Mr. Turing's Machine

A Gentle Introduction to Computer Science

By Deryk Barker

Contents

List of Figures

Preface

This book has arisen out of a number of years' experience teaching COMP112 at Camosun College. Although there are many existing textbooks on Computer Science, they are unsatisfactory for at least two reasons:

1. They are intended for a university-level course, which usually either extends over two semesters or has considerably more classroom time per work or higher prerequisites.

2. They do not have the emphasis on introductory programming which COMP112 latterly requires.

The point of this book, therefore, is:

1. To give an introduction to the broad subject of computer science, sufficient both for the reader to follow up any particular are in which (s)he[1] is interested.

2. To emphasise areas of Computer Science which are likely to have practical benefits, either immediately during the Camosun Computing Systems Technology programme or later, during a career in the computer industry.

3. To provide a grounding in the basics of computer programming, using the python language; again so that this can be built upon in later courses.

Moreover, the Camosun CompTech programme has, as a pre-requisite, a basic knowledge of computers and applications, hence the omission from this text of discussions of, for example, spreadsheets and word processors.

To say that this text and its associated course can do no more than scratch the surface of Computer Science is no more than the truth.

[1] Use of either he or she, or indeed "he or she", is distracting in the text. For this text the form (s)he should be taken to mean a human of either gender.

Part I

Introduction

Chapter 1

Introduction

Please allow me to introduce myself
I'm a man of wealth and taste

Mick Jagger/Keith Richards, *Sympathy for the Devil*

1.1 Introduction – Welcome To The Modern World

The modern world, we are often told, depends on the computer. Look around you and note all the devices with which we are now so familiar – desktop computers, laptops, notebooks, netbooks, tablets, MP3 players, CD players, DVD players, media streamaers, "set top boxes", microwave ovens, refrigerators, kitchen stoves, cars, high-definitions TVs, "smart" phones (in fact any kind of cellular phone), watches, cameras.

Fifty years ago most of that list did not exist and the ones that did – such as cars, fridges, stoves – did not use computer technology.[1]

Moreover, even nations which, a decade ago, had very little high-tech involvement, are now undergoing huge social changes enabled by computer technology: just look at the way the "Arab Spring" popular movements in Egypt and other countries utilised "social networking" sites such as Facebook and Twitter[2]

Within one person's lifetime[3] a hugely-expensive, labour-intensive, difficult-to-use machine which required its own climate-controlled environment, dedicated power supply and highly trained support staff, and which few people would ever actually see, has developed into a ubiquitous, tiny, cheap, disposable "chip"; most of us own a number of them, but are probably unaware of precisely how many.

[1] There were also cars, non-HD TVs, fridges, unintelligent phones and the first microwave ovens, none of which use computer technology and all of which did far less that today's equivalents.

[2] Facts which the Egyptian government of Hosni Mubarak were quick to recognise, which is why they attempted to cut the county's internet access.

[3] Yes, mine.

Not only that, but the range of uses for the computer has expanded dramatically: forty years ago there were no spreadsheets, twenty-five years ago no MP3 players, twenty years ago no world-wide web, a decade ago no "smart" phones.

Computer technology has continually improved since the first computers ran their first programs almost sixty-five years ago[4]; it has also, as we have seen become almost ubiquitous and essential to the daily running of most industrialised nations: for example, it was estimated twenty years ago that if there were no computers, the number of financial transactions carried out daily in North America would require **half** of the adult population to be employed by the banking system.

Much manufacturing (not that North America does much of this nowadays) also involves computers in almost every stage: Computer-Aided Design (CAD) and Computer-Aided Manufacturing (CAM) are common, as are robot assembly lines.

Indeed, so familiar are we today with computer technology that we often think we understand it when in fact we have done no more than scratch the surface of what it can do and how it does it.

And herein lies a trap: for some years school boards have been buying into the notion that teaching "computer literacy" (i.e. how to use a computer) is somehow the path to employment.

Once upon a time perhaps, but then once upon a time knowing how to drive a car would also get you a job – they were called chauffeurs.

1.2 Why Computer Science?

You may be wondering what, if any, is the point of studying Computer Science, when even universities and colleges are now offering degrees and other programmes with titles such as *Information Technology*, *Software Engineering* or even (as with the students for whom this text was originally written) *Computer Systems Technology*.

Moreover, your ideal career maybe as a Business Analyst or Web Designer or even assisting users on a Help Desk. What use do such jobs have for a knowledge of Computer Science? Why should you – in fact, why should anybody? – want or need to study computer science?

Well, you might begin by considering:

Clarke's Third Law:

> Any sufficiently advanced technology is indistinguishable from magic.
>
> Arthur C. Clarke[5] *Report on Planet Three*, 1972

[4] The very first program ran on June 21, 1948. See chapter 5 for more details.

[5] Since you ask, Arthur C. Clarke's other two laws are:

1. When a distinguished but elderly scientist states that something is possible he is almost certainly right. When he states that something is impossible, he is very probably wrong.

And this is indeed the way in which all too many people interact with modern technology, treating it as if it were indeed magical and can only be made to work in very particular ways:- e.g. when the moon is full, when the last day of the month is a Thursday, or any other illogical incantation. For instance, consider the headline from the *Weekly World News* of 18 June, 1991:

> MAN CATCHES COMPUTER VIRUS!
> Bizarre illness jamming up his brain waves!

Perhaps that is too obvious (and the Weekly World News too easy a target); but you may well have read some of the ubiquitous "luser stories", such as the infamous "my coffee cup holder [CD-ROM tray to the rest of us] has broken" or Compaq's seriously considering altering their standard error message from "Press any key when ready" to "Press space bar when ready", as users kept calling help desks and asking where they could find the "any" key on their keyboards.

Indeed, just imagine that people bought cars with as little knowledge as they frequently buy computers:[6]

> Helpine: General Motors Helpline, how can I help you?
>
> Customer: Hi! I just bought my first car, and I chose your car because it has automatic transmission, cruise control, power steering, power brakes, and power door locks.
>
> Helpline: Thanks for buying our car. How can I help you?
>
> Customer: How do I work it?
>
> Helpline: Do you know how to drive?
>
> Customer: Do I know how to what?
>
> Helpline: Do you know how to drive?
>
> Customer: I'm not a technical person! I just want to go places in my car!

Perhaps most telling of all is to be found in the British TV situation comedy *The IT Crowd* in which the technical support team's standard response to a phone call for assistance is "Have you tried turning it off and on again? Are you sure it's plugged in?"

---

2. The only way of discovering the limits of the possible is to venture a little way past them into the impossible

These first two laws stem from Clarke's "Hazards of Prophecy: The Failure of Imagination", in *Profiles of the Future* (1962), although the second was simply offered as an observation and its status as a "law" was conferred by others.

The third, and most famous, law was first published in the 1973 revision of *Profiles of the Future*; Clarke wrote: "As three laws were good enough for Newton, I have modestly decided to stop there."

[6] You can find both of these stories and more at http://monster-island.org/tinashumor/computer.html Some of these stories are probably even true.

1.3 What is Computer Science?

1.3.1 What is Science?

OK, so we need some knowledge of Computer Science. And we know (or we think we know) what a Computer is. But what about Science?

There are people who would dispute that there is any such thing as "Computer Science":- science, these people say, is based on the observation of real-world phenomena. Hypotheses are formed to explain the phenomena, these hypotheses become theories when backed up with predictions which can then be tested.[7]

Given this, these people claim, there can be no such thing as "Computer Science"; what we call by that name is not based on the observation of real computers "in the wild". Some will go further and tell you that "Computer Science" is really a branch of Mathematics. As evidence, they adduce that fact that "real" sciences have "theories" which are testable, whereas Computer Science has "theorems" which are proved.[8]

These are questions for philosophers, not computer scientists (who will probably ignore the philosophers in any case – because they'll be too busy building an interactive philosophy web site).

What we are discussing in this book is the subject area generally known as "Computer Science"[9], which we shall define (at least the outlines) below.

1.3.1.1 Science Is Not The Same As Technology

Technology results from the application of science to the solutions of practical problems. Many particular sciences may contribute to one particular technology – e.g. a modern automobile, which combines elements of physics, chemistry and computer science (in cars built within the last couple of decades).

Incidentally, can you think of at least three other sciences which contribute to the automobile? For another contemplative piece of homework, can you list say five common electronic devices you will find in today's homes or cars which rely on computer technology. Can you think of one that doesn't?

Obviously – and this really should be obvious – technologists need to have some understanding of and familiarity with the underlying science(s) of their field. We should not consider hiring somebody with no knowledge of physics to build a bridge for us – not if we are sane and value our lives.

On the other hand, a physicist is not the right person to design the bridge either.

---

[7] Incidentally, the dismissing of the Theory of Evolution as "just a theory" makes no more sense than to dismiss the Theory of Gravity as "just a theory". It shows a basic ignorance of what scientists mean by "theory".

For an amusing sidelight on this see The Onion's article on the alternative "theory" of Intelligent Falling (http://www.theonion.com/content/node/39512).

[8] These people also frequently point out that having the word "science" as part of the title of a field of academic study usually guarantees that it is not scientific at all: they frequently use the example of "Social Science" as illustration.

[9] Or, if you prefer, "The Subject formerly known as Computer Science".

But if you have no understanding of the underlying science, then you are easily misled by people who have, however shallow their understanding may be. As the old saying has it, "in the kingdom of the blind, the one-eyed man is king".

1.3.2 What Is Computer Science?.

Many people probably feel they know what Computer Science is, although if you asked a cross section of ten people on the street (and they probably would be cross at being stopped on the street and asked silly questions) what Computer Science is you might get ten different answers.

And they would probably all be wrong. Or at least, incomplete.

Part of the reason is that Computer Science is a vast field of study, encompassing many areas. Those cross people you asked would probably suggest that Computer Science is at least some of the following:

1. The Design and Study of Computers
2. Computer Programming
3. Learning to Use Computers

Now while all of those topics do indeed fall under the general heading of Computer Science, none of them covers the whole field. Here is why.

1.3.2.1 Computer Science is not (just) about the Design and Study of Computers.

While there would be no computers without Computer Science[10], we should never forget that:

> Computer science is no more about computers than astronomy is about telescopes, biology is about microscopes or chemistry is about beakers and test-tubes. Science is not about tools. It is about how we use them and what we find out when we do.
>
> M.R. Fellows and I.Parberry: "Getting Children Excited About Computer Science", *Computing Research News*, vol 5, no 1 (January 1993)

We shall be (briefly) studying the design of computer systems later in the book.

1.3.2.2 Computer Science is not (just) about Writing Computer Programs

Programming is an important part of computer science, but primarily as a tool to implement ideas and solutions. A program is only a means to an end, not an end in itself.

People (computer nerds excepted) do not buy computers in order to be able to write programs; they buy them in order to be able to *run* programs.

[10] We shall examine the history of the computer in chapter 5.

Computer programs are the means to *solving problems*, a phrase which will crop up again (and again) in this book.

1.3.2.3 Computer Science is not (just) about Learning How To Use Computers.

We don't expect to learn how to write in an English Literature course, nor do we describe somebody who knows how to drive a car as an "automotive expert".

Many people are very good at using computers for specific purposes, in the same way that many people can drive. Being able to drive does not mean that you understand how the car works or that you could design a new one.[11]

Similarly, being able to use the computer does not imply any understanding of its internals. The Apple Macintosh is an excellent example of a system that is specifically designed so that users do not need to understand what is going on "under the hood" (and frequently cannot find out).

Unfortunately, we have yet to build a completely foolproof computer. And yet...[12]

1.3.3 So, put me out of my misery: what *is* Computer Science?

In 1986, Gibbs and Tucker proposed the following definition:

> "Computer Science is the study of algorithms, including their:
>
> - formal and mathematical properties
> - hardware realisations
> - linguistic realisations
> - applications"
>
> Gibbs, NE & Tucker, AB, "A Model Curriculum for a Liberal Arts Degree in Computer Science", *Communications of the ACM*, vol 20, no 3 (March 1986)

Which is as good a working definition as any and is the one we shall be using for the remainder of this book.

But algorithms deserve their own chapter and so, without further ado...

[11] Although when I look at the PT Cruiser, I begin to wonder.

[12] I'm thinking of making this one of Barker's Laws of Computer Science.

Chapter 2

Algorithms

> But the only thing I knew how to do
>
> Was to keep on keeping on.
>
> Bob Dylan, *Tangled Up In Blue*

2.1 What is an Algorithm?

I'm so glad you asked.

Here are some definition of algorithm found on the Web:

- A procedure or formula for solving problems
- A finite set of well-defined rules for the solution of a problem in a finite number of steps
- A step-by-step procedure used for solving a problem
- An organised procedure for performing a given type of calculation or solving a given type of problem. An example is long division
- A method for solving a particular problem which is guaranteed to terminate in a finite time
- A procedure used to solve a mathematical or computational problem or to address a data processing issue. In the latter sense, an algorithm is a set of step-by-step commands or instructions designed to reach a particular goal

As you will note, there is a good deal in common between these various definitions.

Donald Knuth[1] has proposed the following five essential properties for an algorithm:

1. Finiteness: "An algorithm must always terminate after a finite number of steps ... a very finite number, a reasonable number".
2. Definiteness: "Each step of an algorithm must be precisely defined; the actions to be carried out must be rigorously and unambiguously specified for each case".
3. Input: "...quantities which are given to it initially before the algorithm begins. These inputs are taken from specified sets of objects".
4. Output: "...quantities which have a specified relation to the inputs".
5. Effectiveness: "... all of the operations to be performed in the algorithm must be sufficiently basic that they can in principle be done exactly and in a finite length of time by a man[2] using paper and pencil[3]".

2.1.1 And the name?

Incidentally, you might like to know that the world "algorithm" comes from the name of the 9th century Persian mathematician *Abu Abdullah Muhammad Ibn Musa al-Khwarizmi*.

His name was Latinised in Europe as *Algorismus*, whence "algorithm".

al-Khwarizmi's book *Kitab al-jabr w'al-muqabala* ("Rules of Restoration and Reduction") not only specified the first algorithmic procedures, its title also gave us the English word "algebra". Not too many non-English-speakers have contributed two new words to the English language.

2.2 Algorithms in Daily Life

Algorithms are not limited to mathematical or computer problems. In fact, you might be surprised at how frequently we encounter and use algorithms in our daily life.

Some examples include:

- Cooking Recipes.
- Instructions for assembling things.

[1] Knuth is well-known as the author of a series of books entitled The Art of Computer Programming – volume 1 appeared in the 1970s, volume 4A in 2011 and he estimates volume 5 will appear around 2020 – and as the author of the TEX typesetting language and package. This book is laid out by TEX.

[2] Knuth was writing in 1968, no doubt today he would write "person".

[3] Paper and pencil might not be the appropriate vehicle for some of the algorithms in daily life discussed below.

- Directions for getting to places[4].
- Performing routine tasks.

Unfortunately some "algorithms" fall short of the definition[5].

2.2.1 A pseudo-algorithm

This is a favourite example of computer scientists, but it is not apocryphal. In fact, while preparing for the first time the course for which this book is intended, I happened, while standing idly in the shower[6], to read the label on the back of my own bottle of shampoo.

The instructions read as follows:

1. Wet your hair
2. Lather your hair
3. Rinse your hair
4. Repeat

Let us examine this "algorithm" and see how it measures up to Knuth's criteria:

- "Precisely defined"? There is actually quite a lot of ambiguity in this "algorithm" (most of which must be resolved by the user's "common sense" – for example, what does "lather your hair" mean?)
- "Finite" number of steps? Unfortunately, step four appears to leave us endlessly repeating (steps one to three, presumably) with no end in sight.

2.3 Algorithms and Problem Solving

Life itself presents us with a series of problems to be solved and tasks to be performed. Some of them lend themselves to automation, some don't.

For the purposes of this text, we want to translate the procedure for solving a problem, or performing a task[7], into an algorithm that is:

1. Correct: there is no point in carrying out an incorrect algorithm.
2. Efficient: given a choice between several alternative algorithms we should obviously choose the most efficient. Unfortunately, as we shall see in chapter 4, we don't always have a simple definition of "efficient".

[4] Although not those which begin "well, I wouldn't start from here".

[5] This particularly applies to flat-pack assembly instructions apparently translated from Korean into English by a person with no direct knowledge of either language.

[6] As you do.

[7] With luck we'll also learn to recognise those problems which cannot (reasonably) be computerised.

3. And each step must be "computable"[8]: if there is even *one* step in the algorithm that we do not know how to perform, then we cannot complete the algorithm.

2.3.1 Why Bother?

The magic word here is "automation", because if we can specify an algorithm to solve a problem, then we can automate its solution.

2.3.2 The Computing Agent

A computing agent is an entity capable of carrying out ("executing") the steps described in the algorithm, i.e. we say that the agent can execute the algorithm.

A computing agent does not have to be made of silicon, copper and plastic. It could be:

- A person (in fact, before the 1950s the word "computer" signified a person who made their living by doing calculations, by following well-known algorithms).
- DNA: Scientists are indeed experimenting in the area of biological computation, but it is also arguable that our DNA is a complex algorithm for producing a new human being. Who (or what) the "programmer" of this DNA might be is a discussion well beyond the scope of this course.
- Or, indeed, our computing agent could actually be a computer.

2.4 Computer Science as The Study of Algorithms

In chapter 1 we adopted Gibbs's and Tucker's definition that Computer Science is the study of algorithms, including their:

- formal and mathematical properties
- hardware realisations
- linguistic realisations
- applications

So let us take a brief look at each of these areas in turn.

[8] You may be wondering about this. But consider the innocuous-looking "step": **find the largest even prime number**. Simple to state, impossible to perform (because, and I really hope you already know this, there is no such thing as the largest prime number). So any "algorithm" that includes this step is not worthy of the name.

2.4.1 Formal and Mathematical Properties

Computer Science is about designing algorithms to solve a wide range of problems.

A major part of Computer Science is finding algorithms to apply to new problem domains and determining whether problems are (efficiently) computable: in other words, whether it is feasible to attempt to solve certain problems by computer.

Believe it or not there are certain classes of problems which are known to be *unsolvable* algorithmically (some are unsolvable within any reasonable amount of time, some are simply unsolvable, period). These problems are not necessarily complex to define or understand.

We shall briefly investigate this in chapter 12.

Computer Science also involves studying the behaviour of algorithms to decide whether they are correct and how efficient they are.

For example, when we have a choice of algorithms, how can we choose between them if we have no way of establishing their relative correctness and/or efficiency? We shall be touching on all of these areas in this book.

2.4.2 Implementation in Hardware

Computer Science is also about designing and building computer systems capable of executing algorithms.

Some attributes which have characterised the hardware field in the last few decades are:

Continuous technological advances The increase in miniaturisation, speed, complexity and capacity of computer systems continues to grow. It is probable that no other field in human endeavour has ever changed so quickly or become so prevalent. Today's average MP3 player contains more (computer) memory than existed on the entire planet 50 years ago.

More and more powerful computers Today's computers are almost immeasurably faster than even yesterday's. Are there limitations?[9]

Networks were non-existent in the 1950s and early 1960s. They were a highly specialised subject in the 1970s and 1980s. In the 1990s the World Wide Web became a household name and now many people in developed (and developing) countries use a network ("the 'net") every day of their lives.

Parallel computing If one computer can't solve the problem fast enough, how about using two (or more) in parallel? Sounds easy, but it isn't.

[9] One of those irritating chain emails, this one headed "If Cars were Computers", points out that if automotive technology had improved at the same rate as computer technology today one would be able to buy a Rolls-Royce for under $5, it would fit in a matchbox and it would be cheaper to abandon it downtown and buy a new car rather than pay for parking.

The email never explains the advantage of a car which would fit in a matchbox.

And, apart from the obvious smaller-faster-cheaper, what else does the future hold?

Quantum computing Scientists have been talking about using Quantum Physics for computing for about two decades, since David Deutsch published his seminal paper in 1985. There have been some promising developments early in the 21st century, but it looks as if we may have to wait rather longer before quantum computing changes the world (again!) Although we shall look briefly at hardware principles, this course does not go into depth on computer architecture.

2.4.3 Implementation in Software

In 1966 Peter Landin published a paper entitled "The Next 700 Programming Languages", the title deriving from the fact that Jean Sammet had recently published a survey of the 1700 programming languages – used in some 700 applications areas – which had been created to that time.

Currently (see http://hopl.murdoch.edu.au/backdoor.html) there are over 7500 known computer languages, although the majority of them have fallen into disuse.

Computer Science, therefore, includes:

- Programming Language Design

  Designing programming languages and translating algorithms into these languages so that they can be executed by hardware. Human languages are too imprecise to use for programming computers. The "machine code" that the computer "understands" is too complex for humans to (reliably) program.

- Functional Programming (FP)

  In functional programming everything is seen as a (mathematical) function, which transforms its input into its output. There has been much research in this area with regard to "automatic program proving", a goal the computer science community has long sought after. Functional programming can be traced back to LISP (1959), itself based on Alonzo Church's λ-calculus (1930s).

- Object-Oriented Programming (OOP)

  Object-oriented programming attempts to model the real world by programming with "objects" which "know" how to "behave" when sent the appropriate "message". Not all computer scientists are convinced that object oriented programming is necessarily the panacea for all the computer's problems. OOP (as it is usually called) can be traced back to Simula (1965).

- "Visual" Programming

This is a name which has been proposed for programming by connecting icons on the screen, rather than typing in text. As yet, this is a new and largely untested area. Some authors believe that this approach allows too much detail to be ignored.

Again, we touch very briefly on these matters in this book.

2.4.4 Applications

Computer Science also includes:

- Identifying important problem areas (i.e. new uses) for computers.[10]
- Designing software to solve these new problems.
- Finding and Developing New applications:

  The first computers were used mainly for numerical calculations and massive data storage/retrieval. (Although the very first electronic computers – see chapter 5 on the history of the computer – were built in order to fight wars more effectively.)

  Fairly rapidly, the domain of the computer was extended to include business use, graphics, multimedia, games, the Internet, the WWW, cars, house appliances, greetings cards, etc. The list is potentially endless. The average house today probably contains more raw computing power than existed on the entire planet in 1950.

  Applications are generally not within the scope of this course.

[10] It has been suggested that the one application which convinced many that the Personal Computer could be a useful tool rather than simply a tool was Dan Bricklin's and Bob Frankston's spreadsheet application *Visicalc* on the Apple II.

Part II

Programming

Chapter 3

From Algorithms to Programs

Between the idea and the reality
Between the motion and the act
Falls the Shadow

T.S. Eliot, *The Hollow Men*

3.1 Programming Languages

It is probably becoming clear that when we want to program a solution to a problem, the first thing we should do is develop the algorithm and then derive the program from it.

As we have seen, there have been literally thousands of programming languages devised over the last half century.

3.1.1 Problems with Programming Languages

All of these languages will have their supporters. Some can give you rational reasons for their language of choice, others are suffering from the phenomenon known in the zoological world as "imprinting"[1] whereby the first language they learned becomes their favourite and "obviously" the best choice for any problem.

More experienced programmers will agree that different languages have different strengths, making them highly suitable for some types of problems and highly unsuitable for others.

[1] In biology "imprinting" is when a newly-born animal (the phenomenon was first observed in birds) fixates on something which is not its parent, "a process that causes the newly hatched to become rapidly and strongly attached to social objects such as parents or parental surrogates" to use a formal definition. "What you first see is what you love" might be another way of putting it. Although he did not originate the term the phenomenon was most popularly described by Konrad Lorenz in his observations of greylag geese in the 1930s.

3.2 Representing algorithms

But how do we represent algorithms? We need to be able to remember the details, to communicate them to other people, etc. We need some way of writing down an algorithm.

3.2.1 Natural languages

How about writing out our algorithms in a natural (i.e. human) language, such as English? As we have already seen in chapter 1, it is easy to be ambiguous in natural languages. Natural languages also lack precision.

Both of these facts violate Knuth's principles for algorithms.

3.2.2 Computer languages

The problems with using a known computer language (and which, of the thousands that exist, should you choose?) for expressing algorithms are:

- Anyone wishing to read the algorithm needs to be familiar with the particular language in which it is expressed. This is not really practical.
- When we write computer languages (as you will discover) we tend to be focusing more on things like punctuation, grammar and syntax – i.e. ensuring that we are writing out correct C (or java or C++ or any of the others). This prevents us from seeing the forest for the trees[2].

3.2.3 Pseudocode

Most programmers (worthy of the name) will initially define an algorithm using "pseudocode" and then convert that to the programming language of their choice (or, more often, the programming language they have been instructed to use).

As the name suggests, pseudocode is not "real code", i.e. it is not a precisely-defined language. Typically, everybody invents their own pseudocode, by using a highly restricted subset of (usually) English. There are no rules, except that the pseudocode must help make the algorithm explicit, clear and unambiguous.

3.2.4 An example – adding two decimal numbers

Here is a problem we can all understand – adding together, digit-by-digit, two decimal numbers:

[2] Or, as they would say where I grew up, the wood for the trees.

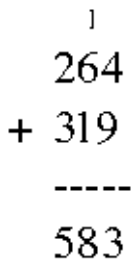


Figure 3.1: Decimal addition

So, how are we to represent the algorithm?

Let us examine the possibilities.

3.2.4.1 In English

Let us imagine how we might begin:

> Add the two rightmost digits of *the first number* and *the second number* and call the result the rightmost digit of *the sum*. If the value of *this digit* is greater than or equal to 10, set the value to *carry* to 1 and subtract 10 from *the sum*, otherwise set the value to *carry* to 0.
>
> Now add the two next-to-rightmost digits of *the first* and *the second numbers* and *the value to carry from the last step* and make the result the next-but-rightmost digit of *the sum*....
>
> Keep doing this until you run of out digits.[3]

And so on....

The obvious problems? How about:

- all of the quantities in *italics* have to be referenced; references such as *the first number* can easily become ambiguous.
- How do we define the "next-but-rightmost" digit of a number?
- What, exactly, is the "this" that we "keep doing"?
- How do we know when we have "run out of digits"?

[3] This is probably no more than a semi-formalised version of the way you learned addition in elementary school.

And this is a simple algorithm!

OK, we could have begun with a preamble ("To calculate the sum of two decimal numbers, say a and b, whose digits are referred to as....") but there will always be too much detail to supply.

Essentially natural languages:

- are imprecise.
- can be extremely long winded.
- if written as one or more unstructured paragraphs are hard to follow.
- rely on context or experience; different people may have different interpretations of the words used.

The "bottom line", as they say, is that it is very, *very* difficult to represent algorithms in a human language – precisely because their imprecision and ambiguity (without which, for example, poetry would be impossible) undermines the necessary qualities that make algorithms algorithms.

3.2.4.2 In an actual, real, honest-to-Goddess computer language

The problems here are several:

- Some languages are good for certain types of problems and less good for others; we can actually have our thinking forced into irrelevant or incorrect directions by an inappropriate choice of language; in other words, when we try to "think" in a computer language, the form of the language itself tends to direct our thinking, towards methods which are "natural" in the language and away from methods which are not.
- Real computer languages have messy details which must be taken into account – how do we represent the individual digits of each number, for example? Where do we get the original numbers *from*? Can we do arithmetic directly with the inputted values, or is some form of conversion necessary? Does our chosen language do such conversions automatically or must we do them manually[4]?
- If we are worrying about the details of syntax, punctuation, conversion etc. then we are not seeing the big picture, we are not seeing the forest for the trees, as suggested above in section 3.2.2.

3.2.5 Pseudocode

Pseudocode is a compromise between the convenience of natural languages and the precision of artificial ones.

Computer scientists use pseudocode to express algorithms. Pseudocode is:

[4]"That is, once a year." Muir & Norden, *Balham: Gateway to the East.*

- Composed of English-like (or other natural language) constructs, but
- modelled to look like statements in typical programming languages.

Moreover, individuals each have their own personal version of pseudocode, but the important thing is that it is *consistent* and easy to understand for other readers.

3.2.5.1 Required operations

Before we can attack our problem and write out the pseudocode, we need to consider what basic operations we need to be able carry out in order to be able to complete our algorithm:

- Getting input and producing output
  - Input (or *read* or *get*) the two numbers.
  - Print (or *display or write*) the outcome.
- Referring to values within our algorithm
  - Add together the rightmost digits of the two numbers.
- Doing something if some condition is true
  - If the outcome is greater or equal to 10 then...
- Doing something repeatedly
  - Do this for all the digits in the numbers.

3.2.5.2 Pseudocode primitives

Our basic operations will include input and output (input, print in the above example) and computation (add 1 to i).

In addition we need to control the order and frequency of performing these operations, this is known as the *flow of control*.

The *Bohm-Jacopini Theorem*[5] proved in the 1960s that any program can be built using three basic flow of control constructs or primitives:

1. **Sequence:** by which we mean performing one operation after another, in the order stated (in the program)
2. **Selection:** choosing between two (or more) mutually exclusive alternatives: *if A is true then do X otherwise do Y* (note that we shall never carry out both X and Y)

[5]Oddly enough, the work of two men named Bohm and Jacopini. It is such happy coincidences that make computer science such a fascinating area of study.

3. **Iteration:** (also known as repetition): perform a certain action (or group of actions) a certain number of times, or as long as a certain condition is true or until a certain condition is true (note the difference between these last two)

3.2.5.3 Variables

We shall frequently need to *remember* a value for a later part of a computation/algorithm, which essentially means that we need to be able to refer to the value *by a name* – the equivalent, in number puzzles, of something like: "[take away] the number you first thought of".

An *assignment* (or *assignment statement*) is the method by which we associate a name with a number[6].

```
name = value
```

The name on the left side of an assignment refers to a variable[7]. As the name suggests, the value represented by a variable can change as the program runs.

Variables have several important attributes, of which we are concerned with two here and now:

- Name: every variable must have a name. Different languages have different naming rules.
- Value: every variable has a value. This value can be updated (i.e. changed); it can also be copied to, e.g., another variable.

Of course, another assignment can potentially change the value of the variable (which is why it is called a variable)

```
counter = 1
print counter
counter = counter + 1
print counter
```

The first print will produce the value 1. The second print will produce 2. This is because

```
counter = counter + 1
```

is *not* a mathematical equation (it would be a contradiction); it is saying that from this point in the program on (until it is changed again) the name *counter* now stands for the value on the right[8]. In this particular example, that value is obtained by taking the current value of *counter* and adding 1.

[6] Believe it or not, the mechanism of this apparently-simple concept varies quite dramatically between different languages. This is not relevant at this point.

[7] Not to be confused with variables in mathematics which, once we have discovered their value *do not change*.

[8] Although some programming languages, specifically those falling under the *functional* paradigm, do not allow assignment to occur more than once. This does not make them any less powerful, but it does require a somewhat different programming style.

3.2.5.4 Computations

Computations calculate values. The resulting value may be saved via an assignment statement or it may be used in some other way.

We have already seen, in the section above, a simple example of this:

```
counter = counter + 1
```

Which is a specific example of the general:

```
<variable> = <expression>
```

There is at least one important point to make about the above: it is showing us the *syntax* (see section 3.2.5.5 below) of a general (the *most* general) assignment statement, in other words how such a part of the language (in this case, pythonic pseudocode, as defined below) must be laid out.

Note that the < and > characters are not part of that syntax (they are called *metasymbols* if you really want to know) and what lies between then indicates the kind of thing which is allowed in its place.

So, the definition above means that an assignment consists of the name of a variable (that's what *<variable>* signifies) followed by the equals sign then an expression. The rules for an expression are complex, but a few examples will give a general idea:

```
fred = 5
jane = 4 + 3
mungo = 6 + (jane * fred) / 12
```

The result of any of these assignments is that afterwards the name on the left hand side of the = now refers to whatever the value of the right hand side is and we can use that name whenever we want that value.

3.2.5.5 Syntax

The syntax of a language is the set of rules which given building a valid sentence (or program) in the language, regardless of what it means.

In general, pseudocode is not strict on syntax. For instance, all of the following can easily be understood (and that is the whole point of pseudocode, understandability) to mean the same thing:

```
set the value of sum to 0
set sum to 0
sum = 0
```

we can even collapse assignments together set the value of sum to 0 and set the value of GPA to 0 as:

```
set sum and GPA to 0
```

or

```
sum = GPA = 0
```

For the purposes of this course, we shall be restricting our pseudocode to what is legal in the python language (or as close as is reasonably possible). The reason for this is that python is very close to pseudocode (as used by the majority of programmers) anyway. Some experience programmers call programming in python "programming in pseudocode"[9].

We are going to use python throughout this book.

In the above cases then, the forms we shall use are:

```
sum = 0
sum = GPA = 0
```

which are both legal python.

3.2.5.6 Input and Output

The computing agent (computer) needs to communicate with the outside world:

- *Input* operations allow the computing agent to receive from the outside world data values to use in subsequent computations
- *Output* operations allow the computing agent to communicate results of computations to the outside world. Unfortunately, input and output are the areas where there tend to be the biggest differences between languages – indeed, between computers.

The most frequently-used pseudocode constructs for input and output (hereinafter referred to as I/O) are something like:

Input

```
obtain value for <variable>
input <variable>
```

Output

```
print <expression>
output <expression>
```

When you are writing pseudocode, as stated before, you make the rules. What you write, though, must be clear to other readers.

[9] When the author was first trying to persuade his colleagues that teaching python was a good thing, he showed them an algorithm in the pseudocode of the introductory textbook then in use – this would be around 2001-2 – then the same algorithm as a python program. The initial reaction of most of his colleagues was "but there is no difference!"

There was, but it was so small as to be virtually unnoticeable.

3.2.5.7 The decimal addition problem in pseudocode

In this book we shall be mostly using the python language as pseudocode, or what I am going to refer to as "pythonic pseudocode" as it will occasionally violate the rules of the python language.[10] Although there are details in an appendix, this text will introduce various features of python as and when they are required.

In the pseudocode below the notation $a_i$means the i*th* digit from the right of the number a (so $a_1$means the rightmost digit itself):

1. input $a_m ... a_1$
2. input $b_m ... b_1$
3. $carry = 0$
4. $i = 0$
5. while i is less than m do steps 6 to 8
6. $c_i = a_i + b_i + carry$
7. if $c_i >= 10$, then $c_i = c_i - 10$ and $carry$ =1; otherwise $carry = 0$
8. add 1 to i
9. c_m = carry
10. print c_m $c_{m-1} ...$ c_0
11. stop

For now this is as far as we can go with this problem.

The difficulty is that we do not know how to represent the *individual digits* of the numbers; it is all very well to write $c_i = a_i + b_i + carry$ in our algorithm, but translating this into a real language requires us to make decisions about data representation that we are not ready for as yet.

Nevertheless, the principle remains the same:

1. Write out the pseudocode
2. Translate to the implementation language

Note, incidentally, that the above pseudocode is not well laid out: you have to read it very carefully to discover just which steps are being repeated by the *while*. Pseudocode (and program) layout is a very important tool for conveying the *structure* of the code to the reader.

Good (pseudo)code layout is a good habit to acquire when learning to program. Fortunately – although some will dislike it – python *forces* us to acquire this habit.

[10] And we don't want our pseudocode to distract us from the algorithm itself – this is why we use pseudocode in the first place.

3.2.5.8 Pseudocode to Python

Here are some simple (except for the last one) rules for translating the pseudocode we have employed above into legal python.

With practice you will find that your pseudocode become very close to legal python.

The major advantage of this, even if python is not to be the implementation language of choice, is that you can *run the algorithm* in its pythonic form before translating to the implementation language.

If the program then fails, you know that the fault, dear Brutus[11], is not in algorithm but in our implementation of it.

So, here are our simple rules:

1. Get rid of the step numbers at the beginning of each line. (After a while you will probably not include these in the first place.)

2. Remove the stop statement (python will stop when it "runs out of program", as will most languages, with the notable exception of assembler – see chapter 9 for more information).

3. Recast assignments and calculations as legal python: e.g. the statement:

   ```
   add 1 to i
   ```

   becomes

   ```
   i = i + 1
   ```

4. Translate output statements to legal python:

   This is easy:

   ```
   print X
   print 5
   print 3+2
   print "hello"
   ```

   are all legal python statements. In each case the system displays on the screen the value of what comes afterwards.

5. Translate input statements into legal python

   Translating input pseudocode is a little tricker and largely depends on what type of input we are expecting.

   The simplest form of reading input in python is:

   ```
   <variable> = input ()
   ```

[11] Apologies to W. Shakespeare.

And yes, those parentheses ("round brackets") are required.

If you don't quite understand the following few paragraphs (from here to "The Bottom Line") don't worry, it will become clearer with a little practice.

There's actually quite a lot going on behind the scenes when we run this. The *input ()* reads in whatever the user types at the keyboard and then calculates its value *as if it were a python expression*!

Here, for example, is a small python script which reads in a number from the user and then outputs twice that number (yes, I know it's not very interesting):

```
number = input ()
print number * 2
```

Let's see what happens when we run this. The first thing to notice is that, because python is *interpreted*, each statement is executed as soon as you type it in, which means that after entering the first line, the system waits for you to type in the value for number before you get to type in the print statement.

This tends to be confusing.

So instead we'll put the script into a file and run it under Linux from the command line. Here is a sample set of three runs, but first we'll just confirm (using the *more* command) the contents of our script file. The user input is in **bold**.

```
$ more double.py
number = input ()
print number * 2
```

which is what we expected, so let's test it:

```
$ python double.py
2
4

$ python double.py
2+2
8

$ python double.py
15*5-(6+7)
124
```

As can be seen, the input typed is not necessarily simply a number; python is quite capable of reading in an arithmetic expression such as 15*5-(6+7) and calculating its value (62).

This is very helpful, but what happens when we want the input to be something other than a number, i.e. text?

Now python allows us to use the '*' symbol with text, in a very natural fashion:

```
>>> print 'Ho!' * 2
Ho!Ho![12]
```

But, if we try to enter a text string into our existing program, see what happens:

```
$ python double.py
Hello
Traceback (most recent call last):
   File "double.py", line 1, in ?
     number = input ()
   File "<string>", line 0, in ?
  NameError: name 'Hello' is not defined
```

Admittedly it's not the world's most helpful error message, but think about what we said about the way *input()* works: it reads what the user types and tries to use it *as a python expression.* If we were to type in:

```
print Hello
```

we'd probably realise that either we have a variable called *Hello* whose value will be displayed, or we shall get an error message.

Which is precisely what is happening when we try to type *Hello* as input, python looks for a variable of that name, in order to print its value.

If we actually wanted to display the string Hello in a program, then we'd type:

```
print "Hello"
```

and that is certainly one way for the user to avoid the problem with *input ()*, type the quotes as well:

```
$ python double.py
"Hello"
HelloHello
```

But it's probably a bit much to expect our user to remember to use the quotes – not to mention dealing with the error message when (s)he forgets them.

[12] Make that multiplier 3 and we have the so-called "Santa Code".

The Bottom Line

So, if you know that your input is not simply a number (or if you want to perform error-checking, in case of poor typing by the user) then you must use python's *raw_input ()*, otherwise use feel free to use *input()*.

We'll amend our doubling program and try it out, showing the file's contents first:

```
$ more double2.py
number = raw_input ()
print number * 2

$ python double2.py
Hello
HelloHello
```

OK! So we have it working perfectly with text.

How about numbers?

```
$ python double2.py
5
55

$ python double2.py
2+2
2+22+2
```

Close, but no cigar.

It is worth studying the difference between the way this behaves and the way the previous version worked, particularly when the input is 5, or 2+2.

In this case, *raw_input* is reading "5" or "2+2" as *strings* (i.e. sequences) of characters and then using the "*" operator accordingly (i.e. produces that many copies of the original string).

Using *input()* does, therefore, have its advantages, especially when we know we are dealing with numbers; actually, when we are not dealing with strings.

We can, for example, read in several variables at a time – as long as the user enters commas between them:

```
>>> x,y,z = input ()
3,5,7
>>> print x
3
>>> print y
5
>>> print z
7
```

Another very useful facet of the *input ()* method is that we can type in python listsf[13] as input, which will be useful in our decimal addition algorithm.

```
$ python double.py
[1,2,3,4,5]
[1, 2, 3, 4, 5, 1, 2, 3, 4, 5]
```

We have entered a list with 5 elements and the '*' operation in python has "doubled" the list. Note the difference when we use *raw_input ()*:

```
$ python double2.py
[1,2,3,4,5]
[1,2,3,4,5][1,2,3,4,5]
```

You should make sure that you understand the difference.

The bottom line is that if you are *certain* that the input will always be a number (or numerical expression) then use *input ()*, otherwise (and especially if you need to do error-checking) use *raw_input ()*.

3.2.5.9 An example

Let us put all of this together and build a small example. We'll write out pseudocode first, then convert to python. As you'll see there is very little difference. Experienced programmers can often write straight python (or something very close to it) as their pseudocode. (Which is why so many experienced programmers enjoy python: it is "runnable pseudocode", or very nearly.)

Problem: write a program which will calculate the average of three numbers.

Pseudocode:

```
1. input N1, N2, N3
2. average = (N1+N2+N3)/3
3. print average
```

Python:

```
N1,N2,N3 = input ()
average = (N1+N2+N3)/3
print average
```

and here's what happens when we run it:[14]

[13] A *list* is a sequence of values which can be referred to by their *position*. We discuss lists in more detail later in the chapter.

[14] Notice that the user can enter multiple values for this assignment, *as long as they are separated by commas*. And as long as the number of comma-separated values entered matches the number of comma-separated variable names on the left had side of the assignment.

```
$ python average.py
1,2,3
2

$ python average.py
2,4,9
5

$ python average.py
2,4,10
5
```

Wonderful!

Except...surely those last two should not give the same answer?

This is a typical example of what's called an *implementation detail*[15]; the problem lies in the second line:

```
average = (N1+N2+N3)/3
```

When our inputs are 2, 4 and 10 (say) then *N1*, *N2*, *N3* and 3 are all integers and python assumes you intend the result to be an integer, so it uses a truncating divide – in other words any remainder is simply thrown away and there is never any rounding up. We could insist that the user makes it clear that non-integers will be involved in the calculation:

```
$ python average.py
2,4,10.0
5.33333333333
```

But again, as when we insisted that the user supplies the quotes for our doubling program, we are the mercy of bad typing or forgetfulness.

All we need to do is convince python that non-integers come into the calculation, so how about changing that second line [16]:

```
$ more average2.py
N1,N2,N3 = input ()
```

[15] And, as we all know, the devil is in the details.

[16] Another way to achieve the same result is to *force* the numbers to be floating point by using python's built in *float* function:

```
average = float(N1+N2+N3)/3
```

but be sure you understand why neither

```
average = float(N1+N2+N3/3)
```

nor

```
average = float((N1+N2+N3)/3)
```

will do what you want. Both float "too late" - i.e. after the (integer) divide has been carried out.

```
average = (N1+N2+N3)/3.0
print average

$ python average2.py
2,4,10
5.33333333333
```

Although these details might seem awkward, they are trivial compared to the amount of detail you have to deal with in most computer languages.

This is essentially why we don't use actual languages for algorithm design but use pseudocode instead.

