive over a cliff! The algorithm has “**crashed**”. A **catastrophic run-time error** has occurred. My instructions were meaningless. They contain a semantic error. The driver could not tell this until they started to follow the instructions – so the error is a run-time error.

Semantic errors do not need to be catastrophic. They can just lead to the wrong thing happening. For example, suppose I gave the following instructions to the driver, getting my left and right mixed up on the first turning:

> “Go down the hill.
> **Take the first turning on the right.**
> Take the first turning on the left.
> You are there.”

Assume the road did have a right turn at some point, from which there was a left turn. Not only would these instructions appear to be sensible, when the driver was writing them down, the driver could follow them without noticing anything wrong. They would turn into the final road, believing themselves to be at their destination. However, because my instructions contained a semantic error, they would be in the wrong place. Unlike the last example, the algorithm did not crash as this time a result was obtained – just the wrong result.

To take a different example, suppose the rulebook for a football tournament included the rule:

> “In the event of a draw at the end of extra time, the team that scored last will go through to the next round”.

That is a perfectly good rule in the sense that it could be followed, but it almost certainly contains a semantic error. The person writing it probably meant to write:

> “In the event of a draw at the end of extra time, the team that scored first will go through to the next round”.

As the compiler cannot spot such errors, they can remain undetected even when the instructions are followed. Notice also that such an error does not necessarily mean the wrong thing is *always* done. If no matches ended up in a draw after extra time, the above rule would never be followed, so the correct team would always go through to the next round. The error would just sit there in the rulebook in an instruction that was never read.

When writing computer programs that kind of error is very easy to make. The Y2K bug is perhaps the most famous. The instructions of many computer programs used 2 digits for the year: 63 meaning 1963 for example. For decades such programs worked perfectly well. It was only when we reached the year 2000 that the problems happened as some computers treated 00 as meaning the year 1900 instead of the year 2000 as intended. One of the Mars probes crashed on Mars due to a semantic error in its instructions about how to land. The problem there was that some of the programmers wrote instructions that calculated in metric units, whereas others assumed the same numbers were in imperial units. Semantic errors are **run-time errors**. They cannot be found during the translation. Instead they are found by following the instructions and determining what happens.

Underlying a language is the concepts it is being used to express. In English if we wish to express that a particular person (say Emma) is doing a specific thing (say eating), there are two basic concepts that the language must allow us to express. It must allow us to identify a specific person (Emma) and also identify what she is doing to it (eating). Consider an imaginary caveman language that did not have a concept of people though did have concepts of actions. The statement that Emma is eating cannot be expressed in such a language – the nearest the caveman could say would be "eat". Who was eating would need to be expressed in some other way than using the language (by pointing at Emma) perhaps. A third concept you might need would also be of time. A film I saw as a teenager called Caveman only had dialogue in such an invented language. As you went into the cinema you were given a leaflet listing all the words used – a dozen or so. From what I remember it had words for food, sex, going to the toilet, you me – just the most important to a caveman. It did not have tenses however. You could therefore say "I eat" but that could mean both I am eating and I have eaten. Not all things can be expressed in all languages. If we want a language for expressing algorithms, what we need to do is work out what basic concepts we need to be able to express and then provide ways of expressing them in the language. The basic thing as we have said that we are trying to express in an algorithm is a series of actions and the order they will occur in. We must work out what kinds of actions we wish to perform and work out the components. For example, we wish to be able to move data from one place to another. We must consider what information we must be able to express if we are to specify something is being moved – as we will discuss later in this case we must be able to indicate what is being moved and where it is to be moved to. In the subsequent chapters we will look at a series of different kinds of concept we need to express in algorithms, breaking them down into the basic parts. Any programming language that uses that concept must have a way of expressing those basic parts. Incidentally, I hasten to add that Caveman was not a particularly good film, before you rush out to order the video.

Completeness

A set of instructions or algorithm is said to be **complete** if it covers all eventualities. It must tell you what to do whatever the situation. This is *not* quite the same as meaning "finished" in the sense of you having finished writing the instructions. For example, suppose I am asked by my wife to go to the bakers to buy her Chocolate Muffins as we have nothing for Breakfast. If that is all the instruction I have been given then I could have problems, as it does not cover all eventualities. The instructions:

1. Go to shop

2. Buy Chocolate muffins
3. Return home

do not tell me what to do if the shop does not have that cereal. The instructions are not complete. They do not tell me what to do in all circumstances. If the shop does not have chocolate muffins (which happens all too often). I can no longer do the task by following instructions. I must instead use my imagination and guess what the appropriate thing to do is (and get into trouble if I return home with the wrong thing). How do we make the instructions complete? We provide instructions for each eventuality that might arise.

```
1. Go to shop
2. If the shop has Chocolate Muffins
           then buy Chocolate Muffins
   else if the shop has Croissants
           then buy Croissants
   else buy nothing.
3. Return back home
```

The important thing for completeness is to have the final "catch-all" instruction: "else buy nothing". This tells me what to do if none of the situations specified actually happen: since we cannot easily list all possibilities. It means that whatever the shop stocks or does not stock, I will have an instruction telling me what to do: buy nothing.

Are the new instructions complete? What if the shop I go to is shut? Should I just return or go to another shop? The instructions again do not tell me. Providing instructions for all situations is near impossible in many real life situations. The way round this is to have a last default case that says for example "Give up" if none of the other rules apply.

An area where having complete rules is often important is in sports competitions. As I am writing this, England have just been knocked out of the Euro 2000 football tournament after losing to Romania due to a last minute penalty. Before the match most of the papers included rules for working out who would go through with Portugal whatever the result of the matches between Romania and England and Germany and Portugal. For example, the Guardian (Guardian 2000) explained it as follows:

> *England will qualify for the quarter-finals with Portugal if they win or draw with Romania.*
>
> *Defeat would put England out...and would mean Romania progressing if Germany fail to win or if Romania win by a greater margin.*

Implicit in this is the extra rule:

> *If England lose and Germany win by a greater margin than Romania then Germany will go through.*

Are these rules complete? In other words is there any combination of results from the two matches not covered. If this is so we could be in a position of not being able to say who goes through. This is possible so the rules are not complete. What if

Germany and Romania win by exactly the same margin? Who goes through then? We need an extra rule – and the Guardian gave one:

> *If Germany and Romania win by an identical score, Romania will progress thanks to a better points coefficient from the qualifying competitions for France98 and Euro2000.*

The rules are now complete. Whatever the scores of the two matches, we can say who goes through. As it happened, Romania won and Germany lost, so from the second rule Romania went through and England, sadly for Kevin Keegan were knocked out.

These rules are based on a more general set of rules about who goes through from each group. To be complete, those more general rules would also need to say what would happen in a similar situation if the two teams also have identical point coefficients from the qualifying competitions. This was only not an issue in the above because it was known that the Romania had done better in qualification so that could be built into the rules.

Humans have intelligence that they use to fill in the gaps in incomplete instructions. Computers on the other hand are not intelligent so complete instructions are very important. Programming languages are generally designed so that all instructions written in them are guaranteed to be complete.

Determinism

Another property that is often important of rules is that they are **deterministic**. By this we mean that if the conditions in which they are followed are identical, then exactly the same result will be obtained by following the result. You can determine what the result will be in advance if you know the rules and the conditions.

One place determinism crops up in real life is in games. Games are often split into two categories: games of chance and games of pure skill. An example of the latter is chess. If in two games the players make the same moves, then the outcome of the game will be identical and so could be predicted. In fact most chess players write down the moves of the games they play so that they can play the game back again later, and work out where they went wrong, or how they could have played the game better. In deterministic games, if you are clever enough you can theoretically work out in advance the best move to make from any position. For example, I (and others throughout history) have analysed Noughts and Crosses and know what moves to make in any situation. The book *Winning Ways* does this for a large number of games. Try and work it out for Noughts and Crosses for yourself! Determinism does not mean all chess or Noughts and Crosses games are identical, just that if the other player does the same things in two games, the rules ensure the same outcome if you also make the same moves. The most interesting games of skill are those like chess that have so many options at any move that their determinism is hard to utilise. The interest of the game is being able to make best use of the determinism.

An example of a game of chance is Roulette. It is non-deterministic. The outcome cannot be predicted in advance as it depends on the spin of the wheel. The whole point of a roulette wheel is to introduce non-determinism into the game. I can place my chips in exactly the same places on the table in two consecutive games. In one I could become a Millionaire and the other despite all the players doing exactly the same thing as last time I could lose everything. Think of some other games of chance and ask yourself, what it is that is providing the non-determinism. Non-determinism is in part about things outside the control of the instructions or players. If it was possible for some race of Aliens to predict the roulette wheel perfectly then the game would be deterministic for them. Crooked gambling houses may of course have crooked wheels that they can control. If they can control exactly where the ball lands (with magnets perhaps) then the game is no longer non-deterministic. If there is any element of chance in a game, however much skill is otherwise involved, it is still non-deterministic as there are still situations where different outcomes could result.

Consider my wife's instructions for shopping given earlier.

```
1. Go to shop
2. If the shop has Chocolate Muffins
           then buy Chocolate Muffins
   else if the shop has Croissants
           then buy Croissants
   else buy nothing.
3. Return back home
```

If on two different weeks, I am sent to the bakers for breakfast, then if on both occasions the shop stocks the same things, I will be guaranteed to return with the same thing (or with nothing both times). The set of rules given earlier are deterministic. If the shop stocked different things on the two days I could of course return with something different – that is why we specify that "the conditions are identical". Here we mean the things stocked by the shop are identical.

```
1. Go to shop
2. If the shop has Chocolate Muffins
           then buy Chocolate Muffins
   else if the shop has Croissants
           then buy Croissants
   else buy anything.
3. Return back home
```

This set of rules is not deterministic: they are **non-deterministic**. What is the difference? The catch-all rule gives scope for different things to happen, even if the shop stocks exactly the same things on the two occasions. On the first occasion, I could return with apple doughnuts by following the rule, and on the second, following the same rule I could return with cherry flapjack. Even if she knows what the shop stocks on a particular day, she would not be able to predict with certainty what I would have bought.

Here is another set of rules that are **non-deterministic**

```
1. Go to shop
2. If the shop has both Chocolate Muffins and Croissants
           then buy Chocolate Muffins OR buy Croissants
   else buy nothing.
3. Return back home
```

Again even if my wife knew that the shop had muffins and croissants, she would not be able to predict which of the two I would pick on any given occasion. Note that determinism is different to completeness. The above rule does tell you what to do in each situation, but what it says to do is pick randomly.

Termination
A final property that it is often important for algorithms to possess is that they are **finite**. By this we mean that when we follow the instructions, we will always finish and produce a result. We will not follow instructions forever, never coming up with an answer. For example, when we type in a calculation in a calculator, we want it to give us an answer. We do not want there to be a situation where it calculates forever, never giving us the answer. We want the calculation to terminate. Non-termination of a set of rules is normally a **semantic error**. It is normally a result of a mistake in the way the rules were written. We will discuss termination in more detail later.

3. Plumbing (Sequence)

"Lights-Camera-Action"

If I am to successfully do something following a set of instructions, I must know two different kinds of things. I must of course know what the individual instructions are. Less obviously I must also know what order I must follow them in. Algorithms, which are just written-out sets of instructions to do something, therefore consist of two things:

1. the **actions** that must be taken, and
2. the **order** they must be done in.

If movies about movies are to believed, when a film is being made, the director shouts "Lights-Cameras-Action" to start a new take. They are giving instructions to the film crew. First switch on the lights to light up the set, then start filming, then and only then should the actors and actresses start acting. The crew are not only being told what to do, they are being told the order to do it in. If the actors started acting before the lights were on, or the cameras were rolling then part of their potentially Oscar-winning performance would be missed! The order that the instructions should be followed is just as important as the things to be done.

When I was learning to drive, I was taught the mantra "Mirrors, Signal, Manoeuvre". What did this mean? Whenever I was about to make a turn, overtake, or in fact do anything other than follow the road ahead, I should first check my mirrors, then signal, then finally do whatever manoeuvre I was intending. To start with I tended to signal then check my mirrors then manoeuvre, for which my driving instructor gave me lots of grief. Sometimes I would start to manoeuvre and only then check my mirrors and signal. Had I still been doing it in the wrong order when I took the test I would have failed. Why? Because the order matters. It is safer if you check what is behind you and what its doing first and only then indicate your intention, and only once you have done that start to do something yourself. Test Inspectors are therefore finicky about whether you were following the algorithm properly – not only doing all the steps but doing them in the correct **order**. (As it happens I did fail my first test but not for failing to follow an algorithm, but because the instructions I was given were not precise enough. The examiner asked me to stop somewhere convenient on the left as we were going up a hill. I decided the most convenient place was away from the hill as hill-starts are a nightmare – unfortunately she wanted me to stop specifically so she could see me do a hill start. After that I hadn't a hope of passing – all because the instructions I was given were ambiguous.)

We will now spend some time looking at what we mean by the order things are done in. This is the plumbing of the algorithm: how are the instructions connected together. It turns out that only three kinds of plumbing are needed to write any algorithm. These are known as **sequence**, **selection** and **iteration**. A style of programming known as **Structured Programming** involves writing instructions only using these three forms of plumbing. It is as though a plumber only had some fixed shaped pipes with common connections, rather than being able to bend pipes into any shape. Sticking to a few well-thought out forms of connection helps keep things neat and tidy and thus easier to understand. It prevents the plumber from connecting pipes so as to look like spaghetti. Imagine trying to work out which pipes in the attic were connected to the mains if there were pipes going everywhere. It would be a little like solving one of those children's puzzles where 3 lines are intertwined, and you have to work out which one connects the picture of the dog to which person holding the end of a leash. The fact that it is a puzzle means it is intentionally harder than it need be. In fact that is exactly the problem I have trying to work out which plug is connected to the hairdryer and which to my alarm clock the cables of which are usually in a tangled mess, so that I do not unplug the wrong thing.

We will look at sequence here. We will look at selection and iteration in more detail in later chapters.

Most simple algorithms found in everyday life are short and involve doing a series of things one after another. Many of the algorithms we have looked at so far are like this. The algorithm is given as a list of instructions. Given a series of instructions we need to know what order they are to be followed in. A common way to indicate this is to number the steps. For example we saw earlier the instructions for opening the emergency door on an airliner:

Emergency opening

1. Pull cover aside
2. Push lever to open position and release.
3. Push door outwards.

The first line of this is not part of the algorithm, but rather the "title". It tells us what following this algorithm will achieve: it will lead to the door being open. The actual algorithm consists of three lines. Each is an instruction to do something, the numbering tells us the order to do them in. First we must pull the cover aside, then push the lever and release it and finally push the door outwards. If we try to do the instructions in a different order to that given we will not manage to open the door and die a horribly painful death. If we miss a step out then we also will fail to open the door.

The instructions to open the door on some old-fashioned high-speed trains that require you to lean out of the window to open the door are similar:

1. Wait until light comes on.
2. Open Window.
3. Lean out of window and turn outside handle.

Again numbers are used to emphasise the order the instructions are to be followed in. However, they are also placed one after another on consecutive lines, so even without the numbers most people would follow the instructions in the intended order.

Numbers might similarly be used on a ticket machine, perhaps for buying rail tickets:

1. Select destination.
2. Select ticket type.
3. Insert correct money.
4. Take change.
5. Take ticket.

A well-designed machine might allow several different orderings. For example, it might allow you to select the ticket type first and then the destination ("I want a return to Cardiff please"). That just means it allows several different algorithms to be followed. There are often many algorithms that achieve the same result. Such flexibility is needed because humans are not that good at following algorithms: "I am a human not an Automaton".

The following instructions given by the BMA (BMA, 1990) to prepare for an emergency childbirth. It is a sequence of actions to take, specifying both what should be done and the order it should be done in.

1. Summon medical help.
2. Reassure the mother, staying clam yourself.
3. Wash hands and scrub nails under running water without drying them.
4. Make sure everything you will use is clean.
5. Prepare a flat surface using a clean sheet.
6. Prop the mother up with pillows, with legs bent and apart and feet flat..

Numbers for steps are not always used to indicate the order actions should be done in. All that is needed is a convention. The most common convention in programs and in everyday life is that we follow the instructions, starting at the top and working down the list, doing the instructions in the order given (unless of course we are from a culture such as Chinese where writing is not left to right, top to bottom). For example the instructions for using a tube of glue might go something like:

Ensure both surfaces are clean and free of grease.
Apply a thin coat of glue to both surfaces.
Wait for 5 minutes.
Push the two surfaces together.
Leave for 30 minutes.

This is a **sequence** of instructions to be followed in order. The next instruction is only started on completion of the one before. The order the instructions are to be followed is just the order they appear on the page – top to bottom. Programs often use the same convention. Start with the instruction at the top of the program and follow them one at a time until you reach the bottom.

The 17th century Scientist, Sir Isaac Newton was a great lover of algorithms. In part his meticulousness over instructions was what made him a great scientist. When doing scientific experiments he was very careful to write down the steps he followed precisely – so that others could then repeat his experiments and so replicate his results. This repeatability of experiments is the backbone of the scientific method, though it was unusual then. His love of such recipes grew as a child. He collected recipes for making various brews and potions. Michael White's biography of him quotes an early recipe that he wrote down in his notebook in 1659 for making a crimson dye (Newton 1659)

"Take some of the clearest blood of a sheep &
put it into a bladder &
with a needle prick holes into the bottom of it then
hang it up to dry in the sun &
dissolve it in alum water according as you have need"

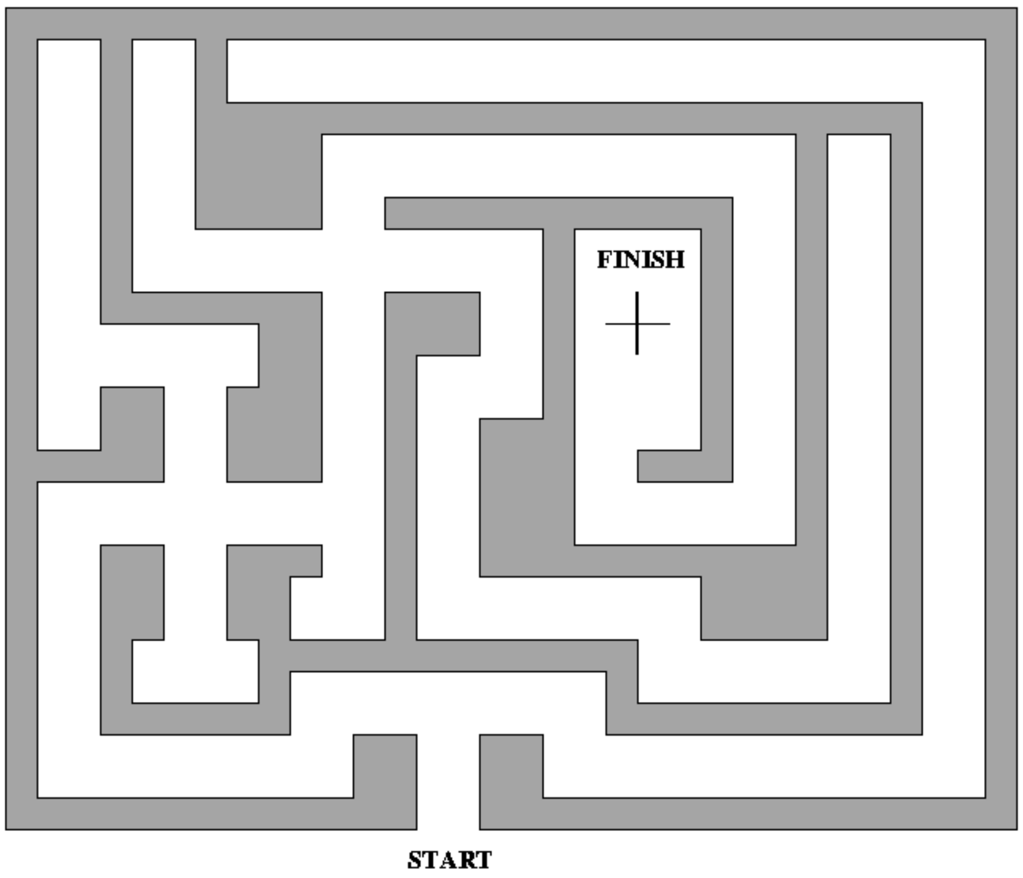
This slightly ghastly algorithm is just a sequence of instructions to be followed one after the other. Newton used the "&" symbol to indicate where one instruction ends and the next starts. In the language he was using to write his recipes "&" means something like "and then do" – it indicates sequencing. The instructions are to be done one after another. Notice though at one point he switches to using "then" to mean the same thing – not as meticulous as he ought to have been! Programming languages also usually have a symbol used to indicate when one instruction in a sequence ends and the next starts. For example a semicolon is often used for this.

The scripts of plays (and film scripts) are, in a simple sense, algorithms. They are a series of instructions that the actors are supposed to follow – either movements to make or things to say. The same things should be done and said in the 1000th performance as in the first (just as with a computer programme, it should do the same on the 1000th time it is run as the first). A script is a very simple form of algorithm as it uses only one method of giving the order of actions – that of doing things one after another. For example, Macbeth starts with the following sequence of actions:

Thunder and Lightning.
Enter three witches.
Witch 1: When shall we three meet again, In thunder, Lightning or in rain?
Witch 2: When the hurlyburly's done, When the Battle's lost and won.
etc.
Witches vanish.

Shakespeare was an early programmer!

When solving pencil and paper mazes like the one below, children normally draw a pencil line to show the solution. Have a go at solving this one.

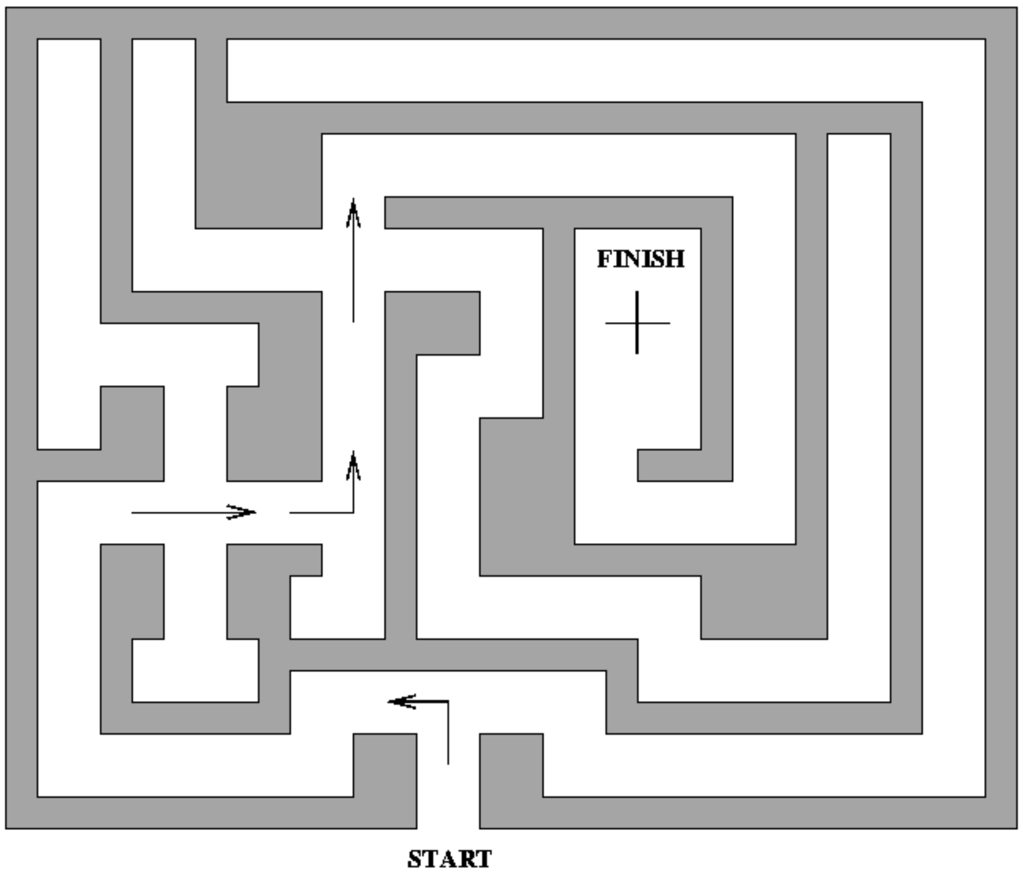


However, now suppose it was the plan of a real maze made of hedges, and you were studying the plan for a race to the centre. During the race you cannot take your plan. Once you have worked out the route to the centre of a maze, you must memorise an algorithm that you must follow to get to the centre. What you would need to do is work out what to do at each junction. Write an algorithm using the following actions:

Enter the maze by the gate following the path ahead.
Turn left following the path ahead.
Turn right following the path ahead.
Go straight across the junction following the path ahead.
Stop at the cross.

You must put the actions in a sequence (not necessarily in the order given) so that if someone followed them they would get to the centre of the maze. We assume that anyone following the instructions will stop and look for the next instruction whenever they get to a junction.You can use each of the above actions as many times as you like in the algorithm and you may not need to use some at all. Your algorithm should be the quickest route. Try this before reading on.

The following is the same maze with arrows drawn to show what to do at each junction to make it easier.



Here is my answer using the actions given.

To get to the centre of the maze:

1. Enter the maze by the gate following the path ahead.
2. Turn left following the path ahead.
3. Go straight across the junction following the path ahead.
4. Turn left following the path ahead.

5. Go straight across the junction following the path ahead.
6. Stop at the cross.

If you can memorise this algorithm then you stand a chance of winning the race to the centre as you will take the most direct route. Taking these instructions would also almost certainly be easier than taking your plan with you and trying to work out where you are from that. With an algorithm you can follow the instructions blindly without thinking.

Here is another algorithmic puzzle that is a variation of one invented by Alcuin who was a medieval mathematician born in the 8th century (Stewart, 1992).

> A man has been to the market and bought a new hen and a sack of corn. He also has with him his Pit Bull Terrier. To get home however he must cross a river in a corracle (a very small boat). This was not a problem on his way out, but when he gets to the river he realises that the corracle is only large enough to take him and one of the sack, the hen and the dog at once. This would not matter except that if he leaves the hen and the corn alone, the hen will eat the corn. If he leaves the Pit Bull and the hen alone, the Pit Bull will kill the hen. How does the farmer get them all safely across the river in the corracle?

This puzzle illustrates sequencing because to solve the puzzle you need to come up with a sequence of river crossings that the farmer can do sometimes crossing on his own, sometimes with one of the corn, hen and dog. You need to come up with an algorithm that the farmer can follow. Try and work out a solution and write down the algorithm as a sequence of instructions before you carry on reading. You will probably find it easier if you use different counters (eg coins) for each of the farmer, corn, hen and dog.

Here is one solution algorithm:

To cross the river:
1. The farmer crosses with the hen.
2. The farmer returns alone.
3. The farmer crosses with the corn.
4. The farmer returns with the hen.
5. The farmer crosses with the dog.
6. The farmer returns alone.
7. The farmer crosses with the hen.

This algorithm would be easier to understand if we added some explanation. Why for example does the hen cross three times? The following version has comments in italics as explanation.

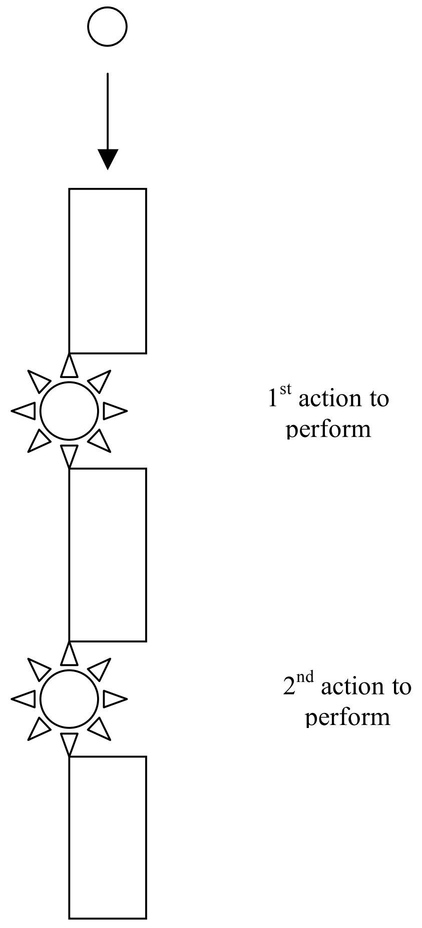
To cross the river:
1. The farmer crosses with the hen.
   As otherwise the dog will be left with the hen and eat it OR the hen will be left with the corn and eat it.
2. The farmer returns alone.
3. The farmer crosses with the corn.
4. The farmer returns with the hen.
   As otherwise the hen will be left with the corn and eat it.

5. The farmer crosses with the dog.
   The dog is left with the corn, which is not a problem.
6. The farmer returns alone.
7. The farmer crosses with the hen.

The extra explanations are not part of the algorithm (they are not instructions to take actions) – they just explain the algorithm, helping the person following the algorithm to understand it. Even with the first version the farmer could blindly follow the instructions without understanding why he was taking each step and still get the job done. We will return to the importance of such **comments** later.

The algorithm given above is not the only one that would work. In fact there are lots, but only one other one that is as fast. All the others involve the farmer crossing the river more times than necessary. Work out and write down the other algorithm that is as fast: it is a sequence of 7 steps (that is actions) too but some of the steps are different.

These examples use the **sequence** form of plumbing (i.e. connection). The separate actions that must be performed are connected together in a line. By following the plumbing you do the actions in the correct order.



Imagine a children's marble construction toy consisting of a series of wheels connected together by pipes as in the above picture. Suppose the parts of our construction kit are each a single pipe connected to a wheel. Each pipe-wheel part can be connected to another pipe-wheel part in a tower. Put a marble in the top pipe and the first wheel will turn. The marble then passes on. The first wheel stops and the second wheel turns. The marble passes out the bottom and everything stops. Only one wheel turns at a time and they do so in order from top to bottom of the tower. Imagine each wheel is an action to be performed and the marble run represents an algorithm. The actions are executed when a marble arrives and only finish when the marble moves on. We do not need to just connect two wheel-pipe parts. We can fasten any number together.

In computer terms, a form of plumbing is known as a **control structure**. The control structure here is the **sequencing** control structure. The plumbing ensures the wheels spin one after another in a fixed order from top to bottom. The marble represents something known as the **flow of control**. It moves from wheel to wheel, its position determining which wheel is active. In a program the flow of control moves from one instruction to the next, each instruction being executed only when the flow of control reaches it.

Another way of thinking of the flow of control is that it is like a baton being passed in a relay race. Only one person holds the baton at once. All the other members of the team stand around waiting for their turn. Think of the action commands in the relay as the individual runners. Only one action is being followed at a time just as only one team member is running at a time. Once that action is completed, "the baton" i.e. the flow of control is passed to the next action, allowing it to be followed. Once it is completed it too passes control to the next action. Sequencing is like a relay on a long straight track, with the runners lined up in a straight line.

Programming is not just about working out the actions to be performed but also requires the order to be given – that is the plumbing must be indicated.

In our Imp computer we have so far glossed over how the Imps know when to do a particular action – we have only looked at the storage Imps rather than the Instruction Imps themselves. Let us return to an earlier example of instructions to swap round the values two Imps were storing. The three instructions required were:

```
Trevor gets Alf's value.
Alf gets Bridgit's value.
Bridgit gets Trevor's value.
```

To do the swap they had to be followed in the order given. Try following them in a different order to see what happens!

How do we ensure they are followed in the correct order? We introduce the Instruction Imps. These are a bunch of nameless, faceless middle-manager Imps. Each such Imp is responsible for a given instruction. For example one would be given the instruction "Trevor gets Alf's value". It would be her job to tell Trevor and Alf to do this at the appropriate time. Similarly another Imp would be responsible for the instruction "Alf gets Bridgit's value." and a third for the last instruction.

At compile time the upper management would tell these three nameless Imps what there instruction today was, and also tell them who came next. At compile time all the storage Imps would also be gathered together according to the declarations. They would all then sit around doing nothing until "run time". Run time would start when the person operating the computer pressed the appropriate button. This would indicate to the Chief Executive Imp (normally referred to in hushed tones by other Imps as 'M') of the computer that action was needed. She would then take a special baton out of its box and pass it to the Instruction Imp who had been given the first instruction to follow. She would then go back to practising her putting whilst she waited for the baton to return. As the instruction of each instruction Imp was completed they would pass the baton on to the next Imp. A final instruction Imp, holding the instruction "return" would eventually get the baton, and return it to 'M'. She would put it back in its box, open the hatch in the roof of her office and shout to the computer operator, that the program had terminated.

Lets look at what happens when we execute our "swap" program. We assume Alf and Bridget have been already initialised with the values that are to be swapped (by some earlier instructions). Note also that we have added an extra instruction on the end to Return the baton to 'M':

```
Trevor gets Alf's value.
Alf gets Bridgit's value.
Bridgit gets Alf's value.
Return.
```

The baton is passed to the Instruction Imp with the first instruction. On getting the baton he knows it is his turn to do some work. He pulls out the instruction he holds and follows it. It tells him that he must get a copy of Alf's value and pass it to Trevor. When this is done he passes on the baton to the next Instruction Imp – who holds the second instruction. The first Instruction Imp goes back to his room and goes to sleep – his job is over for the day. The second instruction Imp now has the baton so must do some work. She follows her instruction, which is to get a copy of Bridgit's value and give it to Alf, then pass the baton on. She returns to the Pub. The third instruction Imp now has the baton so copies Trevor's new value to Bridgit and passes on the baton and returns to building a giant model of Leaning Tower of Pisa out of matches. The final Imp who has the baton has one job only, to return the baton to 'M' to indicate that the program has completed. Each Instruction Imp knows who to pass the baton to next because they were told it during the compilation process. Which it is is just determined by the order of the instructions down the page. Because in this case we are using sequencing, each time the program was run, the Imp passes the baton to the same Imp – there is no choice. In a later chapter we will look at situations where there is a choice of where the baton is passed next.

We saw in the introduction how the solution to a solitaire peg-jumping puzzle was an algorithm. Look at it again and you will see our answer was just a sequence of actions to perform. The traditional game of peg solitaire where a cross shaped board starts full of pegs, apart from an empty hole in the centre, and ends with a single peg in the centre is similar. Its solution is a sequence of jump instructions. If you have a solitaire board, you may wish to see if you can work it out. Solitaire is difficult, however. Here is another solitaire puzzle given by Ian Stewart (Stewart, 1991). It is played on a triangular board, with 10 pieces in the central triangle, but the outer edges empty. The very central piece is red. Any piece can jump over any other into the space directly beyond, with the jumped piece removed as in normal solitaire. The aim is to end with the red piece in the centre and all other pieces removed. Work out and write down the algorithm to do this.

Here is one solution. Follow it to check it works.
1. Jump piece 13 to 11.
2. Jump piece 5 to 12.
3. Jump piece 11 to 13.
4. Jump piece 13 to 6.
5. Jump piece 20 to 9.
6. Jump piece 6 to 13.
7. Jump piece 13 to 26.
8. Jump piece 17 to 19.
9. Jump piece 26 to 13.

Here is a variation that has slightly more complicated rules but with the same start and end position. As before any piece can jump any other into the space beyond. However, the jumped piece is only removed if it was jumped by the central red piece – the King. The above algorithm no longer works as the actions, though superficially similar now have a different effect – they have a different **semantics**. The semantics of the sequencing operation is unchanged, however. You need a new algorithm – a new sequence of moves. See if you can work one out (you may find this harder than the last puzzle – the algorithm needs to be slightly longer). HINT: remember other pieces can still jump the King.

Here is one solution algorithm. Follow it to see if it works.
1. Jump piece 8 to 10.
2. Jump piece 13 to 6 (capture).
3. Jump piece 6 to 15 (capture).
4. Jump piece 15 to 26 (capture).
5. Jump piece 26 to 13 (capture).
6. Jump piece 18 to 9.
7. Jump piece 17 to 8.
8. Jump piece 13 to 11 (capture).
9. Jump piece 5 to 12.
10. Jump piece 11 to 13 (capture).
11. Jump piece 14 to 5.
12. Jump piece 13 to 6 (capture).
13. Jump piece 6 to 4 (capture).
14. Jump piece 4 to 13 (capture).

4. On the Move (Variables, Assignment and Declarations)

"But it does move"

Galileo (1632).

Assignment and Variables
Storage space is very important in programming. Programs process information and that information needs to be stored somewhere. Processing it involves moving it from place to place. Without the ability to move data around, computers would not be of much use. We will examine here the basic features of instructions for moving things around.

Consider the game of chess. Chess players normally record their moves as they make them so they can play the games through again and analyse them. Such written version of games between grandmasters often appear in books and newspaper columns. Also books on how to play often include fragments of games "openings" for example such as one called "Ruy Lopez" and "Queen's Gambit". What is such a written version of a chess game? It is an algorithm of how to get from the initial position to the final position, following the rules of the game (ie how the pieces can move). If you were just told the final position of a game of chess, it would not (usually) tell you much. The algorithm of how that position arose can be fascinating for chess players. So what does such a chess game algorithm actually involve? It is a series of moves of pieces from one position to another – a series of **assignments** in programming terms. An assignment is just an instruction to move data from one place to another. Each square of the chessboard is a storage space that can hold a single piece at any time. The pieces are playing the same role as data values in a program – the things that are stored. The squares are **variables**. A variable in a computer program is just a place where things (data) can be stored.

Each square on the chessboard is given a name. In computer parlance a variable name is referred to as an **identifier**. The most common way in chess is to name each with a letter giving the column it is in and a number giving the row it is in. Thus *b1* is the name used to refer to the square that the white Knight starts in, in row 2 column 1. All identifiers in chess are thus a letter from a to h followed by a number from 1 to 8. Programming languages similarly have rules for what makes a valid identifiers so you can tell when something is and is not an identifier. For example, the rule might be that an identifier is a sequence of digits and letters of the alphabet but cannot start with a digit.

If we wish to specify a move of the game in chess we must specify two things – the piece to be moved and the place it is to be moved to. There are many ways used to do this and different chess books use different ways. The most common involves making moves like R – e4 to mean move the Rook (that is perhaps at e1) to square e4. However this does not always work – there are two Rooks of each colour and in some board positions either could move to square e4. For example perhaps the other Rook was at square e5. It could also be the one we meant. We need a system that uniquely tells us which piece we want to move. An easy way to do this is not to give the name of the piece (which is not unique) but to give the name of the square. This is unique since we have given a different name to each square and only one piece is on a square at any time). Thus the easiest way to specify a chess move is just to give the start square and the destination square. Thus in the above situation we would say e1-e4. Had we meant to move the other Rook we would have written e5-e4. Notice that we are using a special symbol "-" to mean move. Notice also that we do not mean move e5 itself (the square). We mean move the piece that is currently in that square. At different times in a game"e5" could refer to completely different pieces – at one point a pawn, later a Bishop, then another pawn. "e5" is an identifier of the variable. A whole game would be specified by giving a whole series of such moves. In essence we have created a language for writing chess algorithms in. The language used in chess books is usually a little more complicated – it includes instructions that mean "The player Resigns", use a "x" to mean move to a square and capture the piece that is there and also have a way of indicating whether white or black. Why has a special language been created? Why do the books not just use English? It is essentially for the reasons we discussed for computer languages – we need to remove **ambiguity** as just discussed. Such a language is also very concise -–and once learnt is very easy to understand (though looks like meaningless lists of numbers and letters to someone who does not play chess).

A program is an algorithm for moving data (usually numbers) around in a computer rather than pieces around a chessboard. However the same principles apply. How do we specify moving something from one place to another? We need to specify what is to move and where it is to move to. Identifiers (i.e. the names of storage spaces) are used to refer to the thing being moved and also the place it is to move to. We need some symbol to indicate that we are talking about moving something from one place to another (like the "-" we used in the chess algorithms). Otherwise we would not know whether the instruction was to move the data or (for example) do a calculation on it. Different languages use different symbols (like "=" or ":=") to mean "assign". For example, to write an instruction to move a chess piece using programming language notation you might write something like.

```
e4 := e1
```

or

```
e4 = e1
```

rather than

```
e1-e4
```

to mean square e4 gets the piece currently at e1.

This use of different notations to mean the same thing in different languages is just like in human languages where different languages have different words for the same thing – French for example uses the symbols "bouger" to mean what in English is referred to using the symbol "move". Once you understand the concept however switching between languages is much easier.

Consider another example from the world of games. When I was at school we had a craze for playing the game of *Diplomacy*. It is a war game set on a board of Europe. Unlike other war games the fun is not just in moving pieces about but in, negotiation making secret deals with other players and double crossing (like real Diplomacy). Here we are interested in the actual moves. The board is divided into named areas corresponding to places like London or Bulgaria and seas like the North Sea or the Black Sea. On each round of the game, players move their pieces from one location to

an adjacent location. As with chess, each place can only hold one piece so there are rules about how you decide who wins when two pieces are supposed to be in the same square. The moves are intended to happen simultaneously, but obviously, in reality they are done one at a time with a period at the end of each turn for deciding the conflicts. To ensure no one has a chance to change their moves on seeing someone else move, each player writes their orders down. For example, the person playing England might write:
Lon-North S
Den-Kie
As with chess these are a series of assignments. Now the **variables** are the named areas. Their names like "London" shortened to "Lon" are identifiers. Again "-" means "move". The problem is similar to that for chess and the solution is basically the same. The movements are specified by giving the locations moved to and from.

There is one difference in the way assignment is usually written in programming languages that often confuses beginners. Many programming languages give the two locations apparently the wrong way round. e1-e4 in our chess language above was used to mean "Move the piece from e1 to square e4". In programming languages it is often the other way round: i.e. if you meant "Move the piece from e1 to square e4" you would write e4-e1. This looks bizarre but it makes sense if you read it as: "Square e4 gets the piece from square 1".

One way to think of variables is just that they are storage boxes, each big enough to hold just one thing. Assignment is just an instruction to move something from one box to another. However, as each box can only hold one thing, if something else is already there, that thing is first thrown away unretrievably to make room for the new. Assignment in programming languages is a little bit more subtle than that though and this is a way it differs from moves in chess or Diplomacy. The thing being moved is actually copied first and it is the copy that is put in the new box. What this means is that not only is the thing copied placed in the new box, but an identical thing is also left in the original. This is as though the rules of chess were such that if you moved a pawn forward a square, the new position would have a pawn both in the original and new positions.

The memory buttons on calculators are providing an assignment operation of this copy form. They are used to store temporary results in the middle of calculations. This is exactly what a variable in a computer program is for: storing something for later manipulation. The memory is thus a variable: its identifier on most calculators is M. However, when you use the memory button to move a value from the memory to the display, the value stays in the memory too, so can be used again. Some calculators have multiple memories: ie several variables with names like M1, M2 and so on. When you press the memory button you do an assignment to the memory. You are basically saying put the number currently on the display into the memory. Again you are specifying (at least implicitly) the place to get the value from and the place to move it to. As the display is the only place that is allowed as the source of the assignment on calculators you do not have to explicitly specify it. If there are multiple memories you will need to indicate which is to be used – usually by pressing a different memory button.

Many phones have a way of storing commonly used numbers so that they can be dialled quickly with only one or two button presses. For example I often phone my parents and my wife's work. It is useful to have those numbers stored and so be able to phone them quickly. Each of the quick dial locations is a variable in computer science terms – it is a place where numbers can be stored until they are needed. The label on the button is its identifier. The identifiers might just be numbers or sometimes they are special buttons with identifiers like M3. How do I store a number in a quick dial button? The instruction book gives me an algorithm.

1. Press the quick dial button (Q)
2. Press the numbered button where the phone number will be stored
3. Dial in the telephone number to be stored.
4. Press the "enter" (E) button.

This is giving an algorithm for typing in an assignment command. Suppose I wanted to store in quick dial button 3 the number 020-1234567. Following the algorithm I would press the following keys:

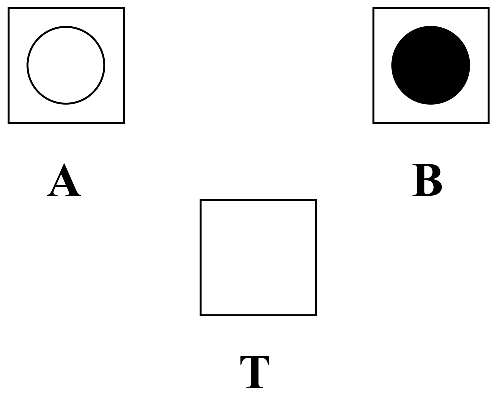
Q 3 0201234567 E

This is just another notation for assignment! Another way of writing this might be:

3 ← 0201234567.

That is the same instruction – just written in a different language – perhaps for a phone with a key marked ← to mean store in a memory key and a key "." to mean "enter". The first Q can be thought of as being a symbol to mean we are doing an assignment. The letter is the place to assign to and the following digits are the value being assigned to that key (i.e. being stored there). The final E can be thought of as punctuation –like a full stop in a sentence. It indicates that we have finished the command. Computer languages have similar punctuation characters. In some languages a full stop is used to mean the end of the current command, in others a semi-colon is used.

Swapping the positions of two things is a common operation that when broken down into its basic steps is a sequence of assignments. We will use it later when we look at sorting things. Try the following simple puzzle (using coins) that requires you to devise a swap algorithm. (One answer is at the end of the chapter).



A board consists of three squares named A, B and T, and two pieces, one black and one white. The black piece is placed on square A, and the white piece is placed on square B. Pieces can be moved from any square to any other. However, if a piece is moved to a square where the other piece sits, then that piece is removed permanently

from the board. Work out a sequence of moves (and write them out in English) that lead to the positions of the two pieces being swapped. Now write down your sequence of moves using the notation *x* ← *y* to mean "square *x* gets the piece currently on square *y*".
Here is one way to do it.

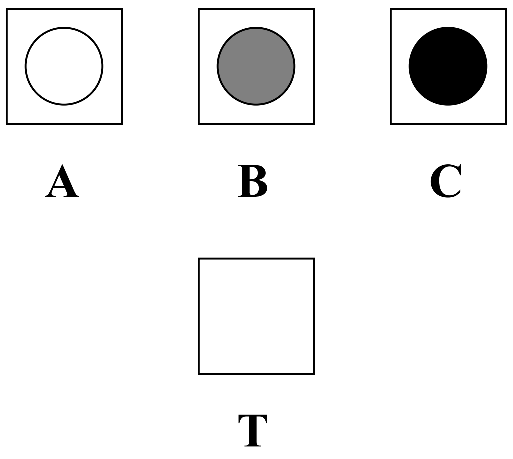
1. Move the piece at square A to square T
2. Move the piece at square B to square A
3. Move the piece at square T to square B

Notice that this takes three moves. You cannot do it in two as you would remove a piece if you for example tried to move the piece on square A straight to square B. We will look at why that goes wrong later but check it for yourself now. Writing the above moves out in our invented notation.

1. T ← A
2. A ← B
3. B ← T

Remember to check you wrote the assignments the right way round to mean for example "square T gets the piece on square A"

Rotating the positions of things needs a similar algorithm to swapping. Try the following similar puzzle.



A board consists of four squares named A, B, C and T, and three pieces, one black, one grey and one white. The black piece is placed on square A, the grey piece on square B and the white piece is placed on square C. Pieces can be moved from any square to any other. However, if a piece is moved to a square where the other piece sits, then that piece is removed permanently from the board. Work out a sequence of moves (in English) that lead to the positions of the three pieces being rotated one square to the right (so that the black piece is on square A, the white piece is on square B and the grey piece is on square C. Now write down your sequence of moves using the notation *x* ← *y* to mean "square *x* gets the piece currently on square *y*" as before.

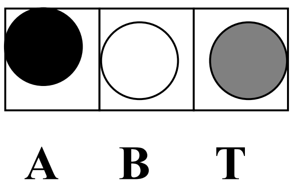
Here is one algorithm to do this rotation in English

1. Move the piece at square A to square T
2. Move the piece at square B to square A
3. Move the piece at square C to square B
4. Move the piece at square T to square C

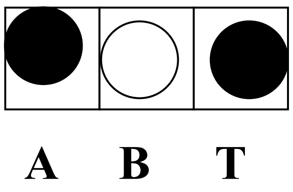
and in the special notation:

1. T ← A
2. A ← B
3. B ← C
4. C ← T

In most of the above we have talked of *moving* things from one place to another. That is not quite what computers do. Rather than move things they just copy them. When you move something it is no longer in the place where it was – so the above swap puzzle, moving a piece means there is no longer a piece in the place it came from. Computer assignment leaves a copy of the original in the original place. Variables can never be "empty", the may contain 0 (but that is still a thing) or you may not know what is in them (but something will still be there!). To see how a computer actually swaps things lets look at a variation on the swap puzzle. There are now three squares with identifiers a, b and t as before. In this puzzle they start with a piece in each, one black, one white and one grey.



The aim is again to swap the pieces in A and B, so that now A is white and B is black. Now you cannot just move pieces. Instead you can just copy them. You have a supply of other black, white and grey pieces. Each "move" now involves doing the following: pick a square, note the piece in that square, get another piece of the same colour and finally place that new piece in some new square discarding whatever was already there. So for example, now the instruction T ← A would mean look at what is in square A (its black at the moment), get another black piece and put it square T throwing away the grey piece that is there. The new board position is:

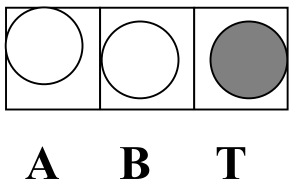


Write down a sequence of moves that would lead to A being white and B black – so they are swapped over (we do not care what T ends up with). In particular the following algorithm does not work, why not (try it and see).