3.2.5.10 Conditionals

If all the computer could do was carry out a specified sequence of steps, no matter how quickly, then it would not be much use to us. However, the computer's ability to choose between two mutually exclusive alternatives and to repeat a series of steps – known as altering the *flow of control* – enable us to use it as mankind's[17] first and only *general purpose* tool.

Conditionals are the way we make the choice between two (or more, although two is sufficient) alternatives. The general format is:

```
if <condition> then
   <then-part>
else
   <else-part>
```

The <*then-part*> and <*else-part*> can each contain any number of steps (including none); the way the above is performed is:

1. evaluate <*condition*> which will be either true or false[18]
2. if it is true, then perform the operations in the <*then-part*>
3. if it is false, then perform the operations in the <*else-part*>

The only slight change to convert that into pure python is that we use colons (all flow of control constructs in python use colons), the first instead of the word "then" and the second after "else"; so the exact pythonic equivalent is:

[17] No gender bias intended. A lot of confusion is caused because we use the term "man" in compound words both to represent what the Ancient Greek's called ανδρος ("andros") – i.e. man, as opposed to γυναικος ("gunaikos", the root of "gynaecology") woman – and ανθροπος ("anthropos", root of "anthropology"), i.e. (hu)mankind, as opposed to animals and plants, etc.

[18] Values and variables which can only be true or false are known as *booleans*, after the English mathematician George Boole (1815 – 1864). We shall encounter him again in chapter 7.

```
if <condition>:
   <then-part>
else:
   <else-part>
```

One other thing: in many computer languages there is a symbol (or symbols) used to delimit (i.e. mark the beginning and end of) the <*then-part*> and <*else-part*>.

Python does not do this and in python everything is controlled by the indentation – in other words everything which is part of the <*then-part*> must be aligned at the left and begin further to the right than the *if* itself[19].

Think of it as a WYSIWYG (What You See Is What You Get) programming language.

A quick couple of examples to make things clear:

```
$ more if1.py
number = input ()
if number >= 10:
   print 'big',
else:
   print 'small',
print 'OK'

$ python if1.py
9
small OK

$ python if1.py
11
big OK
```

In this example the <*then-part*> and <*else-part*> both consist of a single *print* statement.

The final *print* is *not* part of the *if* at all and does *not* depend on the condition.

```
$ more if2.py
number = input ()
if number > 10:
   print 'big',
else:
   print 'small',
   print 'OK'
```

[19] In fact, a simple rule of thumb to remember whether a colon is needed or not is: do I need to indent the *next* line further to the right that this one?

If the answer is "yes", then *this* line needs a colon at the end.

```
$ python if2.py
9
small OK

$ python if2.py
11
big
```

Because of the indentation, the <*else-part*> now contains two *print* statements, and so the output string *OK* is now only produced when the condition is false.

Some beginning programmers find this insistence on indentation a pain in the anatomy.

This is the wrong approach.

Think of it, instead, as developing good habits.

3.2.5.11 Compounding Conditionals

It is possible to combine or invert (i.e. reverse) conditions.

Conditions are combined using *or* or *and*:

Or:

```
E1 or E2
```

this compound condition is true when either *E1* or *E2 or both* are true. (This is known as the *inclusive-or* because it is also true when both of its *subconditions* are)[20].

```
3 > 2 or 2 > 3
```

is true.

And:

```
E1 and E2
```

is only true when both *E1* and *E2* are *both* true, so

```
3 > 2 and 2 > 3
```

is false.

Not:

```
not E1
```

is true when *E1* is false and vice-versa, hence

[20]The *or* that we typically use in speech – "you may have apple pie or blueberry pie for dessert" – excludes the possibility of both parts being true, is known as the *exclusive-or* we shall encounter it again.)

```
not 3 > 4
```

is true.

Example:

```
$ more grader1.py
grade = input ()
if grade < 1 or grade > 9:
   print "Invalid grade"
else:
   print "Grade OK"
$ python grader1.py
10
Invalid grade

$ python grader1.py
0
Invalid grade

$ python grader1.py
5
Grade OK
```

3.2.5.12 Iteration (looping)

The other flow of control construct which gives the computer its power is the ability to iterate, i.e. repeat, a series of steps, millions of times if necessary. There are various flavours of iteration found in programming languages[21], so let's take a look at the most commonly found.

While:

This is the most common of all looping constructs, it carries out the specified series of steps as long as (i.e. while) a certain condition is true.

General format:

```
while <condition>:
   <steps-to-repeat>
<step-after-loop>
```

which works as follows:

1. Evaluate <*condition*>
2. If <*condition*> is true then carry out the <*steps-to-repeat*> (known as the *loop body*) and go back to step 1

[21] As we shall see in chapter 11 there *are* languages which do not provide us with iteration. This does not mean that they cannot repeat, simply that they achieve it differently.

3. If <*condition*> is false then skip the loop body and proceed directly to the <*step-after-loop*>

Example

The following code will read in a series of non-negative numbers from the user, add them up and display the total. The user enters a negative number to stop the loop, but this number is *not* included in the total.

```
$ more while1.py
total = 0
number = input ()
while number >= 0:
   total = total + number
   number = input ()
print "total", total

$ python while1.py
10
13
8
99
-1
total 130
```

Repeat:

Some languages also include this construct, which repeats the loop body until the condition becomes true (i.e. while it is false)

General format:

```
repeat until <condition>
   <steps-to-repeat>
<step-after-loop>
```

which works as follows:

1. Evaluate <condition>
2. If <condition> is false then carry out the <*steps-to-repeat*> (i.e. the loop body) and go back to step 1
3. If <condition> is true then skip the loop body and proceed directly to the <*step-after-loop*>

Python does not have a repeat – nor do most languages; however, this is not a problem as we have the following equivalence:

```
while <condition>
```

is *exactly equivalent* to

```
repeat until not <condition>
```

Essentially, then, it does not matter whether you use *while* or *repeat* in your pseudocode.

Most people use *while* as it is more commonly found in other programming languages.

This book will use *while* from now on.

Termination conditions

Let us look at another example of a loop, in order to examine what happens around the termination condition.

This program prints out the squares of integers from a given value upwards:

```
$ more while2.py
counter = input ()
while counter < 10:
   print counter, counter * counter
   counter = counter + 1
```

Now let's try it with some different input values.

We'll start with one which is well below the loop limit

```
$ python while2.py
5
5 25
6 36
7 49
8 64
9 81
```

OK, now our condition is < 10, so let's try the largest value which satisfies that condition, which is 9:

```
$ python while2.py
9
9 81
```

Now let's go 1 over the limit

```
$ python while2.py
10
$
```

Notice that we have no output from the program in this last case.

To see why take another look at how the while works:

1. Evaluate <*condition*>
2. If <*condition*> is true then carry out the <*steps-to-repeat*> and go back to step 1
3. If <*condition*> is false then skip the loop body and proceed directly to the <*step-after-loop*>

Look carefully at the code again: when the input value is 10, then the condition < 10 is false *immediately* and so *the entire loop body is skipped* and we go on to the next step, which in this particular example is the end of the program.

It is often important when testing code to check what happens around the limits of the loop, these are called the *boundary conditions*; in this case the values of 9 and 10 checked these.

Some examples:

1. Averaging grades

   Problem: Compute the average of three grades. A valid grade has a value from 1 to 9 (inclusive). If any grade is 0 or negative, a message "Bad data" is printed

   Pseudocode:

```
input G1, G2 and G3
if G1 < 1 or G2 < 1 or G3 < 1:
   print "Bad data"
else:
   average = (G1+G2+G3)/3
   print average
```

   Which is almost 100% legal python.

   We need to fix the input and also recall that the calculation of the average is unlikely to produce an integer result[22]:

```
$ more average.py
G1, G2, G3 = input ()
if G1 < 1 or G2 < 1 or G3 < 1:
   print "Bad data"
else:
   average = (G1+G2+G3)/3.0
   print average
```

   Now, testing this to make sure we have covered all of the possibilities is trickier than before: at the very least we should test that it works:

[22] Actually, of course, this chance is – assuming randomly chosen data – precisely one in three.

(a) When all three input values are valid

(b) When the first input value is invalid

(c) When the second input value is invalid

(d) When the third input value is invalid

```
$ python average.py
4,7,8
6.33333333333
$ python average.py
-1,4,9
Bad data
$ python average.py
1,0,5
Bad data
$ python average.py
3,5,-5
Bad data
```

2. Summing n integers

Problem: calculate the sum of n integers, where n is a positive integer

Pseudocode:;

```
input n
input I1, I2...In
sum = 0
k = 1
while k <= n:
   sum = sum + Ik
   k = k + 1
print sum
```

This is our trickiest algorithm so far, in terms of converting the pseudocode into python. But don't despair, it isn't that bad.

Python

```
$ more summer.py
n = input ()
I = input ()
sum = 0
k = 0
while k < n:
   sum = sum + I[k]
   k = k + 1
print sum
```

Note that we have used the *input ()* for reading our input values and so the user must enter a list (complete with brackets) for this to work.

```
$ python summer.py
3
[1,2,3]
6
$ python summer.py
5
[5,17,99,14,3]
138
```

This works. However, it is not especially elegant. Try to think of at least two ways in which you could improve things for the user.

3.2.6 Algorithms and problem solving

Remember that if we can produce an algorithm to solve a problem then we can instruct a computing agent to perform that algorithm for us, in other words we can automate that solution.

3.2.6.1 Algorithm discovery

We solve problems by discovering algorithms.

How?

Mainly by practice ("practice makes perfect"); the more problems you solve algorithmically the better you will get at doing it.

But there are a few guidelines and typically problem solving will involve these steps:

1. Understand the problem
2. Divide it into smaller sub-problems
3. Outline the solution and revise it, probably more than once
4. Test for correctness

Clearly you cannot solve a problem unless you understand it (step 1).

Most real-life problems are too complex for us to find a solution to the entire problem at once; breaking it down into smaller problems and building the smaller solutions (sub-algorithms if you like) into the complete solution, is the way to approach these (step 2).

It is unlikely that you will come up with the complete solution the first time so there will be a process of improvement or refinement (step 3).

Testing is important but can be over-rated. The eminent Dutch computer scientist Edsger Dijskstra once remarked that "testing cannot reveal the absence of bugs, only their presence."

In other words, when you run a test on a program and it works, it does not mean that there are no errors in the program, simply that the test data you supplied did not trigger any.

On the other hand, if you run the test and the program fails, then obviously you have found an error.

It is very difficult – if not practically impossible – to test a program of any size exhaustively, i.e. to test every possible path through the program.[23]

We should be convinced that our algorithm is correct; testing will then confirm the correctness and that we have correctly converted the algorithm from pseudocode into our implementation language (python, C++, java, whatever).

3.2.6.2 Some common algorithms

Certain algorithms or types of algorithms occur frequently and it is worth studying them in some detail at this point.

Some common algorithm types are:

1. Comparisons: for example finding the largest value in a list of numbers.
2. Sequential searches: looking through a list of items for a particular value. The search begins at the beginning of the list and proceeds until we find what we are looking for.
3. Binary search: looking through a *sorted* list of items for a particular value. The search treats the list as if it were a *tree*.

3.2.6.3 Comparison

The problem: given a list of values A_0, A_1, ..., A_{n-1}, find the largest value and its (first) location.[24]

Location	A_0	A_1	A_2	A_3	A_4	A_5	A_6
Value	5	2	8	4	8	6	4

Table 3.1: The largest-item-in-a-list problem

The general approach go through the entire list, each time we look at the next item, find the largest-so-far and record its location.

Here is what we shall find at each step:

[23] Imagine, for instance, a program with ten *if* statements, one after the other.

Each *if* means that there are two possible paths through that code. String them one after another and you *multiply* the number of paths.

So, a program with 10 *ifs* will have potentially 2^{10}, i.e. 1,024 paths.

A program with 20 *ifs* will therefore have 2^{20}, which is 1,048,576 paths.

Nobody is going to sit down and write out over a million items of test data.

[24] We might as well acknowledge at this point that computers begin counting not from 1 but from zero – and this fact is reflected in most programming languages.

largest_so_far	location	current position	value
5	**0**	**1**	**5**
5	**0**	**2**	**2**
8	**2**	**3**	**8**
8	**2**	**4**	**4**
8	**2**	**5**	**8**
8	**2**	**6**	**6**
8	**2**	**7**	**4**

Table 3.2: The Progress of the Algorithm

Pseudocode: (first attempt)

```
input A
i = 0
position = -1
largest_so_far = 0
while i < number of A values:
   if A_i > largest_so_far:
      largest_so_far = A_i
      position = i
   i = i + 1
print largest_so_far, position
```

This is not bad, but it does present one rather inelegant feature:

Clearly we have to have *some* initial values for *position* and *largest_so_far*; ideally they should be values that will never occur later in the algorithm.

Setting *position* to -1 is thus OK, as there is no A_{-1}, however can we be as sure that 0 is a good initial value for *largest_so_far*? What if the entire list of numbers is negative?

Moreover, when we begin looking at the list, the *first* number in the list *will* be the largest we have seen to this point, so why not make that part of the algorithm?

This (outlining and revision) leads us to:

Pseudocode: (second attempt)

```
input A
position = 0
largest_so_far = A_0
i = 1
while i < number of A values:
   if A_i > largest_so_far:
      largest_so_far = A_i
      position = i
   i = i + 1
print largest_so_far, position
```

Notice that having already dealt with A_0 we begin examining the rest of the list at position 1.

In fact, at this point we encounter a more-or-less non-ignorable implementation detail: the problem specification refers to A_i where i is the name of a variable (usually referred to – for obvious reasons – as a *loop control variable*) and A_{i} should somehow refer to the ith A.

How can we do this in python (the details will quite probably be different in another language)?

Python lists:

In python we can group a number of items together under the same name as a *list*. Lists are denoted by placing the items inside brackets[25] and separating them by commas. We can then refer to an individual item of a list by its *position within the list* (with the first item being, of course, at position zero).

Here's a simple example:

```
>>> L = [1,3,5,7,12]
>>> print L
[1, 3, 5, 7, 12]
>>> print L[0]
1
>>> print L[4]
12
>>> print L[5]
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
IndexError: list index out of range
>>> pos = 3
>>> print L[pos]
7
>>> print L[pos+1]
12
```

Note that we can refer to an item in the list either by a specific number (position) or by using a variable to hold the position – this is particularly useful inside a loop, as we shall see below.

Note also that the *bounds* (i.e. the limits) of the *index variable* are from zero to one less than the size (number of elements) of the list: in the above example, the valid indices[26] range from 0 to 4, as there are 5 elements in the list.

[25] Strictly speaking the three kinds of brackets found on the keyboard are known as:

- Parentheses (aka "round brackets") – ( )
- Brackets (aka "square brackets") – [ ]
- Braces (aka "curly brackets") – { }

[26] The plural of index.

Now we know how to deal with lists – and, as we saw in passing earlier, *input()* will read entire lists as long as the user types them in python list format, i.e. elements separated by commas inside (square) brackets.

This python program also has *comments* on each line – everything after the # is a comment.

Commenting programs is an art and is one which should be acquired by everyone writing programs.

Despite what some people might say, *there is no such thing as a self-documenting program.*

The algorithm:

```
$ more largest.py
A = input ()                  # get the list to search
largest = A[0]                # first entry is largest -- so far
location = 0                  # and this is where it is found
i = 1                         # search rest of list
while i < len (A):            # while more to do
   if A[i] > largest_so_far: # a new largest value
      largest = A[i]          # record it and
      location = i            # where it was found
   i = i + 1                  # next position to examine
print "largest:", largest, "location:", location

$ python largest.py
[5,2,8,4,8,6,4]
largest: 8 location: 2
```

Sequential search:

Problem: Find the address of a given name in an (unsorted) list of names and their addresses. We shall have a sequence of names: N_0, N_1, etc. and their corresponding addresses, A_0, A_1 etc. (Note that we shall from this point abandon using subscripts – e.g. A_0 – as no normal programming language has this ability)

So, do we understand the problem? We'll be given (somehow) the names and addresses and a name to search for. When we find the *N[n]* which matches the name the corresponding *A[n]* is the address we require.

First attempt:

```
input N[0], N[1],...N[n]
input A[0], A[1],...A[n]
input name
if N[0] == name:
   print A[0]
if N[1] == name:
```

```
      print A[1]
   if N[2] == name:
      print A[2]
   ......
   else:
      print "Name not found"
```

The use of all those *ifs* might seem a bit problematic: suppose that we have 100 name/number pairs and the one we are searching for is number 5; the algorithm outline above will still check items 6 through 99, which is hardly efficient.

Python (and many other languages) allow for what is called an "else if" for mutually exclusive conditions:

```
if <condition_1>:
   <exec_part_1>
else if <condition_2>:
   <exec_part_2>
else if <condition_3>:
   <exec_part_3>
........
else
   <else_part>
```

which, in pyth on, can be expressed as:

```
if <condition_1>:
   <exec_part_1>
elif <condition_2>:
   <exec_part_2>
elif <condition_3>:
   <exec_part_3>
........
else
   <else_part>
```

The important part about this construct is that once we have established which condition is true we do not waste time or effort (or risk errors) by checking the rest of the conditions.[27]

But even using the *elif* we still end up with a fairly hopeless algorithm:

```
input N[0], N[1],...N[n]
input A[0], A[1],...A[n]
input name
if N[0] == name:
   print A[0]
elif N[1] == name:
```

[27] Other languages typically spell "else if" as *elseif* or *elsif*.

```
      print A[1]
   elif N[2] == name:
      print A[2]
   ......
   else:
      print "Name not found"
```

The problem is that every time we add a new name/address pair we shall have to change the algorithm – even it is only to add one more if/print. And unless there is a preposterously small number of actual names and addresses – this might work for the Tuktoyaktuk voters' list, but hardly for Vancouver – this algorithm is unlikely to be very efficient.

Let us try and protect ourself against future changes (i.e. additions to the phone book). We can do this by noticing that there is a definite *pattern* to the algorithm above: look at every *elif*: apart from the position number, each one is exactly the same:

```
elif N[position] == name:
   print A[position]
```

And when we have a lot of nearly identical pieces of code with but a single change between them, we should start to think "loop".

Second attempt: using a loop

```
input N1,N2,...Nn
input A1, A2,...An
input name
Found = "No"
while Found == "No" and i <= 1000
   if Name == Ni:
      print Ai
      Found = "Yes"
   else:
      add 1 to i
```

What do we notice about this new version of the algorithm?

Perhaps the key is to notice that while we are proceeding from entry to entry in the address book, there are two conditions which can make us stop looking:

1. We find the entry we're looking for, or
2. We get to the end of the book without finding it; this is why our while has a compound condition, which effectively says: keep looping (looking) as long as you *haven't* the name you are looking for and as long as there are *more* entries to look at.

If we *either* find the correct entry *or* hit the end of the book, then we should stop looking.

Here's a python solution. It is unrealistic because I have built the actual address book into the program. Deciding how to store the address book (e.g. on disk) and read it into the program are topics for a later day.

```
$ more addressbook.py
N = ['bill', 'amanda', 'arthur', 'jane',
     'ron', 'kathy', 'tom', 'gillian']
A = ['1223 Any Street', '666 Revelation Avenue',
     '932 Main Street', '3472 Baseline',
     '17 Government Street', '4432 Douglas Street',
     '4461 Interurban Road', '3327 Old Island Highway']
name = raw_input ('Enter name to look for: ')
                            # get name to search for
found = 'No'                # certainly not found it yet
i = 0                       # start from beginning of book
while found == 'No' and i < len (N):
                    # loop until we find name or run out of book
   if N[i] == name:         # this is the one!
      print A[i]            # print address
      found = 'Yes'         # remember we found it
   else:                    # not this one
      i = i + 1             # look at next entry

if found == 'No':           # name not in book
   print 'Name not found'   # tell user
```

What would be good test names to enter?

How about: a name from the middle of the book, the first name, the last name and a name which isn't in the book?

```
$ python addressbook.py
Enter name to look for: arthur
932 Main Street

$ python addressbook.py
Enter name to look for: gillian
3327 Old Island Highway

$ python addressbook.py
Enter name to look for: bill
1223 Any Street

$ python addressbook.py
Enter name to look for: zak
Name not found
```

Chapter 4

Efficiency

> "Well, in our country," said Alice, still panting a little, "you'd generally get to somewhere else – if you run very fast for a long time, as we've been doing."
>
> "A slow sort of country!" said the Queen. "Now, here, you see, it takes all the running you can do, to keep in the same place. If you want to get somewhere else, you must run at least twice as fast as that!"

Lewis Carroll, *Through the Looking Glass and What Alice Found There*

4.1 Algorithm attributes

We can consider several attributes which algorithms should possess.

1. Correctness. In other words, will the algorithm give the correct solution to the problem?
2. Ease of understanding. How easy is it for readers to understand the algorithm? This includes those who wrote the algorithm in the first place and others. The reason is that all real-life programs are subject to maintenance, by which is meant fixing bugs and/or adding new features. The easier the program (algorithm) is to understand the easier it can be maintained. Maintenance is a major aspect of real-life Information Technology.
3. Elegance Elegance is hard to define, but we usually know it when we see it.
4. Efficiency Obviously we should like our algorithms to be as efficient as possible, but what does this actually mean? We can consider algorithm efficiency in two dimensions:
   (a) Space: How much memory is needed?

(b) Time: How long does it take to solve the problem?

Generally there are also two approaches to this:

(a) Benchmarking

Run the program and measure how long it takes. The problem here is that you are not necessarily measuring the performance of the algorithm, but of the system it is running on

(b) Analysis

This does not involve running the program, but analysing the algorithm by inspection. A considerable body of theory has been developed, which we shall only have time to touch on in this course.

4.2 Choosing algorithms

We shall often find that there are two or more different algorithms that we can come up with to solve the same problem. Which one, then, should be choose?

Obviously: the best!

Ah, but which is the best and what criteria are you using?

1. Efficiency

   Do you mean space or time efficiency? We shall often (in fact usually) find that there is a tradeoff between the two: a faster algorithm tends to require more memory than a slower one.

2. Maintainability

   All programs of any real use will, at some point in their careers, need to be maintained. Anyone telling you differently is a fool or a liar. Given the choice between an efficient, but almost incomprehensible algorithm and a less efficient, but easy to understand one, which is better?

In this chapter we are not interested in maintainability per se. We shall be looking at space and time efficiency. Although we have a pretty good idea of what space efficiency means, as has already been mentioned, time efficiency cannot simply be measured by a stopwatch. Instead, we shall measure (or count, or perhaps even estimate) the number of steps the program (algorithm) performs.

4.2.1 A sample problem[1] c

Consider this problem: your manager has decided that the company is wasting too much blank paper when sending memos, etc.

The solution? Remove the spaces from messages[2].

[1] Several of the problems in this chapter are based on those in Schneider and Gerstin.

[2] You may laugh; stranger things have been proposed.

Here, diagrammatically, is what we need to do:

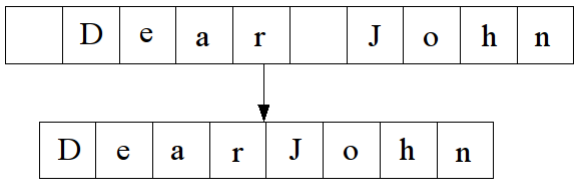


Figure 4.1: Squeezing out the spaces.

We shall examine three different algorithms for doing this:

1. The “shuffle left" algorithm: move everything to the right of a space one position to the left.
2. The “copy over" algorithm copy everything except the spaces into a new list.
3. The “converging pointers” algorithm has two “pointers” approach each other, one from left, one from right, replacing spaces on the left with values on the right.

4.2.1.1 Shuffle-left

Firstly we shall describe the overall strategy in English, then develop pythonic pseudocode.[3]

Strategy Scan the list from left to right, each time a space is encountered, the entire rest of the list – i.e. everything to the right of the space – is "shuffled" one position to the left. Keep doing this as long as there are more spaces to process.

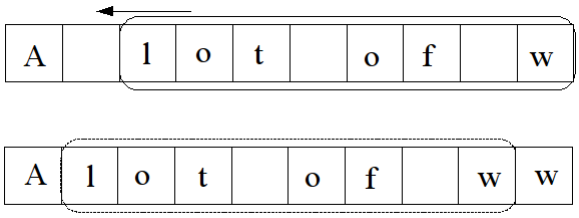


Figure 4.2: A single “shuffle”

Here you can see the first stage of the algorithm (the “shuffle”) in operation. The outlined portion of the list is the part that is being shuffled. Note that we

[3] All of the python in this book has been tested by the author.

are "trailing" a copy of the last element of the list behind every time we shuffle, so we need to keep track of how many elements are left in the list.

Let us now try to develop the pseudocode for this algorithm, remembering the steps we encountered in the previous chapter.

1. Do we understand the problem?
2. Can we divide it into smaller problem?
3. Sketch and refine the solution.
4. Test.

Let us assume that we do understand the problem.

Here is a first attempt at the pseudocode...well python. The problem, though, is rather more complex than most of us can solve in one go, so let us make an assumption: *we know how to shuffle the entire list one place to the left:*

```
L = input ()                 # read in the list
position = 0
while position < len (L):    # work up the list until no more
   if L[position] == ' ':      # a space, lose it
      SHUFFLE
   position = position + 1   # next position in list
```

As long as the rest of the algorithm is correct, we can now proceed to solve the shuffling sub-problem.

Sub-problem: given the position in a list of an entry, shuffle every entry to the right one place to the left.

So, we start at position *pos* we'll need something like this (travel up the rest of the list, moving each item one place to the left):

```
while pos < len (L):
   L[pos] = L[pos+1]
   pos = pos + 1
```

which is *almost* right.

But when we come to the last entry in the list, there is nothing to the right of it to shuffle, so we should actually stop shuffling one place earlier:

```
while pos < len (L)-1:
   L[pos] = L[pos+1]
   pos = pos + 1
```

Apart from anything else, this shows that the $<$ condition allows us to use an *expression* (*len (L) - 1*) in the test.

So we now have code to do a *single* shuffle – even though it has involved a loop, the lines of code above must be used *repeatedly* in order to complete the algorithm: notice that the SHUFFLE above is *inside* a while loop.

Let's now build the sub-algorithm into the whole algorithm and add some comments as we go:[4]

```
L = input ()                  # read in list
position = 0                  # start at beginning of list
while position < len (L):     # until we reach the end
   if L[position] == ' ':       # a space to replace
      pos = position          # start shuffling here
      while pos < len (L)-1:  # shuffle rest of list
         L[pos] = L[pos+1]    # move one place left
         pos = pos + 1        # move up sublist
   position = position + 1    # move up list
print L                       # print result[5]
```

This is good, but not perfect.

Let's see what happens when we try it out.

```
$ python cleanup1-0.py[6]
['h','e',' ','s','a','i','d']
['h', 'e', 's', 'a', 'i', 'd', 'd']
```

Close, but no cigar! We are printing a "trailed" 'd'. We forgot that the list effectively gets one entry shorter each time we shuffle.

Let's take that into account in our algorithm.

```
L = input ()                  # read in list
position = 0                  # start at beginning of list
size = len (L)                # items in list to begin
while position < len (L):     # until we reach the end
   if L[position] == ' ':       # a space to replace
      pos = position          # start shuffling here
      while pos < len (L)-1:  # shuffle rest of list
         L[pos] = L[pos+1]    # move one place left
```

[4]The addition of the comments is also useful in aiding maintainability, because (repeat after me)

There is no such thing as a self-documenting programming language.

There is no such thing as a self-documenting programming language.

There is no such thing as a self-documenting programming language.

There is no such thing as a self-documenting programming language.

[5]If we don't actually print the result of all that hard work, how will we know that it has worked? (Or, indeed, done anything at all)

[6]This may strike you as a rather poor interface, insisting that a string be typed in as a list of single, quoted, characters. You might think that we could use **raw_input** instead and simply type in the string. Unfortunately, a text string is what python considers an *immutable* value, i.e. one that cannot be changed under any circumstances.

There are other ways of manipulating strings.

```
            pos = pos + 1        # move up sublist
          size = size - 1        # list  is now 1 shorter
       position = position + 1   # move up list
    print L
```

And let's try the revised algorithm

```
$ python cleanup1-1.py
['h','e',' ','s','a','i','d']
['h', 'e', 's', 'a', 'i', 'd', 'd']
```

Which is exactly the same! What is going on here?

Well, didn't it strike you as a bit pointless to do those calculations with the size of the remaining list (i.e. after we've squeezed out a space or two) and yet nowhere do we seem to do anything with that information.

Somehow we need to print out just the first *size* items in the list.

We can certainly do this with a loop, but fortunately python allows us a shorthand, called a *slice*.

In the case of a list it works like this:[7]

```
listname[first:last]
```

is a sublist of *listname* consisting of elements *first* up to but *not* including element number *last*. So

```
print L[5:9]
```

would print the elements 5 through 8 of the list L (and always remember that the first element is number 0).

Python allows you to omit the *first* and/or the *last* value between the brackets.

- If you omit *first* the sublist starts with the beginning of the entire list.
- If you omit *last* the sublist ends with the end of the entire list
- If you omit both, you get a copy of the complete list, so

```
print L[3:9]
print L[:7]
print L[5:]
print L[:]
```

These four examples will print:

1. the list from element number 3 to 8 inclusive.

[7] As with many features of python, we don't have room to go into all the details, just the highlights. Any good python text, or the online documentation at http://www.python.org, will fill in the blanks.

2. the list from beginning to element number 6.
3. the list from element number 5 to the end.
4. the entire list.

Furthermore, we can use the name of a variable within our slice, which is exactly what we need to finish cleanup1-1.py:

```
L = input ()                  # read in list
position = 0                  # start at beginning of list
size = len (L)                # items in list to begin
while position < len (L):     # until we reach the end
   if L[position] == ' ':       # a space to replace
      pos = position          # start shuffling here
      while pos < len (L)-1:  # shuffle rest of list
         L[pos] = L[pos+1]    # move one place left
         pos = pos + 1        # move up sublist
      size = size - 1         # list  is now 1 shorter
   position = position + 1    # move up list
print L[:size]
```

The only change is the last line.

We have used the variable *size* to keep track of how many (non-space) items remain in the list – and they will all have been shuffled to the front. So we need to print from the beginning of the list (item zero) up to *but not including* item number *size* (think: if there are six items left, they will be at positions 0, 1, 2, 3, 4 and 5).

This is precisely the "slice" *[:size]* and so we amend the last line of the algorithm accordingly.

As one final refinement, let us put the shuffle into a separate *function*, which will make the entire program easier to read (and therefore improve its maintainability); it will certainly make our main loop easier to read.

But what is a function?

Well, in our original sketch of the algorithm, at one point we simply wrote

SHUFFLE

because we knew that this was the correct place to do the shuffling and, if only python included a "primitive" called SHUFFLE we'd be finished.

Writing a function (aka a *procedure* or *subroutine*) is, if you like, a way of expanding python's basic repertoire.

In this case, there are two pieces of information that we need to give ("pass") to the function called *shuffle* (we can choose our own names, this seems as good as any):

1. which list is to be shuffled
2. where in the list we start shuffling

These values that we pass to the function are called *parameters* and, in general, make the function more useful: imagine doing without one of them, say the position. That reduced function would now be able to shuffle *any* list you told it to, but it would *always* begin shuffling at the same position (zero?) because there would be no way to tell it any different.

A function starts with the "word" *def* then has a list of parameters enclosed by parentheses and finally a colon (:).

The *body* (the code) of the function must all be indented to the right; anything which aligns with the 'd' of *def* ends the function. (Similar to the layout rules for *if* and while*)*.

Here is the complete version of the algorithm, in python, with a *shuffle* function and decent comments.

Note that you must define the function before you can use it.

```
$ more cleanup1-2.py
#
# The shuffle-left version of the data cleanup algorithm
#
#
# shuffle moves every element in the list L, beginning at
# position pos, one place to the left.
#
def shuffle (L, pos):        # shuffle list 1 position left
   while pos < len (L) -1:   # until the end of the list
      L[pos] = L[pos+1]      # move 1 entry left
      pos = pos + 1          # go to next entry
#
# main code
#
L = input ()                 # read in list
position = 0                 # start at beginning
size = len (L)               # current count of valid entries
while position < size:       # while there's more to do
   if L[position] == ' ':      # found a space
      shuffle (L, position, size)  # get rid of it
      size = size - 1        # 1 fewer valid entries
   position = position + 1   # move along 1 position
print L[:size]               # print valid portion

$ python cleanup1-2.py
['h','e',' ','s','a','i','d']
['h', 'e', 's', 'a', 'i', 'd']
```

Incidentally, there is still (at least) one bug in the above algorithm/program. Can you identify it? (Hint: consider a list with two or more spaces in succession)

4.2.1.2 "Copy over"

Strategy we employ two lists; the new one begins empty and we traverse the original list, copying the non-space entries into the new list. When we've finished traversing the original list, print the new list.

Again, let's start with an initial attempt:

```
L1 = input ()
L2 = empty list same size....we'll come back to this
inpos = 0
output = 0
while inpos < len (L1):
   if L1[inpos] != ' ':
      L2[output] = L1[inpos]
      outpos = outpos + 1
   inpos = inpos + 1
print L2
```

Again there are a few things that we need to refine. Notice that we need to keep track of our position in *two* lists – hence *inpos* and *outpos* – also that we somehow need to create the second list as the same size as the first (there may, after all, be no space entries, in which case everything should be copied across).

We should also keep track of how many entries we have put into the second list.

Hence:

```
#
# The copy-over version of the data cleanup algorithm
#

L1 = input ()
L2 = [' '] * len (L1)          # make new list same size
inpos = 0                    # position to start copying from
outpos = 0                   # position to start copying to
count = 0                    # new list size
while inpos < len (L1):      # iterate up input list
   if L1[inpos] != ' ':        # need to copy this one
      L2[outpos] = L1[inpos] # copy it
      count = count + 1      # count it
      outpos = outpos + 1    # move along new list
   inpos = inpos + 1         # move along old list
print L2[:count]             # print valid entries
```

Note that we have used the python multiply on an empty list. You may recall that:

```
print "Hello" * 2
```

resulted in

```
HelloHello
```

As python considers multiplying a string as meaning creating a string from that number of copies of the original.

Similarly we can "multiply" a list:

```
print [1,2,3] * 2
[1, 2, 3, 1, 2, 3]
```

The result list is a list consisting of (in this case) two copies of the original. So in our cleanup2-1.py we have the line:

```
L2 = [' '] * len (L1)
```

which breaks down as follows:

- get the length of the list L1
- make a list of that many spaces (i.e. that many copies of a list containing a single space) and call it L2

Clearly this new list will be the same size as the original list and never any shorter – even if the original list consists mostly (or even entirely) of spaces.

A more "pythonic" way of writing this, which would give us a new list of *exactly* the right length to contain just the non-space entries from the original list, would be to initialise the output list as empty ([]) and using the append () list *method* to add new entries. This means that *outpos* is not required and the entire second list can be printed at the end.

This refinement is left as an exercise, but the key line, inside the *if* would read:

```
L2.append (L1[inpos])   # add inpos element of L1 to end of L2
```

4.2.1.3 "Converging pointers"

Strategy scan the list from both directions simultaneously. Whenever we encounter a space at the left position we move the entry at the right position over it.

A few thoughts: obviously (this should be obvious by this time) we are going to use a loop, but the loop condition is likely to be different: do we really expect to get the left pointer all the way to the end of the list? We should stop when the pointers overlap.

So, how about this?

```
L = input ()
left = 0
right = len (L) - 1
count = 0
while left <= right:
   if L[left] == ' ':
      L[left] = L[right]
      right = right - 1
   count = count + 1
   left = left + 1
print L[:count]
```

This looks reasonable, let's try it.

```
$ python cleanup3-0.py
['h','e',' ','s','a','i',' ','d']
['h', 'e', ' ', 's', 'a', 'i', ' ']
```

Hmmm, que pasa? Try to figure out what is going on here before you read the next paragraph.

One thing that we did not take into account: what if the entry that we move from right to left is itself blank?

Our algorithm does not take this possibility into account and so we might (as in the example above) end up with a moved space in the resulting list.

We could fix this in two ways:

1. When we copy from right to left, do not increase the left pointer or the valid count, so that if a space was copied it will simply be overwritten next time through the loop and will not have been counted.

   This is an easy change, so we'll try it first:

   ```
   L = input ()
   left = 0
   right = len (L) - 1
   count = 0
   while left <= right:
      if L[left] == ' ':
         L[left] = L[right]
         right = right - 1
      else:
         count = count + 1
      left = left + 1
   print L[:count]

   $ python cleanup3-1.py
   ['h','e',' ','s','a','i',' ','d']
   ['h', 'e', 'd', 's', 'a', 'i']
   ```

2. Test the right entry before we copy it; if it's space don't copy it but skip it.

```
L = input ()
left = 0
right = len (L) - 1
count = 0
while left <= right:
   if L[left] == ' ':
      if L[right] != ' ':
         L[left] = L[right]
         right = right - 1
      count = count + 1
      left = left + 1
print L[:count]

$ python cleanup3-2.py
[' ','d','g','r','b',' ','h','e',' ','!']
['!', 'd', 'g', 'r', 'b', ' ', 'h', 'e']
```

Again, close but no cigar.

Look very carefully at the loop body. In our original attempt every time through the loop increased the size of the resulting list because we either moved an entry from the right or we kept a non-space entry at the left.

In our "improved" version we increase the result list size even when the right entry is space and hence not moved.

OK....let's make sure that we only increase the count of legitimate entries when either we do move a right entry (because it is not space) or when the left entry is non-space.

```
L = input ()
left = 0
right = len (L) - 1
count = 0
while left <= right:
   if L[left] == ' ':
      if L[right] != ' ':
         L[left] = L[right]
         count = count + 1
         left = left + 1
      right = right - 1
   else:
      count = count + 1
      left = left + 1
print L[:count]
```

```
$ python cleanup3-3.py
[' ','d','g','r','b',' ','h','e',' ','!']
['!', 'd', 'g', 'r', 'b', 'e', 'h']
```

This is now rather more complex than the solution we got earlier, when we allowed spaces to be moved and then overwritten.

Which of our algorithms is the best?

4.2.1.4 Comparison of the data cleanup algorithms

We now have the interesting situation that we have three different algorithms, all of which solve the problem posed.

You have probably noticed that the final, converging pointers algorithm leaves the characters in a different order. The problem specification said nothing about this[8]. If it had it would have restricted our choice of algorithm.

Algorithm	Space efficiency	Time efficiency
shuffle left	no extra space	requires many comparisons
copy over	space required for new list	goes through list once only
converging pointers	no extra space	goes through list once only

Exercise Can you come up with a more efficient algorithm for the data-cleanup problem, that:

- does not require any additional space
- does less copying than shuffle left
- maintains the order of the non-space elements

Hint: can you modify the second (copy over) algorithm so that it does not require the second list but copies into the same list?

4.2.2 Time efficiency

But how do we measure time efficiency?

- With a stopwatch?

  If we run algorithm A on machine a and algorithm B on machine b what, if anything, does this tell us about the relative efficiencies of the two algorithms? Even if we run them both on the same machine, we are still possibly measuring things we are not really interested in – e.g. the current load on the machine. We cannot be certain that we are only measuring our algorithm.

[8] But then we already suspected the specification was "a tale told by an idiot, full of sound and fury, signifying nothing".

- How many steps does the algorithm execute?[9]

  This is a better metric (standard of measurement), but it is a lot of work to count all those steps.

- How many *fundamental* steps does the algorithm execute?

  Now all we have to do is define and identify the fundamental steps. This is not as hard as you might imagine.

Performance of our algorithm will depend on the size (or number) and type of the input. In particular we are interested in knowing, for our algorithm, the best- worst- and average-case performances.

The bottom line: we need to analyse the algorithm itself.

4.2.2.1 An example: sequential search.

Let us revisit the address book search algorithm from the last chapter.

Here is the heart of the algorithm (omitting reading in the address book) with step numbers added:

```
 1) name = raw_input ()
 2) found = 'No'
 3) i = 0
 4) while found == 'No' and i < len (N):
 5)    if N[i] == name:
 6)       print A[i]
 7)       found = 'Yes'
 8)    else:
 9)       i = i + 1
10) if found == 'No':
11)    print 'Name not found'
```

Let us see how often (in terms of the 'input' size – i.e. the number of entries in the address book) each step will be performed:

- Steps 1, 2, 3, 6, 7, 10 and 11 will be performed (at most) once each
- Steps 4, 5 (8) and 9 depend on input size:
  - Worst case:
    - Steps 4, 5 and 9 are executed at most n-times[10].
  - Best case:

[9] And what, precisely, do we mean by a "step"? One machine instruction? They will be *very* small steps. Or perhaps one language statement? Now we are depending on the language used to implement the algorithm. We must somehow define "steps" in a non-language-specific fashion.

[10] To be absolutely accurate, the comparison – step 4 – a maximum of n+1-times: n times through the loop and once for the test to result in False and stop the loop.

    - Steps 4 and 5 are executed only once.
- Average case:
    - Steps 4 and 5 are executed approximately (n/2)-times.

Clearly the number of times we perform step 5 (and 4, but 5 is simpler) depends precisely on whether we have the best, worst or average case.

In conclusion: we can use the name comparison (step 5) as a fundamental unit of work!

4.2.2.2 Order of Magnitude ("Big O" notation)

We are not interested in knowing the exact number of steps the algorithm performs, instead we are interested in knowing how the number of steps grows with increased input size.

Why? Because, *given large enough input* (i.e. large enough number of input items), the algorithm with faster growth will always execute more steps.

Order of magnitude, notated as $\mathbf{O}(...)$, measures how the number of steps grows with input size n.

4.2.2.3 Linear algorithms

As illustration of the fact that we are not interested in the exact number of steps, consider example algorithms which would, for input size n, perform the following number of steps:

1. n
2. 5n
3. 5n+345
4. 4500n+1000

We are not interested in the coefficient of n or of the constant addend: all of these algorithms as classified as $\mathbf{O}(n)$, because – when n is *large enough* – the number of steps performed grows proportionally with n. So when you double the input size, you (more or less) double the number of steps.

Obviously with, say, algorithms 3 and 4, this is not true for small values of n. But when n is *large enough* (that's the last time – you must supply the phrase mentally from now on) it is true – when n is 100000000 for example.

Algorithms which are $\mathbf{O}(n)$ are known as linear algorithms because a graph of their performances (expressed as steps performed vs. input size) is always a straight line.