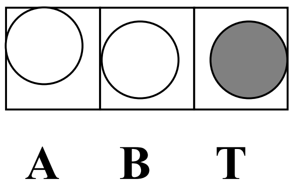
1. A ← B (A gets a copy of B)
2. B ← A (B gets a copy of A)

Each "move" in this puzzle is just like an assignment in a simple programming language, the squares are storage spaces – variables and the assignment copies whatever is in one variable (square) to some other square.

Why did the above 2 step algorithm work? It is because the first step makes a copy of B and puts it in A:



There are now no black spaces on the board. When you do the second step you are now making a copy of A and putting it in B, but A is white so you just throw away the old white piece and put in a different but identical looking white piece. The board looks exactly the same after the second step:



As there are no black pieces after the first move there are no blacks to make a copy of, so there can not do any move that will get a black after.

What is the correct algorithm? It is exactly the same one as for the original puzzle. We must make a copy of one of the pieces in T first so that we do not lose it when we copy something different into its position.

1. T ← A (T gets a copy of A)
2. A ← B (A gets a copy of B)
3. B ← T (B gets a copy of T)

Let us return to our design of the Imp computer – a computer that works using Imps to do the computation. Storage space in our Imp computer consists of boxes. There are two kinds of Imps: memory Imps and instruction Imps. The instruction Imps are the middle managers giving orders. The memory Imps are the workers – who pass data around. Each memory Imp owns a box, capable of storing an object or perhaps a piece of information. The boxes act as the memory of the Imp computer (in our world Imps themselves are very forgetful so for the computer to be reliable it is important that their own memories are not relied on). Each Imp that is required to remember something is referred to by name in instructions concerning that Imp. This is the way individual Imps are identified: it is their identifier.

The instruction Imps do not have boxes, but instead have pieces of paper containing the instruction they must follow if ever it is their turn. We will return to how we determine when it is a particular elves turn later. Here let us look at the individual instructions. An Assignment elf is one holding an instruction to make a copy of a piece of data held in the box of one Memory Imp and put a copy of it in some other memory Imp's box. An example instruction might be "Joe gets a copy of whatever is in Fred's box". The Assignment elf with this instruction would go to Fred, copy down onto a new piece of paper whatever was in the box and put it into Joe's box, hiding forever what had been in that box. The two elves Joe and Fred here are acting as variables. Their identifiers are the names "Joe" and "Fred".

Let us return to the puzzle above of swapping the positions of two counters but look at it in terms of an Imp computer. Our aim now is to swap the things that two Imps are looking after (storing in their boxes). Suppose the two Imps concerned are called Alf and Bridgit. Alf has the number 6 on a piece of paper in his box. Bridgit has the number 42 in her box. We need Alf to pass his number to Bridgit and Bridgit to pass her number to Alf. However they can not do it simultaneously – one has to go first but that would leave one Imp temporarily holding two things. Imps only have one box and refuse (due to Union rules and their poor memories) to remember more than one thing at once. We therefore need another Imp to temporarily store one of the pieces of information. We there for rope in another Imp, Trevor, to help. What is the program to do this on the Imp computer:

1. Trevor gets Alf's value.
2. Alf gets Bridgit's value.
3. Bridgit gets Trevor's value.

If we agree that an arrow means the Imp named on the right of the arrow passes a copy of their value to the Imp on the left who puts the number in the box, discarding whatever was there before then we can use the following shorthand.

1. Trevor ← Alf
2. Alf ← Bridgit
3. Bridgit ← Trevor

This is the same algorithm as we saw earlier (just with different identifiers for the variables). Notice that in each of these instructions, the two Imps named have to do something different.
In the instruction:

    Trevor ← Alf

Alf must make a copy of his value to pass it on (All Imps come with a supply of rice paper and a pen to do this). Trevor on the other hand is not being told to do any writing, just to take the piece of paper given and put it in his box, throwing away whatever was there. (Imps actually eat the paper rather than throw it away as Imps always use rice paper to write on – once eaten the value can never be recovered).

If on the other hand the instruction had been

    Alf ← Trevor

then Trevor would be doing the writing and Alf would eat his old value.

Initialisation
The point of variables is to store things that can then be moved around or manipulated as we discussed. Before you start doing this you have to **initialise** the variables. By this we just mean to set starting values. In chess, we initialise the squares with pawns on the second row, and Rook, Knight, Bishop etc on consecutive pieces of the first row. You cannot start a new game until you have initialised the board. At any one time there are always some squares that are blank. Those squares have not been initialised with a value. Obviously you cannot give a move that moves a piece from an

empty square so you cannot in the first move refer to any of those squares as the source of the move. A first move of e3-e4 is nonsensical as there is nothing in square e3 to move at the start. You could of course move a piece into e3 first and then move it out again on a later move.

It is possible to buy chess puzzle books. They give chess positions that are 2 moves away from checkmate. The puzzle is to work out what the moves are for that position. Each puzzle uses a different initialisation of the board. Somehow the book has to tell you which pieces too put where. This is usually done with a picture of the board, but this is giving the same two bits of information a square (a variable) and a piece (value) to go in that square at the start.

A chessboard can be used to play draughts. It is just initialised in a different way (even though the board itself is identical). Different pieces are placed on different squares to start the game.

My phone has a feature that allows me to store in special "quick dial" buttons numbers I use frequently like those of my parents. The idea is that you can phone common numbers with the press of just two buttons. What would have happened if I pressed one of these buttons when I first bought the phone before I had stored any numbers? There are several possibilities. One is that the buttons could do nothing, another is that they dial a random number, another is that an error message might be spoken in the handset. None of these options would result in a sensible number being dialled – pressing the button before they have been set is not useful. It most probably would be a mistake on my part. Before I use those buttons I must initialise them by doing an "assignment" as we saw earlier.

The Diplomacy board also has to be initialised. The instruction book does it saying what piece goes where. For example England is initialised with:

```
F London
F Edinburgh
A Liverpool
```

i.e., Fleets are placed at London and Edinburgh and an Army at Liverpool – my apologies to the Scots (and Welsh) about the above – the Diplomacy Board really does have Scotland (and Wales) as regions of "England". These instructions are again just assignments, but this time rather than moving a value from one place to another we are moving a value not currently at another location to a location on the board. We use a notation that allows us to specify the place moved from and the value (i.e., piece) to be moved there. A similar facility is provided in programming languages – though the notation used is normally the same as for assignment. You thus might write something like

```
  London = F
```

rather than F London to mean "London gets a Fleet".

The above example is different to earlier examples in that rather than just moving existing things from one storage location to another, we are putting a new piece or value into a storage space. We thus have a notation not only for referring to the storage locations (the variables) but we also need a notation for referring to values. In Diplomacy there are only two kinds of piece – Fleets and Armies so we use F to mean a value of the former and A to mean a value of the latter in our notation. Suppose we wished to give an algorithm for initialising a chessboard. Normally, this would be done using a picture. To give it as a series of instructions (an algorithm) we need a distinct (i.e. unambiguous) way of referring to each piece. What are the values of a chessboard? – Pawns, Rooks, Knights, Bishops, Kings and Queens. The normal convention used by chess players is for the pieces to be named by a single letter – P, R, N, B, K, Q. However that on its own is not enough. We also need to indicate whether the piece is black or white otherwise our notation would be ambiguous. We might therefore add an extra letter (W or B) to each to indicate the colour. The algorithm might be then written something like:

```
        a1 = WR
        a2 = WB
        etc
```

We have thus given ourselves a notation for referring to the chess pieces, the values. This raises an important issue with respect to readability. The above notation is only readable once you have used it enough to be familiar with the code. It would be much more readable if we had actually used longer identifiers.

```
        a1 = White_Rook
        a2 = White_Bishop
        etc
```

That way anyone who was familiar with chess pieces would immediately understand the notation. It is similarly important with programs that names chosen both for values and variables are easily understandable without explanation. This is known as good style. A program is not wrong if it uses obscure names for things in the sense that it will (possibly) do the correct thing. However it is bad if it is hard for other people to understand.

When we think in terms of chessboards, moving a piece from an empty square seems a silly idea. However, a common fault in programs is to do precisely that – forget to initialise a variable and then give an instruction to move a value from it when one does not exist. Different programming languages vary on what happens when you try this but in all cases it would mean the program was not as intended.

When you switch on a calculator, it automatically initialises the display to 0. Some programming languages automatically initialise variables to zero. More commonly they start with some random and unpredictable value unless the program explicitly contains an instruction to put something else there.

In terms of our Imp computer, initialisation is just a question of ensuring all the memory Imps being used have been given something to store in their box at the point when they are told they are needed for a particular program run. If they are not then their box will have whatever number happened to be there from the last time the box was used (which could be anything). If this Imps value was used before anything new was put in the box, then the Imp computer would have an unpredictable result as it would depend on the old value being used.

Expressions

So far we have seen assignments where a literal value is moved to a storage location. We have also seen how rather than referring to a value we can refer to a storage location to indicate which thing we wish to move. A third thing that we can do is to move the value of a calculation to a variable. Instructions to do calculations are referred to as **expressions**. Often the calculations we wish computers to perform are on numbers – arithmetical calculations. For example, suppose I am told the time of my train is at 17 hundred hours. The first thing I do is mentally convert it to a 12 hour clock as that is what I am most familiar with. What I do is a calculation – I subtract 12 from the 24-hour time given. How might I write an instruction to do such a calculation for someone else to follow – I just use the notation from school

(17 – 12)

giving the answer 5 if followed. At school we learnt notation for writing more complicated calculations such as how to work out the average of several numbers.

How would our Imp computer deal with expressions? We would require an Imp who was expert in each of the arithmetical operations – an addition Imp, a subtraction Imp and so on. Whenever a calculation (say addition) needed to be done the memory Imps holding the values concerned would pass their values to the appropriate Imp who would do the calculation and pass the result to wherever the result was to be stored. For example Arthur might be the addition Imp, doing any addition needed, Sol might be the subtraction Imp.

For example, suppose the instruction to be followed was

Joe gets the answer of Arthur adding Bridget and Alf's numbers.

perhaps written in our short hand notation as:

Joe ← Bridget + Alf

Arthur here is the addition Imp referred to using the symbol +. This means get a copy of the value held by Bridget in her box and a copy of the value held by Alf, pass them to Arthur who will add them (the only thing he is useful for) and pass his answer on to Joe to store in his box.

The Imps would be able to cope with more complicated calculations by working together. For example if the instruction were:

Joe ← (Bridget + Alf) - Conor

This would mean that first Arthur would be passed the values from Bridget and Alf as before. However, rather than passing the result to Joe he would pass it to Sol (the subtraction expert) who would also get the other value from Conor (an Irish Imp) and do the subtraction, before passing the final result to Joe.

If we needed our Imp computer to keep a count (perhaps of the number of times a dice had been rolled), then we would need an instruction to add 1 to the count. Suppose Zack was charged with keeping count then he would do this by keeping the current count in his box. Every time the dice was rolled, Zack would need to look in his box, note the number, add one to it and store the new answer back in his box. However, adding one is an addition calculation and Arthur is the only one who can do that (there are poor levels of numeracy amongst Imps generally though Union rules also mean only registered addition Imps can do addition). The instruction we write to tell Zack how to get the addition done is thus:

Zack ← Zack + 1

What does this tell Zack to do? He must pass a copy of the value in his box to Arthur. Arthur will add 1 to it and pass the result back to Zack, who will then put it back in his box. The result: Zack has added 1 to his count without knowing how to add 1 himself!

Remember at the start we talked about how in the Second World War, the British broke the German Enigma codes? This allowed them to read all their encrypted communications. This was done by having teams of people working in separate Huts. Each did a specialist part of the calculation and passed on the results to someone else in another Hut who could do the next part. The result was that together they cracked the codes, without understanding anything other than their own part. Only a few Mathematicians such as Alan Turing understood the whole process. In essence those people were just acting like an Imp computer with Turing their programmer.

Boolean Expressions

A calculator is used specifically to do calculations on numbers. The answers it produces are also numbers. The expressions use operators such as + and -. However expressions do not have to only be about numbers. We also frequently use expressions that give **boolean** answers. A boolean is just one of the values **true** or **false**. Quizes and tests often use "true or false" questions. Here are some examples that might be used on a Radio quiz:

True or False:

1. Freddy Mercury is alive.
2. Britney Spears is alive.
3. David Beckham is dead.
4. Elvis Presley is dead.
5. Clint Eastwood is alive.

At the time of writing the answers to these questions are

1. False 2. True 3. False 4. True 5. True.

Here is another set that might appear on a school test.

True or False:

1. Glasgow is the capital of Scotland.
2. Sidney is the capital of Australia.
3. Diamonds are made of crushed Carbon.
4. Sheffield is in South Yorkshire.
5. Rubber comes from a plant.

The answers to these questions are:

1. False 2. False 3. True 4. True 5. True.

The answers are not numbers as for questions calculators are used to answer but either true or false. They are thus questions with **boolean** answers. As we will see later booleans are very important in computer programming, as they allow computers to make decisions. Notice that to get a question with a true or false answer it has to be phrased in a particular way. You cannot ask "Is Freddy Mercury Dead or Alive?" The answer to that question is either "Dead" or "Alive" not "true" or "false". Programming languages similarly require questions to be phrased in a particular way to ensure the answers are either true or false.

Expressions are made up by applying operators to values: 6+2 for example applies the arithmetic operator + to the numbers 6 and 2. We can and do build boolean expressions in a similar way. The operators in boolean expressions are things like **and** and **or**. We can use these words to convert true/false questions into more complicated true/false questions. For example the following question uses the logical connective **and**:

True or False:
Clint Eastwood was in the film The Good, the Bad and the Ugly **and**
Harrison Ford was in the film Mad Max.

This is made up of two separate questions: one about Clint Eastwood and the other about Harrison Ford. Depending on the answer to the parts you may need to know the answer to both to be able to answer the question. Only if both separate parts are true can the whole statement be true. If either were false the whole thing would be false. In fact it is false because it is not true that Harrison Ford was in Mad Max.

A similar question using the connective **or** is a different question all together:

True or False:
Clint Eastwood was in the film The Good, the Bad and the Ugly **or**
Harrison Ford was in the film Mad Max.

This is true because Clint Eastwood is in The Good, The Bad and the Ugly even though Harrison Ford was not in Mad Max.

Buses sometimes have signs using boolean connectives. They give tests that must be applied by the driver to determine if the bus is full. For example, the sign may say:

35 sitting **and** 10 standing

What this means is the driver must ask the question of his or herself before letting a passenger on if the bus appears to be close to full: Are there already 35 people sitting and 10 people standing. If the answer is yes (so the original statement is true) then the driver must not allow anyone on the bus. Notice that boolean tests are special kinds of actions that gather information rather than actually doing anything. The bus driver evaluates the test so that they can make a decision about which action to take next. If the bus is full then the action they must take is to refuse to allow another passenger on. If it is not yet full their next action can be to take the fare of the next person. We will be looking in more detail at how the results of such tests determine our actions in subsequent chapters.

The film *Anna and the King*, which is about an English schoolteacher who arrives in Siam (now Thailand) to be the teacher of the King of Siam's children. It had been agreed before she arrived that she would be given a house to live in outside the palace. However, despite being there for months she is still in the palace. One day she is taken by surprise when the King finally gives her a house of her own. Taken aback she asks

"*Is this because of our agreement* ***or***
are you simply trying to get rid of me?"

Wishing to be evasive, the King answers

"Yes".

This highlights yet another ambiguity in English as well as being an example of boolean logic. Anna of course was asking which of the two options was the true one, expecting to be told "Its because of our agreement". However, the King has quite legitimately answered as though he was asked a boolean true/false question. If it is true that he is doing it because of the agreement or it is true that he wants rid of her, (or both) then the whole statement is true. It *is* either because of the agreement **or** because he is trying to get rid of her.

In a similar way if I am asked “Are you going to do the washing up **or** not” I can legitimately (and irritatingly) answer “Yes” – I am going to do one or the other (probably the latter).

We can build arbitrarily large questions using *and*, *or* and some simple basic true/false questions. For example, the rule of when you have won a game of chess can be thought of like this. You win if the other person resigns **or** their King is in check and all positions the King could move to also leave it in check.
On each move you ask the true/false question:

True or False:
the other person has resigned **or**
their King is in check **and**
all positions the King could move to also leave it in check.

Each part of this is a true/false question in its own right.

Returning to the bus example, more modern buses have spaces that can take wheelchairs. The presence of a wheelchair perhaps takes up two spaces. The sign about the test for when the bus is full will need to take this into account.

35 sitting **and** 10 standing **or**
1 wheelchair **and** 34 sitting **and** 9 standing

Menus in restaurants often have boolean like expressions in them to give descriptions of the meals:

Fish and Chips.

Treat this as a true/false statement:

The meal has fish **and** the meal has chips.

What they are saying is that this true false statement will be true for the meal you get. Sometimes they get into trouble due to the ambiguity of language:

Fish and Chips or Baked Potato and Salad.

If you were served just Baked Potato and Salad would you have cause for complaint? That depends on where the brackets are supposed to be

(Fish and Chips) or (Baked Potato and Salad)

is a different statement to

Fish and (Chips or Baked Potato) and Salad

Since there are no brackets, unless an order of precedence has been agreed the statement could be interpreted as meaning a meal without fish. I have had arguments with waitresses over menu entries similar to that. The waitresses concerned have just assumed I was mad (probably a true statement anyway). Just as the position of brackets can change the result of an arithmetic calculation, they can also change the result of a true/false calculation.

The sign on the bus was written with the intention of the brackets being as follows:

(35 sitting **and** 10 standing) **or**
(34 sitting **and** 9 standing **and** 1 wheelchair)

With the brackets in different places it would

((35 sitting **and** 10 standing) **or** (34 sitting **and** 9 standing))
and 1 wheelchair

then it would suggest that there was room for a wheelchair even if there were 35 people sitting and 10 standing.

Boolean expressions can have variables in them just as arithmetic ones can. If we think of a variable as something whose value can change with time then “The Queen of Great Britain” is like a variable. Its value (ie the actual person concerned) is different at different times. (Coronations are then just assignments putting a new person to the post!) Similarly “the Prime Minister” can be treated as a variable.

The Queen of Great Britain is Elizabeth **and** *the Prime Minister* is Tony Blair is a true/false statement whose value changes over time. In the year 2000 it is true. In the year 1980 it was false as then the variable “the Prime Minister” had the value Margaret Thatcher. The answer to a true/false statement changes if the variables within it change their value. Similarly in our bus sign stating “35 sitting **and** 10 standing”, “sitting” and “standing” can be thought of as variables holding numbers that vary as people get on and off the bus. As a person gets on the bus and sits down the number of people sitting goes up by one – so in effect that variable does in the mind of the bus driver if they are keeping track of how full they are.

In programming languages tests have to be written in a rigid form. Remember programming languages are designed to remove ambiguity and do this by restricting how you are allowed to say things. In our bus example, we wrote “10 standing”. In a programming language we would probably be required to write this in a form similar to:

standing **equals** 10

using the **equality** operator. It is true if the two things given with it are equal – here we are testing whether the number of people standing is equal to the number 10 or not. Different programming languages would use different words or symbols for equals for example one (such as the language Pascal) might require you to write

standing = 10

and another language (such as the language Java) requiring you to write

standing = = 10.

In these two cases the meaning is intended to be the same – its just that the different languages designers have chosen different words to mean “equals”. The full sign would need to be written something like:

(sitting **equals** 35 **and** standing **equals** 10) **or**
(sitting **equals** 34 **and** standing **equals** 9 **and** wheelchair **equals** 1)

Similarly if we were trying to write in the style of a programming language we might write for our earlier examples:

Glasgow **equals** the capital of Scotland

The Queen of England **equals** Elizabeth **and**
the Prime Minister **equals** Tony Blair

where before we were using the word “is” to mean “equals.

Whenever you are writing a test to go for a program you must look for situations where you are saying two things are equal and convert it to this form of asking whether it is true that one thing equals another.

Equality is a **relational operator**. It relates two things. Other relational operators, familiar from School Maths lessons are “less than”, “greater than”, “less than or equal to” and “greater than or equal to”. These are normally written by mathematicians as <, >, ≤, ≥. The bus sign was written in a way so that the test is true when the bus is full.

Alternatively the sign could have been written to be true when the bus still had space. The bus is legal if:

$\text{sitting} \leq 35$ **and** $\text{standing} \leq 10$

Declarations

Here is a recipe:

Fiorentina Pizza

Ingredients

Yeast
Water
Flour
Tomato Puree
Cheese
Spinach
1 egg

1. Mix yeast and water, add flour and stir
2. Knead for 10 minutes.
3. Leave to rise for 30 minutes.
4. Roll out the dough.
5. Place in a large round pizza tin.
6. Spread with tomato puree, and cheese and spinach.
7. Crack an egg into the middle.
8. Bake in oven for 25 minutes.

This recipe for pizza, starts with a list of ingredients. They are not part of the algorithm as such since they are not instructions of how to do something. The algorithm consist of the number sequence of instructions. They could be given in any order. Why do recipes universally start with an ingredients list? It lists the resources that will be needed for the instructions to be followed. However, all that information is contained within the instructions themselves. On seeing an instruction, crack the egg into the bowl, I can see I need an egg. The advantage is that if the ingredients are listed at the start, you can ensure that you have everything you need to hand before you start.

Craft books, from how to make wooden toys to how to make necklaces use a similar list of resources. Here the "ingredients" are the materials:

Materials
5mm plywood (160 x 100mm)
2 20mm round-head woodscrews
PVA glue
paint

In addition, unlike recipe books craft books often include also a list of tools with each separate set of instructions.

Tools
Drill
Fretsaw

Paintbrush

This is just a different kind of resource being listed. That would be the equivalent of a recipe book also listing the pots and pans needed (something I would often find useful as I frequently run out of pans).

The script of a play also has the same structure. A play is just a series of instructions to actors playing different parts. Before starting you need to allocate parts to actors, so you need to know what parts this play has. My copy of Shakespeare's *Macbeth* has such a list on the first page before the actual play itself.

Persons Represented

Duncan, King of Scotland.
Malcolm, Son of Duncan.
Donalbain, Son of Duncan.
Macbeth, General of the King's army.
etc

Programs have something similar to an ingredients list: a list of declarations. They also list the resources that a program needs. For programs the resources that matter are places to store data. Having declarations allows the compiler to ensure that enough storage space is available. Declarations in programs have other purposes as well, however. One is to give labels or names to the resources so that they can be referred to later in the program and we will know which resource is being referred to.

Consider another source of algorithms in everyday life: the instructions that come with construction kits (whether children's toys or flat-pack do-it-yourself book shelves). They also normally have a list of parts at the start. However, those parts are also given a label: often just a letter. The purpose of the letter is so that that part can be referred to without ambiguity later on in the instructions. For example, the list of parts might be as follows:

6 x **A**: 200x1000x20mm
2 x **B**: 200x2000x20mm
1 x **C**: 1000x2000x3mm

The instructions then might be:

1. Place part **C** face up on the floor.
2. Place the 2 parts **A** side by side against the edges of part **C**.
3. etc.

Thus declarations give a list of resources needed, but also give names to each resource that can be referred to within the algorithm itself. These names are referred to as **identifiers**: they are used to identify a specific resource. In the shelf construction algorithm above, three identifiers were used: A, B and C. Using letters for identifiers removes ambiguity but make algorithms harder to understand. Looking at the shelf instructions it is not immediately easy to see which pieces of wood are the shelves, which the back and which the sides. We could easily solve this problem by using more meaningful identifiers in our instructions (and also of course in the declarations). For example, the following instructions would be much easier to follow:

Parts
6 x **shelf**: 200x1000x20mm

2 x **side panels**: 200x2000x20mm
1 x **back panel**: 1000x2000x3mm
etc.

Instructions

1. Place the **back panel** face up on the floor.
2. Place the 2 **side panels** side by side against the edges of part **back panel**.
3. etc.

Text books use glossaries and lists of acronyms for a similar purpose to allow a label to be used to refer to something else. A glossary is just a list of unusual terms used in the book. It gives a name for each unusual word or phrase in the book, by looking at the glossary you can find out what is meant by a term. Acronyms are just shortened versions of long phrases. By putting a list of acronyms at the start of a book, we can then use the acronym throughout the book in place of the term.

Variable Declarations
The declarations in programs are not quite the same as those in recipes in that they list a slightly different thing: storage spaces indicating the kind and size. It is as though a recipe also gave a list of storage jars, pots, pans and dishes needed for a recipe at the start. This would not be that silly a thing for them to do even though I have never come across it. On several occasions, part way through a recipe I have run out of pans, because I, for example, used a large pan for a cheese sauce, only to find I needed it later to fit the amount of vegetables I had to cook. Mid recipe I have a panic whilst I transfer things from one pan to another, and wash things – the last thing I need with guests arriving in 15 minutes. The point of declarations is to allow all the things you need to be gathered and organised before you start – precisely to avoid that kind of crisis.

Computers move and manipulate data: patterns of 1s and 0s. All of this data needs to be stored somewhere whilst it is processed. That is what a variable is used for: a storage jar for data. Each jar has an associated identifier: a name as we saw above. It is as though each jar is labelled with a name so that in the instructions of the algorithm, we can refer to each jar by name and be sure of which one we are referring to.

In our Imp computer, declarations would be used to gather the storage Imps together as they were needed. Without declarations, we might get part way through a calculation and find another Imp was needed. If they had not been booked in advance using a declaration they would unlikely to be available – most likely they would be down the Pub and in no state to do anything.

Types
Consider the following list of things.

Red
42
Blue
Bus
a

Car
64
b
c
Bicycle

Now group them into sensible categories. The chances are you grouped them in the same way as I did:
42, 64
Red, blue
a, b, c
Car, Bicycle, Bus
It is arguably a human instinct to divide the world into categories in this way. It also turns out to be very useful in programs – letting the computers into the secret of what our categories are.

Declarations, as we saw, are used to give information at the start of a set of instructions about the resources that will be needed to complete the instructions. For programs the resources are places to store data. Often by data we just mean numbers – a persons age, the salary of a person. However, data can represent more than just numbers: letters of the alphabet (eg representing a students grade) words or sentences (like a person's name) colours (such as the colour to paint Laura Croft's vest), and many more. An important piece of information about data is what is known as its **type**. The type of something in the computer science sense is just what kind of object it is: in fact everything of interest about it apart from its actual value. Common programming examples of types include integers (whole numbers), floating-point numbers (decimals), characters (letters and symbols), and strings (sequences of characters like words or sentences).

The children's game of Categories is based on the fact that we group things together. In the version for young children, it involves the father writing down a category and the child having to think of something that belongs in that category not thought of by another player. Part of the point of the game (at least from the father's point of view) is to teach the child what things are grouped together. Think of a bird: "Penguin". Think of a city: "Sheffield". For older children, the game is made harder by requiring the thing to start with a given letter of the alphabet.

Categorising things allows us to classify the world and draw conclusions about the properties of objects just from knowing their category. If it is a bird then without seeing it I can guess it has feathers as that is one of the properties of birds. In real life the categories blur. If it is a bird then it must be able to fly is not always true for example – categories are used more as rules of thumb. Computers require preciseness however, so the categories they use are more hard and fast. "If it is a character then it definitely does fit into a byte". Knowing the type (i.e. category) of things in the program allows the **compiler** to make appropriate preparations for use of the piece of data such as how much space it will need. It also allows the compiler to check that things are used in the program in the way intended. You do not throw a pig out of an aeroplane and expect it to fly in the real world as ability to fly unaided is not a property of pigs. Neither do you include instructions in a program to do multiplication on strings of letters as ability to be multiplied is not a property of strings.

Let us return to food and recipes. We group kinds of food into categories like "pasta" or "cheese". There are many different cheeses and many different kinds of pasta. What use are these labels? One reason why they are useful is that knowing something is pasta, immediately tells us a whole series of things about it. In particular it tells us which actions can be performed on it in a recipe and also which operations cannot be performed. What can we do to (dry) pasta? We can boil it or bake it in the oven within a Lasagna like dish. What else do recipes do to pasta? My wife looked at me as though I was mad when I asked her that question! Why? Because it would be weird to think of doing anything else. If when we were cooking dinner together I passed her the packet of spaghetti and asked her to "sieve that" she would assume I was losing my grip. She would not get out a sieve. Why? Because you do not perform the action "sieve" on things of type "pasta". Similarly, there are sensible ("grate") and silly ("sieve" again) actions to do on cheese. The types of ingredients can thus be used to check for errors when looking at recipes. If I invented a new type of pasta "Taggliaroni" – just by telling you it was pasta, you would automatically know many of the things you could and should not with it. You could immediately substitute it into a whole range of recipes and get edible results. Similarly, if you know how to make Blackberry and Apple Crumble, then it is a fair bet that Raspberry and Apple Crumble would work too as both Blackberries and Raspberries are berries.

Types in programs serve a similar purpose – they allow errors to be spotted. You can not do arithmetic on colours for example – it is meaningless. You might on the other hand convert a colour to a number, do arithmetic on that and then convert it back to a colour in some way. For example you could arbitrarily decide that Blue converts to 1 and Red converts to 2. Blue could be converted to a number, 1 added to that number. Converting the resulting number back would give you the colour red. Such an operation that converts things from one type to another is known as a **cast** operation.

In an Imp computer, different types correspond to Imps holding different sized boxes. An integer Imp has a box big enough to hold a number. A character Imp has a smaller box big enough only for a character. Variable declarations ensure that the correct mix of Imps is obtained and that each Imp has the correct kind of box. If ever an Imp was given something of the wrong type for their box, they would mangle the result trying to fit it in.

Enumerated Types

Most computer languages come with some predefined types. By this we mean they just know about some values and how they are grouped together – which categories they belong to. These are usually the things that most programs will need to use like numbers and characters. Many languages also allow the programmer to write instructions that create new values and put them into new categories. That is the languages allow instructions for creating new types. What does this involve? If we are to create a new type we will need to give it a name and we will need to say which things are in the type. The simplest way of doing this is just listing them. For example if I was the manager of a consumer electronics shop and had a new trainee assistant, I would have to give him instructions about his job. Suppose he was very slow and so did not know much about televisions and videos. The first thing I might do would be to explain to him about the different things he had to sell – as he would have to explain the choices to the customers. The first thing I might do is say something like:

> "One thing we sell is *video recorders*. We sell the Technics-501, the ThornE54 and the Sony-P45". "We also sell *televisions*. We sell the Sony-T100 and the Ferguson170." etc

What we have introduced to the trainee is two new categories of things: two new type: televisions and videos. We have then explained what those types are by listing the things: the **values** that belong to that type one after another. The values of type television are the Sony-T100 and the Ferguson170, for example. A type defined in this way by listing all the things in the type and giving this collection a name is known as an **enumerated type**. Once the type has been defined, then instructions can be given that use the type (in the above case instructions of how to get a customer to exchange money for one of the items of that type!) Notice however that the type definition is not strictly part of the algorithm itself. It is not an instruction of how to do something. It is rather a declaration of information needed to understand the algorithm. Type declarations are thus similar to variable declarations in this way.

5. If only... (Selection)

"Is all good structure in a winding stair?"
George Herbert, *Jordan* (1633).

Sequencing is not the only form of plumbing used in programming: it is not the only way instructions can be ordered. Another form of plumbing is known as **selection** or **branch statements**. It allows a choice of actions to be performed depending on the situation. By choice here, we do not mean free choice, where you are just given two options and could do either. A branch statement tells you the two alternatives, but it also tells you precisely in which situation you would do each alternative.

Branch Statements
Return to the example of emergency instructions on an airliner, one of the things you are told is what to do if the cabin depressurises. You are told that oxygen masks will drop down from the ceiling. The basic instruction is simple:

1. Place mask over your nose and mouth.
2. Secure with the elastic behind your head.

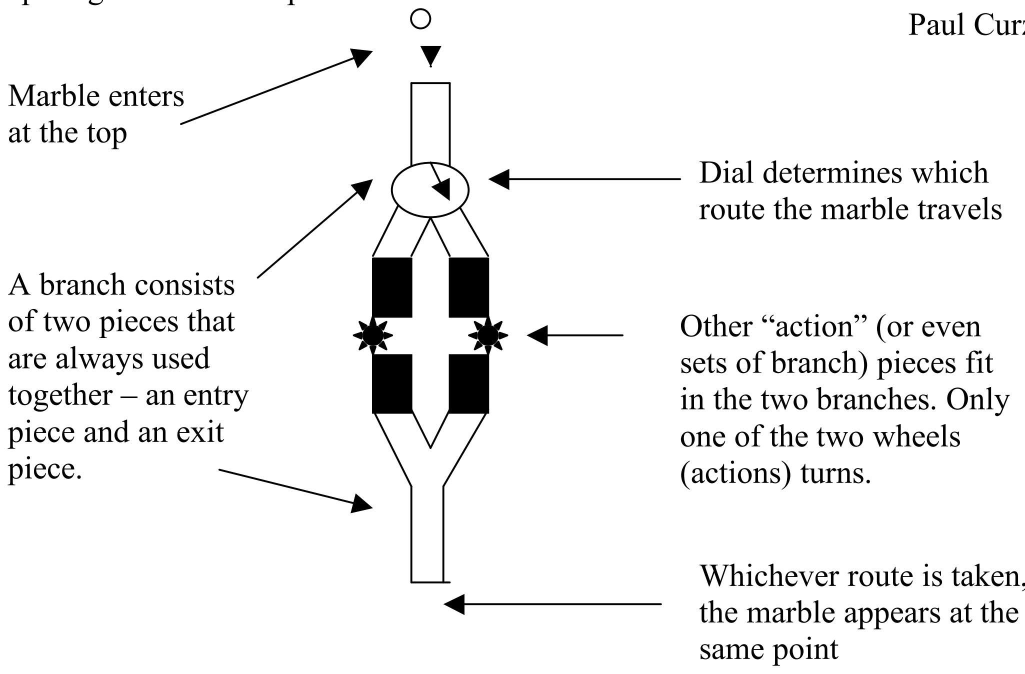
However, the instructions given also consider other situations than the basic one. "if you are travelling with a child then secure your mask first, then secure the mask of your child." Here the air-hostess is giving a second situation from the basic one and telling you exactly what to do if that situation is so. In English, we most commonly use the word "if" when there is a branching situation. We describe the particular situation and say what should be done in that situation.

if **you are travelling with a child** then ...

Here the situation that determines what we do is whether you are travelling with a child. This part of the instruction is not an action statement telling us to do something. Instead it is a descriptive statement. It asks a question about the state of the world. It is splitting the world in to two alternatives. Either you are travelling with a child or you are not. If you are you will need to follow one set of instructions. If you are not you will need to follow a different set of instructions. Both sets of instructions need to be listed. Thus a basic branch statement consists of three things: a question to ask yourself, and two sets of instructions to follow depending on the answer:

```
if you are travelling with a child
then
        1. secure your own mask
        2. secure the mask of the child
else
        1. secure your mask
```

Branch statements are the equivalent to having another part (actually a pair of parts that are always used together) in our marble run. The part has on pipe in, but two out of the bottom. A marble put in the top could appear out of either bottom pipe. Which one it appears out of depends on a two-position switch in the middle. When the switch is switched to the left, the marble goes one way. When it is switched to the right, the marble goes the other way. The part is always used with another part that connects the two pipes back to a single pipe.


Marble enters at the top
Dial determines which route the marble travels
A branch consists of two pieces that are always used together – an entry piece and an exit piece.
Other "action" (or even sets of branch) pieces fit in the two branches. Only one of the two wheels (actions) turns.
Whichever route is taken, the marble appears at the same point

This selection part is not used on its own but with the basic wheel parts. Wheel parts or sequences of them can be connected on each branch. If the marble goes one way it will turn one set of wheels, but if the switch is switched the other way, the marble will turn the wheels on the other branch. Eventually, whichever branch the marble passed down it gets to the join piece and drops out of the same slot at the bottom. In terms of a program, the flow of control passes down one of the branches, and only the actions on that branch are executed. Either way, the flow of control ends up in the same place at the bottom. Of course other wheel parts could be put on the bottom of our construction allowing further wheels to be turned (whichever branch the marble passed down.

We have seen selection already when we were looking at the properties of algorithms.

```
1.    if the shop has Croissants
      then
           buy Croissants
      else
           buy Muffins.

2.    Leave shop.
```

One branch of this selection is the action (wheel) "buy Croissants". The other branch is the action "buy Muffins". The selection plumbing in this example is indicated by the words if...then...else. The switch is the question "Does the shop have croissants?" If the answer to this question is yes (the switch is one way) then we move to the "Buy Croissant" action. If the answer is no (the switch is the other way) we move to the "Buy Muffins" action. In either case we drop off the bottom of the selection and do the next action in sequence – here "Leave Shop". We have combined a selection with a sequence. We will either "Buy Croissants then Leave shop" or we will "Buy Muffins then Leave shop".

Notice that the question is a boolean question. It is really "True or false: the shop has croissants?". There are two alternative answers and two possible actions. This is the situation where boolean calculations are used in programs – to determine which branch to take in a selection statement.

We discussed sequencing – doing actions one after the other – in terms of a relay race. Sequencing is a relay race where on each leg only one person is waiting to take the baton for the next leg. Selection is a relay race where there are two people waiting for the next leg. The baton is only passed to one of them who must then run the leg. The other person (and the other action) waits for another race (i.e. another run of the program) where they may or may not be passed the baton when it comes to their leg. The decision of who to pass the baton to is decided by the answer to the true/false question. If the answer to the question is true at that time then the baton goes to one person. If it is false it goes to the other runner. At the end of the leg, however, the baton gets to the same place whichever person ran the leg.

In the relay race version of computation, compiling a program corresponds to putting all the runners in the correct start positions and ensuring the questions that must be asked are available in the correct places. The race is not being run while this is happening – it is just preparation for the race. Running a program is like starting the race. The actions in the program are runners running legs. The end of execution of the program is when the last runner gets to the finish line. At this point the baton is given up. The race and the program are over.

The Imps handle branch statements by having an instruction that gives a choice of who to pass the baton to next. However they do not have a free choice. In fact the instructions tell them which of the two possible other instruction Imps to hand the baton to. The way this is done is by a boolean instruction.

For example, suppose the Imp computer was to follow the following instructions:

```
if (Bridget's value equals Alf's value)
then Alf gets the number 13
else Alf gets  the number 42.

Return.
```

Three instruction Imps will be allocated to this program – one for each line. The special instruction Imp that must do something more than we have so far seen is the first one. He has as instruction to find out if Bridget is holding the same value as Alf. Notice that this is a boolean expression which will have a true or false answer. When the instruction Imp for this instruction gets the baton, they get copies of the values of Bridget and Alf and pass them to the Equality Imp (who happens to be called Eric). Eric specialises in telling if two things are equal or not. Eric checks if the two things he has been given are equal and if so announces that the answer is true. If they are different he announces that the answer is false. The instruction Imp waits for this pronouncement eagerly because it determines what he does with the baton. If the pronouncement is “true” he passes the baton to the instruction Imp in charge of the “then” instruction who will give 13 to Alf. If the pronouncement is false however, he passes it to the “else” instruction Imp that gives 42 to Alf. Only one of these two instruction Imps actually takes part. The other can remain asleep. The baton is not passed to them. That does not mean that on some later day when the same set of instructions are run, the other branch will not be taken if the values held by Bridget and Alf have changed. Whichever way the baton is passed, it always ends up at the same point. Both the "then" Imp and the "else" Imp on finishing their instruction will pass the baton on to the Instruction Imp whose instruction follows the if-then-else. Here it is a Return statement, but it could be any instruction.

Our Imp computer can thus now do different things depending on the results of calculations or the values of data supplied. It is still however done without the Imps needing to know anything about what the overall task is. They just follow their instructions blindly, each knowing and being able to accomplish one small well-defined task.

Let us consider a more useful example. Suppose we wished our Imp computer to advise students on their exam results – telling them whether they have passed or not. An A is more than 80%, a B is more than 60% and a C is anything else. The student tells the Imp computer their mark and the Imp computer will tell them their grade. Alf will be used to store the mark and Bridget the corresponding grade. The instructions the Imps must follow is as follows:

```
Alf gets the score from the student.
If Alf's value is greater than 80
then Bridget gets the grade A
else if Alf's value is greater than 60
    then Bridget gets the grade B
    else Bridget gets the grade C.
Bridget's value is written on the screen (a large blackboard)
Return.
```

This will need 8 instruction Imps (one for each line) and an expert in determining when a value is greater than another, in addition to Bridget and Alf. What happens when we run this program will depend on the value given to Alf at the start. Let us watch some sample runs.

The first instruction is executed – the student provides a mark (84) and it is put in Alf's box. The baton passes on to the next Imp. Who gets a copy of Alf's value (84) and passes it and the number 80 to the "greater-than" expert. The expertt pronounces that 84 is indeed greater than 80, so the baton is passed to the first then Imp who holds the instruction "Bridget gets the grade A". Bridget is given the letter 'A' to store and the baton is passed to the Instruction Imp for the instruction immediately after the if-then-else statement. That is the instruction to write Bridget's value on the screen. This is done (by another specialist Imp who abseils down the wall of the computer with some chalk and writes 'A' (Bridget's value) on the screen. The baton then passes to the Return and on to M who announces the program terminated.

Another student arrives. This time they give the mark 69 to Alf. Now the answer to the first question is false. The greater than Imp pronounces "false: that Alf's value is **not** greater than 80 this time so instead of the baton passing to the "then" Instruction Imp, it passes to the "else" Imp whose instruction is actually another branch. He must determine the answer to the question "Is Alf's value greater than 60?" The greater than Imp pronounces "true" this time. The baton goes to the second then Imp who gives Bridget the value B. The baton passes on to the instruction after the if-then-else which again results in the abseiling Imp writing on the board. This time however he writes 'B' as that is the value in Bridget's box on this run of the program.

A final student has a mark of 2 (they got their name right). The first "greater than" Imp pronounces true leading to the second greater than Imp being questioned as above. This time however he answers "false". Bridget is given the value C which is written on the board.

Thus the Imps have executed the same program three times, with three different results. In one an A was written on the board, in the second a B was written on the board and in the third a C was written up. In each case the letter written was the correct grade for the mark presented. Branch statements thus give us a way of having the program make decisions based on the data input.

When writing computer programs we have to turn all such decisions into this more confined form. First you must recognise that there is a choice of actions in a situation, then work out what the question is, and finally decide what actions are done when the answer to the question is yes and what actions are done when the answer is no.

Sometimes the way instructions are phrased in English the question appears to be missed altogether. It is still there lurking however. For example, a recipe might say: Add the Pecorino (or Parmesan) cheese.

This really means

```
If you could find the exceedingly hard to find Pecorino cheese
        (that I am only mentioning to impress you)
then Add the Pecorino
else Add Parmesan
```

"For a simpler ice cream replace the coconut milk with milk." (Abensur, 1996) can be rephrased as

```
If you wish a simple ice-cream
then add milk
else add coconut milk
```

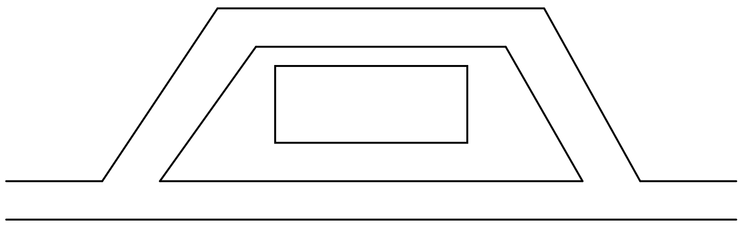
Single-branched if
Selection requires us to work out three things: the question that corresponds to the decision being made; the action(s) if the answer is yes; and the actions if the answer is no. Sometimes in one of the cases nothing is to be done until the branches meet again. This is known as a **single-branched if statement**. We can assume that nothing is done when the answer is yes, as we can always turn the question round to make this so.

The example we saw earlier of determining who goes through to the next round of the Euro football competition:

> *England will qualify for the quarter-finals with Portugal if they win or draw with Romania.*

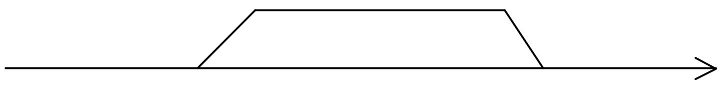
Here the true/false questions is "True or false: England won or drew with Romania". If this is true, this rule states that the action is "England qualify". This rule gives no action if the answer to the question is false. You must move on to the next rule in either case.

A motorway with a service station is a little like this. A driver, drives down the motorway. On seeing the sign for the Service Station, she asks herself the question "Do I need a break". If the answer is yes, she pulls in and has a coffee (the action). On finishing the coffee she pulls back onto the motorway, using the exit slip road. If the answer is no, she just drives on and performs no action. In either case she arrives at the same point to continue her journey. There was no other way out of the service station other than back onto the motorway, a little further along.



Now imagine the journey is a long one (from London to Edinburgh perhaps) and the driver makes the journey several times a year. Each time she makes the drive she will pass many service stations. On different trips she might need a break at different points, so on each time will take a different route. Occasionally she might make the whole journey without pulling in to any service station. If she had diarhea she might stop at every one. This is like a program consisting of a series of if statements placed end to end being executed. The road is the control structure (the plumbing), the service stations are actions (drinking coffee, going to the toilet) and the driver represents the flow of control. Each journey is an execution of the program, and on each different actions are performed depending on the particular situation as it effects the questions asked at each service station.

Suppose a person is walking down the street and comes to a ladder. Some people are superstitious and walk into the road to get round the ladder. An action stepping out into the road is needed. Others (like me) are perverse and make a point of walking under the ladder. Whichever you do once on the other side of the ladder, both people end up at the same point and continue down the road in the same way. No person walks both routes on one journey however (though on different journeys they may take different paths).



As with selection we have two possible routes. The switch in this case is the question "Is the person feeling superstitious". If the answer is yes, they will step out into the road. If the answer is no, they will continue along the pavement performing no action.

In its written form a singly-branched if statement might appear something like the following:

1) **if** the shop has Croissants

```
        then buy Croissants
    2) Leave shop.
```

The question here is does the shop have Croissants. If it does then we buy some if not we are not instructed to do anything by the if statement. In both situations we then Leave the shop – the next instruction after the if statement.

This kind of decision making often appears in instructions but the structure is often hidden within the language. A recipe might include an instruction "Season to taste". This is just a singly-branched if statement in disguise. It means

```
If you like salty food
    then add the amount of salt you prefer
```

Notice if you do not like salty food, you do nothing for this step.

Multiple If Statements
Often there are more than just two alternatives but a whole range. To write instructions that cope with this we can combine together several branch statements. For example, the following is a joke that appeals to my sense of humour spoofing the emergency instructions on airlines. It was supposedly actually announced on a real flight. Here we write it as an algorithm.

```
1. if you are travelling alone
    then
    1.1 secure your mask
2. if you are travelling with a child
    then
        2.1 secure your mask
        2.2 secure the child's mask
3. if you are travelling with two children
    then
    3.1 secure your mask
    3.2 decide which child you love the most
    3.3 secure the mask of that child
    3.4 secure the mask of the other child
```

This algorithm consists of three main instructions (numbered 1, 2, 3) that are performed in sequence. Each is an if-statement. They describe a situation and then say what to do if that situation holds. All three questions will be asked. However, the actions of only those that are relevant will actually be followed. In this case as only one of the questions can be true in a given situation, then only one will be followed. When putting branch instructions in sequence in this way, care has to be taken that all possible cases are covered, otherwise nothing will be done. For example, if following the above instructions, a father travelling with three instructions would not have any instructions to follow as the instructions only deal with the situations of none, one and two children. This is the issue of algorithms being **complete** that we discussed previously. You must cover all the cases.

Another common way to combine branch statements is to **nest** them. What this means is put one *inside* the other, rather than one after the other. For example we could rewrite the above instructions about putting on oxygen masks as follows:

```
1. if you are travelling alone
   then
          A.    secure your mask
   else
          B.    if you are travelling with a child
                then
                      secure your mask
                      secure the child's mask
                else
                      if you are travelling with 2 children
                      then
                            secure your mask
                            decide which child you love the most
                            secure the mask of that child
                            secure the mask of the other child
```

This only has one main instruction: an if statement with two options.

```
1. if you are travelling alone
   then
          A ...
   else
          B ...
```

If you are travelling alone then you do instruction A – you just secure your mask. All the rest of the algorithm is the something different that you do if you are not travelling alone – instruction B. Since you are travelling alone you can ignore it. It is not the instructions that are meant for you this time. You ask one question, take the action and you are finished.

However, suppose you are travelling with a child. You ignore the first part of the instruction as that is only for people travelling alone and instead follow the instructions for people for which this does not apply: you follow instruction B. But here B is actually another branch statement. B is the statement:

We used the example of the instructions for opening an emergency door on an airliner, earlier. On a different plane there were slightly different instructions: a different algorithm. It involved my making an observation (a test) and doing something different depending on the result. Before opening the door, I was instructed to check if there was a fire outside as if so I should use a different door. Only **if** the access was free should I open the door. I should in that case follow instructions similar to the original ones. Finally I needed to make another observation – I was only to use the door if the escape slide was there. This is an if statement inside another one – I only have to worry about the slide if their was no fire – if there was a fire the slides status is immaterial as I am not going to open the door in the first place. We use a nested if statement. The algorithm is as follows:

```
1. If there is a fire outside the door
   then
      1.    Go to another exit.
   else
      1.    Remove the flap covering the handle.
      2.    Turn the handle.
      3.    Pull the door into the plane
      4.    Throw it out the doorway, as far away as possible
      5.    If the slide is out
            then
                1.    Climb out onto the wing.
                2.    Slide down the slide.
            else
                1.    Go to another exit.
```

Notice how the second if statement is part of the first. After each then and else is a mini-algorithm. The observation made as part of the test decides which mini-algorithm to follow, the other one being ignored. Either you go to another exit or you start to open the door – depending on what you see. The instruction to check if the slide is out is in the mini-algorithm that is followed only if there was no fire. This example also shows the distinction between the test and the actions that are carried out. Tests are the observations that appear after the word "if" like "the slide is out". Me looking does not make the slide go out, all I am doing is checking whether the slide is out. In general the act of making an observation does not change the state of the plane or people on it. The test observes what the state is. On the other hand the actions like "Pull the door into the aeroplane" change the plane – it changes the state of the plane from being a plane with a closed door to being a plane with an open doorway. It has an actual effect on the plane itself. "Climb onto the wing" is similar but this time it has an effect on the person – After doing this I am in a different place. The actions change the state, whereas the tests check what the state of something is. An if statement combines them – first making an observation, then based on that observation does one thing or another. Nested if statements require you to make one observation, then depending on the result possibly make other observations to decide what to do next.

The phone systems from hell that some companies use instead of employing real people are giving you instructions in this form. You know the ones where you press a different button depending on what service you want. The message will say something like this:

1. **if** you wish to buy a ticket **then** press 1.
2. **if** you wish to listen to some horrible tinny music **then** press 2.
3. **if** you wish to listen to recorded information that is of no use to you **then** press 3
4. **if** you wish to speak to an operator **then** press 4

This is a sequence of if statements, each with a different question. In each case the action to be performed is to press the button. You only press the button if the answer to the corresponding question is yes. The last possibility is there for completeness – if none of the other if statements covers your situation then this one will – you can talk to a real person! (Real phone systems from hell – do not have this as an option leaving you with nothing to do but hang up). Notice also that because the if statements are followed in sequence you still have to check (ie listen to) all the questions before the one that applies to you. The person who just wants to talk to a person still has to work through all the other questions before finding which button they need to press.

Often in real life the equivalent of this kind of information is given in terms of tables of information. We will see later how tables can be dealt with directly in a more general way. For now we will look at how they can be turned into if statements that have the same effect. Most shop doors have a sign on the door that looks like this.

Monday	8am-8pm
Tuesday	8am-12pm
Wednesday	8am-8pm
Thursday	8am-8pm
Friday	8am-8pm
Saturday	9am-5pm
Sunday	10am-4pm

This notice is allowing you to work out what the opening times are. It says that if the day is Monday then we are open 8am-8pm, if the day is Tuesday then we are open 8am to 12pm, and so on. Writing this out as a sequence of if statements, we get.

```
if the day is Monday        then the opening hours are 8am-8pm.
if the day is Tuesday       then the opening hours are 8am-12pm.
if the day is Wednesday     then the opening hours are 8am-8pm.
if the day is Thursday      then the opening hours are 8am-8pm.
if the day is Friday        then the opening hours are 8am-8pm.
if the day is Saturday      then the opening hours are 9am-5pm.
if the day is Sunday        then the opening hours are 10am-4pm.
```

This is not normally how it is done on doors, but the same information could have been done more compactly using the following table:

Tuesday	8am-12pm
Saturday	9am-5pm
Sunday	10am-4pm
Monday, Wednesday, Thursday, Friday	8am-8pm

This can be written in fewer instructions and is quicker to follow.

```
if the day is Tuesday       then the opening hours are 8am-12pm.
if the day is Saturday      then the opening hours are 9am-5pm.
if the day is Sunday        then the opening hours are 10am-4pm.
if the day is Monday or the day is not Wednesday or the day is Thursday or
the day is Friday           then the opening hours are 8am-8pm.
```

We have grouped together cases that require the same action. This does not actually save us that much but with a slight change it can do. Notice that only one of the if statements will ever be true. The last one is really a catch all – it is saying “and any case I have not covered do the following”. We really want to use an else statement.

However we do not want it to be an else to just one of the other rules: the following would be wrong

```
if the day is Tuesday        then the opening hours are 8am-12pm.
if the day is Saturday       then the opening hours are 9am-5pm.
if the day is Sunday         then the opening hours are 10am-4pm.
                             else the opening hours are 8am-8pm.
```

Can you see the problem? Suppose the day is Tuesday, then the first rule's question is true so we are told that the opening hours are 8am-12pm. However we then move on to the next instruction. Tuesday is not Saturday so we do nothing, but now move on to the third instruction. Tuesday is not Sunday, so the else case comes into action. We are now told that the opening hours are 8am-8pm. We want the else case to apply to all the other ifs as well. That can be done by nesting them.

```
if the day is Tuesday        then the opening hours are 8am-12pm.
else if the day is Saturday  then the opening hours are 9am-5pm.
else if the day is Sunday    then the opening hours are 10am-4pm.
                             else the opening hours are 8am-8pm.
```

Notice that we have added **else** before the second and third **if**. What that means is we ask the first question. If it is Tuesday, then we are told the opening hours and we ignore the else – which is everything else up to the final else. Only if it is not Tuesday do you check if is Saturday, Sunday or none of these.

Problem
Sad Joe's Cafe uses the following Breakfast Price List on the wall. Write an algorithm using if statements like that above that Joe must follow to Charge the correct price for a single meal.

Sausage, Egg and Chips	**£4**
Bacon and Eggs	**£3**
Omelette	**£3**
Blueberry Muffin	**£1**

Here is another algorithm that saves lives that uses a series of if-statements. This version is adapted from a medical encyclopedia (BMA, 1990) but the same algorithm is taught on first aid courses:

First Aid in case of Suffocation:
1. Remove any obstruction.
2. Move the victim into fresh air.
3. **If** the victim is conscious **then** offer reassurance.
4. **If** the victim is unconscious **and** breathing normally **then** place in the recovery position.
5. **If** the victim's breathing is difficult **or** has stopped **then** begin artificial respiration.

This is a sequence of 5 instructions to be performed in order, but where three of the instructions are if statements. The action in each case is only taken if some condition is true. There is only a point offering reassurance to victims that are conscious for example. Only having checked that the person is conscious do you move to the next check – are they unconscious and breathing normally. If so place in the recovery position and then go to the next instruction. Notice that in fact only one of the three conditions will ever be true so only one of the actions will be taken. This means nested if-then-else statements could have been used instead.

Problem
Rewrite the suffocation first-aid algorithm using nested if-then-else statements.

Arguably the order given to do the checks is wrong. If the victim is conscious then things are basically okay – you do not need to offer reassurance instantly. The most important thing to heck is actually the last one as if their breathing has stopped you need to apply artificial respiration immediately or they will die. It should be the first if statement.

Problem
Rewrite the suffocation first-aid algorithm to ensure that the checks are made in order of importance.

6. Play it Again Sam (Iteration and Recursion)

"Oh No Not Again"
The Bowl of Petunias (Douglas Adams),
The Hitch Hiker's Guide to the Galaxy (1979).

General Repetition
Repetition (often called **Iteration**) is important to Computer Scientists because one of the things that computers are very good at is doing the same thing over and over and over again. They increasingly are taking over all the mundane, boring, repetitive tasks that require little intelligence – just an ability to follow instructions blindly. Examples include spraying cars with paint and ensuring we all get paid at the end of the month (or start of the term for students). For computers to just follow instructions blindly, we need a way of saying to do something repeatedly. That is what a **loop** is for.

Let us look at a simple task to examine what makes a loop. At School as a punishment you might have to write lines: writing the same thing over and over again. "Write out 100 times, 'I must not throw chewing gum at the teacher' ", or "Write out 30 times, 'I must not break up my desk and pass the bits out of the window behind the teacher's back' ", for example). The teacher will have told you the sentence you have to write out repeatedly. They will also have told you when you can stop – probably by telling you the number of times you had to write it (perhaps 100 times – or maybe 500 times if your aim with the chewing gum had been accurate). There are thus three basic things we must specify if we are to give an instruction that something must be done over and over again:

- we must make it clear that we do want something to be repeated and not just done once,
- we must clearly say what it is that must be repeated (this is called the **body** of the loop), and
- we must indicate in what circumstances the repetition is to continue and when to stop (this is called the **termination condition** of the loop).

Without knowing the termination condition the body would be repeated forever. Any repetitive behaviour can be written just with those two pieces of information. In computer programming, one of the basic kinds of loop contains just this information and is known as a **while loop**.

Suppose I wish to read all the books on my bookshelf. This is a situation where I am doing a similar thing over and over again. It should therefore be possible to describe it by giving the instructions to be repeated and the instructions on when to stop. The repetitive task (the **body**) is:

1. Pick a book that you have not read.
2. Read the book.
3. Put the book back.

How do we describe the circumstances when I should continue and when I should stop? We most commonly phrase this question the other way round in computing: we describe the situations in which we should keep going. For example, here the **termination condition** is:

Continue provided there are books on the shelf you have not read

Putting these together with an indication that we want something to be repeated:

While there are books on the shelf you have not read **do the following repeatedly**

1. Pick a book that you have not read.
2. Read the book.
3. Put the book back.

Suppose the task is to search for something. For example, suppose I wish to find an article I remember reading about "Sabre-tooth Wombats". I think the article was in the magazine *Dinosaurs Today*, but am not sure. If it was, then it will be in the pile of *Dinosaurs Today* in my loft (I have every one). The task to be repeated is to

1. Take the next *Dinosaurs Today* from the Pile
2. Check whether the article is in it
3. Add it to a discard pile

What is the condition for continuing to search (i.e. do the repetitive task)?

I am not at the bottom of the pile **and** I have not found it yet

When either of the above are not true I would stop. Notice that this is just one of our true/false boolean questions again. Putting them together:

While I am not at the bottom of the pile **and** I have not found it yet **do the following repeatedly**

1. Take the next *Dinosaurs Today* from the Pile
2. Check whether the article is in it
3. Add it to a discard pile

The **termination condition** that tells us when to continue and when to stop can thus be thought of as a question meaning "Do I continue". In the first example above, the question is

"Are there still books on the shelf I have not read?"

If the answer to this question is YES then I must continue. If the answer is NO then I stop. Similarly in the second situation we are asking the question

"Am I not yet at the bottom of the pile **and** have not yet found it?"

We ask this question once every time we have done the repetitive task to see if we need to do it again.

Remember the maze puzzle we looked at when discussing doing things one after another (sequencing). We came up with the following algorithm to solve it:

To get to the centre of the maze:

1. Enter the maze by the gate following the path ahead.
2. Turn left following the path ahead.
3. Go straight across the junction following the path ahead.
4. Turn left following the path ahead.
5. Go straight across the junction following the path ahead.
6. Stop at the cross.

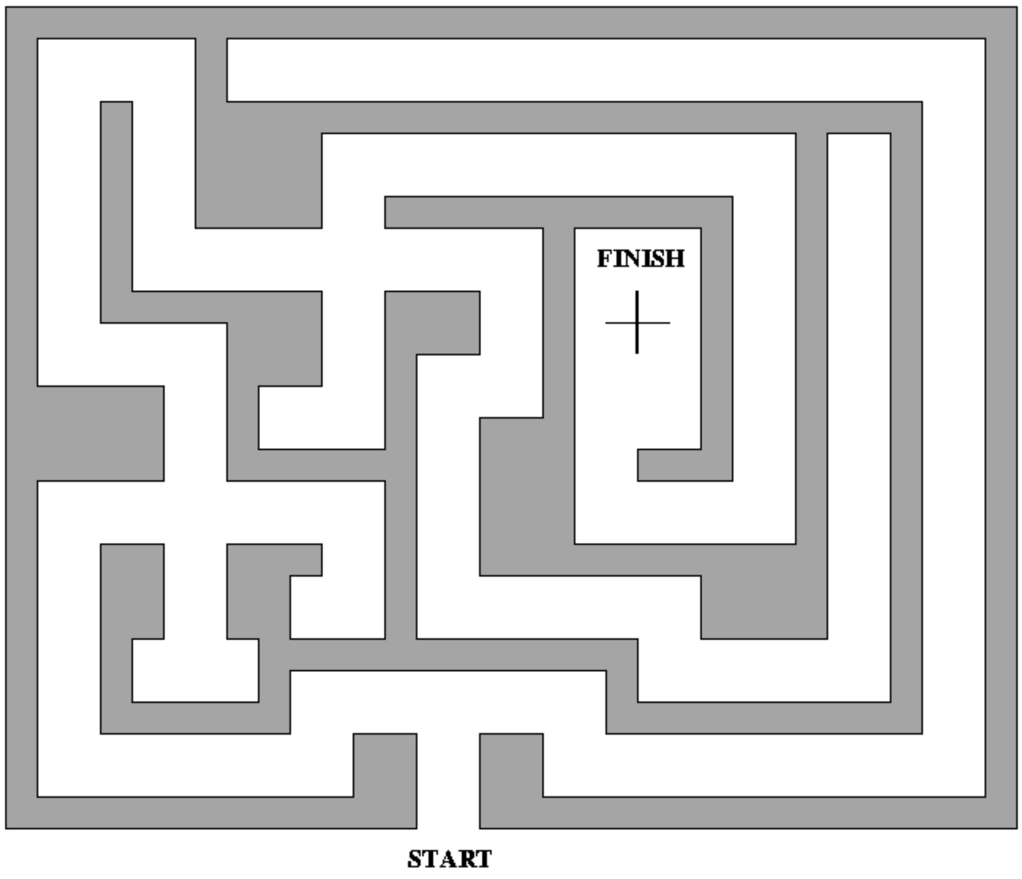
We can actually simplify this algorithm using a loop. It contains some repetition.

Problem

Which instructions are being repeated in this algorithm?

Why write the same instructions out several times if we can avoid it? This maze is small so the amount of repetition in the instructions is not great – but imagine if the maze was much bigger with more junctions, then if there was repetition it would be very worthwhile avoiding repeatedly writing the same instructions. Also if we can make the instructions simpler they will be easier to learn (remember you were learning them for a race). See if you can rewrite the above algorithm using a while loop. We will go through the answer to the above maze later, but if you find it difficult read on.

Before we look at the above maze together, lets look at a simpler version that is easier to see as a loop.



Solve this maze by drawing a line, then try and write an algorithm just like for the last one, before reading on.

Here is my solution (without a loop).

1. Enter the maze by the gate following the path ahead.
2. Turn Left following the path ahead.
3. Turn Left following the path ahead.
4. Turn Left following the path ahead.
5. Stop at the cross.

If you were trying to memorise this I would hope you would notice the pattern and memorise a simpler version once you have entered the maze: “At every junction I must turn left and then follow the path ahead". Can we write it as an algorithm in this way without writing out "Turn Left following the path ahead" over and over again? We must ask ourselves two questions to devise the loop. The first question is "How do we know when to continue?" We continue as long as we are *not at the cross*.

The second question we must ask is: "what in the above is being repeated?" It is the instruction:

1. Turn Left following the path ahead.

We can write the algorithm as:

```
1. Enter the maze by the gate following the path ahead.
2. **While** not at the cross **do the following repeatedly**
       Turn left following the path ahead.
```

This says that what we should do is first enter the maze and follow the path ahead. We then must follow the loop instruction. If we follow this we expect to repeatedly turn left and follow the path ahead. However the first thing it tells us to do is to check if we are at the cross because if so we have finished. If not then we turn left and follow the path ahead. We have not finished there however. Unlike sequencing, with a loop we do not just follow instructions down the page, stopping when we get to the last one. With a loop, we want to repeatedly do the instruction that is inside the loop "Turn left following the path ahead" in this case. On getting to the bottom of the loop we go back to the top and ask the question again. Perhaps we are at the cross in which case we could stop. If not we will need to continue and follow the instructions in the loop again. We therefore check if we are at the cross to decide whether we have finished – we go back to the top of the loop. Only if we are at the cross do we carry on with the rest of the algorithm.

This algorithm is slightly more general than the original. It wont get us to the centre of any maze, but it would get us to the centre of any maze where you must turn left at any junction, irrespective of the number of junctions.

Let us now return to the original maze. In it you sometimes had to turn left and sometimes go straight on. Here it is again.

To get to the centre of the maze:

1. Enter the maze by the gate following the path ahead.
2. Turn left following the path ahead.
3. Go straight across the junction following the path ahead.
4. Turn left following the path ahead.
5. Go straight across the junction following the path ahead.
6. Stop at the cross.

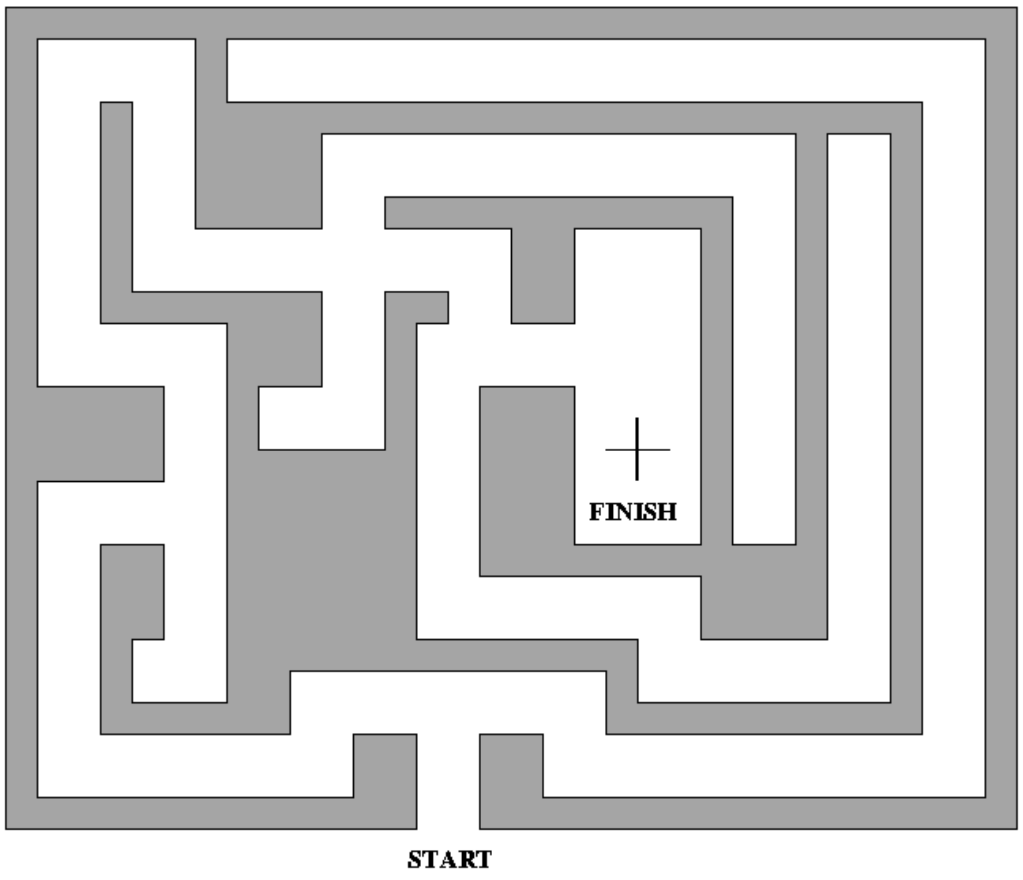
What is the thing that we want to do repeatedly. This time it is two instructions: turning left then going straight on at the next junction. We stop, as before, if we are at the cross. Using a loop, the algorithm is:

To get to the centre of the maze:

1. Enter the maze by the gate following the path ahead.

```
2. While not at the cross do the following repeatedly
        1. Turn left following the path ahead.
        2. Go straight across the junction following the path ahead.
```

What we have here is a loop where its body is not just a single instruction but is a sequence of instructions. We can put any combination of control structures inside a loop. For example, we could put an if statement inside a loop. Here is a maze where that might be useful. Write an algorithm that solves it.



Writing out a solution to this in full we get:

To get to the centre of the maze:

1. Enter the maze by the gate following the path ahead.
2. Turn left following the path ahead.
3. Turn left following the path ahead.
4. Go straight across the junction following the path ahead.
5. Turn left following the path ahead.
6. Stop at the cross.

The secret of this maze is that whenever you get to a T-junction you turn left and whenever you get to a crossroads you go straight on. In other words, if you are at a T-junction turn left else go straight on. We can use this observation to rewrite the instructions so that there is repetition. The above algorithm can be replaced by the slightly more general one below.

To get to the centre of the maze:

1. Enter the maze by the gate following the path ahead.

```
2. If at a T-junction
   then turn left following the path ahead
   else go straight across the junction following the path ahead.
3. If at a T-junction
   then turn left following the path ahead
   else go straight across the junction following the path ahead.
4. If at a T-junction
   then turn left following the path ahead
   else go straight across the junction following the path ahead.
5. If at a T-junction
   then turn left following the path ahead
   else go straight across the junction following the path ahead.
6. Stop at the cross.
```

Convince yourself that this still works by using it to get to the centre of the maze. Now it is obvious what the repetition is. We can put the if statement in a loop to avoid repeatedly writing it out.

To get to the centre of the maze:

```
1. Enter the maze by the gate following the path ahead.
2. while not at the cross do the following repeatedly
       if at a T-junction
       then turn left following the path ahead
       else go straight across the junction following the path ahead.
```

Again convince yourself that this still works – you check if you are the cross and if not check if you are at a T-junction taken the specified action depending on the answer. Then you do it all again, stopping if you are at the centre of the maze.

Problem
Does this algorithm work for all three mazes that we have looked at?

We described sequencing and selection in terms of relay races with a baton being passed. What sort of relay race is iteration? It is a race round a circular track where at the start a termination question is asked. If the answer to the question is false then the race (the execution of the loop) ends. If the answer is true the runners must do another lap. When the baton gets back to the start line again the question is asked again. The runners keep doing laps until the answer to the question tells them they can stop. Our earlier relay races were always the same fixed length. A race containing a loop could keep going for any length – it all depends on the question. Notice the question must be something whose answer could change as the race progresses. If it always has the same "true" answer, the runners would keep going for ever (or at least until they collapsed – the equivalent of the computer crashing as it runs out of resources to continue).

The Imps would deal with the loop in a very similar way to branch statements. The instruction Imp responsible for the question either passes the baton to the instruction Imp responsible for the instruction immediately after the whole while loop, or to the Imp whose instruction is the first of those to be repeated. The Instruction Imp responsible for the last instruction to be repeated always passes the baton straight back to the Imp responsible for the question. It is only that Imp that has the power to terminate the repetition.

Here is an algorithm for treating heatstroke that saves lives, adapted from BMA (1990). Can you identify the loop in it and say what the repeated action is and what is the termination condition?

1. Move the victim to a cool place.
2. Remove clothing.
3. Place the victim in the sitting position.
4. Support the head and shoulders using pillows.
5. Cover the victim with a wet sheet.
6. Fan the victim with a magazine until the temperature drops to 38C.
7. Seek medical help immediately.

The loop is in line 6. What are we told to do repeatedly? "Fan the victim with a magazine". What is the condition we have to keep checking to determine if we can stop? "The temperature drops to 38C". Rewriting line 6 we get

```
While the temperature is above 38C do the following repeatedly
    Fan the victim with a magazine.
```

All loops can be characterised as above with a termination condition and body. However, certain kinds of termination question come up over and over again and so we can characterise two more specific kinds of loop. In particular, **counter-controlled loops** and **sentinel-controlled loops** are common.

Counter-controlled Loops

One of the most common forms of loop occurs where you know in advance how many times the repetition must be done. The example of the teacher giving a pupil a 100 lines is an example. The pupil knows exactly how many times they must write the sentence before they start. If they have any sense they will keep a count of how many times they have written the sentence: perhaps keeping a tally (making marks in groups of five each time they write a line). When the tally gets to a 100 they stop. Without a counter of some kind they would have no way of knowing when to stop. Doing a task a fixed number of times whilst keeping a count is known as a **counter-controlled loop**.

What do we do repeatedly? We write the punishment line, but we also make another tally mark. What is the termination condition? We continue while the tally is not 100.

```
while the tally is not showing 100 do the following repeatedly
    Write "I must not pick my nose in class".
    Add one to the count by making another tally mark.
```

Here is a dice game. Each player throws the dice 5 times. The player who gets the highest score when all their throws are added together, wins the round. This is an example of a counter-controlled repetition. On each repetition the player will throw the dice, and add their score to the total but they will also add one to their count (perhaps by putting up an extra finger). The termination condition that tells them when to stop is when their count has got to five (or they have run out of fingers).

Problem

Write out the instructions to this dice game in the form of a while loop.

The simplest counter-controlled loop is one where the whole point is just to count. The counting is the repeated task. The game of hide and seek is an example. The child doing the seeking must cover their eyes then count to 10 then shout "Coming, ready or not" then look for the other children. The counting part involves adding one to the last number you thought of.

```
while the number you are thinking is not 10 do the following repeatedly
        add one to the number you are thinking of.
```

What number are you thinking of to start with? One presumably. We ought to say that in our instructions as otherwise a devious child might count "9, 10. Coming ready or not" and could justifiably say that they were not cheating if those were the instructions given. We need to **initialise** the counter: say what its first value is. In fact strictly we should have done this in each of the above examples too – make sure our tally was zero when we started writing lines and making sure that our fingers were showing zero before we started rolling dice.

```
1. Think of the number 1.
2. while the number you are thinking is not 10 do the following repeatedly
        add one to the number you are thinking of.
```

The full instructions for hide and seek are:

```
1. Think of the number 1.
2. while the number you are thinking is not 10 do the following repeatedly
        Add one to the number you are thinking of.
3. Shout "Coming, ready or not".
4. Look for the other children.
```

Notice that these instructions do not tell you to shout "Coming, ready or not" repeatedly. That instruction is not part of the loop. It comes after the loop and is not in the body. It is the instruction you follow after the loop has finished: after the termination condition allowed you out of the loop. You will only shout when the number you are thinking of gets to 10.

All counter controlled loops have the basic elements seen in the above examples:

```
Set the counter to its initial value.
while the counter is not the final value do the following repeatedly
        Do the tasks that are to be repeated the known number of times.
        Change the counter.
```

For example,

```
Set the counter to zero.
while the counter is not 100 do the following repeatedly
        Write "I must not point my gun at Jimmy in class".
        Add 1 to the counter.
```

In the above examples we always added one to the counter when we changed it. That makes us count upwards. We could equally well count down too. Then we would set the counter to the number we wanted to count down from. We would subtract one from our count each time and the number we would stop at would be 0.

```
Set the counter to 10
while the counter is not 0 do the following repeatedly
        Write "I must not set fire to Sally's hair in class"
        Subtract one from the counter.
```

There is a whole genre of children's nursery rhymes that are loops like this. The most obvious is

There were ten in the bed and the little one said, "Roll over, Roll over"
So they all rolled over and one fell out.
There were nine in the bed and the little one said, "Roll over, Roll over"
So they all rolled over and one fell out.
etc
There were none in the bed and the little one said "Good Night".

This rhyme is just describing a series of actions done in a given order. Re-interpreting it as an algorithm and writing it as a loop we get:

```
1. Ten get into bed.
2. while there are any in the bed do the following repeatedly
    1. The little one says "Roll over".
    2. They all roll over.
    3. One falls out.
3. The little one says "Good night".
```

This has the same form as the previous example. We initialise the number in the bed. Our termination condition is that this number is not zero. Some actions occur repeatedly and after each time we reduce the number (in the bed) by one. It is a counter controlled loop that counts down.

Problem
Think of another nursery rhyme that is a counter-controlled loop, and rewrite it as one.

When playing darts, each person has three darts. For each of their turns, they have to get the highest score they can with just three darts. This is a repeated task where you know in advance that you will do the task three times. It is a counter-controlled loop where the darts themselves act as a counter that is counting down – start with three darts and when you have no darts stop! What is being repeated? Throw a dart, then subtract the score from your total.

Problem
Write the instructions for throwing darts as a counter-controlled loop.

Sentinel-controlled Loops
Consider the game of Dice Cricket. A special Dice has on its sides the numbers 0, 1,2,4 and 6 and on the last side is written "OUT". One player acting as the batsman

repeatedly rolls the dice, adding up the scores obtained which correspond to the number of runs scored on that bowl. If they roll "OUT" then that batsman is out and it is the end of their turn. Here the termination question is

"Is the dice roll OUT"

The sentinel value is OUT showing on the dice.

To give an algorithm with sentinel controlled repetition we need to indicate that some instructions need to be repeated, what the question is and which instructions exactly we want to be repeated while the answer to the question means carry on. We can write it out similarly to above. For example the above game could be written out as follows:

```
while the dice is not showing OUT do the following repeatedly
        1. Add whatever is showing on the dice to the current score.
        2. Roll the dice.
```

We roll the dice. If the dice shows out we stop. If it does not, then we add the dice value to the score and roll again. We then repeat this all again and keep doing so until the dice does show stop, at which point we have finished the while instruction and carry on with any subsequent instructions. If we tried to play the game following the above rules alone, they would not quite work. Can you see what is wrong? If not get a dice and do exactly what the instructions tell you and then try to correct the instructions above before reading on.

There are two problems. The first is that we have not said what the score is at the start. Do I start with a score of 100 or of 0? Understanding the game I know that the score starts at 0, but the algorithm does not say that, so somebody who was trying to learn the rules from my instructions would be confused. In fact if we were playing a series of rounds of the game then we might actually want the score to be its previous value. For now we will assume we are only playing one game so zero is the correct initial score. We need to add an extra instruction "Set the score to 0". Notice this is something that we only want to happen once so we need to make it clear that it is not part of the repetition, but comes first.

```
1. Set the score to zero.
2. while the dice is not showing OUT do the following repeatedly
        1. Add whatever is showing on the dice to the current score.
        2. Roll the dice.
```

This still has a problem however. We set the score to zero, then go into the loop, asking if we should do the instructions to be repeated or stop straight away: is the dice showing OUT? However, we have not rolled the dice yet! We have not got to the instruction that tells us to do that yet! We need to roll the dice once first before we do the repetitive part, to get us started.

```
1. Set the score to zero.
2. Roll the dice.
3. while the dice is not showing OUT do the following repeatedly
        1. Add whatever is showing on the dice to the current score.
        2. Roll the dice.
```

Both these extra instructions are **initialisation** instructions. They are needed to set initial values for the things the loop manipulates. Most loops need something initialising to work and they are easy to forget when writing instructions, so it is an important thing to double check.

The game of Pass-The-Pigs is similar. This game involves tossing a pair of special Pig shaped dice. They can land on their feet, on their back, on either side or even occasionally on their snouts. Different positions score different amounts, but if a particular position of both pigs on the same side is rolled, the players turn ends. Here the termination question is

"Are the pigs both on their sides"

(In fact the termination question is slightly more complicated as the turn could also end by the player deciding to give up and so keep their score).

Darts has a similar termination condition. There each player takes turns throwing 3 darts and subtracting the score from their total. The game (and so the repetition) ends when the score hits exactly 0. The termination condition is thus

"Is the score of the current player 0"

These termination conditions have the same form: they all ask

"Has a particular value arisen".

In motor racing, drivers repeatedly lap a circuit. On each circuit the pit crew hold up boards to give information to the driver. One of the situations in which a driver can be made to stop is if the information on the board tells him to come into the pits for a tyre change. Here the termination condition is

"Does the board on this lap say Change Tyres"

The particular value being checked for differs completely in each case but the form of the question is the same. Because this kind of termination condition arises frequently the corresponding kind of loop is given a special name: a **sentinel-controlled loop**. The special value that if it arises causes the repetition to stop is called the **sentinel value**.

Sentinel values are normally values that are arising from outside the system (rather than in the Darts case where it is generated by a calculation done inside the loop). This means that there must be some value that arises in the same way as the others but that is not needed to hold an actual value. For example, in dice cricket, if we only had a six-sided dice and wanted batsmen to be able to score 0,1,2,3,4 or 6, then there is no side of the dice left to be the sentinel. If we pick any of the values to be a sentinel, it can no longer be used as a score as then we would not be able to tell whether on a given roll it should mean the score or out. We must always "waste" one value that is not treated in the same way as the others.

Non-terminating Loops

The following appeared on the grave stone of Catherine Alsopp who hanged herself:

"Don't mourn for me now, don't grieve for me never,
For I'm going to do nothing for ever and ever" (quoted in Lovric 2000)

Suppose we wanted to give precise instructions to do nothing for ever and ever. How would we do it? Is it possible using one of our loop constructs? it clearly is a repetitive thing we want to be done. What is it that we wish to be done over and over? Nothing! Under what circumstances do we keep doing it? Always! Our loop will be something like:

```
    While true do the following repeatedly
        Do nothing.
```

We needed a test to mean "always" and used "true" for this. How does that work? Well we want to always keep going and the loop keeps going when the test is true. So if we always want it to keep going we just use "true" as the test. We are basically seeing "While true is true do the following repeatedly". Since true is always true the answer to the question is always yes. This kind of loop with a test that is always true is known as a **non-terminating loop**. Usually non-terminating loops are bugs – run-time errors in the program. However sometimes you do never want a program to stop (e.g. it might be useful if a program controlling a heart pace maker would keep going forever).

A Harder problem: Loops inside loops

Back in the first chapter we looked at a puzzle to exchange the positions of two sets of pieces on a board of seven squares, either by sliding or jumping pieces. Here is the algorithm we came up with:

A 15-step algorithm for solving the puzzle is as follows:

1. Move the piece in square 2 to square 3.
2. Jump the piece in square 4 to square 2.
3. Move the piece in square 5 to square 4.
4. Jump the piece in square 3 to square 5.
5. Jump the piece in square 1 to square 3.
6. Move the piece in square 0 to square 1.
7. Jump the piece in square 2 to square 0.
8. Jump the piece in square 4 to square 2.
9. Jump the piece in square 6 to square 4.
10. Move the piece in square 5 to square 6.
11. Jump the piece in square 3 to square 5.
12. Jump the piece in square 1 to square 3.
13. Move the piece in square 2 to square 1.
14. Jump the piece in square 4 to square 2.
15. Move the piece in square 3 to square 4.

This was the answer for the puzzle where there are three pieces of each colour. However, this is just one of a whole family of similar puzzles with different sized boards. For example, while on holiday in Dorset I came across a wooden version called "Puffin Round Up" being sold by Puffin Wooden Games (Portland, Dorset DT5 2LN). It is identical to the one we looked at, but has 5 pieces of each colour on a board of size 11. It adds the extra rule that pieces cannot be moved backwards. Our algorithm above never moves a piece backwards anyway so that rule makes no difference. We can similarly add a rule a piece cannot jump over another piece of the same colour without it being a problem as none of our moves do that either. The numbers 3 and 5 are not special. You could also have a version with 20 pieces of each colour or even a 100. In fact for each number there is a version of the puzzle with that number of pieces of each colour. Do we need to devise a new algorithm for each version? Or can we write one set of rules that can be followed and would work whatever the version of the puzzle we are trying to solve? It turns out that the algorithm for each version is very similar. Using loop instructions it is possible. See if you can do this before reading on (its quite hard). HINT: A good problem solving approach to general problems like this is to try some specific cases first, looking for patterns. Only once you understand the answers for the individual cases is it worth trying to solve the general case: try and write algorithms for the versions of the game with 4 and 5 pieces and see if you can spot the pattern common to all.

Here is the algorithm for the puzzle with 4 black and 4 white pieces. It looks similar to the original. What is the common pattern they both share?

1. Slide the white piece.
2. Jump the black piece.
3. Slide the black piece.
4. Jump the white piece.
5. Jump the white piece.
6. Slide the white piece.
7. Jump the black piece.
8. Jump the black piece.
9. Jump the black piece.
10. Slide the black piece.
11. Jump the white piece.
12. Jump the white piece.
13. Jump the white piece.
14. Jump the white piece.
15. Slide the black piece.
16. Jump the black piece.
17. Jump the black piece.
18. Jump the black piece.
19. Slide the white piece.
20. Jump the white piece.
21. Jump the white piece.
22. Slide the black piece.
23. Jump the black piece.
24. Slide the white piece.

To make the pattern more obvious we will change the way we write the algorithm to get rid of all the numbers of board positions: it will be the same algorithm just a less precisely written version. We will concentrate for now on the three piece version. Notice that given we are not allowing a piece to move backwards towards its original position, at any time there are only four possible moves: Jump a black piece over one of the other colour, Jump a white piece over one of the other colour, Slide a white piece forward, and slide a black piece forwards. Rewriting our solution algorithm using those instructions instead of one giving board positions (each instruction will still be referring to a unique unambiguous move, but it we are now leaving the player to work out where the piece to move actually is). The algorithm becomes:

1. Slide the white piece.
2. Jump the black piece.
3. Slide the black piece.
4. Jump the white piece.
5. Jump the white piece.
6. Slide the white piece.

7. Jump the black piece.
8. Jump the black piece.
9. Jump the black piece.
10. Slide the white piece.
11. Jump the white piece.
12. Jump the white piece.
13. Slide the black piece.
14. Jump the black piece.
15. Slide the white piece.

Follow this algorithm to check you understand it and it does work. Then think about its repetitive pattern. What is the pattern?

The pattern is that it alternates a series of jump moves with a single slide move of the same colour. The number of jumps increases by one each time in the first part of the algorithm and then decreases by one in the second part. The colour also alternates- first you slide/jump white pieces, then you slide/jump black pieces, then go back to white pieces and so on. There is a pattern there but it is all really complicated, so if we are going to get to grips with it we will need to make things simpler for ourselves. One way to do this is to write it out in a way that makes the pattern even more obvious. Another good problem solving technique is to split the problem into smaller parts and tackle each in turn. First let us make the pattern more obvious. We decided that the first part of the pattern is that it repeats a series of jumps and a slide move. Let us make that more obvious by writing each series of jumps as a single instruction.

1. Slide the white piece.
2. Jump 1 black piece.
3. Slide the black piece.
4. Jump 2 white pieces.
5. Slide the white piece.
6. Jump 3 black pieces.
7. Slide the white piece.
8. Jump 2 white pieces.
9. Slide the black piece.
10. Jump 1 black piece.
11. Slide the white piece.

Here when I give an instruction like "Jump 2 white pieces". I mean do jumps in a row (though they will be different pieces jumping. Each of those instructions is actually a counter controlled loop: we are doing something (a jump) a known number of times. We will return to that problem later – it is a smaller problem we can solve separately. First let us see if we can replace the above algorithm as it is written into one or more loops. There is obviously something being repeated – jump-slide pairs so we ought to be able to replace the above by something a bit like:

```
while .... do the following repeatedly
        Jump ...
        Slide ...
```

That is repeatedly do a jump followed by a slide move (or maybe it is the other way round). But writing out the detail is tricky – and what is the continuation condition?

One problem is that the number of slides first increases and then decreases again. We have seen counter controlled loops where we have a counter going up and others where we have a counter going down. But how do you do both? The answer is to use that problem solving strategy again:

If the problem is difficult break it into smaller bits that you can solve.

Here, we can break it into two parts: the count up and the count down part. The count up part involves doing the same puzzle but where the aim is to get the pieces half way so that they alternate: black piece-white piece (see the diagram). Let us suppose we were set the puzzle just to get to that position, the algorithm would be just the first half of ours above:

1. Slide the white piece.
2. Jump 1 black piece.
3. Slide the black piece.
4. Jump 2 white pieces.
5. Slide the white piece.
6. Jump 3 black pieces.

Now what is the best way to think of the pairings: a slide followed by a jump or a jump followed by a slide? You could do it either way. Given how we have written it, the most obvious is as a slide followed by a jump. (Surprisingly it turns out it is easier to do it if you think of it as a jump followed by a slide so we will look at that in a moment). Now it is looking more like something we can sort out with a counter controlled loop. However we still have a problem: the colours alternate. Here is another problem solving tip.

**If the problem is too difficult solve an easier version of it first.
Only then go to the original harder version with you improved understanding.**

So let us ignore the colours for now – if we cannot solve it without the colours we will not solve it with them. Our algorithm becomes:

1. Slide the piece.
2. Jump 1 piece.
3. Slide the piece.
4. Jump 2 pieces.
5. Slide the piece.
6. Jump 3 pieces.

Now it is much easier to see how we turn this into a loop: it is just a counter controlled loop! Write it out yourself before reading on.

We stop when the counter gets to four (there is no "Jump 4 pieces"), and what we do repeatedly is a slide followed by a jump.

```
Set the counter to 1.
While the counter is not four do the following repeatedly
    Slide the piece.
    Jump counter pieces.
    Add 1 to the counter.
```

However, that is not quite the algorithm we want as it does not say which coloured piece to slide or jump each time. Looking back to the earlier version, the first time we wanted to slide a white piece, then jump black pieces. However we cannot just put that in the algorithm:

```
Set the *counter* to 1.
**While** the *counter* is not four **do the following repeatedly**
    Slide the white piece.
    Jump *counter* black pieces.
    Add 1 to the *counter*.
```

This does not work as the second time round the loop we want to slide black, then jump white! If something varies, then we need a variable to represent it! Here the easiest thing is to have two that we will call: *Slide Colour* and *Jump Colour*. As we saw above, to start with we want the Slide Colour to be white and the Jump Colour to be black. We initialise them as such.

```
Set the *Slide Colour* to white.
Set the *Jump Colour* to black.
Set the *counter* to 1.
**While** the *counter* is not four **do the following repeatedly**
    Slide the piece with colour *Slide Colour*.
    Jump *counter* pieces of colour *Jump Colour*.
    Add 1 to the *counter*.
```

That only partially solves the problem though: the colours still do not alternate. We need to add extra instructions into the loop, to make them alternate. If the Slide Colour is white then we want to make it black for the next time round the loop and vice versa. We are doing one of two different things depending on the current colour: so we need an if statement.

```
**If** the *Slide Colour* is currently white
**then** Set the *Slide Colour* to black.
**else** Set the *Slide Colour* to white.
```

Notice how whichever colour the Slide Colour is currently this instruction switches it to being the other one. The Jump colour needs to be changed in exactly the same way. We need to add these lines to the end of our loop, ready for the next time round the loop.

```
Set the *Slide Colour* to white.
Set the *Jump Colour* to black.
Set the *counter* to 1.
**While** the *counter* is not four **do the following repeatedly**
    Slide the piece with colour *Slide Colour*.
    Jump *counter* pieces of colour *Jump Colour*.
    Add 1 to the *counter*.
    **If** the *Slide Colour* is currently white
    **then** Set the *Slide Colour* to black.
    **else** Set the *Slide Colour* to white.
    **If** the *Jump Colour* is currently white
    **then** Set the *Jump Colour* to black.
    **else** Set the *Jump Colour* to white.
```

That is now our algorithm for doing the first part of the puzzle – just getting the pieces to halfway. The second part is similar though.

1. Slide the white piece.
2. Jump 2 white pieces.
3. Slide the black piece.
4. Jump 1 black piece.
5. Slide the white piece.

It is just a counter controlled loop counting down doing slide-jump pairs repeatedly, then finishing with an extra slide move after the loop.

The pattern is that it alternates a series of jump moves with a single slide move of the same colour. The number of jumps increases by one each time in the first part of the algorithm and then decreases by one in the second part. The colour also alternates- first you slide/jump white pieces, then you slide/jump black pieces, then go back to white pieces. With more pieces this

Recursion

Fanny Robin, one of the characters in Thomas Hardy's novel "Far from the Madding Crowd" (Hardy, 1874), suffering from exhaustion, is trying to walk to Casterbridge and safety. Eventually her exhaustion overcomes her and she collapses. After lying in the road for 10 minutes or more, she struggles up again. Seeing the lights of Casterbridge she calculates how far she must still go and it seems desperately far:

> "*Five or six steps to a yard, ... I have to go seventeen hundred yards. A hundred times six, six hundred. Seventeen times that. O pity me, Lord*"

Rather than give up she comes up with a way of beating this seemingly impossible problem.

> " *'I'll believe that the end lies five posts forward and no further and so get strength to pass them.'*
> *She passed five posts.*
> *'I'll pass five more by believing my longed-for spot is at the next fifth'*
> *She passed five more.*
> *'It lies five further.'*
> *She passed five more.*
> *'But it lies five further.'*
> *She passed them.*"

In this way she drags herself towards her destination. Her approach to problem solving is one that is widely useful. Rather than trying to solve a seemingly impossible problem in one go, she devises a way of breaking it down into a series of identical but increasingly smaller problems that are easily achievable: walking five posts further down the fence. After each smaller problem is solved she is left in the situation of solving an identical looking problem: her destination is still an impossibly long way away. However, it can be attacked in the same way as the original problem was. Gradually her problem is solved.

Recursive problem solving uses this trick. Solve a problem by finding a simpler version of the same problem together with a way of converting your problem into that simpler one. **Recursion** is about having a series of things that are the same except for some property that gradually gets simpler due to small identical steps being taken. For

Fanny Robins, dragging herself along the highway, the property that was getting smaller was the distance to her destination. Otherwise the problem looked the same: an impossibly long line of fence posts. The individual step that made the property smaller was the passing of five posts.

The following poem based on a quote of Swift (the author of *Gulliver's travels*) gives a feel for the idea of recursion.

> *Great fleas have little fleas*
> *Upon their backs to bite 'em*
> *And little fleas have lesser fleas*
> *And so ad infinitum*
>
> Augustus De Morgan (quoted in Gardner, 1977)

Toy robots are all the rage at the moment. I have therefore just invented a robot, called Robbie, that can climb down a flight of stairs, or at least so far I have got it to go down a single stair. He can also do recursive problem solving to solve big problems by breaking them down into smaller but similar problems. The basic things I originally taught him to do was to go down a single step. Robbie also knows how to stop and knows how to check when at the base of a flight of stairs. I stand him at the top of the stairs tell him to get to the bottom of the stairs...and hold my breath. Does he know enough to recursively solve the problem of getting down a whole flight of stairs? Robbie thinks of his state at the top of the steps and his problem. How does he see the problem ahead? He sees a long flight of stairs ahead of him. He does not know immediately how to descend a whole flight of stairs. If he is to come up with a recursive solution he needs to find a way of converting the problem into a similar problem that looks the same but is a little bit simpler. What does he know how to do that will leave it with a similar situation? Well he can take a step. What will his problem be then? He will see a long flight of stairs ahead just as before, but the new problem is simpler – the flight of stairs are not quite as long. How can he get down the stairs? He can take a step and then tackle the remaining flight of stairs in the same way. A part of a recursive solution that involves turning the problem into a similar problem is called a **step case**. For Robbie the step case is taking an actual step, but in general it is anything that converts the problem into a simpler but similar problem. Based on the above reasoning Robbie formulates the recursive algorithm:

To descend a flight of stairs:

1. Take a step.
2. Descend the remaining flight of stairs (using this same algorithm)

He takes the plunge and starts to follow the algorithm he has devised. By following this algorithm as expected he takes step after step and eventually gets to the bottom but then it all goes wrong - he keeps trying to take steps even though he is already at the bottom and once he starts to follow an algorithm he does not stop until an explicit instruction in the algorithm tells him to. Luckily, eventually his batteries run out and he falls over exhausted. The problem is that the algorithm never terminates. Robbie did not include an instruction to stop. The algorithm needs a way of checking when to stop: it needs a **base case**. For Robbie, the base case is when he gets to the base of the stairs. In general the base case is just the case when the problem is so simple it can be solved in some trivial way (like just stopping). With new batteries Robbie learns his lesson and adds a base case to his algorithm:

To descend a flight of stairs:

```
if at the bottom of the stairs
then (the base case)
        STOP
else (the step case)
    Take a step.
    Descend the remaining flight of stairs (using this same algorithm)
```

Now once at the bottom it just stops and does not try to carry on descending stairs.

A physical thing that has this kind of **recursive** property is a set of Russian Dolls: wooden dolls that break in half and fit inside one another. Each wooden doll is identical except for the property of being smaller than the last. However, the dolls do not go on getting smaller forever. Eventually as you take the dolls apart you get to the smallest doll, which does not split in half and does not have a smaller doll inside. The smallest doll is equivalent to the **base case** of the recursion. The dolls that do break apart to reveal yet more dolls are the **step cases** of the recursion. If there was no base case in a set of Russian dolls it would mean every single doll had another one inside it. You would open one up and there would always another one inside that could be opened itself to reveal another which....just like the fleas.

The Children's story "The Cat in the Hat Comes Back" (Dr. Seuss, 1997) has recursion as a central part of its plot. The Cat in the Hat has left a stain in the bath that he can move from one place to another (from the bath to a dress to some shoes, etc) but not remove. Eventually he realises the problem is too much for him to solve alone, so instead he lifts off his hat to reveal a smaller but otherwise identical cat in a hat ("Cat A") standing on his head. This cat brakes up the spot and moves the stain to a place where the cat in his own hat ("Cat B") can deal with it. He splits it into smaller spots and moves it to a place where the cat in his hat ("Cat C") can deal with it.

" 'With some help we can do it!'
Said Little Cat C.
Then POP! On his head
We saw Little Cat D!"

The stain is split and moved around by a whole chain of identical but increasingly smaller cats. Eventually "Cat Z" takes off his hat to reveal not another Cat but something totally different called "Voom" that can clean up all the small spots on its own.

"Then the Voom...
It went Voom!"

Voom is a base case if ever I saw one! Base cases solve problems in a different way to the step cases (the cats in the hats) but can only deal with the problem once they have been broken down by the step cases.

The idea of recursion can be used as a way of solving problems and thus of structuring an algorithm: giving **recursive algorithms**. To solve a problem using the recursive approach you must do the following.