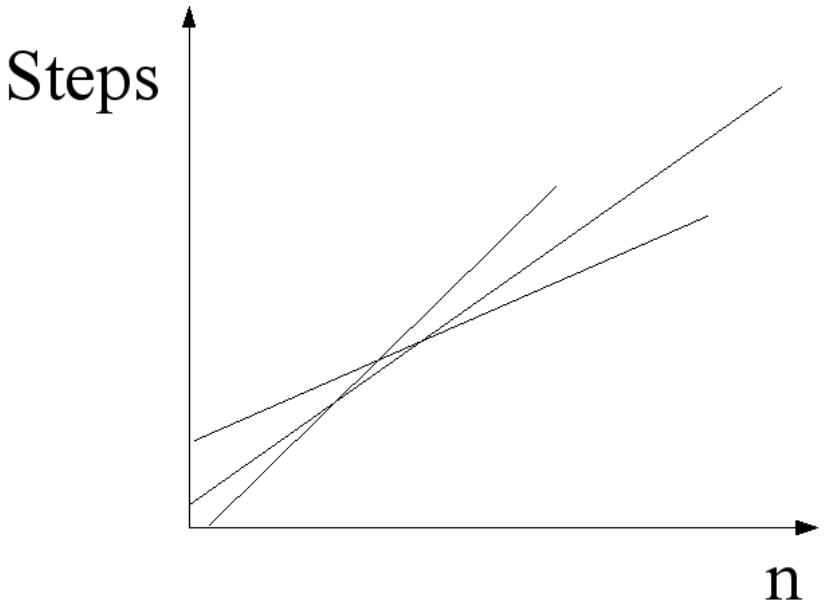


Figure 4.3: Typical linear algorithm growth

Linear algorithms are highly desirable.
Unfortunately they are also fairly rare.

4.2.2.4 Non-linear algorithms

Consider printing out the n-times multiplication table:

1	2	3	...	n
2	4	6	...	2n
3	6	9	...	3n
...	...	...	...	...
n	2n	3n	...	n^2

Here is code which will do just that:[11]

```
$ more mul.py
```

[11] It also introduces another couple of python "featurettes":

1. If the *print* statement ends with a comma, the next print will print *on the same line*, this is required within the inner loop, otherwise our 9-times table will come out on 81 separate lines.

2. If the *print* statement ends without a comma, *even if there is nothing after the word print*, the *next* print will print *on the next line.*

```
N = input ('Enter size ')
row = 1
while row <= N: # print one row
   col = 1
   while col <= N: # print one column position
      print row * col,
      col = col + 1
   print
   row = row + 1
```

```
$ python mul.py
Enter size
9
1 2 3 4 5 6 7 8 9
2 4 6 8 10 12 14 16 18
3 6 9 12 15 18 21 24 27
4 8 12 16 20 24 28 32 36
5 10 15 20 25 30 35 40 45
6 12 18 24 30 36 42 48 54
7 14 21 28 35 42 49 56 63
8 16 24 32 40 48 56 64 72
9 18 27 36 45 54 63 72 81
```

It should be fairly self-evident that the amount of work this program does (let's take the multiplication as the 'fundamental step') grows with the square of the input size (N): after all, the program is going to print N rows, each with N columns (hence N^2 columns in all) and each column requires a multiplication to calculate the number to be printed, so we can use the multiplication as our fundamental step.

This algorithm, then, is not linear, it is quadratic.

It is $\mathbf{O}(n^2)$

4.2.2.5 Calculating Order of Magnitude

But how, given any algorithm, can we calculate its order of magnitude?

They are not all as obvious as the two we've just examined, but there are a few simple rules:

1. Consecutive (i.e. sequential) statements:

   Simply add the number of steps in each statement

2. If-then-else:

   The total number of steps is the number in the condition test plus the number of steps in either the if- or the then- part, whichever is greater.

3. Loops:

To calculate the number of steps involved in a loop we multiply *test* by *iterations* (the number of steps in the condition multiplied by the number of times it is tested) and add the number of steps in the loop body multiplied by the number of iterations through the loop (the number of tests performed at the top of the loop should be one more).

4. Nested Loops:

   To calculate the total number of steps we multiply the number of inner iterations by the number of outer iterations.

4.2.2.6 A simple example: summing cubes.

This examples introduces two more bits of python:

1. The concept of a *returned value* from a function. Think of this as the answer to a question. Our *sumcubes* function is going, given a parameter value N, to calculate the sum of the cubes from 1^3 to N^3 and give us back the result.

   Using our *sumcubes* function, then, is like asking the question:

   > What is the sum of all the cubes from 1 up to this number N?

   So, the last line of our script below, *print sumcubes(N)* , is equivalent to:

   > Print the sum of the sum of the cubes from 1 to N.[12]

```
$ more sumcubes.py
def sumcubes (n):
   sum = 0
   i = 1
   while i <= n:
      sum = sum + i * i * i
      i += 1
   return sum

N = input ('Enter N: ')
print 'the sum of cubes from 1 to', N, 'is',
print sumcubes(N)

$ python sumcubes.py
Enter N: 1
the sum of the cubes from 1 to 1 is 1

$ python sumcubes.py
```

[12] You can also think of it in this way: to print the number I have to know the number, so I *call* (ask) *sumcubes(N)* and it *gives me the answer* (this is the purpose of the *return* statement), which I can then print.

```
Enter N: 2 the sum of the cubes from 1 to 2 is 9

$ python sumcubes.py
Enter N: 3 the sum of the cubes from 1 to 3 is 36

$ python sumcubes.py
Enter N: 4 the sum of the cubes from 1 to 4 is 100
```

Analysing the function sumcubes we find

```
def sumcubes (n):
   sum = 0                  # 1 step
   i = 1                    # 1 step
   while i <= n:            # (n+1) * 1 steps
      sum = sum + i * i * i # 4n steps
      i += 1                # 2n steps
   return sum               # 1 step
```

The 1 step statements are obvious; the loop needs a little more explanation:

The test ($i <= n$) is performed once for each pass through the loop plus once more to establish that it is now false (i.e. when i becomes greater than n). Each test is 1 step.

The assignment to *sum* actually involves:

1. two multiplications
2. an addition and
3. an assignment

And is performed every time we go through the loop – n times in all.

The incrementing of i also involves an addition and an assignment and is also done n times in all. The total amount of work done, therefore, is:

`1 + 1 + (n+1) + 4n + 2n + 1 = 7n + 4 =` **O**`(n)`

4.2.2.7 Analysis of the three data cleanup algorithms

As might be expected, the three data cleanup algorithms have differing efficiencies.

Firstly, let us consider what will constitute the best, worst and average cases:

1. Shuffle left
   - The best case is easy (by best case we mean the one involving least work): there are no spaces in the list and therefore nothing to shuffle. Our fundamental unit of work will be the comparison against the length of the list. We'll pass once through the list (i.e. n compares) and so do the comparison *n+1* times. In other words the algorithm, for this case, is **O**(n).

- There is only one list, hence the space requirement is directly proportional to the number of items.
- The worst case will be a list in which there is only one non-space entry, at the end. We'll end up doing the shuffle n-1 times.
- When we are shuffling into position 0, we'll need to do n-1 compares, to shuffle into position 1 n-2, etc.

  The sum is

  `1 + 2 + ...n = n(n+1)/2`
  `and our algorithm requires` *`(n-1)*n/2`* `steps, which is`
  $(n^2-n)/2$ `which is` $O(n^2)$.

- The average case will involve some proportion of the entries being space. In which case we will only need to do proportionally many shuffles.

  Let's say that 1 in 100 numbers are space on average, then we'll only have to do 1/100 of the shuffling.

  But if that results in, say (and this is only an approximation) *n*(n-1)/200* steps, the expression $n^2/200$ - n/200 is still $\mathbf{O}$(n2)
- As none of these cases requires any more space than the original list, whose size is directly proportional to the number of entries, all three cases have space requirements of *n*.

2. Copy over

   - With this algorithm the cases are not the same as with the shuffle left (and why should they be?)
   - The best case is when the list is all spaces and there is nothing to copy. We'll just go through the list once, doing *n+1* comparisons.

     No extra space is required.
   - The worst case is when the list is completely non-space and we must copy every item to the new list. A new list is required which will be the same size as the original.
   - The average case will only require us to move a certain proportion of the original list. We shall therefore only require enough new space to take the non-space entries.
   - In each case we must go through the list once, doing *n+1* comparisons, hence this algorithm is $\mathbf{O}$(n) for each case.
   - But the space requirements (assuming that the new list grows each time we copy something onto the end) are completely dependent on the input data.

3. Converging pointers

   It is harder to state exactly what the best, worst and average cases are for the converging pointers algorithm.

- The best case will be when there are no space entries; each pass through the loop will move the left and right pointers, so we'll do approximately *n/2* compares.
- The worst case will be when every item in the list is space. Each pass through the loop will adjust the right but not the left pointer. So we'll do approximately *n* compares.
- The average case is obviously somewhere in between, i.e. the number of comparisons will be between *n* and *n/2*.
- All three expressions are $\mathbf{O}(n)$ and, as there is only one list, our space requirements are always the same.

	Shuffle left		Copy over		Converging pointers	
	Time	Space	Time	Space	Time	Space
Best case	$\mathbf{O}(n)$	n	$\mathbf{O}(n)$	n	$\mathbf{O}(n)$	n
Worst case	$\mathbf{O}(n^2)$	n	$\mathbf{O}(n)$	2n	$\mathbf{O}(n)$	n
Average case	$\mathbf{O}(n^2)$	n	$\mathbf{O}(n)$	n<=x<= 2n	$\mathbf{O}(n)$	n

4.2.2.8 Sorting

Sorting is a very common task, for example: sorting a list of names into alphabetical order, or sorting numbers into numerical order.

It is obviously very important to find efficient algorithms for sorting and we shall look at several.

All of these, incidentally, are known as *internal* sorting algorithms as the entire data set being sorted can be held in the computer's memory simultaneously.

When this is not possible, then *external* sort algorithms come into their own. These are far more complex and well beyond the scope of this course.

Some common sorting algorithms are:

- Bubble sort
- Selection sort
- Quick sort
- Heap sort

We will analyse the complexity of the bubble and selection sorts.

4.2.2.9 Bubble sort

This is probably the most intuitively obvious sort strategy. Unfortunately, it is not very efficient.

The basic strategy

Move along the list, one position at a time, checking each adjacent pair and exchanging them if the left one is greater than the right one. Keep doing this until a sweep through the entire list does no swaps.

Here is how it works, our initial list being: 4, 6, 9, 2, 5

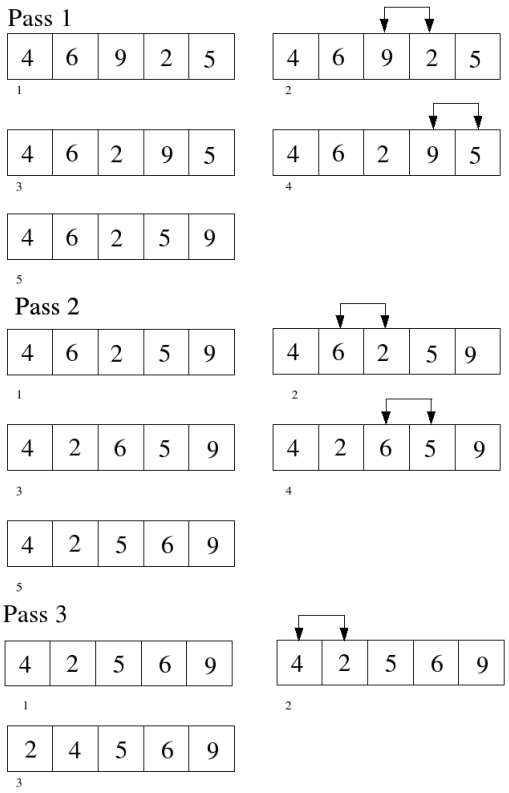


Figure 4.4: The bubble sort in action

The first pass through the list swaps the 9 and the 2, then the 9 and the 5. You should now begin to see where this algorithm gets its name: the 9 has "bubbled" up the list into its proper place.

As the first pass did some work (i.e. exchanged at least one pair) we must go through the list at least one more time. The second pass first exchanges the 6 and the 2, then the 6 and the 5.

The third pass exchanges the 4 and the 2: leaving us, at the end of the third pass, with the sorted list: But...that pass (number three) did an exchange and so we must go through the list one final time – which finds nothing to exchange – and then we know we're done.

This algorithm is the most "obvious" sorting technique and the one which beginning programmers tend to code without thinking about whether there might be a better, more efficient way.

Let us code it anyway.

A single pass looks like this:

```
index = 0 # start position in list L
while index < len(L)-1:
   if L[index] > L[index+1]: # need to swap
      L[index], L[index+1] = L[index+1], L[index][13]
   index = index + 1 # next position in list
```

Note the termination condition on the loop: the last item in the list is at position *len(L)-1* but we must stop *before* our loop control variable hits that value, as our test checks each item in the list against the *next* one and we *cannot* check the last item in the list against the next, as there *is* no next (and depending on which language we have programmed this in, our program may blow up or simply work incorrectly at this point) and so we must stop after we have checked the penultimate (last but one) item in the list against its next, i.e the last.

Now we have to put that loop inside a bigger one, which drives the program; as stated above it needs to stop when a pass has done nothing, so we need a variable (usually called a switch) to tell us whether or not we did any swaps in a pass – we don't care how many, just whether or not it was space.

Each time we do a swap we must set this switch to some value (depends on the language) which indicates True. Each time we begin a pass we must set it to the corresponding value for False because we have done no swapping so far this pass.

We should like our outer loop to look something like this:

```
while swaps_done:
   swaps_done = False; # nothing so far this pass
   code for single pass, with setting of swaps_done included
```

Again we've extended our knowledge of python a little.

You may be thinking: what does that first line *mean*? Shouldn't it be:

```
while swaps_done == True:
```

and that would, indeed, be valid python and it would work.

But, think about it carefully: when we first introduced *while* we said that the format was:

```
while <condition>:
   <body-of-loop>
```

[13] Another tremendously useful python feature, simultaneous assignment. Without this feature – and in many other languages – we should have to write something like:

```
temp = L[index]
L[index] = L[index+1]
L[index+1] = temp
```

Now we have naturally been thinking of <*condition*> in terms of comparisons: is x less than y? for example.

Yet, in fact the point of the comparison is that it results in a True/False value, known (for reasons which will become clear in a later chapter) as a *boolean* value.

So, the process that python goes through in deciding whether to continue looping is simple:

1. Evaluate <*condition*>
2. If it is True, keep looping, otherwise continue to the next step after the loop.

But, as we remarked above, the evaluation of <*condition*> results in either True or False; in this particular case the long version of <*condition*> is *swaps_done == True*. The result is that when *swaps_done* has the value True, so does our condition; when *swaps_done* has the value False, so does our condition.

In other words, the two *conditional expressions*:

```
swaps_done == True
```

and

```
swaps_done
```

always give the same result, they are *exactly* equivalent, so why not use the shorter version? especially as, if we choose the boolean variable's name well enough, the shorter version is more readable, as in this example*:*

```
while there_is_more_to_do:
```

Back to the algorithm: we need an initial setting for *swaps_done* – it should be clear that it is not False (why not?)

Our final algorithm looks like this:

```
while swaps_done:
   swaps_done = False; # nothing so far this pass
   index = 0 # start position in list L
   while index < len(L)-1:
      if L[index] > L[index+1]: # need to swap
         L[index], L[index+1] = L[index+1], L[index] # do swap
         swaps_done = True # did something this pass
      index = index + 1 # next position in list
```

If we throw in reading the list at the beginning and printing it at the end, here is the result:

```
$ python bubble.py
Enter list to sort:[4,6,2,5,9]
[2, 4, 5, 6, 9]
```

The performance of this algorithm is highly influenced by how "sorted" the original list is.

The best case is a completely ordered list: a single pass through it will result in *n-1* comparisons, no swaps and no further passes.

It should also be fairly clear that the worst case is when the list is in reverse order, with the first number in the list having the largest value and the last the smallest. The first pass will "bubble" that largest value up to the end, using *n-1* comparisons and *n-1* swaps. The first swap will have placed the next-largest value into the first place, so the second pass will result in bubbling that along to the second-to-last place: n-1 comparisons and n-2 swaps. And so on. There will be a total of n passes.

The result is that if we use the comparison as our fundamental unit of work, then the algorithm does: $\mathbf{n*(n-1)} = \mathbf{n^2}\text{-}\boldsymbol{n}$ units.

If we use the swap as our fundamental unit, the algorithm does:

```
(n-1) + (n-2) + (n-3) + .. + 1 = (n-1)*n/2 (by Gauss's formula)
```

In either case, for the worst case, the algorithm is $O(n^2)$

What is the average case? It is difficult to say exactly what an averagely out of order list might look like, but let us say that each element is roughly half the list away from its proper place and so requires half the number of swaps to get there as in the worst case.

The average case therefore will take say half the number of comparisons (on the assumption that the number of passes is halved) and half the number of swaps. In other words: $\mathbf{n*(n-1)/2} = \mathbf{(n^2\text{-}n)/2}$ comparisons and $\boldsymbol{(n^2\text{-}n)/4}$ swaps.

It doesn't make any difference: the average case is still $O(n^2)$ for both comparisons and swaps.

4.2.2.10 Selection sort

The basic strategy:

Divide the list into an unsorted and a sorted section, initially the sorted section is empty.

Locate the *largest* element in the unsorted section and replace that with the *last* element of the unsorted section. Move the marker between the unsorted and sorted section one position to the left.

Repeat until unsorted section of the list is empty.

It is probably easier to understand with a diagram of the algorithm's progress (read it left-to-right, top-to-bottom).

The thick vertical line marks the boundary between the sorted and unsorted parts of the list. Initially it is at the far right, as the sorted part is empty. After step 2 we no longer show the result of the swap, just which elements are being swapped.

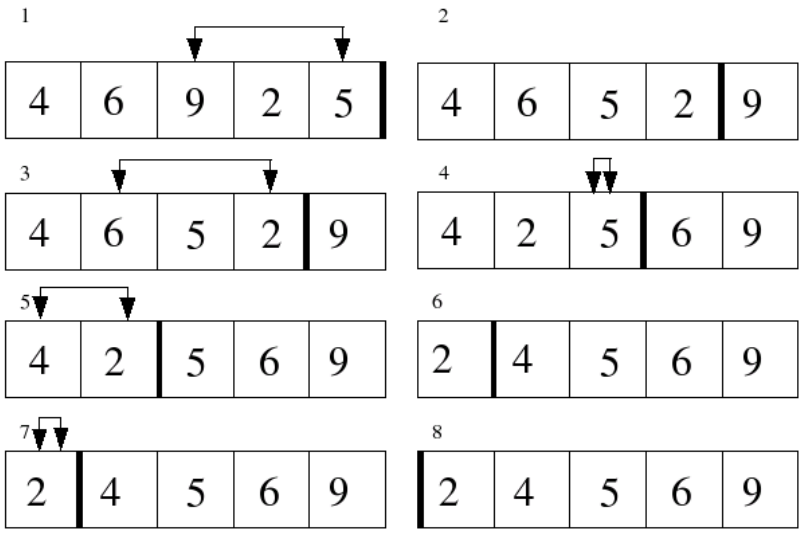


Figure 4.5: Selection sort at work

It is going to be easier to divide this problem into smaller ones in order to solve it.

- Firstly, we need to be able to find the largest item in a list: this is the largest_so_far problem we saw earlier.
- Let's build a function to find the largest value, then we can call that from our main *sort_list*

```
$ more select-sort.py
def find_largest (L):
   # find largest item in List L
   largest = L[0] # first item largest so far
   lpos = 0       # position of largest element
   i = 1          # check rest of list
   while i < len (L):    # search rest of list
      if L[i] > largest: # a new largest value
         largest = L[i]  # remember it
         lpos = i        # and where it was
      i = i + 1          # next element
   return lpos           # where largest  was

def sort_list (L):
   # do selection sort on list L
   unsorted = len (L) # whole list is unsorted
```

```
        while unsorted > 0: # something left to sort
           lpos = find_largest (L[:unsorted])
           L[lpos], L[unsorted-1] = L[unsorted-1], L[lpos] # swap
           unsorted = unsorted - 1 # one fewer left to sort

L = input ('Enter list to sort:')
sort_list (L)
print L
```

Now let's try it:

```
$ python select-sort.py
Enter list to sort:[5,7,1,13,99,4,22,99,5]
[1, 4, 5, 5, 7, 13, 22, 99, 99]

$ python select-sort.py
Enter list to sort:[10,9,8,7,6,5,4,3,2,1]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

If we use the number of comparisons (in *find_largest*) as our fundamental unit of work, then the number of steps is:

```
n-1 + n-2 + n-3 + ... + 1
```

which, as we have already seen, give us a sum of $(n^2-n)/2$, which is $\mathbf{O}(n^2)$.

Interestingly, the best- worst- and average-cases for this algorithm are all the same.

Presumably the best case would be a list which is completely sorted before we begin and the worst might be one which is in reverse order when we begin.

But the algorithm has no way of knowing whether either of these is the case and always takes the same number of steps.

4.2.2.11 Searching

We have already seen (the address book search) a sequential search algorithm. This is required when there is no order imposed on the entries we are searching.

The sequential search is $\mathbf{O}(n)$. The best case is when the item you are looking for is the first, the worst is when it is the last and on average it will be halfway along.

Although an $\mathbf{O}(n)$ algorithm is often good news, it is bad news for searching, which is even more frequently performed by the computer than sorting.

But there are things that we humans frequently search manually and can search fairly efficiently: two examples being dictionaries and telephone directories.

Why can we search them fairly quickly?

Because they are *ordered*, i.e. they are sorted. Imagine looking for John Smith in a phone book with no order...

The algorithm we use to look in the phone book (simplified) can be adapted to the computer.

4.2.2.12 Binary search

The basic strategy:

- Open the book about halfway.
- If that's the entry you're looking for, we're done.
- If it isn't then we restrict our search to the half of the book we *know* the desired entry must be in – i.e. the first or the second half – and keep repeating the above until we find it or are sure it isn't there.

This, when adopted to the computer, is called a *binary search*, because with each search we are dividing the (scope of) the problem in two, i.e. halving the amount of work left to do.

Imagine that we have the following entries in our table of names to look up:

Name	Ann	Bob	Dave	Gary	Nancy	Pat	Sue
Position	0	1	2	3	4	5	6

To find, for example,

- Nancy, we go to
- Gary (mid point at 3) to
- Pat (mid point of items 4:6) to
- Nancy (mid point of a single item).

Notice that there is an odd number of elements in our table, so there *is* a mid-point.

We can find:

- Gary in 1 step
- Bob or Pat in 2 steps
- All other entries in 3 steps

When we have an even number of elements there is no exact mid-point, so we must either choose the end of the 'low' half or the beginning of the 'high' half as our mid-point.

It should not matter which we choose.

Name	Ann	Bob	Dave	Gary	Nancy	Pat
Position	0	1	2	3	4	5

If we use the 'end of the low' as our midpoint then we can find:

- Dave in 1 step

- Ann or Nancy in 2 steps
- All other entries in 3 steps

Analysis:

Let us return to our initial table. We process the table as if it were a *tree* (of course, in computer science trees always grow *downwards* from the *root*).

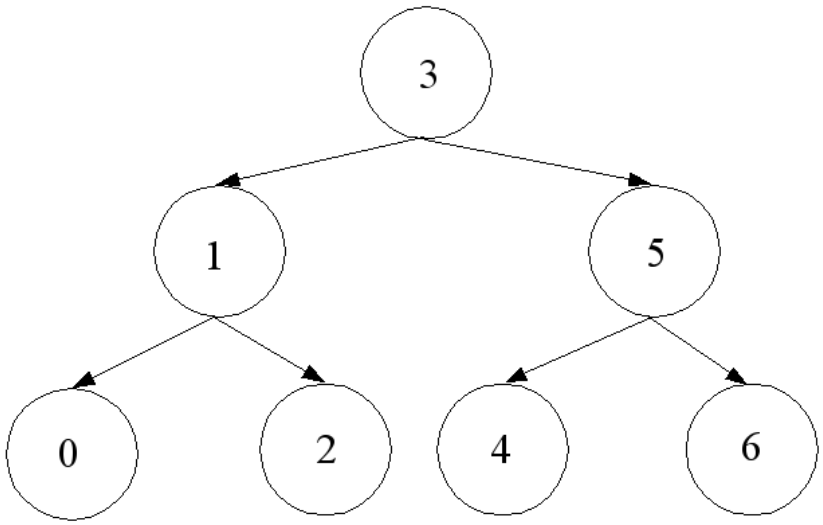


Figure 4.6: A binary tree

Note that the tree is organised so that at each *node* (circle in the diagram):

- every node that can be reached by going down and left has a smaller value ("is less than") than the node we're at
- every node to the right has a larger value.

As we check each name, we are essentially dividing the problem in half, as the number of names left to search is cut in half.

The number of times we can divide a number by 2 and not go below 1 is called *logarithm of n to the base 2*: the notation for this is: *log2 n* or *lg n*.

Hence we can see that the maximum number of name comparisons we need to perform in order to find a particular name (or find that it is not there) is the depth of the tree. In our running example that is 3.

So, if there are n names in our table we shall need approximately *lg n* comparisons.

Hence, the order of magnitude of the binary search is $\mathbf{O}$(lg n) – we can double the number of entries and only increase the number of steps (i.e. name comparisons) by one.

Unfortunately there are very few algorithms which are this efficient. Here is the algorithm:

```
$ more binsearch.py
Book= input ('Enter names ')
Name = raw_input ('Enter name to search for') left = 0
right = len (Book) - 1   # entire book
found = 'no'             # haven't found anything yet
while left <= right and found == 'no':
   # there's still something to search
   mid = (left + right) / 2 # calculate mid point
   if Name == Book[mid]: # is this the one?
      found = 'yes'      # it is
      position = mid     # remember where it is
   elif Name < Book[mid]: # is it in the first half?
      right = mid - 1    # yes, search that part
   else:                 # must be in second half
      left = mid + 1     # so search that
if found == 'yes':
   print 'Found', Name, 'at', position
else:
   print Name, 'not found'
```

It is worth examining the algorithm carefully and tracing its progress with a few test cases.

Given the list of names below – 'al', 'brian', 'charlie', 'deryk', 'elspeth', 'sheila', 'yoko' – imagine searching for 'charlie':

- Firstly left will be set to 0 and right to 6, so the first calculation for mid yields 3.
- The entry at position 3 is 'deryk' and our search name is less than this, so we set right to mid-1, i.e. 2.
- The condition on the loop is still true, so we enter it again and calculate the next value for mid as (0+2)/2, i.e. 1.
- The name at position 1 is 'brian', which is less than the name we are looking for, so now we set left to mid+1, i.e. 2.
- The loop condition is still true (now left == right) and so we enter the loop body again, setting mid to (2+2)/2, i.e. 2.
- Now the test finds that the entry at position 2 is the value we are looking for – 'charlie' – we set found to 'yes' and the loop terminates.
- We have found the requested name.

Now let us imagine searching for a non-existent name, say 'arthur'.

- Once again, initially left will be set to 0 and right to 6, so the first calculation for mid yields 3.
- The entry at position 3 is 'deryk' and our search name is less than this, so we set right to mid-1, i.e. 2.
- The condition on the loop is still true, so we enter it again and calculate the next value for mid as (0+2)/2, i.e. 1.
- The name at position 1 is 'brian', which is still greater than the name we're looking for, so once again we set we set right to mid-1, i.e. 0.
- The loop condition is still true (left == right) and so we enter the loop body again, setting mid to (0+0)/2, i.e. 0.
- The name at position 0 is 'al', which is less than what we are searching for, so we now set left to mid+1, i.e. 1.
- The loop condition is now false: $left > right$ $(1 > 0)$ and so we stop looping.
- As *found* still has the value 'no' the requested name does not exist in the table.

Note that this all depends on the entries in the table being sorted.

And here is the binary search algorithm in action (after wrapping a little test code around it):

```
$ python binsearch.py
Enter names: ['al', 'brian', 'charlie', 'deryk', 'elspeth', 'sheila', 'yoko']
Enter name to search for: al
al found at 0
Enter name to search for: yoko
yoko found at 6
Enter name to search for: brian
brian found at 1
Enter name to search for: sheila
sheila found at 5
Enter name to search for: charlie
charlie found at 2
Enter name to search for: deryk
deryk found at 3
Enter name to search for: elspeth
elspeth found at 4
Enter name to search for: aaron
aaron not found
Enter name to search for: zak
zak not found
Enter name to search for: david
```

```
david not found
Enter name to search for:
```

Note the testing strategy:

- Use as many valid (i.e. in the book) names as feasible,
- Try non-existent names which should be
  1. Before the first name in the book
  2. Somewhere in the book or
  3. After the last name in the book.

4.3 Linear vs. logarithmic algorithms

The logarithmic algorithm is very efficient indeed:

n	lg n
2	1
4	2
8	3
16	4
32	5
...	...
1024	10
2048	11
4096	12

Number of steps for n and lg n

Would that all algorithms could be so efficient!

4.3.0.13 Polynomial Algorithms

All of the algorithms we have looked at up to now are examples of polynomial algorithms. For example, $lg\ n$, n, n^2, n^3, ... , n^{3000} are all polynomial algorithms.

More generally any algorithm where:

```
#steps = an^m + bn^m-1 + cn^m-2...+z
```

(where a..z are constants), i.e. has order of magnitude $\mathbf{O}(n^m)$, is a polynomial algorithm.

The comparative growth curves of several polynomial algorithms (n^2, n, $\sqrt{n}$, $lg\ n$) are shown in the graph.

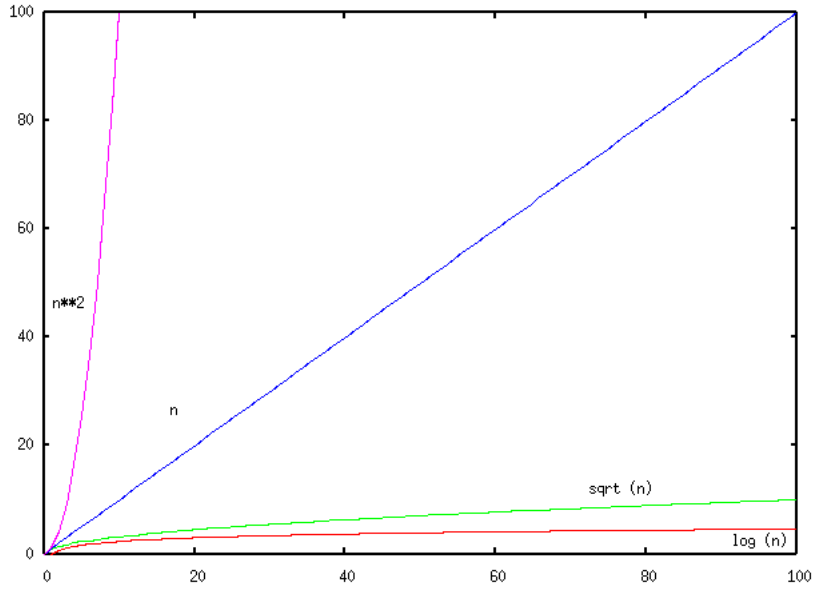


Figure 4.7: Some polynomial performance growth curves

4.3.0.14 When It All Goes "Pear-Shaped"

Some algorithms are *exponential* rather than polynomial: an exponential expression is one such as 2^n .

A classic example is playing chess, where at every stage the number of possible moves to be analysed increases exponentially with the number of moves ahead one is trying to play. This is why chess-playing programmes have only recently managed to beat human masters and then by using subtler strategies (and customised hardware) rather than the "brute force" approach.

Another classic example is the recursive definition of the Fibonacci numbers[14] defined so that each number is the sum of the proceeding two:

[14]Fibonacci, or Leonard da Pisa, was an Italian mathematician who, among other things, was the first to introduce arabic numerals (what we use today) into Western Europe.

He is best remembered today for the numerical series which bears his name, which he developed as a model for the growth of a population of rabbits.

Interestingly enough, the numbers in the Fibonacci series – which begins 1, 1, 2, 3, 5, 8, 13, 21, 34 - tend to appear in nature and the numbers which are *not* in the series tend not to. Count the leaves on a clover (4 is not in the series), the veins on a leaf, the number of spirals on an anemone: chances are you'll find one of the Fibonacci numbers.

One other interesting thing about the Fibonacci series: the ratio between successive numbers in the series tends to a limit. The ratio is known by the Greek letter φ (phi) and has the value $\frac{1}{2}\left(1+\sqrt{5}\right)\tilde{}= 1.6180339887498949...$

This is also the number known as the *Golden Ratio* or *Golden Section*, the ratio of the lengths of the sides of that rectangle which is most pleasing to the human eye.

This rectangle appears frequently in human art throughout history (check out the sides of the Parthenon in Athens, for example).

$\texttt{fib}_1 = 1$
$\texttt{fib}_2 = 1$
$\texttt{fib}_n = \texttt{fib}_{n-1} + \texttt{fib}_{n-2}$

and here is a python function which calculates the nth Fibonacci number. (For simplicity we assume that the supplied n is greater than zero):

```
def fibonacci (n):
   if n == 1 or n == 2:   # fib(1) and fib(2)
      return 1            # are defined to be 1
   else:
      return fibonacci (n-1) + fibonnaci (n-2)
                          # by the definition
```

Here are some timings of the function (with a little code wrapped around it as a user interface) on my home computer, an AMD Athlon-64 3200+ 2GHz cpu with 2GB of RAM; all times are in CPU seconds:

n	Iterative	Recursive
10	0.012	0.02
20	0.024	0.07
30	0.024	5.00
35	0.024	55.25
40	0.024	605.46
50	0.024	74840.35
60	0.032	

Note that the iterative timings are so small that factors like system load and the accuracy of the measuring tool are distorting the results.

Neverthess, it is quite clear that, as n increases, the recursive algorithm starts to take *much* longer. And the recursive calculation of fib(60) is still running (after several days) as I type this.

Or, to be precise, the *vanilla* recursive algorithm. It is quite possible to produce what is called a *tail-recursive* version of the factorial algorithm which has a performance growth graph much more similar to the iterative one[15].

Spooky, or what?

[15]The point of tail recursion is that nothing is done with the value of the recursively-called function except to return it. Our recursive function makes two recursive calls and *then* adds the results.

Here is a tail-recursive version: note that the main **fib** function simply checks for our two base cases, then calls the tail-recursive **trfib** function, passing it the first two fibonaccis. The variable **n** is used to countdown until it hits zero.

```
def fib (n): # recursive fibonacci
    if n < 3:
        return 1
    else:
        return trfib (n-3, 1, 1)
def trfib (n, prev, poprev):
    if n == 0:
        return prev+poprev
```

An exponential algorithm is $\mathbf{O}(a^n)$, where a is a constant.

For *sufficiently large n* any exponential algorithm will eventually do significantly more work than a polynomial algorithm. Problems for which only exponential algorithms are known to exist are called *intractable*.

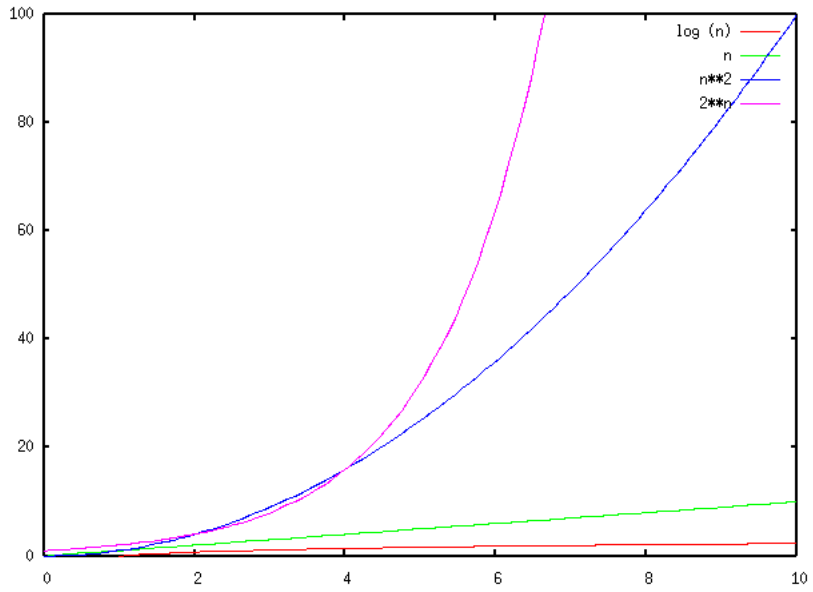


Figure 4.8: Exponential algorithm growth curve

The graph shows how n^2 (which was the fastest growing in the previous example) is dwarfed by the growth of 2^n.

The existence of exponential algorithms does not mean, however, that we should abandon recursive algorithms altogether – just that we need to be aware of the order of magnitude of any algorithm that we propose to use, and to be aware of when, and when not, to use that algorithm

As a final example, consider four algorithms which are, respectively, $\mathbf{O}(\lg n)$, $\mathbf{O}(n)$, $\mathbf{O}(n^2)$ and $\mathbf{O}(2^n)$, and a computer system which can process1 step in 0.0001 of a second.

Although for small input size all four have reasonable performance, the magic of the computer is that it does not get tired and can process huge data sets.

So we really need to examine the algorithm's performance with large data sets as well.