1) First work out a series of simpler versions of the same problem that are otherwise identical.
2) Next, find a way of transforming a version of the problem to the next simplest version from part 1 (the step case).
3) Finally find a version of the problem that can be solved directly (the base case)

The recursive algorithm will then be of the form:

```
To solve the problem:
    if the problem is the simplest version (the base case)
    then solve it directly
    else
      transform the problem into the next simplest version (the step case)
      solve the simpler problem in the same way
```

The following is a recursive algorithm for completely taking apart a Russian Doll.

```
To take a series of Russian dolls apart:
    if you are holding the smallest doll
    then place it on the table
    else
        Take the halves of the largest doll apart
        Take out the doll inside it and place it on the table.
        Take the smaller Russian dolls apart in the same way
```

Notice that the problem has been transformed into a repetitive task: doing something over and over again. It therefore should consist of the same basic elements as the loops we saw earlier and it does. There is a test to see if the repetition has finished (the test to see if we have arrived at the base case: "Are you holding the smallest doll") and a series of instructions to be repeated each time (the step case: "Take the halves ..."). The base case appears to be extra, though in many respects it is similar to the initialisation stage with a loop. In fact any solution that can be written recursively could instead be written in the form of a loop. They are just alternative ways of thinking of the same thing. Some problems are more obviously described using recursion and some using loops, but any repetitive problem could be described either way.

Recursion does not have to involve doing the transformation task first and then doing the recursion to finish the problem. Sometimes you do the recursion first to get a result to work with, then do the **step** part of the task. This is the situation when putting a Russian doll back together.

```
To put a series of Russian dolls together:
    if you are holding the smallest doll
    then just hold on to it
    else
        Put the smaller Russian dolls together in the same way.
        Put the next smallest doll (which already has all the others
        inside it) inside the base of this doll.
        Put the top on this doll
```

The feature of a recursive algorithm is that it is defined in terms of itself – there is always a line that says something like "solve the simpler version using this algorithm in the same way". The problem is reduced to just transforming the problem into a slightly simpler version, rather than solving the whole thing in one go. At first sight recursive algorithms look paradoxical in that you appear to be saying something like "to solve this problem, you solve it in the same way" which does not appear to get you anywhere. For example, if you were given the instructions below they would be no help at all.

```
To get to Liverpool Street Station:
    First go to Liverpool Street Station
```

Although this looks like a recursive solution it is not. The secret of recursion is that on each step you have made the problem smaller. Even that is not enough on its own as you would then go on forever always needing to solve a simpler problem. However the base case stops this happening as you will eventually hit it and so no longer need to solve a simpler version of the problem.

Not all problems can easily be solved using recursive problem solving, but ones that can, can often be solved very easily using recursion. The secret is

The following algorithmic puzzle for which a recursive solution can be given is adapted from Kordemsky, 1975.

> *A company of soldiers arrive at a river that they must cross. However, it is too deep to wade, and due to the cold weather it is important that they do not get wet, so they cannot swim. Luckily two children are playing in a small rowing boat nearby. Unfortunately, the boat is large enough only for the two children or a single soldier to be in it at once, though one child can row it on their own. The soldiers do not have any rope or anything else that would be of use. The sergeant gives your group the task of working out a way to get the whole company across the river. How do you do it?*

Try to solve this problem before reading on, writing out a recursive solution. Rather than trying to do it in your head, you may find it easier if you use coins to represent soldiers and children.

The recursive part of the solution is given below. We need to make the observation that if we can move one soldier across the river leaving the children back in the boat, then we will be back in an identical situation, but with one less soldier to get across the river. The base case is when no soldiers remain to cross – then there is nothing more to do.

```
To get any remaining soldiers across the river:
    if all soldiers are across
    then stop
    else
      Move one soldier across ending up with the children back in the boat
      Get any remaining soldiers across in the same way
```

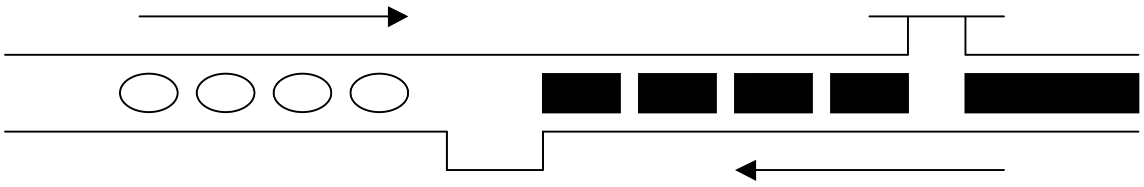
We still need an algorithm for getting a single soldier across. For the recursive solution to work the children must be back in the boat too, as otherwise we will not be back in the same situation as when we started.

```
To move one soldier across ending up with the children back in the boat:
    Both children go across in the boat.
    One child comes back leaving the other behind.
    A soldier goes across, leaving the child behind.
```

The child on the far bank comes back and picks up the other child.

The following puzzle also can be solved recursively (an equivalent puzzle concerning balls is given by Kordemsky, 1975)

A long narrow road contains a single passing point. A single car or van can pull into the passing point allowing cars and vans then to pass it along the road. A side road to the right of the passing point contains a low bridge that cars can get under but that lorries and vans cannot. 4 cars are trying to move from the left and wish to go down the side road. 4 vans followed by a lorry are moving from right to left along the main road (see diagram). The lorry driver has stopped just before the junction and is refusing to reverse as she can see how the jam can be solved without the lorry moving.



Find a way for solving the traffic jam, without the lorry reversing or the cars having to reverse back the several miles to the point where the road widens.

Before we get to the recursive part of the solution we need to do an additional step, to give us a situation that we can get back to. This first step is to reverse the back 3 cars far enough to leave room for the 4 vans to fit between them and the passing place. A recursive solution without this first step would require moving the cars forward and then immediately backwards unnecessarily. This kind of initial manipulation of the problem is often needed for a recursive solution.
The recursive solution is then:

```
To get the next car down the side road:
    if all cars are out
    then the road is now free to move the vans and lorry on down the road
    else
        Move one car out leaving all the other vans in their original
        positions.
        Move the remaining cars out in the same way
```

Now the step case must free one car leaving the other cars and vans in the same positions at the end.

```
To move one car out leaving the vans in their original positions:
    Move the first car into the passing place
    Move all the vans along the road past the passing place.
    Move the car in the passing place off down the side road.
    Reverse the vans back to their original position.
```

The Tower of Hanoi puzzle lends itself well to a recursive solution. It consists of three poles. On the first pole are a series of rings of increasing size (see diagram). At no time can a ring be placed on top of a smaller ring. The aim is to move all the rings to the last pole. Give a recursive algorithm to do this.

Lets suppose we have a way to move all but the bottom ring from one peg to another (this will be complicated to do, but it must be simpler than the current problem as it involves one less peg, so lets not worry how for now). If we could do that we could solve the puzzle. How? We move all but the bottom ring to the middle peg, out of the way, then move the bottom ring to its correct place on the empty end peg (we can do this because it no longer has any pegs on top of it). We then move all the other pegs over to sit on top of the bottom ring on the last peg. Done! Ah, but how do we move that smaller pile of rings? We do it in the same way – move all but one (using the same method again) to the peg other than the one we really want them to move to, move the bottom one of them, then move the rest back (again using the same technique. We have a recursive algorithm.

```
To move a series of rings from one pole to a target pole using a spare pole:
        if there is only one ring
        then move it directly to the target pole
        else
                Move all but the bottom ring to the spare pole in the same way.
                Move the bottom ring to the target pole.
                Move all the rings currently on the spare pole to the target pole in the
                same way (the original pole is treated as the spare pole to do this step).
```

This algorithm has two recursive calls in each step. To solve the problem, you have to solve two identical but simpler versions of the problem. Both recursive calls are simpler than the full puzzle as they involve moving only one less ring. In both cases they involve moving all but the bottom ring from one peg to another. How do you do that? Not by moving them all together in one go, but by moving them one at a time (over and over again) – in exactly the same recursive way.

I once worked as a floor mopper and toilet cleaner for a company called Loadsa Electronics Gizmos Ltd in Cambridge. The main workshops were full of wonderful gizmos at various stages of development. The management were always worried about their competitors sending in spies to steal the best ideas. They therefore had a special lock on every door in the building that you could only get through if you knew the special code (with a special code for each door). The locks were the push button ones – a series of 4 buttons, numbered from 1 to 4 that you had to push in or out just the right order to get in. The buttons stayed in or out as you pushed them. If you pushed an in button it would pop out, and if you pushed an out button it would go in. There was also an extra button marked “clear”. As the main workshop was really important, the sequence on its lock was really long, judging by the length of time people stood there pushing buttons.

I was not, as a lowly toilet cleaner, allowed in to that room, though I knew the codes to every other room so that I could mop the floors. I was not a spy, but I was curious about what was in the room. I spent a great deal of time mopping the corridor outside the room, so had ample time to watch the engineers as they laboured pressing the correct sequence. It was clear since they could all remember it, despite its length, that it was some easy to remember algorithm that was being followed. Counting the number of presses the engineers made, it was clear that the correct sequence was 15 presses long plus the clear button that was sometimes pressed first, if the buttons had been left in random positions) or if an engineer made a mistake in the sequence. It had the effect of making all the other buttons pop in. Also the door always opened only when all but the last button was pressed in.

After several more days watching it was clear that every button push involved either pushing button 1, or pushing the button immediately after the lowest numbered one that was currently pushed out (if any were pushed out). Thus if buttons 1 and 2 were in and button 3 out (so it was the lowest numbered button out) then button 4 could be pressed. After more watching I realised that engineers that managed to unlock the door in only 15 button pushes also never pushed the same button twice in a row.

With this information I knew the sequence. Can you write down the sequence of button pushes that gives the algorithm? Assume it starts with all buttons in so that clear does not need to be pressed and that "push 1" means for example, push button 1.

The algorithm is as below (where we add comments to the instructions in bold that give the position of all the buttons that results after that move – 0 means pushed in, 1 means pushed out):

	0000
Push 1.	**1000**
Push 2.	**1100**
Push 1.	**0100**
Push 3.	**0110**
Push 1.	**1110**
Push 2.	**1010**
Push 1.	**0010**
Push 4.	**0011**
Push 1.	**1011**
Push 2.	**1111**
Push 1.	**0111**
Push 3.	**0101**
Push 1.	**1101**
Push 2.	**1001**
Push 1.	**0001**

There are some interesting things about this sequence. The first thing is that it includes every possible combination of 1s and 0s (though not in the normal binary sequence of counting). In fact the sequence is simpler than normal binary counting as you move on from one sequence to the next, just by flipping one of the 1s to a 0 or vice versa. It also has a recursive property, but we will get to that in a moment.

I wrote it down so I would not make a mistake. Then early one morning, before anyone else was around and the security guard was having his morning cuppa, I typed in the sequence and went into the room. As I said, I'm not a spy so I wont say what was in there, but it was full of many fascinating gizmos. Unfortunately, I left the piece of paper with the code on, on one of the work benches and forgot it when I left just before the guard was due to finish his cuppa. Later that afternoon the Manager noticed the note and all hell broke out, as it was Company policy that no one should write down the code of any door. The next morning the lock had been replaced by a new lock specially built in the workshop, with 5 numbered buttons instead of 4 for extra security. Now the engineers had to push 31 buttons instead of just 15! However, with careful observation I realised that the rules were the same and that it was possible to come up with a recursive algorithm that would work whatever size they made the lock. In fact there was also a safe inside the workshop, that had 6 numbered buttons, and the same algorithm worked for it too. Can you work out what the recursive algorithm is?

Look at the sequence of numbers pushed (1,2,1,3,1,2,1, **4**,1,2,1,3,1,2,1). Button 4 is pushed once in the middle. Before and after it, exactly the same sequence of pushes is followed: (1,2,1,**3**,1,2,1). This sequence has the same property again. The middle number is the largest, with the same sequence on either side. How do you generate the sequence for 4 buttons? You first follow the sequence for three buttons, then press button 4, then follow the sequence for three buttons again. But that begs the question, what is the sequence for three buttons? Well it is worked out in the same way: follow the sequence for two buttons (1,**2**,1), press button 3, then follow the sequence for two buttons again. How do we follow the sequence for two buttons: follow the sequence for one button (1), press button 2, then follow the sequence for one button. The sequence for one button is trivial: just press it!

We can write this down recursively, by replacing the number of buttons by a variable n.

To go through the sequence of pushes for n buttons:

```
    if n is 1
    then  Push button 1
    else
          Go through the sequence of pushes for (n-1) buttons in the same way.
          Press button n.
          Go through the sequence of pushes for (n-1) buttons in the same way.
```

This has virtually the same recursive pattern as that for the tower of Hanoi puzzle! Just replace pushing numbered buttons by moving numbered discs.

Summary

Loops are ways of repeating something over and over again. To describe a loop you need to specify what is to be repeated and when to stop. There are several different kinds of loop that involve giving different kinds of termination condition. Counter-controlled loops involve keeping a counter and stopping when it gets to a previously known number. In such a counter-controlled loop you know in advance how many repetitions are needed. An alternative is to stop when a given distinguished value arises. This is a sentinel controlled loop. Unlike a counter-controlled loop, in a sentinel-controlled loop you do not know how many repetitions will be required before you stop.

Recursive algorithms can be used to solve a wide range of problems. They involve transforming a problem into a similar but simpler problem that can then be transformed in the same way. As each transformation is done, the problem becomes simpler and simpler until it is so simple that it can be solved directly without transforming it further. Recursion is just an alternative to giving a loop. Any algorithm that can be expressed recursively could be described without using recursion. However, it is often simpler to give a recursive algorithm than a non-recursive solution. Many of the algorithms we have seen in the earlier chapters such as binary search can very easily be described using recursion.

7. Bigger and Better (Comments, Functions and Procedures)

Most of the sets of rules an instructions we come across in everyday life are small. For example, the instructions on the side of a packet of pasta about how to cook it are usually a few lines long. Individual recipes are also usually only a dozen or so instructions long. With such simple sets of instructions, the way they are organised is not too critical. However, as the number of instructions involved increases it becomes more and more important that there be some structure. The instructions on the side of a clothes handwash are usually just written as unstructured sentences:

> "Add 50ml of hand wash to 10 litres of water. Allow it to dissolve completely. Wash clothes then rinse well in water. Rinse and dry hands thoroughly after use."

Computer programs can be sets of instructions that are millions of lines long. Even the most trivial programs are likely to be hundreds of lines long. With such large sets of instructions we need a way of making them manageable. How is this done with large sets of instructions in real life? Recipes are generally longer than clothes wash instructions and they are normally structured as a consequence. They often have numbered steps to make them easier to follow – the unstructured paragraph becomes a list. The recipe is given a title, and some description is given about it. An ingredients list is also often given separately. Ingredients lists correspond to type declarations as discussed earlier. We will look at each of these other structuring techniques and how they relate to programming in the subsequent sections.

Fiorentina Pizza
This is a vegetarian pizza that makes a really good meal served with a Broschetta starter. Serves 4.

Ingredients
Yeast
Water
Flour
Tomato Puree
Cheese
Spinach
1 egg

1. Mix yeast and water, add flour and stir
2. Knead for 10 minutes.
3. Leave to rise for 30 minutes.
4. Roll out the dough.
5. Place in a large round pizza tin.
6. Spread with tomato puree, and cheese and spinach.
7. Crack an egg into the middle.
8. Bake in oven for 25 minutes.

Craft books, such as one I have on how to make animated wooden toys (Holland 1995) have a near identical structure. They are divided into separate parts each giving an algorithm for achieving a separate task (i.e. a different toy: make a Flapping Goose or a Tug-of-war Toy). A contents page lists them all at the start. Each has a title and a short passage giving general information at the start:

> “**The Shirt**
> Who will win: the goat or the farmer’s wife? This parallelogram stick toy is at a large scale for added impact...” (Holland 1995)

This is followed by a list of materials and tools needed and then the actual instructions:

> “Copy the shapes from the pattern on page 45 on to the plywood. ...”

The book of card games I have had since a boy (Harbin, 1972) is similar. Each card game is described in a separate titled section, followed by what is required for the game (for example, two packs of cards, pencil and paper. This is followed by some general introduction: “This variation of rummy is included because it is widely played ... and is an ideal game for two players” (Harbin, 1972). Finally the actual rules of the game are given.

Comments

Recipes normally have with them passages that are not instructions to be followed, but are instead information about the recipe. They give information about how and when to use the recipe: “Good as a light snack”, “Serves 6”. Sometimes they give general information about the background of the recipe or how it has been adapted from some earlier recipe: “This is a traditional Indian meal”; “This recipe is adapted from one I came across when visiting Vienna. I replaced the Bavarian Cheese with Smoked Applewood to give it a more tangy flavour”. They may also give some indication about whether the recipe is a quick or slow one: “Cooking Time: 20 minutes”. One of the aims of this information is that a person trying to pick a recipe should be able to make that decision without having to look at the actual instructions. The information should answer the question: “Is this the recipe I am after?”

In programming terms this kind of information that is not an instruction to be followed is known as a **comment**. All programs should have comments to help others understand the situations in which the program should be used. Students learning to program often include comments that just repeat the instructions using different words. Such a comment is pointless as it gives no new information. As in a recipe, comments should give different kinds of information – the history of modification of the program, what situations the program should and should not be used and give any limitations on how well the program works.

In recipes the comments are normally placed up at the top of the recipe, so that they can easily be read before looking at the details of the recipe. Similarly comments are put at the top of a program where they are easily found. Other comments might be put within the recipe or program itself, giving extra information about a particular step to help understand what it is for:

> “Heat the paste for 2 minutes.(This helps ensure the sauce does not have lumps).”

Note that this comment (the sentence in brackets) does not just repeat the instruction, but instead gives a different sort of information.

Comments can also be used to give a shorter description of what a whole group of instructions do (the instructions themselves say how this is done). The comments then give a less detailed description of the algorithm. For example the following instructions are based on those for putting together our travel cot (Mothercare 2000):

Opening the travel cot $\longleftarrow$ **Comment**

1. Unzip the carry bag.
2. Remove the contents.
3. Undo the tabs that secure the mattress.
4. Put the mattress aside.
5. Place the cot upright.

Straightening the top rails $\longleftarrow$ **Comment**

6. Lift the lock in the centre of the rail.
7. Rotate the lock.
8. Repeat the procedure for the other three sides

Without comments, it can be very hard to understand what a set of instructions are really for. Even if it was you who wrote the instructions it can be hard to understand if it is several months since you wrote them. I often write instructions to myself on post-it notes or on the back of my hand. Often by the next day I have no idea what the instructions were for. This is only a problem for humans – computers just follow instructions blindly without caring what they are for – but humans have to instruct the computer to follow a given set of instructions so there has to be something to tell those humans what the instructions do.

To give you an idea of how hard it can be to work out why comments are so useful, look at the following algorithm that was one of my favourites as a child.

1. Take an A4 piece of paper 30cm long.
2. Mark points every 3cm along its long edge to give 11 marks including those at each corner.
3. Draw a line parallel to the long edge of the sheet, 2.6cm from the edge.
4. Make a mark on this line 1.5 cm from the short edge of the paper.
5. Cut down the line, giving a thin strip of paper.
6. Starting at the mark just made, make marks at 3cm intervals along the line.
7. Draw two lines 3cm long from each of the 10 marks along the long line to the two marks nearest to it on the edge of the paper creating 19 triangles.
8. Cut off the two ends that do not make equilateral triangles.
9. Write the number 1 in the first triangle, 2 in the next, 3 in the next, 1 in the next and so on, so that 6 triangles have the number 1, 6 the number 2 and 6 the number 3.
10. Leave the last triangle blank.
11. Turn the strip over.
12. Draw on triangles on this side of the strip in the same way.
13. Leave the first triangle blank, then write 4 in the next two, 5 in the two after that, 6 in the two after that, 4 in the next two and so on until all but the first triangle has a number.
14. Fold a crease along each of the edges of the triangles.
15. Take the strip and fold the first two “4” triangles together.
16. Fold the first two “5”s on to each other.
17. Fold the first two “6”s on to each other.

18. Fold the next two “4”s on to each other and so on until the last two “6”s are folded together giving a shorter strip with only the “1”s “2”s and “3”s showing.
19. Now do a similar thing, folding each pair of “3”s onto each other to give a hexagon with a triangular tab that is blank on one side.
20. Fold the tab down and glue it to the other blank triangle giving a hexagon with six “1” s on one side and six “2”s on the other.

Do you have any idea what the above instructions do? I’ve told you in the instructions you will have a hexagon but why? In what circumstances would you follow these instructions? Those are the kinds of things comments would tell you. Even if you followed them, you might not have any idea what you had made or what to do with it. The instructions are virtually useless to a human without something else – comments. Good comments for these rules should answer all these questions. If you cannot answer them, think what it would be like to try and understand an algorithm containing a million instructions – programs are often that long.

Here are some comments that could have been given before the instructions above that should help you understand the instructions a little more. Even this may not be enough for you to understand – more documentation still is needed!

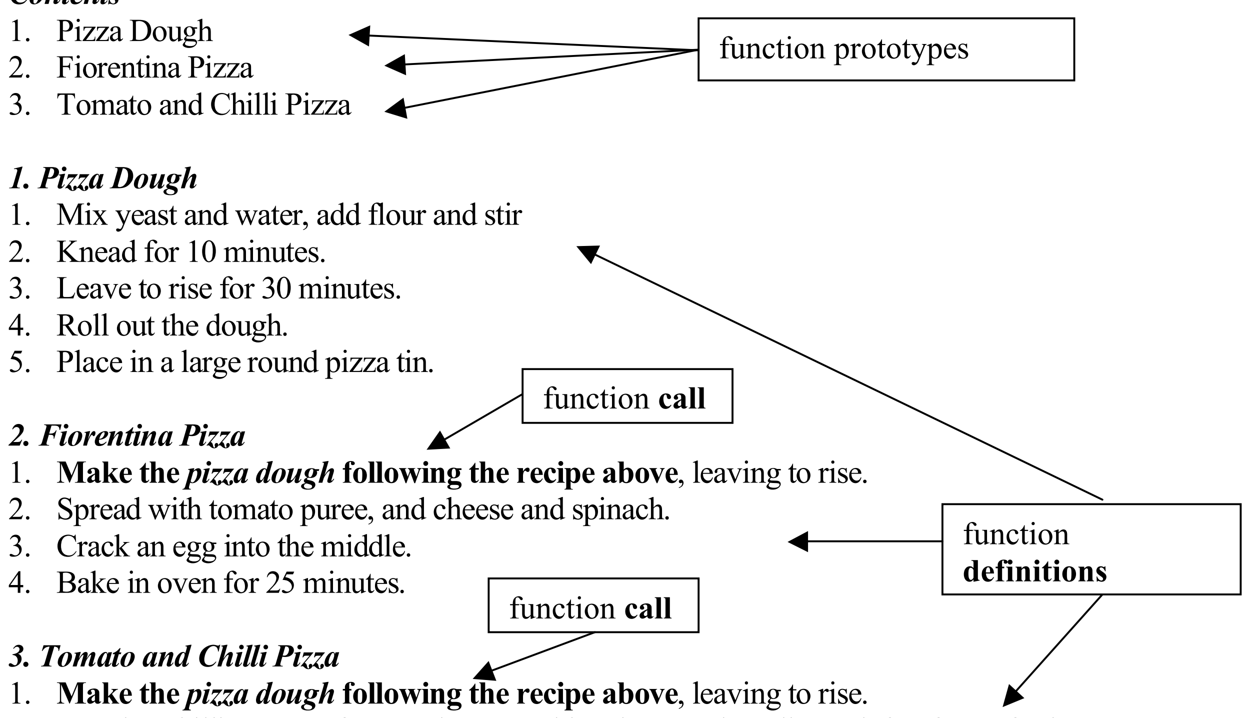
> **Instructions for Making a Hexaflexagon**
> A hexaflexagon is a puzzle. It is a six-sided piece of paper, of which only two sides can be seen at any one time. By “flexing” the flexagon each of the sides can be revealed. To flex a hexaflexagon you pinch together two adjacent triangles, and push the opposite two sides flat. Open up the flexagon from the centre to reveal a new side. Sides 1, 2 and 3 are quite easy to find. Sides 4, 5, and 6 are much harder to find. It takes at most 9 flexes to bring all sides to the surface.

You can find out more about flexagons in the book *Mathematical Puzzles and Diversions* (Gardener, 1965) which is where I first came across them.

If you have trouble following the instructions, then you have probably learnt something about the ambiguity of English – you have thought the instructions meant something different to that which I intended in writing them.

Functions and Procedures

As the number of instructions gets larger new measures are taken. Recipes are generally combined into recipe books that contain in all thousands of instructions. The general structure of a computer program is very similar to a recipe book, though precise details vary depending on the programming language used. This is a difference to most written human languages: programming languages are very finicky about not only the grammar and spelling but also the format used. It is as though all written English had to be written following the format of a Delia Smith cookery book. Even following the format of a different author’s of cookery book would lead to the computer not being able to read the recipes. Let us consider a short recipe book for Pizza. It might start with a contents page, that list all the recipes inside. It then has a series of recipes, each having a name, followed by a set of instructions, to make that recipe. Suppose you wish to make Fiorentina Pizza, you would check where to find it in the contents, and then follow its instructions. In doing so you might, part way through be referred to another recipe, for example, to make the pizza dough. A similar structure exists in many programming languages.



Contents

1. Pizza Dough
2. Fiorentina Pizza
3. Tomato and Chilli Pizza

1. Pizza Dough

1. Mix yeast and water, add flour and stir
2. Knead for 10 minutes.
3. Leave to rise for 30 minutes.
4. Roll out the dough.
5. Place in a large round pizza tin.

2. Fiorentina Pizza

1. **Make the *pizza dough* following the recipe above**, leaving to rise.
2. Spread with tomato puree, and cheese and spinach.
3. Crack an egg into the middle.
4. Bake in oven for 25 minutes.

3. Tomato and Chilli Pizza

1. **Make the *pizza dough* following the recipe above**, leaving to rise.
2. Fry the chilli pepper for 3 minutes, add onions and garlic and fry for a further 5 minutes.
3. Add tomatoes and spread over the base.
4. Bake in oven for 25 minutes.

In computer program terms, each separate recipe is a **function definition**. It is a self-contained set of instructions to do something that can be referred to in several other places in the program. Whenever, that thing is to be done, the person following the instructions is referred to the appropriate recipe/function definition. The point where one recipe refers to another is known as a **function call** when used in a program. To follow an instruction that is a function call, the person temporarily stops following the current recipe and jumps to the other instructions. When those instructions have been completed, the person returns to where they were in the original recipe and continues. Similarly when a computer executes a function call, it stops executing the current function, jumps to the other set of functions. On finishing following them it returns to the original instructions and continues where it left off.

This way of structuring a recipe book or program has several advantages. It means that the same instructions do not need to be written out over and over again. This saves space, but also reduces the chances of the different versions being different. It also makes it easy to substitute one recipe to do a particular thing for another. Perhaps we came across a pizza dough recipe that taster nicer, or was easier to make. We can change the instructions in just one place in the book and all the pizza recipes that use it now refer to the new version. If the dough instructions had been written out in every single recipe then we would need to make the changes in all those sets of instructions. Splitting up recipes or programs in this way can also make them easier to understand.

An instruction such as “make the dough as on ...” makes it clear to someone who has made dough before, what they are doing. Reading the series of instructions

1. Mix yeast and water, add flour and stir
2. Knead for 10 minutes.

   etc.

does not make it immediately clear what you are doing in the same way.

Similar issues arise from programs split into functions. It makes programs easier to write and understand, and helps ensure run-time errors do not creep in. It also makes programs easier to change. We will look at some of these issues in more detail later.

Each separate set of instructions in a recipe book is a recipe in its own right: it is an algorithm that does something distinct and worthwhile in its own right. Similarly functions in a program are individual algorithms that should do something coherent in their own right. You do not just take a random set of instructions from a recipe, and split them off as a separate recipe. Similarly you do not just take a random set of instructions from a program and turn them into a separate function.

Parameter Passing

When we cook dinner at home, my wife and I usually split up the tasks. I do the frying and stirring whilst my wife does the chopping. When it is my turn to be in charge I rarely cook to a recipe but tend to make up things as I go along (often ending up with a red or brown mosh). I rummage through the fridge and see what we have. “Chop this” I might say to Margaret on finding an onion. In essence what I have just done is execute a function call. Margaret is the chopping function – she specialises in chopping. I ask her to chop something and she chops it. The rules she knows about chopping vegetables are followed and on finishing she stops and waits for the next thing to chop. A difference to a function call is that with a traditional computer, only one thing can be done at once – so it is as though when I ask her to chop I stop and wait for her to finish before I go on to the next thing. However, something that is the same to a computer’s function call is the way we pass the onion about. A computer program executing in a computer is more like the situation when my wife is away and I am cooking on my own – then I have to stop what I was doing and take her role chopping the onion. When I have finished I go back to what I was doing. I can only do one thing at once. Similarly when a function call is made when a computer is executing a program, the computer halts what it is doing to execute the instructions in the function.

One of the reasons for splitting a set of instructions up into functions is so that a general purpose set of instructions can be used to do a range of related things. I could have passed Margaret a tomato or a carrot and she would have chopped those just as easily. She is not just a specific “chop onion” function. She is more flexible than that. I **pass** her something and she **returns** back to me the result of her labours. The result returned depends on the thing I pass (If I were to pass her a pepper I would not be given back a chopped courgette for example, but a chopped pepper). She is acting as a general chop function so she chops whatever she is given but that means I must give her the thing to chop. In doing so I am **passing a parameter** to her. A parameter is just a thing that is passed to a function that the function performs its operation on. In computers the thing is a piece of information – data – rather than a real object. I may need to pass my wife some information when she is chopping as well as an object.

She will need to know whether to chop the vegetable into big or small pieces. Thus I might actually say “Chop this onion into small pieces”, or “Chop this carrot into large pieces”. I am passing two parameters to the chop function: the thing to be chopped and the way it should be chopped. Functions do not need to I could pass even more information

A function call thus consists of two parts. I must specify which function I wish to be performed (here “chop”). I must also specify the parameters – in this case the thing that is to be chopped: “this onion” and the way it is to be chopped “small”. A written instruction (as in a program) to do this must thus give a way of indicating these two things: the name of the function and a list of parameters. An instruction in a recipe reflects this:

Chop the *potato* into *large* pieces

Any programming language that allows function call must also give a way of indicating the same pieces of information.

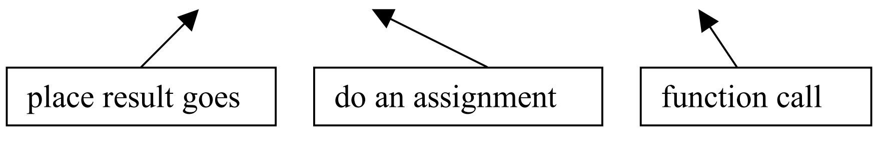
Once the chopping has been done, the result must be passed back. I did not just want the onion to be chopped for the sake of it, I wanted it chopped so that I could then do something further with the chopped onion: fry it for example (another function call, incidentally, this time to a fry function). The function must therefore **return** the result (chopped onions) back to the place where the operation was requested (me). A written instruction in the chop instruction must include something that indicates that the instruction is to return something and also indicate what it is that is to be returned.

We must also have some way of saying what is to be done with the result. In the case of chopped potatoes, we may wish to put them to be placed in a pan of water. Margaret returns the chopped potatoes to me and I put them in the pan. The kind of instruction we commonly use for this is an **assignment**. As we saw earlier, assignments are used when we want to specify that something is to moved from one place to another. We must specify what is to be moved and where it is to be moved to. The place they are to be moved to is the pan of water. The thing to be moved is whatever the result of the function call was which is usually indicated by the function call itself

“Chop the onion and put it in the pan”

Remember though that we said that in computer languages assignments usually are written the other way round – the place the thing is to go is placed first. It would be written more like:

“The pan gets the result of chopping the onion”



An alternative to doing an assignment with the result is to just pass it straight on to another function. On chopping the onion we then fry it. First we pass it to the chop function. The result returned is then passed to the fry function. In a recipe this might be written as “Fry the chopped onion putting the fried onion in the casserole dish”. Again we need to know where the final result goes (the assignment here is to a casserole dish), but the result from the chop function is passed straight on to the fry function.

Procedures are just like functions in that they are a packaged up set of instructions for doing something specific. The difference is that a procedure is not intended to return a result. The point of a procedure is just to do something, not to produce something (the thing returned). To see what we mean by this distinction, suppose I had a servant Egor who did whatever I said. One day I was thirsty so I gave Egor some money and told him "Get me a can of Lilt". To ensure he did this properly, I wrote the instructions of how to get it on a piece of paper. He would follow the instructions, whilst I sat waiting under the shade of a tree waiting. Eventually he would **return** with a can of Lilt that he would give to me. We have executed a function call with a result (a can of Lilt) being returned to me.

On another day I noticed that Egor looked thirsty, so being a kind master I gave him some money and said "Buy yourself a Cola". He would go away and follow my instructions (written on another piece of paper). Eventually he would return as before. This time however, I would not expect him to give me anything. He would not return to me until he had drunk the Cola. This time we executed a **procedure call**. We have followed a set of instructions, but the point was not for a drink to be returned, but just that some action be taken.

8. In-Out, In-Out, Shake it all About (Input-Output)

Communicating with the outside world
Humans think. The human mind is locked inside the body that is a human. Humans do more than just think though – they take note of things happening around them and react to it, taking actions accordingly that affect the outside world. The human mind is like a computer processor. The processor can process information – the computer equivalent of thinking – but to be of any use it needs to be able to sense and interact with the outside world. This is where **input-output** comes in. **Input** is the way information from the outside world enters a computer. **Output** is they way the computer presents information back out to the outside world and so alters it. Humans have a variety of input mechanisms: the five senses. Humans can see (take in visual input), hear (aural input), smell, taste and touch. Each is a different way the human brain obtains information from the outside world to process, think about, react to. Those are not the only way of sensing the outside world. Some fish can sense electric fields. Some birds can sense the Earth's magnetic field and use it to navigate. Counsellor Troy in *Star Trek: The Next Generation* can sense emotions. Computers also have a range of input mechanisms. The most common ones are the ability to detect key presses on a keyboard and movement of a mouse. Some ticket machines and information booths have touch screens allowing the computer to sense touch. Computers connected to the Internet can receive messages from other computers. Many other mechanisms are possible that allow a computer program access to information about the outside world. These are ways the computer senses the outside world.

Humans can also affect the outside world. They can do this by using speech or movement of various parts of the body. Throwing a ball, smiling, kissing, winking, and pushing a button are all ways that people can change things outside the confines of their heads using movement. They are all forms of output. Computers can output to a screen, or by sending messages to other computers along telephone lines.

Consider what it would be like to be able to see, hear, smell and taste but to be unable to move or speak. Such a person would have a perfectly normal intelligence and would be able to sense all that was happening around them. However they would have no way of communicating with the outside world: no way to communicate with their family and friends, no way to express their desires. All they could do is watch and think. It would be a form of hell, and unfortunately it can happen to people who have a major stroke. They regain consciousness in a hospital bed with no way of communicating. It is called locked-in syndrome. Someone with locked-in syndrome has input but no means of output. In fact, people with locked-in syndrome are often not totally paralysed. They may be able to blink one eye. That is enough with ingenuity for them to communicate if slowly as we will see in a later chapter.

In the song "Pinball Wizard" by *The Who,* Tommy is deaf, dumb and blind. He has severely limited input abilities being unable to see or hear. Despite this he is still brilliant at Pinball. How? Because he "plays by touch alone". Blind people can still read using Braille – reading by touch. They are using a different sense to achieve the same input.

A person who has locked in syndrome and who has lost all their senses so they can neither see, nor hear, nor taste, smell or feel, has no way of sensing the outside world and no way of communicating with it. They can do nothing to affect their surroundings. Their mind is cut off from the world: hell on earth. A computer with no input nor output is similarly unable to do anything useful in the outside world. That is why computers are supplied with a range of devices that allow communication: keyboards, mice, screens, printers and Internet connections. These are the things that ultimately make the processing power useful. However, to be used, the computer programs must include instructions that allow the information from the outside world to be used and allow information to be sent out. When I am working I often have music on in the background. Once I am engrossed in my task I do not actually listen to the music. It could be something I hate but I would not notice. My mind is not executing any instructions to process the lyrics being sung. Similarly, I do not speak much as a general rule – only when I have something to say – only when I decide to say something. My daughter, two at the time of writing, often does not seem to have any restrictions she chatters constantly, almost as though she is giving a commentary on everything she is thinking. She is running a program at those times in her head that outputs all she thinks. At other times, she sits silently reading books. Now she is executing input instructions but no output instructions.

If we were to design an instruction to do output, what would it need to consist of? It would need to indicate at least what was to be output and where it was to be output to. What do we need from an input instruction in a program? It must indicate somehow where the input is coming from (that is what device), and it must indicate where the information received is to be put (usually what variable to store it in until it can be further processed).

Interrupts

Persistent Storage
One special form of input-output that computers use is to and from **persistent storage**. The aim of this kind of input-output is a form of long term memory outside the computer.

When I am on holiday on the beach I write my daughter's name in the sand for fun. As long as such messages are only for the afternoon then that is a perfectly good place to leave them. When we return the next day, though the messages are gone – the tide comes in and washes them away. There would be no point me leaving a message when she was not there for her to see the next day as it would be gone. Sand messages on a beach are **non-persistent**: they do not stay around permanently. I can write names and smooth them away as much as I like during the afternoon, but that is as long as they last. Persistence thus does not mean it is impossible to delete a piece of information, just that it will stay around until you actively delete it (by smoothing the sand yourself say). Computers use **non-persistent storage** as their everyday memory. That is why it is so important when using a computer to remember to save things and why it is so easy to "lose" hours of work if your daughter decides to press the off button while you are typing. When you save something you transfer it to **persistent storage**: a place where it will stay until actively replaced.

Vandals writing graffiti do want persistence. They want their graffiti to still be there the next time they come by. They therefore do not write their graffiti using something that will wash away the first time it rains, but instead use permanent markers and aerosol sprays. This gives **persistent storage** for their handiwork.

Before writing was invented and in common use, knowledge and stories were passed orally. The only way the knowledge was preserved was by it being constantly passed from one person to the next. The Koran is still, I am told by a Muslim friend, passed on in this way being learnt word perfect by each new generation to ensure the words are exactly those of the original. The memories of Muslims throughout the world are acting as a storage method for the words in the original Koran. Languages and culture are passed on in the same way from generation to generation. The problem is that if one generation decide not to do the storage, the knowledge may be lost forever, if that is the only storage method used. Many languages have disappeared because a generation has lost interest. it takes great dedication, as with the Koran, to use this method of ensuring knowledge is not lost. Writing was invented as a form of persistent storage. The written word stays around much longer than any individual person. That is why we can learn much about ancient civilisations that have long been extinct.

Libraries are a form of persistent storage. The things they store are books. I just tried to order a book from my local bookshop about Computers and the Mind. I was told it was impossible for me to get as it was out of print. However, that just means I cant buy it. I can still get a copy to read as the British Library keeps a copy of every single book printed. The British Library's mission is to act as a persistent storage for books to ensure that no English book ever printed disappears.

The main mechanism for persistent storage in computers is the **file**. Most variables used in a computer program that hold strings, integers, etc are non-persistent and so are lost as soon as a program ends (if not sooner). Passing on knowledge orally is like keeping a computer switched on permanently to ensure information is not lost. The problem is it needs constant attention (or more specifically for the computer, constant electricity). Similarly when the life force leaves a person and they die, their memories cease to exist too (though unfortunately switching a person back on is not currently possible after they die). On the other hand, something placed in a **file** will still be there even if the computer is switched off and it is days before you return to it. Storing something in a file is like storing something by writing it down on paper rather than trying to remember it in your head. Persistent storage is something that by its nature exists outside the program, in the same way as paper exists outside the head of the person who writes on it. It must still be there even if the program itself disappears.

Putting something in a file is a form of **output**. Getting something out of a file is **input**. This is because information is leaving from or arriving in the program. Programs can also get input from things like a keyboard or a mouse, and most commonly they output to a screen. Notice that writing something to a screen is not like writing it on paper. The screen is not persistent, paper is. The aim of outputting to a screen is not to do with **storage** but just presentation so a human can see the information. When outputting to a file, the aim is long term storage.

A **file** can be thought of as a special kind of value or object that has an existence outside the program as well as in it. In a program a file can be stored in a variable, so that it can be passed around. However, if we think of a variable as a box to hold something, a file variable is like a box with a hole in it. Things that are stored in the box end up somewhere else: in the file. The file itself is not stored in the box, the box just provides a link to it. The file is elsewhere, outside the confines of the program. The file itself has a name that it is known as in the outside world that is different to the name of the variable that is linked to it in the program. The variable's identifier (the name of the box) is just the temporary name used in the program. The file name is the name used in the real world that is the permanent name of that file.

This is all a little like the wardrobe in "*The Lion the Witch and the Wardrobe*". A normal wardrobe contains clothes, just as a normal variable is used to store things like numbers, but that special wardrobe contained a link to a different world. Step into the wardrobe and you would step out the back into the magical world of "*Narnia*". Similarly when you put something into a file variable it escapes the confines of the program and into the wider world outside. As we said the variable has a name associated with it, but that may be different to the name of the associated file in the outside world. Similarly everyone called the piece of furniture in the side room "The wardrobe", but the name of the thing it was connected to had a different name "Narnia". The name of a variable may be different to the name of the file that it is connected to.

Buffered Input-Output

Input-output (whether to files, the screen, keyboard or otherwise) is often **buffered.** What this means essentially is input-output requests are stored and then a whole bunch of them done in one go. Libraries often do this. Small village libraries have all the books on the open shelves, so if you want to take a book out of the library (output), then you just go and find the book, take it off of the shelf and check it out at the desk. This is not buffered output – the single request for a book is done on its own. However, if you want to take several books out at once, you do not go and get one, get it stamped, then go back and find the next one, get it stamped and so on. Instead you go into the library, find all the books that you wish to take out, and take them all to the library desk at once, and get them all stamped in one go. You are buffering all the requests. As you collect the books you tuck them under your arm until eventually you have a stack of them. Your arm is acting as an **output buffer**: a place where the books are collected until you can process them all in one go. Why do you do it this way rather than one at a time? It is basically because it is quicker and more efficient. Finding a book is slow and involves wandering round the shelves, but by getting several at once, lots of time and legwork can be saved.

When you bring a book back, you hand it over to the librarian at the desk. He or she does not then immediately go and put it back on the shelf where it lives. Instead they will probably have a temporary "returned book" shelf which holds all the books brought back that day. At the end of the day, or if it becomes full, the librarian puts all those books back in their correct places at once. The "returned book shelf" acts as an **input buffer**. It is used to gather together all the books so they can be returned efficiently in one go. Of course there is a trade-off between the size of the shelf and how often the librarian has to do a tour round the library taking books back. The bigger the shelf, the less frequently the librarian has to do the rounds but the more books are unavailable for other people to take out (or at least they are harder to find).

Big libraries that are used as reference libraries have books that are in areas deep in the library where only the librarians are allowed to go. The British Library in London is like this. Rather than going to get a book off the shelf yourself, you have to put in a request and a librarian goes and finds it. However, you have to wait before your request is processed. In the British Library items ordered take about 70 minutes to arrive (presumably meaning they get them every hour and take about 10 minutes fetching them). The librarian collects a whole series of requests from different people and gets them at the same time. The library is using an **output buffer** to process requests.

Most airports have long corridors linking the terminal to the departure gates, often with moving walkways to get people there quickly. This is like a non-buffered input-output system. Passengers leave the terminal as and when they like. Stansted airport uses a buffered system. You do not go to the gate when you decide, but instead wait for a tram. It fills up with a group of people who have all been waiting to go. They are all then transported together.

Computers use input and output buffers to buffer data that is to be written to output devices such as the screen or files. The data is collected in a buffer and then actually input or output all at once when the buffer is full. If we think of an output file variable as a box with a hole in it. Buffered file output is like having another big box under the hole. Anything dropped into the file variable box, falls through the hole and into the bigger box. There it waits as other things fall through the hole and join it. Eventually the box is full, at which point a trap door opens due to the weight and everything dropped into it, falls down a tube that is connected to the file – travelling down the tube in the same order it entered the box. Thus everything ends up in the file just as it would have done if it the pipe had been connected directly from the first box, the only difference is that it arrives there in groups.

Programming languages may also provide an instruction to empty the input or output buffer now – the equivalent of the head librarian telling a junior librarian to clear the returned book shelf now rather than wait until it is full. For example, when typing at a keyboard, the characters typed in are not processed until a return key is hit. They are stored in a buffer and shown on the screen in the meantime. The return key is used as an instruction to mean “clear the buffer and process the characters typed so far”.

In the British Library most items are stored on site and so take 70 minutes to arrive. Others are stored off-site and so take longer – up to 2 days. For off site books, the buffers are emptied, and so the requests processed, less frequently. This also demonstrates the point that things stored in some places can be accessed more quickly.

In some programming languages, the buffering is invisible. In others, such as Java, it is explicit, instructions exist for separately creating a file and creating a buffer that is then connected to it.

9. Repair, Reuse, Recycle (Object-oriented Programming)

I have done one braver thing
Than all the worthies did,
And yet a braver thence doth spring
Which is to keep that hid.
John Donne, Songs and Sonnets 'The Undertaking'

What we have looked at so far has been procedural programming. This is just one way of writing programs or **programming paradigm**. A paradigm is a way we see the world. For example, people with different political views – a Right winger versus a left winger, for example – have different political paradigms. They see events in different ways. A right-winger might see the British Welfare State as being about paying money to scroungers who have not bothered to look after themselves, and as just encouraging more such behaviour. A left-winger on the other hand would see it as a humane thing ensuring basic human rights. A left-winger would see an increase in taxes to improve services as a good thing. A right-winger would see the same thing as a bad thing. The two people concerned have a different view of the world. They have different paradigms.
A programming paradigm is a different way of seeing the task of writing a program. There are a variety of different programming paradigms including procedural programming, object-oriented programming, functional programming and logic programming. The procedural paradigm, can roughly be thought of as "a program is a recipe book". This idea was the way early programs were written in part because it is tied closely with the definition of what an algorithm is, but there are other ways of thinking of the task of programming – the task of writing instructions. Object oriented-programming is now the most popular approach. A procedural programmer would see the task as determining which procedures need to be written – about the state of the system and how to change it. They are thinking in terms of actions that the program performs: each procedure is a recipe for how you do some action. Given this input, how do I turn it into that output? An object oriented programmer, would instead think about what objects are to be manipulated to carry out the task, what their relationship would be. They still have to think about operations but in the context of what operations could be performed on each kind of object that has been identified. The objects come first.

To more clearly see the difference, imagine you are a chef running a small but select restaurant. Because you are expensive, you only offer a set menu, with a small number of options each evening (but boy is it good). How would you think about the task of preparing that meal and plan for it: it would be a series of dishes, one for each course. One obvious way is to write out a recipe for each dish. The menu acts as a contents page into this list, but also indicates the order that the dishes are needed – it is a sequential set of instructions. First prepare the Brishotta as it is the starter, then get to work on the Fiorentina Pizza while the guests eat the first course. You are thinking procedurally. You might split up the procedures and produce an action list that determines the order that everything should be done in. I must put the potatoes on first, then while they are boiling, I must prepare the starter. You have lost the procedural paradigm as you are no longer thinking in terms of separate recipes for each course, but one big recipe for the meal. However, you are still thinking in terms of actions and the order they are achieved. Perhaps you might have a series of helpers

– they could be each given a course to prepare. You have moved to a parallel paradigm as now several things can happen at once – though still one procedurally or action based.

Now think again. Your team of helpers could be trained with specialist skills. One could be a vegetable expert perhaps. He would know how to chop a carrot with a big sharp cleaver extremely quickly and without pieces of finger ending up mixed in. He can core a cauliflower by a flick of the wrist and a tap on the table... Another is a meat expert with similarly appropriate skills. Someone else is a bread, pastry and dough expert. Another is a pizza expert. Given the prepared ingredients, she can make a pizza pass as a piece of modern art and was once shortlisted for the Turner prize. Now think again about the activities in such a kitchen. Instead of focussing on recipes you can think about the objects that are being manipulated: spinach, dough and pizzas. He can twirl dough like Tom Cruise can mix a cocktail. Customers pay just to watch him. The waiter, passes an order through the hatch – perhaps there are a row of hooks that each order is spiked on so that they form a queue to ensure they are processed in order. Fiorentina Pizza is needed. The order is taken up by the Pizza specialist. She realises she needs some Spinach, an egg, dough etc. She writes out orders on a new set of slips and passes them to the appropriate people – putting them on their hook list, to process in their turn. The spinach is been washed and chopped by the vegetable expert, the dough made by the bread expert, etc. Once the prepared ingredients are passed back from the separate people to the pizza expert, she puts them together into a pizza to die for. Eventually it is ready and she passes it to the waiter who passed the order on. In the meantime, an order comes in for Crispy Duck. The meat expert similarly sets in motion a chain of sub-orders. At various points the various experts need new vegetables or whatever to work on. Whenever a request for chopped cabbage comes in, the first job is to get a new cabbage, for example. We are now thinking in a more object-oriented way. We no longer have a single set of instructions for a meal. Instead the instructions, the operations are distributed – they reside with the objects they apply to. The knowledge of how to make dough by kneading and twirling the dough is with the dough expert. Operations are associated with the objects they apply to. Whenever an object is needed – you just go to the expert of that object.

The second view of how the kitchen works, gives a completely different picture, even though ultimately the same operations are performed on the same things. It is a different paradigm. What is the difference? Well, one part of the difference is between verbs and nouns. In the first version, the chef concentrates primarily on *verbs* – action words – the things that need to be done. He thinks in terms of "I must chop this, then mix that, boil,...roast, slice, wash...". The focus is on the actions and the order that they must be done to obtain the final result. Remember that was the way we talked of an algorithm: writing down a series of actions and the order they must be performed in. People are allocated to those actions and when done they come back to be given a new action to do. First they might chop a cabbage, then marinade a chicken, then wash some spinach...

In the second description of the kitchen, the focus is on the *nouns* – things – like cabbages, chicken and spinach rather than the verbs – chop, marinade and wash. Now when people are allocated, it is not to tasks, but to things or categories of things: "You are the cabbage expert. Learn all the skills needed to do anything I might ask with cabbages". You on the other hand are to be the chicken expert and you are to be the cake expert". Now the people – the work units of the kitchen – jump into action whenever one of the things that they specialise in are needed, rather than just being assigned tasks. The focus of the organisation is on nouns rather than verbs. It is closer to an object-oriented description rather than a procedural description.

A procedural programmer thus thinks of the task in terms of the actions that must be taken – the verbs. An object-oriented programmer thinks in terms of the things that actions will be performed on – the nouns. Their thinking is oriented towards things: they are oriented towards objects rather than actions. This difference is then reflected in the language used to write the program. In a way there are similariites in human languages. When babies learn to speak, their first words could be nouns or verbs. In English the first words are likely to be nouns. One of my daughter's earliest words were for things: "teddy" and "car", for example. In some other countries the first words are likely to be verbs, because verbs are predominant in the language and so when the parents are pointing out things they are likely to be emphasising the verbs rather than the nouns. Similarly some programming languages make it easier to focus on verbs – action-oriented languages (i.e., procedural ones), while others make it easier to focus on the nouns – object-oriented languages. We will look at the features that make a language object-oriented in the subsequent sections.

The instruction booklet for my hi-fi is written within an object focus. Rather than being organised around tasks – with sections based on tasks I might wish to perform. The sections are organised around the objects in the hi-fi: a section on the CD player, a section on the radio, a section on the tape deck and so on. Actions are tied to those objects. The manual writer thought of objects first then actions. It is an object-based description. My clock radio on the other hand is organised by tasks. There is one task for each task I might perform. "set the time", "Set the alarm", "Switch on the radio", "Change radio channels". There is no clear focus on the separate objects, the clock, the alarm and the radio in the way the manual is organised. It is a task-based description.

We also see this distinction between organising around nouns (categories of things) and organising around verbs in the way shops are organised. In a traditional high street the shops specialise in one category of objects: the butchers, the bakers, the fruit and vegetable shop and so on. Supermarkets tend to do the same thing too – there is usually a bread counter, a cheese counter, a meat counter and so on. It does not have to be like that though. Many DIY stores organise themselves differently – around tasks – there is a tiling section (tiles, grout, tile-cutters), a painting section (paint, paint brushes), a wallpapering section (wallpaper, paste, papering tables), a gardening section (plants, spades, compost). A supermarket could organise itself in a similar way, with a dinner section, a breakfast section and so on. Or it could have a baking section, a boiling section. It works for DIY because people buying DIY materials tend to be task-oriented. They are thinking of wallpapering the living room on Sunday. When shopping for food, people tend to be much more object-oriented. They are buying for more than just one meal (more than just one task), so in the butchers they might buy, lamb and chicken. In the bakers, sliced bread for breakfast toast and doughnuts for tea. Also the supermarket organised round meals or actions would have to have particular objects replicated in several sections. Was that toast for breakfast of for tea? Were those potatoes for boiling or for baking? This gives a hint as to why object-oriented thinking is so much more powerful in programming. It helps make programs more reusable. When writing an object-oriented program, you are thinking in terms of writing instructions that will be reused for a whole range of different tasks. A task-oriented program is written for a single task, so is likely to need more work to make it fit a different task.

There are many concepts tied up with object-oriented programming called things like classes, objects, encapsulation, inheritance, message-passing and so on. We will look at each in turn. Many of the basic concepts appear in other paradigms. It is only when they are all combined that we truly get the object-oriented paradigm. Hidden within the above description of the kitchen are many of these concepts. We will gradually separate them and explore what these concepts are individually.

Objects

One of the aims of object-oriented programming is to think about programs more like the real-world – just like this book does! The world is made of objects of various kinds, and since most programs are concerned with modelling something in the real world it makes sense to think of the virtual world a program creates to also be made up of **objects**. Objects could represent cabbages, people, houses, aeroplanes ... anything that can be thought of as a person in its own right. Objects come equipped with **attributes** or **state** – something that makes this one different from the next – the way you tell them apart. For example, cabbages have weights, people have eye colour, ages and heights, aeroplanes have flight numbers and destinations, and so on. Attributes are nouns: weight, destination, etc. If two objects have the same state then they will look identical, just like two identical twins appear to the same person. They may act the same and look the same but they are really two completely different people. For real objects the state is just the way it is – the weight of a cabbage is intrinsic within it. For virtual objects the values have to be stored somewhere. As we discussed previously, in computers, things are stored in variables. Think about a car. One of its properties is its registration number. It is stored on a registration plate – in a known place for all cars. If you want to know the registration number of a car then you look at the registration plate. It is acting like a variable for one of the pieces of state of the car. With objects, it is as though all the different parts of the state were all stored in a separate plate that could be checked when the information was needed. For example, it is as though a person carried an identity card with them, and that identity card became the essence of them as far as the state was concerned. If your ID card says you have blue eyes then your eyes are blue! If the police want to check if you are the person they are looking for they just ask to see your ID card and compare it with the profile of the person they are after.

Objects come equipped with more than just **attributes**, however. They also have **behaviours**: the operations that can be performed on an object or more in line with object-think: actions that objects can perform. Each object comes with its own set of behaviours specific to that kind of object. For example, a cabbage can be *chopped* and *boiled*, dough can be *kneaded* and *twirled* methods. Paint can be *mixed*. Thus whereas attributes are described using nouns, behaviours are described using verbs: doing words. The behaviours of an object are implemented as **methods. Methods** are just functions personalised for a particular object. For each kind of object in a program a programmer writes methods to describe its behaviour and gives instance variables to describe its attributes. The instance variables can be checked to see what the values of the attributes are: is the hair of this person black? The methods can be followed to take the actions of the object. In doing so the attributes might change. For example, if a person whose hair colour is brown (hair colour attribute: brown) *dyes* their hair blonde (following the instructions (the method) on the bottle of dye), then when the method has terminated, that person's hair attribute will now have changed to value blonde.

In the Restaurant at the End of the Universe (Adams, 1980) genetically modified animals are bred to delight in being your meal. They take great personal pride in being perfectly fattened and might suggest their leg as being particularly tasty. We can take this idea a little further. Imagine a genetically engineered future in which vegetables and animals are bred to have such intelligence (if you can call it that) and are able to cook themselves. Each chicken has access to a knife and can slice itself to perfection at your request. It can roast itself to death or alternatively fry itself to perfection. Now, when a request comes in messages go to the cabbages, the ducks, etc and they get to work preparing themselves as requested by the pizza, who on creation promptly bakes itself in a hot oven as requested. This is roughly how an object-oriented programmer sees a program. Each different kind of object is supplied with instructions about all the operations that can be performed on it. If such an operation is required, a request goes to the object itself to do it.

Children's films, television programs and books often personify objects in this way. For example, in the Disney version of *Beauty and the Beast*, most of the objects in the castle such as the cups and candles can move around and talk. They not only have attributes they also have behaviours. They get things done by communicating with each other and the object concerned just gets on and does whatever is required. Both the attributes and the behaviours are intimately toed together. Similarly in *Bob the Builder*, all the diggers and cranes can do things for themselves when requested. To be a truly object-oriented television programme however, the bricks, gates walls and so on should also have minds of their own.

Futurologists and science fiction writers will have us believe that all the objects of the future will be responsible for their own actions in this way. In *The Hitchhikers Guide to the Galaxy*, the lifts have control of their behaviours (and personalities to go with them). In the TV programme *Red Dwarf* the vending machines do. In the film *Dark Star* even the missiles have control of their own behaviour – they have the method: *explode* which is fine if they only do so when told to but leads to one depressed missile becoming suicidal. In futuristic houses of the moment, there are fridges that order milk when it runs out – so changing the attribute of amount of milk in the fridge; the garage doors *open* when the car pulls into the drive, and so on. Why not take it further and have the mugs take control of making the coffee.

Even inanimate objects in the real world often have behaviour – a sponge soaks up water on its own, even though it is inanimate, so you do not need to move to a fantasy world to get the idea of objects having both attributes and behaviours. Most commonly inanimate objects only do things when they get some kind of signal or message telling them to do so. For example, my hi-fi switches itself on when I use the remote control to send it a message to do so. The light bulb switches itself on when I throw a switch at the other side of the room to tell it to do so. When I pull the chain, or press the handle on the toilet, the toilet flushes. That is how objects communicate: by **message passing**.

When I am driving along the road and the brake lights of the car in front (an object) go on, it is sending my car (another object) a message to take an action: apply your brakes. The message is the light going on, but because it is a specific light, it sends a message about a specific action I should perform. In taking that action my car will similarly send a message to the car behind it.

The lights on cars are not strictly directed at a specific single other object – and if no one is behind then the message may not go anywhere. The objects in programs send messages to specific other objects. Thus, when writing instructions, two things at least are needed to communicate to an object that it should perform some behaviour. You must indicate the object that must take the action and the behaviour out of the many associated with it that it must take. Many dogs can perform tricks. They are "objects" (appologies to dog lovers) each being able to exhibit a range of doggy behaviour. When we want them to do tricks we give instructions.

"Fido Sit",

"Bonzo Beg",

or even

"Caesar Kill",

all consist of the name of the object followed by the behaviour to be performed (here separated by a space). Commands to objects in a programming language are similar though the symbol used as the separator might be different (for example using a dot instead of a space).

The attributes of objects can be other objects. You could consider a person as having an attribute of blue eyes and another of short sight. Alternatively, you can think of a person of having an attribute that is another object: the eye. Eyes then have attributes of colour and short-sightedness. Because people have eyes and eyes have colour, then by default, people have an eye colour. This can be a good way of organising a description of an object. If you have defined what you mean by eye objects for people, then you can reuse that description for Kangaroo eyes should you later be describing them instead because a kangaroo object also has eye objects as attributes.

Classes

As we discussed earlier, we naturally group and classify objects together into things of the same kind. For example, in the above, I have spoken about cabbages. There are many cabbages in the world, but the **class** of cabbage all share the same properties and the same operations. That is why when I said "cabbage" you pictured the right thing – not any specific cabbage, but all cabbages. A class is thus just a category that groups a variety of objects together. In this sense it is just a type as we discussed previously, but one whose values come equipped with operations too. They are a complex form of **user-defined type** as new categories can be created if needed for a new set of instructions.

As it stands the bricks in Bob the Builder are more like primitive types than classes. A value of that type – a brick for example – is acted upon by other objects but cannot do anything for itself – it has state (its size and colour, for example) but no operations – it is passive. The cement mixer in Bob the Builder on the other hand is not a passive thing that operations happen to but one that comes with the skills to accomplish various tasks (and attitude!) Given the ingredients it can apply operations to itself.

A **class definition** thus is a description of what it means to be an object of a particular class: that is of a particular category. It defines what state objects of this category have and what operations can be performed on it – and how the objects of the category perform those operations. For example, some of my recipe books have sections that are organised along these lines. The Pocket Encyclopedia of Vegetarian Cookery for example (Dorling Kindersley, 1991), has sections on "Nuts", "Beans", "Fresh Fruit" and so on. Each is in effect a class definition. The "Nuts" section not only defines what it means to be a Nut ("[it has a] kernel containing the whole future plant in embryo") but also lists the operations explaining how each is done. "To blanch nuts...To roast nuts, ...To grind nuts..." If you need to do something with nuts, the nut section tells you how. That is basically what a class definition for nuts would do: want to train to be the Kitchen Nut expert? Then make sure you know and can do everything in the nut section.

Constructors
One of the things a class definition should tell you is how to get hold of a new object of that class. The Encyclopedia of Vegetarian Cooking does this too. The nut section contains a passage about buying nuts. You need a new nut, then follow the instructions in that passage and you shall then successfully be holding a nut. The Grains section contains a passage about buying grain. Such instructions are known as **constructors**. Given some information about its state – the values of the attributes it should start with, eg of the weight of cabbage needed, by following the cabbage **constructor** you will have an appropriate cabbage. A constructor is thus a special form of method associated with a class, rather than with an individual object. It creates an object of that class. In our restaurant example above the vegetable specialist will also specialise in buying vegetables. He will know the market stall that sells the freshest vegetables at the lowest price, for example. In doing so he is exhibiting the vegetable constructor behaviour.

Information Hiding / Abstraction
An important subject in computing and especially with respect to that of data structures is **abstraction**. Abstraction is the process of hiding or ignoring detail so as to make something clearer. It is used throughout everyday life. One of the most successful abstractions is that of the map. We use maps all the time because they are such a good abstraction. They are an abstraction of the things you would find on the ground – not every last detail is represented – a map is not a virtual reality simulation of the area it represents. Instead the map-maker picks out the details that you need to know to do the task the map is designed for. Different maps are for different things so use slightly different abstraction i.e. they hide or accentuate different features. A road atlas is designed for drivers. They do not usually worry about hills so most hills are not visible on a road atlas. A hill-walker's map is for walkers, however, and they do care about hills so contours are used to plot the hills to a very fine level. Walkers are not interested in exit numbers of major roads or whether a road is a one-way street, so those details are hidden on a walker's map.

Games and puzzles are often intended to be abstractions of real life. Chess, for example, models some of the features but not all of ancient battles. The battle involves destroying the units of the opponents and is won when the leader is captured, for example. Other aspects of real warfare such as the horrors of death are lost.

Monopoly is intended as an abstraction of the property market. The buying and selling of property is modelled, but other aspects such as getting surveys done and paying solicitors are not.

The important skill of note-taking is an abstracting task – indeed a summary of a document is often called an "abstract". What is important when taking notes in a lecture is that you get down the important points that the lecturer makes, whilst omitting the less important. If done well your notes should contain sufficient information that you can reconstruct the rest yourself. Similarly when summarising a passage read in a book for an essay, your aim is to extract the important facts that are relevant to the question you are answering, omitting less relevant points even if they are interesting. Novels and films often contain gushing abstracts written by reviewers. They abstract all the details of the plot, leaving only enough detail to convince you to read the book or see the film. Sometimes they go too far and lose so much detail that you are left without a clue what the book is about – just that it is "a novel of rich diversity that triumphantly integrates rationality and imagination". Does that tell you details that help you decide whether to buy the book? Is it a spy novel? A romance? Abstraction is only of use if appropriate details are hidden. Sometimes it can have the opposite effect. For example, if a review contained the quote "the most interesting book I've read" you might decide to buy it. If you knew that the full sentence was "the most interesting book I've read was not this one" you might think again. An abstraction should hide details but leave behind the essentials so that the message is not corrupted.

There are several different kinds of abstraction, grouped around the kind of information that is being hidden. We will look at each in turn.

Hiding the Value
One form of abstraction is to hide the value or **abstract away from the value** of something replacing it with a label or a name. You know the attribute exists, just not what its value actually is. This is done in mathematics all the time. For example, the name pi refers to a number with approximate value 3.14. When we learn equations about circles we just learn them all using the name rather than the number. We similarly use labels to refer to particular roles. For example, if you wished to apply for a place for your child at Primary School, you would ask to speak to "The School Secretary" and you would be directed to the correct person. If by the time you were applying for your second child the person had left and been replaced by someone else, the label "School Secretary" would still get you to the right person. Similarly if you have a complaint to make about a product you bought, you could write to "The Complaints Department" and it would get to the correct person, even if you did not know who the person was. This form of abstraction again is of use because it allows the details that have been hidden, to be changed without other people needing to be aware of the change.

Fitted kitchens normally have identical doors on all the cupboards and drawers. Think of each cupboard as a variable – a place where something can be stored. The doors are to hide all the things that are in them – hiding the contents or the value. Some of the doors may even have fridges, freezers or ovens behind them. You cannot tell.

In drug trials, often different people in the trial are given different doses of a drug and some are not given any dose at all – a placebo. However, all are given an identical pill so that knowledge of the dose they are taking does not affect the outcome and so that it can be determined how much better the drug is than taking nothing. The participants do not know the size, the value, of the dose they are taking.

Hiding the Implementation
Another form of abstraction that is important in computing is that of hiding the details of the *way* a thing does whatever it does. We **hide** or **abstract away from the implementation**. For example, cars are designed so that on the whole you can drive one without having a clue how an internal combustion engine works. In fact you do not even need to know a car contains an internal combustion engine at all to drive it. You just need to know the function of each of its controls but not how they do it. Modern cars often have electronic rather than mechanical controls. However, someone who learnt to drive in one car can easily switch to another because even if the insides are different, the controls are the same – all cars have a steering wheel and a brake, for example. This is an important reason for using this kind of abstraction in programming. If the abstraction is done well, you can completely change the internals of the thing, and as long as the controls are the same, the change will not be noticed by whoever or whatever is using it.

Pianos similarly hide the details of how they work. All pianos have the same controls, and if you do not lift the lid, you do not need to know how they make the music to play one. A person may normally play a normal piano that works by plucking strings. However, they can be put in front of an electric piano that works using a completely different mechanism and still play it and again, unless they have a good ear, may not be able to tell which mechanism is being used.

We have already seen this form of abstraction. The use of methods, functions and procedures allows us to use the name of the function or procedure and so hide the details of how it is done. When we write “Make the pizza dough” in a recipe we are hiding the details of the recipe for making pizza dough – it will be on another page if we really do need to look at the details. This kind of abstraction allows us to replace instructions of *how* to do something by an instruction of *what* it is we wish to be done.

Hiding the Attributes themselves

We saw above that one form of abstraction is to hide the *value* of an attribute. Related to this is the idea of hiding **attribute** itself. Not only is the value hidden but the existence of that atribute is also hidden. We are hiding or **abstracting away the details of the representation**. By wearing dark glasses, you hide various attributes of your eyes such as their colour, but also the direction they are looking (the direction they are pointing is an attribute, whereas actually looking and taking in what is in that direction is a behaviour). That is why some people wear dark glasses indoors – to hide the direction so other people can not tell where they are looking.

Good poker players hide their emotions behind a stony face – they are “poker-faced” – they are “emotionless”. When they turn up the fourth ace, there is no way of knowing from the other side of the table that they have done so. It is as though they have no emotions. The attributes corresponding to their emotions are hidden to the outside world – not just which emotion but that they have any emotions at all. You could be playing a robot. That does not mean they do not have emotions. Inside the person with all the Aces is grinning. It just does not get to his or her mouth. It just means that the emotions are internal, private attributes that cannot be seen on the outside. At the other side of the table you cannot tell what emotions the person possess or even if they have any at all. Similarly when haggling in a market, a good market trader will not let the person know the minimum they would be willing to sell it for. It is a private attribute. You may suspect that there is such a price, such an attribute in the trader's head, but you cannot be sure. For all you know the trader has not thought about it at all. Business negotiations are also like this. The two sides go into a negotiation knowing lots of details of their position, but not knowing those of the other side. Good negotiators are ones who can break through the other person's abstraction

Rich people (in films at least) often have hidden safes. A room has pictures round all the walls, but one of them has a safe behind it. To a casual observer, there are no safes, no storage spaces, in the room. This is a different situation to the fitted kitchen. There you did know there were cupboards, just not what was in them. With the hidden safe you do not even know the cupboard exists. The attribute of the room, the safe, and so its contents, is not visible. Only a small select number of people know of its existence and can access it.

Hiding Behaviours

A person wearing dark glasses hides whether or not they can actually exhibit the behaviour of seeing. They hide the behaviour of seeing. Perhaps they are blind. They can then not perform the particular operation of seeing. You may be able to infer this from other behaviour, but perhaps they can achieve the same thing in a different way. In one episode of *The Avengers*, a blind man has arranged his room so he can negotiate it by touch and by knowing the positions of things, to the point that he can walk over to his firing range, pick up a pistol and score a bulls-eye. The fact that he was "seeing" using his other senses was not noticeable to anyone else. A car has visible behaviour like moving forwards but also lots of behaviour that is hidden. For example, when driving do you ever notice the behaviour of exploding that your car is repeatedly doing? On every cycle petrol is exploded in the engine but that behaviour is hidden.

Objects and Abstraction

Objects combine all these forms of abstraction. Instance variables give a name to be used in place of the value of an attribute. Similarly methods give names to be used in place of having to write out all the commands to do the action. However, the facilities for information hiding with objects are much deeper than this. It is also possible to hide the *existence* of an attribute (ie instance variable) or behaviour (ie method) to the outside world. That the behaviour or attribute even exists at all cannot be seen outside the object – just like the poker player hiding the fact that he is experiencing a range of emotions – not just what the emotions are, but that there are emotions being felt at all. When writing a program the programmer decides which attributes and behaviours need to be seen by the outside world

Polymorphism

Often the same basic algorithm works in a range of situations. By following exactly the same instructions, except for applying them to different objects, we can get the same effect on the different objects. For example, for example "slicing" is something that can be done on lots of different things. I can slice an egg, slice a tomato or slice a loaf of bread. If I was trying to teach someone how to do basic cookery, I could teach them how to slice each thing separately. One week we would go through slicing an onion, another time we would learn how to slice a boiled egg. However, the instructions I would give would be the same each time, whether it was egg or tomato being sliced:

Instructions for slicing a given thing

1. Place the thing on a chopping board.
2. Hold the thing with one hand.
3. Place a sharp knife just off one edge of the thing.
4. Make a downward, back and forth motion with the knife

In these instructions I use the word "thing" to refer to whatever it is I am trying to slice this time. What I have done is given a reusable set of instructions that can be applied to lots of objects. This is a little like the idea of method/function arguments: we write one set of instructions then apply them to different objects – we pass different values as arguments that replace the word "thing". For example if thing is a particular egg, then the instructions become as though we had written "hold the egg...". If slicing only worked for eggs we would have written:

Instructions for slicing a given *egg*

1. Place the *egg* on a chopping board.
2. Hold the *egg* with one hand.
3. Place a sharp knife just off one edge of the egg.
4. Make a downward, back and forth motion with the knife

However, it is more than just that going on in our original instructions. As we have discussed all objects have **types**: they belong to a particular class. Any particular egg in my fridge belongs to the class of eggs: it has type *egg*. If we just gave a normal method for slicing eggs, we could use it to slice anything in the class of egg: any given *egg* could be sliced that way, but that does not mean other kinds (ie classes) of things could be sliced in the same way. To be useful a method must work on any object of the same class. The egg slicing instructions should work for slicing any eggs. **Polymorphism** gives us a way of writing methods that work for different kinds of objects: not just eggs but tomatoes and bread too. The above instructions can be followed for a whole range of classes of object. Not only can the word "thing" stand for different eggs: objects of one particular kind, it can stand for different classes of objects too. A **polymorphic** function or method is one where the class of object it works on can vary.

As another example, consider the task of searching a series of things (perhaps that are in a pile) for a particular thing you have lost. We will look at searching in more detail later, but for now it serves as an example of a situation where a polymorphic set of instructions can be written. Whether the pile is a pile of DVDs, a pile of books, a pile of papers, or a pile of tins of food, the same algorithm can be followed.

```
1. Look at each thing in the pile in turn
2. If the thing currently being looked at is the thing you are looking for
     then stop – you have found the thing you were looking for
     else  move on to the next thing and repeat this step
```

Replace "thing" in these instructions by any class of objects: DVDs, books, tins, people, then it will still result in you finding the thing you looking for (as long as it was there).

There is a related but less powerful notion to polymorphism called **overloading**. With **overloading** the same name is given to instructions for (usually related) tasks on different kinds of things. This notion is different in that even though the names are the same the actual instructions are actually different: unlike in the above example, you really need to do something different for the different objects. For example, we talk about "folding up" a blanket but also "folding up" a deck chair. However we are using one phrase to mean two completely different tasks. We are **overloading** the phrase "fold up". If I gave you instructions for folding up a blanket it would not help you any if I then gave you a deckchair to fold up. Try it:

Instructions for folding a blanket.

1. Spread out the blanket on a large surface.
2. Take two adjacent corners of the blanket and move them to touch the opposite corners.

If this was just the same as polymorphism, I would be able to take the above instructions and replace the word "blanket" which indicates the class of things the instruction applies to and replace it with "deckchair" which refers to a different class of objects and get instructions that work:

Instructions for folding a deckchair.

1. Spread out the *deckchair* on a large surface.
2. Take two adjacent corners of the *deckchair* and move them to touch the opposite corners.

This makes no sense. **Overloading** is different to **polymorphism**. I cannot just replace the word "blanket" with "thing" and get a set of instructions that apply to different classes of thing. Therefore I can not write one set of instructions on how to fold things that can apply to both deckchairs and blankets. For any given set of "fold-up" instructions I would need to say what it was designed to apply to.

A truly **polymorphic** function would apply to absolutely any object: you could replcae the word "thing" with anything and the instructions would work. In practice, most sets of instructions in real life only work for related classes of objects. The slicing example only really works for food (and then only particular kinds) – it would not as it stands work for slicing planks of wood, for example. This leads to a further concept: **inheritence**.

Inheritance

When learning to be an expert in a particular area, it is usual to start by learning the general things that are relevant to the whole area. For example, if training to be a Heart Consultant in a hospital, you would first learn the same general medicine as someone who was to be a GP or a Paediatrician. Only later do you specialise. When studying to be a Computer Professional, you are likely to study many common things such as networks, programming, ethics and so on. In the second and third year you will probably specialise more and more. An advantage of structuring educational programmes in this way is that single modules can be used on a whole range of degrees. Everyone does the same first year modules. They are reusable. Designing a new degree? Then you can just use the existing modules. You do not to design a new first year – just use the existing one.

In the second description of the kitchen at the start of this chapter we suggested people were allocated to single kinds of objects: a cabbage expert or a carrot expert. More generally someone might be assigned to a whole class of things where similar skills are needed – vegetables rather than just cabbages. This is sensible since much that you learn and skills you acquire about one vegetable apply to others. For example, knowing that if you need to steam or boil vegetables it is best done in a microwave is a general fact that applies to a whole range of vegetables.

The idea of **inheritance** is similar. We define general categories of things – defining the attributes and behaviours that are common to anything in that category, then as we need more specialist objects within the category we can build on top of the basics. We create new categories that **inherit** the properties of the more general ones. The general classes can be reused in different programs. When programming we are writing instructions rather than just following some we have learnt, so the issues are about how you structure the way they are written down. The programmer is more in the role of the Academic designing the course rather than the student following it.

We previously discussed categories in the context of types. We argued that splitting things into classes is almost a natural human instinct. However we can go further, humans do not just divide the world into categories, they also divide those categories into sub-categories. The world contains animals, but animals can be mammals, birds, insects etc. Mammals can be marsupials or non-marsupials. Marsupials can be broken into categories such as Kangaroos, Kangaroos can be broken into Red Kangaroos or ... As we get to smaller and smaller categories, the attributes and behaviours become more specialised. Notice that we are talking about the categories: the classes of objects now, rather than any particular object. A particular kangaroo that we meet in the street will be an **instance** of all the classes. Skippy the Kangaroo is a Red Kangaroo. She is also a Kangaroo, and a Marsupial, and a mammal and an animal. At different times we might wish to refer to her as being in any one of those classes. For example in a discussion of rights: “Should she as an animal have the same rights of life and liberty as we assume humans have?”, we are happy to talk of the whole class of animal. In a discussion of reproduction we might talk just about her being a marsupial.

Similarly shapes can be split into circles, triangles and rectangles. These categories can be broken down further. A square is a form of rectangle. It has all the attributes of a rectangle – 4 sides, 90° angles and so on. However it is a more specialist case of the rectangle – it is a rectangle with all the sides the same length. Any particular square is a rectangle. However there are rectangles that are not squares. Thus the type-subtype relationship is one-sided. A subclass inherits the attributes of the class, but not vice-versa.

Notice that this idea of inheritance is different to the notion of an object having other attributes that are objects. Then we were talking about for example an eye being a part of a person, but that did not mean a person was an eye, or an eye was a person. However, a Kangaroo *is* an animal. Then we were talking about objects and their parts. Now we are talking about categories and how everything in a given category might also be in some other category of object.

In computing jargon we talk of defining **sub-types** of **types**. Classes are just a form of user defined type. Inheritance is about defining types in terms of more general types: that is defining classes from less specialist classes. We define a general type first, then define a more specialist sub-type by describing more specific attributes and behaviours.

Reuse
One of the most important issues when writing programs is that of reuse of code. Writing programs is time-consuming, expensive and difficult to do without making mistakes. One of the ways to avoid this is to reduce the size and number of programs you write by writing them in a way that allows reuse. Object-oriented programming is also primarily about making it easier to reuse code and many of the features covered in this chapter are important because they facilitate reuse..

Summary
Abstraction is concerned with making something easier to understand, change or use by hiding details. There are many different forms of abstraction. Examples include hiding the value of things, replacing them with a label, and hiding the details of the way something works.

Part 2: Data Structures

10. The Pecking Order (List and Array Data Structures)

Blot out, correct, insert, refine,
Enlarge, diminish, interline.
J. Swift, *On Poetry*, (1733) l.85.

Lists
One of the most common structures, both in Computer Science and in real life is the list. Write down a list of your all time favourite records. The list I thought of was (no doubt you disagree):

1. *Bohemian Rhapsody* - Queen
2. *Stairway to Heaven* – Led Zeppelin
3. *Your Song* – Elton John
4. *Who made who* – AC/DC
5. *Sweet Child of Mine* – Guns 'n Roses

A list is just a sequence of things in some order.

After writing this list I could not think of any more, but later in the evening I realised I had missed out *Every Breath You Take* by the Police, and Bruce Springsteen's *The Ghost of Tom Joad*. My new list is

1. *Bohemian Rhapsody* – Queen
2. *Every Breath You Take* - Police
3. *Stairway to Heaven* – Led Zeppelin
4. *Your Song* – Elton John
5. *Who made who* – AC/DC
6. *Sweet Child of Mine* – Guns 'n Roses
7. *The Ghost of Tom Joad* – Bruce Springsteen

Lists can thus have new entries added anywhere, including in the middle, and their length can increase as needed.

In fact on second thoughts, Guns 'n Roses are not that good, so they should not be in the list at all:

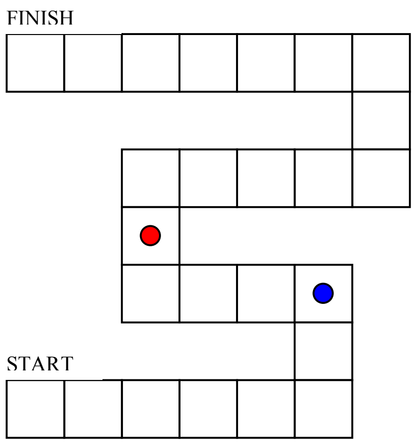
1. *Bohemian Rhapsody* – Queen
2. *Every Breath You Take* - Police
3. *Stairway to Heaven* – Led Zeppelin
4. *Your Song* – Elton John
5. *Who made who* – AC/DC
6. *The Ghost of Tom Joad* – Bruce Springsteen

Entries can be removed from lists.

We use lists all the time. Whenever I go shopping, I take a list of things to buy, as I have a terrible memory and would otherwise be bound to come home without the main thing I went for. I do not go round the shop finding the items in the order they appear on the list as I am also lazy and that would involve walking round the shop several more times than I need to. As I walk past an item I take it and remove it from the list by crossing it out. Again things are being removed from the middle of the list.

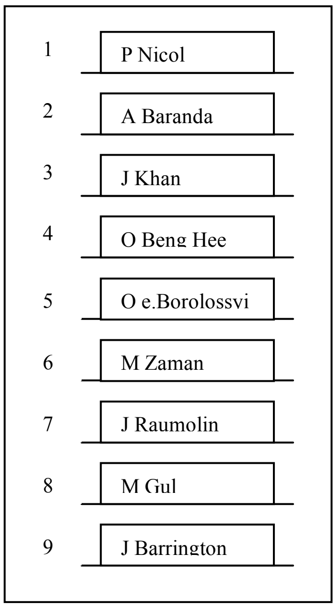
The "To-Do" list is one of the most common ways that people use to organise their work or studies. It simply involves keeping a list of the things that need to be done. When a new task arrives, due perhaps to an email or phone call, it is added to the To-do list. When a task is finished, it is rather satisfyingly crossed off the list. This way nothing is forgotten. Most people keep their To-do lists on a piece of paper. Periodically the list gets in such a mess that they copy out the items remaining on to a fresh piece of paper. This reflects the fact that a list is a dynamically changing thing, but that paper is static. One way devised to get over this problem is to use post-it notes (small slightly sticky squares of paper) instead of paper (Norman, 1993). Norman describes post-it notes as "*the decade's most important artefact*". This is in part due to their ability to implement **dynamic data structures**: ever changing structures. They can be arranged along the edge of a shelf, on the desk, on the monitor, etc. Each new task is placed on a post-it note. The medium used to store the list is now as dynamic as the list itself. Items in the list can be shuffled about. Deleting an item involves simply removing and throwing away that post-it note. Items no longer need to be repeatedly copied onto a new sheet.

Lists can also be fixed, never changing in length, though with the values of the entries changing. Many simple children's board games are based on list-like structures. Remember all those games you played which involved racing round a snake-like board where how far you moved depended on the throw of a dice. The board is just a list of squares, the list entries being either empty or holding pieces.



Lists can, in general, increase or decrease in length with the extra ones being added at the start, end or anywhere in between. Elements in a list have a position, but when something new is added the positions change. By making restrictions or generalisations on the idea of a list, we get a range of different data structures with different uses as we shall see.

Arrays
A common variation on list-like structures is that of an **array**: a fixed sized list whose elements are accessed by labels or **subscripts**. Most squash clubs operate a squash ladder (similarly chess clubs). This is physically a card with numbered slits in. Each member of the ladder writes their name on a tabbed card that fits into the slots. If you play someone higher than you on the ladder and beat them, you swap places, the aim being to be the person at the top of the ladder. The labels on the slits thus give each player their current rank in the club.



The squash ladder is an array structure. All the entries are of the same kind: they are all names of squash players. Each position is numerically labelled, so you can easily ask questions of it such as who is the current No 1 or who is three positions higher than me? The ladder also has a predetermined number of positions. If there are more people wishing to be on the ladder than there are slots, a waiting list may be set up, or a second (division) ladder created, but the original physical ladder cannot have new slots added as there is no space. When a new person arrives who is known to be good, a new slot cannot be created in the middle of the ladder to start them in a position close to their correct one. The best that could be done if this were required is to shuffle each player down one position to create a gap. This would be time-consuming.

At sports events such as ice-skating (gymnastics and boxing are similar), marks are awarded by a panel of judges. The judges sit in a line of seats, with the seat labelled by their country. Before technology took over, the judges would hold up cards with their scores. On the television screen each skater's scores are given in a row, again with the marks. This is another example of an array: a fixed size list of things of interest (such as scores) each with a label to identify them (such as the country).

6.0	5.9	5.4	5.9	5.8	6.0
GB	FR	DE	US	GR	AU

In ice skating scores, the "things" are thus numbers and the labels are letter pairs that indicate the countries of the judges.

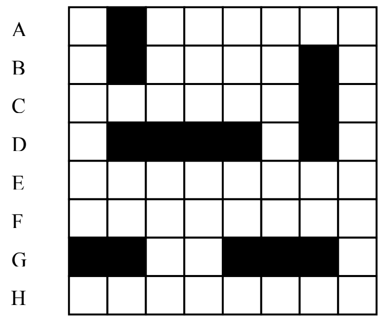
A row of houses down one side of a street is also organised like an array. Each house is labelled by its postal address: for example 2, 4, 6, 8, etc. assuming it is on the even side of the street in Britain. When a postman has a letter to deliver, they know where it goes from the number. The number is just a label; the actual house is the thing it is labelling. In normal circumstances, new houses are not added in the middle – once your house is given a number, it is not changed.

My photo albums are similar to an array. They are “flip albums”, with leaves for holding each photo. I sort through my photos when they come back from the developers and put them in albums in the order they were taken. Putting the films into the individual slots is very time consuming taking time proportional to the number of photos taken. When I go on holiday however I often take dozens of films. Sometimes one of the films gets left in a coat pocket and is not developed with the others. It of course turns up just after I have put the other photos into the flip album. I then have the problem of putting the new photos in their correct position somewhere in the middle. I must find the correct position for the new photos and shuffle many of those already placed up to make space. This takes as much time per photo moved as originally – the original time was wasted. Arrays have the same problem, to put new things into the middle without just overwriting the existing ones (putting new photos so they cover the old ones for example) takes a lot of time. We will look at linked list data structures that overcome this problem of arrays in a later chapter.

A one-dimensional array such as the examples above is in many ways similar to a list. The difference is that arrays are of fixed sizes, with each cell "named" by a subscript. An array consists of a fixed number of slots, that things can be put in or taken out of. New slots cannot however be created.

Multidimensional Arrays
Arrays do not need to be just a single row of things. They can also come in grids or tables – a whole series of rows like a chessboard. To allow for games to be recorded, each square on the board has a label, but here the labels are pairs. One part of the label gives the row of the square and the other the column. In chess the rows are given numbers from 1 to 8. The columns are labelled by letters a to h. Thus a common first move is for white to move their pawn from e2 to e4.

The game of Battleships is similar. In this game, you each position battleships of various sizes on a grid (a 2-dimensional array), then try to find the other person's ships. You do this by calling out the grid reference (subscripts) of squares such as "B3" where you think the ship might be. If part of a ship is there you are told you have a hit, otherwise a miss.



1 2 3 4 5 6 7 8

Cinemas and theatres use a similar system to label seats, with a letter indicating the row, and a number giving the seat with in the row. To find your seat you find the row labelled with the letter that is indicated on your ticket. You then look at the numbers on the street backs to find your seat. Cinemas are thus just arrays of people.

Arrays have a fixed size. In general you cannot add extra seats in a cinema when it gets full. You have to turn the extra people away. If you are in a group of three, and the cinema is so full that there are only pairs of seats left, you cannot add an extra seat in the middle so you can all sit together. You could try asking everyone in a row to move up one, but that would make you unpopular, especially if the film had started as it takes time and is disruptive. Such are the problems of arrays.

Maps also often label positions on the map by super-imposing a 2-dimensional grid over the map. The index in a London A-Z map for example tells you where each street is on a given page. This is done by giving a column letter from A to K and a position number from 1 to 7. This narrows down the street to a single square. In fact an A-Z book is actually a 3-dimensional array as it is a whole series of pages stacked on top of one another. Any particular square in the book is accessed by giving three labels, rather than just two. The third label is the page number. Each page is labelled in exactly the same way using the further two labels.

Arrays do not need to stop at 3-dimensions. For each extra dimension we need an extra set of labels. The company that produce the London A-Z book also produce A-Zs of many other cities. The whole collection of A-Zs sat on a shelf is organised like a 4-dimensional array. The fourth set of labels is the name of the city. For example, the position of Nelson's Column is given by the four labels (London, 77, 1, H). The Crucible Theatre in Sheffield where the World Snooker Championships are held each year is in the square (Sheffield, 4, 3, F).

A final example of a 2-dimensional array is the multiplication table. It is a constant array (its values never change) that is considered so important that we are made to learn it off-by-heart in childhood. The subscripts to this array are the 2 numbers we wish to multiply. The value in the cell is the pre-computed answer to each multiplication.

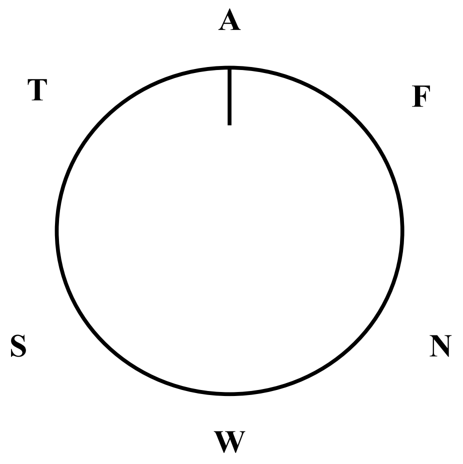
Look-up Tables

Lookup tables (in some circumstances called **bucket arrays**) are really just arrays, where the subscripts correspond to known information, and the array contains further information that might be looked up in the table.

The children's book, "The Pop-Up Mice of Mr Brice" (Dr Seuss, 1998) contains a lookup table. It is about a house full of 26 mice. Their names each start with a different letter of the alphabet. At the beginning is a pop-up picture showing twenty six doors, each with the name of a mouse and corresponding letter of the alphabet. The doors can be opened to reveal a message. Open Ursula's door to find the message "Ursula is out". Open Xavier's door to find the message "Xavier is in". The sequence of doors are being used as a lookup table. The subscripts are the letters of the alphabet, representing the names of the different mice. The contents of the lookup table are the messages. If you know the name of a mouse, you can use the table to find out whether they are in or out.

There is a lookup table on the front of my washing machine at home. The washing machine has many different settings depending on whether I wish to wash whites or coloureds, hot or cold, etc. Each programme has a separate position on a dial. Somehow I need to easily work out the correct setting for the current wash. Each position of the dial is labelled with a letter. On the front of the machine is a table describing the programme that each letter stands for. It is a look-up table. To do a wash, I find the entry in the table for the programme I want, and it tells me the letter I should set the dial to.

Whites	95	**A**
Fast Coloured	60	**F**
Non-fast Coloured	40	**N**
Woolens	40	**W**
Spin Dry		**S**
Tumble Dry		**T**



A multiplication table is similarly also a lookup table. It allows you to look up the answers to multiplication problems that have been pre-computed. Most multiplication tables stop at the 12 times table. Why do you rarely see a 20 times table or a 100 times table? This is because lookup tables use up a lot of space, either on paper on in your memory if you rote-learnt it as a child. You therefore only want to use it as a look up method if you will be using its contents many times. This is likely for the entries in 10 or 12 times tables but larger values are likely to be used less, so you find a different method when this is needed (long multiplication for example).

Maps also contain lookup tables, usually called *Legends*, which tell you what each symbol used on the map means. A symbol such as a black dashed line can be looked up in the table. There its meaning, a footpath, is found.

Adjacency Matrices

A special form of 2-dimensional array /lookup table is known as an **adjacency matrix.** These are used to indicate when pairs of things are connected in some way, possibly giving some information about the connection. The two sets of labels used are the same in this case. This kind of structure is often used in road atlases to give distances between cities. Both the rows and columns of the table are labelled by the same set of cities. To find the distance between two cities, you look up the row of the place you are starting from and the column of the place you are going to. The number in the table where they meet gives the distance between them. This is an adjacency matrix, where every pair has an entry, since a distance is known for every pair of cities.

Several newspapers give the season's football scores in an adjacency matrix that is updated each week. The rows and columns are labelled by team names. The rows represent the home team and the column the away team. Blank entries indicate that the teams have not met. If the teams have met then the score is given as the entry. Here two teams being "connected" means that the corresponding match has been played. The same information could just be given as lists of the scores from each week. This would make it easy to ask questions such as "who played who in the 4th week of the season?" or "what were this week's scores?" However, it would take more space, and would make it harder to find the score of any particular team. It would also be harder to ask the question as to whether two teams have played yet. The organisation of data chosen thus depends on what question is being asked. Papers that include an adjacency matrix also give the current week's scores as a normal list as most readers are interested mainly in the current week's scores. Different readers at different times want different information, so need different data structures.

Records

A **record** is a very common list-like structure that is used to package lots of different information together to describe an object or entity. A record differs from an array or list in that the order of entries has no significance. Entries are usually accessed by named labels. Furthermore, each entry in an array holds the same kind of information as every other entry: all entries are names of football teams or all entries are numbers. A record differs in that the different entries can be different: one a name, another a number. Each entry in a record is called a **field** and the labels are **field identifiers**.

Record-like structures abound in everyday life. Dates are one example. A date might consists of a day field, a day of the month field, a month field and a year field, such as:

Saturday 28th August 1999

The order we write the entries does not matter:

Saturday August 28th 1999

is still the same date, for example.

Times are also given as a record such as:

12:03 PM

Here we have given an hour, a number of minutes and an indication of morning or afternoon.

Tracks on CDs are often described by records (of the computer science rather than the vinyl kind) on the cover. For example the 7th track on the album *Rumours* by

Fleetwood Mac (which happens to be the old BBC Grand Prix Theme Music) is described by a record consisting of the track's name and its running time:

The Chain 4:28

Since a time can be considered as a record, this is an example of a record where one of the fields is itself a different kind of record. The inside sleeve of the Album *Harvest* by Neil Young uses a longer record with an extra field to give additional notes about other contributors to the track. For example, the track *Heart of Gold,* is described by the record:

Heart of Gold
3:05
with the Stray Gators / additional vocals by James Taylor & Linda Ronstadt

Whenever you fill out a form, you are filling in a record. Each entry on the form is a record field. Here the fact that fields are labelled is made explicit since the labels such as "Name" are written next to the boxes. I recently filled in a Birth Certificate for my daughter or rather I supplied information to a Registrar who filled it out in front of me. It had fields such as name, date of birth, place of birth, mother's name, mother's occupation etc. A Registrar is someone whose job is thus to fill out records about the major events in people's lives: births, deaths and marriages. These records have been filled out for hundreds of years for every person in the country. It is these records that my father searched through to build up my family's family tree back to the 15th Century. Unfortunately he still has not shown that I am related to the former Viceroy of India, George Nathanial Curzon. Nor can I be sure I am part of the Norman family who came to Britain with William the Conqueror and whose stately home is Keddleston Hall. To date I just know I am descended from a family of Lead Miners.

Records are often grouped together in other structures. We have already seen records inside records. We also often have an array of records. The Football League tables given each week on Grandstand and in the Sunday Newspapers are an example of this. Each line in the League table is a record, giving for example, the name of a team, the number of games played and points scored. Each line describes an entity: a football team. These records are then grouped together in an array. Notice that each entry in the array is of a single kind: a football team record, even though each record is combining several different pieces of information. Take for example the fantasy Premiership table below. An example record is:

Newcastle United: Played 2, Points 4.

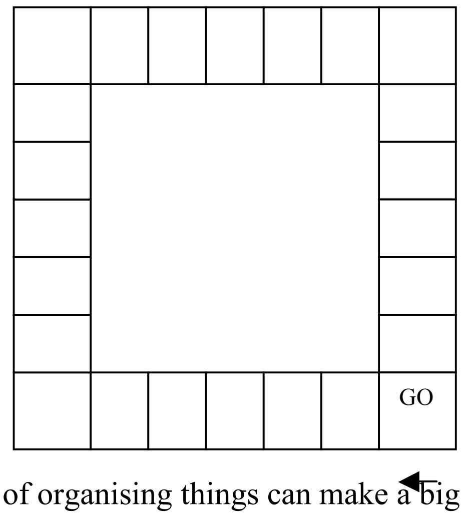
This record is the 3rd entry in the array. Each record has fields: Team, Played and Points. The array has subscripts from 1 to 7.

	Team	**Played**	**Points**
1	Leeds United	3	9
2	Sheffield Wednesday	3	5
3	Newcastle United	2	4
4	Chelsea	2	3
5	Liverpool	2	2

6	Arsenal	3	1
7	Manchester United	3	0

Circular Lists

When you think of a list, you probably think of something with a start and an end, but it is possible to have never-ending lists. An example is seen in the board game Monopoly. It has a board that is just a list of squares representing the streets of London. However, the last square, *Mayfair*, is connected back to the start square, *Go,* in a loop, so do not get to the end of the game by crossing a finishing line. There is no end point – you could travel round and round forever. The game finishes when someone is bankrupt, not when a finish line is reached.



Summary

Choosing a good way of organising things can make a big difference to the task of manipulating those things. Lists and their variations are some of the simplest and most common ways of organising things.

A **list** is just a sequence of things in a given order. In general a list's length can increase or decrease. New elements can be added to or removed from the start, end or in the middle. However, a list does not have to change. The elements of a lists are usually accessed in order.

An **array** is a list of a fixed size where all entries are the same kind of thing. It can neither grow nor shrink though its elements can be changed. Its elements are accessed using position labels (know as **subscripts**). Given a position the element at that position can be accessed. In a **multidimensional array** each position is identified by more than 1 label. For example, in a 2-dimensional array, 2 labels are needed to identify each element – a row label and a column label.

A **lookup table** is an array used to look up information about something. The things we want to find information about are used as subscripts. The array contains the information that is to be looked up.

An **adjacency matrix** is a 2-dimensional lookup table that contains information about the connectivity of a set of things. The same labels are used for both sets of subscripts. Entries in the table indicate whether the pair given by its position labels are "connected" or not.

A **record** is a collection of items that give information about a single entity. The separate pieces of information are called **fields**, and they may be of different types. The positions of the fields are not relevant as they are accessed solely by **field identifiers**.

A **circular list** is just a list, whose last element is followed by the first element. It is thus a never-ending list.

11. The Other Queue Always Goes Faster (Queues and Stacks)

Waitin' for when the last shall be first and the first shall be last
Bruce Springsteen, The Ghost of Tom Joad,1995,
cf *The Bible*, St Mathew ch. 19, v. 26.

Queues and queuing are an integral part of our Society, especially for the English since if the stereotype is to be believed "*An Englishman, even if he is alone, forms an orderly queue of one*" (Mikes, 1946). There are many different ways to organise a queue however. Here we shall look at the three main types.

FIFO Queues
When I was at University, every year there was a draw held for accommodation. The higher up the draw you came, the greater the choice of accommodation you had for the following year and so the better room you got. Each week the next 20 names of the draw were announced. Even though each group of 20 names was drawn in order, that order was not used. Instead when the names were announced, each of the 20 had to fill in the form with their choice and get to the Accommodation Office as fast as possible. Whoever could run fastest (preferably while filling in the form at the same time) got first choice. The result was fairly chaotic.