```
    else:
        return trfib (n-1, prev+poprev, prev)
```

n	10	50	100	1000
lg n	0.0003s	0.0006s	0.0007s	0.001s
n	0.001s	0.005s	0.01s	0.1s
n^2	0.01s	0.25s	1s	1.67minutes
2^n	0.1024s	3570years	$4x10^{16}$centuries	$3x10^{287}$centuries

It is worth remembering, in connexion with the exponential algorithm, that the current estimate for the age of the universe – i.e. the time which has passed since the "big bang" – is somewhere between 7 and 16 billion years and that the age of the earth is roughly 4.8 billion years.

In other words the universe is at most $1.6x10^8$ centuries and the earth roughly $4.8x10^7$ centuries old.

We should therefore avoid exponential algorithms if we possibly can.

Part III

History

Chapter 5

A Brief History of the Computer

> The first law for the historian is that he shall never dare utter an untruth.
>
> The second is that he shall suppress nothing that is true.
>
> Moreover, there shall be no suspicion of partiality in his writing, or of malice.
>
> Marcus Tullius Cicero, *Pro Publio Sestio*

5.1 Introduction

Unfortunately, most "histories" of computing have been written in the USA and have tended to ignore anything which occurred outside that country. The full history of mankind's attempts to automate calculating is a long and interesting one, but we only have time for the edited highlights.

5.2 Pre-Mechanical

5.2.1 The Abacus

The abacus has, in some form or another, been used for around two thousand years. It is found in the Middle East, in ancient Greece and Rome, in China, Russia and Japan. The word itself has been variously traced back to the Phoenician abak (sand) or the Greek abax (calculating board) possibly derived from the Hebrew avak (dust).

The basic principle is simple: each column of beads represents one decimal digit. The Chinese (suan pan, figure 5.1) and Japanese (soroban, figure 5.2) abacus differ in that the former has two beads above and five below "the bar"

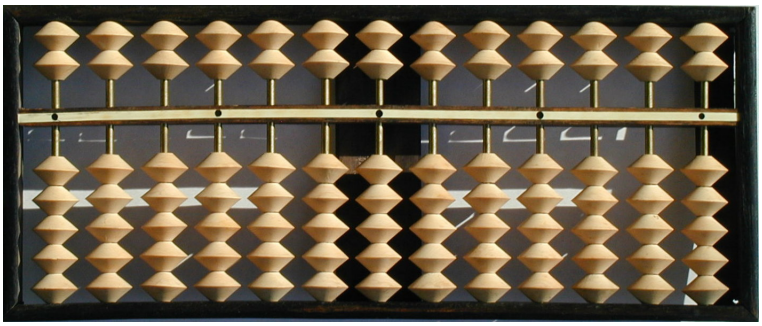


Figure 5.1: The suan-pan, or Chinese abacus

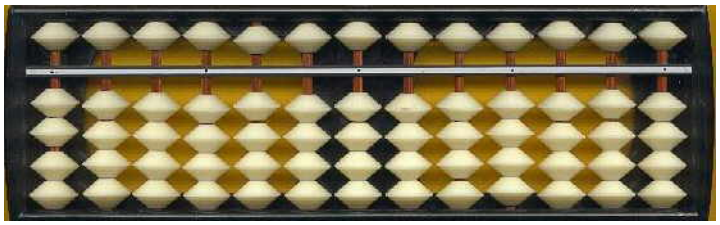


Figure 5.2: The Soroban, or Japanese abacus

(also known as the "top" and "bottom" decks), whereas the latter has one above and four below. The beads above the bar represent 5 and the beads below 1 each.

The Chinese version contains redundancy and, from about 1850, the Chinese abacus has only had a single bead above the bar.

In November 1946 a now-legendary contest between Mr. Kiyoshi Matsuzaki – a champion operator of the abacus in the Savings Bureau of the Ministry of Postal Administration – and Private Thomas Nathan Wood – a US army clerk of the 20th Finance Disbursing Section of General MacArthur's headquarters armed with an electronic calculator – was held. In problems of addition, subtraction and division the abacus won; only in multiplication was the calculator faster. Abacuses are still used in many parts of the world and by the visually-challenged.

5.2.2 Napier's Bones

In 1614 the Laird of Merchiston, Scottish mathematician John Napier, invented logarithms as a means to make calculations (especially multiplication and division) more efficient and more accurate – particularly for astronomers.

Logarithms are based upon the fact that

$$x^a + x^b = x^{a+b}$$

Figure 5.3: John Napier (1550-1617)

So, if we wish to multiply, e.g. 2.479 by 33.964 we look up both numbers' logarithms (to the same base – table are usually to base 10) add the numbers and then do a reverse-lookup to find the product.

Napier's 1617 text *Rabdologiae* ('numeration by little rods') explained the use of his calculating system, known as Napier's Bones (or rods), a collection of 10 rods on which the multiplication tables were engraved – see figure 5.4. Although the bones (or rods) constitute an early attempt at automatic calculation, Napier is now mainly remembered for logarithms.

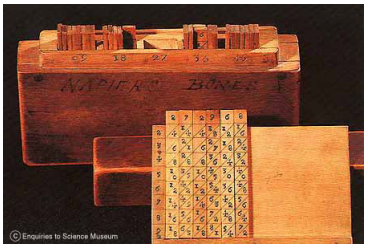


Figure 5.4: Napier's Bones

The bones are now merely of historical interest; logarithms ($O(\ln n)$ anyone?) are still used daily.

You can read all about the workings of the bones at

`http://en.wikipedia.org/wiki/Napier's_bones.`

You can find out more about Napier (although his biographical details are still largely mysterious) at

`http://www.cee.hw.ac.uk/~greg/calculators/napier/`

5.2.3 William Oughtred and the Slide Rule

Figure 5.5: Edmund Gunter's logarithmic scale

Until the 1970s anyone undergoing any kind of technical education had to become familiar with the workings of the slide rule. The slide rule is an analogue device, derived from the logarithmic scale of Englishman Edmund Gunter, itself inspired by the logarithmic aspects of Napier's bones. In 1622 Oughtred, an Anglican minister, put two of Gunter's sticks side by side and used them to multiply and divide. Oughtred also invented a circular slide rule.

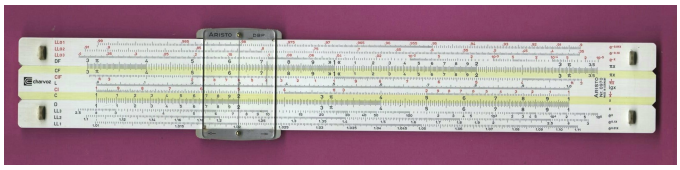


Figure 5.6: A Modern slide rule

By 1675 Sir Isaac Newton had used three parallel logarithmic scales to solve cubic equations and also made the first suggestions about using the device you see in the illustration above, which has a fine-drawn line for accurate viewing. This line is called the cursor.[1]

Modern slide rules can be very complex artefacts.

5.3 Mechanical Calculation

5.3.1 Early Attempts

Among the earliest known attempts at mechanical calculation was the 1623 "Calculating Clock" of Wilhelm Schickard (1592-1635), of Tübingen, Württemberg (now in Germany). This was a 6-digit machine that could add and subtract, and indicated overflow by ringing a bell. Mounted on the machine was a set of Napier's Rods.[2]

Schickard's original machine and plans were lost and forgotten in the war that was going on in the 1620s. The plans were finally rediscovered in 1935, only to be lost in war again, and then re-rediscovered in 1956 – by the same

[1] Oh yes indeed!

[2] Schickard was a friend of the astronomer Kepler. According to an informal communication, Schickard sometimes used the device for 7-digit calculations, counting rings of the overflow bell by putting rings on "another part of his body" – we shall not go there.

man! The machine was be reconstructed in 1960, and found to be workable – which apparently is more than could be said for...

5.3.2 The Pascalene

Figure 5.7: Blaise Pascal (1623-1662)

The French mathematician and philosopher Blaise Pascal made a number of significant contributions to both fields. He also invented one of the first successful calculators, the Pascalene. The devices (really a mechanical adding machine) was invented to help Pascal's father (a tax collector) to calculate taxes more quickly and accurately. Some things never change...

This 5-digit machine used a different carry mechanism from Schickard's, with rising and falling weights instead of a direct gear drive; it could be extended to support more digits, but it could not subtract, and was probably less reliable than Schickard's simpler method.

Figure 5.8: The Pascalene

Where Schickard's machine was forgotten – and indeed Pascal was apparently unaware it ever existed – Pascal's became well known and established the

computing machine concept in the intellectual community. He made more machines and sold about 10-15 of them, some supporting as many as 8 digits.[3] Patents being a thing of the future, several others also built and sold copies.

You can read about the workings of the Pascalene in some considerable detail at

`http://www.macs.hw.ac.uk/~greg/calculators/pascal/About_Pascaline.htm`

The Pascalene was very influential. As late as 1920 the Reliable Typewriter and Adding Machine Co. of Chicago introduced its Addometer based on the Pascalene. It was not a commercial success.

5.3.3 The "Stepped Reckoner"

Figure 5.9: Gottfried Leibnitz (1646-1716)

In 1674 Gottfried Wilhelm von Leibniz (1646-1716), of Leipzig, designed the "Stepped Reckoner"; it was constructed by a man named Olivier, of Paris. It used a movable carriage so that it could multiply, with operands of up to 5 and 12 digits and a product of up to 16. [4]The user had to turn a crank once for each unit in each digit in the multiplier; a fluted drum translated the turns into additions. But the carry mechanism required user intervention, and didn't really work in all cases anyway.

Leibniz's machine was not forgotten, but it did get misplaced in an attic within a few years – and stayed there until 1879 when it was noticed by a man working on the leaky roof.[5]

[3]Several survive to the present day.
[4]The Pascalene could only add and, rather clumsily, subtract.
[5]Leibniz, or Leibnitz, was also the co-inventor of calculus.

5.3.4 Punched Cards

5.3.4.1 Jacquard's Loom

Figure 5.10: Joseph-Marie Jacquard (1752-1834)

In 1801 Joseph-Marie Jacquard invented a mechanical loom which could weave patterns under the control of a "program" punched onto rectangular cards. The illustration is a portrait of Jacquard woven from silk on one of his looms under the control of a 10,000 card program.

Figure 5.11: A modern Jacquard loom
Strictly speaking we should speak of the Jacquard *head*, but after two centuries of misuse it is probably too late to correct now.

Although the loom was not in itself in any way a calculating device, it was Jacquard's invention of the punched card which was to influence the development of the computer in several ways.

5.3.4.2 Hollerith's tabulator

Figure 5.12: Hermann Hollerith (1860-1929)

The US Census of 1890 was estimated to involve gathering information from some 62 million Americans. Given the existing manual methods of tabulating the information the results of the 1890 census would not be available until after the 1900 census, by which time it would be less than timely.

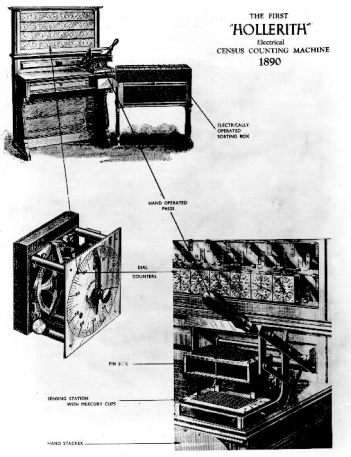


Figure 5.13: An early advertisement for Hollerith's Tabulator

Herman Hollerith had the idea of storing the information using Jacquard's punched cards and invented a machine which could sort and tabulate the cards. The machine could do the work about eight times faster than it could be done by hand. In 1896 Hollerith founded the Tabulating Machine Company to exploit his inventions and was its consulting engineer until his retirement in 1921. In 1924 the company changed its name – to International Business Machines (IBM to you).

5.4 The Age of Engineering

5.4.1 Charles Babbage

Figure 5.14: Charles Babbage (1792-1871)

From Harrison's invention of the chronometer until the mid-19th century navigation at sea was done using manually-calculated tables of longitude. Unfortunately, the hand-calculations were error-prone – and an error in a navigation table could easily result in the loss of a ship, its crew and cargo.

Cambridge mathematician Charles Babbage decided that the process could and should be automated and so in 1822 he began work on the Difference Engine which used the method of finite differences to calculate its results.

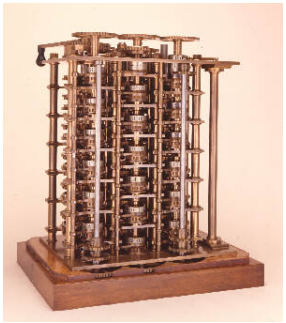


Figure 5.15: Part of the Difference Engine constructed in 1832

The complete engine, which would have been room-sized, was planned to be able to operate both on 6th-order differences with numbers of about 20 digits, and on 3rd-order differences with numbers of 30 digits. Each addition would be done in two phases, the second one taking care of any carries generated in the

first. The output digits would be punched into a soft metal plate, from which a plate for a printing press could be made.

Unfortunately and for various reasons, the full scale machine was never completed, although a demonstration model was and was very fashionable for a period in the 1830s. This operated on 6-digit numbers and 2nd-order differences (i.e. could tabulate quadratic polynomials).

There has been considerable discussion about the reasons for the failure, but the one usually given – that it was not possible at the time to build a machine to the precise tolerances required – is specious and does not hold water. A combination of factors led to the demise of the difference engine including a major row with his engineer, Joseph Clement, over money.

Clement, it transpired, was padding his bills by making everything (including the metal frame for the engine) to the same precise tolerances, therefore increasing the cost to no purpose. But perhaps the major reason for the abandonment of the Difference Engine was that Babbage had had a better idea: a general purpose calculating machine, which he called the Analytical Engine.

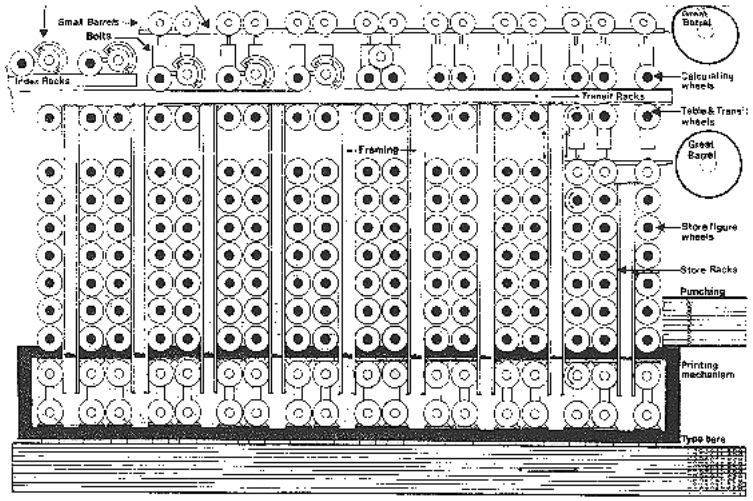


Figure 5.16: Plan of the Analytical Engine dating from 1858

With the one exception of the stored-program concept Babbage's Analytical Engine was (or would have been) essentially a mechanical computer. It had the major components of what is now known as the von Neumann Architecture: data was input on punched cards.

Babbage was a doer, not a writer and much of what we know about his engines today comes from a paper by an Italian (Menebrea) which was translated into English and annotated by Ada Augusta, Countess Lovelace. Ada, who was Lord Byron's daughter, has been described as the world's first programmer.

Pleasing fiction though this is, it does not stand up under scrutiny. In his book The Difference Engine, Doron Swade, who is Assistant Director and Head

of Collections at the Science Museum in London and who oversaw the project to build the Difference Engine from Babbage's original design, remarks that:

> Because of her article 'Sketch of the Analytical Engine', Ada's role in Babbage's work has been both exaggerated and distorted down the years, like a Chinese whisper...The notion that she made an inspirational contribution to the development of the Engines is not supported by the known chronology of events. The conception and major work on the Analytical Engine were complete before Ada had any contact with the elementary principles of the Engines. The first algorithms or stepwise operations leading to a solution–what we would now recognise as a 'program', though the word was not used by her or by Babbage–were certainly published under her name. But the work had been completed by Babbage much earlier.

Figure 5.17: Augusta Ada King, countess of Lovelace 1815 - 1852

Swade also quotes Bruce Collier, co-author of the primary biography of Babbage:

> There is one subject ancillary to Babbage on which far too much has been written, and that is the contributions of Ada Lovelace. It would only be a slight exaggeration to say that Babbage wrote the 'Notes' to Menabrea's paper, but for reasons of his own encouraged the illusion in the minds of Ada and the public that they were authored by her. It is no exaggeration to say that she was a manic depressive with the most amazing delusions about her own talents, and a rather shallow understanding of both Charles Babbage and the Analytical Engine... To me, this familiar material [Ada's correspondence with Babbage] seems to make obvious once again that Ada

> was as mad as a hatter, and contributed little more to the 'Notes' than trouble... I will retain an open mind on whether Ada was crazy because of her substance abuse...or despite it. I hope nobody feels compelled to write another book on the subject. But, then, I guess someone has to be the most overrated figure in the history of computing."

The reader can gauge for him or herself: here is an excerpt from one of Ada's letters to her mother, written in February 1841:

> And now I must tell you what my opinion of my own mind & powers is exactly; - the result of a most accurate study of myself with a view to my future plans, during many months. I believe myself to possess a most singular combination of qualities exactly fitted to make me pre-eminently a discoverer of the hidden realities of nature. You will not mistake this assertion either for a wild enthusiasms

Ada's tragic death does nothing to undermine Collier's view, except to suggest that the hatterish aspects of her personality were probably inherited – from both sides.

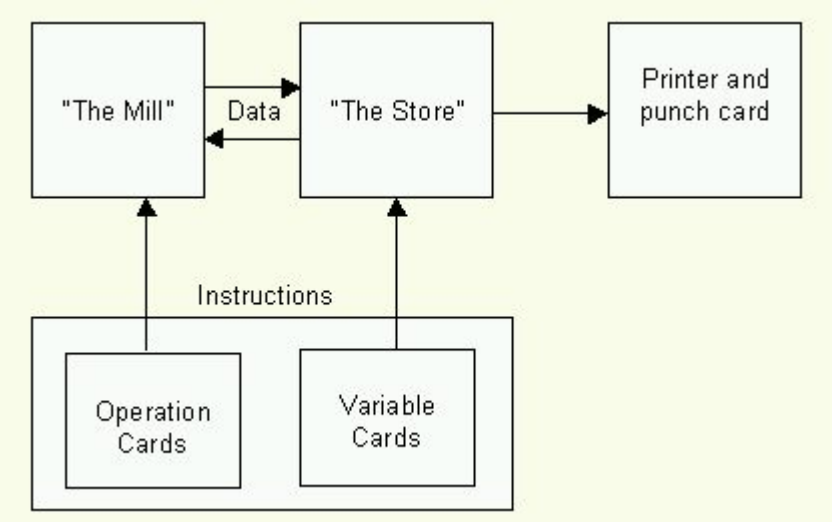


Figure 5.18: Block diagram of the Analytical Engine

Babbage continued to work on the design for years, though after about 1840 the changes were minor. The machine would have operated on 40-digit numbers; the "mill" (CPU) would have 2 main accumulators and some auxiliary ones for specific purposes, while the "store" (memory) would have held perhaps 100 more numbers. There would have been several punch card readers, for both programs and data; the cards would have been chained and the motion of each chain could be reversed. The machine would have been able to perform conditional jumps

and could theoretically perform an addition in 3 seconds and a multiplication or division in 2-4 minutes.

In 1879 a committee investigated the feasibility of completing the Analytical Engine and concluded that it was impossible now that Babbage was dead.

5.4.2 Other 19th Century Developments.

And there the main story rests for several decades. There were other developments in the general field of mechanical calculation, but no real progress towards what we would today call a computer. Interesting 19th century facts include:

- In 1853 a Swedish father-and-son team of engineers called Scheutz complete the first full-scale difference engine, the "Tabulating Machine". Babbage was delighted. The machine operated on 15-digit numbers and 4th-order differences, and produced printed output as Babbage's would have. A second machine was later built to the same design by the firm of Bryan Donkin of London In 1858. The first Tabulating Machine was bought by the Dudley Observatory in Albany, New York, and the second one by the British government. The Albany machine was used to produce a set of astronomical tables; but the observatory's director was then fired for this extravagant purchase, and the machine was never seriously used again, eventually ending up in a museum.[6]

- In 1878 Ramon Verea, living in New York City, invented a calculator with an internal multiplication table, which is much faster than the shifting carriage or other digital methods. He wasn't interested in putting it into production; he just wanted to prove that a Spaniard could invent as well as an American.

- In 1886 The "Comptometer" was designed by Dorr E. Felt (1862-1930), of Chicago. This was the first calculator where the operands were entered merely by pressing keys rather than having to be, for example, dialled in. It was feasible because of Felt's invention of a carry mechanism fast enough to act while the keys return from being pressed In 1889 Felt invented the first printing desk calculator.

- In 1892 William S. Burroughs (1857-98), of St. Louis, invented a machine similar to Felt's but more robust, and this was the one that really started the office calculator industry. This machine was still hand powered, but it wasn't many years before electric calculators appeared.[7]

[6] The second machine, however, had a long and useful life.

[7] Yes, the William S. Burroughs, author of The Naked Lunch and The Soft Machine, is a relative.

5.5 The Early 20th Century

- In 1906, Henry Babbage, Charles's son, with the help of the firm of R. W. Munro, completed the mill of his father's Analytical Engine. Babbage wanted to vindicate his father and to show that the Analytical Engine would have worked. It did, but the complete machine was never produced.
- In 1919 W. H. Eccles and F. W. Jordan published the first circuit design for a flip-flop. A flip-flop is a bistable circuit which provides the basis for computer memory.
- In 1926 Derrick Henry Lehmer, at Berkeley, CA, constructed a machine for factoring whole numbers. Lehmer's machine was based on 19 bicycle chains. A later machine used punched tape – not paper tape, but film stock.
- In 1931/2 E. Wynn-Williams, at Cambridge, England, used thyratron tubes to construct a binary digital counter for use in connection with physics experiments.

5.6 A Theoretical Basis – Alan Turing

Figure 5.19: Alan Turing (1912-1954)

In 1935 a young Cambridge mathematician, Alan Turing, began thinking about one of David Hilbert's 23 problems, which he had posed at a 1900 conference as being the outstanding problems of mathematics.

The *Entscheidungsproblem* ('decision' problem) is Hilbert's problem number 3: Is mathematics decidable? i.e. is there a 'mechanical' method that can be applied to any mathematical assertion so that—at least in principle—we can know whether that assertion is true or not?[8]

On Computable Numbers, with an application to the Entscheidungsproblem was the paper Turing eventually published in 1937. In it he described what is now known as a Turing machine.

[8]To find out more about Hilbert and his programme for mathematical advance see: http://babbage.clarku.edu/~djoyce/hilbert/

Here is Turing, in Hugh Whitmore's play *Breaking the Code*, describing the paper to Dilwyn Knox, who is interviewing him for secret work at Bletchley Park (see below):

> TURING: Hilbert had, as I said, thought that there should be a clearly defined method for deciding whether or not mathematical assertions were provable. The decision problem he called it. 'The Entscheidungsproblem'.
>
> In my paper on computable numbers I showed that there can be no one method that will work for all questions. Solving mathematical problems requires an infinite supply of new ideas. It was, of course, a monumental task to prove such a thing. One needed to examine the provability of all mathematical assertions past, present, and future.
>
> How on earth could this be done? One word gave me the clue. People had been talking about the possibility of a mechanical method, a method that could be applied mechanically to solving mathematical problems without requiring any human intervention or ingenuity.
>
> Machine! – that was the crucial word. I conceived the idea of a machine, a Turing Machine[9], that would be able to scan mathematical symbols – to read them if you like – to read a mathematical assertion and to arrive at the verdict as to whether or not that assertion is provable. With this concept I was able to prove that Hilbert was wrong.
>
> My idea worked.
>
> KNOX: You actually built this machine?
>
> TURING: No, of course not. It was a machine of the imagination, like one of Einstein's thought experiments. Building it wasn't important; it's a perfectly clear idea, after all.
>
> KNOX: Yes, I see; well, I don't, but I see something. I think. Forgive me for asking a crass and naive question – but what is the point of devising a machine that cannot be built in order to prove that there are certain mathematical statements that cannot be proved? Is there any practical value in all this?
>
> TURING: The possibilities are boundless – this is only the beginning. In my paper I explain how a special kind of Turing Machine – I call it the Universal Machine – could carry out any mental process whatsoever.

Later Turing went to Princeton to do his PhD, under the supervision of logician Alonzo Church. Turing's thesis demonstrated the equivalence of his universal machine and Church's L-calculus ('lambda' calculus). The so-called Church-Turing Hypothesis says that either mathematical formulation describes the limits of what can be 'mechanically' (by which we include electronically) calculated.

[9] Turing himself never referred to "Turing Machines", his term was "a-machine".

In its description of the machine which has a single 'tape' which contains both the instructions ('program') and the data on which it is to operated, Turing essentially invented the stored program concept.[10]

Turing was also a 'hands-on' designer; while in the US he designed and partially built an electronic multiplier (itself merely part of an encryption machine that was, for its time, unbreakable). He will re-enter the story shortly.

5.7 Electromechanical Calculation

During the 1930s a number of significant attempts were made to build an electromechanical calculator, using relays as switching elements. Some of these have been claimed as the first computer.

5.7.1 Konrad Zuse

Figure 5.20: Konrad Zuse (1910-1995)

The German-born engineer Konrad Zuse was responsible for some of the most significant developments in electromechanical calculating and computing.

In 1938 he built the Z1 (which did not, in fact, even employ relays) in his parents' living room – the world's first home computer? It was called V1 (*Versuchsmodell* – i.e. trial model) at the time but retroactively renamed Z1 after the war. It worked with floating point numbers having a 7-bit exponent, 16-bit mantissa, and a sign bit. The memory used sliding metal parts to store 16 such numbers, and worked well; but the arithmetic unit was less successful. The program was read from punched tape.

Like Lehmer, Zuse used film rather than paper for his tape; specifically, discarded 35 mm movie film. Data values could be entered from a numeric keyboard, and outputs were displayed on electric lamps.

Zuse next went on to the Z2, built using 800 ex-telephone-exchange relays. Unfortunately all plans and pictures are lost.

[10] "The fundamental conception is owing to Turing – insofar as not anticipated by Babbage, Lovelace and others" (John von Neumann to S. Frankel)

Figure 5.21: The Z1 in Zuse's parent's apartment

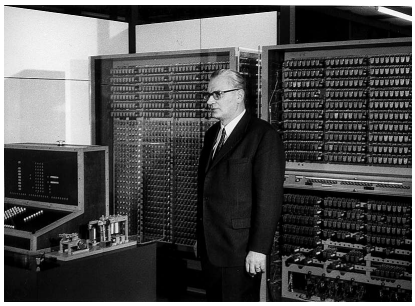


Figure 5.22: Konrad Zuse and the rebuilt Z3

The Z3 was built between 1939 and 1941 in Berlin, with some government funding. It was essentially a programmable electromechanical computer, although it did not employ the stored program principle (its memory was far too small).

Zuse had, though, independently of Turing, developed the concept, as can be seen in this 1937 diary extract

> Die Operationen folgen einem Plan ähnlich einem Rechenplan. Mit Ausgangsbedingungen und Resultat. Dementsprechend Speicherplan. Jedoch kann der Speicher- oder Arbeitsplan sich aus den vorhergehenden Operationen ergeben (z.B. die Nummern der Speicherzellen) und sich so aus sich selbst aufbauen (vgl. 'Keimzelle').[11]

In 1961 Zuse supervised the rebuilding of the Z3. The Z3 used binary and floating-point arithmetic, but it had no conditional jump instruction.

[11] The operations follow a plan similar to a computing plan. With initial conditions and result. Accordingly, a storage plan. However, the storage or work plan can still result from the preceding operations (e.g. the numbers in the storage cells) and in this way be built from itself (cf. 'germ cell').

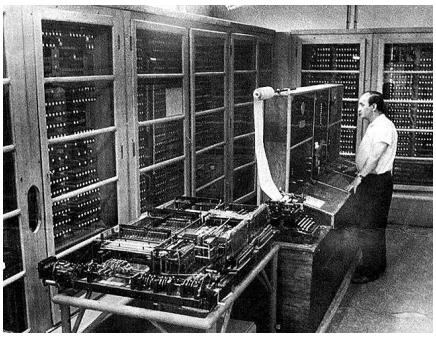


Figure 5.23: The Z4

The follow-on, Z4, intended for eventual mass-production, was not completed during the war – in fact, after seeing the V2 factory an Nordhausen, with its 20,000 forced labourers from the concentration camps, Zuse hid parts of the machine. The Z4 was completed in the late 1940s and one was sold to ETH Zűrich, where it was in use for a number of years. It has claims to be the first commercially-available computer, yet it still lacked the stored program capability.

Zuse, who also in 1945-6 constructed (yet never implemented) the first high level language ('Plankalkul'), has only in recent years been acknowledged as one of the pioneers of the field.

5.7.2 Other Developments in the 1930s

- In 1935 International Business Machines introduced the IBM 601, a punch-card machine with an arithmetic unit based on relays and capable of doing a multiplication in 1 second. The machine became important both in scientific and commercial computation, and about 1,500 of them were eventually made.

- In September 1939 World War II began. As usual, the USA turned up two years late.

- In November 1939 John V. Atanasoff (1903-95) and graduate student Clifford Berry (1918-63), of Iowa State College (now the Iowa State University), Ames, Iowa, completed a prototype 25-bit adder. This was the first machine to calculate using vacuum tubes. To store the operands, it had two 25-bit words of memory in the form of capacitors (with refresh circuits using more vacuum tubes – the first regenerative memory) mounted one word on each side of a revolving disk. There was no input device; the user entered the operands directly into memory, by tapping the appropriate capacitors with a wire!

- Two years later, Atanasoff and Berry completed a special-purpose calculator for solving systems of simultaneous linear equations, later called

the "ABC" ("Atanasoff-Berry Computer"). This used the same regenerative capacitor memory as their prototype, but with 60 50-bit words of it, mounted on two revolving drums. The clock speed was 60 Hz, and an addition took 1 second. (For the purposes of this calculator, multiplication was not required). There were circuits to convert between binary and decimal for input and output; the machine included several hundred vacuum tubes altogether. For secondary memory the ABC used punch cards, moved around by the user. The holes were not actually punched in the cards, but burned by an electric spark. The card system was a partial failure; its error rate of 0.001% was too high to solve large systems of equations.[12]

Although designed specifically for certain problems, not generally programmable, lacking the stored-program capability, never finished and dubiously reliable, in an extraordinary (and from some points of view, extraordinarily ill-informed) 1973 judgement, a US district court ruled that the ABC was the world's first computer.

- In the same month (November 1939) at Bell Labs, Samuel Williams and Stibitz built a calculator which could operate on complex numbers, and gave it its imaginative name – the "Complex Number Calculator". It was later known as the "Model I Relay Calculator". It used telephone switching parts for logic: 450 relays and 10 crossbar switches. Numbers were represented in "plus 3 BCD"; that is, for each decimal digit, 0 was represented by binary 0011, 1 by 0100, and so on up to 1100 for 9; this scheme required fewer relays than straight BCD.

  Rather than requiring users to come to the machine to use it, the calculator was provided with three remote keyboards, at various places in the building, in the form of teletypes.[13] Only one could be used at a time, and the output was automatically displayed on the same one.

5.8 The Second World War

5.8.1 Enigma

The Enigma machine used a series of disks ("rotors") with sets of 26 contacts wired so as to permute and re-permute the alphabet; the sequence of rotors and their initial settings are changed from time to time, forming a key.

The military machine was based on a commercial design first exhibited in Paris in 1923. The machine underwent a series of modifications which continued during the war; breaking the codes was therefore a continuously challenging task.

The machine's basic configuration depended on:

[12] Atanasoff left Iowa State after the US entered the war, and this ended his work on digital computing machines. The ABC was largely forgotten within a few years, and dismantled in 1946 when the storage space was needed.

[13] This was therefore arguably the first remote access computer system.

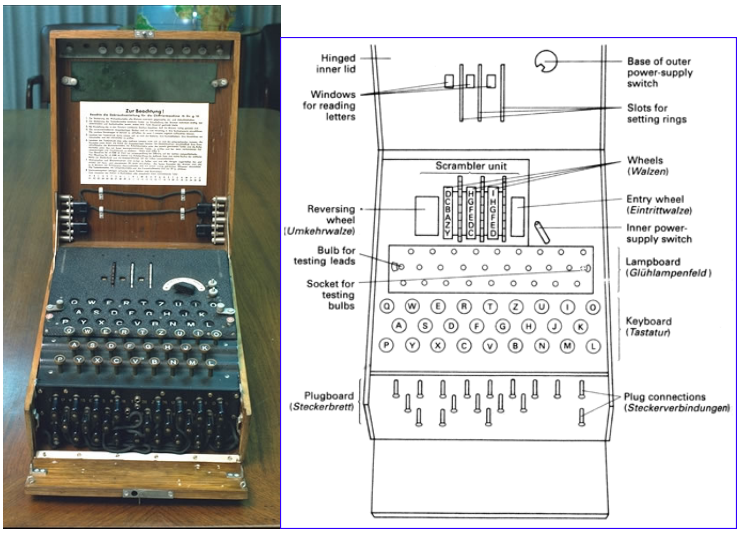


Figure 5.24: An Enigma machine and schematic

1. Which wheels were selected: in 1938 the Germans increased the number of available wheels to five, so that the three being used were chosen from these. In 1940 it was confirmed the Enigma being used by the *Kriegsmarine* (German Navy) for U-boat communications had a total of eight wheels to choose from and in 1941, it was modified to take a fourth wheel

2. The order in which the wheels are installed

3. The initial position of each of the wheels

4. The plug setting on the plugboard (Steckerbrett)

In 1938, Marian Rejewsky and his group, working for Poland's Biuro Szyfrów (Cipher Office), completed the first "bomba", a machine using electromechanical digital logic for trying out combinations of letters to solve the Germans' Enigma cipher.

Figure 5.25: Marian Rejewsky (1905-80)

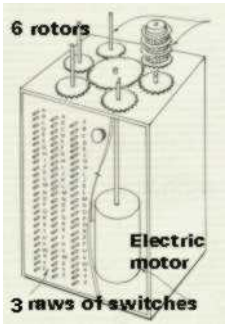


Figure 5.26: The Polish bomba

The bomba contained its own set of rotors like the Enigma's, and its function was to determine, through a combination of logic with an exhaustive search of rotor positions, whether a particular short piece of guessed plaintext and a particular piece of encrypted text could correspond. If the plaintext was correctly guessed, then the key could be derived from the bomba results, and not only the rest of that message, but all others using the same key could then be decrypted. And if it wasn't, then the same guess would be tried against other messages.[14]

During World War II, Turing was at Bletchley Park, the British government's code-breaking establishment. Turing and fellow Cambridge mathematician Gordon Welchman designed the Bombe (later referred to as the Turing Bombe), an

[14]But the following month, the Germans added a selection of additional rotors to their Enigma machines. The Poles, not having the resources to build more bombas, in July 1939 turned over all their discoveries to the British and the French.

Figure 5.27: Gordon Welchman

electromechanical device for breaking the code of the German 'Enigma' encryption machines. Despite the confusion offered by movies such as 'Enigma' and numerous web sites, the first electronic computer (see below) was not built to break the Enigma codes.

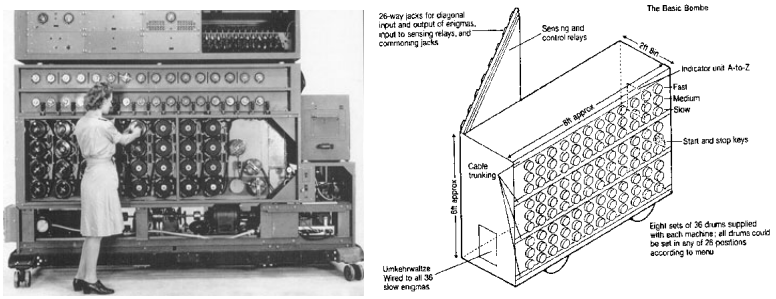


Figure 5.28: A Turing Bombe and operator (note sensible shoes) and schematic diagram

However, Turing's (and others') work at Bletchley certainly effected the conduct of the war. During a three-month period after the *Kriegsmarine* (German navy) strengthened their U-boat Enigma codes, Allied shipping losses tripled in the Atlantic. It has been suggested that the work carried out at Bletchley, while it may not have won the war in Europe, certainly shortened it by months, if not years.

There is currently a project under way at Bletchley Park to reconstruct a bombe. Alan Turing's theoretical work before the war has had a lasting and profound impact on computer science and Turing is arguably its greatest single figure. In 2002, the BBC conducted a poll to discover the 100 Greatest Britons

of all time. Alan Turing was voted number 21.

5.8.2 More Electromechanical developments

Figure 5.29: Howard Aiken

In 1943 the ASCC Mark 1 (also known as the "Harvard Mark I") was finally ready – work had begun on it in 1931. This electromechanical machine, designed and built by Howard H. Aiken (1900-73) and his IBM-backed team at Harvard University, Cambridge, MA, was the first programmable calculator to be widely known: it has been suggested that Aiken is to Zuse as Pascal to Schickard.

Figure 5.30: The Harvard Mark 1

The machine was 51 feet long, weighed 5 tons, and incorporated 750,000 parts. It included 72 accumulators, each incorporating its own arithmetic unit as well as a mechanical register with a capacity of 23 digits plus sign. The arithmetic was fixed-point, with a plugboard setting determining the number of decimal places. I/O facilities included card readers, a card punch, paper tape readers, and typewriters. There were 60 sets of rotary switches, each of which could be used as a constant register – a sort of mechanical read-only memory.

An addition took 1/3 second, and a multiplication, 1 second. The program was read from one paper tape; data could be read from the other tapes, or the card readers, or from the constant registers. Conditional jumps were not

available. However, in later years the machine was modified to support multiple paper tape readers for the program, with the transfer from one to another being conditional, sort of like a conditional subroutine call. Another addition allowed the provision of plugboard-wired subroutines callable from the tape.

The Mark 1 was in use for over fifteen years – long into the electronic computer age.

5.8.2.1 Bletchley goes Electronic – the First Computer

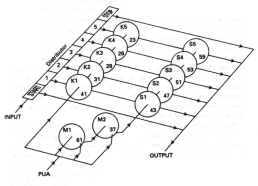


Figure 5.31: Schematic of Tunny the Fish emulator

In addition to the Enigma machines, used largely for tactical messages, the German employed an electronic encryption machine, used for strategic messages. This was the so-called Fish system; it used bit-level manipulations rather than permutations of the alphabet.

In early 1943, the "Heath Robinson" a prototype machine for breaking the new ciphers was built. The machine, designed and built by Max Newman, Wynn-Williams, and their team at Bletchley Park, used a combination of electronics and relay logic. It read data optically at 2,000 characters per second from 2 closed loops of paper tape, each typically about 1,000 characters long.[15]

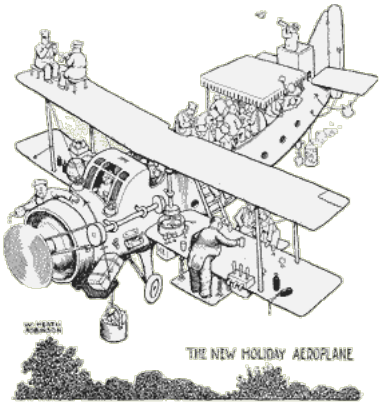


Figure 5.32: A typical Heath Robinson cartoon

[15]Newman had taught Turing at Cambridge, and had been the first person to see a draft of Turing's 1937 paper.

Heath Robinson is the name of a British cartoonist known for drawings of comical machines, like the American Rube Goldberg, whose work came later. Two later machines in the series were be named for London department stores: "Peter Robinson" and "Robinson and Cleaver". The Robinsons used paper tape to contain the messages and keys on which they were working.

Unfortunately synchronising two paper tapes was very difficult and it occurred to Tommy Flowers that it would be more reliable if the wheel patterns were stored in the machine's memory. This machine, named Colossus, has strong claims to be the world's first electronic digital computer (although not stored-program).

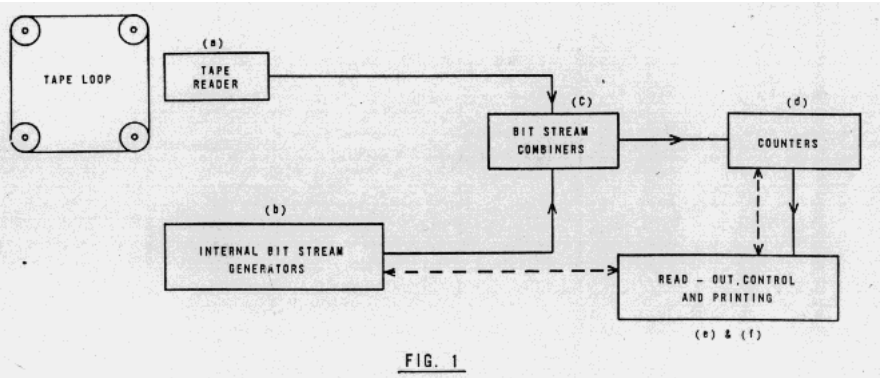


Figure 5.33: Probably the only remaining Colossus design drawing

Colossus was designed by Tom Flowers (1905-98), Sid Broadhurst and W.W Chandler and built (from off-the-shelf parts) in great secrecy at the Post Office Research Establishment, Dollis Hill, North London; only these three men saw the complete plans. This was a full-scale successor to the "Robinson" series machines and was entirely electronic. Colossus I incorporated around 1,500 vacuum tubes, its successor Colossus II used around 2,500 vacuum tubes for logic. It had 5 paper tape loop readers, each working at 5,000 characters per second. The Colossuses were used for "wheel-setting" and (later) "wheel-breaking" (Donald Michie realised how to program this) of the Fish intercepts, which provided valuable strategic (as opposed to Enigma's tactical) information.

Ten Colossuses were eventually built; eight were destroyed after the war to maintain secrecy, the other two being moved to Cheltenham with the retitled GCHQ, they were decommissioned in 1960 at which time the original engineering drawings were burned. Turing did not work directly on the machines – he was in Washington, DC, when Colossus was designed. However, there is no doubt that its designers were inspired by his 1937 paper and the Turing Machine concept.

In the 1990s Bletchley Park became a museum, and in 1996 a replica Colossus was completed there. A Pentium was programmed to perform the same task and Colossus was found to be just as fast. The reason that the Colossus has been so little publicised is that all knowledge of it was classified until the 1970s

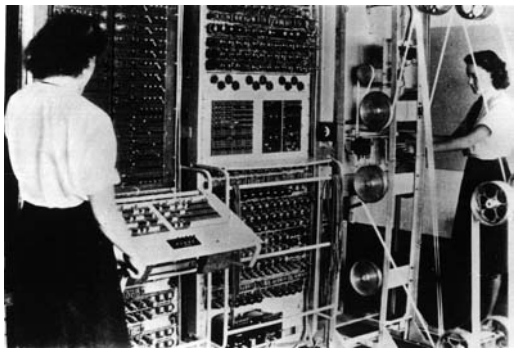


Figure 5.34: The original Colossus in action

– indeed, any information about Bletchley Park was classified.

In fact, even today there are some aspects of Bletchley Park which are still classified.

5.8.3 The other "First Computer" – ENIAC

In April 1943 John W. Mauchly (pronounced Mawkly; 1907-80), J. Presper Eckert (1919-95), and John Brainerd at the Moore School of Electrical Engineering, of the University of Pennsylvania, Philadelphia, wrote a "Report on an Electronic Diff. Analyzer" for the US Army's Ballistics Research Lab. The abbreviation "Diff." was intended to reflect the fact that the proposed machine, eventually named the ENIAC ("Electronic Numerator, Integrator, Analyzer, and Computer"; some sources omit "Analyzer" or have "Calculator" as the last word), was designed to use differences to compute digitally the same results that a differential analyzer would compute by analog means.

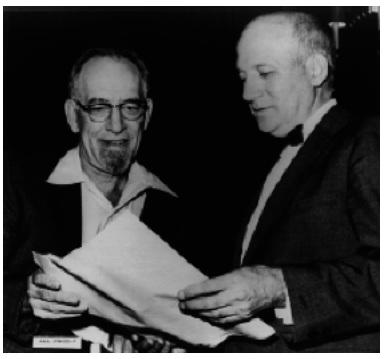


Figure 5.35: Eckert and Mauchly

The BRL, which had a great interest in calculating shell trajectories to produce gun aiming tables, accepted the proposal and work on the ENIAC

began in secret. Mauchly and Eckert and their team at the Moore School finally completed the ENIAC, in November 1945 – too late for the war. The total cost of $486,800 far exceeded the original budget of $150,000 (problems that Eckert and Mauchly encountered again on later projects), but it worked.

The ENIAC's architecture resembled that of the Harvard Mark I, but its components were entirely electronic, incorporating 17,468 vacuum tubes and more than 80,000 other components. The machine weighed 30 tons, covered about 1,000 square feet of floor, and consumed somewhere between 130 and 174 kilowatts of electricity (sources differ). Many of the modules were made to plug into the mainframe, to shorten the repair time when a tube or other component failed. The cost and downtime were further reduced by using circuits designed to work even if their components were off-specification, and wire of the type least preferred by hungry mice in experiments.

Figure 5.36: Betty Jennings (Mrs. Bryant) and Frances Bilas (Mrs. Spence) operating the ENIAC (note more sensible shoes)

The machine incorporated 20 accumulators (the original plan was for 4). The accumulators and other units were all connected by several data buses, and a set of "program lines" for synchronisation. Each accumulator stored a 10-digit number, using 10 bits (tubes) to represent each digit, plus a sign bit, and also incorporated circuits to add a number from a bus to the stored number, and to transmit the stored number or its complement to a bus. A separate unit could perform multiplication (in about 3 milliseconds), while another did division and square roots; the inputs and outputs for both these units used the buses. There were constant registers, as on the Harvard Mark I: 104 12-digit registers forming an array called the "function table". 100 of these registers were directly addressable by a 2-digit number from a bus (the others were used for interpolations).

Finally, a card reader was available to input data values, and there was a card punch for output. The program was set up on a plugboard – this was considered reasonable since the same or similar program would generally be used for weeks at a time. For example, connecting certain sockets would cause accumulator 1 to transmit its contents onto data bus 1 when a pulse arrived on program line 1; meanwhile several accumulators could be adding the value from

that data bus to their stored value, while others could be working independently. The program lines were pulsed under the control of a master unit, which could perform iterations. The ENIAC's clock speed was 100 kHz.

Mauchly and Eckert applied for a patent. The university disputed this at first, but then settled. The patent was finally granted in 1964, but was overturned in 1973, in part because of the previous work by Atanasoff, whom Mauchly had visited in June 1941. (This was the infamous "First Computer" case discussed above.)

In February 1946, ENIAC was revealed to the public, thus beginning the myth that the USA had scored a first in matters computational – a myth which, to this day, dies hard. A panel of flashing lights was added to the machine to show reporters how fast it was running. Hollywood apparently took note.

In June 1943 John von Neumann (1903-57) had joined the ENIAC team, and drafted a report describing the future computer design called the "EDVAC" ("Electronic Discrete Variable Automatic Computer" (!)) – although no machine of this name appears ever to have been built.

This was the first detailed description of the design of a stored-program computer, and gave rise to the term "von Neumann computer". The first draft of the report failed to credit other team members such as Eckert and Mauchly; when this version was widely circulated, von Neumann got somewhat too much credit for the design. The final version corrected the oversight, but too late.

In July and August 1946 The Moore School gave a course on "Theory and Techniques for Design of Electronic Computers"; lectures were given by Eckert, Mauchly, Stibitz, von Neumann, and Aiken among others.

The course led to several other projects being started.

5.8.4 The Harvard Mark II and the first 'bug'

In July 1947, the Mark II became operational. Built by Aiken and his team, this was a large programmable calculator using relays both for its 50 floating-point registers and for the arithmetic unit, 13,000 of them in all.

In September a moth (?-1947) made the mistake of flying into the Harvard Mark II. A whimsical technician made the logbook entry "first actual case of bug being found", and annotated it by taping down the remains of the moth. The term "bug" had, of course, already been in use for some time, which was the point.

Grace Murray Hopper (1906-92), a programmer on the machine, told the story so many times in later years that people came to think she had found the moth herself or even invented the term 'bug' for a programming failure. She does, however, appear to have been the first person to use the term 'debugging'.