Think of a group of people in a bank waiting for a cashier to come free. One way the people waiting could be organised is in a similarly random way – when a cashier comes free whoever gets there first is served next. Banks do not work like that, because fights would break out (as came close to happening outside the accommodation office). Instead, Banks organise the people. One way is to have a separate queue in front of each cashier. That is one data structure. However, it is very infuriating if you end up in the wrong queue, behind the child paying in 2000 5p pieces from their piggy bank that need counting! Banks do not want their customers angry, so they often use a slightly different organisation (i.e. data structure). They have a single queue, with the person at the front going to the next free position. That way it is fair for everyone.

Queues can thus be arranged in many different ways. The above queues have something in common, however. They are attempting to organise the people in a way that is fair in the sense that they are served in the order that they arrive. The first person to arrive is the first person to be served. This kind of queue is known to Computer Scientists as a **First-In-First-Out queue**, or **FIFO queue**. The queue at a bus stop is another typical example. In fact most things that we refer to as "queues" in every day life are FIFO queues. It is not the only form of queue however as far as computer scientists are concerned as we will see later.

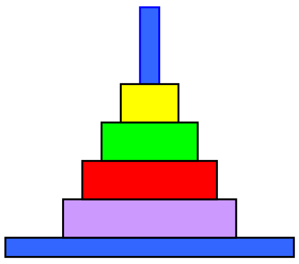
Chocolate, crisp and drink vending machines also use FIFO queues. They are filled from the back, with the next item being released from the front. They are not filled from the front because then some stock could remain in the machine for a long time, assuming the machine was rarely allowed to go empty. Items might then pass their sell-by-date. Supermarkets stock their shelves in the same way, adding new items with the longest sell-by-dates to the back, so the earlier sell-by-date items that were put there first are taken first from the front. Despite the slogan *Stack 'em High, Sell 'em Cheap*, in Computer Science terms, supermarkets use FIFO queues not stacks, which are something different.

Stacks

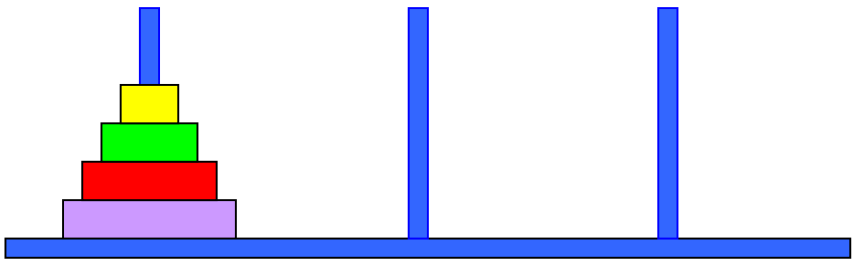
Think of a stack of chairs. They are also organised in a queue structure. However, it is a different kind of queue. It has the property that the last chair added to the stack is the first one to be removed. Try taking the first one that went on to the stack out first in a FIFO way. **Stacks** are **Last-In-First-Out Queues** or **LIFO queues**.

Stacks appear frequently in card games. For example, games like Gin Rummy have a discard pile. Players discard cards on to the top of the discard pile. A player can also pick up the top card from the discard pile and add it to their hand. They can not take any card that is not at the top. Similarly they cannot place a card into the middle of the discard pile. The discard pile is a LIFO Queue. The solo card game of Patience is also based around stacks. The cards are organised into a series of piles. Only the top card of the pile can be removed at any time and cards can only be placed on the top of each pile not in the middle. Each pile is thus a stack in Computer Science terms.

The way that stacks work is so important that most children are given a stack or two to play with by well-meaning relatives when they are toddlers! They obviously do so hoping the child will grow up to be a Computer Scientist! The toys that consist of a plastic pole on a base with a series of multicolour Polo-like disks are really just stacks. The disks can be threaded onto the pole, but can only be removed in the opposite order to the order in which they were placed. By playing with them, toddlers learn the rules for adding and removing elements from a stack.



A similar adult puzzle that relies on stacks is the Towers of Hanoi game. This game consists of three poles together with a series of disks of different sizes. To start the game the rings are placed on to the first pole in order of size with the biggest at the bottom. The aim of the game is to move all the disks to the third pole. However, only one disk can be moved at a time. Furthermore, at no time can a ring be placed on top of a smaller one. Again each pole is acting as a stack for disks though with extra restrictions.



One of the most unpleasant stacks is found in struggling businesses. The law requires that when people are made redundant, people are chosen for redundancy by some fair method. The most common "fair" method is called LIFO – last-in-first-out. The newest employees are the first to go. It is just a stack again.

Archaeology also works on a LIFO queue basis. As time passes, the layers of civilisation build upon the ruins and foundations of earlier generations. Centuries later Archaeologists start to remove the layers using fine tools and brushes. Thus the last layers added are the first layers removed. The most interesting things are likely to be the earliest, so the temptation must be to dig deep and start near the bottom in a FIFO manner, but this would risk damaging important finds, so a rigorous LIFO regime is followed.

A final place where stacks are evident is in the undo buttons in word processors and similar software. A stack of each operation performed is kept. Each time you type something a record of it is added to the stack. These actions can be undone, but only in the order that they were done. If you want to undo something you did 10 steps earlier, but leave everything else as it was, you cannot do it using undo. You would have to undo all 10 steps and start again from there. That is because only the top action can be removed from the stack at any time.

Priority Queues

A third kind of queue is known as a **Priority Queue**. This is a queue in which entries in the queue are given priorities and are dealt with in order of priority. This is the kind of queue often used at airline check-in desks. Airlines divide their customers into a series of classes such as "economy", "business" and "first" class. The higher the class, the more the customer pays. One of the things they are paying for is to be given priority so they do not have to queue for as long. Economy customers like me often have to queue for an hour or more to check in. First class customers expect not to have to queue at all. Business class passengers tolerate short queues but nothing more. If the airline gets it wrong they will lose their high paying customers to other airlines. They therefore do not use a FIFO queue, but use a priority queue. Each customer is given a priority corresponding to the class of their ticket. The check-in desks are then run in a way that ensures that passengers are served before those of a lower class. This is usually done using separate check-in desks for the different classes of passengers. The first class desk will only check-in business class customers if there are no first class passengers waiting. The business class desk will only process economy class customers when no business class passengers are waiting. You only have to wait if someone of equal or higher priority arrived before you. Thus the people of a single priority are arranged within a FIFO queue, as they are served on a first-come first served basis. However pushy he may be, a first class customer will not get served before another first class customer who arrived earlier.

Hospital waiting lists are also run on a priority queue basis. When someone is placed on a waiting list for an operation or to see a consultant, the urgency of the case is noted. If the complaint is minor, and the person in little pain, they will wait longer than someone who could die if their operation does not take place. All cities have a special high priority Casualty department, so that injuries that need immediate attention can get it. In recent years, there have also been suggestions that in Britain the patients of GPs from fund-holding practices have been able to jump to the front of the queue. Private patients have always been able to do this, as access to consultants is partly what they are paying for.

To-do lists can be organised as priority queues. This is especially easy if post-it notes are used. Most to-do lists contain urgent items that need to be done today, and less urgent items that do not need to be done for months, if ever. Ideally the most important or urgent things should be done first, irrespective of when they arrived. If your Boss phones to say he needs a new report by the end of the afternoon, you jump and ignore whatever you were doing. When a new task comes in, its post-it note should be placed in a position relative to its urgency or importance. The most important things are at the front of the list and they are done first. This is also a situation where the priorities of things in the priority queue may not be fixed. If your Boss phones and says the deadline for a report has changed from next week to first thing in the morning, its priority changes accordingly. The post-it note moves to a higher position in the list.

It is also possible to have a 2-dimensional priority queue, where each thing in the queue is given more than one priority assigned to it. Our to-do list could be organised like this. We spoke of two reasons for things to have high priority: urgency and importance. It is possible for something to not be too important, but if it is going to be done, it must be done by the end of the day. An example might be if I had an idea for a new exercise to give to a class. I already have other exercises, so it is not too important if I were not to turn it into a hand out, and perhaps I could do it on the board. However, if the class is tomorrow morning, if I am to do it I must do it today. It is urgent but not important. Writing exam questions on the other hand is important but not (right now at the start of term) urgent. I have to do it or I will be sacked, but not right now. If I give each task an urgency rating and an importance rating, I then can ensure I do not miss deadlines as well as ensuring that all critical things are done. A list is no longer the best way to organise my post-it notes. Instead I organise my priority queue as a table (i.e. array) with importance along one axis and urgency along the other. Post-it Notes are put in the boxes according to their urgency and priority. The thing I do next (the first thing out of the queue) is anything in the *very urgent – very important* box. If that is empty I then remove things diagonally *urgent – very important* and *very urgent – important* next and so on.

	very important	**important**	**not important**
very urgent			
urgent			
not urgent			

Summary

Queues are just lists where restrictions are placed on how elements can be added or removed.

A **FIFO queue** is one where entries can only be added at the end of the list and removed from the front. The first element to be added is thus always the first removed.

A **LIFO queue** or **stack** is a list in which elements are added and removed from the front: the last element added is always the first to be removed.

A **priority queue** is a list where the elements have priorities associated with them. The element with the highest priority is always removed first. If more than one element has the same priority, they are removed on a FIFO basis.

12. Seeing the Wood for the Trees (Dynamic Data Structures)

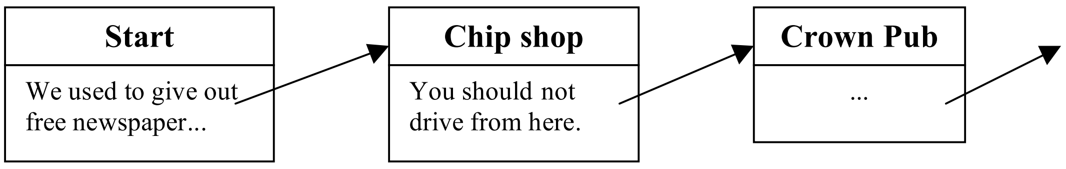
It is often better to be in chains than to be free.
Franz Kafka. *The Trial* (1925) Ch. 8.

Linked Lists
Queues and stacks are variations on lists where entries can only be added and removed in certain ways. Another variation on a list is known as a **linked list**. A linked list is just a list whose entries are spread around rather than being placed together. Each entry contains something indicating where the next one is so that the list entries can still be followed in turn.

Suppose you were organising a summer fete. As one of the events, you decide to hold a quiz about your local village high street (these days perhaps it should be about your supermarket) where each question is on a particular location. You could just give out to entrants a piece of paper with the questions on. This would be a list (or perhaps an array if the number of questions were fixed) in the sense of our previous discussion. An alternative however, would be to turn the quiz into a treasure hunt. You would then initially give out just a single question rather than them all. For example,

1. *We used to give out free newspapers but no one ever read them.*

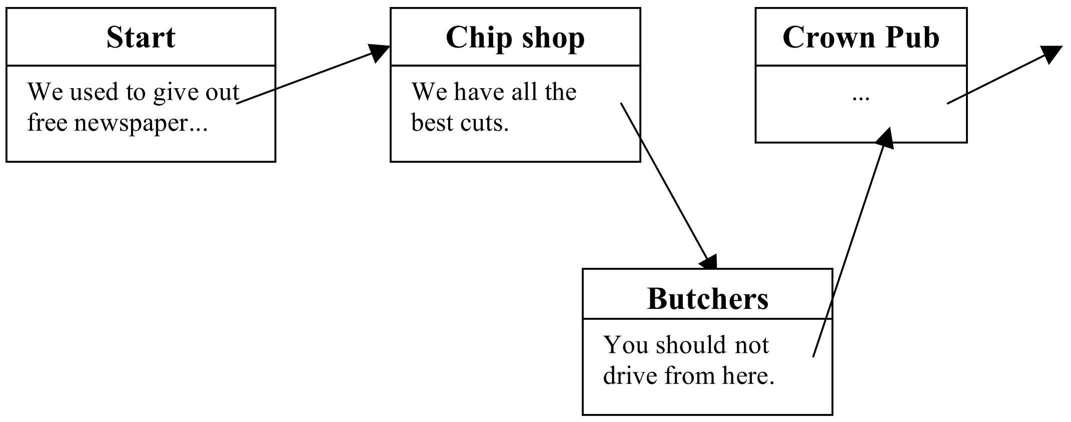
The answer to this would be the fish and chip shop, since fish and chips used to be wrapped in newspaper. To get the next question, you would need to correctly work out the answer to this question, and then actually go to the fish and chip shop. At the Fish and Chip Shop you would find the next question. Its answer might lead you to the pub, and so on. Each question is pointing to the next location. The different buildings are forming a linked list.


Start
We used to give out free newspaper...
Chip shop
You should not drive from here.
Crown Pub
...

By representing the information in a linked list, the quiz has changed. Now you can not just ignore questions. The only way to know what Question 3 is, is to go to the location pointed to by Question 2. To get to the end of the treasure hunt (or even know where the end is), you must go to every node in the list in order.

Suppose having designed the treasure hunt, and put out all the questions round the village – ie set up the linked list structure – you fall ill. The local butcher takes over the organisation from you. He wants to boost his trade so he decides to add an extra question about his own shop. Not wanting to be too obvious, he makes it the answer to the second question. However, since the treasure hunt was already set up, you only gave him the first question not the other locations. How does he add his question?

He must follow the chain to the chip shop, remove the question that is there, pointing to the pub and replace it with a new question pointing to his Butchers shop. He then follows the link to the Butchers and puts the question about the Pub he removed from the Chip Shop at the Butchers.

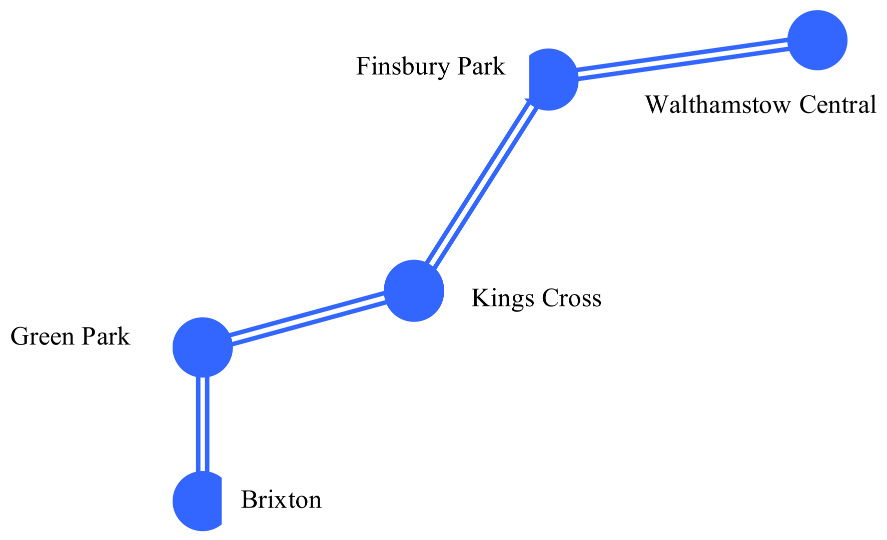

Start
We used to give out free newspaper...
Chip shop
We have all the best cuts.
Crown Pub
...
Butchers
You should not drive from here.

Adding a new entry in any linked list is done in the same way. We must follow the links to the node before the right position, and put pointer to the new position there. We then put the pointer that was there in the new location to re-establish the broken chain.

Orienteering courses also use the linked list structure: a series of places linked together in a chain. Orienteering is the sport where you have to map read you way round a course. It is like cross-country running except that you do not have to stick to paths and are not told the route to take. You are just told the series of points you must visit and the order you must visit them. You can either try to go in straight lines (through streams and thickets, down steep valleys only to go straight up the other side), or you can navigate round obstacles (going further to a bridge or contouring round the valleys). You are given a map at the start, with all the points you have to visit marked on it. The fastest person to complete the course wins. This is not necessarily the fastest runner, but the best combined runner and navigator. Orienteering is thus the sport of following a linked list through forest and moors as fast as possible. This is made more obvious in the string courses. These are special orienteering courses put on for toddlers at big events. A large cut-out figure of a cartoon character, or similar, is placed at each site that must be visited. A long piece of string is then run all the way round the course, linking the cartoon characters together. Each toddler is shown the end of the string at the start, and their task is to follow it to each of the characters. The characters are the nodes of the linked list, the string the links.

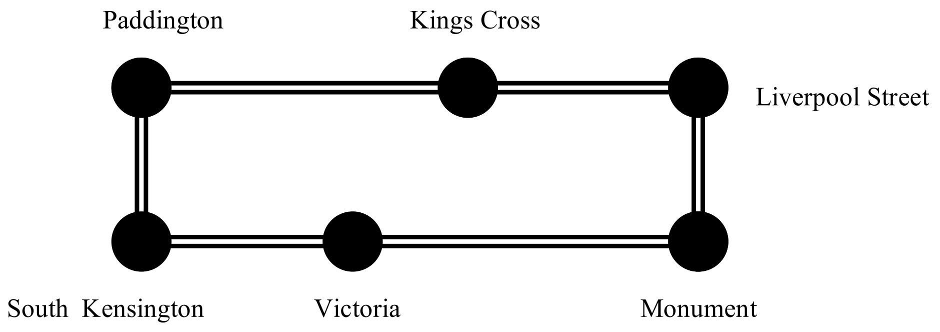
Doubly-linked Lists

The Victoria Underground line in London is also a linked list. It starts at Walthamstow Central, then chains together a series of stations in a single line, finishing in Brixton. The stations are the nodes of the list and the track the links. In fact it is a **doubly-linked list**. In all the previous examples of linked lists, the links are traversed in one direction only. A person travelling on the Underground can travel in either direction. Several times, I have been so engrossed in the book I was reading that I have missed my stop. This is not a problem as I can just hop platforms and travel back in the opposite direction along the same link.



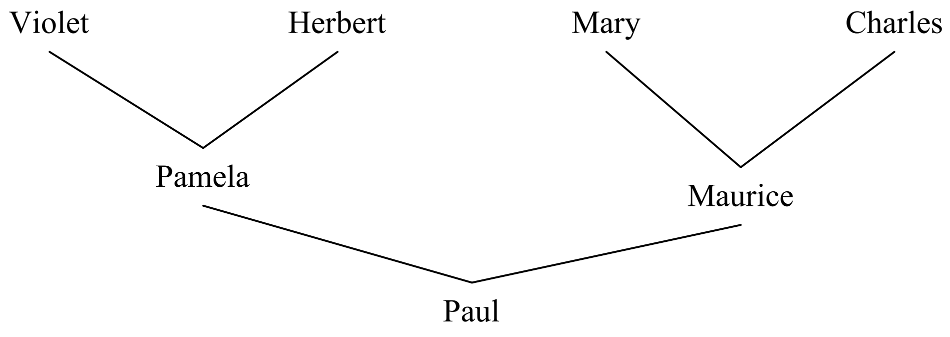
Circular Linked Lists
Linked-lists can be never-ending just like normal lists. An example of a **circular linked list** from childhood is the Daisy chain. A series of Daisy flowers are linked together into a necklace by piercing a hole in the stalk with your finger nail, and threading the next one through. Think of the flower heads as the nodes and the stalks as the links. Whilst being made they form a normal linked list. Once finished and the last daisy is threaded through the first, you have a circular linked list.

Back on the London Underground, the Circle Line is a **circular doubly-linked list**. Trains travel in either direction round a never-ending loop. Break the loop and you get a normal doubly-linked list again. Thus if a part of the Circle line needed to be closed for repairs, to get between what used to by adjacent stations, you would instead have to travel the full length of the line unless you switched to a different line.



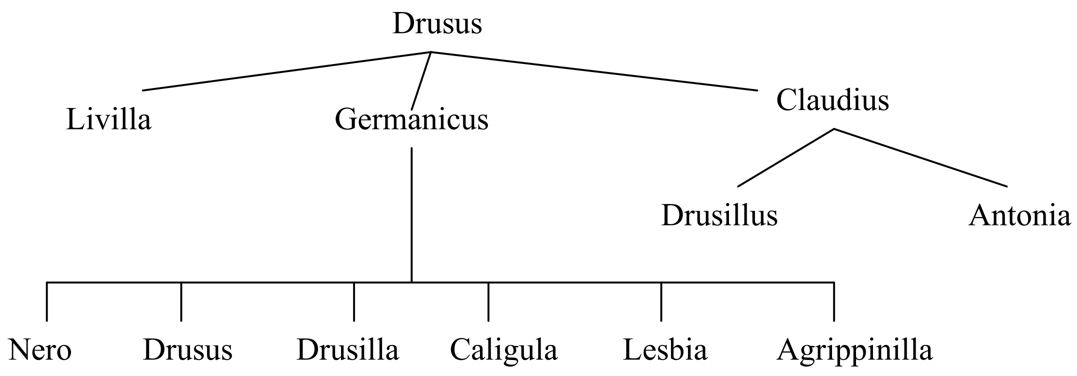
Trees
Whilst list and array like structures are common there are other non-linear ways of organising information that can make the structure easier to use. When someone tracks down their ancestry, the way they represent the information collected about their grandparents, great grandparents and so on is not as a list but as a family **tree**. At the bottom of the page you write your name, placing your mother and father above

you, each connected by a line to you to show the relationship. Their parents are each placed further up the page, and so on. In this way you get a tree like structure, with information (here the names of people) stored at the joints. The bottom-most joint (with your name in it) is known as the **root** of the tree. Each junction is called a **node** and the lines joining nodes are called **edges**.



This tree is a **binary tree** since each person only has *two* (genetic at least) parents so each node has two edges coming out of it.

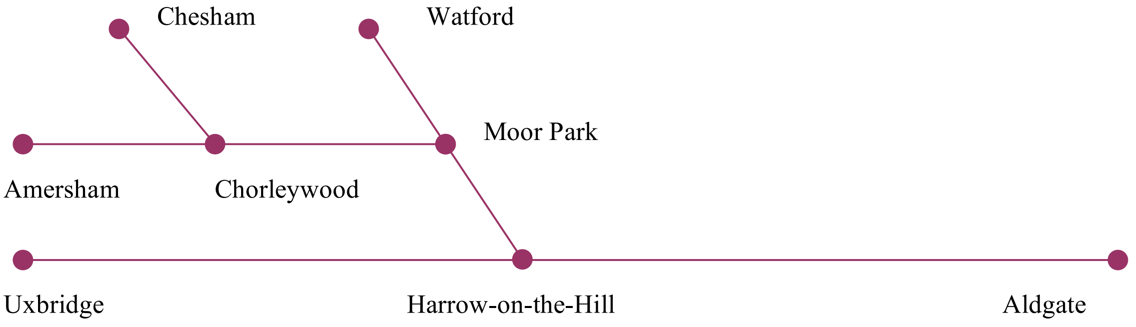
The similar problem of representing the descendants of a famous person from history would also normally be given as a tree. This time however it is not binary, since now we link a person with their children, and there is no fixed restriction on the number of children a person can theoretically have.



The Descendants of Drusus:
A Portion of the Family Tree of the Roman Emperors
Adapted from (Graves, 1941)

Taxonomists also use tree data structures to classify living things. Each level of the tree is given a name. The first level gives the *Kingdom* (plant or animal), the next the *division*, and so down to the individual *species* (Human, Herring Gull, Oak, etc). This tree structure was not the first way that living things were classified. Early attempts including listing things by properties such as their colour or sexual characteristics. These other ways of organising the information were not very successful, however. The reason that a tree structure works best is because it mirrors the way living things evolved. Species within the same family had a common ancestor, later than those of different species. It is thus another form of family tree.

Rail networks are sometimes constructed as trees. This allows trains to travel from end to end, but with outlying areas having fewer trains visit. The London Metropolitan Underground line is an example of this. It starts at Aldgate, in the city (its root) and through most of its length is a single chain of stations. However, at Harrow-on-the Hill it branches, with one branch continuing to Uxbridge. The other branch splits again at Moor Park, to Watford, and again at Chorleywood to Amersham and Chesham.



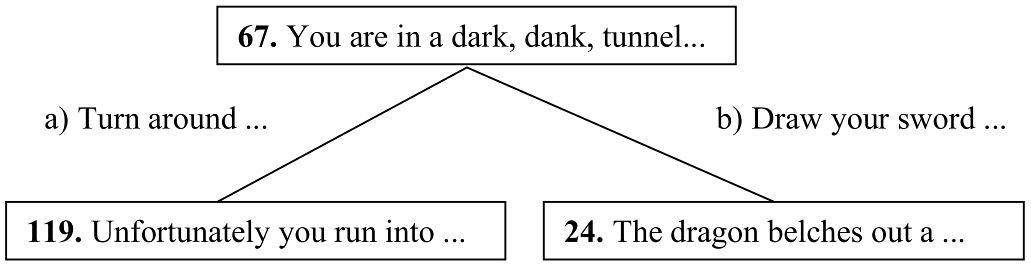
Each of the stations in between (not shown above) are also nodes in the tree. They are just nodes with a single edge in and a single edge out. A linked list is thus a really simple tree in which all nodes have only one link out.

A tree is a good organisation of the line, because it is intended to carry commuters into the city. On the whole people wish to travel between the leaves and the root, rather than between two leaves (Uxbridge and Amersham say). The trains will get fuller as they get closer to the City as more people are picked up at each station. Assuming all trains run between Aldgate and one of the other terminal stations, this also means trains in the busy centre (between Harrow-on-the-Hill and Aldgate) are very frequent where they are most needed, whereas trains in the suburbs are less frequent. Thus by organising the line in this way its efficiency is increased. However, if the situation changed and Uxbridge say turned into a city centre in its own right, this would be a poor organisation.

It is possible to buy adventure game books, where you play the main character and make decisions about what you do and so change the story. They consist of numbered paragraphs with questions at the end of each. Depending on what your answer is, you read a different numbered paragraph next. So, for example, a paragraph might be something like:

> ***67.*** *You are in a dark, dank tunnel. Icy water drips onto your arms and things scuttle over your feet in the dark. A faint light appears ahead. You round a corner to be faced by a fire-breathing dragon.*
>
> *Do you*
>
> *a) Turn around and run back down the corridor, screaming for help at the top of your voice.*
>
> *Go to Paragraph **119.***
>
> *b) Draw your sword and scream "Prepare to die".*
>
> *Go to Paragraph **24.***

Assuming it is not possible to get back to a paragraph you have already read, or end up at the same paragraph by two different routes, such a book is also a tree data structure. The very first paragraph, where the story starts is the root. Each of the questions represents an edge in the tree, and the nodes are the paragraphs themselves. The **leaves** of the tree are the final paragraphs (e.g. where you die). Reading the book once involves traversing the tree from the root, out to a **leaf**, and there are as many different story lines as there are leaves. The above paragraph is actually a tree fragment of the form:

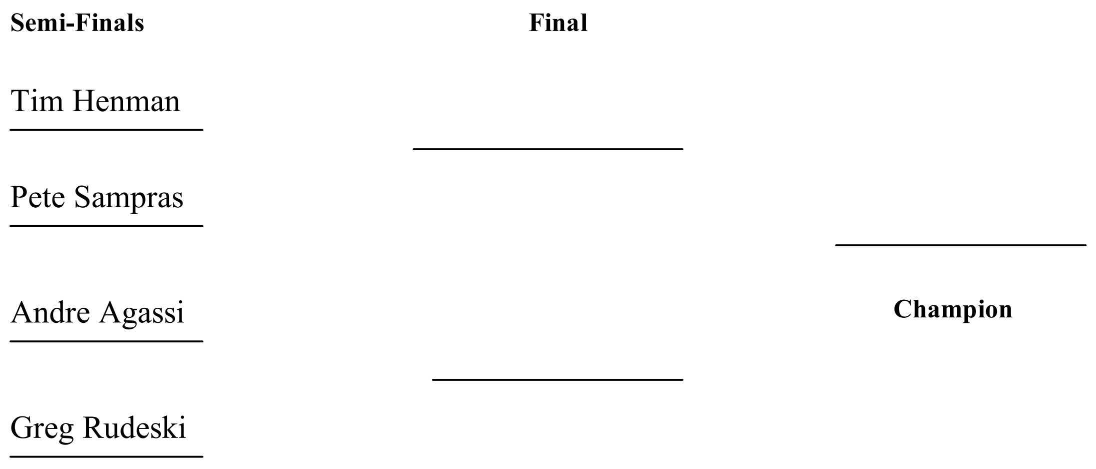

67. You are in a dark, dank, tunnel...
a) Turn around ...
b) Draw your sword ...
119. Unfortunately you run into ...
24. The dragon belches out a ...

In this kind of tree, we not only place information in the nodes, but also on the edges – the question you are answering. We similarly could have labelled the edges of the family tree with an indication of whether the parent was the mother or father.

A similar idea to the treasure hunt is sometimes used in "fun" orienteering events. Instead of being given a map with all the points on at the start, competitors are just given a map with the first point to visit on. At that site you pick up a map with the next point. Usually the map will have three points on it, and be accompanied by a multiple-choice question. Each answer corresponds to a circle drawn on the map. Only the circle corresponding to the correct answer will hold the next question. This is just a variation on the treasure hunt, but with the pointers given by circles on a map, and their being several possible destinations from each point. It is thus a tree with one long trunk running from start to finish, but with one or more additional single edge branches from each node.

Trees are also the normal way of organising people at work. Schools have one Headmaster, several Deputy Heads, a whole series of Year Heads, and then the masses of teachers. The Army has a similar tree-like hierarchy of Majors, Colonels, Sergeants and so on. Most firms have a single Managing Director, several Department Managers, and so on. One of the first pieces of paper I was given when I first started working at Middlesex contained a tree-like diagram showing me the hierarchy used there (with my position firmly at one of the leaves). Over the next couple of weeks I was given 3 more copies of this by different people – it was obviously considered very important that I understood the tree that I was now a part. Why are Organisations organised as trees? One reason is that it makes communication and so management easier. Everyone has a single Boss above them in the tree to whom they report. No one has the problem of having two people ordering them to do different things. That person also reports upwards so that information can be filtered so that the more important people at the top are not overwhelmed with trivial information. Similarly, information can be passed down efficiently either to everyone, or just to those parts of the Organisation that it concerns. Each manager also has a small manageable number of people that they are directly responsible for. Recently many companies have "de-layered". They have scrapped layers of the tree – the middle management. They have not scrapped the tree however. They have just made it shallower, so that information does not need to travel so far.

We saw earlier how an adjacency matrix is used to keep track of football results. It is a table indicating the scores between each pair of teams, or the date of the fixture if they have not played. Is this the structure chosen for all sporting competitions? It works best for leagues. Knockout competitions use a different structure for keeping track of fixtures: the tree. Every year as Wimbledon starts all the newspapers have pull-out specials, the centrepiece of which is a tree for you to fill in as the competition progresses. Each level of the tree is a round. The leaves are the starting competitors and the root will (eventually) name the winner. The same structure is used for the FA Cup, the World Snooker Championships and in fact every other knockout competition whether international or just an internal club competition. The tree structure works best because knockout competitions are in effect themselves trees. They are competitions the aim of which is to find the root of a tree.



Why is a tree structure the basis of so many competitions? It is because it is an efficient way of narrowing down a group to a single person by pair-wise games in that the number of actual games played is small. The tree structure ensures that the person who wins has proven themselves to be better than everyone else. Also the players in each round have played an equal number of games up to that point so there is fairness in the number of games played.

Trees are so useful that evolution has selected them in the construction of our bodies. For example, blood is carried through our bodies in arteries in a tree structure:

> "*One large artery, the aorta, leaves the heart and gives off smaller branch arteries to the various parts; these branch again and again to form finally very fine arteries known as arterioles*" (Armstrong, 1932)

Our veins, which carry the used blood back to the heart, use a similar structure. Our lungs also have a branching tree structure to get the maximum amount of oxygen into the blood. Why are trees so useful for this? The tree structure gives a way of covering a very large area. A tree with only a small number of branches has a very large number of leaves. It also reduces the distance to the leaf, so the blood or oxygen travels the smallest distance possible whilst reaching the maximum number of locations.

Graphs

The first railway lines constructed were single chains of towns and cities. To get between any two, you had to pass through all stations in between. As the network grew, this single chain could have been continued until all the towns and cities in the country formed a single chain, zig-zagging around the country. That is the equivalent of putting information into a list. It could alternatively have developed in to a tree like the Metropolitan line described earlier. This would be appropriate if most people wished to travel between their home city and London say (to some extent the British network was built like this by the London-centric planners). In general, however, large numbers of people wish to travel between other cities too. For example, the parents of a student at Manchester University, might live in Sheffield. It would be very inconvenient if they had to travel South towards London, changing in Peterborough, say, every time they wished to take some washing home, rather than by going directly over the Pennines. For reasons like this the rail network is not a tree but a **graph** data structure.

A tree connects nodes by edges so that there is a single root, and such that each node has a single edge in to it, though it may have any number leaving. With a tree, each node is on one and only one path from the root node. Graphs are like trees in that they connect nodes with edges. However, they are not restricted to the tree-like shape. Any node can be connected to any other node.

The edges of graphs can have directions associated with them, indicating which way along them you can travel. Consider a map showing the roads of a city. Most cities these days have one way systems, so if the roads were represented just by lines, it would not be much help in driving around. Instead a **directed graph** is used to represent the streets. You are told the direction along which you can travel for each edge. One way of doing this would be to use arrows to give the direction (with double-headed arrows to indicate a normal road). A-Z maps use a more subtle method but with the same intention of turning the graph into a directed rather than undirected one. Two-way streets have bold lines down both lines bounding the road. If it is a 1-way street, it has a bold line only down one side, with the side this is done indicating the direction of travel allowed.

Tree data structures are just graphs with the special "tree" shape. Suppose the author of the adventure game book liked happy endings. She had thought one up that she was especially proud of; so much so in fact that she wanted every one who played the book to have that ending. This was except of course for those who were such sad individuals as to have had been toasted by dragons, had their juices sucked out by a hoard of spiders or had their skin peeled off by a Pog Troggler. Thus all the stories in the book that did not end abruptly would have as the second last step: *Go to paragraph 42*, where paragraph 42 holds the happy ending:

> ***42:*** *You triumphantly arrive at the palace your quest completed. The hand of the Prince is yours. You march into the thrown room, to find it deserted. Blood is splashed around the walls. An uneasy feeling descends upon you. The whole palace seems devoid of life. Eventually you find a single serving boy, cowering in a flour bin in the pantry. George (as that is his name) explains that while you were away a whole army of Pog Trogglers had ravaged the city and slaughtered everyone. You thank your good luck that you were away on the*

quest at the time, as otherwise you too would have died a very unpleasant death. You ride off into the sunset with George and live happily ever after.
THE END

The book is no longer a tree as several branches converge on this node. However, it is still a graph. Similarly, if any of the choices take you back to a paragraph (node) you have visited before then whilst the book is still a graph, it is not a tree. Note that the book is also a directed graph. You are told how to get from one paragraph to the next, but once you get there, there are only instructions of how to continue on, not of how to get to the paragraph you came from. All the links have a direction through the story associated with them

More interesting variations on the simple list-based board games involve turning the list into a directed graph. Snakes and Ladders is an example. Instead of each square only being linked to the next, some squares contain the base of ladders, ie edges connecting squares to later squares. Other squares contain snakes – edges going back to earlier squares.

Random Access or Serial Access Data Structures

Lookup tables work on the principle that given some label or index you can go straight to an object. Music CDs have this property. I bought a CD "Californication" of the Red Hot Chili Peppers because of the brilliant track "Otherside". Once I know that "Otherside" is track 4, I can use that information to go straight to that track at any time. It takes no (noticable) difference in time between my making the selection and the track starting to play than if it had been track 15. I can go straight to the entry. That is a basic property of an array structure: arrays allow **random access** in the sense that you can get to any entry, given its position just as quickly as any other.

Contrast this with the situation had I bought a tape of "Californication" instead of a CD. Now to get to track 4 I would need to play or fast forward past the first three tracks keeping count in some way of which track I was up to. Accessing track 15 would now take much longer than accessing track 4. In fact the later the track appears on the album, the longer it would take to get to it. A tape allows only **serial access**: the tracks have to be accessed in turn starting from one end (or the current position). Linked list structures are like this, as you cannot go directly to any node but the next one.

Tapes are fine if we want to play an album from start to end (such as when in a car). CDs are much better if you frequently want to play single tracks. Indexes in books allow random access to an essentially serial structure (books are written to be read from start to finish). Text books usually have an index as random access is often needed: you dip into them looking for information on a specific thing. Novels never have an index as they are intended to be read from cover to cover. The last thing the author of a detective novel wants to give you is a way of looking up whodunnit before you get to the end.

Summary

By connecting elements by links rather than requiring them to be placed one after another, gives a data structure more flexibility.

A **linked list** is just a list where each element or **node** is connected by a link or **edge**. To move from one element to the next we follow the link.

Doubly-linked lists have links in both directions between connected nodes. This makes it possible to move from a node to both the next node in the list and to the previous node.

Circular linked lists have links between the first and last element of the list.

Lists are a single sequence of things. **Trees** have a branching structure, with each node linked to multiple next nodes. Each node has only one previous node, however. The first node in a tree is called the **root** of the tree.

Graphs are the most general linked node structures. Any node can be attached to any other. Linked lists of the various kinds and trees are just examples of graphs with restrictions on how the graph can be constructed. Graphs can be **directed** if the edges have directions associated or **undirected** if there are no directions.

Part 3: Algorithms

13. The Search is On (Search Algorithms)

Canst thou by searching find out God?
The Bible, Job Chapter 11, verse 7.

Linear Search
Take a pack of cards. Shuffle it thoroughly. Now find the Four of Spades. How did you do it? Put that card back and shuffle the pack again. Now find the Nine of Hearts. Did you look for the card the same way as the first time? Suppose the card you were looking for was missing. How would you have discovered this fact when searching? At what point in the search would you have known for certain that the card was not there?

If you used some methodical way of finding the card that you followed each time then you were using a search algorithm. Most people would search for a card in a shuffled pack by starting at one end of the pack, and checking the cards one by one until they came across the card they were looking for. This method of searching is well known in Computer Science. It is called **Linear Search**. It is one of the simplest ways of searching. You know when the thing you are looking for is not there when doing a linear search if you have reached the end and have not seen it. This is obviously so, because by that point you have inspected every card in the pack.

Suppose like me you have a pile of CDs next to your Hi-fi. Each time I play a CD I put the old one that was in the machine back on the top of the pile. This means the pile ends up in a totally random order. Suppose I want to find the CD *Regatta de Blanc* by the Police. How do I find it? I start at the top of the pile and check them in order until I get to it. If I got to the bottom of the pile I would know it was not there, and was probably in my CD walkman. I am again using the linear search algorithm.

Suppose you need a taxi from home to the airport. How would you find the phone number of one? Get a copy of the Yellow Pages and go to the Taxi section. Now pick a taxi firm. You could just take the first, but its probably best to get one that is based near to where you live so that it is more likely to be on time. Chances are you would start with the first one and check through them in turn until you came to one with an address near to you. The owners of taxi firms know this is what you are likely to do. They also know that the Yellow Pages puts firms in alphabetical order. That is why so many taxi firms are called things like AA Taxis. Guesthouses use the same trick. I have stayed in one called A&B Guest house, for example. The names of many businesses are thus the way they are because of the linear search algorithm!

On the corner of my desk I have a pile of papers that represent my "TO DO" list. New things to be done get added to the pile, which I gradually work through. Suppose my Boss came in asking me why I never returned the form he sent me that should have been filled in the day before. I would have to quickly find it. Unless I had some idea where it was in the pile, I would have to start at the top and work down the pile one at a time. Again I am using linear search.

The book, *The Diving-bell and the Butterfly*, (Bauby 1998) was written (or at least dictated) using linear search. It is the autobiography of Jean-Dominique Bauby, the former editor-in-chief of the French magazine Elle. He suffered a massive stroke and was left totally paralysed, unable to move or speak, other than being able to move a single eyelid. Rather than give in to his disability, he instead decided to write his autobiography. In the book he describes what life is like for someone who is totally paralysed, including how he communicated with people and so managed to write the book. He dictated it to a secretary, just using blinks. This is where linear search came in. The secretary read through the alphabet a letter at a time. When she read out the letter that came next in the word Bauby wanted, he would blink, and she would write it down. She would then start at the beginning again looking for the next letter. Letter by letter, the whole book was written in this way. Each letter was found by a linear search of the alphabet. The algorithm was improved slightly from a straight linear search of the alphabet in that the letters were ordered by frequency in French (Bauby's native language and that in which the book was written). E is the most common letter so it was first in the list, then S, A, R and so on finishing with W, the least commonly used letter. All Bauby's communication to the outside world was done in this way.

The game of Hangman is played using a variation of linear search. The problem is to work out a word one of the other players has thought of. You only know the number of letters in the word to start with. The search task is to search through the letters of the alphabet and work out which are in the word. You might start by guessing E as it is the most common letter in the English language. Next you try S, then perhaps A and R. With each letter tried, you are told either that it is not in the word (and lose a life) or the positions that word occurs. In the earlier stages of the game you have effectively lined up the letters of the alphabet in order: E,S,A,R... and are doing a linear search down them, until all the letters of the word have been found. Of course if you are playing well, you will not have the alphabet in a fixed order but will start to guess the word and change the order of the letters based on the results of the previous guesses.

The game of I-spy is also played by something similar to linear search. One person thinks of a word and tells the other players its first letter: “I spy with my little eye something beginning with T”. The players then list the things they can see that start with the letter T: “Toe”, “Toy”, “Television”, ... When a person names the thing the first person had thought of the search (and the game) stops. Otherwise the game continues until the players have tried every possibility they can think of. The search is being done in a linear fashion: one by one, and on each search question one possibility is ruled out. The difference here is that you do not start with a full list of the things to search. You may not ever come up with the answer if you do not think of it. You are performing a linear search of the things you can think of and see that start with the given letter.

Linear Search is so commonly used because it is simple and relatively quick provided the amount of things to sort through is small. It is a natural way to search when the things being searched through are in random order, and you do not expect to have to search for things very often. Does the fact that it is so common mean that it is the only way to search for things, or that it is always the best search algorithm? Things certainly do not always seem to go as well as they should. How often have you searched for something in one place after another, only to have the thing you were looking for in the very last place you could possibly have looked? That is one of the disadvantages of linear search. In its worst case you have to check everything.

Let us try and write out the algorithm for linear search. If in all the above situations we are following the same algorithm, then we ought to be able to write one set of instructions that works for all. Lets start with the problem of searching through a pile of books. We are doing something over and over again – so we have some form of repetition. We probably do not know how many books there are to search through so it is not a counter controlled loop. We stop if we have found the thing we are looking for or we have run out of books to check. In other words we keep going while the current book is not the one we are after and while there are still books to look at. Our algorithm is going to look something like:

```
while  you are not holding a book and
       your finger is not at the bottom of the pile
       do the following repeatedly
              ....
```

What do we do repeatedly? We check the current book and ask if it is ours or not. We are making a decision and will do different things depending on whether it is ours or not. That sounds like a 2-branched if statement.

```
while  you are not holding a book and
       your finger is not at the bottom of the pile
       do the following repeatedly
              if your finger is against the book you want
              then ...
              else ...
```

What do we do if the book is the one we want? We take it out of the pile – we have found it. What do we do if the book is not the one we want? We move on to the next one.

```
while  you are not holding a book and
       your finger is not at the bottom of the pile
       do the following repeatedly
              if your finger is against the book you want
              then take that book from the pile
              else move your finger to the next book.
```

So we check that we are not holding the book, and have not run out of books to check. If so we ask if the current book is the one we want. If it is we take it from the pile. If not we move to the next book (the current book is now the next book). We then go back to the loop question and see if we have finished yet.

There is one thing still missing: initialisation (remember I warned you about forgetting that!) What is our finger doing before we start? (it could be painful if it is picking our nose at the time when we start to follow the algorithm). We have not explicitly said where in the pile of books to start: we must start at the top as otherwise we will not check some of the books so fail to find the book we are looking for (so its important!). We also have not said whether or not we are holding a book at the start.

To find a book in a pile **do the following:**

1. Put down any books you happen to be holding.
2. Put your finger against the top book.

```
3. while you are not holding a book and your finger is not at the bottom of
   the pile do the following repeatedly
      if your finger is against the book you want
      then take that book from the pile
      else move your finger to the next book.
```

Try this algorithm out on a real pile of books to make sure it really does work – remembering to follow the instructions rather than doing what you think you have to do.

Problem 1

Write out a version of the linear search algorithm for the situation when you are searching for a given DVD in a pile.

Problem 2

That should have been easy as you just have to replace the word "book" for "DVD". Write out a version for a Nurse who Jean Dominique Bauby is trying to communicate a single letter to (assume normal alphabetic order is used).

That needs slightly more changes but the basic structure of the algorithm should be the same – the same loop with similar tests and a similar if then else statement inside.

```
To find a letter being thought of from the alphabet do the following:
   1. Take a blank piece of paper.
   2. Say the letter A.
   3. while you have not written a letter down and
            you are not at the end of the alphabet
            do the following repeatedly
               if the person blinked
               then write down the last letter you said
               else say the next letter of the alphabet.
```

The structure of this algorithm is identical to the version for books. Notice we have still accounted for the situation when you end up not writing a letter down – if you get to the end of the alphabet the search failed. Perhaps he was not trying to communicate anything to you! However I have been slightly lazy in that I have assumed that if the last letter said is Z then when given the instruction to say the next letter the person will say something sensible like "there's nothing left" or just say nothing. Strictly I ought to have spelled this out. In fact whatever is said then is a sentinel value – not a letter of the alphabet but something else that means the end. By "at the end of the alphabet" I mean the person has just said that sentinel value.

Problem 3

Modify the above algorithm to explicitly use the sentinel value "END".

Since it is a single algorithm, we should be able to write a general version of it that works whatever kind of thing we are searching through. We can then use it whenever we wish to do a linear search. We do this by taking the parts of the algorithm that are specific to a particular problem (like books or letters) and change them to something more general, but leaving the structure alone. For example we could replace the word "letter" by "thing". We also need to change various of the sentences to make sense when talking about general things. Here is one more general version of the algorithm.

```
To find a thing from a series of things do the following:
    1. Note that you have not found the thing yet.
    2. Set the current thing to be the first thing.
    3. while you have not found the thing you are searching for and
              you are not at the end of the series of things
              do the following repeatedly
                  if the current thing is the thing you are searching for
                  then you have found the thing you are looking for
                  else make the current thing the next thing in the sequence
```

We now have a general set of instructions (algorithm) called linear search that we can use to search for things. There are other ways of writing it that amount to the same thing and so are still linear search. However linear search is just one way of searching (and actually not that good a way in many situations). We will now look at some others.

Binary Search

There are actually many search algorithms, used for different purposes, and you probably use variations of several without even thinking about it. Have you ever played the game of 20 Questions? This involves one person thinking of a famous person. The other player has to work out who it is just by asking Yes and No questions. You try to do it in as few questions as possible. If you take more than 20 questions you lose. This is just a search problem – we are searching for the name of a person out of the millions of famous people there are in the world. If the game is just a search problem then we could play it by doing a linear search. How would this work? We would ask a series of questions of the form "Is it X?" as with I-spy. For example a typical game might go:

"Is it Nelson Mandella?"
- *"No"*

"Is it Arundhati Roy?"
- *"No"*

"Is it Freddy Mercury?"
- *"No"*

If you played this way, you would probably lose every time except for perhaps on a few very lucky occasions. These lucky occasions would also probably convince you that you could read minds! You would only win if the correct answer were one of the first 20 people you thought of. Given there are millions of people to choose from you do not have much hope (unless of course you *can* read minds!) So why do people ever play? And how come they often manage to get the right answer in much less that 20 questions? The problem with linear search is that you rule out only one answer (or thing in the pile you are looking through) at a time.

Suppose you were playing 20 questions with me. What would you ask first? A common first question is

"Is the person male?"
Other questions that you might ask early on are
" Is the person alive?" and " Is the person fictional?"
Why are these questions good ones to ask and in particular better than giving a series of names? They all have the obvious disadvantage that you have no chance of winning on that question itself. Are they then wasted questions?

The big advantage of such questions that makes it worthwhile "wasting" the chance of winning on that turn is that they rule out large numbers of people in one go. Most importantly they do this whatever the answer to the question. Questions such as "Was the person a member of the rock group Queen?" rule out a large number of people if the answer is YES. However, they only rule out the 4 members of Queen if the answer is NO. Since the latter is the most likely outcome until, perhaps, the later stages of the game, it is not a good question. Thus the ideal question is one that rules out half of all people if the answer is yes, and the other half if the answer is no. That means it does not matter to you what the answer is, you are equally close to the person's identity. The "Queen" question would be worthwhile if somehow you had narrowed down the search on the previous questions to being a member of either Oasis or Queen for example. The best of the above questions to ask first is thus the male/female one. The more questions that divide the remaining possibilities in half that you come up with, the quicker you will get to the answer. Halving the population repeatedly very quickly takes you to a single person. If there were a million possible candidates, it only takes 20 such questions to *guarantee* narrowing down the search to a single person: and that really is a guarantee! Compare that with linear search. If you are very, very lucky you might get the right answer the first time (I once won in 2 questions!). In the worst case, however, the correct person could be the last one you asked about, assuming you had the time to ask a million questions. The skill of the game is of course to come up with good questions that do keep splitting the field in half. If a question does not split the field in half every time it could take more questions, and you no longer have that guarantee.

Does this approach to searching only work for 20 questions or can it or variations work on other kinds of thing being searched through? Get a residential telephone directory and find a friend's telephone number in it. How did you do it? Did you use linear search – starting at the first page and checking every entry until you found your friend's? If you did, it probably took you a very long time (unless their name happens to be something like Aahann, Aammir or Aaronovitch – and it is very bad news if their name is Zwiebel, Zygovistinos or Zykun). It is more likely that you used a variation on the 20 questions algorithm (if a little haphazardly). If the name was Steinbeck, for example, you would not start at the first page, but open the directory somewhere in the middle. There is little point starting at the beginning, after all, as a name starting with S is more likely to be near the end. However, it is possible that you have over shot the page you wanted. You would therefore check what the names started with on the page you had opened the directory at. If you are at a page before the name you want, then you can rule out the first half of the directory and concentrate on the second half. This is possible because the telephone directory is sorted. If it were in a random order, you would be no nearer finding the right entry. If you have overshot, then you can rule out the second half and concentrate on the first half. Now you have only half as many entries to search, and can continue in the same way, go to a page roughly half way through the part you have not discarded and discard another half. Keep doing this until you get to the single entry that is your friend's name.

Let us suppose I opened the book at McDonald. It is before Steinbeck, so I ignore all pages before that point. I move to a page half way between McDonald and the end of the book. This time the first name on that page is Tambe, so I overshot. I now go half way between McDonald (assuming I remembered to keep my finger in that place) and Tambe finding Poulter: I overshot in the other direction this time. I go forwards again to a point halfway between Poulter and Tambe and find Shah. Not far enough, so I split between Shah and Tambe. Smith: not far enough. Split between Smith and Tambe: Stanton. I am now on the correct page (so could continue with the same process working through the page). With 7 questions I have narrowed down a whole telephone directory to a single page. A few more and I would be down to a single entry. With Linear Search I would only have made it as far as Aarrons in the same time!

This approach of searching things by dividing the field in two is called **binary search.** As with good 20-questions play, this search algorithm halves the number of entries to search on each "question". The great thing here, though, is you just keep asking the same question every round! The question in this case is "Is the entry in the place I opened the book earlier than the one I want". It is making use of the fact that the information to be searched has been pre-organised: it is sorted.

Let us return to searching a pack of cards. If you thought the pack was shuffled, you would probably start the search by linear search. However, if after checking a few cards they seemed to be in order, you would quickly abandon that approach and switch to something similar to binary search, jumping ahead and ruling out whole portions of the pack in one go.

The reason we can find entries in books such as dictionaries so quickly is because time was spent by the editors organising the entries. It is only because they are organised in a known (alphabetical) order that we can use variations on binary search. Sorting the entries will have been a great deal of work for the editors, but by doing that work once, the time taken by the many people subsequently using the dictionaries to search for things is much shortened. A similar approach is used in a variety of algorithms: you spend longer at the start organising the information once and for all, so as to save time doing something that will be repeated many times later. Provided the repeated part is done often enough this will save time overall. Today it seems inconceivable that a dictionary or telephone directory would be in anything but alphabetical order. However, one of the first ever dictionaries (in the 16th century), John Withals' *Shorte Dictionarie for Yonge Begynners*, was actually ordered by subject (Winchester, 1999), so the realisation that alphabetical order would be the most useful organisation was clearly not immediately obvious.

Jean-Dominique Bauby, the person who was totally paralysed but still managed to write a book, could have made his task much easier if he had known of binary search. Instead of having the person he was communicating with work through the alphabet a letter at a time, they could have started with M. Instead of a blink meaning "Yes", it could have meant "later in the alphabet", with no blink meaning "earlier in the alphabet". A double blink would mean "that is the letter". Each time the next letter read out would be roughly half way through the interval remaining. This would have taken the secretary longer to learn to do, perhaps, but it would have meant *every* letter of the alphabet could have been identified in only 5 blinks. With Bauby's algorithm, only the letters E,S,A,R and I could be identified that quickly. The least common letters would take up to 25 blinks. The book is139 pages long with around 180 words per page and perhaps 7 letters per word: something over 175 000 letters. Binary search would have saved an awful lot of blinks. That is just for the book. If all Bauby's communication had been with binary search, communication would have been significantly less frustrating both for him and for his friends and relatives. An understanding of algorithms can thus make real difference to a person's life.

The German developers of a brain scan based system developed for sufferers such as Bauby did know of Binary search. They developed an electroencephalogram (EEG) headset that can read the wearer's mind, at least as far as recognising when they think "yes". They needed a way of turning this small ability into a full communication system. Here is a description of what they did with the person's thoughts:

> *"If the subject's brain activity gave a "yes" signal, the group of letters was split, and then split again until only one letter was left – a letter that began to spell a word, and then a sentence."* (Radford, 1999)

Thus binary search has even be used to read minds.

It probably occurred to you that when you actually search for a name in a telephone directory, (or a word in a dictionary) you would probably do it slightly differently. If looking for Steinbeck, you would not open it in the middle as that would almost certainly take you to the middle letter – M – as it did in the example above. S is nearer the end of the alphabet so you might open the directory two-thirds of the way through. You would do this on each step: using your knowledge of the positions of letters in the alphabet, to make a better guess than halfway each time. This is an optimisation of binary search where you are using even more knowledge (the positions of letters in the alphabet) about the thing being sorted to speed things further.

Here is a new game I have just invented that is similar to 20-questions. It is called 10-questions. Just as in 20-questions, one person thinks of a famous person. The rules for questions are now slightly different. The questions no longer have to be ones with YES/NO answers but can have up to four alternatives (including as one of the alternatives, "none of those"). The person giving the answers must say which of the alternatives is the case. For example the person guessing might ask as their first question: "Is the person you are thinking of a) Asian b) African c) European d) none of those". The answer given might be "a) Asian". A question later in the game might be: "Is the person you are thinking of a) Nelson Mandella b) Archbishop Tutu c) neither of those. Notice that you do not have to always give 4 alternatives, the questions could have two three or four alternatives, as long as one of the alternatives is "none of those". However, you now only have 10 questions to get the answer. Play a game or two to get the idea. We decided that a perfect player of the original 20-questions would come up with questions that divided the possible answers in two. Why? Because that way, whatever the answer you always rule out the same number of people. Does the same thing apply to this game? Is the best strategy still to ask essentially yes/no questions? That is still allowed but is it still best? Perhaps one of the options should cover half the alternatives and the others can be anything?

Problem
Before reading on decide what you think the best questions to ask are.

Problem
Is it easier or harder to get the answer in 10 questions than in the original game with 20 questions to ask?

Suppose with earlier questions we have narrowed down the search so that we now know the answer is one of the four Beatles. We have several questions left but obviously want to be sure to get the answer as quickly as possible. Here are some questions we could ask.

> Is it a) John Lennon or b) one of the others?

or we could ask

> Is it a) John Lennon or Paul McCartney, or b) one of the others?

or we could ask

> Is it a) John Lennon or b) Paul McCartney or c) one of the others?

or we could ask

> Is it a) John Lennon or b) Paul McCartney or c) Ringo Star or d) the other one?

Which of these questions would you ask?

If you ask the first question then you would not even be playing good 20-questions! If you are right and it is John Lennon then fine. However if wrong you still have 3 choices left, which is not so good.

If you ask the second question then you are playing good 20-questions. That question splits the options in half, so whatever the answer there will be two choices left and one more question is guaranteed to get it. That sounds good, but we only have half as many questions over all so really we need to be able to do better than 20-questions.

If you ask the third question you are trying to make use of the extra flexibility you have but not fully – perhaps you thought John Lennon and Paul McCartney are the most famous so its worth naming them. However, if that hunch is wrong we still have two possibilities left and need a further question.

Of course the last question is the best question. Whatever the answer it rules out three people: three quarters of the alternatives. We are sure of the answer with only one question. Suppose the question before we knew the answer was either a member of the Beatles, a member of the Spice Girls, a member of Queen or a member of REM (all pop groups with 4 members). Obviously the question to ask is

> "Is the person in a) Queen, b) REM c) the Beetles or d) none of the above (ie the Spice Girls)?"

Why because whatever the answer we rule out three-quarters of the options. We will then be sure of getting the answer in one more question. The tactic to use is to try and come up with questions that *always* rule out roughly *three-quarters* of the population whatever the answer. That is each choice should cover an equal number of people. This is actually the same reasoning as with the original game. We only have 2 choices of answer so we want to be left with half of them with each question. Now we have 4 choices of answer so we aim to have only a quarter of the people left with each question. In both situations we want each possible answer to cover the same number of people as that is the only way we can be sure that whatever the answer we get the same outcome. Suppose we were allowed only questions with a choice of three possible answers, what fraction of people would we want each answer to cover?

So when playing the game of 10-questions, the best players will try to come up with questions where each option covers a quarter of the people left. This rules out more people with each question than in 20-questions so it must take fewer questions to get the answer. How fewer? Suppose we have asked questions in both games so that there are exactly 64 possible people left.

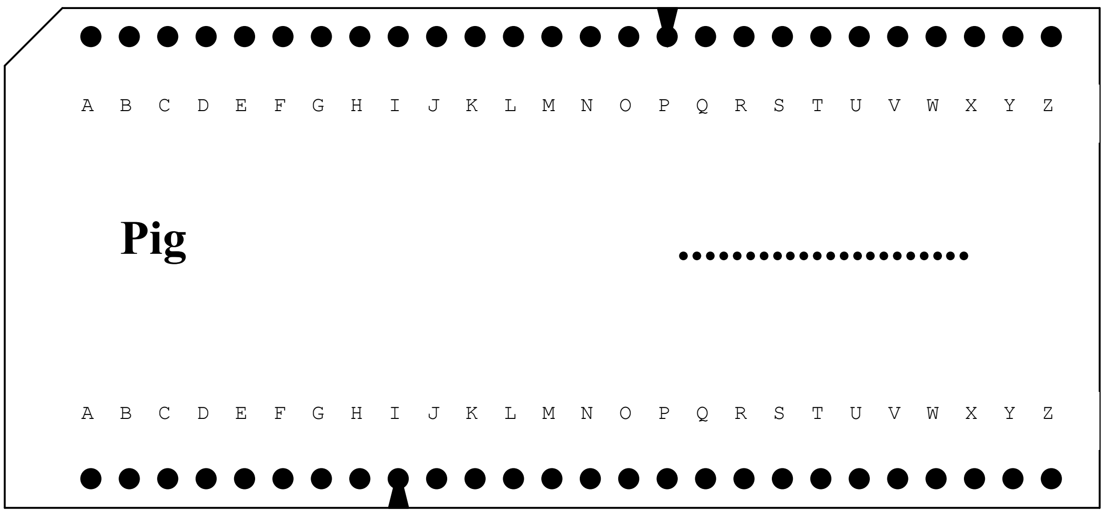
With the rules of 20-questions and each question ruling out half the alternatives. The first question leaves 32 people (dividing 64 in half), the second question leaves 16 people, the third question leaves 8 people, the fourth question leaves 4 people, the fifth question leaves 2 people and with the sixth question there is only one person left – we know who it is in 6 questions.

With the rules of 10-questions on the other hand and each question ruling out three quarters of the alternatives. The first question leaves 16 people (dividing 64 in 4), the second question leaves 4 people, and with the fourth question there is only one person left – we know who it is in only 3 questions. It is twice as fast. If you do the same reasoning starting with a million or so people you will find that 10 questions in the game of 10-questions is exactly as efficient (ruling out the same number of people) as 20 questions in the game of 20-questions. Just as 20-questions played well corresponds to binary search. The game of 10-questions corresponds to a faster algorithm of dividing into four. The more categories you can divide into in a single question the faster the algorithm will be. When we open a dictionary in roughly the correct place for the first letter of the word we are looking for, we are very roughly doing this. The "question" is which of the 26 letters of the alphabet does it start with. We then go directly to that section.

Could we use the observation that 10-questions is faster than 20-questions to help come up with a faster algorithm for someone with locked-in syndrome to use to communicate? This would require the paralysed person to be able to indicate one of four options. If they could only blink one eye then you would need some system of single or double blinks. If they could blink both eyes then there are four signals possible: no blinks, blink left, blink right, blink both at once. The question now asked would not be first or second half of the alphabet remaining, but, first quarter (no blinks, second quarter (left blink), third quarter (right blink) and fourth quarter (both blinks). The first question would narrow the search down to A-F, G-M, N-S, or T-Z. Each of these has 6 or 7 possibilities in. Suppose no blinks were communicated, this would mean A-F. The next question might be AB, CD, E or F. (As there are not 8 alternatives we do not have 2 choices in each category). That question might give us the answer, but if not (say left blink was communicated meaning C or D) a third question (C or D?) would give the answer. At worse we get the answer in 3 questions but at the expense of being able to blink with both eyes and it being harder to determine what the answer means.

Binary search works well for computers because the basic "questions" computers can ask are binary yes/no ones as we saw when discussing if-the-else statements. There are situations where we can overcome this restriction as we will see. Of course, if we would ask a question that can have any number of options as answer then it would tells us which of the alternatives the answer is directly in one question. As we will see that idea leads to a different algorithm (lookup-tables and bucket searching) that have other disadvantages.

When I was at School, our R.E. teacher arranged for a Missionary who had just returned home from Papua New Guinea to talk to us about his life there. The most interesting thing he described (to me at least) was the way he learnt the local language. He did not have an English-Papua New Guinean dictionary as none existed at the time so he had to write one himself as he went along. I kept a similar vocabulary book of new words when I was learning French at school. The problem with my vocabulary book, however, was that the words were not in alphabetical order but in the order I came across them. This was very infuriating, as often I knew I had come across a word, but just could not find it. The Missionary used a different storage mechanism (data structure) which allowed him to use a search algorithm similar to binary search even though the words were never sorted. Each time he came across a new word, he would write it and its translation on a card, so that he had one card for each word. These cards acted as his dictionary. He did not keep the cards in alphabetical order as in a normal dictionary, however. Instead each card had a series of 26 holes along each edge, with each hole labelled by a letter of the alphabet.

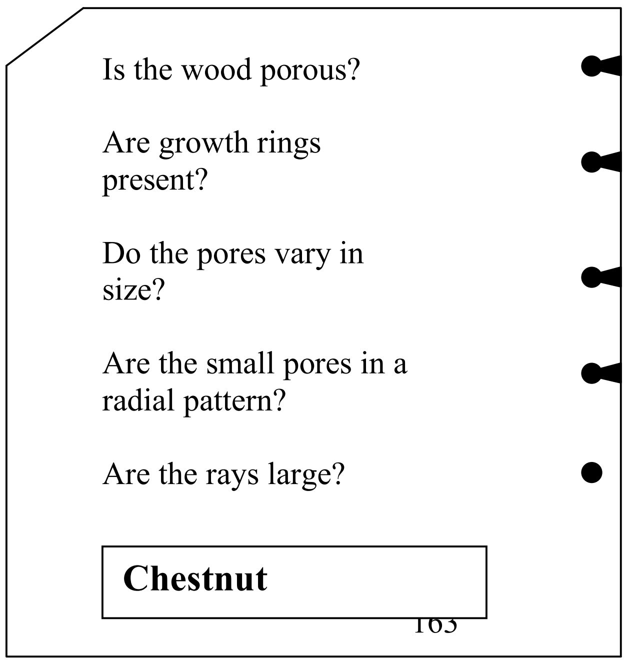

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
Pig
....
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

The topside of the card represented the first letter of the word. The bottom side of the card represented the second letter of the word on the card. Suppose the Missionary was told the Papua New Guinea word for Pig. He would cut a notch in the edge of the card into the hole labelled P. He would then cut a notch in the bottom edge against the letter I (as in the above diagram).

The word Pig and its translation would be written on the card and it would be put in the pile of other words. If at some later time, he needed to know the word for Pig, he would take the pile of cards and place a knitting needle through the hole labelled P on the top edge. He would then hold up the knitting needle and shake it. All the cards with a notch cut in the P (cards for words starting with P) would fall to the ground, leaving all the others on the knitting needle. They would be discarded, and the knitting needle would be put through the hole labelled I on the bottom edge of the cards that had fallen to the ground. After another shake only cards for words starting PI would fall down. Usually this left a small enough number of cards that the correct card could be found using linear search.

The cards are being used to do a variation on binary search in a similar way to 10-questions. The first "question" narrows down the search task to words starting with P. The second to those starting PI. By putting more holes round the cards, further letters of the words could have been labelled. The first question leaves you with a 26th of the original possible words, and the second a 26th of those (assuming their are roughly the same number of words starting with each letter).

The same search algorithm using cards can be used to help identify objects – in the same way as 20 questions works. In the 1950s, the Forestry Products Research Establishment used a similar card system to classify the thousands of different species of trees that exist. Given a series of known facts about a tree the cards were used to work out which species it was. Here, the holes on the cards represented YES/NO questions, such as "Are growth rings present?" and "Is the wood porous?" A hole with a notch represented "YES" and no notch represented "NO" as the answer to the question. The name of each tree species was written on a card, and notches cut in the holes for which the answer to those questions was YES for that tree. The same approach, inserting a knitting needle through the pack was used to answer the series of questions about the tree to be identified. If the answer to the question for the wood under scrutiny was YES, then the cards that dropped were kept. If the answer was known to be NO, then the cards on the knitting needle was kept. Eventually, only one card would remain and it would hold the name of the tree under scrutiny. Note that the order the questions are asked is important. This is because the answer to one question may lead to another being superfluous. For example, there is no point asking if the smell is leather-like if we already asked if the pores are in a radial pattern since there are no trees with these two properties. We will see later how our questions actually form a tree data structure.

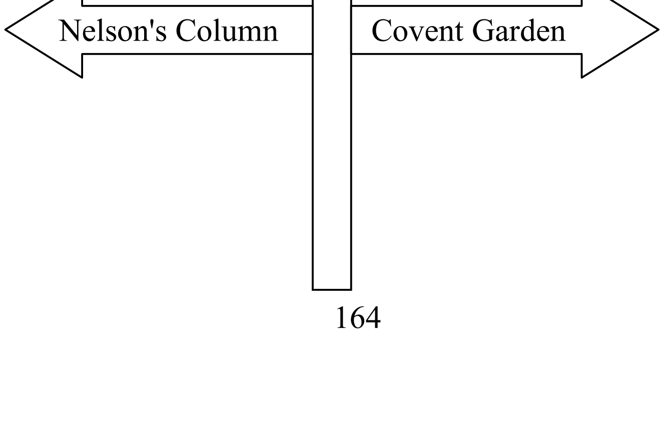

Is the wood porous?
Are growth rings present?
Do the pores vary in size?
Are the small pores in a radial pattern?
Are the rays large?
Chestnut

Tree Searching

Binary search is fast because it makes use of the existing organisation of the data: that the data is sorted. However, the information is still placed in a long list both for linear search and for binary search. We can organise data in other ways, however. A tree is a way the data can be organised to give a search algorithm similar in some ways to binary search.

Imagine you are a tourist on a day-trip to London. You arrive at Charing Cross Station and wish to visit Nelson's Column. As you have never visited London before, you do not know which way to go. However, a friend has told you it is in easy walking distance. You are faced with a search problem. How to find Nelson's column among the thousands of other places in London. You could buy a map, but they can be difficult to follow. You could ask for directions. However, perhaps you are a foreign tourist and do not speak English very well so you are not confident that you would understand the instructions. Perhaps like me you know that even if you did understand you would still get mixed up and go left-right-right instead of left-left-right. You could get a taxi, but that is expensive (and you are worried that you might be making a fool of yourself if it turns out that Nelson's Column is actually just round the corner!)

Luckily the London Authorities want you to find Nelson's Column too along with many of the other places you might have been wishing to visit such as Covent Garden, Leicester Square, the National Gallery, etc. They have therefore done some organising in advance. This is only worth their while because they know there are lots and lots of people attempting the same search problems every day – all those tourists. As you walk out of the station, and are wondering which direction to go in, you see a signpost. It does not give you all the directions, just the first stage: which street to go down immediately. It also gives directions to other places too, so you might find yourself with people going to Leicester Square, though the tourists wanting Covent Garden have gone the other way. As you arrive at the next junction your step starts to falter, as you realise you do not which way to go again. However, just as the doubt sets in, you see another signpost. The Leicester Square tourists now split off another way, and you continue. At each junction you are told the way to go next. You are never given the whole instructions, just the information needed immediately at each point.

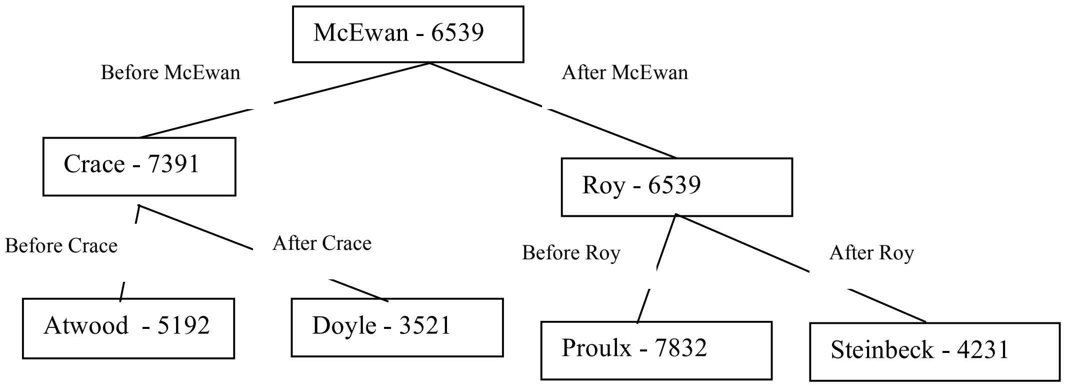


This differs to the approach used by a taxi driver. London cab drivers spend months learning the best ways from any place in London to any other. They spend their days before being licensed, riding round on mopeds learning the streets. Before they become a cab driver they must pass a test to prove they do know the way from anywhere to anywhere else. All the information of all routes is stored in one place: the cab drivers head. With the signposts, we have spread the information out amongst the street junctions. The workman who put up the signs was effectively building a tree of information. Each junction is a node in the tree, and each road an edge. It allows you to search for Nelson's Column using an algorithm called **tree searching**. Think of a (simplified) view of the streets from Charing Cross Station. Charing Cross is the root of a tree-like structure. There are two branches out of it – the street in each direction. Each road junction is a node in the tree: at each junction there is a split and two or more different ways to go.

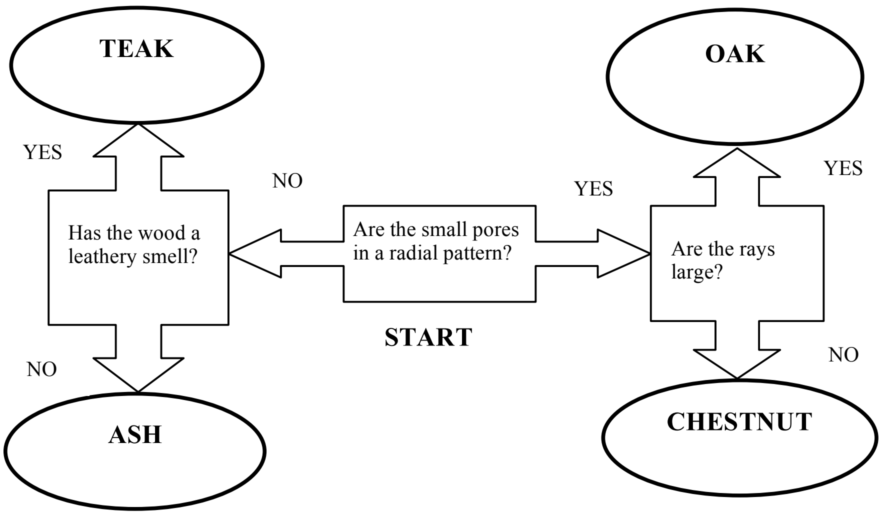
Tree searching involves building a structure like the tree of roads and junctions out of the things to be searched through, rather than just putting them in a long list. The data is placed in the nodes (junctions) along with signposts indicating what is down each path onwards out of the node. By starting at the root, the thing being searched for is found by following the signposts.

If there were lots of data, then this suggests that each branch would have lots and lots of signs – one to each destination. Imagine having a signpost outside Charing Cross Station that gave the direction to go in for every single tourist sight in the whole of London. It would be a gigantic signpost (and finding the direction to your destination would be a time-consuming search problem in its own right). To avoid this problem signposts give directions to classes of places. Leaving London on the M1 the signposts list the nearest cities. All the others are classed together as, for example, "The North". In the opposite direction, they are grouped as "The South".

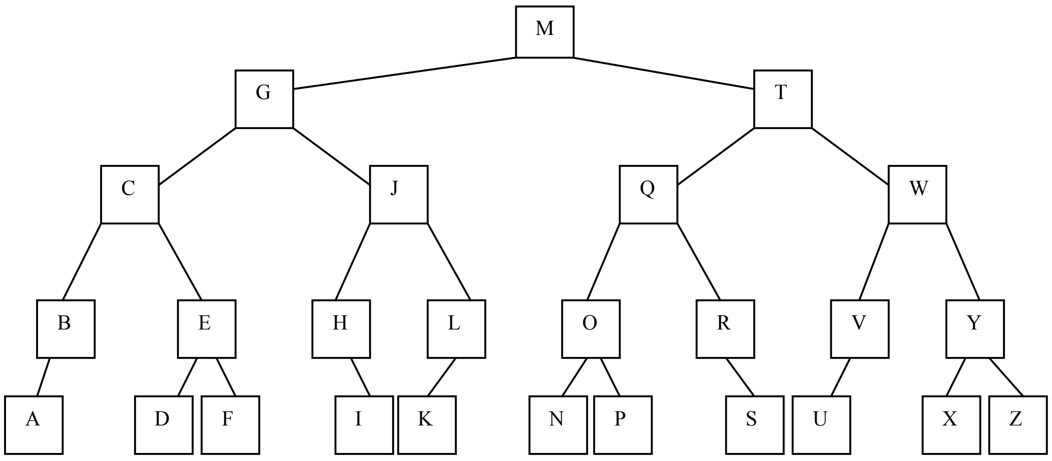
In a tree built for tree searching, we do the same trick. Suppose we are searching for names in a tree-based directory. We start at the root node (which might hold the telephone number of McEwan say) and look at the signpost. It would not list all the names in the directory, but would be arranged so that all the names before McEwan in a dictionary order lie down the left branch, and all those above McEwan lie down the right branch. Unlike with tourist attractions, we can put things where we like in our tree. Organising things in this way is similar to the way a normal telephone directory is sorted: names are not just entered in the order that people applied for telephones. Thus we are using two "classes" of names. Those alphabetically later than the name in our node and those alphabetically earlier. Our signpost in the first node just needs two signs "Less than McEwan" and "Greater than McEwan". Each node is organised similarly. It holds a name and the corresponding phone number. One branch leads to all the names that come earlier alphabetically than the name in this node. The other leads to all the names that come later. You compare the name you are looking for with a node, and if its not yours move on to the node indicated by the signpost. In this way, you quickly find the phone number you are after. Why is it quick? Just like in Binary Search, we discard half of the data with every decision. In a tree search we discard the whole sub-tree down the branch we ignore. Once we decide that the name we want is after McEwan we have discarded in the following example, Crace, Atwood and Doyle in one go.



Search trees are also used for classifying objects as an alternative to the card system we saw earlier. For example, the diagram below is adapted from one in an encyclopedia about timber (Bramwell, 1976) for identifying wood. We show only the branch for porous trees with growth rings. Even though at first sight this diagram does not look very tree shaped, it is a tree in our sense. It has a root node (labelled "start") and each node has two branches out each leading to either a leaf or another node. By adding more questions at the start (root) node, we would be able to classify a wider variety of wood. We start at the start node and ask the question there. The answer sends us one way or the other down the search tree, ruling out half the possibilities as we go. Eventually we get to a leaf of the search tree where we are given the name of the wood we are trying to identify.

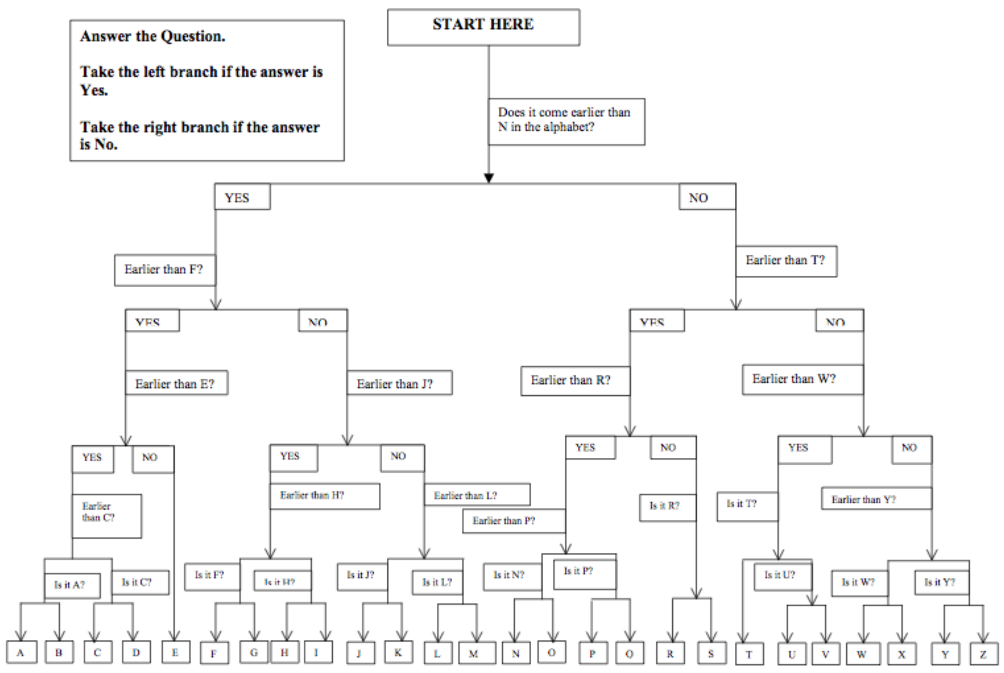

TEAK
OAK
YES
NO
YES
YES
Has the wood a leathery smell?
Are the small pores in a radial pattern?
Are the rays large?
START
NO
NO
ASH
CHESTNUT

Jean-Dominique Bauby could have made use of tree searching to aid his communication with visitors. We saw earlier how binary search would have made it quicker for him to communicate. The problem with this is that without practice, working out the next letter to ask, halfway through the interval left, is hard. Bauby had a similar problem with his method, since few people can recite the alphabet by letter frequency: E, S, A. R, etc. The problem was solved by giving visitors a list with the correct order on. We would similarly solve our problem by giving visitors a tree. Instead of working along a list, they follow the branches of the tree. Bauby's blinks now indicate whether to go down the left branch or the right branch.



This tree could be optimised for letter frequency in the same way as Bauby did with his list. As it stands, infrequent letters like W and Q are at the top of the tree, and common ones like E, S, A and R are at the bottom so take more blinks to get to. By rearranging the order we could have common letters at the top. "Earlier" and "later" in the alphabet would now be different so Bauby would need to learn the tree himself so he could blink appropriately, rather than just knowing the normal alphabetic order.

In the above style of tree searching, the things we are looking for are stored all the way through the tree – in its nodes. At each stage we therefore have a choice of three possibilities: “it is here”, “it is down the left branch”, and “it is down the right branch". The advantage of this is that it means some things are found immediately (the letter M in the above tree for example). However it comes at a cost: you need to ask 3-way questions with 3 possible answers instead of binary ones. Bauby would therefore need 3 signals such as no blink, 1 blink and 2 blinks to indicate which way to go. You therefore lose some of the advantage of reducing the questions asked by needing more blinks per question. An alternative is to just put binary YES/NO questions in the nodes and put the things being searched for at the ends of the branches: in the trees leaves, as in the search tree for identifying wood given above (see below). Now a blink means go one way and no blink the other. However you have to descend to the leaves of the tree whatever the letter. With an alphabet of 32 letters you would always need 5 questions as the tree would be 5 layers deep. With our alphabet of only 26 letters, some of the branches can be shorter than this, however. This again gives us an opportunity to optimise the questions asked so that the common letters such as E can be found more quickly.



Search trees are, in one sense, the data structure equivalent to if-then-else statements. If you wish to make a decision based on information such as that for identifying timber, you could write an algorithm as a series of if-statements. However, such an algorithm can only make decisions about a fixed set of things such as trees. To make decisions about other kinds of object you would need to write a new set of instructions – a new algorithm. Alternatively, you could put the same information in a search tree data structure. Now the general tree searching instructions allow you to make decisions. To make decisions about a different kind of object you use the same algorithm but applied to a different search tree – containing different data. You do not need to learn and follow a new set of instructions, just follow the old instructions but on a different diagram.

Bucket Search

If we have plenty of space and are going to do lots of searching for the same things, a very fast way of searching is to do a **bucket search**. This involves pre-computing the answer to the question "is it there?" for each thing we might search for. We store the answers in an array with one entry for each thing we might search for (the array is effectively a **look-up table** containing answers to the search question). The array subscripts are thus labels naming each thing we might search for. In the simplest case the array entries are just YES or NO, indicating whether the thing was found or not. This is exactly what we were suggesting when taking binary search to the extreme and saying we want a way so that in one question we can be told the answer to our search question.

This is one of the most common ways that people organise appointments (meetings, parties, etc) they must keep. One way to do this would be to keep a list. Every time you found out about some event, you would add it to the list, noting the date. Finding if you have an appointment on some date would take a lot of time. You might try and keep this list sorted into order, but you would have to keep writing it out over and over as new appointments were added. Not surprisingly most people use a completely different way of organising appointments. We buy and use diaries and calanders. These are booklets with one entry for each day of the year (office diaries even have an entry marked out for every hour of the year). That is it has one entry for every possible day you could have an appointment, whether or not you do have appointments on that day. Initially all entries are empty, but as you are invited to parties or whatever, you fill in the appropriate entry. To discover if you have a party on a particular day, you simply go straight to that entry. It takes no time at all, because you do not need to scan through all the entries. Now a diary can actually be very wasteful – perhaps you are different but I do not have parties (or even other appointments) every day. Many of the entries in my diary remain blank. In a sense a diary is therefore a waste of paper. Putting appointments in one long list as I found out about them as suggested originally would use far less paper. This is the downside to being able to check for appointments quickly. The fact that diaries are so popular suggests that the trade-off is considered worthwhile in this case.

Consider the following memory game. A pack of cards are shuffled and you are given12 cards. You have 30 seconds to memorise those cards during which time you must place them face down (anywhere) on the table. You are then asked to turn up a given card, or state that it is not on the table. If you turn up the wrong card, or incorrectly say it is not on the table, you lose. One way to do well at this game, would be not to spend your 30 seconds memorising the cards, but instead spend the time organising them. Mentally divide the table into a series of card size slots with one row for each suit, Kings at one end, Aces at the other. Put the cards down on the table in place corresponding to their slot. You have just created a bucket array. To find a given card, you just look at its slot. If the slot is empty, the card is not on the table. If the slot is full, that is the card you are searching for. The skill of the game is then in visualising the array of cards rather than in remembering things. Organisation beats rote-learning any day!

When I was at University, my Hall of Residence used something similar so that people could tell whether I was in or out. At the entrance was a board with everyone's name on, with an IN/OUT sign partially covered by a slider. When I was out I would leave OUT visible by my name, and when I returned I would slide it across to leave IN visible. This is acting like a bucket array, in that visitors did not need to do a full search – i.e. go all the way to my room to find whether I was in or out – the answer is pre-computed on the board. However, some extra work is required beforehand in that someone had to set up the board, and presumably put everyone's names to OUT before we arrived at the start of term. Also each time I went in or out, I had to do some work moving the slider.

Hash Tables

The trick to making bucket search fast is having a quick way of getting to the correct bucket. If you cannot go straight to the correct bucket, you have just replaced one search problem by another. If the buckets are numbered and in numerical order (ie the subscripts are numbers) it is relatively easy to go straight to the correct one. If the buckets are labelled some other way (for example by names) then you need a way of turning that label into a position quickly. Otherwise how do you find the right bucket! That is effectively what you are doing when you do a search – finding the correct position. The examples of lookup tables we saw earlier have this problem. When searching for a symbol in a map legend, we more or less have to search through each key of the table. We cannot work out from the symbol exactly where its position in the table will be.

Hash functions give a way of doing this. When a hash function is used in a lookup table, the table is known as a **hash table** rather than just a lookup table to show it is a special sort of lookup table. In the first instance the hash function is used to determine where to store things in the lookup table when constructing it (the pre-processing phase). You must put things in a place where you will then be able to find them. If you have read the Harry Potter books (if not why not?), then you have come across something very much like a hash function: the *sorting hat* (Rowling, 1997). Every year when the new batch of first years arrive at Hogwarts School of Witchcraft and Wizardry, the first thing that happens to them is that they are sorted into houses. Every pupil is in one of the houses: Gryffindor, Hufflepuff, Ravenclaw and Slytherin. Each house has its own tower with dormatories containing a bed for each pupil, so by organising the pupils into houses means (amongst other things) that they can be found when needed (eg to punish them). The four houses are thus like a lookup table with 4 entries – one for each house. Look in the Gryffindor tower and you will find all the students that were put there. The ceremony that allocates them is a pre-processing process. The pupils are processed one at a time. Each goes forward and puts on the "Sorting Hat". It is a magical hat that given a pupil can tell which House they should go in. It is like a **hash function**: a function that given a pupil works out where they should go. The pupil then goes to the house that the hash function (the "sorting hat") says they belong. The four house towers together are the **hash table**. As there are lots of students but only four houses, there are obviously lots of students in each house (unlike a straight lookup table where each entry has one thing in it). Thus having found the right house, a little more searching is needed some other way to find the right pupil. Normally hash functions are used in situations where there are more places to put things than things to put – eg more houses than there are pupils so that each pupil currently at the school gets their own house. For the "Sorting Hat" to be a proper hash function is would need to also be able to answer questions about its decisions. If one of the teachers (Professor McGonagall, say) asked it which house a pupil was in (say Harry) then it ought to be able to tell them. (I will not spoil the book by saying where Harry Potter ended up, in case you somehow inexplicably have not read the book yet.) That way the Professors could use the hat to quickly find any pupil it had originally "sorted". We will see later that this kind of "sorting" is slightly different to what computer scientists usually call sorting (putting things into order, rather than putting them in the right place as here) – so do not get confused.

Thinking of hash tables as being like lookup tables where a number is needed top work out where to go. That is as though each Hogwarts house had a number (eg Gryffindor is House 1, and so on. Then the Sorting Hat would just give a number for each pupil. Knowing the number you would know which house to go to.

The sorting hat uses magic to determine where to look. Computers cannot do that. They do things by *calculation* instead. We are free to order the objects in a lookup table in any way we like. If they are placed appropriately we can often *calculate* the position to look at from the label. Address books do something similar. Addresses are put into slots in the book calculated from the first letter of the name. You then go straight to the correct page, assuming you know the order of the alphabet. Bookshops use a similar idea to help you find books. Instead of ordering books completely alphabetically, they are organised into shelves labelled by subject: Computing, Politics, Mathematics, Literature, etc. To find a particular book you first have to work out its subject, then go straight to that shelf. Map makers similarly try to help by grouping similar symbols together in the Legend, for example, however it can still take several seconds to find the correct symbol.

With both address books, bookshops and maps, the calculation does not take us to a unique book, just one of many that we must then sort through some other way. If you had a very large number of pages in your address book, it could be organised in a different way so that you could go straight to the correct place without needing much if any further searching. You would just need to do a slightly harder calculation to get there. Since there are in most search problems a massive number of possibilities, having a lookup table with one entry for each possibility is impractical. Imagine having an address book, with one pre-written entry for each possible name any human was called, with telephone numbers filled in only for the people you know. It would be enormous in size, so is just not done. We could come up with a compromise that was better than just labelling sections by the first letter.

For example, suppose you knew the positions of every letter in the alphabet off by heart (A=0,B=1, ... Z=25). The normal method used in address books effectively involves turning the first letter into a page number in this way. You could alternatively turn the first two letters into a position and have one page per pair. Assuming dictionary order is still used with BA following AZ, for example, the page would be calculated by turning each letter into its position in the alphabet as above then multiplying the first by 26 and adding the second. Thus, for example, AZ is converted to (**0** × 26) + **25** = 25 and BA is converted to to (**1** × 26) + **0** = 26. The address of someone called *Azar* would therefore be placed on page 25. How do you find the same person's address later? You just calculate the number 25 in the same way and go straight to that page. As long as two people you know do not have the same first two letters in their name, you will get a unique address. A function for calculating a position in this way is called a **hash function**, and a lookup table where the information is accessed via a hash function is called a **hash table**. When two entries end up in the same place it is called a **hash collision**.

The above method uses up quite a lot of space. Many pages are likely to be blank as there will be many pairs of letters that are not the start of the name of anyone you know. Hash tables make a trade-off between search time and space wasted. At one extreme everything goes into one bucket (page), so you only need as much space as you have entries. However, you have to search through them all, which is slow. At the other extreme there is one page for each name that exists in the world. Then you can go immediately to the correct entry, but you are wasting a vast amount of unused space.

Using two letters is a compromise - there probably are many names that start the same way, so there will still have collisions where you have several names on one page, but hopefully you will only have a few entries to search. Since many members of my family have the name Curzon, however, the CU page in my address book would still be full. The calculation we used makes a poor hash function for names. We could avoid the problem by using a different hash function. For example, using initials together with the 3rd letter of the surname might be better. A good hash function is one that spreads the data evenly throughout the table, and so avoids clashes.

A hash table is thus a lookup table, where a calculation must be performed to find the correct entry, and where entries are shared by several different things, on the assumption that most of the time only one will be present. When a collision occurs, linear search is used to find the correct entry between those that have collided.

Summary
Searching is a commonly performed operation. There are many different algorithms for searching with different properties.

Linear search is the simplest search algorithm. It involves checking the elements being searched one at a time in order. It is, in general, a slow way of searching.

Binary search is a faster method of searching that uses the fact that the things being searched are sorted. We first check the central element, and decide whether the thing being searched is in the top or bottom half, then just search those elements in the same way.

Tree Searching first involves placing the things to be searched into a tree structure. A signpost is placed at each node indicating which part of the tree the things to be searched will be found – larger things are down one sub-tree, smaller ones down the other. The things being searched could be placed in all the nodes or just in the leaves of the tree.

Bucket Searching involves allocating a fixed position or "bucket" for each thing to be searched for, for example in a lookup table. The things are placed in their appropriate positions. Searching then involves going directly to the correct bucket. If the thing is there it is found, if not then it is not possessed.

Hash tables are lookup tables used for a form of bucket search where a calculation (given by a **hash function**) is performed to determine the place to look in the table. A hash table does not have one entry for each thing as would a full lookup table. Instead different elements make share the same location. This saves space as the table is smaller, but means **hash collisions** can occur, and so takes longer to find the element being searched for when this happens.

14. All Sorts of Everything (Sort Algorithms)

All things began in order, so shall they end and so shall they begin again.
Sir Thomas Browne, *The Garden of Cyr*us (1658) Ch. 5

We often need to sort things into some given order, and there are very many ways of doing it. The solitaire card game Patience is a sorting game. The rules are a partial algorithm for sorting cards. They are step by step instructions that if followed lead to the cards being sorted into order within suits. However, the rules of Solitaire are not all we might desire of an algorithm. In many rounds of Patience you will end up being stuck, with no rule giving you instructions of how to continue (You just lost). The rules are not **complete**. You can also have a game that never finishes in that if you follow the rules blindly you will never stop – though eventually you will get bored and give up. This happens when none of the cards in the discard pile can be placed. If you strictly follow the rules you will just keep checking the same sequence of cards over and over again, none of them with a place to go. The rules do not always **terminate**. The rules are not **finite**. Worse, the rules do not specify exactly what to do in each case – you have choices where two or more moves are possible – make the right one and you win ending up with a sorted pack, make the wrong one and the pack remains unsorted. The rules are **non-deterministic**. The rules do not tell you which to do because that is what makes it an interesting game. However, if the aim is not to have fun but to actually sort things, we need a proper algorithm that guarantees sorting every time.

Patience is also obviously quite a slow way to sort; ideally we want a fast algorithm. Sorting is relatively simple if there are only a few things to sort, however, as the number of items increases, the time taken increases disproportionately. By changing the sort method, big time savings can be made.

Bucket Sort
One of the simplest ways to sort things is to do a **bucket sort**. Each item to be sorted is allocated a bucket, and when you get to that item you toss it in the bucket. Once all the items are in a bucket, the buckets form a sorted list. My father sorts the large and random collection of nails, screws, washers, etc that he has acquired over the years in this way in his garage.

Imagine you have a shuffled pack of cards that you wish to sort. If you have enough space on a table or on the floor, you could just start laying out the cards in its appropriate position on the table. Mentally allocate separate rows for each suit, with the Ace to go at one end and the King at the other. Then just turn the cards up on at a time and slot it in its correct position. Once all the cards are laid down, scoop up the rows, put them back together and you have a sorted pack. You have just done a bucket sort. This is a very quick way of sorting as each card is looked at only once. It takes only as long to do as it takes to work through the pack once.

The post office has a similar sorting problem sorting letters. Thousands of letters arrive at the sorting office (a whole building dedicated to sorting!) as they are collected from post offices. They have to be sorted into piles corresponding to the areas they are to be delivered to, so they can be put in appropriate vans. Airports have a similar problem sorting checked-in suitcases onto trolleys to be put onto the correct plane. When things go wrong, you end up in Florida, with your luggage in Cairo. What went wrong? Someone tossed your case into the wrong bucket!

This approach was also used during the original compilation of the Oxford English Dictionary in the 1800s. This was the first dictionary to aim to have entries for *every* word to have appeared in print in the English language. It was a monumental task that took 75 years to complete. The editors made use of thousands of volunteers who read books, making lists of all the words within them. The words found were then to be sent to the editors, who wrote definitions and compiled them. One of the volunteers stood out to the editors as being amazingly useful. His name was Dr William Minor and he was a convicted murderer who at the time was confined in Broadmoor Asylum for the Criminally Insane (Winchester, 1999). His being as mad as a Hatter did not stop him devising the most efficient way of working on the Dictionary that surpassed the approach taken by every other volunteer. His secret was to create an index of words (we will look at indexes later) and use bucket sorting to create the lists. The way Minor worked was to carefully read a book looking for interesting words. When he found one he would write it in his notebook. However, he did not just write it anywhere, but in a very precise position. As Winchester describes (Winchester, 1999, page 123), suppose he came across the word buffoon, he wrote it in small neat letters in a precise position towards the bottom of the page. The position was chosen based on the number of words he expected to encounter that should appear before or after it. When he later came across another word it would be written in a position estimated to leave enough room for the number of words that might come between them. In other words, he had mentally divided his notebook into a series of buckets, one for each word he would encounter. By the time he got to the end of the book he was reading, he had a sorted list of the interesting words in it.

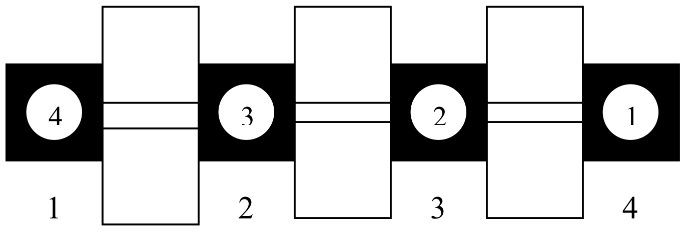
Straight Selection Sort
As it is so quick, bucket sorting seems an ideal way to sort things until you consider the space problem. You need a separate bucket for each (class of) item you are sorting. With lots of things to sort this can be a problem. Suppose you were in the back of a car, squashed in the back seat with your brother and sister on a long journey and decided to sort that pack of cards. How would you do it now? You do not have the space anymore to do a full bucket sort.

A simple approach known as **Straight Selection Sort** would be to scan through the pack until you found the Ace of Spades and put that to the front. Next you would look for the Two, and so on. Instead of just working through the pack once, you would have to scan through it 51 times since on each pass one more card is put in the correct position (Note it is not 52 passes since it is impossible to have all but one card in the wrong position). You would not need to look at the ones that were already in position each time, so every pass would be quicker than the pass before. Still, sorting just became very slow in comparison to the bucket sort, though you needed no extra space beyond the pack itself to do the sort

Bubblesort

The above way of sorting can get tricky with only two hands, trying to move a card from the back to its correct position without scattering half the pack over the floor. The **Bubblesort** algorithm gets over this problem, by requiring that cards only ever be swapped with their neighbour. Work through the pack a card at a time, checking it is in the correct position relative to the next card. If the next card is the biggest leave them alone, otherwise swap them over. Move on to the next card and do the same again. Repeat this until you get to the end of the pack, then go to the front and start again. Do this over and over until the pack is sorted.

On each pass you will have noticed that the big cards bubble down the pack until they meet something bigger that then takes over. Once you get to the biggest card on the first pass (the King of Diamonds, assuming you were putting them in Hearts, Spades, Clubs, Diamonds order), it stays with you until the bottom of the pack. So after 1 pass, 1 card (the King Of Diamonds) is in the correct position. After 2 passes, the Queen of Diamonds will have done the same trick. Again each pass puts one more card in the correct place so by the end of 51 passes they will all be in the correct position. It thus takes a similar amount of time to the earlier algorithm, but with less likelihood of the cards spraying over the driver.
The following sorting puzzle can be solved using bubble sort. Four pieces numbered from one to four are placed in reverse order on the black squares of the following board as shown (so that piece 1 is placed on square 4, piece 2 is placed on square 2 and so on).



A move consists of moving a piece from one square to an adjacent square of the opposite colour. The aim is to finish with each piece on its own numbered black square in 24 moves with the following restriction:

- each piece can only stay on a white square for one move of another piece before moving on.

The white squares act as swapping points for adjacent pieces. The layout of the board and rules of the puzzle mean that the restrictions on swapping pieces are those used by bubble sort. These same restrictions mean that the other sorting algorithms described here cannot be used. The choice of algorithm is often restricted by the conditions under which it must operate in this way.

Binary Bucket Sort

You could still get part of the way bucket sorting if you had a little extra space. This is easier to describe if you have 32 cards rather than 52, so consider a pack with only the 1-8 cards present. Instead of having a bucket for each card, allocate two – red on one knee, black on the other. One pass through the pack has now sorted the pack into two reds versus blacks (so this is going to be much slower than a full bucket sort).

Collecting the cards back up into a single pack, now allocate the knees differently: one Spades, the other Clubs. Repeat the process as you work through the black cards. When the blacks are done, collect them together in a single pile placed at the back of the pack in your hand. Next, reallocate your knees with one knee Hearts, one Diamonds. When you gather the piles up, you must do so in the correct order.

The cards are nearer to being sorted than they were, as now all the cards of each suit are together after only two passes. What we have done is similar to binary search, we have thought of a question each time that divides the part of the pack we are looking at in two, then used that question to split them. We continue in the same way. We now have the pack in 4 suits of 8 cards (remember we are sorting a reduced pack to make it easier to describe). We need a question to split those groups of 8 in half. We ask whether the card is greater than 4, with one knee for those greater and one for those less. After each group of 8 has been split in two, put the two piles to the back of the pack in your hand (in the correct order). After we have gone through the whole pack once more, it will consist of a series of groups of 4. The cards in the 4 are still in the wrong order, but the group as a whole is in the correct position with respect to the other groups. Next do the same with each group of 4, this time asking whether the card is greater than the mid-card of the group (either the 2 or the 6 depending on the group). At the end of this you will have a series of groups of two cards, with those pairs possibly swapped, but the pairs themselves in the correct relative position. One more pass working on the groups of two and the pack is sorted.

As mentioned the approach used was very similar to binary search. We repeatedly broke the problem in half, giving a similar but simpler problem. Algorithms that work in this way are called **Divide-and-Conquer** Algorithms. Because of the divide and conquer approach we only need 5 passes through the pack instead of 31. If we had been sorting a full pack instead of a reduced pack, the complication would just have been that some of the groups would have been different sizes. It is easiest to do when the number of things being sorted is a power of two, because of the way we halve the pack each time.

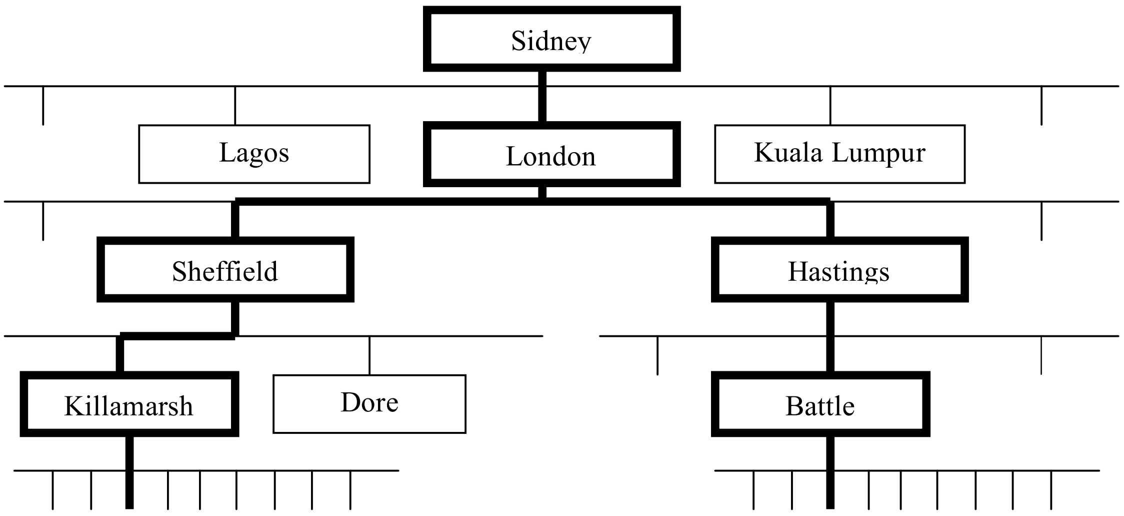
Radix Sort

The above **binary bucket sort** is more or less what I do after every exam I am the examiner for. When the exam scripts come back from the exam hall, my first job as an examiner is to sort them into student number order. Since there can be up to 400 scripts, a full bucket sort is out of the question – I would need a sports hall floor to spread the scripts out onto. It would also take far too long to use a naive sort. After all I have only a week or so to both sort and mark them – the more time spent sorting the less time I would have to mark them. Instead I use the digits in the student numbers to form the buckets. The first 2 digits correspond to the year: in the 98/99 year the first set of piles correspond to 98, 97 and 96-and-earlier. Then each of those piles is split using the next digit, normally requiring 10 piles. These piles are then divided based on the next digit, one at a time and so on. Eventually the whole pile is sorted, using no more than 10 piles, plus a few others for the piles waiting to be split. I thus need nothing more than a large table to do the sort. Using successive digits as the buckets in this kind of repeated bucket sorting is known as **Radix sorting**.

Tree Sorts
As we saw in the last chapter, tree searching is similar to binary searching. A tree structure is introduced and the questions asked during binary search are associated with a tree node. We can similarly use a tree structure to do a variation of bucket sorting: **tree sorting**. Each question that is asked is associated with a node of a tree, and instead of adding the things being sorted to piles, they are sent down one of the edges of the tree, towards the leaves.

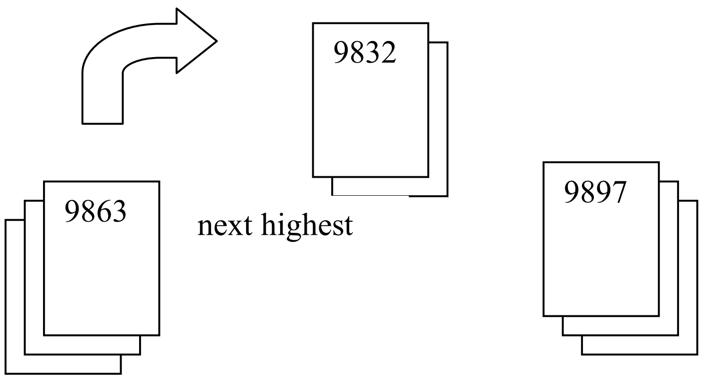
This is more or less what the Post office is doing when they deliver letters. Sorting offices are joined together in a tree structures. The sorting offices are nodes, the links provided by planes, trains and vans. They do not sort all the letters in a single sorting office, but ask a single question: Which country should this letter go to? Letters are then sent along the appropriate link (i.e. put on the appropriate plane) to get to the main sorting office in that country. At that sorting office they ask: What city should this letter go to, and they are sent on their way by train to the next node in the tree. At the city sorting office, they split the letters again by postal district and put them in vans that take them to the local sorting offices, where the postman picks them up to deliver them to the leaves of the tree: our homes.

Suppose, when on holiday in Sydney, I post two postcards one to *Killamarsh*, a village near Sheffield, the other to *Battle*, a village near Hastings. They will travel together on a plane to the London sorting office, then be sent down different branches, one to Sheffield, the other to Hastings. From there they will travel to local post offices, to be delivered to the appropriate homes.



There are several different tree-sorting algorithms. In the post office one above, the things being sorted are placed at the leaves of an existing tree. In the variations, the things being sorted are stored in the nodes, and the tree is built as the items are put into it. This is similar to the post office having a strategy of only building a sorting office in a place the first time someone sends a letter to that city! In these alternative algorithms, sorting is done by putting things into the tree one by one according to some rule, then taking them out in an appropriate way again one by one, so that they come out in sorted order. We will not go into details here

Merge sorts
When sorting exam scripts I usually have someone to help. Ideally we wish to divide the task up, so that we can both work independently and are never idle. **Merge sorting** is one way to do this. We just split the unsorted pile in half, and sort them completely independently, to give two sorted piles. We could even sort our own piles using different algorithms if we wished. Once we both have a fully sorted pile. We need a way of putting them back together. If we just put one pile on top of the other, we would not be much better off. Instead we **merge** the two piles into a third. The merge algorithm is as follows. We each look at the script on top of our pile. Whichever is largest is placed first on the new pile. We repeat this again only looking at the top two scripts until one of the piles has gone. The remainder of the other pile is then added one script at a time onto the new pile. This works because, since the two piles are sorted, the next one must be on top of one of the two piles, so we only ever need look at the top scripts.



In the above we only did a single merge. If we had 4 people to do the sort, we could split the pile in 4 and each sort one pile. We could then merge each pair of piles together to give 2 sorted piles. They could then be merged as above. With enough helpers, each person could be given a pair of scripts to put in the right order. Then the piles are gradually merged together, pairs of piles at a time.

This idea of repeatedly splitting and merging can be extended to the whole pile to give a sort algorithm called **mergesort**. Lets go back to sorting the packs of cards in the car. Suppose there is even less space. This time you only have one free knee (your baby sister is asleep on the other knee). Go through the pack two cards at a time, merging them into order in their pairs, using your free knee to create the new pile. Once you have gone through the whole pack, start again, taking adjacent pairs (one pair in each hand) and merge the pairs into sorted groups of 4 cards, putting them onto a pile on your knee. Keep doing this for turning groups of 4 sorted cards into groups of 8 and then groups of 16 until the whole pack is sorted. The whole sort was achieved by doing nothing but merging! You just used another divide-and-conquer algorithm. Think of what you did backwards. The last thing was to merge two halves of the pack. How were these sorted? – by splitting them into two piles, sorting and merging them back into order. You were repeatedly breaking the pack in two. You used a different divide-and-conquer algorithm though as you did different things with the two halves, and put them back together in a different way.

Multiple views and Indexes

Lists are often ordered in a specific way to make searching or other processing easy: alphabetically by name as in a telephone directory, by record sales as in the Top 40 singles list, by order of arrival as in the queue at a supermarket. This is fine if the lists are small or if that is the only order we wish to access them. Often, however, we really want to access the same list in different ways at different times. Suppose the phone rings, but the person hangs up before you get to it. In Britain you can ring 1471 to find the number of the person who just called. However, if you do not recognise the number, it might just have been a telesales company and the last thing you want to do is phone them back. I have also had people phone me back late at night after I dialled a wrong number. The resulting conversation was very surreal. The phone book is in alphabetical order because it is intended that you look up the numbers of known people. It would be nice in our situation above if the phone book were also in number order. You could then easily find the name of the person who called. As it is finding the name is near impossible! In fact BT rely on this as they are not allowed to give out information that would allow a name to be derived from a number to protect the privacy of the customers.

In other situations, it is even more desirable to get at the elements of a list in different orders. Railway stations use multiple views of timetable information. The full timetable is usually given in order of destination, since that is the information that you usually know about the train you wish to catch. On the screens as you enter the station, timetable information is given in order of departure time however, since people are likely to arrive shortly before their train departs. Such a list is therefore the quickest way to find the data they want.

In the above case the data is copied out again in full. This does not need to be so, however. In other situations, only one full copy of the data is kept to save space. **Indexes** are used for this purpose. An index is just an alternative ordering of the same information (or at least part of it) created by giving pointers into the original. A book, for example, is just a list of words in a particular (literary) order. However, in textbooks, it is useful to be able to access the text in other orders: looking up a given word. For example, in a book on Data Structures, you may wish to read about trees. Rather than read the whole book, you just wish to read that section. The index provides you with a quick way of finding it. It orders the main words of the book in a different order to the original (alphabetically). The index gives references to the places where each word occurs in the normal order.

A *concordance* is a whole book that is just an index of another book – usually the Bible. If you want to know about all the passages in the Bible where the word *plague* appears, just look in a concordance and it will tell you. The academics working on the Dead Sea Scrolls (ancient fragments of biblical texts that were painstakingly pieced back together) did not wish to give out copies of the works, as they wanted exclusive rights to study them. As a compromise, however, they did release a detailed concordance. However, since a concordance is just a list of the same words in a different order with references to the correct order, some other academics took the concordance and reconstructed the original order of words from it. They were happy to release the texts to the world and did so!

Travel agents have a similar queuing problem to Banks. However, they do not always make their customers stand in a queue. The people seem to be sat in a completely random order. An index is being used to maintain two orders – the order of arrival and the order of seating. Each person takes a ticket with his or her arrival position on. This means they can sit in any free chair and do not need to keep moving. The ticket numbers are providing an index. Cheese counters in large supermarkets often use a similar system as it is too difficult to make people stand in an orderly queue in this situation – they want to be able to wander up and down the counter deciding what to by whilst waiting.

When I get my hair cut a similar problem occurs. There are four seats for those waiting and people sit in them at random as they come in. When the next person's turn comes we do not all move up a seat but instead stay put. Each person knows how many people were in front of them when they arrived and remember this number. Each of the other people are similarly remembering their positions. When the next person's turn arises they know because the are thinking "I'm now first". As they move to the Barber's chair all the others mentally subtract one from their number. In effect there are two arrays. There is an array of people that stays unchanged other than when a person arrives or leaves. There is also a second array of the numbers (that bears no relation to the chair positions) in the heads of the people in the queue. This is in essence an index array. As with other index arrays it is being used to avoid moving something big around: the people. Instead the easily exchanged numbers are juggled in the peoples heads leading to less effort for all.

You do not have to restrict yourself to one index. Libraries keep several different indexes to their collections of books. The University library I used stored books on the shelves in the order that they were acquired. This was sensible as space was short and manpower limited, so books were just added to the end of the next free shelf, labelled with an acquisition number, so they could be returned to the correct position. However, as a student I usually knew the author of the book I wanted, not its acquisition number, so finding it by browsing the stacks would be near impossible. The library provided a filing drawer by author – a set of index cards in alphabetical order by author giving acquisition numbers of each book. At other times I knew the title but not the author. The librarian was well prepared, however, as a second index was in order of title. And just in case a student knew neither an author nor a title, there was a third index in order of subject. Nowadays, the index cards have been replaced by computers, but those computers have to do the same thing – keep lots of index lists.

The word lists that the insane dictionary writer William Minor created were also indexes. It was this that made him the most useful volunteer to the editors. One of the special things about the Oxford English Dictionary is that every word has quotations from the literature to illustrate its meanings. The editors had asked the contributors, on finding a word, to write it on a slip of paper, together with a full reference of the source and a quotation of the sentence in which the word was found. These slips were then to be sent in to the editors to compile. This led to the editors having a paper mountain of thousands of word slips to sort many of which duplicated ones they already had, and many that would not be needed for years given how slow the progress of compiling every word in the language was. Minor worked differently. For the first few years of work he sent in no words at all. He just compiled his lists of words. However, each time he wrote a word in his notebook, he also noted the page number it was found on. In other words, he created an index for every book he read. He then told the editors of the dictionary to let him know the word whose entry they were currently compiling whenever they were short of a quotation. He would then look up the word in his indexes, and if he had ever encountered it, which was most of the time, he sent in a quotation by return. Because of his carefully compiled indexes, he could provide words and quotations exactly when the editors required them. He also only needed to copy out quotations and full details of sources of words that were actually to be used in the Dictionary, unlike the other contributors much of whose efforts would ultimately never be used. Insane or not, Minor had a firm grasp of the power of choosing appropriate data structures and algorithms for a task.

Summary
Sorting is another common operation we perform. As with searching there are many different sort algorithms.

Bucket sorting involves having a fixed position for each element to be sorted. The things to be sorted are processed in turn, placing each in its correct position. The things are then just collected in order of position – sorted. This can be very fast, but usually impractical due to the amount of space required.

There are various ways of sorting naively – ie using very slow algorithms. One involves scanning through the things in a series of passes from start to finish, each time looking for the next thing in order and putting it in its correct position.

Bubble sort is similar to the above naive sort. However, on each pass adjacent things are swapped if out of order relative to each other, so that many of the elements move towards the correct position on each pass. On each pass the next biggest thing automatically will end up in its correct position, so does not need to be examined on subsequent passes.

Binary Bucket sorting involves partitioning the objects into two buckets so that the two halves are in the correct order relative to each other. Those halves are recursively sorted in the same way, until the halves consist of single things that require no sorting. **Quicksort** is another algorithm that works on a similar principle, but uses a slightly different partitioning algorithm.

Radix Sorting is similar to binary bucket sorting except that 10 buckets are used in each round corresponding to the digits in the numbers being sorted. In each round the sort is performed on the next digit.

Tree sorting algorithms do the sorting by placing the elements to be sorted into a tree structure as used for searching. By removing the elements from the tree in an appropriate way, they come out sorted. There are many variations.

Merge sorting is similar to binary bucket sorting in that it uses a divide-and-conquer approach. The things to be sorted are split in to and sorted separately (by recursively doing the same thing). Once the two halves have been sorted, they are recombined by merging – using the principle that the next thing in the sorted sequence will be the next in one or other of the two halves.

Often we wish to have a set of things in many different orders at the same time. An **index** is used for this purpose. By having multiple indexes we can have multiple orders without changing the actual order of the things of interest.

15. The Path of Righteousness (Path Algorithms)

Thou shalt find it after many days
The Bible, Ecclesiastes Chapter 11, Verse 3.

As we saw previously, graphs occur in a wide range of situations. What kind of computation might we then want to do on them? One of the most common problems that arises is that of finding a path between two points. Usually we wish to do something slightly harder than this – find the shortest path.

Finding a path

The most common situation where we must find a path through a graph is when driving from one place to another. Dirk Gently, the hero of The Long Dark Tea-Time of the Soul uses a "Zen" method of finding his way to a destination when driving "*which was to find any car that looked as though it knew where it was going and follow it*"(Adams, 1989, page 29). This, however, is not much good as an algorithm since "*the results were more often surprising than successful*".

Path finding is considered such a difficult task that it is used as a test of intelligence. One of the most lasting stereotypes of a scientist is of a person in a white coat timing a rat or mouse finding its way round a maze in search of food. The aim of such tests is usually to test the rodent's memory skills, intelligence or ability to learn (though if you believe *The Hitch Hiker's Guide to the Galaxy*, it was the mice who are conducting subtle tests on humans rather than the other way round (Adams, 1979, page 119). A maze is just an undirected graph, and the problem facing the rodent is to find a path through the graph from start to finish. The same problem is used in children's quiz books, where the puzzle is to draw a line through a maze. Mazes made from hedges are often found in the gardens of stately homes such as at Hampton Court and are major tourist attractions.

Path finding is obviously fun when set as a maze, but is there any systematic way of quickly finding a path, or is the only possibility to wander randomly? Since with a maze there is no way in advance to know which paths are the best to try, we thus really have to solve the more general problem of coming up with a systematic way of visiting every node in a graph. If we do this, then we just stop when we get to the destination node (i.e. the centre of the maze). One systematic way would be to retrace your steps back to the start and begin again completely every time you get to a dead end. This would be incredibly slow however. Worse still, unless you used the Hansel and Gretal approach of leaving a trail of stones, you would probably have just set yourself a new path finding problem of finding the exit again, each time you got to a dead end.

Obviously, the first thing to do is remember junctions you have already been at – possibly by marking them in some way. Once you can recognise junctions you have previously been to, you must recognise exits you have already explored – there is no point going down a path you have already been down. Instead you choose a different path. If you have already explored all the exits at a junction, then you go back the way you came, retracing your steps until you get to the previous node, where you again either take a new exit or retrace your steps once more. This algorithm corresponds to the well known trick for exploring a maze – put your hand on one wall and just keep going – as you go into a dead end, you automatically retrace your steps as you come back along the hedge down the other side of the path. As you return to a junction, by following the edge of the hedge you go down the next exit.

Exploring Trees
If we know something of the structure of the graph we are dealing with, we can use it to tailor the algorithm. For example, consider the problem of determining if someone is a direct ancestor of you. My father has collected a large amount of family tree details of a wide range of people in the village I grew up in, and it turns out that most of the old families are interconnected. Suppose his aim when he started was to determine if some particularly interesting person from the past history of the village is one of his ancestors. How does he answer this question? The ancestors of everyone in the village are connected in a graph. He is the start node. He must determine some way of exploring his ancestry in a way that ensures he does not miss anyone. He could therefore just follow the general maze-exploring algorithm. However, as we noted before the part of the graph he is interested in exploring – the part containing his ancestors – is actually a binary tree with him at the root. We can therefore simplify the problem to that of searching a tree. There are two main strategies that could be used called **Depth-first search** and **Breadth-first search**. In essence these are just giving different orderings of the way we back up and choose the next exit to explore in the general graph exploring algorithm.

One way he could work would be to follow one lineage back, checking his father, then his grandfather, then his great grandfather, etc. Only when getting to the furthest point possible down that lineage does he back up to the previous level to follow the other line from this point. This is called depth-first search as the depths of the tree are searched first. This algorithm has the disadvantage that if the tree is infinite you could never back up. If my father followed this algorithm strictly he would have to follow the chain all the way back to Adam or Eve (or perhaps Neanderthal Man) before starting on a second branch – not a sensible way to conduct the search. Instead depth-first searches are often cut short. In our case, if my father knows the birth date of the person of interest, he could stop the search of a branch, once arriving at a person who died before that date.

Chess players follow a similar strategy, here the tree consists of the possible positions that could result from a given move. Each move from a position gives another branch. A player explores each move by attempting to explore the tree, following sequences of moves. Better players are capable of descending further into the trees of weaker players – i.e. looking more moves ahead.

My father could alternatively work back through his family tree a generation at a time, checking that none of the current layer of the tree is the person of interest before moving to the next layer. Before moving to the great-grandparents, he would check all the grandparents. If he did this he would be using breadth-first search. This is a safe algorithm in that you are guaranteed to get the answer eventually. However, it can be slower. Whatever level the person is to be found at, using this algorithm, you will have fully checked all nodes at earlier levels of the tree first. With a depth-first search, you could hit lucky, and find the node you were searching for in the first branch you looked at. Depth-first search thus has a better best case, but worse worst case.

Finding the Shortest Path

If path finding is a good test of intelligence, then it is neither the Dolphin's nor the mice that are the most intelligent creatures on Earth as *The Hitchhiker's Guide to the Galaxy* might suggest (Adams, 1979). The most remarkable path finding creatures are Ant colonies. Once a source of food has been found they very quickly start to follow the shortest path round obstacles between the nest and the food source. Even more remarkably, their algorithm can cope with new obstacles being placed in their path. They soon find the shortest way round it, including it into their route. Their algorithm is thus adaptive.

How do they do it? In fact it is actually based on the Zen approach used by Dirk Gently, which turns out not to be so silly if you are an ant. They basically use the Hansel and Gretal method of dropping a trail – though in their case they leave a scent trail, rather than white pebbles. For finding the shortest path, they rely on the fact that subsequent ants will follow the trail left and reinforce it with more scent.

Richard Feynman, who won a Nobel prize for Physics, but who had a passion for understanding things whatever the subject, described how he worked out for himself how ants find shortest paths one year when his bath was invaded by a colony (Feynman, 1992). He put down piles of sugar for them, then once an ant had found a pile, he used a coloured pencil to careful marks its trail. As other ants found the sugar he followed their trail too but with different colours. Eventually, he had a multicoloured bath but also the answer. All the subsequent ants follow the trail of the first ant to find the sugar. It wiggles its way back to the nest and its route is not remotely the shortest path. As others travel along it however, in their rush, they often overshoot some of the wiggles, going in more of a straight line, though eventually making it back to the trail. Thus gradually the trail is straightened and also shortened.

Consider the situation where a short path has been found, but then a new obstacle blocks it. The trail disappears, so new ants arriving decide at random to go one way or the other. Roughly half can be expected to go each way. However, those choosing the shortest route will get back to the original trail more quickly. In any given period of time, more ants will get back to the trail via the shorter of the routes, thus that route will have a stronger scent. Subsequent ants coming the other way will therefore be more likely to choose it. When ants are creating the original path a similar thing happens. Once later ants have followed a slightly different path, at the branches the shorter path will gradually get more scent.

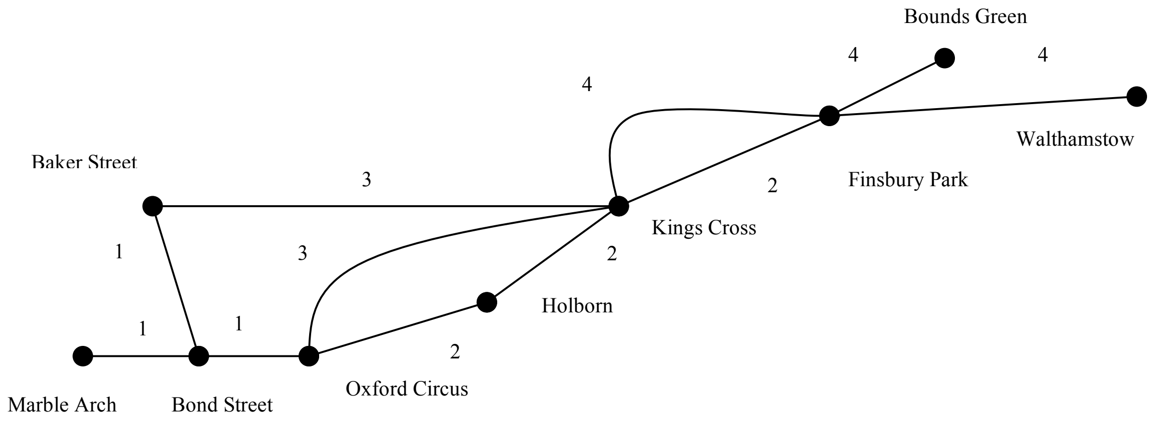
The ant's approach has been suggested as a method of finding shortest paths by computer by generating a colony of artificial ants. British Telecom has used this method to solve networking problems. However, in most human situations it is not so useful (though something very similar seems to happen every morning in Finsbury Park Station in the rush hour, as trails of people moving in opposite directions each try to optimise their route in or out of the station). It does however hint of an algorithm that we do use.

The most straightforward algorithm is to follow each path through the graph in turn from start to finish and make a note of the length of each. We keep a note of the shortest found so far. Once all paths have been checked, the shortest found so far is *the* shortest path. Note that unlike when just trying to determine if a path exists we do apparently have to check every possible path. This is more or less what the ants do when they come to a new obstacle. They are using parallel processing however – making use of the fact that there are many hundreds of them all working on the problem at the same time.

Can we do better than this? Well we can start by using the trick from maze searching. Rather than going back to the start each time, we just back track to the previous junction. We can make a note of the shortest distance found so far to every node at the node. If we get to a node, and the sum of the distance to the previous node and of the last edge we travelled along is shorter than the current distance then we have just found a shorter path to the node we are currently at.

Combining the above idea of keeping track of the shortest route to each node so far with the idea of always choosing the nearest node to the start next, we obtain an algorithm known as Dijkstra's Algorithm after its inventor.

Suppose after a hard day at work I decide to go to see a film at the Odeon Marble Arch followed by Pizza at Ask. I must solve the problem of finding the shortest route from Bounds Green to Marble Arch using the London Underground. If we count distance as number of stations travelled, then we are dealing with a labelled graph of the following form:



In planning my route, I start at Bounds Green and put it in a list. I look at the possible legs I could initially follow and pick the shortest. There is only one route (at least on the simplified version we are considering), so I add it to a list. I now know the shortest route from Bounds Green to Finsbury Park and that it is 4 long. I now look at all possible routes from either of the stations in my list. There are three, both from Finsbury Park, two to Kings Cross and one to Walthamstow, one of length 2, the others of length 4. Since both are via Finsbury Park, they are of distance 6 and 8 from Bounds Green. I ignore the longest ones and irrespective of the destination pick the nearest one via the shortest route. I thus add Kings Cross to my list. I now know the shortest distance to it from Bounds Green – via Finsbury Park on the Victoria Line with length 6. My list of known shortest distances from Bounds Green looks as follows:

Bounds Green	0
Finsbury Park	4
Kings Cross	6

I now look for the station with the shortest route out of any of the three stations in my list from Bounds Green to it, and then on to the new station. There are still the two of distance 8 that I originally ignored, but now there are three more, two of distance 3 and one of distance 2 from Kings Cross, so of distances 8 and 9 from Bounds Green. We add the two stations of distance 8 from Bounds Green.

Bounds Green	0
Finsbury Park	4
Kings Cross	6
Walthamstow	8
Holborn	8

Walthamstow adds nothing new, but at Holborn we have a new shortest link of length 2 to Oxford Circus. However, we can get to Oxford Circus directly from Kings Cross on a link we previously ignored. Now it makes Oxford Circus the next nearest station to Bounds Green at distance 6+3, i.e. 9 stops. Baker Street is also connected to Kings Cross and of distance 9 from Bounds Green, so we add both.

Bounds Green	0
Finsbury Park	4
Kings Cross	6
Walthamstow	8
Holborn	8
Oxford Circus	9
Baker Street	9

Bond Street can be added next and then Marble Arch at distances 10 and 11 from Bounds Green. Our final list of distances to stations is thus:

Bounds Green	0
Finsbury Park	4
Kings Cross	6
Walthamstow	8
Holborn	8
Oxford Circus	9
Baker Street	9
Bond Street	10
Marble Arch	11

We have thus processed the stations in order of shortest distance from our source station. We have then used the fact that we already know the distances to the nearest ones to work out the next nearest. If we already have calculated a distance to a station, we ignore any other links to it. In the above we just recorded the distances. We could similarly note the actual routes to each station, using the routes already in the list to give us the shortest route to the next one added.

When we are doing route planning on a map, we do not look at all possible routes. We quickly discard many alternatives because they are going in the wrong direction, or are too long. Similarly the ants quickly discard routes that are going in completely the wrong direction. At each junction the shortest path over the next leg is taken.

16. (Greedy Algorithms)

A simple and common way to solve problems and that forms the basis of many algorithms, is the idea of Greed! Sometimes this works really well. At other times it is a really bad thing to do. Consider the survival strategy of the Crocodile. Most animals are fairly picky about their food, only eating their proper food. The Koala bear takes this to the extreme by only eating Eucalyptus leaves. The crocodile does the opposite. It lies waiting in the mud and if anything that moves gets near its mouth – Galumph! it eats it. The idea is that you solve a problem at each decision point by doing the thing that immediately takes you closer to your goal. In the case of the crocodile, rather than wasting time working out whether something really is worthwhile food, just eat it and worry later. How does this apply to other problems? Consider the problem of packing a series of books into a packing crate (perhaps because yo are moving). You want to get as many books into the box, but they are all different sizes. Putting the wrong ones in first may lead to a waste of space. A greedy algorithm to do this uses a simple rule: Always put the largest book left in the crate.

```
While there are books left and the crate is not full do the following repeatedly
    Put the largest remaining book in the crate.
```

In what sense is this greedy? Each step uses as much space as possible. It will not necessarily make the best use of the space available but has the advantage of being very quick and simple to follow.

What is the general problem solving rule to follow to use a greedy strategy?

Always take the option that takes you the furthest towards your goal.

One of my favourite sports is Orienteering. This involves finding your way between points marked on a map as quickly as possible. The first time I entered an orienteering contest, I used a greedy algorithm. I always worked out the direction of the next point and tried to run in a straight line towards it – the option that takes me most directly towards the goal of the next point. This is a form of greedy algorithm. Unfortunately it was not necessarily the best algorithm. I spent alot of time running very quickly. Oddly two girls who seemed to be walking round, seemed to keep getting to the next point before I did. They always seemed to already be there when I arrived (even if I left before they did). They were also keeping to the paths which meant that not only were they going more slowly than me, but they were also planning routes that were further. So how come their algorithm was better than mine? Well even though I was taking the options that most directly took me to my goal, I kept hitting problems along the way – finding a river that I had to wade in my way, or a cliff I had to climb up, not to mention gorse bushes I had to crawl through. So even though I appeared at the point of making the decision to be taking the route that would get me to the destination most quickly, the girls were spending more time planning ahead and pre-processing. They were therefore able to miss all the obstacles I ploughed into blindly. That is the problem of a greedy algorithm, what appears to be taking you most directly to solving the goal may actually contain obstacles or lead you into a blind alley.

17. The Right Choice (Choosing Algorithms)

Sometimes I sits and thinks, and sometimes I just sits.
Engraving on a bench, adapted from Punch, Volume 131, page 297 (1906)

Suppose I told you I wanted a pizza for lunch and asked you to sort it out. There are several different approaches you could use to solve the problem. You could for example phone for a take-away. You could take me to a pizza restaurant. You could buy a frozen one and heat it in the microwave, or you could cook one from scratch, base and all. How would you decide which to do? At different times any of these choices might be the best thing to do. Either you choose one at random, or you find out more information first. That is often the problem a software developer is in. The customer has said what they want, but not given enough detail to be sure of the best solution. The person who is to solve the problem must determine more about the problem first. Back to pizza, it is important to do two things: find out the advantages and disadvantages of each possible solution, and find out what other restrictions there are on your choice. For example, who is paying? What pizza topping is required – something straightforward like cheese and tomato pizza or something more exotic like banana and peanut butter pizza? How much time do you have to come up with the pizza? How much time will you have to pay for the pizza? There are many more questions that could be considered, the more that are considered, the better the final decision is likely to be. What then are the advantages and disadvantages of the different choices? An advantage and disadvantage of each is given below – no doubt you can think of many more.

Solution	**Advantages**	**Disadvantages**
Delivery	Low-hassle	May never arrive.
Restaurant	Good quality	Expensive
Frozen	Cheap	Taste like cardboard
Home-made	Any topping possible	May be inedible

If you compare the restrictions you have identified with the advantages and disadvantages, you are then in the position to solve the problem.

In the previous chapters we have looked at a whole range of different algorithms for doing the same thing, whether searching, sorting or whatever. How do you choose? As with pizza, in general there is rarely a single "best" way. Each solution may be the most appropriate in a given situation. The main considerations are how fast the algorithm is and how much space the data structure takes up. Sometimes it is space that matters the most, and sometimes it is speed. However, there can be other factors. For example, for some tasks the most important thing is that you do it accurately without making mistakes. In terms of software, how sure are you that you can write a program that implements the algorithm correctly? Sometimes what matters is the ease with which you work out how to do the task. There is little point spending hours working out the perfect solution to a problem if the worst solution would only have taken minutes to solve the problem adequately. Even if we are only concerned with speed, it may only be one part of the task that needs to be done quickly. We discuss some of these issues in this section.

Time to solve the problem
Often the most important thing is to get the task done quickly. However, this may not just mean using the fastest algorithm, though that is usually important. If the task must only be done once or twice then the time taken to decide which is the best algorithm may be the most important thing. It may take longer than that algorithm will take to complete the task once selected. It is for similar reasons that when politicians are bogged down in some controversial issue, they set up a committee to determine the best course of action. The idea is to use a form of decision making that is so slow, the solution will no longer be needed by the time it has been determined and can be quietly shelved. Usually we do want the solution to be used, however. It is therefore important to decide before we start how much time it is worth spending devising the solution.

If I am late for a class, perhaps because I over sleep, I could sit down with a map of the site and work out the fastest route through the maze of corridors to get from where I am to where I want to be. Alternatively, given the difference between different routes will only be a minute or so anyway, I could spend the time going to the class by the first route that comes into my head. In this situation there is little point spending much time coming up with the best route (algorithm) because the time that matters is the combined time to devise the algorithm and to execute it.

Critical Operations
When considering which of two algorithms to choose, a useful idea is that of critical operation – some important step of the algorithm that you wish to do least often.

We looked at a puzzle that involved a farmer getting across a river.

To cross the river:
1. The farmer crosses with the hen.
2. The farmer returns alone.
3. The farmer crosses with the corn.
4. The farmer returns with the hen.
5. The farmer crosses with the dog.
6. The farmer returns alone.
7. The farmer crosses with the hen.

I claimed that this was the fastest algorithm. However I have no idea how long it would take the farmer to cross the river each time. However I do not need to to be able to say it is faster than some other algorithm. I left you to work out the other equally fast algorithm: here it is.

To cross the river:
1. The farmer crosses with the hen.
2. The farmer returns alone.
3. The farmer crosses with the dog.
4. The farmer returns with the hen.
5. The farmer crosses with the corn.
6. The farmer returns alone.
7. The farmer crosses with the hen.

The farmer just takes the dog across before the corn. I claim this is just as fast, but how do I know without getting a farmer, a corracle, etc and timing them following the two sets of instructions? My claim is based on the fact that they both have the same number of steps. I am assuming that every time the farmer crosses the river it takes roughly the same time and that is what takes the most time. It may be that it is faster to cross with the corn than the dog, or quicker to load the dog if there is no corn on the bank, but such time differences are negligible compared with the time to cross the river. Both the above algorithms require the same number of crossings so they are equally fast algorithms. If you come up with an algorithm that has 8 crossings, then I will deduce that it will take longer. I have decided that *crossing the river* is the critical operation. By counting it I get a measure of the efficiency of the algorithm that I can use to decide between two algorithms – and I can make the decision without ever going near a river. Similarly with algorithms that have been turned into programs, it is possible to decide that some step is the critical operation and work out which of two programs will be faster by comparing them. This can be done without running and timing the programs – just by looking at them.

Consider again the plight of Jean-Dominique Bauby and other people suffering from Locked-in Syndrome. Bauby had only the use of his eyelid to communicate due to his otherwise total paralysis. We argued that the method he used to communicate was inefficient. The person he was talking to recited the letters of the alphabet in turn until he blinked to indicate the correct letter had just been spoken (a form of linear search). We assumed that the number of questions asked should be kept as small as possible: it was being treated as the critical operation. We work out the number of questions that must be asked on average to find the correct letter. Here it is 26 questions in the worst case as we will run through the whole alphabet before finding the correct letter if the letter happens to be Z. Other algorithms considered were based on binary search methods: asking questions such as "Does it come before M in the alphabet?" that rule out half the remainder of the alphabet with each question. These methods take at most 5 questions. Thus the first algorithm takes 26 questions (critical operations) in the worst case whilst the binary ones take at most 5 questions. We conclude that the binary approaches are better, and that Bauby chose a poor algorithm.

Our reasoning above was based on an assumption: we assumed that what mattered most was the number of questions asked. However, given Bauby was paralysed it is possible that blinking took a monumental effort from him. Reading out letters of the alphabet on the other hand, though boring, would not be that difficult for his visitors. If this were so then the step he would want his communication algorithm to minimise, not the number of questions, but the number of blinks. It is blinks that are now the critical operation. The first algorithm required Bauby to blink just once per letter. The second required him in the worst case to give 5 blinks for each letter. This is because assuming a blink is used to mean "yes", the worst case is when the answer to each question is yes. 5 yes answers then require 5 blinks. Thus perhaps Bauby had not chosen a bad algorithm after all. Which algorithm was best for him would depend on what the critical operation was which in turn depends on how easy it was for him to blink. As with any problem solving, it is important to understand the problem in the sense of knowing as much as possible it if you are to devise a good solution.

Pre-processing

Another important consideration with respect to speed, is over which part of the algorithm you wish to be fast. A common feature of algorithms is known as pre-processing. This involves doing some extra work once at the start before doing the task you are really interested in. It is essentially the organisation of things in a way that makes the task easier. We have seen examples of this already. Linear search, as we saw, involves searching straight away. If I want to find a book on my bookshelf, I start at one end and work to the other checking each book. I do not make any attempt to organise things first. That is fine as long as there are only a small number of books or I will be looking for books only occasionally. If there are lots of books it is worth pre-processing them: that is spending some time before doing any searching to organise them. That is what libraries and bookshops do. In both cases there are lots of books and lots of searches for books are being made. The time spent organising the books (which takes a very long time) is worth it because each time anyone searches for a book they save some time. Each person does not save the amount of time the librarian spent organising the books and writing out index cards. That is not the point however. When you add together all the time saved on all the searches made, time is saved in the long run. It is only worth it if there are lots of searches as then you are saving the small amount of time more often. Similarly if the number of books to search through is large, each time someone searches, the time saved on each search is larger. There is thus a point when the number of searches and number of books makes it worthwhile organising first. Below this number there is no point.

Compilation (translating a set of instructions from one language to another that can be executed) is a pre-processing task. We compile a program once and then can execute the code as many times as we wish without having to translate the original again. An interpreter on the other hand does not do this pre-processing. The translation is done every time the program is run as each step is executed. Interpreted languages consequently are slower to execute. When we discussed compilers and interpreters we used the example of my using Arabic directions to get to the market. If I am willing to wait and get the whole set of instructions translated before I start, then if I am to go to the market on several days, and so ultimately save time. If instead I am keen to set off immediately and stop at each junction to get the next instruction translated then on the first trip i will get there at the same time. However, on subsequent trips I would have to repeat the work and continue to stop at each junction. Thus subsequent trips will be slower than if I had had the whole document translated.

Many of the algorithms we have looked at involve some form of pre-processing. For example, binary search and related algorithms require the data to be sorted first – that is why the telephone directory is in alphabetical order. Bucket search requires all search questions to be pre-computed. For example, if we kept appointments as just a list, we would do limited pre-processing. On being told of each appointment we would just write it down on the end of the list. If we use a diary, however, we not only write it down, we need to search for the correct place to write it. Thus on being given new information we do extra work, but the pay-off is that it is now much quicker to look up if a given day is free. The pre-processing is in setting up the bucket array style data structure. In fact, if any special data structure is used, then putting the data in to that data structure could be considered as a pre-processing task.

In many cases, pre-processing involves moving tasks from the point when the task is to be done, to the point when the data is input. Most of the time my sink is surrounded by dirty pots – I usually wash the breakfast pots when I get in from work in the evening, as I do not have time in the morning – I would miss my train. I am thus organising the way I do the task in a way so that I can do the time-consuming part when I have most time. However, if I have porridge in the morning, then by the evening it is set on the dish like concrete and takes much longer to wash. I can solve this by quickly rinsing it under the tap in the morning when I put it by the sink. I am spending a little extra time when the pots arrive at the sink, to speed the actual task of washing them when I finally get round to it.

One reason for pre-processing can be, not that it saves time overall, but that it saves time at a point when time is of the essence. A few years ago I ran a marathon. In the months before doing it I spent lots of time out running, to train myself so that I could do the actual marathon as fast as possible. The total time I spent running was thus much longer than if I had done no training, and taken 8 hours or so walking the 26 miles. In this instance, the total time running (including the training) was not the point. I had plenty of time in the months before hand to spend as long as I liked on the road. What mattered was that I was fast on one particular day. Computing applications are often like that. There is plenty of time to prepare. What matters is that when someone asks for a job to be done, at that point the computer is as fast as possible.

Speed versus Space
When choosing algorithms, there is often a trade-off between use of space and speed. One way to increase the speed with which a task is done is to use up more space. Diaries are an example of this – we are willing to waste many blank pages because it allows us to easily check what we are doing on a given day.

Different modes of transport can also provide a similar trade-off. I could go to work by car. However, the roads are very crowded. Next time you are in a traffic jam, count the number of people in each car. There is mainly only one person per car. Each person in the traffic jam is taking up a whole cars worth of road. I often go to work by Bus. It is usually crammed full with 40 or 50 people. Together we take up a fraction of the road space of the car drivers. If everyone went by bus, we would need fewer lanes on the roads (and there would be fewer traffic jams). Why do all those people prefer to go by car? One reason is the speed. Even with traffic jams it would still be quicker for me to drive to work. I am personally willing to take longer to get to work, in part because of the environmental benefit of using less road space. There is thus a trade-off between speed and space. For other people the speed is more important so they use a different solution to the problem of getting to work from me.

Accuracy
Often the accuracy of the solution is the most important thing. A slightly wrong answer is worse than for example, a slow answer. Returning to the pizza example considered above. Suppose I had asked for a vegetarian pizza. On looking at the options you discovered that using a frozen Meat Feast Pizza was the cheaper than any other form of pizza and could be obtained most quickly as you had one in your freezer already. Given this you decide to give me the Meat Feast. Your solution is close to one that is correct – it is after-all a pizza and in many situations (eg if I was not actually vegetarian might be a good solution). However, in this case accuracy in fulfilling my request was most important as I do not want to eat meat and am willing to pay for the accuracy.

18. The End is Nigh

In my end is my beginning.
Mary Queen of Scots, embroidered motto.

The main aim of this book has been to introduce some of the main data structures and algorithms in a familiar context. We use many of the data structures and algorithms without much thought, naturally choosing something suitable. At a bus stop we form a queue, faced with a pile of chairs, we add and remove from the top. If searching a shuffled pack of cards we use linear search, but given a dictionary we do something much closer to binary search.

When programming the data structures and algorithms, we are effectively just simulating their real world equivalents. Once you are familiar with the basic idea of a data structure or algorithm and can relate it to the real world, it is then much easier to understand the computer equivalent. It is also much easier to program it on a computer. The things being organised become data, and the instructions being followed must be written in a particular programming language and so must be much more precise. However, the ideas are the same.

Despite looking at a wide variety of search and sort algorithms, many more have been invented that we have not mentioned. Some are variations on the algorithms we have examined. Others are completely new. The more specialised algorithms that we have barely touched upon, such as Quicksort (one of the fastest sort algorithms devised) are rarely used in real life outside computer programs. This is because they are more complicated to follow (and similarly to program). However, they are commonly used in programs as they are so efficient. There are also many other areas where a wide range of algorithms have been devised. We have also only touched on some of the most important aspects of algorithms: how you determine their efficiency. We have noted for example that binary search is significantly faster than linear search and mergesort is faster than bubble sort. However, how much faster are they? Are there faster algorithms still? Are they always faster? How do we compare the space requirements of different algorithms? My hope is that this booklet has given you a glimpse of how interesting the subject of data structures and algorithms can be, and whetted your appetite to find out more.

Acknowledgements

I am grateful to Margaret Curzon, Maurice Curzon and Ann Blandford who provided some of the examples described, and to the students of COM2060, COM1501 and COM1000 on whom I tried out the ideas and who suggested many others. I also have had many useful conversations with Harold Thimbleby on the ideas contained here. A similar approach to teaching computing using interactive games and puzzles to that used in part here is taken by Bell et al.

References

1. N. Abensur (1996). *The New Cranks Recipe Book, Weiden and Feld and Nicholson*

2. D. Adams (1979). *The Hitchhiker's Guide to the Galaxy*, Pan.

3. D. Adams (1980). *The Restaurant at the End of the Universe*, Pan.

4. D. Adams (1989). *The Long Dark Tea-Time of the Soul*, Pan.

5. K. F. Armstrong (1942). *Aims to Anatomy and Physiology for Nurses*. 3rd Edition, Bailliere, Tindall and Cox.

6. J-D. Bauby (1998). *The Diving-bell and the Butterfly*, Fourth Estate.

7. T. Bell, I. Witten and M. Fellows, *Computer Science Unplugged*, http://unplugged.cantebury.ac.nz

8. E. R. Berlekamp, J. H. Conway and R. K. Guy (1982). *Winning Ways for your Mathematical Plays.* Academic Press.

9. BMA (1990). *The British Medical Association, Complete Family Health Encyclopedia*, Medical Editor Dr Tony Smith, Dorling Kindersley.

10. M. Bramwell (1976). *The International Book of Wood*, pp 229-230, Mitchell Beazley.

11. K. Connolly (2000). Poland's language police wage war on 'Polglish'. *The Guardian*, page 12, Monday September 4.

12. V. Cronin (1994). *Napoleon*, Harper Collins.

13. M. Dorigo and L. M. Gambardella (1997), 'Ant Colonies for the Travelling Salesman Problem', In *BioSystems*.

14. Dorling Kindersley (1991), *Pocket Encyclopedia of Vegetarian Cooking*, Contributing Editor, Sarah Brown.

15. R. P. Feynman (1992). *Surely You're Joking Mr.Feynman!*, edited by E. Hutchings, Vintage.

16. M. Gardner (1965). *Mathematical Puzzles and Diversions*, Pelican.

17. M. Gardner (1977). *Further Mathematical Diversions*, Pelican.

18. R. Graves (1941). *I, Claudius*, Penguin Books.

19. D. Lacey (2000). ‘England need to summon spirit of Rome’, *The Guardian*, Euro 2000, page 4, Tuesday June 20, 2000.

20. T. Hardy (1874), *Far from the Madding Crowd*, Penguin Classics.

21. R. Harbin (1972), *Waddingtons Family Card Games*, Pan.

22. A. Hodges (1985). *Alan Turing: the Enigma*, Vintage.

23. P. Holland (1995). *How to Make Moving Wooden Toys*, Cassell.

24. B. Kordemsky (1975) *The Moscow Puzzles*, edited by M. Gardener, Pelican.

25. M. Lovric (2000) *Eccentric Epitaphs*, Past Times.

26. G. Mikes (1946). *How to be an Alien*, p10.

27. Mothercare (2000). *Slumber Fun Travel Cot/Play Pen User Guide*, p4.

28. I. Newton (1659). Quoted in M. White, *Isaac Newton, The Last Sorcerer*, Fourth Estate Limited, 1997.

29. D.A. Norman (1993). *Things That Make Us Smart*, p162-167, Addison-Wesley.

30. T. Pratchett (1983). *The Colour of Magic*, Corgi.

31. T. Radford (1999). ‘Brain Scans Turn Thoughts into Words’, *The Guardian*, Thursday March 25, pp 11.

32. J. K. Rowling (1997). *Harry Potter and the Philosopher's Stone*, Chapter 7, Bloomsbury.

33. Dr Seuss (1997). *The Cat in the Hat Comes Back*, The Classic Collection, Harper Collins.

34. Dr Seuss (1998) writing as Theo. Lesieg. *The Pop-Up Mice of Mr Brice*, Harper Collins.

35. W. Shakespeare, *Macbeth,* Bradbury, Agnew & Co.

36. I. Stewart (1992). *Another Fine Math You’ve Got Me Into...*, Freeman.

37. I. Stewart, (1991). *Game, Set and Math*. Penguin.

38. E.O. Wilson (1999) *Consilience: The Unity of Knowledge*, Abacus.

39. S. Winchester (1999). *The Surgeon of Crowthorne*, Penguin.