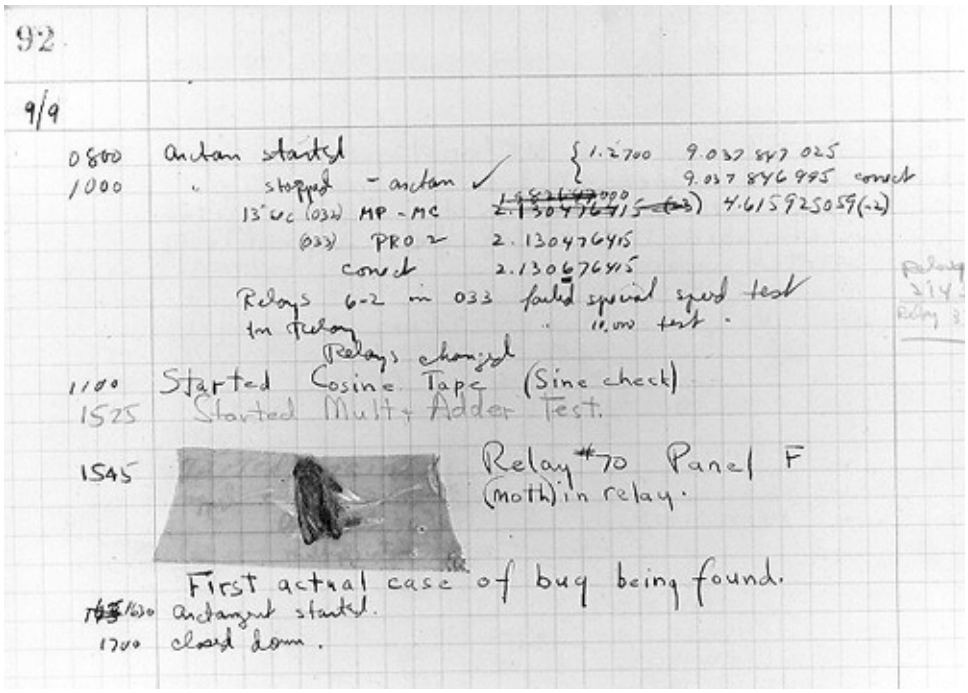


Figure 5.37: The first 'moth', as recorded in the Mark II technician's notebook

5.9 Generation Zero

5.9.1 The First Practical Memory

In December 1946, Freddie C. Williams (1911-77) and Thomas Kilburn (1921-2001), working under Newman at Manchester University, completed a new type of digital memory (possibly from an original suggestion by Presper Eckert), which came to be called the Williams-Kilburn tube or CRT memory.[16]

It used the residual charges left on the screen of a CRT after the electron beam has been fired at it; the bits were read by firing another beam through them and reading the voltage at an electrode beyond the screen, then rewriting. The technique was a little unreliable, but was fast, and also relatively cheap because it could use existing CRT designs; and it was much more compact than any other memory existing at the time.

A further advantage was that if the CRT face was exposed to view, the values in the memory were visible!

[16]Most US sources simply use 'Williams tube'.

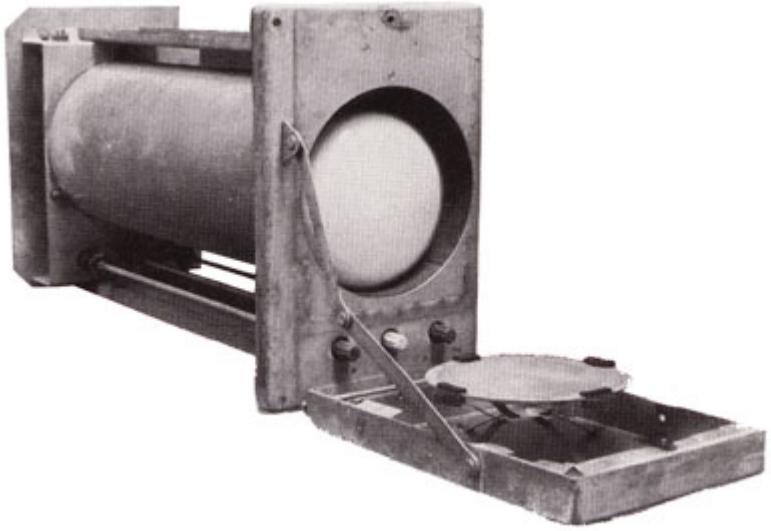


Figure 5.38: The world's first electronic memory – an early Williams-Kilburn tube

5.9.2 The rough before the storm

In the late 1940s several apparently unconnected developments occurred – and one fantastic prediction.

- Frederick Viehe (?-1960), of Los Angeles, applied for a patent on an invention which was to use magnetic core memory, tiny rings of ferrite which held data by using magnetism. The magnetic drum memory was independently invented by several people, and the first examples constructed. Some early machines used drums as main memory rather than secondary memory.

- In the same year Howard Aiken predicted that the United States would need a total of six electronic digital computers.

- In January 1948, IBM's Wallace Eckert (1902-71, no relation to Presper Eckert) and his team, completed the "SSEC" ("Selective Sequence Electronic Calculator"). This technological hybrid had 8 vacuum tube registers, 150 words of relay memory, and 66 paper tape loops storing a total of 20,000 words. The word size was 20 digits, stored in BCD in the registers. As with the Harvard Mark I in its later form, the machine could be switched to read instructions from any of the paper tapes.

  There was also some use of plugboards in its programming. But it could also cache some instructions in memory and read them from there; thus,

in effect, it could operate either as a stored-program computer (with a very small program memory) or not.

Because of this, IBM's point of view is that this was the first computer. Nobody else appears to agree.

5.9.3 The First Computer

In June 1948 Max Williams, Tom Kilburn, and their team completed their prototype computer, which was actually built to test the Williams-Kilburn memory(!) This was the first machine that everyone would call a computer, because it was the first with a true stored-program capability. At this point it had no formal name, though one paper called it the "Small-Scale Experimental Machine"; later the completed machine became known as the "Manchester Mark I", while its initial form at this date was nicknamed the "Baby".

Figure 5.39: Tom Kilburn and Freddie Williams at the console

The Baby had:

- 32-bit word length
- Serial binary arithmetic using 2's complement integers
- A single address format order code
- A random access main store of 32 words, extendable up to 8192 words
- A computing speed of around 1.2 milliseconds per instruction
- The machine's main memory of 32 32-bit words occupied a single Williams tube.[17]

[17] There were others on the machine, but less densely used: one contained only an accumulator.

The machine's programs were initially entered in binary on a keyboard, and the output read in binary from the face of another Williams tube. Later Alan Turing joined the team and devised a primitive form of assembly language, one of several developed at about the same time.

The first program was to find the highest factor of a number. The initial number chosen was small, but within days they were trying the program on 218, and the correct answer was found in 52 minutes, involving about 2.1 million instructions with about 3.5 million store accesses.

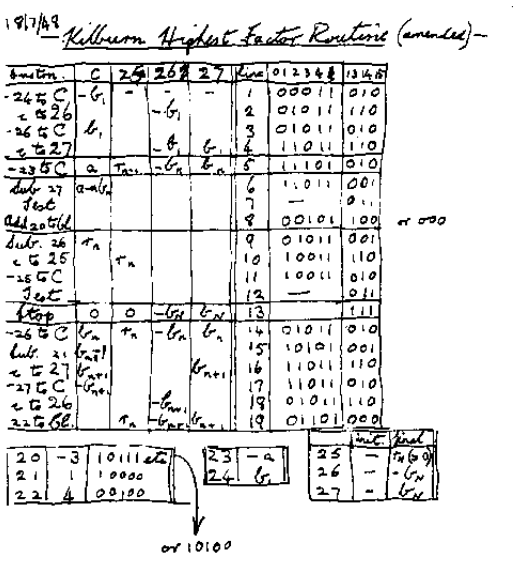


Figure 5.40: The world's first program

A program was laboriously inserted and the start switch pressed. Immediately the spots on the display tube entered a mad dance. In early trials it was a dance of death leading to no useful result, and what was even worse, without yielding any clue as to what was wrong. But one day it stopped, and there, shining brightly in the expected place, was the expected answer. It was a moment to remember. This was in June 1948, and nothing was ever the same again.

Freddie Williams

...the most exciting time was June 1948, when the first machine worked. Without Question. Nothing could ever compare with that.

Tom Kilburn, 1992.

In the 1990s a replica of the Baby was constructed for the 50th anniversary of the running of the world's first stored program. A competition was held for programs (several emulators were available) and the winners invited to Manchester and their programs run on the reconstruction.

5.9.4 Liquid Memory

The Manchester BABY was built, as has been stated, to test the Williams (-Kilburn) Tube memory system, described above. In the late 1940s and early 1950s there were two memory technologies in use, the Williams tube and the Mercury Delay Line.

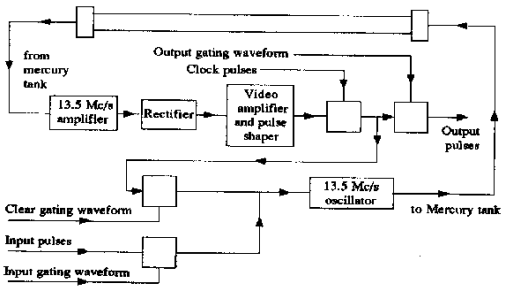


Figure 5.41: Schematic of a mercury delay line

In mercury delay line memories an electrical signal was transformed into an acoustic signal via the well-known piezo-electric effect possessed by some crystals. The sound was carried in a column of mercury where it travelled at a velocity of about 1,500 m/sec, eventually arriving at the far end where it was transformed back into an electrical signal. This received electrical signal could then be re-constituted, re-timed and re-launched back into the column of mercury. The electrical signal was thus "stored" by the delay line but only available at one delay intervals.

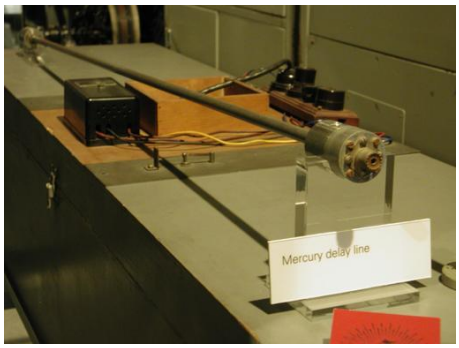


Figure 5.42: A mercury delay line

A sequential bit pattern corresponding to the contents of adjacent words in a modern memory could be stored, but a particular word could only be accessed as its bits emerged from the far end of the delay line. Pulse intervals of one microsecond were used. New data could be entered by substituting the new bit pattern for the old pattern as it emerged from the delay line.

Mercury was and is expensive; while at the National Physical Laboratory Alan Turing calculated that ethyl alcohol could also be used and be nearly as efficient. Colleagues accused him of wanting to use gin in the memory tanks.

5.9.5 The Second Computer

One of the attendees at the 1946 Moore School course was Maurice Wilkes, a Cambridge physicist who had worked on radar during World War II. In 1945 he was appointed Director of the University of Cambridge Mathematical Laboratory. Having borrowed a prepublication copy of the EDVAC report (which he had to read overnight in order to return it the next day, as promised) he decided that this was the way to proceed. In order to provide a computational facility to the university he started work on the EDSAC.

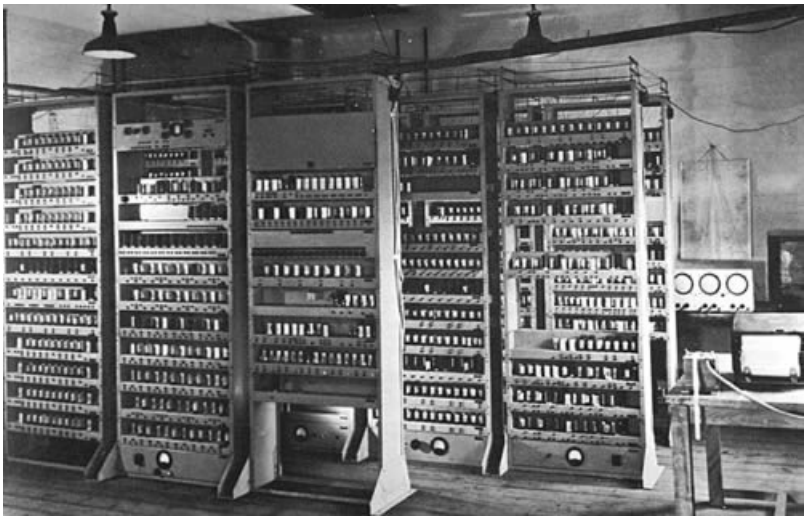


Figure 5.43: The EDSAC

The EDSAC ("Electronic Delay Storage Automatic Computer") which used mercury delay line storage rather than Williams Tubes, ran its first program on May 6, 1949. Sixteen mercury tanks provided a total of 256 35-bit words of memory in the initial version. Each mercury-filled tube or 'tank' was about 5 ft long and stored 576 binary digits. Eventually, the main store consisted of 32 such tubes, with additional tubes acting as central registers within the processor.

Unlike the Manchester Baby, it was not intended as a prototype, but as a production system.

The EDSAC was the first such and therefore has claims to be the world's first (non-prototype) computer. The time taken to perform an addition instruction was 1.4 milliseconds. Input was via a 5-bit electro-mechanical paper tape reader and output was to a teleprinter. EDSAC contained about 3000 thermionic valves (tubes) and filled a large room.

Figure 5.44: Maurice Wilkes and the Mercury delay lines

The financing of the project was via normal University research channels, plus a (for those days) sizeable donation of £2500 from J. Lyons & Co. Ltd, whose primary business enterprises were corner tea shops and cakes.

5.9.6 The First US Computer

In September 1949 the USA finally got a computer to work. This was Eckert and Mauchly's BINAC a dual-processor design intended as a first step towards in-flight computers.[18]

Each of the two BINAC processors had 700 tubes and 512 31-bit words of memory. It had sixteen instructions, each 15 bits long. The BINAC was not as reliable as it might have been – according to some at Northrop, to whom it was delivered in August 1949 (fifteen months behind the original, hopelessly optimistic schedule), it never worked properly once installed. This has been attributed to the fact that Eckert and Mauchly's minds were probably elsewhere: they were still working on their UNIVAC (Universal Automatic Computer).

BINAC ran its first program in March 1949, but it was not a completely finished machine at the time. The price paid for BINAC was $100,000 – but it cost $278,000 to build. Eckert and Mauchly needed to control costs.

[18] The name, incidentally, had nothing to do with the fact that there were two CPUs, rather a reference to using binary arithmetic, where the ENIAC had used decimal.

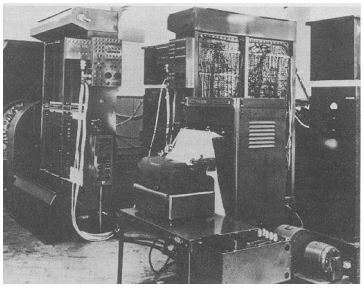


Figure 5.45: The BINAC

BINAC was a bit serial binary computer with a 512 word acoustic mercury delay line memory [the memory can be seen on the left of the photograph] divided into 16 channels each holding 32 words of 31 bits with an additional 11 bit space between words to allow for circuit delays in switching. The clock rate was 4.25MHz which yielded a word time of about 10 microseconds. The actual instruction execution rate was dominated by the access time for instructions and data and would have averaged about 3000-4000 instructions per second, unless minimum latency programming was employed.

Each BINAC word held two instructions. Each instruction had a five bit operation code and a three octal digit address. All operands were 31 bit words. Arithmetic was two's complement and there were single bit arithmetic right and left shift instructions as well as addition, subtraction, multiplication and division. There were no logical instructions and no subroutine calls. Jump on negative was the only conditional instruction.

Description of the BINAC from *Annals of the History of Computing*

The input/output for the BINAC was octal. The instructions were absolute machine language and since it was a serial access memory, the trick was to have the data for the instruction follow the instruction far enough behind that it can be acted on by the instruction. An engineer at Northrop converted an IBM 010 keypunch (a small 11 key unit, digits 0 - 9 and a space) to put in three binary bits on tape for each octal digit punched. After processing the input, the output came out in octal digits. All of the data had to be converted from decimal to octal and back since there wasn't room in memory for a conversion subroutine. The most successful run we made was on a deicing problem for an airplane. Two operators on electric calculators worked for 6 months computing steps in resolving the differential equations. The BINAC did these steps and completed the calculations in 15 minutes. The reduction steps

could be verified by the output from the two operators, thus adding to the confidence in the results.

The idea behind BINAC was to have all operations checked by running two sections of the computer independently and comparing each step on a high speed bus. Northrop engineers went back to Philadelphia for the acceptance tests. After the BINAC was shipped to Hawthorne, it sat out under a tarp for 6 months before it was assembled in its air-conditioned room. The poor engineers were constantly working to get BINAC to run. One side would be running while they worked on the other side. The two sides never worked together as long as I was there. Doing all arithmetic in octal sure cause havoc with your check book balance if you weren't careful.

Roger Mills

5.9.7 The Fourth – and First Antipodean – Computer

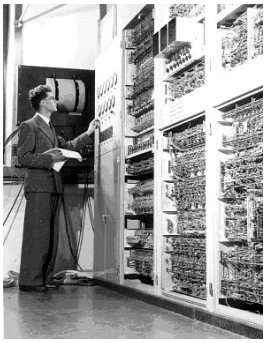


Figure 5.46: Trevor Pearcey and the CSIR

In November 1949, Australia's first computer ran its first program. The CSIR Mark 1 was built for the Australian Council for Scientific and Industrial Research, whence the acronym. It was later renamed CSIRAC (Council for Scientific and Industrial Research Automatic Computer).

Designed and built by a team led by Trevor Pearcey and Maston Beard, the machine used mercury delay lines and had a typical memory size of 768 20-bit words.

One of the most remarkable facts about the CSIR is that it was the first computer in the world to play music. In 1951, Geoff Hill, the CSIR's first programmer, who came from a highly musical family, programmed the machine to play popular melodies – this was long before any theory of digital sound creation. The computer's first public performance was of the tune "Colonel Bogey".

In 1955 the machine was moved to Melbourne where Professor Thomas Cherry (a mathematician) created a system whereby anyone who understood

Figure 5.47: Trevor Pearcey and Maston Beard

musical notation could create a punched paper tape from which CSIRAC (as it was now called) could play the music.

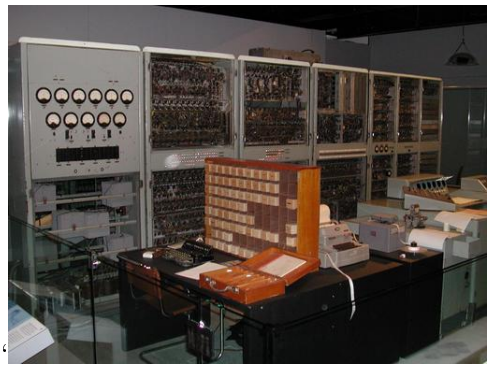


Figure 5.48: The CSIRAC-1 on display in Melbourne (2004)

The CSIRAC-1, as it was still later renamed, is still on display in Melbourne and is the oldest surviving computer in the world.

5.10 The First Generation

5.10.1 The Ferranti Mark 1

In Manchester the team had not allowed the grass to grow beneath their feet. The "Baby" had proved the practicality of both the Williams tube memory system and the stored" program concept.

In the months after the Baby's debut, it had been extended – in two or three incremental stages – into what was known as the Manchester Mark 1. The extent of the innovation taking place can be gauged by the fact that in the period 1948-50, some 42 patents were taken out based on the work in Manchester.

Figure 5.49: The Manchester Mark 1

The Mark I contained 128 40-bit words of Williams tube storage, backed up by 1024 words on a slower magnetic drum store. An addition instruction was performed in 1.8 milliseconds. Input and output was via a 5-bit paper tape reader and a teleprinter. The Mark I was mostly built out of war surplus thermionic valves supplied by TRE, the UK radar research centre.

In case you are wondering what it was like to program the Mark I, here is a description by Alan Turing's assistant[19] the delightfully-named Cicely Popplewell:

> Starting in the machine room you alerted the engineer and then used the hand switches to bring down and enter the input program. A bright band on the monitor tube indicated that the waiting loop had been entered. When this had been achieved, you ran upstairs and put the tape in the tape reader and returned to the machine room. If the machine was still obeying the input loop, you called to the engineer to switch on the writing current, and cleared the accumulator (allowing the control to emerge from the loop). With luck the tape was read. As soone as the pattern on the monitor

[19] Two "assistants" were assigned to Turing during his Manchester years. In fact – perhaps because they were both female and he was never at his best with women – he scarcely spoke to them, leaving them largely to their own devices.

> showed that input was ended the engineer switched off the write current to the drum. Programs whcih wrote to the drum during the execution phase were considered very daring. As every vehicle that drove past was a potential source of spurious digits, it usually took many attempts to get a tape in – each attempt needing another trip to the tape room.[20]

You might like to reflect on this the next time your own code fails. At least the likelihood that it was brought down by passing traffic is very remote these days!

In October 1948 the Chief Government Scientist was paying a visit to Professor P.M.S. Blackett, who suggested paying a visit to see the computer working. So impressed was the CGS that the UK government took out a contract with the electronics company Ferranti to built a production version of the machine "to Professor Williams' specification". The contract ran from November 1948 for five years.

The commercially-engineered version of the Manchester machine was called the Ferranti Mark I. At the insistence of Alan Turing, by now working at Manchester University, the machine included a true random number generator.

Here is one consequence: a "love" letter created on the Mark 1 (calling itself MUC – Manchester University Computer) using the random number facility:

> Darling Sweetheart, You are my avid fellow feeling. My affection curiously clings to your passionate wish. My liking yearns to your heart. You are my wistful sympathy: my tender liking. Yours beautifully, M. U. C.

Figure 5.50: At the Mark 1 console: Brian Pollard, Keith Lonsdale, Alan Turing

[20] Quoted in Andrew Hodges excellent biography: *Alan Turing, the Enigma*.

The first Ferranti Mark 1 off the production line was delivered to Manchester University in February 1951. The Ferranti Mark 1 was thus the first computer in the world offered for sale rather than built to commission. The second Mark 1 was delivered to the University of Toronto, where it was used, among other things, to assist in the design of the St. Lawrence Seaway.[21]

The final specification of the Ferranti Mark 1 included:

- Store organised in 20-bit addressable "line"s, an instruction taking one line and a number two consecutive lines.
- Serial 40-bit arithmetic, with hardware add, subtract and multiply (with a double-length accumulator) and logical instructions 8 modifier registers (called B-lines) for modifying addresses in instructions; simple B-line arithmetic and tests.
- Single-address format order code with about 50 function codes 8 pages of random access main store (one CRT per 64 * 20-bit line page).
- 512 page capacity drum backing store, 2 pages per track, about 30 milliseconds revolution time
- Standard instruction time: 1.2 ms multiplication 2.16 ms.
- Peripheral Instructions : read and punch a line of 5-hole paper tape; transfer a given page (or track) on drum to/from a given Williams-Kilburn Tube "page" (or page pair) in store.

The B-line registers, which were built into the original BABY machine, would today be referred to as index registers. Other computers began to use them from about 1955 onwards.

[21] William Kahan, co-author of the original Intel floating point design which essentially became IEEE 754, was an undergraduate at UofT and learned to program on the Mark 1.

5.10.2 The First US Computer for Sale

Figure 5.51: The Univac-1

Eckert and Mauchly, probably discouraged by the cost overruns of the BINAC, sold their company to Remington Rand, but remained with the company to finish the UNIVAC (Universal Automatic Computer). The machine was 25 feet by 50 feet in length, contained 5,600 tubes, 18,000 crystal diodes, and 300 relays. It utilised serial circuitry, 2.25 MHz bit rate, and had an internal storage capacity of 1,000 words or 12,000 characters. It utilised a Mercury delay line, magnetic tape, and typewriter output.

Figure 5.52: Operating the UNIVAC at Lawrence Livermore Laboratories

The UNIVAC was used for general purpose computing with large amounts of input and output. Power consumption was about 120 kva. Its reported processing speed was 0.525 milliseconds for arithmetic functions, 2.15 milliseconds for multiplication and 3.9 Milliseconds for division. The UNIVAC was also the first computer to come equipped with a magnetic tape unit and was the first computer to use buffer memory.

The first UNIVAC I was delivered on June 14, 1951 to the US Census Bureau (Hollerith anyone?)

5.10.3 The First Business Computer

Figure 5.53: The LEO I

The J. Lyons company – bakers of cakes and dispensers of refreshments in their corner teashops throughout Britain – despite its cosy public image, was determinedly self-reliant: the company built its own factory equipment, its own trucks, grew its own tea, etc. In 1947, O.W. Standingford and T.R Thompson, assistant comptroller and chief assistant comptroller of the company, visited the US to see what was going on and to judge whether the new computers, which to this point had been used exclusively for mathematical calculations (as far as most people were aware – Bletchley Park and Colossus were still classified), "were capable of being used in commercial offices."

Encouraged by Professor H.H. Goldstine of Princeton, the pair contacted Douglas Hartree and Maurice Wilkes of Cambridge University. "We found them both keenly interested in our proposals for a commercial machine and prepared to make their knowledge and advice available."

The board's ultimate decision was to assist in funding the EDSAC but also to go ahead and construct their own machine, which was named the LEO (Lyons Electronic Office). The decision to build their own was contingent on the EDSAC actually working. When it ran its first program, in May 1949, Lyons was actually holding a board meeting. The phone call from Cambridge came though, telling them that EDSAC had compiled a table of prime numbers.

The decision to go ahead with building the LEO was made within five minutes.

Figure 5.54: John Pinkerton (1919-97)

The LEO design team, which essentially took Maurice Wilkes's EDSAC and turned it into a commercial machine, was led by John Pinkerton, whose design philosophy was "... the simpler the conception and the design of any item of calculating equipment, the better it will be understood by the operators who use it and the engineers who maintain it."[22]

The LEO I ran its first program – which calculated (among other things) stock value and balances for each of its factories – on 17 November 1951. The machine "resembled a battleship" which occupied 5,000 sq. ft. of floor space and employed 6,000 electronic tubes deployed in 21 tall racks. To this date, no more than 20 people – including clerical assistants – had been involved with the design, building and programming of the LEO. Another important first for the LEO came on 12, February 1954 when it ran the company payroll, printing checks for 1670 employees directly. That same year, the LEO II was designed and the board decided to build two of them. More remarkably, at the board meeting of June 30, 1954, the board declared: "We should let it be known that we are prepared to build other Mark IIs for sale or hire."

5.10.4 The First Compiler

In 1952 Grace Hopper, by now working for Remington Rand, invented the modern concept of the compiler. (Needs more)

5.10.5 The First IBM Computer

In 1952 IBM announced its "Defense Calculator", later renamed the IBM 701, which entered production at Poughkeepsie, New York.

[22] In 2000, the Institution of Engineering and Technology inaugurated the annual John Pinkerton Lecture. The 2007 lecture will be given by Sir Tim Berners-Lee.

Interestingly, like Wilkes, Pinkerton had spent the second wold war engaged in radar research.

Figure 5.55: Thomas Watson Senior at the console of the IBM 701

The first one was delivered in March 1953; 19 were sold altogether. The machine was available with 2,048 or 4,096 36-bit words of CRT memory; it did 2,200 multiplications per second.

IBM stayed out of the computer market for some time because its president, Thomas Watson Sr., didn't want the company competing against its own business machines. His son and eventual successor, Thomas Jr., disagreed, and realized that if it was the US military that wanted to buy a computer, Thomas Sr. would not say no to them.

The 701 was (apart from the SSEC) the first IBM computer and its descendants, the 704, 709 and 7090 dominated scientific computing for a decade.

For more information visit http://www.columbia.edu/acis/history/701.html

5.11 The Rest of History

5.11.1 Hardware Trends

By the early 1950s much of the future of computing–certainly in terms of hardware–had effectively been foretold. The last half-century has seen several trends in the evolution of hardware: smaller, faster, cheaper.

The switch (1958) to transistors brought the first dramatic improvements in reliability, power consumption, heat generation and physical size.

With the coming of Integrated Circuits (ICs) in the early 1960s a trend was begun which continues to this day and is known as Moore's Law.

First propounded c1965 by Intel founder Gordon Moore – the "Law" says that there will be a 60% annual increase in the number of transistors on a chip[23]. Figure 5.56 shows the original graph drawn by Moore in 1965.

Despite repeated warnings that "there must be a physical limit" Moore's Law has held for over four decades. Figure 5.57 shows how Moore's predictions have matched reality during that period – the example CPUs are all Intel, as

[23] Or approximately 100% every 18 months.

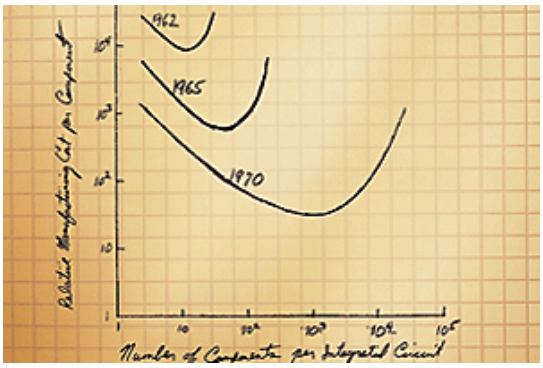


Figure 5.56: Moore's Original (1965) graph

this graph comes from the Intel web site. The law has a significant effect on CPU speeds, memory sizes etc.

Visit http://www.intel.com/technology/mooreslaw/index.htm to find out more.

5.11.2 Software Trends

Software is another matter – it has tended to become larger and more complex.

Most of the post-1954 history is readily available, particularly on the Internet. The rest of this chapter merely points out some of the highlights with references for further reading.

5.12 Hardware Since 1955

5.12.1 The First Hard Disk Drive

Although magnetic drums had been employed since the late 1940s – the Manchester machine had one in 1949 – it was not until 1956 that the first hard disk drive appeared, in the form of the IBM RAMAC 305.

The RAMAC305 was the world's first computer to feature a hard disk drive, although by today's standard the drive itself was somewhat less than high performance:

- Capacity 5MB (7-bit characters).
- 50 stacked metallic platters, spaced .4 inches apart, each was 24 inches in diameter, the usable recording band was 5 inches across.
- Access time 800 ms.
- Weight, approximately 1 ton (1100Kg).

  For more information about the RAMAC and other early disk drives see http://www.magneticdiskheritagecenter.org/

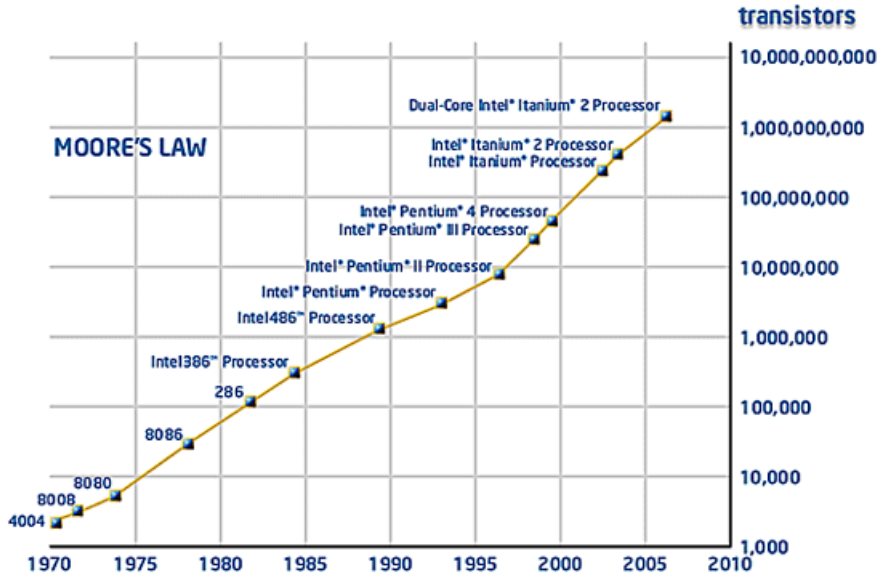


Figure 5.57: Moore's Law as it stands today

5.12.2 The First Transistorised Computer

In 1956 Electrotechnical Laboratory in Japan built what was probably the world's first fully-transistorised computer, the ETL Mark III (the 1955 Mark II had employed relays).[24]

See http://www.ipsj.or.jp/katsudou/museum/index_e.html for more on this and other early Japanese computers.

5.12.3 The First Integrated Circuit

In 1958 Jack Kilby of Texas Instruments constructed the first Integrated Circuit (IC)

- He did this during the summer when most of his colleagues were on vacation – as a recent hire, Kilby had not accumulated sufficient vacation time.
- In 2000 Kilby shared the Nobel prize for Physics for his work on the IC.

  See http://www.icknowledge.com/history/history.html for more on the history of the IC.

[24]The Mark II was Japan's second electronic computer; the first was Fiju Photo Film's FUJIC, a vacuum tube based machine.

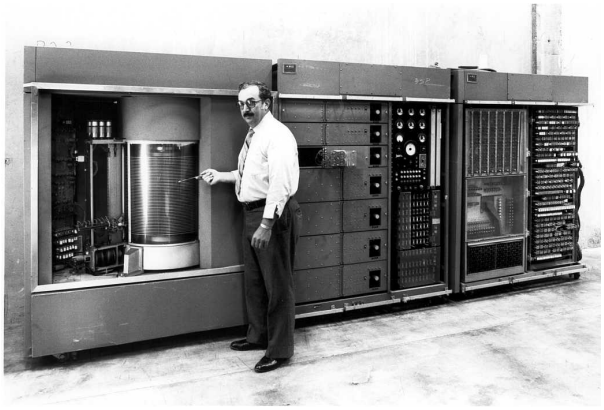


Figure 5.58: The IBM RAMAC 305

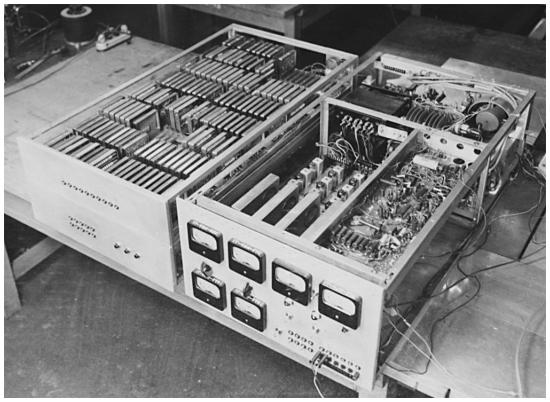


Figure 5.59: The ETL Mark III, the world's first transistorised computer.

5.12.4 Transistorised Computers Make A Public Appearance

In September 1958 NEC completed the NEAC-2201 – based on the ETL Mark IV, the successor to the Mark III – which was demonstrated at the UNESCO-sponsored AUTOMATH Exhibitionheld in Paris in June 1959, thus becoming the first transistorised computer to be seen in public.

Several other countries exhibited transistor computers at AUTOMATH, but only the NEC actually worked.

5.12.5 The First Minicomputer

In 1960 Digital Equipment Corporation (DEC) announced the DEC PDP-1, the world's irst minicomputer.

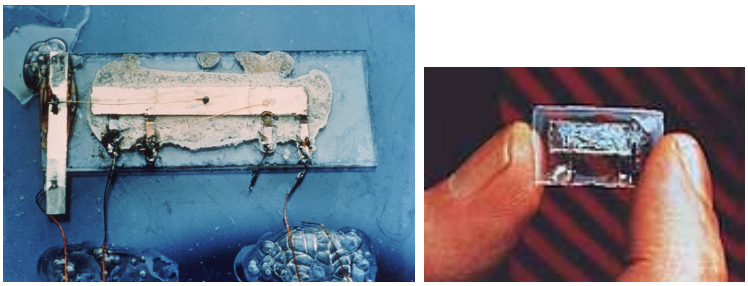


Figure 5.60: The first Integrated Circuit

The machine was designed by Benjamin Curley and had around half the performance of a mainframe, but at 1/10th the price. Although DEC did not sell many PDP-1s, their successors, the PDP-8, PDP-11 and VAX-11/780 made DEC the market leader in the minicomputer field for almost three decades.

5.12.6 The Third Generation

In February 1961 Burroughs announced the B5000, which:

- was the first computer with a hardware *stack*.
- was the first computer with a *tagged-architecture* (i.e. each memory word had three non-data bits which specified what kind of thing was contained in the work – e.g. integer, floating point, character, etc.
- was the first computer designed specifically with high-level languages (ALGOL, COBOL) in mind.
- was the first computer to feature a *Segmented Memory*. This meant that programs could, in principal, have read-only areas of memory and could share them with other programs. It did not, unlike the Atlas (see below) allow programs larger than physical memory to run.
- was arguably the first *Third Generation* computer.[25]
- rented for between $13,000 and $50,000 per month. (And that's 1960s dollars.)

  See http://www.smecc.org/The%20Architecture%20%20of%20the%20Burroughs%20B-5000.htm for more on the B5000 series and its influence on future developments.

[25] According to Peter Denning's classification. See http://community.corest.com/~ek/fis/p175-denning.pdf for more information.

Figure 5.61: The NEAC-2201

Figure 5.62: The DEC PDP-1.

5.12.7 The First Paged Memory

In 1962 Manchester University's ATLAS was completed. Designed by Tom Kilburn at Manchester University, it was built by Ferranti.

ATLAS was:

- the world's first *Demand-Paged Virtual Memory* computer, it could run programs larger than its physical RAM size.
- at the time, the fastest, most sophisticated computer in the world.

  http://www.chilton-computing.org.uk/acl/technology/atlas/overview.htm contains much interesting information about the ATLAS.

Figure 5.63: Advertisement for the Burroughs B5000: *Business Automation*, December 1961

5.12.8 Removable Hard Disks

In October 1962, IBM announced the 1311, a disk drive which employed a removable pack. This meant that disks could be treated like tapes, kept on the shelf until required.

5.12.9 The First "Supercomputer"

In August 1963, Control Data Corporation announced the CDC6600, a 60-bit word machine with significant internal parallelism, designed by Seymour Cray. It remained the world's fastest machine for several years.

Somewhat miffed by this, IBM's Thomas J. Watson asked:

> "Last week, Control Data...announced the 6600 system. I understand that in the laboratory developing the system there are only 34 people including the janitor. Of these, 14 are engineers and 4 are programmers...Contrasting this modest effort with our vast development activities, I fail to understand why we have lost our industry leadership position by letting someone else offer the world's most powerful computer."

Cray's reponse?

> "It seems like Mr. Watson has answered his own question."
>
> Several of Cray's innovations in the 6600 would be adopted by the rest of the industry – sometimes years, even decades, later.

5.12.10 The First Computer Family

On April 7, 1964 IBM announced the Series 360. For the first time customers could buy (or lease) an entry-level machine (360/20) and then upgrade the

Figure 5.64: The ATLAS Computer. Sebastien de Ferranti and Tom Kilburn standing at the console.

hardware without changing software (or so the theory went).

This compatilility throughout the family was achieved by microcoding various different hardware architectures.[26]

IBM's announcement stated that:

> System/360 monthly rentals will range from $2,700 for a basic configuration to $115,000 for a typical large multisystem configuration. Comparable purchase prices range from $133,000 to $5,500,000.[27]

The first models shipped in 1965.

5.12.11 Segmentation, Paging and Rings

- The GE-645, a modified version of the 635 designed specifically for the Multics project, had a segmented, paged memory and a nested set of levels of priority, called "rings"

[26] Microcoding was a technique first described by Maurice Wilkes – yes, he of the EDSAC – in 1951. Essentially the Information Processing Cycle (see chapter 8 for more information on the IPC) is run by a program and only this program understands the underlying hardware. If you like, it is simulating the hardware that the regular instruction set believes it is running on.

[27] Remember, these are mid-1960s $$$.

Figure 5.65: The IBM 1311 Disk Drive (man is holding removable pack).

- Although ultimately dropped by Honeywell (who acquired GE's computer business in 1971) Multics was used as a security testbed by the USAF; hardware modifications suggested by the "Tiger Team" were incorporated into the 6180 follow-on processor.

5.12.12 The Floppy Disk

- The 8 inch diskette was built for IBM by Alan F. Shugart (an ex-employee) in 1970. It was used to distribute firmware updates for mainframes.

5.12.13 The First Microprocessor

- The Intel 4004, commissioned by a Japanese calculator company, and released in 1971, was the first IC complex enough to contain an entire (albeit simple) CPU. The Japanese company did not realise what it had and allowed Intel to buy back the rights.

5.12.14 Winchester '73

- In 1973, IBM introduced the first hermetically sealed hard drive unit, the 3340 disk. This drive had two spindles each with a 30MB capacity. It was named the known as the "Winchester" (after the 30-30 rifle in his closet) by project manager Ken Haughton. All current drives are descendants of this.

Figure 5.66: A dual CDC 6600 installation.

Figure 5.67: Seymour Cray at the CDC6600 Product Announcement.

5.12.15 The First Personal Computer

- The MITS Altair – designed by Ed Roberts and Bill Yates, named after a Star Trek episode, A Voyage to Altair) – was launched in 1975. The kit cost $397 for a 256 byte computer. The I/O consisted of switches and lights.
- Surprisingly they could not meet demand and there are stories of potential customers camping out in their parking lot to ensure that they would get one.

Figure 5.68: Series 360 advertisement.

5.12.16 Optical Disks

- In 1986 Sony launched the first writable (WORM – Write Once, Read Many [times]) optical disk, it used a single 12" platter. Performance became an issue (optical disks were an option with the original NexT computer) and writable optical disks finally became a niche market in the forms of the CD-R, CD-RW and (eventually) DVD-R, DVD+R, DVD-RW, DVD+RW.

5.13 Software

5.13.1 Programming Languages

5.13.1.1 Assembly Language

In the late 1940s and early 1950s all programs were written in assembler.

- It was a distinct step up from programming in binary (or hex or base-32). Among the people who first devised them were Alan Turing (who else?) and Maurice Wilkes.

5.13.1.2 FORTRAN – A Scientific Language

- Between 1954 and 1957,John Backus and a team from IBM designed and implemented the first widely-used high level language, although Backus envisaged it as a "high-level assembler for the IBM 704" and believed that other machines would have their own, if similar, high level languages.

- Four and a half decades later, FORTRAN is still the staple of the scientific programming community.

5.13.1.3 COBOL – The First Business Language

- Based on a number of experimental commercial languages (including Hopper's Flowmatic for Sperry-Rand) the COmmon Business Oriented Language was created at the behest of the US government who had already encountered the maintenance problem: they had to deal with dozens of differing hardware systems and a similar, if not greater, number of programming languages.
- It was designed by the CODASYL COnference on DAta SYsystems Languages) committee between may 1959 and April 1960.
- Predictions that managers will be able to read programs were found to be true – as long as being able to read did not include understanding

5.13.1.4 LISP – The Language of AI

- In 1959, John McCarthy created LISP.
- It was originally intended as the intermediate step towards a more fully-realised expression evaluation language, LISP (LISt Processor) turned out to "have legs" and became the main vehicle of the Artificial Intelligence community.

5.13.1.5 1960 ALGOL – The First Block-Structured Language

- Like the camel, the ALGOrithmic Language was designed by a committee. It was the first "block-structured" language and the first language (after LISP) to support recursion.
- An earlier proposal, dating from 1958, was entitled IAL (International Algorithmic Language) and was never implemented.
- The notation invented by John Backus and Peter Naur for defining the syntax of ALGOL, became known as BNF (Backus-Naur Form) and is still widely used.

5.13.1.6 1963 CPL – An Influential Language

- Designed by Strachey, Barron et. al. this was originally the Cambridge Programming Language, but when it became a collaborative effort with London University, it became the Combined PL.
- CPL was a huge language for the time (like PL/1 it was a Swiss Army Knife of a language) and there were no full compilers until at least 1970.

- Meanwhile a cut-down version, called BCPL (Martin Richards), was implemented and eventually would parent first B, then C and all other C-derived languages (C++, Objective-C, Java, etc, etc)

5.13.1.7 1964 PL/1 – The Final Solution?

Taking features from COBOL, FORTRAN and ALGOL, Programming Language/1 was designed at IBM's Vienna Language Laboratory.

Perhaps the largest single language ever designed, its complexity made writing compilers difficult and few full-scale implementations were ever written.

There were, however, a number of derivatives of PL/1 including PL/V, PL-6 and PL/M.

5.13.1.8 1964 APL – Programming in Greek?

Ken Iverson of IBM develops A Programming Language, which was so terse and symbol-rich that it could do in one line what many another language would take pages to do. It also required a special keyboard.

APL is arguably the second candidate (after assembly languages) for the title of First Write-Only Language.

5.13.1.9 1964 (May 1) BASIC – The First Interactive Language

Developed by John G. Kemeny and Thomas Kurtz at Dartmouth College, was the first language designed to be run from a terminal.

On May 1, two BASIC programs run simultaneously on Dartmouth's GE-225 under the control of DTSS (Dartmouth Timesharing System).

The intention was to design a language so simple that students could write their first programs after only three classes. This proved to be a pessimistic estimate; their students thought the third class a waste of time and it was eliminated

5.13.1.10 1965 ISWIM – The Shape of Things to Come

- Peter Landin's paper "The next 700 Programming Languages" suggests a family of languages called If you See What I Mean, based on Church's Lambda Calculus.
- All modern (non-LISP) functional languages ultimately derive from ISWIM.

5.13.1.11 1964 Simula – The First Object Oriented Language

- Designed by Ole-Johan Dahl and Kristen Nygarad between 1962 and 1964. As the name suggests, Simula was a language originally designed for simulation studies.

5.13.1.12 1970 Pascal – A Teaching Language

- Designed by Niklaus Wirth of the ETH in Zurich, Pascal was designed specifically as a language for teaching. Its success can be seen from the fact that it was still being used for this purpose at many universities and colleges over 20 years later, as was Wirth's Pascal derivative Modula-2.
- The UCSD P-code system made Pascal a "write once, run anywhere" language, like Java a generation later.

5.13.1.13 1970-2 PROLOG – Logic Programming

- Alain Comeraurer at the University of Marseilles invented PROgramming in LOGic. The language rapidly became at least as popular as LISP in the European AI community.

5.13.1.14 1972-80 Smalltalk – Language or Environment?

- Designed by Alan Kay at the Xerox PARC, Smalltalk was a part of the Dynabook project; Smalltalk is both an (O-O) language and a GUI – the first GUI.

5.13.1.15 c1973 ML – A Typed Functional Language

- Designed by Robin Milner at the University of Edinburgh, ML (later SML etc) was inspired by ISWIM and developed as part of the LCF program-proving system.

5.13.1.16 1978-80 Ada – Another US Government Language

- In the late 1970s, the US Department of Defense commissioned a series of competitive language proposals, each addressing an increasingly tight specification (Strawman, Woodman, Tinman, Ironman, Steelman)
- The winning final design was by Jean Ichbiah and a team from CII-Honeywell Bull in France, Ada is a language for concurrent processing and was named after Ada Lovelace (see Babbage above)
- More recent Ada (Ada95) standards have evolved into an O-O language.

5.14 Sytems Software And Applications

5.14.1 Late 1950s The Operating System

Initially users had to submit their jobs to human operators, whoc would completely supervise their running and scheduling. Only one program would be run at a time.

In the late 1950s the first Monitor Systems began to appear, which allowed for primitive control by the system itself and less intervention by the human operator.

5.14.2 Early 1960s Multiprogramming

Early machines had no memory protection hardware, thus it was impossible (or certainly very unsafe) to try to run two programs in memory simultaneously.

The first multiprogramming operating systems used hardware which had simple base and limits registers, enabling multiple programs to timeshare the same processor with protection from each other.

This also involved the CPU being able to operate in two modes: supervisor mode and user mode.

Early machines had no memory protection hardware, thus it was impossible (or certainly very unsafe) to try to run two programs in memory simultaneously.

The first multiprogramming operating systems used hardware which had simple base and limits registers, enabling multiple programs to timeshare the same processor with protection from each other.

This also involved the CPU being able to operate in two modes: supervisor mode and user mode.

5.14.3 Late 1950s/early 1960s Timesharing

The term timesharing has two meanings: one is the simple sharing of CPU time between different programs, the other is applied to systems which do this in order to enable online users to interact with the systems.

Among the first such systems (if not the first) was MIT's CTSS (Corbato et. al) implemented on an IBM 7090. This was also the first system to implement a hierarchical file store (i.e. one with directories – aka folders – and files with them)

Part IV

Hardware

Chapter 6

Data – Bits, Bytes and more

1 Die Welt ist alles, was der Fall ist.

(The world is everything that is the case.)

Ludwig Wittgenstein, *Tractatus Logico-Philosophicus*

6.1 Data Representation

In this chapter we examine how the computer stores information internally. Over the years since the computer was invented a variety of different internal representations have been used. We shall look at the more common ones only.

6.1.1 Bits and Bytes

Although the ENIAC (see Chapter 5, on the history of the computer) used a decimal system to store numbers, it is far easier and more reliable, electonrically, to store everything as "bits" (a contraction of "binary digits"), with each bit able to hold just two values: 1 and 0. These are also sometimes referred to as "ON" and "OFF" (or even "true" and "false").

Within the computer's memory (see Chapter 8, on the basic architecture of the computer and its associated memory system) bits are grouped together into "bytes" (not always, but today invariably eight bits) and bytes are grouped together into larger chunks often called "words" (usually sixteen bits, but not always).

How we can use these individual on/off bits to store basic types of data, such as numbers and characters (letters of the alphabet, punctiation characters, etc.) is the subject of the rest of this chapter.

6.1.2 Binary Numbers

6.1.2.1 Integer Basics

The decimal system which we all learned in school is an example of a positional numbering system, in which the position of a digit affects the value it represents. This might seem obvious, but it was not to the ancient Romans (or Greeks). The Roman numeral for the number 17, for instance, is XVII.

Although this was the conventional sequence of writing it (listing the various numerals in descending order of value) in principle we could as easily write it as IVIX: the X always represented 10, V always 5, I always 1.

But decimal 123 and decimal 321 are clearly not the same number. The reason is because it is a positional system:

```
321
|||--- represents 1*10^0 = 1*1
||---- represents 2*10^1 = 2*10
|------ represents 3*10^2 = 3*100
```

In our decimal system we use a base of 10:- each position as we move to the left represents the next higher power of 10.

Binary numbers are simply numbers represented in base 2, so, in a binary number:

```
1011
||||-- represents 1 * 2^0 = 1 * 1
|||----represents 1 * 21 = 1 * 2
||-----represents 0 * 2^2 = 0 * 4
|------represents 1 * 2^3 = 1 * 8
```

For a total of 11 decimal.

When there is potential confusion about the base of a number – was that 11 decimal or 11 binary? – then the base is often written as a subscript to avoid confusion:- 11_{10} or 11_2.

The smallest positive (or, to be precise, non-negative) binary integer (in 8-bit format) is 00000000, or 0_{10}.

The largest is 11111111 (2^7+2^6+2^5+2^4+2^3+2^2+2^1+2^0or 128+64+32+16+8+4+2+1) = 255 decimal (or 2^8-1).

Coincidence? No.

Consider that the number 1 greater than 11111111 would be 100000000 (which takes 9 bits). This is evidently 2^8, and so we can see that the largest positive number we can represent in n bits is $2^n - 1$. Similarly, the largest value we can get into an n-digit decimal number would be: 10^n-1 (e.g. when n is 3, then the largest value is $999 = 1000 - 1 = 10^3$-1).

6.1.2.2 Conversions

From decimal to binary Simply repeatedly divide the original number by two, noting all the remainders. The answer is the remainders strung together from left to right. Example: what is 106 decimal represented in binary?

```
2| 106
 -----
 2| 53 r 0
  ----
 2| 26 r 1
  ----
 2| 13 r 0
  ----
  2| 6 r 1
   ---
  2| 3 r 0
   ---
  2| 1 r 1
   ---
     0 r 1
```

Reading the remainders from bottom (left) to top (right) we get: 1101010.

6.1.2.3 From binary to decimal

This is even easier.

Simply recall that each bit, reading from right to left, represents one higher power of 2; the rightmost bit represents multiples of 2^0 (i.e. 1), the next multiples of 2^1 (i.e. 2) and so forth.

So 1101010 = 1 * 2^6 + 1 * 2^5 + 1 * 2^3 + 1 * 2^1 (leaving out the zeros) = 64 + 32 + 8 + 2 = 106 decimal.

6.1.2.4 Arithmetic

Binary arithmetic is simple, especially addition. Here is the addition table for binary:

+	0	1
0	0	1
1	1	0_c

Where the subscripted 'c' represents a "carry" to the next position on the left.

Example:

```
 0101 = 5 decimal
+0011 = 3 decimal
 ----
 1000 = 8 decimal
```

Subtraction is similarly easy. Here is the subtraction table for binary:

-	0	1
0	0	0_b
1	1	0

Here the subscripted 'b' represents a "borrow" from the left.
Example:

```
  0101 = 5 decimal
- 0011 = 3 decimal
  ----
  0010 = 2 decimal
```

Multiplication is even simpler. Here is the multiplication table for binary.

x	0	1
0	0	0
1	0	1

And here is an example, done in both North American and British order (you'll notice that, happily, both produce the same answer)
North American:

```
  0011 = 3 decimal
* 0101 = 5 decimal
  ----
001100
+ 0011
------
001111 = 15 decimal
```

British

```
  0011 = 3 decimal
* 0101 = 5 decimal
  ----
  0011
+ 001100
  ------
  001111 = 15 decimal
```

6.1.2.5 Signed numbers

The above applies to positive numbers, but how can we represent a negative number?

We cannot use a '-' sign because *all we can store* in the computer is zeros and ones. There are three methods which have been used. The last is still in use today. For the sake of simplicity all numbers in the following discussion are 8 bits long. The arguments apply to any length of binary number.

6.1.2.6 Signed Magnitude

With this system the leftmost bit is not considered part of the number, but as a representation of its sign:

- 0 indicates a positive number
- 1 indicates a negative number

The remaining n-1 (in our case 7) bits represent the number – the magnitude.

So: 00000101 would be 5 decimal and 10000101 would be -5 decimal.

Unfortunately there are a few problems with this.

Firstly consider zero.

00000000 represents 0 but so does 10000000 (-0 to be precise)

This is inconvenient as it complicates the circuitry. We should like to be able to compare two numbers (a very common operation) by saying: "if the bit patterns are the same the two numbers have the same value; if they are different then they have different values."

Unfortunately the second half of this is not true for signed magnitude binary numbers: the two patterns could represent +0 and -0, so extra circuitry must be incorporated to test for this – and this represents overhead that will be incurred *every time* we compare two numbers; the machine will be slower.

Secondly, arithmetic with signed magnitude is complex. Although adding two positive numbers is not a problem (unless the result overflows into the sign bit).

But consider adding +6 to -4:

```
+6 = 00000110
-4 = 10000100
```

If we simply add the bit patterns we shall get:

```
10001010
```

Which is -10 decimal, and not the -2 we wanted.

The answer is simple (!): if the two numbers have the same sign then we can add them (except for the sign bit) and put the common sign bit in the result.

Here are two examples:

```
+3 = 00000011
+6 = 00000110
     --------
   + 00001001 = +9 decimal
```

and

```
-7 = 10000111
-4 = 10000100
     --------
   + 10001011 = -11 decimal
```

If the two numbers are of opposite signs – i.e one plus and one minus – then the rule is: subtract the smaller magnitude from the larger and give the result the sign of the larger original number.

```
+3 = 00000011
-6 = 10000110
     --------
   + 10000011 = -3 decimal
```

Signed magnitude therefore has two strikes against it: 1.the arithmetic circuitry has to be more complex to deal with the problem of oppositely-signed numbers. 2.there are two representations of zero.

6.1.2.7 1's Complement

In the 1's complement system you get a negative number simply by inverting ('flipping') every bit in the positive version.

Thus 00000101 is +5 decimal and 11111010 is -5 decimal.

The advantage of 1's complement is that arithmetic is now so much simpler: we don't have to worry about opposite signs – it will all "come out in the wash".

The rule for adding numbers in 1's complement is simple: just add the two numbers together, *plus* if there is a carry out to the left this is added in as a 1 at the right. (Simpler view: the left carry out is *always* added at the right, whether it's a 1 or a 0).

This is sometimes known as the *end-around-carry.*

Let's look at an example:

```
+7 = 00000111
+4 = 00000100
     --------
   + 00001011 = +11 decimal
```

now -7 = 11111000 and -4 = 11111011, so

```
-7 = 11111000
-4 = 11111011
            1 the 'end around carry' bit
     --------
   + 11110100 = -11 decimal (flip the bits and you get 00001011 or +11 decimal).
```

If the two numbers are of opposite signs – i.e. one plus and one minus – the the *same* method works:

```
+7 = 00000111
-4 = 11111011
            1 'end around carry'
     --------
     00000011 = +3 decimal
```

also

```
-7 = 11111000
+4 = 00000100
            0 'end around carry'
     --------
     11111100 = -3 decimal (flip the bits and you get 00000011 = +3 decimal)
```

Now, what if we add

```
+7 = 00000111
-7 = 11111000
            0 zero 'end around carry'
    ---------
     11111111 = -0 (flip the bits and you get 00000000 = +0)
```

So, although the arithmetic works a treat we still have two representations for zero.

6.1.2.8 2's Complement

With 2's complement arithmetic we make the conversion from positive to negative (and vice-versa) slightly more complex.

To go from one to the other now takes two stages:

1. flip all the bits
2. add 1. (This is almost like adding in the end around carry ahead of time).

(Yes, we now have an *algorithm* for converting between positive and negative numbers)

Let's try some examples:

```
+4 = 00000100
```

so to generate -4:

```
1. 11111011
2.      + 1
   --------
   11111100
```

Another:

```
+7 = 00000111
```

so:

```
1. 11111000
2.      + 1
   --------
   11111001 = -7
```

Now we can do arithmetic as usual, ignoring carries and sign.

```
+3 = 00000011
+6 = 00000110
     --------
   + 00001001 = +9 decimal
```

and two negative numbers:

```
-7 = 11111001
-4 = 11111100
     --------
   + 11110101 = +11 decimal
  (flip: 00001010+1 = 00001011 = +11_10)
```

(Don't forget we ignored the carry out to the left).

If the two numbers are of opposite signs – i.e one plus and one minus the the same method works:

```
+7 = 00000111
-4 = 11111100
     --------
     00000011 = +3 decimal
```

and the remaining combination:

```
-7 = 11111001
+4 = 00000100
     --------
     11111101 = -3 decimal
(flip: 00000010+1 = 00000011 = +3_10)
```

Now, what if we add:

```
+7 = 00000111
-7 = 11111001
     --------
     00000000 = 0
```

Interesting! Now what if we try to generate -0 from +0?

```
+0 = 00000000
     11111111
          + 1
     --------
     00000000
```

Aha! In 2's complement we: have

- simpler arithmetic and
- only have a single representation for zero

This is why 2's complement is today exclusively used for integer binary arithmetic.

A tip In order to convert say 10011011 from binary to decimal we must first ask whether the bit pattern represents a signed or unsigned binary number.

There is *absolutely nothing* in the bit pattern itself that tells us – it is important that you appreciate this.

In fact those 8 bits might represent an ASCII or EBCDIC character – see later – or a machine instruction for all we know.

If it is unsigned then the usual conversion method (rightmost bit represents 1, next 2, next 4 etc.) works. E.g.:

```
10011011
||||||||-- represents 1 * 2^0 = 1 * 1
|||||||----represents 1 * 2^1 = 1 * 2
||||||-----represents 0 * 2^2 = 0 * 4
|||||------represents 1 * 2^3 = 1 * 8
||||-------represents 1 * 2^4 = 1 * 16
|||--------represents 0 * 2^5 = 0 * 32
||---------represents 0 * 2^6 = 0 * 64
|----------represents 1 * 2^7 = 1 * 128
```

If it is a 2's complement signed number then we can either use the algorithm described above (flip bits and add 1) to turn it into its positive value (any binary number with its leftmost bit 1 will be negative: this is true – albeit for different reasons – for all three systems).

But then we still have to convert the resulting positive number.

On the other hand, we can view a 2's complement number like this:

```
10011011
||||||||-- represents 1 * 2^0 = 1 * 1
|||||||----represents 1 * 2^1 = 1 * 2
||||||-----represents 0 * 2^2 = 0 * 4
|||||------represents 1 * 2^3 = 1 * 8
||||-------represents 1 * 2^4 = 1 * 16
|||--------represents 0 * 2^5 = 0 * 32
||---------represents 0 * 2^6 = 0 * 64
|----------represents 1 *-2^7 = 1 * -128
```

Clearly, now, the leftmost bit represents a large, negative value: in fact it is large enough to outweigh every bit to its right, which is why, in 2's complement, when the left bit is on we have a negative number.[1]

But you should *not* call it the "sign bit". It contributes value as well.

So the bit pattern above, if a positive number, represents 155 (= 1 + 2 + 8 + 16 + 128)

But if it is a 2's complement number it is negative and has the value -101 (= 1 + 2 + 8 + 16 + -128).

We can confirm this by using the slow algorithm:

```
      10011011
flip: 01100100
           + 1
      --------
      01100101 = 1 + 4 + 32 + 64 = 101_10
```

So the original bit pattern represented -101_{10} as claimed.

And some days the bear eats you.

6.1.3 Other Bases

We have now seen how to convert between binary and decimal.

Everything inside a modern computer is stored in binary (for historical variants see chapter X). However, it is difficult for humans to read long strings of 1's and 0's and so two other bases have historically been used when interpreting the binary contents of memory for their human convenience.

These are base 8 (octal) and base 16 (hexadecimal).

Conversion between them is very simple, because both 8 and 16 are powers of 2 ($8 = 2^3$ and $16 = 2^4$). Hence each 3 binary bits = 1 octal digit and 4 binary bits give us 1 hexadecimal digit. The tables for the two are shown below.

[1] Which of these two methods you use to convert from 2's complement binary to decimal will often depend on how many of the bits are 1's.

Binary	Octal	Hexadecimal	Decimal	Binary	Octal	Hexadecimal	Decimal
000	0	0	0	1000	10	8	8
001	1	1	1	1001	11	9	9
010	2	2	2	1010	12	A	10
011	3	3	3	1011	13	B	11
100	4	4	4	1100	14	C	12
101	5	5	5	1101	15	D	13
110	6	6	6	1110	16	E	14
111	7	7	7	1111	17	F	15

Converting from binary to either octal or decimal does not involve any arithmetic. Simply divide the bits into groups of three (octal) or four (hexadecimal)[2] and look up each group in the table.

```
Octal:       100111101100     = 100 111 101 100     = 47548
Hexadecimal: 1011111000101101 = 1011 1110 0010 1101 = BE2D16
```

6.1.4 Floating Point Basics

We have seen how to represent integers – positive and negative – in binary. Unfortunately, in the real world we must deal with numbers which are not integers, numbers such as 1/2, 3.14159260, 2.71828, etc.

We can construct binary fractions using a binary "point", in the same way as we do with decimal:

```
101.101
|||  |||--represents 2^-3(1/8)
|||  ||---represents 2^-2(1/4)
|||  |----represents 2^-1(1/2)
|||------represents 2^0 (1)
||-------represents 2^1 (2)
|--------represents 2^2 (4)
```

And so the above number represents 4 + 1 + 1/2 + 1/8 = 5 5/8 (5.625)

Unfortunately there are a number of problems with this scheme.

Firstly, just as we cannot represent some fractions exactly as decimals (1/3 springs to mind), so there are some decimal fractions which cannot accurately be representing as non-terminating binary fractions (e.g. 1/10, 1/100, which is particularly unfortunate when we use decimal currency).

Nor does the above scheme cope well with both large and small numbers; scientific applications, in particular, need enormous ranges of values.

Scientists have faced a similar problem for decades when writing calculations etc. For example, looking quickly, are the following two numbers the same?

[2]Octal was used on machines with word-sizes of 9-, 12-, 18-, 24- and 36-bits, hexadecimal on machines with word sizes of 4-, 8-, 16-, 32- and 64-bits.

0.000000000000000000000024 and 0.0000000000000000000000024? How about 5300000000000000000000 and 530000000000000000000?

In order to avoid this problem (and save writing) scientists have developed the so-called scientific notation for writing down both very large and very small numbers.

The general format is:

```
[S] M e E
```

Where S is an optional sign, M is the mantissa and E is the exponent.

The number written as -1.25e20 signifies -1.25 x 1020 or 125000000000000000000.

Note that the exponent can also be signed, to represent extremely small numbers, e.g.: -1.25e-20 signifies -1.25 x 10-20 or -0.00000000000000000125.[3]

Floating-point representation is the binary equivalent of scientific notation. In floating-point the number is divided into two parts, the exponent and the mantissa (aka significand). Because these numbers are being represented inside a computer, inevitably there will be limits to each of the field lengths within the storage used.

In the case of the IEEE 754 standard (see below), a single-precision number uses 1 bit for the mantissa's sign, 8 for the exponent and 23 for the mantissa's magnitude.

In order to maximise the precision (number of significant digits) of numbers stored they are normalised; this means that the exponent is adjusted so that there are no significant bits before the binary point.

In decimal the equivalent would be to have no significant digits before the decimal point. In other words 1.25e10 normalised would be 0.125e11; 634259876e4 normalised would be 0.634259876e13

Multiplying and dividing FP numbers is simple; addition and subtraction pose problems. For example multiplying 3e7 by 4e-2 is simple: the rule (algorithm) says to multiply the mantissas and add the exponents, so the result is: 12e5 (3x4 = 12, 7 + -2 = 5).

Or, using the normalised versions: 0.3e8 x 0.4e-1 = 0.12e7 (which is the normalised version of 12e5)

For division we simply divide the mantissas and subtract the exponents.

This is basically the method of using logarithms, which scientists used for centuries before the invention of accurate calculators (see HISTORY section).

But how can we *add* 0.3e3 and 0.25e-1?

Firstly we must arrange matters so that both numbers have the same exponent, a process usually called *denormalisation*. This can be time-consuming, but will be necessary. We then add the mantissas.

In this case, suppose that the numbers are denormalised to 3e2 and 0.00025e2, the sum is then easily calculated as 3.00025e2, which normalises to 0.300025e3.

[3] This notation also allows scientists to write down numbers so large that they can be represented in no other way. For example, *Skewes' number*, originally calculated as $10^{10^{10^{34}}}$ which is such an enormous number that if you wrote a '1' and then a '0' on *every atom in the universe* you would not come close to writing it out in full.

6.1.4.1 Dangers

Floating point numbers arrived at in calculations are *always* approximations and this should not be forgotten.

Historically, floating-point representations differed between manufacturers and it was not uncommond for different machines from the same manufacturer to have differing accuracies (for example the IBM Series 360's floating point was, in some cases, less accurate than that of the IBM 7090, an earlier machine)and even used different bases (2, 10, 16).

Programmers had to learn (and must still learn some) strange and counter-intuitive procedures in order to work successfully with floating point.

For example: Floating Point Arithmetic is not associative: in arithmetic we rely on the fact that

```
(x + y) + z = x + (y + z)
```

In floating point arithmetic, it ain't necessarily so.

To see why, imagine a decimal floating point scheme with the following limitations:

- 6 digits of mantissa
- 3 digits of exponent

Now imagine the following addition:

```
1.0e20 + 1.0 + -1.0e20 (i.e. 1020 + 1 -- 1020)
```

Let us assume addition from left to right; now, in our representation the numbers will be normalised as 0.1e21, 0.1e1, -0.1e21. When we try to denormalise the first two to use the same exponent we have a problem: 100000000000000000e1 and 0.1e1 have the same exponent, and so do 0.1e21 and 0.00000000000000000001e21, but in both cases one of the two mantissas is far too large for our representation and the smaller number will inevitably be truncated to zero, so:

```
1.0e20 + 1.0 = 1.0e20
```

and hence

```
(1.0e20 + 1.0) + -1.0e20 = 0.0
```

and not 1.0 as expected.

In this case our representation prevents doing this calculation accurately (I told you everything was approximate).

There are other occasions when we can get the right answer but only if we add the numbers in the correct order; the rule is: when adding (or subtracting) floating point numbers work from the smallest to the largest.

To see why consider this example:

```
0.1e3 + 0.1e-4 +....
```

If we add these two numbers, we shall get 0.1e3 (for the same reasons as above, we can't denormalise both numbers accurately).

But, if these are just the first two terms of a long series, and the next 1000 numbers are all 0.1e-4, then these 1000 items add up to 0.1e-1 and we *can* add that accurately to 0.1e3:

```
0.1e3 + 0.00001e3 = 0.10001e3
```

The introduction of the IEE 754 Floating Point Standard (and its successor IEEE 854, which is virtually identical) has at least guaranteed that almost every computer being used is at least compatible in the floating point area (although there are some major exceptions still being phased out: IBM mainframes and Cray supercomputers to name two).

Testing two floating point numbers to see if they are equal is asking for trouble. Are the following two numbers equal?

```
1.000000000000000000
```

and

```
1.00000000000000000000000000001?
```

Obviously they are *not* equal mathematically speaking, but if each is the result of a calculation (necessarily approximate, to rub the point in yet again) shouldn't we perhaps *consider* them equal?

Usually in floating point arithmetic we'll check two values to see if they are *close enough* (obviously the meaning will change for different algorithms). As in the calculation:

```
if (x -- y) < delta
```

where delta (a traditional term) is a small number representing how close is "close enough".

6.1.4.2 Rounding

Related to the above is the danger of *rounding*.

The basic rule is simple: *don't round until you absolutely must.*

To see the dangers, imagine we are rounding to 1 decimal place after each addition and we want to add the following numbers: 1.23, 2.32, 6.45.

The true sum, to one decimal place, should be 10.0 (these numbers are *not* chosen at random).

But, if we round to one decimal place after each addition:

```
1.23 + 2.32 = 3.55
```

which rounds up to 3.6, hence

```
3.60 + 6.45 = 10.05
```

which rounds up to 10.1

A 1% inaccuracy in just two additions!

In addition to learning the rules above, historically suspect implementations (all manufacturers were guilty at one time or another) meant that programmers had to be wary of things like:

```
if x == 0.0
  z = 1.0
else
  z = y / x
```

In one case (the CDC6600) the comparison was done by the CPU's add unit and the division (obviously) by the divide unit.

Unfortunately the two units tested different numbers of bits to decide whether a number was (effectively) zero. This meant that for very small x the test for equal to zero would fail (the adder didn't consider the number zero), but the divide unit would consider x to be zero and would cause a divide by zero hardware error and the program would crash.

The "fix" was to program:

```
if (1.0 * x) == 0.0
  z = 1.0
else
  z = y / x
```

Unfortunately, this was not portable and would fail on other hardware.

For a similar reasons, on the original IBM S/360 the test:

```
if (1.0 * x) == x
```

would also fail.

And many programmers became accustomed (on several different machines) to writing:

```
y = (0.5 - x) + 0.5
```

instead of

```
y = 1.0 - x
```

which would give the wrong answer.

6.1.4.3 IEEE 754 Floating Point Standard

Here are the layouts for the IEE 754 floating-point standard:

0	1 2 3 4 5 6 7 8	9..31
s	exponent	mantissa/significand

IEEE 754 floating-point: single precision (32-bit) format

IEEE 754 floating-point: double precision (64-bit) format

If we use S for the sign, E for the exponent and M for the mantissa (significand), then the number stored is equivalent to:

$$(-1)^S \times M \times 2^E$$

(Notice that the sign is stored as a separate bit, but do not confuse with signed magnitude, which was solely used for integers)

The IEEE standard has been in use since ~1980 (although the standard was not ratified until 1985[4]) and uses some at-first-sight strange optimisations to fit as much as possible into as small a space as possible.

For example, when we normalise a binary number, as it says above, the exponent is adjusted so that there are no significant bits before the binary point. This means that the first bit after the binary point must be a 1 (it wouldn't be significant if it was a zero), so it is not stored! This increases single-precision accuracy to 24 bits.

Furthermore, the exponent is stored in what is called *biased format*: whatever the exponent field contains, you must subtract 127 (1023 in double precision) to get the true exponent (so 1 => -126, 254 => 127). The reason for this is to make sorting floating point numbers easier – and we have seen why one might want to sort them (adding/subtracting?).

[4]The current standard is IEEE 754-2008, ratified in August 2008. It includes both the previous IEE 754-1985 as well as IEEE 854-1987, the standard for radix-independant floating point (which allows for bases other than 2 to be used).

Single precision		Double precision		Represents
Exponent (8 bits)	Significand (mantissa) 23 bits	Exponent (11 bits)	Significand (mantissa) (52 bits)	
0	0	0	0	0
0	non-zero	0	non-zero	+/- denormalised number
1-254	anything	1-2046	anything	+/- normalised floating point number
255	0	2047	0	+/- infinity
255	non-zero	2047	non-zero	NaN (Not a Number)

The Full IEEE754 Floating Point Format

A denormalised number is a way of allowing very small values (which don't have a 1 immediately after the binary point) and is used in specialised operations.

The two representations for + and - infinity mean that a division by zero can be dealt with without having to cause a run-time hardware error.

NaN values result from attempts to divide zero by zero, or subtract infinity from itself.

The Bottom Line

Dealing with floating point is tricky: if you don't have to, don't.

6.1.5 Limitations – A Slight Digression

6.1.5.1 Integers

Clearly there are limits on the range of integers we can store in a given size of binary number.

In 2's complement, given n bits, we can store integers in the range: $-2^{n-1} \leq x \leq 2^{n-1} - 1$

Now, we know that there is an infinite number of (positive) integers. In fact we can say rather more than this: there is also an infinite number of negative integers, an infinite number of even numbers, of odd numbers, etc., etc.

How do we cope with these different (if there are different) infinities?

Enter the 19th century mathematical genius Georg Cantor, who defined the term *countably infinite*.

A countably infinite set is one that can be put into a 1-1 correspondence with the positive integers, in other words it can (if you have infinite patience) be counted.

The easiest example is the even numbers: the first is 2, second 4, third 5, etc. In fact the nth even number is simply $2n$. We can deal with the odd numbers similiarly ($2n$-1). Indeed, there is also a countably infinite set of prime numbers.

6.1.5.2 The Hilbert Hotel Part 1

> "Welcome to the Hilbert Hotel, the known universe's all-in-one vacation resort. Equipped with an infinite number of luxurious guestrooms, the hotel can always accomodate your party, no matter how large."[5]

The manager of the Hilbert Hotel finds himself (or herself) in charge of a hotel which is:

1. Always full to capacity, yet
2. Always able to accomodate more guests.

How can this be achieved?

Imagine you arrive at the front desk. In order to accomodate you the manager will simply ask every guest to move to the next room up (i.e. the guest from room 1 moves to room 2, the guest from room 2 to room 3, etc.) and you will be installed in room 1[6].

Clearly this works. Moreover, using this technique, we can easily accomodate n new arrivals, where n is any finite number: simply get each guest to move from their room to the room whose number is n greater.

One evening, though, an infinite number of new guests arrive.

Can we find rooms for them all?

No problem! Simply tell all the current guests to move to the room number which is double their present one (guest in room 1 moves to room 2, 2 to 4, etc.).

We now have an infinite number of odd-numbered rooms available for the new arrivals.

6.1.5.3 Rational Number (fractions)

We have seen that the integers (and many subsets of the integers) are *countably infinite*.

[5] Although this famous example probably created by the great mathematician David Hilbert, he never described it in print. It was first described by George Gamow in 1947 in his book *One Two Three...Infinity: facts and speculations in science.*

The exposition here is based on that in John D. Barrow's *The Infinite Book* (2005).

[6] The reception desk has a broadcast message system which can send a message to every room simultaneously. Obviously sending individual messages to an infinite number of rooms would take infinitely long.

But what about fractions?

Firstly, mathematicians define a *rational* number as a fraction of the form $\frac{p}{q}$ where p and q are both integers.

Our first question – in order to appreciate how accurate (or otherwise) our representation can be – is: are there more rationals than integers?

Well, the only way to find out is to count them – or try to.

But how can we count them? Does $\frac{1}{2}$ come before or after $\frac{2}{3}$?

Enter Cantor again.

Imagine arranging rational numbers in a grid, like this:

$$\begin{array}{ccccccc}
\frac{1}{1} & \frac{1}{2} & \frac{1}{3} & \frac{1}{4} & \frac{1}{5} & \cdots \\
\frac{2}{1} & \frac{2}{2} & \frac{2}{3} & \frac{2}{4} & \frac{2}{5} & \cdots \\
\frac{3}{1} & \frac{3}{2} & \frac{3}{3} & \frac{3}{4} & \frac{3}{5} & \cdots \\
\frac{4}{1} & \frac{4}{2} & \frac{4}{3} & \frac{4}{4} & \frac{4}{5} & \cdots \\
\frac{5}{1} & \frac{5}{2} & \frac{5}{3} & \frac{5}{4} & \frac{5}{5} & \cdots \\
\cdots & \cdots & \cdots & \cdots & \cdots & \cdots
\end{array}$$

Obviously, each row is infinitely long and, equally obviously, there is an infinite number of rows. How then does this help?

Well, if you are a genius (which Cantor was) the answer is simple[7].

We draw diagonals connecting each number whose *numerator* (top) and *denominator* (bottom) add to the same value. When we have processed these, we go along to the next value.

This diagram shows us how:

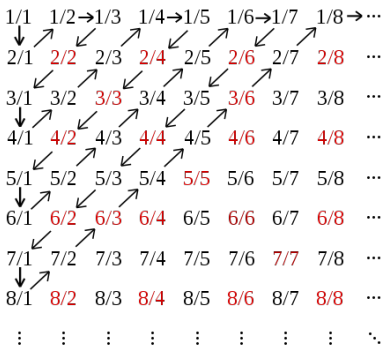


Figure 6.1: Cantor's enumeration of the rationals

And here is a program which will print out as many of the rationals, in order, as you wish:

[7] This is one of those solutions which seems obvious and simple *once it is explained to you.* It is coming up with the explanation which takes genius.

```
#!/usr/bin/python
import sys
def cantor (n):
    top = 1
    bottom = 1
    for i in range(n):
        while bottom >= 1:
            print '%d/%d' % (top, bottom),
            bottom -= 1
            top += 1
        print
        bottom = 1
        while top >= 1:
            print '%d/%d' % (top, bottom),
            bottom += 1
            top -= 1
        print
        top = 1

if __name__ == '__main__':
    cantor (int(sys.argv[1]))
```

The command line parameter specifies the number of *zigzags* to print; each zigzag consists of a *zig* (top decreases, bottom increases) and a *zag* (vice-versa).

Here it is in action:

```
$ python cantor.py 5
1/1
2/1 1/2
1/3 2/2 3/1
4/1 3/2 2/3 1/4
1/5 2/4 3/3 4/2 5/1
6/1 5/2 4/3 3/4 2/5 1/6
1/7 2/6 3/5 4/4 5/3 6/2 7/1
8/1 7/2 6/3 5/4 4/5 3/6 2/7 1/8
1/9 2/8 3/7 4/6 5/5 6/4 7/3 8/2 9/1
10/1 9/2 8/3 7/4 6/5 5/6 4/7 3/8 2/9 1/10
```

This means that the (positive)[8] rationals are *countable*, which means that there are no more rationals than there are integers.

Hence....

[8] Extending this treatment to include the negative rationals is trivial and left to the reader as an exercise.

6.1.5.4 The Hilbert Hotel Part 2

The Hilbert Corporation actually owns a (countably) infinite number of hotels across every galaxy in the universe.[9]

However, the one you are managing is seen to be the best and most efficient, and so the corporation decides to close all of the others except yours *and transfer all of the infinite number of guests from each of the infinite number of hotels to your hotel.*

Can these infinity-squared guests be accomodated?

To quote Bob Dylan, "yes, I think it can be easily done"[10]: simply associate with each new guest the number of the hotel they are coming from and the room number they had in that hotel. Write this number as a fraction (rational number) thus: $\frac{hotelNumber}{roomNumber}$ and then send the new guests to their new rooms in Cantor diagonal order (using the program above to produce this order).

So:

- the guest from hotel 1 room 1, goes to room 1
- the guest from hotel 2 room 1, goes to room 2
- the guest from hotel 1 room 2, goes to room 3
- the guest from hotel 1 room 3, goes to room 4
- the guest from hotel 2 room 2, goes to room 5
- the guest from hotel 3 room 1, goes to room 6
- etc., etc.

and everyone gets a room!

Of course, the hardest part of all this to understand is why the guests put up with having to move rooms every time a new guest arrives.[11]

6.1.5.5 Irrational Numbers

But there are numbers which are not rationals, i.e. which *cannot* be represented as $\frac{p}{q}$ where p and q are integers[12].

Examples are $\sqrt{2}$, π and e.

Are these numbers countable?

Again, it was Cantor who provided the solution, a *proof by contradiction*[13].

---

[9] If the universe is actually finite, as many cosmologists maintain, this argument falls to the ground.

[10] "Just take everything down to Highway 61." *Highway 61 Revisited*

[11] Perhaps the Hilbert Hotel is like the Eagles' *Hotel California*, where "you can check out any time you want, but you can never leave".

[12] The ancient Greeks, for instance, were terribly offended by this and approximated π as $\frac{22}{7}$; more recently, in 1897, a bill was tabled in the Indiana State Legislature which would have legislated the value of π as 3.2. The bill did not pass.

[13] In such a proof, we assume what we are trying to disprove and then show that the assumption leads to a contradiction, meaning that the assumption is false.

Assume that the irrationals *between 0 and* 1 can be counted, in other words we can put them into some order.

Assume that the order is:

$0.d_{11}d_{12}d_{13}d_{14}d_{15}\ldots$
$0.d_{21}d_{22}d_{23}d_{24}d_{25}\ldots$
$0.d_{31}d_{32}d_{33}d_{34}d_{35}\ldots$
$0.d_{41}d_{42}d_{43}d_{44}d_{45}\ldots$
$0.d_{51}d_{52}d_{53}d_{54}d_{55}\ldots$
...

Where d_{mn} signifies the nth digit of the mth number. Notice that we make no assumption about what the actual digits are.

Now, consider the number X:

$0.f(d_{11})f(d_{22})f(d_{33})f(d_{44})f(d_{55})\ldots$

where the function f is defined as:

```
f(0) = 1
f(1) = 2
...
f(8) = 9
f(9) = 0
```

Clearly the X is different from the first number of our supposed sequence in the first decimal place ($d_{11} \neq f(d_{11})$ no matter what the value of $d_{11}$is), different from the second number in the second decimal place, etc.

Therefore this new number X *is not in the sequence.*

In other words, our supposed countable sequence of irrationals is incomplete, hence the irrationals are *uncountable* and we can see from our proof that there are more irrationals *between 0 and 1* than there are rationals between -$\infty$and $+\infty$.[14]

The number of irrationals is sometimes known as c, the *cardinality of the continuum.*

Cantor went on the show that there is an infinite series of infinities, which he labelled with the first letter of the Hebrew alphabet and a suffix.

The "regular" infinity that we are all familiar with, i.e. the number of integers, is represented as $\aleph_0$, pronounced "Aleph-null".

The next infinity would be $\aleph_1$(Aleph-1) and so forth.

[14]This form of proof is called a *diagonalisation proof*, you can probably see why. Diagonalisation was also used by Alan Turing in an important – probably *the* most important – computer science proof. We shall encounter Turing, later in chapters 5 and 12.

One question Cantor asked, but could not answer, was:

Is $c = \aleph_1$?

In other words, is the number of irrationals the next smallest infinity after the number of integers?

Cantor proposed that it is, and this proposition is known as the *Continuum Hypothesis*.

It was not until the early 1960s that this question was answered, by Paul Cohen, in a two part paper published in 1963 and 1964.

Gödel had already shown that no contradiction arises if the continuum hypothesis is added to the basic axioms of set theory.

Cohen showed that no contradiction arises if the *negation* (i.e. opposite) of the continuum hypothesis is added to the axioms.

This shows that the continuum hypothesis is *undecidable*.

Having made all our heads spin, it is now time to return to the topic at hand.

6.1.5.6 What does all this mean?

Simply this: there are limitations to our representation of numerical data; in particular floating-point values are often *approximations*[15].

We should not only be aware of this, but endeavour to minimise the inaccuracies.

There are entire books and courses on this topic.

6.1.6 Character Data

Although some early computers could only deal with numbers, the digital representation of letters of the alphabet, digits, punctuation symbols etc. predates the computer.

For example, the Telex system used a 5-bit code known as Baudot Code (invented by Emile Baudot in 1870), giving 32 different bit patterns. This is not enough for both the 26 letters and the 10 digits, so the code works in two modes and the *Figure Shift* and *Letter Shift* codes switch from one mode to the other.[16]

In the 1950s computers generally used some variant of BCD (Binary Coded Decimal) to encode character data derived from punched cards. BCD is a scheme where each decimal digit is stored separately in binary form. The minimum size for storing one digit is thus 4 bits. The bit patterns 0000 to 1001 represent 0 to 9 decimal; 1010 to 1111 are not used.

In the late 1950s and early 1960s a character set known as BCD was in widespread use. This was a 6-bit character set, thus yielding 64 bit patterns – sufficient for the digits, the alphabet (upper case only) and some punctuation

[15] Although it has been shown that the IEEE854 standard is accurate enough to add a *single penny* to the US National Debt. Of course, this was before George W. Bush and the Iraq War.

[16] See Apendix A for the complete encoding.

characters. BCD was still in use in the 1970s. There was, unfortunately, no single standard for BCD, which varied between different manufacturers, even between different machines from the same manufacturer. The one thing they had in common was the representation of the decimal digits by patterns which were numerically equal to the value of the digit (e.g. '1' = 0000001 in all BCD codes).

In the 1960s two new character sets were created, both of which are still in use today.

ASCII

The American Standard Code for Information Interchange was first published, by ANSI (the American National Standards Institute), in 1963 and was a 7-bit code, yielding 128 bit patterns: sufficient for both upper and lower case alphabets, the digits and plenty of punctuation. Given the name of the character set, we should not be too surprised that there was no allowance for accents above (or below) letters, as used in many European languages. Here is the standard 7-bit ASCII code, ranging from 0016 to 7F16.

	0	1	2	3	4	5	6	7
0	NUL	DLE	space	0	@	P	`	p
1	SOH	DC1 XON	!	1	A	Q	a	q
2	STX	DC2	"	2	B	R	b	r
3	ETX	DC3 XOFF	#	3	C	S	c	s
4	EOT	DC4	$	4	D	T	d	t
5	ENQ	NAK	%	5	E	U	e	u
6	ACK	SYN	&	6	F	V	f	v
7	BEL	ETB	'	7	G	W	g	w
8	BS	CAN	(	8	H	X	h	x
9	HT	EM	)	9	I	Y	i	y
A	LF	SUB	*	:	J	Z	j	z
B	VT	ESC	+	;	K	[	k	{
C	FF	FS	,	<	L	\	l	|
D	CR	GS	-	=	M	]	m	}
E	SO	RS	.	>	N	^	n	~
F	SI	US	/	?	O	_	o	del

Figure 6.2: Standard (7-bit) ASCII

Virtually all computers today use an 8-bit version of ASCII, which allows for accented characters used by most European languages (although not all). Unfortunately there are two differing "standard" ways of extended ASCII. Here are the two normally used:

The table on the left is "OEM extended ASCII", as built into most PCs. Unfortunately it is usually redefined by localised characters for each country. The table on the right is ANSI extended ASCII and is used by Windows, some versions of Unix and numerous applications. Even 8-bit ASCII is, however, inadequate for representing all of the world's written languages.

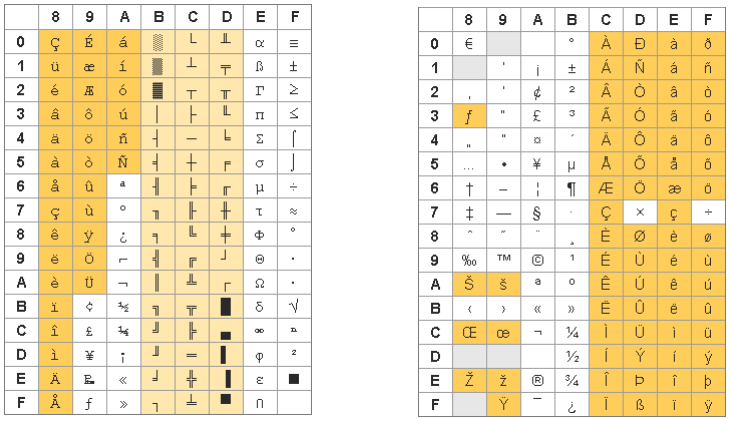


Figure 6.3: The two 8-bit ASCII extensions

EBCDIC

IBM were not a party to the standards body which defined ASCII and, instead, they created their own 8-bit (256 bit pattern) successor to BCD, EBCDIC (Extended BCD Interchange Code) which was first used on the System/360 in 1964 and is still used today on IBM mainframes, such as the S90, and mid-sized systems, such as the AS/400.

Although EBCDIC has been widely used, it is problematic, particularly in the PC-dominated world.

Perhaps the major problems is that the two sets do not include the same characters. IBM, which states that there are 27 variants of EBCDIC, has published an "invariant EBCDIC" table which specifies which characters exits in every version of EBCDIC.

Unfortunately, the following characters, which are in ASCII, are not in this invariant core EBCDIC:

```
! " # $ ( _ @ ) [ \ ] ^ ‘ { | } ~
```

Another problem is that the letters of the alphabet in EBCDIC are not contiguous, as they are in ASCII; to be precise, in ASCII the upper-case alphabet, A to Z, forms one contiguous set (i.e. there are no bit patterns in the range which are anything other than upper-case letters) and the lower-case alphabet, a to z, another. This means that a simple ASCII test for an upper-case letter in variable x ('A' <= x <= 'Z') must be broken into three tests in EBCDIC ('A' <=x <= 'I' OR 'J' <= x <= 'R' OR 'S' <= x <= 'Z').

Unicode

The Unicode standard grew out of work originally done in 1986-7 at Xerox into representing the common Chinese and Japanese ideograms. Version 1.0 of the

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	DLE	DS		SP	&	-									0
1	SOH	DC1	SOS				/		a	j			A	J		1
2	STX	DC2	FS	SYN					b	k	s		B	K	S	2
3	ETX	TM							c	l	t		C	L	T	3
4	PF	RES	BYP	PN					d	m	u		D	M	U	4
5	HT	NL	LF	RS					e	n	v		E	N	V	5
6	LC	BS	ETB	UC					f	o	w		F	O	W	6
7	DEL	IL	ESC	EOT					g	p	x		G	P	X	7
8		CAN							h	q	y		H	Q	Y	8
9		EM							i	r	z	`	I	R	Z	9
A	SMM	CC	SM		C CENT	!		:								
B	VT	CU1	CU2	CU3		$	,	#								
C	FF	IFS		DC4	<	*	%	@								
D	CR	IGS	ENQ	NAK	(	)	_	'								
E	SO	IRS	ACK		+	;	>	=								
F	SI	IUS	BEL	SUB	|	--	?	"								

Table 6.1: The invariant EBCDIC subset

standard was printed in two volumes, in October 1991 and June 1992. The latest version is 4.1.0.

Unicode was originally a 16-bit standard, which used two bytes to represent a single character and thus imposed an upper limit of 65,536 (216) representable characters. This is not even sufficient to represent the entire set of Han characters used in Chinese, Japanese and Korean. Since Unicode version 2.0 (1996) this limitation has been removed.

The current Unicode standard supports the 1.1 million plus code points (an integer representing the character) of the Universal Character Set (ISO 10646). The "Basic Multilingual Plane" consists of the first 65,536 points. The current Unicode standard defines over 96,000 characters and their code points. The standard is sufficient to embrace every language that is currently being used, together with a number of dead languages, languages (e.g. Canadian Aboriginal) which originally had no written form and even fictional ones. The following languages are among those currently supported by Unicode:

Arabic	Armenian	Bengali	Braille embossing patterns
Canadian Aboriginal Syllabics	Cherokee	Coptic Cyrillic	Devanagari
Ethiopic	Georgian	Greek	Gujarati
Gurmukhi	Hangul (Korean)	Han (Kanji, Hanja, Hanzi)	Japanese (Kanji, Hiragana and Katakana)
Hebrew	Khmer (Cambodian)	Kannada	Lao
Latin	Malayalam	Mongolian	Myanmar (Burmese)
Oriya	Syriac	Tamil	Thai
Tibetan	Yi Zhuyin (Bopomofo)		

Just some of the many languages supported by Unicode.

In addition, there is support for "dead" languages and scripts such as Runic, Egyptian hieroglyphs, Mayan, Linear-B, Cuneiform, Ugaritic and Old Italic; support has even been suggested for the two Elvish scripts (Tengwar and Cirth) invented by J.R.R. Tolkien – as well as Klingon.

Operating systems are currently supporting UTF-8 and UTF-16 (8- and 16-bit Unicode Transformation Format).

Unicode is a large and complex topic. There is a great deal of information to be found at the Unicode Consortium web site www.unicode.org.

Chapter 7

Hardware Basics

> We are becoming the servants in thought, as in action, of the machine we have created to serve us.
>
> John Kenneth Galbraith

7.1 Digital Logic

Computers are built from electronic circuits, but these circuits are only useful because they can implement the *logical* functions that are required.

7.1.1 Boolean Logic

The English mathematician George Boole (1815-1864), in his 1854 book "An Introduction to the Laws of Thought", decribed a two-valued algebra, now universally known by his name.[1]

We can either use the basic values 0 and 1, or we can replace them with F(alse) and T(rue). From now on we'll use 0 and F(alse) and 1 and T(rue) as interchangeable.

7.1.2 Basic Boolean Functions

When we study Boolean logic, the first operations we encounter are usually AND, OR and NOT.

We define these functions by giving the *truth tables* in which, for every possible combination of the two inputs, the output is specified.[2]

Here, then, are the truth tables for AND, OR and NOT.

[1] This was not his first publication, but he considered it as his ultimate statement on the subject.

[2] Another advantage of using binary is that, for n inputs there are only 2^n possible combinations of input values.

A	B	A AND B
0	0	0
0	1	0
1	0	0
1	1	1

A	B	A OR B
0	0	0
0	1	1
1	0	1
1	1	1

A	NOT A
0	1
1	0

The surprise here is in the table for OR, which says that the expression *A OR B* is true when:

1. A is true
2. B is true
3. A and B are both true

Now this is *not* the way we usually use the word "or" in writing or speaking English—if a menu offers you "apple or cherry pie" for dessert, it is highly unlikely that you will be able to have both.

The OR we have seen above is known as the *Inclusive OR*; there is another version, which corresponds to the either/or we use in speach. This is known as the *Exclusive OR* (XOR) and its truth table is:

X	Y	X XOR Y
0	0	0
0	1	1
1	0	1
1	1	0

Note that *X XOR Y* is true when *X* is true or when *Y* is true, but not when *both* are true.

7.1.3 Important Boolean Laws

Long before the computer, Boole had already derived and proved a number of important relationships between what we would now call boolean functions.

These laws are important for computer scientists for a number of reasons, one of which is that they can make programming complex conditions simpler.

In the laws specified below we adopt the usual boolean notation: *A+B* means *A OR B*, while *AB* means *A AND B* and *A'* means *NOT A*.

7.1.3.1 Commutative Law

- $A + B = B + A$
- $AB = BA$

7.1.3.2 Associative Law

- $A + (B + C) = (A + B) + C$
- $A(BC) = (AB)C$

7.1.3.3 Distributive Law

- $A(B + C) = AB + AC$
- $A + (BC) = (A + B)(A + C)$

7.1.3.4 Identity (or Idempotent) Law

- $A + A = A$
- $AA = A$

7.1.3.5 Complementary Law

- $A + A' = 1$
- $AA' = 0$

7.1.3.6 De Morgan's Law[3]

- $(A + B)' = A'B'$
- $(AB)' = A' + B'$

7.1.4 Transistors and Gates

The boolean functions—AND, OR, NOT—shown above can all be implemented electronically using the transistor; more complex circuits, known as *gates*[4], can easily constructed from these basic building blocks.

As we have already seen in XXX, it is entirely possible to implement these functions mechanically (Babbage), electromechanically (Zuse, Aiken) or electronically using thermionic valves ("tubes")—indeed every computer built until the late 1950s utilised tubes.

The transistor (invented in 1947 at Bell Labs, Murray Hill, NJ by John Bardeen, Walter Brattain and William Shockley - they were awarded the Nobel Prize for Physics for the invention in 1956) is used in digital logic as a switch:

[3] Augustus De Morgan (1806-1871) was a mathematician and friend of Boole. He introduced (in 1838) the term *mathematical induction*.

[4] No relation.

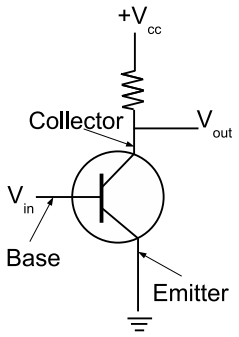


Figure 7.1: A Transistor

When a low voltage is applied to V_{in} the transistor turns off, and V_{out} is very close to V_{cc}. When a high voltage is applied to V_{in} the transistor turns on and pulls V_{out} to ground (conventionally zero). Thus the transistor can function as an *inverter* (i.e. a *not* gate).

You might expect that we shall now go on and build AND and OR gates. In fact, in terms of the electronics there are two easier functions to build: NAND and NOR, which are short, respectively, for NOT AND and NOT OR.

Here are the truth tables for NAND and NOR:

X	Y	X NAND Y
0	0	1
0	1	1
1	0	1
1	1	0

X	Y	X NOR Y
0	0	1
0	1	0
1	0	0
1	1	0

Each of these can be built as a two-transistor circuit:

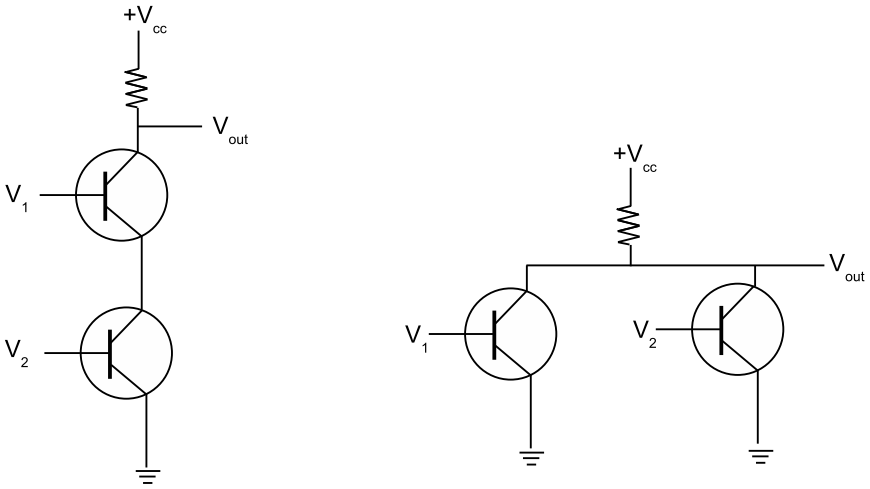


Figure 7.2: Two-transistor NAND and NOR gates

From these it is relatively easy to build the more conventional AND and OR gates. For the purposes of this text we shall skip the details of NAND and NOR gates (see any good text on Computer Architecture for further information) and consider AND, OR and NOT as our basic building blocks.

7.1.4.1 Gate symbols

The symbols used in circuit diagrams are more-or-less standardised. Here are the symbols used in this book.

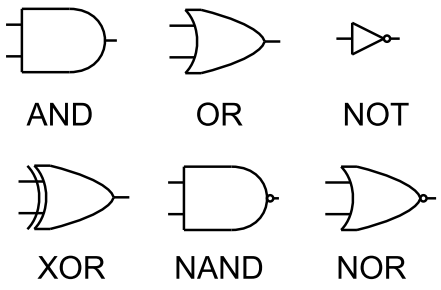


Figure 7.3: Standard gate symbols

Note the circle ("bubble") on the NOT, NAND and NOR gates; this indicates inversion (NOT operation) and so the difference between the OR and the NOR gate diagrams is simply the bubble on the output of the NOR.

Sometimes a bubble appears on one or both of the inputs; this just means that that input is inverted—it makes the diagram a little simpler:

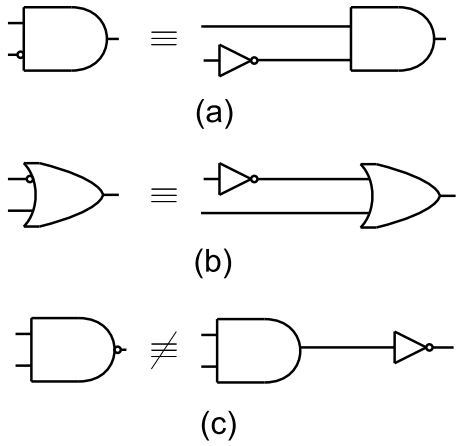


Figure 7.4: Some gate equivalencies

Note that although *logically* the two circuits in figure 7.4 are equivalent—and they would certainly achieve the same result—they are not considered to be, because, apart from the inverter (NOT) gate which, as we have already seen can be implemented with just one transistor, the NAND (shown) and NOR gates are the simplest of all–each requires just two transistors.[5]

7.1.4.2 Gates with more than two inputs

In the discussion below we shall often find it convenient to use, for example, an AND gate with three, rather than the usual two, inputs.

This is not a problem, because of the associative law specified above: $A(BC) = (AB)C$ means that we could actually write ABC[6] as a shorthand and that the circuit shown is the *exact* equivalent of a three-input AND:

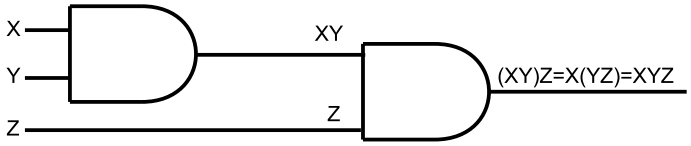


Figure 7.5: Circuit to AND together three inputs

[5] AND and OR actually require three transistors each, so the two circuits in figure 7.4 would in fact require two and four transistors respectively—hardly equivalent in terms of implementing them in hardware!

[6] Or, in this particular diagram, XYZ.

7.1.5 Combinatory Circuits

A *combinatory* circuit is one whose outputs depend upon its inputs and *nothing else*.

This may seem obvious, but there are circuits whose output depends not on the current input, but on *previous inputs*. We call these circuits *memory* for obvious reasons. A combinartory circuit is distinguished by the fact that it has no memory of what has gone before.

There are several import combinatorial circuits which we shall see being used through the computer.

7.1.5.1 A Simple Example

In fact, our three-input AND above is a combinatory circuit, but here is another, very simple, example:

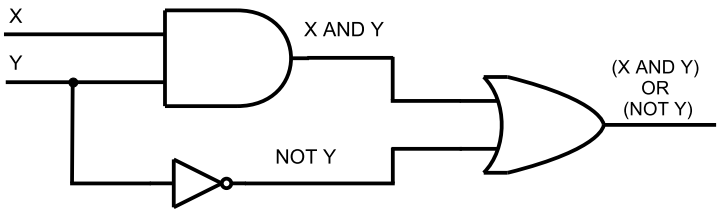


Figure 7.6: An example circuit

We can construct a truth table for the circuit by tracing the progress of signals through it.

X	Y	X AND Y	NOT Y	(X AND Y) OR (NOT Y)
0	0	0	1	1
0	1	0	0	0
1	0	0	1	1
0	1	1	0	1

7.1.5.2 Multiplexors

A *multiplexor* is a circuit with several data inputs, several *selector* inputs and a single output.

The selector inputs specify *one* of the data inputs and route it through to the output.

A good analogy is the pre-amplifier section of a stereo system. We may have several inputs: CD, DVD, tape, radio, maybe even a turntable; we only have one ouput—the speakers (eventually).

The pre-amp will have a way of selecting which input to send to the speakers. One important feature to note is that there will not always be sound coming out of the speakers—sometimes the input source slected will be silent and, at such times, we want silence from the speakers too.

Below is a four-input multiplexor. Note that the control inputs need to be

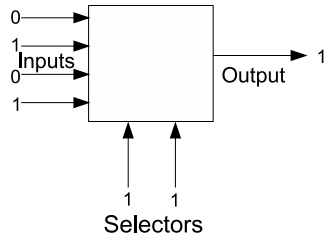


Figure 7.7: A 4-input multiplexor (black box view)

able to select one of *four* inputs, i.e. we need four different bit patterns available on the control, which means that we need two control input lines (2^2being 4, of course).

The control inputs specify a *binary* number and that is the number of the input we want to select (assuming, of course, that we start numbering at zero).

In other words, in general a multiplexor has:

- 2^n input lines
- 1 output line
- n selector (control) lines

We cannot construct a genuine truth table, as the ouput value depends not only on the control inputs, X and Y, but also on the data values themselves (the D_ns).

Here, though, is what we are trying to achieve:

X	Y	Output
0	0	$=D_0$
0	1	$=D_1$
1	0	$=D_2$
1	1	$=D_3$

Below is the actual circuit; it is worth taking the time to understand why it is built the way it is and exactly why it works.

Looking at the "truth table" we can see that:

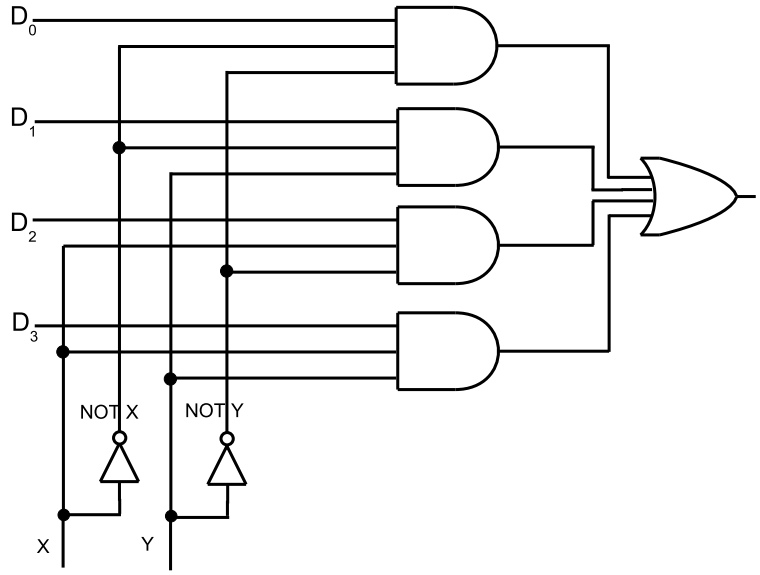


Figure 7.8: A multiplexor—circuit view

- when X and Y are both 0 (i.e. $NOT\ X$ and $NOT\ Y$ are both 1), D_0should be routed to the output ($00_2 = 0_{10}$)
- when X is 0 (NOT X is 1) and Y is 1 ($NOT\ Y$ is 0), D_1should be routed to the output ($01_2 = 1_{10}$)
- when X is 1 (NOT X is 0) and Y is 0 (NOT Y is 1), D_2should be routed to the output ($10_2 = 2_{10}$)
- when X and Y are both 1 (NOT X and $NOT\ Y$ are both 1), D_3should be routed to the output ($11_2 = 3_{10}$)

Notice that the inputs, X and Y are each fed to an inverter (NOT gate), so that all four combinations of X, Y, $NOT\ X$ and $NOT\ Y$ are available.

Notice that each of the data inputs, $D_n$is routed to exactly one AND gate, together with *one* of the control input combinations.

For example, let us look at the AND gate to which $D_2$is connected: now, according to the specification above, we want D_2sent to the output when X is 1 and Y is 0 (NOT Y is 1).

For D_2 to become the output of the AND gate, the three inputs need to be D_2, 1 and 1. The inputs to this gate are D_2, X and $NOT\ Y$, but both X and $NOT\ Y$ are 1, so the output from the AND is *whatever D_2 happens to be, whether that is zero or one.*

So, when X is 1 and Y is 0 the output from the corresponding AND gate is *an exact copy of D_2(which could be 0 or 1*, to belabour the point), as a result

of the Identity (or Idempotent) Law.

Moreover, at least one signal into each of the other three AND gates is *guaranteed to be zero*—consider the gate to which $D_1$is connected: its other inputs are $NOT\ X$ and Y. In this example case $X$is 1 and so $NOT\ X$ will be zero.

Which means that the other three ANDs are *guaranteed* to produce a 0 output.

The same logic applies to the other three data inputs, so that, whatever the combination of 0 and 1 fed in via X and Y, three of the ANDs are bound to produce 0 output, while the other produces an exact copy of whatever the $D_n$is selected by the control lines.

All four AND gates feed the final OR gate, but three of its inputs are bound to be zero and so, by the Identity Law again, the output of the OR gate will be an exact copy of its (possibly) non-zero input.

In other words if the inputs to our OR are 0, 0, 0 and $D_n$then the output will be D_n—and, just to hit you over the head with this, this value could still be zero (silence on the stereo, remember?) but it will be *exactly* the same value as is present on the D_n input line.

7.1.5.3 Decoders

A decoder is a circuit which has:

- n input lines
- 2^noutput lines (numbered from 0 to 2^n-1)

The binary value present on the input lines selects exactly one of the output lines; this line has a 1 on it and all of the others have 0.

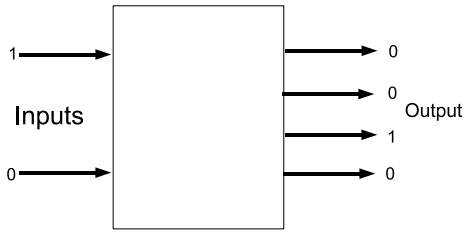


Figure 7.9: A Decoder (black box view)

In this example–the lines are numbered from the top—the input is 10_2, i.e. $2_{10}$and so output lines 0, 1 and 3 carry 0s, while output line 2 carries a 1.[7]

[7]Calculators use decoders in order to drive the display: a 4-bit binary value is fed to a 4-16 decoder (only the first 10 output lines are used, as the binary value may only be 0-9_{10}. Each of these output lines is connected to a different combination of LEDs—light emitting diodes—to create the digit to be displayed.

This is known as a 2-4 decoder, as it has two input lines and four output lines.

Producing a complete truth table is possible for the decoder, although it will be somewhat larger, as there are—in this case—four output lines.

X	Y	D_0	D_1	D_2	D_3
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

Here, then, is the circuit for the 2-4 decoder.

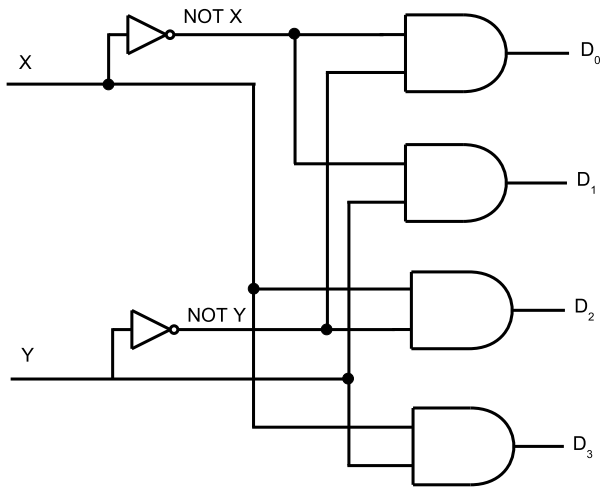


Figure 7.10: A 2-4 decoder—circuit view

Again, it is worth spending the time to make sure that you understand exactly how this works.

Firstly, we have four AND gates driving the output lines. According to the specification of the circuit, for any given combination of inputs, exactly one of those output lines should have a 1 on it and the other three should all have zeros.

Look very carefully at the circuit: each AND gate is fed a *different* combination of X, $NOT\ X$, Y and $NOT\ Y$, therefore only one gate can produce a 1 and the other three must produce 0s.

7.1.5.4 Comparators

A *comparator* is a circuit which compares two sets of input bits and produces a 1 if they are identical and a 0 otherwise.

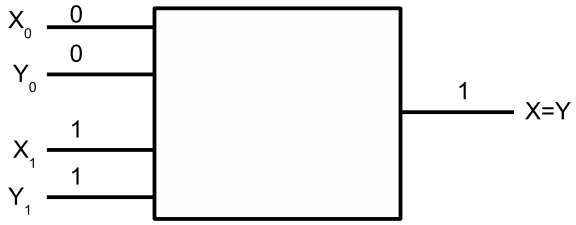


Figure 7.11: A 2-bit comparator—black box view

In the black box view we see that the inputs are not the same and therefore the output is 0.

The truth table for this is quite large, as there are four inputs and therefore 16 different combinations:

X_0	Y_0	X_1	Y_1	X=Y
0	0	0	0	1
0	0	0	1	0
0	0	1	0	0
0	0	1	1	1
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	0
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	0
1	1	0	0	1
1	1	0	1	0
1	1	1	0	0
1	1	1	1	1

This circuit uses the XOR (eXclusive OR) and NOR (NOT OR) gates: here is the truth table for XOR once more and the thruth table for NOR; notice that XOR produces a 0 when the inputs are the same and a 1 when they are different and that NOR produces a 1 only when both inputs are 0.

X	Y	X XOR Y
0	0	0
0	1	1
1	0	1
1	1	0

X	Y	X NOR Y
0	0	1
0	1	0
1	0	0
1	1	0

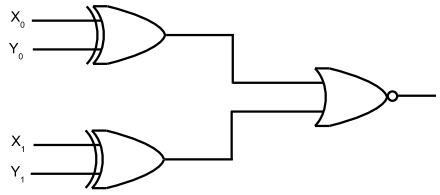


Figure 7.12: A 2-bit comparator—circuit view

The NOR gate therefore receives 2 bits: if either of them is non-zero then X and Y differ in at least one position.

The output from the final NOR gate will only be 1 if all four inputs are 0, i.e. when all four bit pairs match. (A0 = B0 AND A1 = B1 AND A2 = B2 AND A3 = B3) Note that this circuit only tells us whether the A and B bits are the same or not. It cannot tell us whether A > B numerically, which is why this is not the method used for comparisons in most CPUs.

7.1.5.5 Arithmetic Circuits

7.1.5.6 Shifters

A shifter, as the name suggests, is used to shift entire groups of bits to the left or the right. The simplest version allows for a 1-bit left shift or a 1-bit right shift.

Original bits	Shifted left by 1 bit	Shifted right 1 bit
00110101	01101010	00011010

The single control line, C, is used to determine the direction of the shift. When C is 1 the shift is to the right, when it is 0 the circuit shift is to the left. The vacated bit position receives a 0

Notice that in each case we fill the shifted position (the rightmost for a left shift, the leftmost for a right shift) with a 0.

7.1.5.7 Adders

An adder, as the name suggests, is a circuit which performs addition.

Let us look as the simplest possible for of arithmetic: adding together two single bits. The addition table looks like this:

+	0	1
0	0	1
1	1	0_c

There is no problem until we add together two 1s: the result (10_2) will not fit into a single bit (decimal equivalent: 5 + 5 won't fit into a single decimal digit) and that's why the last entry in the table is 0_c, where the subscripted c indicates a carry into the next column.

In fact, we can consider that every addition of two individual bits generates a carry value, which is 0 in three of the four cases. (Check back on the decimal addition algorithm in section 3.2.4.)

So a circuit to add two single bits must generate *two* bits of output, the *sum bit* and the *carry bit*, according to the following truth tables:

Sum	0	1
0	0	1
1	1	0

Carry	0	1
0	0	0
1	0	1

Now these truth tables should look familar: the one that produces the sum bit is identical to the XOR truth table, while the table for the carry bit is identical to the AND.

So, using one XOR and one AND we can produce a circuit to add two single bits and prdocue two result bits: sum and carry.

The circuit which performs this, is called a half adder (figure 7.13).

Figure 7.13: A Half Adder

But, as the name suggests, a half adder is not enough for general addition.

To see why, consider how binary arithmetic works in this example:

```
  010001
+ 100101
  ------
  110110
```

In order to get the correct answer, when we added the second column from the right, we needed to add not just the two bits from the two numbers we are adding, but also the carry from the rightmost column.

Realistically, then, we need to be able to add three bits: the two input bits and the carry in, producing, as before, the sum and carry out (which is carried in to the next column to the left).

The circuit to perform this is known as a full adder (figure 7.14) and they can be chained together to perform arithmetic of any number of bits in length.

A	B	Carry in	Sum	Carry out
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

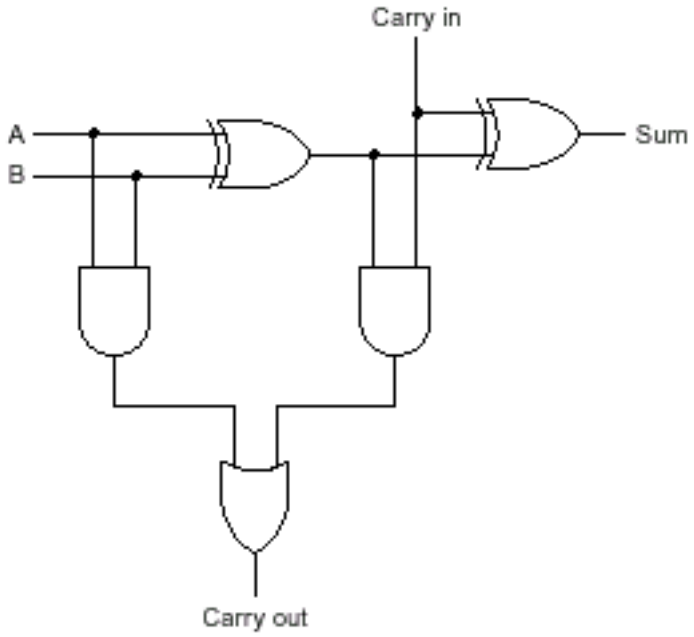


Figure 7.14: A Full Adder

Such a chained circuit is known as a ripple carry adder, because each full adder cannot produce the correct result until its carry in from the right is ready. In other words, the carry ripples along the circuit from right to left.

The circuit is built by connecting the carry out of each full adder to the carry in of its left neighbour.

A carry in of 0 is presupplied at the right end and the carry out from the left is saved. It can be tested later to see whether the addition succeeded or overflowed.

This ripple delay is not particularly bad news for 4-bit arithmetic, but for (e.g.) 32 or 64-bit arithmetic more sophisticated circuits are required. Such circuits include the carry select and the carry lookahead adders. (Which are byond the scope of this course.)

Chapter 8

The "von Neumann" Architecture

> Architecture begins where engineering ends.
>
> Walter Gropius.

8.1 Introduction

Although, as we have seen in chapter 5, the EDVAC design was not solely due to John von Neumann, his was the only name on the first draft; however, his name has been attached ever since to the basic architecture of computer systems, although, as we have already seen, Eckert and Mauchly also deserve credit.

Virtually all modern computer systems, at the highest level, conform to the so-called von Neumann architecture.

8.2 The von Neumann Architecture

Figure 8.1 shows the logical layout of a modern computer system: this arrangement of components is what is generally known as the von Neumann Architecture.

- The control unit (CU), as the name suggests, has overall responsibility for the system.
- The arithmetic-logic unit (ALU), as its name also suggests, performs arithmetic (addition, subtraction, etc.) and logic (and, or, etc.) operations.
- The registers (register file, scratchpad memory) form a small memory within the CPU. This is the *only* storage to which the ALU has direct access. To manipulate any values from memory they must first be loaded into registers.

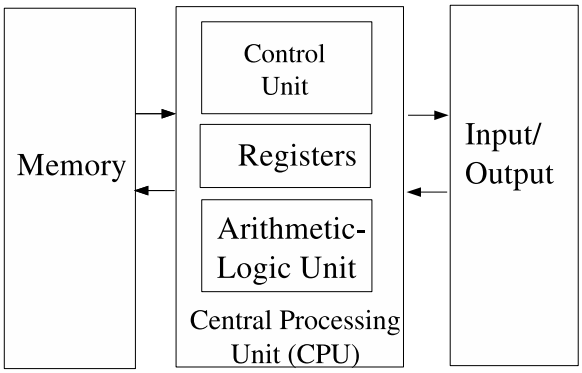


Figure 8.1: The von Neumann Architecture

- Memory (on modern computers a combination of ROM – Read Only Memory – and RAM – Random Access Memory) is where both program code and data reside while the program is in execution.

  This is the *stored program* concept, often attributed to John von Neumann, although he was clear – at least in written correspondence – that Alan Turing should really be credited.

  Memory (or Main Memory) is sometimes referred to as *Primary Storage*, as opposed to *Secondary Storage*, which usually refers to disk drives, tape drives and the like. The major difference between primary and secondary storage is access time.

- Input and Output (usually jointly referred to as I/O) are how the system communicates with the outside world – outside, in this case, signifying outside the CPU-Memory subsystem. On today's computers some I/O takes place within the computer system as a whole (e.g. to hard disks, CD-ROMs) whereas some takes place between the computer system and what the average user would consider the outside world: across a network, for example. I/O devices are somewhat beyond the scope of this course.

We shall be looking at Memory and the CPU next.

8.2.1 The Memory Subsystem

8.2.1.1 What is Memory?

Memory is where the computer stores information and (running) programs.

Memory is made up of *bits*: the bit is the smallest unit of storage, it can hold one of two values: 1 and 0 (zero).

For convenience[1] bits are grouped into larger units called *bytes*. Today the byte is universally defined as being comprised of eight (8) bits; this was not always the case and the term (invented by Dr. Werner Buchholz in 1956) has been used to describe six, eight and nine bit quantities.

Bytes are often grouped together into larger fixed-size chunks, called *words*. Unfortunately[2] the term *word*, which once referred to a computers "natural unit of processing", now invariably means a sixteen (16) bit quantity. A 32-bit chunk is variously referred to as a *doubleword* or a *longword*[3]. And 64-bits? Nobody has yet invented an accepted term for that.

An important term, though, is *cell*, usually defined as "the smallest addressable piece of memory".

What exactly does this mean?

You can imagine memory as being somewhat like a (very long) street with houses which are all exactly the same size. The computer designer's choice of a size for the memory cell is like deciding how big to build the houses: it determines how much information we can access at once (how many people can live in each house).

Historically cell size have ranged between 1 bit and 64 bits; today the cell size is almost universally the 8-bit byte.

There are two important attributes associated with every chunk of memory:

- its *address* and
- its *contents*

The address is analogous to a house's street address: it tells us how to find the cell but not what it contains. Given an address like 3380 Douglas Street we can immediately find the house, but the address *itself* does not tell us who we shall find living there.

The contents are the equivalent of the people living in the house. We shall have to look into the cell (go inside the house) to find them out: there is *no way* of knowing the contents from the address, or vice-versa.[4]

8.2.1.2 Memory Operations

We can deduce from the discussion above that memory needs to be able to do two things:

1. Deliver the current contents of a designated memory cell to the CPU.

[1] Usually that of the hardware designer.

[2] We can probably lay the blame jointly at Intel's and Motorola's feet.

[3] As opposed to *really* long words like *floccinaucinihilipification* or *antidisestablishmentarianism*.

[4] Like all analogies, we can stretch this one too far.

Properly our street would have houses along only one side, consecutively numbered, beginning – well, this *is* a computer we're talking about – at zero.

Not only that, two separate memory cells can have exactly the same contents; we don't expect to find the same people in residence in two separate houses – not at the same time, anyway.

2. Store new contents in a designated memory cell.

For the first operation (known as a *memory read*) the CPU must pass two pieces of information to the memory subsystem:

1. The *address* of the memory cell to be read.
2. An indication that a read is requested (as opposed to a *write*, see below).

The memory subsystem, having retrieved the information, must then forward it on to the CPU.

In order to carry out the second operation (a *memory write*), the CPU must pass:

1. The *address* of the memory cell to be stored into.
2. The new *contents* to be stored.
3. An indication that a write is requested.

In order to be able to achieve this, we need the following *interface* to the memory subsystem:

The *Memory Address Register* (MAR)

- which contains the address of the requested cell, the one we want either to read from or to write into.

The *Memory Data Register* (MDR) into which

- the memory subsystem places the retrieved contents of memory (read)
- the CPU places the new contents of memory (write).

A *Read/Write Line* (R/W)[5]

- used to indicate whether the required operation is a read or a write.[6]

Figure 8.2 shows the memory subsystem interface.

Our basic memory operations, then, are performed as follows:

Read

[5] A single line is capable of carrying one bit of information at a time. Which voltage level is used to signify 1 and which 0 is not relevant at this point.

[6] In practise there will be two lines, one for read and one for write. If you think carefully you will realise that it must be so, otherwise how can the memory subsystem tell when it is being asked to do something? The read and write lines will normally carry a zero then, if one of them changes to a one, the memory subsystem must react.

Another possibility is having one line for Read/Write and another for Memory Operation Requested: when the latter changes from a 0 to a 1, the memory subsystem will examine (called *sampling*) the R/W line to see which operation is requested.

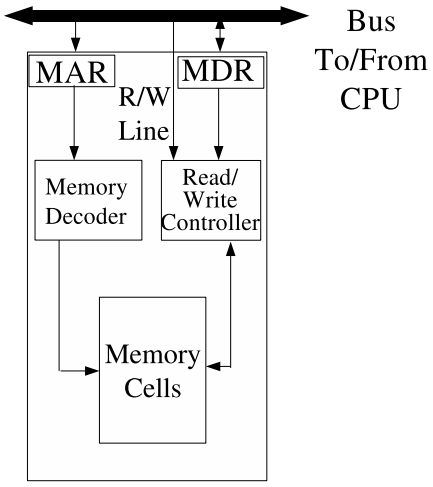


Figure 8.2: The Memory Subsystem Interface

1. CPU places the requested memory address into the MAR.
2. CPU places the appropriate value to indicate a READ[7] onto the R/W line.
3. After waiting for memory to do its work (the *memory cycle time*) the CPU retrieves the value which is now present in MDR.

Write

1. The CPU places the new contents to be stored in the MDR
2. CPU places requested memory address into the MAR.
3. CPU places the appropriate value to indicate a WRITE onto the R/W line.

Note that the CPU does not have to wait after a write.

8.2.1.3 Memory Organisation

Conceptually, memory is organised as a rectangular (usually square) array of memory cells into *rows* and *columns*.

In order to select one memory cell by address, the subsystem:

1. Divides (splits) the address into two parts, the *row address* and the *column address*.

[7] The values for READ and WRITE will *probably* be 0 and 1 respectively, but we should not assume this.

2. Using decoders, these addresses are used to select a particular row and column

3. The intersection of the row and column is the selected memory cell and it is read or written according to the value present on the R/W line.

Imagine, for a moment, that we have a spiffy new memory technology which used decimal addresses – as you've probably guessed by now, real memories use binary addresses.

Imagine, further, that our initial test system contains a massive 100 (count 'em) memory cells and that each cell can contain a single ASCII character.[8]

We can organise these cells as 10 rows by 10 columns; this is a computer, so *of course* we start counting from zero. Our memory cells, then, have addresses ranging from 0 to 99.

Here is a representation of our memory and its contents. Note that we have laid the memory out Into rows and columns.

	0	1	2	3	4	5	6	7	8	9
0	X	Y	6	F	8	G	B	9	K	N
1	G	N	B	5	9	J	F	E	C	N
2	H	J	Q	B	G	H	B	Y	B	M
3	{	h	}	[	\	b	6	g	n	|
4	#	D	5	f	^	G	&	n	*	x
5	‘	#	c	T	O	0	_	+	v	&
6	k	Y	!	P	m	v	6	7	c	0
7	(	’	*	Z	A	8	9	n	“	f
8	b	I	H	j	j	*	m	L	1	7
9	4	B	O	i	r	s	l	%	5	z

If we want to know the contents of the cell with address 74 (or "cell 74") then we look at row 7, column 4 and we find the letter "A".

Here is a more normal binary organisation: in order to save space (and typing!) this contains a mere 16 cells, organised as 4 rows by 4 columns.

The memory addresses need to range from 0-15_{10}, i.e. 0000-1111_2. Clearly, we need four bits for the addresses, so we shall have two-bit row and column numbers:

	00	01	10	11
00	01001101	10111010	00101011	11001011
01	10101011	01101011	00111101	11101101
10	00010111	00100110	01010101	10101010
11	10100110	01000001	10101011	01010100

[8] In fact, this would make each cell 8 bits or 1 byte, which is the case for most modern computer systems.

If we want to retrieve the contents of cell 13_{10}, i.e. 1101_2, then we split the binary address into two: 11 and 01, use 11 for the row address, 01 for the column address; at the intersection the desired memory cell contains 01000001, which (coincidentally[9]) is an ASCII "A".

8.2.2 The Input/Output Subsystem

In order to communicate with the outside world – by which we mean anything outside the CPU/Memory system – we need an I/O subsystem. This subsystem needs to be able to control many different types of device, for example: Screen, keyboard, printer, are all "obvious" input and output devices, but they all have very different characteristics.

For information storage (mass-storage) we shall need to employ devices such as hard drives, floppies, CD, tapes, whose operating characteristics are different again.

Mass storage devices, such as disks and tapes, can be categorised by their access method. There are two:

Sequential Access Storage Devices (SASDs) Tapes (for example, used as backup devices). The cassette tape is a good analogy: if you have 10 tracks on a tape and you want to play the 10th, then you have to play (or at least fast forward past) the first 9 tracks. This means that the further away the desired data is, the longer it takes to access.

Direct Access Storage Devices (DASDs) This includes any device on which we can access data directly without needing to Hard-drives, floppy-disks, CD-ROMs. The analogy hear is with an LP (for those who can remember them). We can go directly to (i.e. put the stylus down at) any track on the LP in (more or less) the same amount of time.[10]

8.2.2.1 Disk Drives

Both floppy and hard disk drives use a similar scheme to store and retrieve information. The difference is that the read/write in a floppy disk drive actually touches the surface; this limits the speed of rotation and also restricts the lifespan of a floppy disk.

Hard disk drives actually date back to 1955 and the IBM RAMAC. Floppy disks were invented in the 1970s (also for, but not by, IBM).

The basic idea is simple: each platter of the disk has two surfaces, each of which is divided into tracks. Each track is divided into sectors. The sector (usually 512 bytes) is the smallest unit of storage that can be written to or read from the disk.

The key numbers which tell us the performance characteristics of a disk are:

[9] If you believe this is a coincidence I have some development land in Florida you might like to consider buying.

[10] Obviously it is not *exactly* the same amount of time – we have to move the read/write head physically – but the variation in access times across the disk are far lower than those for a tape, for example.

1. The seek time, i.e. the time it takes for the read/write head to move into position over the desired track. Note that all heads move together.
2. The latency, i.e. the amount of time between the moment the head arrives at the track and the moment when the beginning of the requested sector is under the head.
3. The Maximum Transfer Rate (MTR), which specifies the number of bytes per second which can be read/written from/to the disk. The Transfer Time for one particular I/O request will be inversely proportional to the MTR.

We can therefore say that:

```
disk access time = seek time + latency + transfer time
```

Note that the seek time and latency are given as averages, because (obviously) it takes the arm longer to move farther across the disk.[11] The averages are on the assumption that – on average – the arm has to move halfway across the disk and that the head has to wait half a revolution for the sector. The seek time is a function of the speed at which the arm can move. The latency and MTR are functions of the disk's rotational speed. Typical disks today rotate at between 7200 and 10000rpm.

8.2.2.2 I/O Controllers

Compare to RAM, I/O devices are extremely slow. For example, RAM might have an access time of around 50 nanoseconds (50×10^{-9} seconds), whereas a hard drive's access time is typically around 10 milliseconds (10×10^{-3} seconds).

In other words, accessing the hard drive takes around 200,000 times as long as accessing memory.

If the CPU had to wait for peripheral I/O to complete before carrying on (and this is how the earliest machines did work) then the CPU is going to spend a good deal of its time doing nothing useful.

The solution is to use an *I/O Controller*, which is a special purpose processor which has a small memory buffer, and a control logic to control the I/O device (e.g. move disk arm).

In its simplest form, the procedure for performing an I/O request (input or output, the scheme is the same) is as follows:

1. CPU sends I/O request information to the I/O controller. This is usually done by having the controller's registers accessible to the CPU.
2. The controller takes over responsibility for the I/O request. For example, on a disk read, the controller will be responsible for moving the arm to the appropriate cylinder. Data arriving from the disk will be buffered in the controller's internal memory and then – in the scheme called Direct Memory Access or DMA – transferred to RAM.

[11] This is the variation mention above.

3. When the I/O request has completed, the controller sends an i*nterrupt signal* to CPU.

4. The CPU then processes the completed I/O.

The great advantage of this scheme (called *interrupt-driven I/O*) is that the CPU is free to do other work while I/O controller reads/writes data from/to device into I/O buffer.

8.2.3 The Central Processing Unit

The CPU is split into two parts:

- the *Control Unit* (CU)
- the *Arithmetic/Logic Unit* (ALU)

The Control Unit is responsible for the IPC, the ALU does most of the actual work.

8.2.4 The Information Processing Cycle

All computer systems today use the "stored program concept", by which we mean that the program being executed is contained in the computer's memory, as is the data the program is operating on.[12]

The Central Processing Unit[13] continually runs the following steps, known as the *Information Processing Cycle* (*IPC*):

- Fetch
- Decode
- Execute[14]

By which is meant:

- Fetch: get the next instruction from memory
- Decode: analyse the instruction and decide what is to be done
- Execute: do it

[12] As we have already seen, this concept is due to Alan Turing, a point von Neumann was happy to concede, if only in private correspondence.

[13] To be precise, the Control Unit, within the CPU, is responsible for this.

[14] In earlier days, the cycle was often extended with a fourth step: Store. In modern architectures it is no longer permissible for any instruction to access memory, only load and store instructions. Therefore the Store step is not required for the vast majority of instructions and is, in fact, the Execute step for a store instruction.

This is, essentially, a consequence of the difference between RISC and CISC processors and is rather beyond the scope of this book, although there is some discussion of the differences in chapter 9.

Figure 8.3: A Simple ALU

The IPC can easily be recast as a python algorithm, if we are prepared to gloss over a few details.

However one or two important details cannot be omitted: one is the *Program Counter* register (PC), which has but one mission in life: to contain the memory address of the *next* instruction to be fetched from memory.

Another is the *Instruction Register* (IR) which also has a simple aim in life: at any time the IR contains the instruction which has most recently been fetched from memory.

Remember that the CPU has only one type of directly-accessible storage: registers. So, when we fetch (i.e. get from memory) the next instruction *we have to have somewhere within the CPU to put it.* That somewhere is the IR.

Here is the pythonic version of the IPC:

```
run_bit = True
PC = some_initial_value     # built in
while run_bit:
   IR = fetch (PC)          # get next instruction
   PC = PC + 1              # next instruction address
   info = decode (IR)       # figure out what it is
   execute (IR, info)       # and do it - may turn off run_bit
```

The `run_bit` boolean is simply explained: most computers have an instruction called HALT, or something similar, whose purpose is to stop the CPU. Clearly in our algorithm above, if the `execute` resets this bit to False (i.e. 0) then the IPC will indeed stop.

The decode is difficult to describe in pseudocode, as it analyses the instruction and enables the appropriate circuits in the ALU to produce the result (e.g. an ADD). We shall look at the ALU next.

8.2.4.1 The Arithmetic/Logic Unit (ALU)

The ALU is responsible for what of us consider the "real work" of the computer; it is inside the ALU that the arithmetic (e.g. addition) and logic (e.g. AND) operations are performed–hence the name.

What will probably surprise you about the ALU is that it *always* calculates *every* operation it is capable of and then delivers the result asked for – this is actually faster than trying only to calculate the desired result.

The CU is responsible for setting the values on the lines that control the ALU.

In figure 8.3the ALU has two inputs, A and B, and is capable of just four operations: A AND B, A OR B, NOT A and $A + B$ (A plus B) . Internally, it has an AND gate, an OR, an inverter and a full adder.

There are two outputs: the normal output bit, X and a carry bit. Obviously (we hope) the carry bit will only ever be one if:

- The desired operation is $A + B$ and
- Adding A and B produces a carry of 1.

The two control lines are set by the CU and drive a decoder; the output lines from the decoder are the *enable lines* which select the desired output: each output from one of the functional units is tied to an AND gate together with the enable line. Only if the enable line is 1 can the AND gate possibly produce anything but zero and, as with the multiplexor, it is important to realise that even when the enable is on, all it does is ensure that the output bit is *exactly the same* as the data input.

The outputs from the AND gates are tied to an OR gate. Now it is guaranteed that only one enable line will be on (the enable lines are the outputs from a decoder, go and re-examine 7.10 in chapter 7 right now if you are unsure.)

8.2.5 The Control Unit

The Control Unit is responsible for running the Information Processing Cycle; it is, therefore, as the name suggests, in control of the entire machine.

We are now in a position to examine each phase of the IPC in more detail.

8.2.5.1 The Fetch

We know that the interface to the Memory Subsystem consists of:

1. The Memory Address Register (MAR)
2. The Memory Data Register (MDR)
3. The Read/Write control line.

And so by manipulating these – and *only* these – the CU fetches the next instruction from memory into the CPU. We also know that the *address* of the next instruction is contained in the PC register and that the arriving instruction from memory should be stored in the IR register.

The steps required, then, are as follows:

1. Copy the contents of PC to MAR.
2. Set the R/W control line to R(ead).
3. Wait.
4. Copy the contents of MDR to IR.

Step 1 places the address of the desired instruction into the Memory Address Register, which is where we are supposed to put addresses whose contents we wish to retrieve from memory.

Step 2 indicates to the memory subsystem that we want to read from memory (not write to it).

Step 3 will always be necessary, as no memory system has yet been invented that can deliver its contents in zero time. Believe it or not, this step is one of the most significant in determining the overall speed of the machine.

Step 4 puts the newly-arrived instruction into the CPU's holding area – the Instruction Register. Now the CU can examine the instruction at its leisure.

8.2.5.2 The Decode

Surprisingly (or not) the main circuit used for this stage is a decoder.

The decode examines the instruction and sets up the appropriate control lines for the ALU to carry out stage 3 of the IPC, the execute.

An instruction needs to have at least two parts:

1. Bits specifying what kind of instruction this is – these bits are called the *operation code* (*opcode*) for short.[15]

2. Bits specifying where to find the data to carry out this operation on – known as the *operand*. Modern CPUs often have up to three sets of these, which enables them to carry out the equivalent of:

   ```
   X = Y + Z
   ```

   as a single instruction (X, Y and Z will specify registers inside the CU).

   We shall examine the structure of instructions in more detail in chapter 9.

The decode needs to analyse the opcode bits and enable the ALU accordingly.

Imagine a (very) simple example machine which has the following characteristics:

1. A 2-bit opcode, allowing 4 different instructions

2. An ALU which is capable of 4 different functions.

The ALU will require four enable lines and we can drive these directly from the output of a decoder which is fed the opcode bits from the IR.

8.2.5.3 The Execute

The execute stage, by its very nature, will be different (at least potentially) for each individual instruction: some specify addition, others subtraction, others logical operations, etc.

We shall look at some examples later on which should make this clearer.

[15] In the early days of computing, the term was often *order code*.

Chapter 9

"Machine Code" and Assembly Programming

> On two occasions I have been asked: "Pray, Mr. Babbage, if you put into the machine wrong figures, will the right answers come out?"
>
> I am not able rightly to apprehend the kind of confusion of ideas that could provoke such a question.
>
> Charles Babbage.

9.1 Machine Language Instructions

A machine language instruction consists of (at least):

- An operation (or order) code, telling which operation to perform
- Address field(s), telling the memory addresses of the values on which the operation works.

Example: ADD X, Y (Add content of memory locations X and Y, and store back in memory location Y).

Assume: opcode for ADD is 9, and addresses X=99, Y=100

Opcode (8 bits)	Address1 (16 bits)	Address2 (16 bits)
00001001	0000000001100011	0000000001100100

9.1.1 Classes of Instruction

Machine code instructions generally fall into one of four classes:

1. Data transfer: operations that move information between or within the different components of the computer

2. Arithmetic: these are the operations that cause the ALU to perform a computation. Arithmetic operations (+, -, *, /), logical operations (AND, OR, NOT, XOR).

3. Compare: These are the operations that compare two values and set an indicator on the basis of the results of the compare. Most von Neumann machines have a special set of bits inside the processor called condition codes (also known as the flags register) and it is these bits that are set by the compare operations.

4. Branch: Alters the normal sequential flow of control. Enables the programming of loops.

9.1.2 Instruction Set Design

The format and type of machine instructions is not set down in stone but must be designed.

There are two broad categories of instruction set design:

- Reduced Instruction Set Computers (RISC)
  - Keep the instruction set as small and simple as possible.
  - Minimizes amount of circuitry and makes for faster computers.
- Complex Instruction Set Computers (CISC)
  - More instructions, many very complex.
  - Each instruction can do more work, but they require more circuitry and are, in general, slower than RISC instructions.

Most contemporary machines are either pure RISC (e.g. Motorola PowerPC, MIPS, Sun SPARC) or have a RISC "core" inside a CISC architecture (Intel x86, AMD Athlon).

9.1.3 Assembly Language Notation

The actual instructions that the CU fetches, decodes and executes are, of course, binary.

However, writing code out in binary is error-prone, not to mention excessively tedious.

Beginning in the late 1940s a number of computer scientists began to design *assembler* (or *assembly)* languages.

The assembler language has an essentially one-to-one relationship with machine code: each single statement (instruction) in the assembler language corresponds to a single machine code instruction.

The point of the assembler language is that it is much easier to remember to write *ADD* in a program than *000010001*.

9.2 A Simple Von Neumann Architecture[1]

The architecture in this section is ultimately based upon instructions sets used in the very earliest computers.

The salient characteristics of the machine are:

- 16-bit words
- 16-bit instructions, consisting of:
  - 12-bit operand
  - 4-bit opcode
- a single working register—the *accumulator*

The 4-bit opcode field means that we have a maximum of sixteen (16) instructions.

The fact that we only have one register means that we do not have to specify the register in an instruction.

In a modern CPU, for example, we might see instructions that look like this:

```
add r1, r2, r3
```

which (probably) means: add (the contents of) register r1 and register r2 and put the result into register r3.

Or we might see something like:

```
add fred, r1
```

to signify: add the (contents of) memory word *fred* to register r1 and put the result back into r1[2].

In each of these examples, the instruction word would have to specify not only the instruction type (the opcode field) but also the location of every operand: the registers r1, r2 and r3 and the memory word called *fred*.

In our simple machine we never need to specify which register we mean, *because there is only one.*[3]

This will save space in the instruction word, but make programming rather more difficult.

---

[1] Based on the example in Schneider and Gersting.

[2] Although you should be aware than several assemblers do things "backwards": in MASM, Microsoft's assembly language for the Intel IA-32 (x86) chips:

ADD Fred, BX means add the the contents of the memory location *Fred* and the BX register and store the result *back into memory* at location *Fred*.

[3] To be precise, there is only one programmer-accessible, or program-visible register. Every CPU needs several registers that the user cannot touch: e.g. MAR, MDR, PC and IR.

9.2.1 The Instruction Set

In the description of the instruction set given below, the notation

```
M[address]
```

signifies the *contents* of the memory word at *address,*

```
Acc
```

indicates the accumulator register and

```
->
```

indicates "replaces".

Opcode	Instruction	Semantics
0000	LOAD X	M[X] -> Acc
0001	STORE X	Acc -> M[X]
0010	CLEAR X	0 -> M[X]
0011	ADD X	Acc + M[X] -> Acc
0100	INC X	M[X] + 1 -> M[X]
0101	SUBTRACT X	Acc - M[X] -> Acc
0110	DEC X	M[X] - 1 -> Acc
0111	CMP X	Compare M[X] to Acc and set flags (condition codes) accordingly.[4]
1000	JUMP X	X -> PC
1001	JUMPGT X	if GT (is true) X -> PC
1010	JUMPEQ X	if EQ (is true) X -> PC
1011	JUMPLT	if LT (is true) X -> PC
1100	JUMPNEQ	if EQ (is false) X -> PC
1101	IN X	Input integer store as M[X]
1110	OUT X	Write out M[X] as a decimal number
1111	HALT	Stop CPU

Once again, remember that the *only* place in the CPU that you can store anything is in the accumulator register.

If we want to add two numbers from memory together and have the result in memory, this will take three instructions: LOAD, ADD and STORE.

If you are getting the distinct impression that doing *anything* in machine code takes a lot of work (by the programmer) then you are dead right.

Adding the ability to give a name to a memory location, we have an assembly language that we can use to write programs[5].

[5] The TASS assembler, which can be found at http://turing.cs.camosun.bc.ca/COMP112/resources/tass.zip can be used to write programs for this CPU and will run all the examples in this chapter. It will run on any system which has python installed.

Here, for example, is a programme to input two numbers, add them together and output the result. Comments (everything after the #) explain the code[6].

```
        IN    X       # Get first number
        IN    Y       # Get second number
        LOAD  X       # First number to Acc
        ADD   Y       # Add second number to Acc
        STORE Z       # Store sum
        OUT   Z       # Print sum
        HALT          # Stop
X       DATA  0       # First number stored here
Y       DATA  0       # Second here
Z       DATA  0       # Sum here
```

[6]Writing assembler without comments should be an offence punishable by something extremely nasty indeed.

Part V

Software

Chapter 10

Software

10.1 Categories of Software

Software is the whole point of computers. Nobody would spend hundreds (thousands? tens of thousands? insert number of your choice here) of dollars to buy a HiFi system just to look at. Its purpose is to play sound (music, speech, etc) and to do that requires CDs, tapes, or some other source – even if it is MP3s.

In computer terms, we need *software* to run on the computer to make it do what we want[1].

Software is usually divided into two basic categories: Systems (sometimes System) Software and Applications.

10.2 Systems Software

Put simply, systems software is made up of programs that you don't have any great desire to run, but which would make your life a whole lot more difficult if they didn't exist.

OK, let's put it a bit less simply.

System software is divided into two sub-categories: *operating systems* and *utilities*.

10.2.1 Operating systems

You probably know that computers are controlled by a program called the *operating system*.

This was not always the case. In fact, it was only in the mid-late 1950s that the first *monitor systems* began to appear. These enabled the programmer/user/operator of the machine to automate certain common procedures, e.g. running the FORTRAN compiler (which, in those days, would have been on a

[1] Interestingly, in the last couple of decades, the music and film industries have also started to use the term software to refer to the films or albums they market.

tape). In fact one of the earliest proto-operating systems was FMS (Fortran Monitor System), developed by North American Aviation for the IBM 709.

The name *monitor* came about because the high cost of leasing and running the early mainframe computers meant that company accountants wanted some way to monitor who wa using the machine and for how long. The ability to keep track of such information was an important aspect of monitor systems.

Jobs could now be "batched" together. A typical early jobstream might look something like this:

```
$JOB user_spec;           identify the user for accounting
$FORTRAN;                 load the FORTRAN compiler
source program cards
$LOAD;                    load the compiled program
$RUN;                     run the program
data cards
$EOJ;                     end of job
$JOB user_spec;           identify a new user
$LOAD
binary application cards (punched earlier)
$RUN
data
$EOJ
```

The first genuine *operating system* is generally considered to be the *General Motors Operating System* (guess which company built it) for the IBM 701 in late 1955 or early 1956.

*O*perating systems were so called because they automated a number of functions which previously had been done by a (human) *operator*. Notable among these functions was that of *scheduling*, i.e. deciding which, from a queue of jobs, should be run next.

Most early (up to the mid-1960s) systems worked exclusively in *batch mode*, meaning that programs, usually on punched cards or tape, were gathered together into batches and submitted to the machine. Output would be hard copy, usually printed.

By the late-1960s many of the major functions of an operating systems were established, these included:

- *Multi-programming*: the ability to have more than one program in memory simultaneously, swapping the CPU between them. This works (still) because the CPU is so much faster than even the fastest hard disk (even the fastest memory, come to that, although this problem is solved by very different methods) so programs spend a lot (well over 99% typically) not executing instructions but *waiting*, for data to arrive. Which is how multi-programming (and its later sibling, multi-tasking) make things more efficient: while one program (or task) is waiting for data, another can use the CPU. It looks, to humans, who are so much slower than computers,

as if two or more programs are running simultaneously – they're not, but we can't tell, because the switching from one to another is so fast.

- *Filesystems*: a way of referring to data held on peripheral devices, such as a disk. Before filesystems the programmer had to know precisely where on a hard disk a file was located. The most advanced filesystems were *hierarchical*, in which files could be contained in directories, which could be contained in other directories, etc.[2] There were even filesystems which would automatically archive files which had not been accessed in a certain amount of time to tape, bringing them automatically back to disk when they were once again accessed – although the access time was very large in this particular case!

- *Timesharing*: although this term is sometimes used to refer to the process of switching the CPU between different *processes* (essentially a running program), it was used mostly to refer to systems which enabled users to interact directly with the computer via TTYs (TeleTYpe machines, which were essentially the same devices that were used to send/receive telexes) or VDUs (Visual Display Units), basically a screen, like a TV, with a keyboard attached. These early *terminals* mimicked the operation of the TTY, which had a roll of paper on which keyboard input and computer responses appeared, in the sense that it simply scrolled down a line every time Enter was pressed, with no way to move the cursor and certainly no graphical capabilities – it was a text-only world.

- *SPOOLing*: (from Simultaneous Peripheral Operation OnLine) a way whereby, for example, two programs both wanting to print were able to write their output into a SPOOL file (on disk) and it would be printed when the printer was free. A program's output could, therefore, be printed some time after it had finished. It was also possible to feed jobs punched on cards into an input SPOOLing area, from where they would be selected by the scheduler for running, on the basis of policies established by the site.

- *Security*: a vast subject, initially it meant little more than ensuring that two programs in memory simultaneously could not, either accidentally or maliciously, interfere with each other. Many security features required *hardware assist*, in other words the CPU needed certain features in order to make the security work at all[3]. Later systems allowed user with differing security requirements (and clearances) to use the system simultaneously.

- *Virtual Memory*: the first such system was the Atlas, designed and built in Manchester, which introduced the concept of *paging*, which allowed a

[2] In fact the first hierarchical filesystem was on CTSS, MIT's pioneering timesharing (1961).

[3] The simplest example is the so-called *base and limit registers*, a feature on multiprogramming-capable OSes: every memory reference by a program is checked against the limit register, to make sure it is referring to its own memory, then adjusted by the base register, so that any program may run at any location in memory.

program which as bigger than physical memory to run. Most modern OSes include some form of virtual memory – once again, hardware assist is required.

- and much more...

Today's operating systems also provide (usually) two styles of User Interface, text-based (often referred to as a CLI - Command Line Interface) or graphical (a GUI, or Graphical User Interface) and networking facilities.

10.2.2 Utilities

An operating system, by itself, is not a great deal of use – the whole point of an OS is that it controls the hardware and offers a convenient interface to complex functions (such as reading files from a filesystem) *for programs to use*.

In the early days utilities typically included language compilers, linkers, loaders and things that we would probably not even think about today for example a program to copy one file to another.

This last may seem very strange, but consider: using a batch operating system, with no interactive capability (so, no copy "command" or graphical file manager), just how *would* you copy one file to another? Without the bundled utility program, you would need to write a custom program yourself – perhaps many of them, one for each different file format.

Until 1969, when you bought (more likely leased) a computer the operating system, utilities and a certain amount of training came with it.

In 1969 IBM created the software and services industries when they *unbundled* these features; no longer did the OS, utilities and training come "bundled" with the hardware, they were now seperately-priced items.

Although not all manufacturers followed suit immediately, most did eventually.

One definition of a utility program today might be something like: *an application that is so commonly used that it is not considered to be a seperate application, but an essential feature.*

One interesting example of this is a DataBase Management System, or DBMS.

The first databases appeared in the 1960s and were initially *hierarchical* in nature.

There soon followed *network* databases (e.g. Bachman's IDS and IDMS), which dominated until the late 1980s.

In the 1970 E.F. ("Ted") Codd, working at IBM, described a *relational* database system, but the first implementations would not appear for almost a decade.

Today relational DBMSs dominate.

Interestingly the DBMS, which undoubtedly came under the heading of utility between 1965 and, say, 1990 or 1995, today can stray over the major boundary into the world of applications: partcularly in the case of a product like

Oracle which is not only a DBMS but also has tools for designing and building applications.

10.3 Applications

Every program which does not come under the heading of Systems Software is an application.

The earliest applications were scientific – and programmed by the user!

In fact, well into the 1970s applications were programmed *in-house*, in other words, by employees of the company which wanted to run the applications.

Of course this led to a great deal of duplication of effort – most large companies, for example, ran a *payroll* system, which they had written themselves[4].

Thus was born the idea of the *package* system, a generic solution which could be customised by the user. Payroll systems were some of the earliest such packages.

The application-as-product only really took off with the advent of personal computers in the late 1970s and early 1980s.

Except that the price of the software package had to drop dramatically in order to seem reasonable in comparison with the cost of the computer itself: when your mainframe computer cost in the millions of dollars, purchasing a software package for tens of thousands did not seem unreasonable.

If the computer system only cost $5,000 (like the original, 1981 IBM PC) then buying a package which cost an order of magnitude more than the hardware *did* seem unreasonable.

The first *killer app*, by which we mean an application which is so desirable that it can often be used to justify the purchase of a computer just to run it, is almost certainly Dan Bricklin and Bob Frankston's *Visicalc*, the first-ever spreadsheet program, which ran only on the Apple II.

Other killer apps from the 1980s included Wordstar and Lotus-1-2-3, both for IBM PCs.

Today we speak of *application domains*, i.e. areas which may be addressed by applications, and the search is always on for new ones.

Today certain apllications are almost taken for granted as necessary for a small (i.e. personal or small business) computer; these include a spreadsheet, a word processor, a *presentatino manager* (e.g. Powerpoint), a database (which, unlike the earliest DBMSs will also include tools for designing and creating databases and applications), a drawing package, a graphics package (these two are not the same) and – presumably not so much for small businesses – a media player for video and audio files.

We also expect to pay far less for software than was the case twenty-five years ago. This is not a problem for the software industry: a quarter of a century ago

[4]One company the author worked for in the 1970s had written their payroll system no fewer than three times.

Unfortunately, the 1970 and 1974 rewrites were both abandoned and so the company continued to run the original, 1965, system until well into the 1980s.

a new software application might have a potential customer base measured in the thousands; today it will be in the millions.

Chapter 11

Programming Languages

11.1 Types of Programing Language

There are two ways in which programming languages are categorised: as *High* or *Low Level* Languages and according to the *programming paradigm* into which they fit.

11.1.1 High and Low Level Languages

Generally speaking, *High Level Languages* are ones which are far removed from the way in which the computer actually functions and *Low Level Languages* (basically, assembler) are very close to the way the machine works.

You will often find it said or written that high level languages are "closer to human languages".

Oh really?

Here is a brief extract from a program in Erlang, a very high level language:

```
map (_, []) ->
    [];
map (F, [H|T]) ->
    [F(H)|map (F, T)].
map (_, _, []) ->
    [];
map (M, F, [H|T]) ->
    [apply (M, F, [H])|map (M, F, T)].
```

Does that seem to resemble a human language to you?

But it clearly doesn't resemble the way the computer actually *works* either, does it? We say that there is a large *semantic gap*[1]

[1] *Semantics* is the study of meaning; the semantics of a programming language are the definition of what a program *means*, i.e. the effect of running the program.

The brief Erlang extract on this page is comprised of just 10 lines of code – it is actually

Here is a brief extract from a program written in CHASM, an assembler for the CRAPS CPU.

```
recheck
    cmp   r1, r2
    bae   palindrome
    lodbu 0, r1, r4
    lodbu 0, r2, r5
    cmp   r4, r5
    bne   notpalin
    inc   r1
    dec   r2
    br    recheck
```

Clearly this doesn't resemble any known human language either, but it *does* resemble the way in which this particular computer works: every line – except for the first – corresponds to *exactly* one machine instruction. The semantic gap is essentially zero[2].

11.1.2 Low Level Languages

By "low level languages" we really mean assembly (assembler) languages.

The distinguishing features of a low level language are:

- Each line of code corresponds to a single machine instruction on a *particular machine.*
- Each different CPU requires its own assembly language.
- The language uses *Instruction Mnemonics* to stand for the individual instructions e.g. *add*, *br*.
- Programs are difficult to write, difficult to get working and take a long time to develop.

So, are there any advantages to using low level languages?

Well, compared with what went before – writing *machine code* in binary or perhaps hexadecimal (the Manchester BABY and Mark I computer was programmed using base-32) – writing assembler is simple.

If you don't believe me, here is the assembler example from above, but in (hexadecimal) machine code. is this any easier to understand?[3]

---

a single function definition made up of four *clauses*. But the low-level language equivalent would be dozens, if not hundreds of lines long.

[2] I have cheated a little by removing the comments from both code examples. The names left in the second example provide a small clue to its purpose, but otherwise, without comments, this is more-or-less incomprehensible, even to somebody who knows the assembler well.

The Erlang code, on the other hand, may seem like gibberish, but if you know Erlang, then the code is almost self-explanatory.

[3] Hint: of course it bloody isn't!

```
00c22000
81400190
d2020000
d6040000
00c85000
81000198
04830001
08c50001
80000168
```

Also, if you want to extract every last ounce (gram?) of performance from the system, then assembler is almost certainly the way.

However, before you decide to start coding in assembler, consider:

- Your code may be more efficient than a competitor's, but the competitor (assuming them to be using a high level language) will probably get their product to market long before yours.
- Your code will not be *portable*, i.e. to get it to work on another computer system will require a rewrite.
- Your code is more likely to contain bugs than your competitor's.
- Your code will be far more difficult to *maintain* (i.e. fix bugs and add new functionality) – it's the nature of assembly programming.

In other words, you really, *really* need a good reason to code in assembler today.

11.1.3 High Level Languages

The history of programming languages is a long and convoluted – although fascinating – one.

It was in the late 1940s that the computing pioneers realised that programming in machine code was not efficient[4] and the first assembler languages (often called things like "autocode") were invented.

But it was soon realised (and some pioneers like Alan Turing and Karl Zuse had realised this even before there were working computers) that something higher level than assembler, something further away from the way the physical machine worked and closer to something recognisably human in origin, was required.

The first high level languages appeared in the 1950s, these were:

- FORTRAN (FORmula TRANslator) 1954-7, built for the IBM704 by a team led by John Backus.
- LISP (LISt Processor) 1959, built, also on the IBM704/9 at MIT by a team led by John McCarthy.

[4] By which we mean not efficient in terms of time spent writing code.

- COBOL (COmmon Business Oriented Language) 1959, defined by an industry committee and based on earlier work done by, inter alia, Grace Hopper.

Within a few years, as we have seen, there were hundreds more languages in use. However, the first three "have legs" and are still in use, albeit greatly changed in many ways, today.

11.2 Programming Paradigms

A *programming paradigm* is a way of thinking about programming, a programming *style*, if you like, although when programmer's refer to their "style" they are usually talking about indentation, naming conventions etc. Programming paradigms are much, much more than that.

We'll briefly look at each of the paradigms, in the order of their appearance.

11.2.1 Imperative Programming

The earliest programming languages were assembly languages and their form was entirely dictated by the CPU architecture. We have seen a simple assembly language in chapter 9.

The earliest high level languages essentially generalised the Information Processing Cycle and von Neuman architecture: programs run sequentially, from one statement to the next, unless the flow of control is diverted by a conditional, a loop (or even, in pre-1970 languages, a jump or goto).

A crucial aspect of imperative languages is the storing of values in locations: i.e. *assignment*.

Moreover, the programmer has to specify exactly *how* the algorithm is to be carried out by the computer.

Two of the first three high level languages – FORTRAN and COBOL, both still in use – were imperative languages, as indeed were the majority of programming languages created until the 1990s (at the earliest).

The most commonly used imperative language today is almost certainly C, although the number of lines of COBOL still in use is large and unknown[5].

11.2.2 Functional Programming

The functional paradigm views every program as equivalent to a mathematical function, which transforms its input into its output.

In a functional language, for the first time, the programmer only specifies *what* is to be done, but not *how* to do it.

Most functional languages do *not* have assignment statements and do not have looping constructs – repetition is achieved by using recursion. The major

[5] In a late-1990s study for the Y2K problem it was discovered that there were 800 million lines of COBOL code still in use – in the 22 square miles of Manhattan alone...

(often only) data structure is the list which is defined recursively, so instead of saying that a list is "an ordered collection of elements" the functional definition of a list is:

> A list is either *empty* or
> It has two parts: a head, which is an element and
> A tail, which is a list.

They also treat functions as *first-class values* which means that a function may be passed as a parameter or returned as the result of another function.

Many computer scientists believe that functional programming is the best way to write reliable error-free code[6].

The first functional language was LISP, which is still in use today. ML, created for an automatic program-proving system, is perhaps the most influential, reviving interest in the paradigm. Haskell is the functional language of choice for many today, with Erlang and up-and-coming alternative.

11.2.3 Object-oriented Programming

In OOP, as it is often called, the world is viewed as a collection of *things*, called *objects* (well, it does sound better, doesn't it?).

An object not only contains data, but also code to manipulate it. In OOP we do not write the code to display, say, a cube on the screen, we ask the cube object to *display itself* on the screen.

Other key concepts of OOP are the *class*, which is a sort of template for creating objects from and *inheritance* (which goes hand-in-hand with *subclassing*).

Put simply we define a parent (or super-) class and then *derive* subclasses from it; the parent defines everything which is common to all subclasses; the subclasses define what is specific to themselves.

To use an often-cited example: a class called *Animal* could have subclasses *Cat*, *Dog* and *Mouse*. Every animal in our hierarchy can make a sound (*speak*) but each one makes a different sound, so when we ask a *Cat* to speak, the sound we expect is "miaow", etc.

The first object-oriented language was Simula-67, the most important was Smalltalk (which, apart from anything else, brought us the world's first GUI) and the commonest today are C++ and Java.

[6] The Ericsson AX301, a digital telephony-over-ATM switch, is programmed with 1.25 million lines of Erlang, a functional language designed by Ericsson. During trials for British Telecom, the device reach "nine nines" reliability, i.e. was up 99.9999999% of the time, which allows for 31 milliseconds of downtime *per year*.

Part VI

Formal Models

Chapter 12

The Formal Approach: Models and Notation

This Year's Model

Song title by Elvis Costello

12.1 Introduction

Everything we have been discussing is about algorithms: the concepts, the correctness, the implementation, etc.

All of this, though, must be based on some *formal* approach, something that can be manipulated mathematically, to allow us to *prove* things about algorithms and programs.

In this chapter we look – briefly – at two formal models of computation and at a notation system for defining (one aspect of) programming languages.

Eventually we shall see an outline of the proof that there are some problems that no computer can solve – not now, not ever.

12.1.1 Models

Models[1] are very important to help our understanding of various objects and their behaviors. A model must capture the essence, and the important properties of the real thing, while suppressing the unnecessary details.

A model could be either physical, like a model car, or mathematical (we can use an equation to model the driving distance of a vehicle).

There are problems that do not have any algorithmic solution (this is *not* the same as saying that there are problems for which no algorithmic solution has yet been found, but for which such a solution may exist, if we were sufficiently clever to discover it).

[1] No, *not* Claudia Schiffer or Kate Moss.

In April 1984, TIME magazine claimed: "Put the right kind of software into a computer, and it will do whatever you want it to. There may be limits on what you can do with the machines themselves, but there are no limits on what you can do with software."

As we shall see in this chapter, this is arrant nonsense and anyone with a basic knowledge of computer science – such as yourself, after reading this book – can easily prove that it is.

The algorithmic problems we wish to discuss in this chapter are such that no amount of money, time or brains will suffice to yield solutions.

We still require and/or can give: termination for each legal input even allowing any amount of time (algorithm can take as long as it wishes on each input, but it must eventually stop and produce desired output) algorithm can be given any amount of memory it asks for.

12.1.2 Features of the Computing Agent

A computing agent must be able to follow the instructions in an algorithm notwithstanding the format of the instructions, so the CA must be able to

- read instructions
- decode instructions (decide what they mean)
- ask for and use pertinent data (input)

A computing agent, then, carries out an algorithm reacting both to "input" and to the "present state".

A computing agent is expected to produce output – the outcome of an algorithm must be an observable result.

12.2 Finite State Automata

The Finite State Automaton[2] or *Finite State Machine* is an easily-learned, easily-understood model for computation.

Indeed, the FSM is often used to describe the operation of something which is not a computer at all, but a (mechanical) machine.

As we shall see.

An FSM is defined by:

- A set of *states* indicated by circles
- A set of input values which cause the *transitions* from one state to another

[2] *Automata* is the plural.

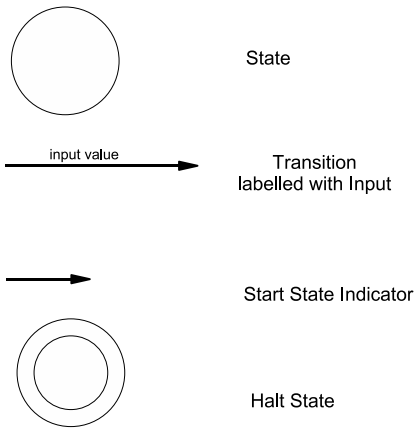


Figure 12.1: FSM Symbols

  - the transitions are shown as directional arrows joining two states (joining the same state to itself is perfectly legal, but not used in this machine)
  - the input values are shown as labels along the transitions

- An initial *start state* indicated by an incoming arrow
- A final *halting* or *stopping* state, indicated by a double circle.

12.2.1 A Simple Example – A Turnstile

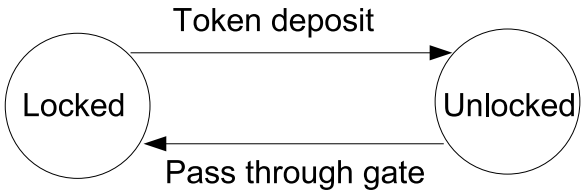


Figure 12.2: Fine State Machine for a turnstile.

Figure 12.2 shows the FSM for a turnstile.

The gate is opened by inserting a token – so there is no need to worry about exact amounts or giving change.

Passing through the gate locks it for the next person.

Note that there is no start state–although presumably on the day that the turnstile is installed it will originally be in one state or the other (probably closed).

Neither is there any stop (halt) state for this FSM – although, again,the turnstile must be in one of the two states when it is finally decommissioned (removed).

In order to conform to our definition of the FSM, we could arbitrarily designate one of the states as the start state and one (possibly the same one) as the stop state.

12.2.2 A More Complex Example – A Vending Machine

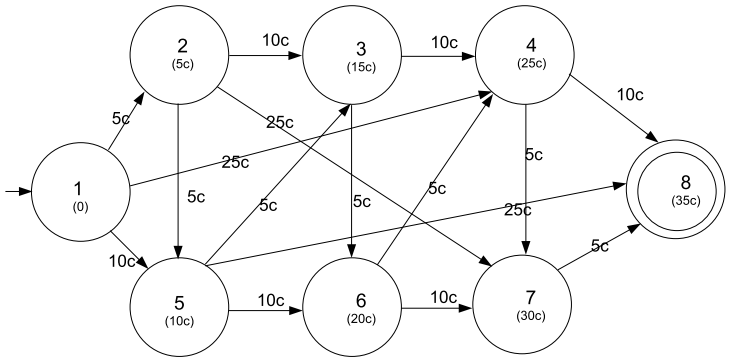


Figure 12.3: FSM for a simple vending machine.

Figure 12.3 shows the FSM for a simple vending machine.

- The machine accepts nickels, dimes and quarters only.
- The exact amount (35c in this case) must be tendered.

A more realistic machine would, at the very least:

- Reject invalid coins (pennies, drachmae, rupees)
- Give change when too much money is tendered

But this would make the FSM considerably more complex.

Note that there are many different paths through this machine to the stopping state, but that *all of them* involve depositing precisly 35 cents – no more and no less.

12.2.3 Uses of FSMs

Finite State Machines may seem of only passing interest, but they are, in fact, a very powerful tool.

12.2.3.1 Regular Expressions

A *regular expression* is a string of characters typically used in an editor program – or something like the unix *grep* command – to match patterns of characters within a file.

Regular expressions are precisely equivalent to FSMs: for any regular expression there is an equivalent FSM and vice-versa.

Suppose, for instance, that we want to match strings which are made up of:

- the letter a, followed by
- any number of b's or c's and then
- a d.

The regular expression for this, as used in unix editors, *grep*, etc. is:

```
a[bc]*d
```

where the brackets – [and] – indicate any of the characters within the brackets and the asterisk[3] – * – indicates any number, *including zero* of the character before (in this case, either b or c).

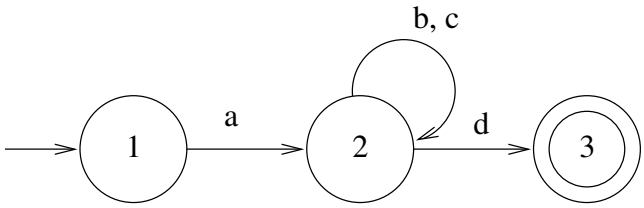


Figure 12.4: Finite State Machine for a[bc]*d

The FSM for the regular expression is shown in figure 12.4. The final state is known as the *accepting state* because if we reach it, we are accepting the string as valid.

The FSM in the diagram is said to *recognise* the regular expression.

Programs which use regular expressions will typically "compile" the regular expression at run time and build an FSM to implement it.

Let us see how (some of) this works.

Here is the python program function which will validate a string against this particular regular expression.

The function *valid* is written as an FSM.

Note, in particular, how easy it is to deal with errors: if an unexpected character is found (unexpected for the particular state we are currently in) then the string doesn't match the regular expression.

Nor does it match if we run out of input before we reach the accepting state.

[3]Known in language theory as the *Kleene Star* after logician Stephen Kleene.

```
#!/usr/bin/python
#
# An example FSM to validate the regular expression
#       a[bc]*d
#
# Deryk Barker, August 30, 2007
#
def valid (str):
    state = 1
    pos = 0
    while state != 'OK' and state != 'Error' and pos < len (str):
        char = str[pos]
        # quit when we know the result or fall off end of string
        if state == 1:
            if char == 'a':             # valid input
                state = 2               # next state
            else:
                state = 'Error'         # invalid
        elif state == 2:
            if char == 'b' or char == 'c':
                                        # valid input
                state = 2               # stay in this state
            elif char == 'd':           # also valid
                state = 'OK'            # final "accepting" state
            else:
                state = 'Error'
        else:
            state = 'Error'             # should never get here
        pos = pos + 1                   # next input character
    if state == 'OK':                   # valid string
        return True
    else:                               # invalid - bad char or too short
        return False
```

And here is the rest of the program, which allows us to type strings on the command line:

```
if __name__ == '__main__':
    import sys
    for arg in sys.argv[1:]:            # process all strings
        if valid (arg):
            print 'string', arg, 'is valid'
        else:
            print 'string', arg, 'is not valid'
```

And here is the result when run:

```
python fsm.py string1 abbbcccd abcbcbccccbbbbe abc
string string1 is not valid
string abbbcccd is valid
string abcbcbccccbbbbe is not valid
string abc is not valid
```

Although FSMs and regular expressions are very useful tools, they are not without their limitations.

For example, it is impossible to define a regular expression – or build an FSM – to recognise strings which consist of, for example, a number of a's followed by *the same number* of b's.

There is a hierarchy of models of computation, classified by the complexity of the languages they recognise. The FSM is at the bottom of this hierarchy.

We shall encounter the top of this hierarchy later in the chapter.

12.3 Defining Language Syntax – BNF

12.3.1 Notation systems

Every formal system must have some form of notation; a well-designed notation can make life easier, a poorly-designed notation can make things more difficult.[4]

One of the major topics in computer science is defining a programming language in a formal way. There are two parts to a language: its *syntax* and its semantics.

12.3.2 What is syntax?

Syntax is the name we give to the rules for constructing, e.g., programs in a particular language. It is important to distinguish between the *syntax* and the *semantics* of a programming language.

- Syntax tell us how to put together valid programs

  It is not difficult to define the syntax of a programming language clearly and unambiguously, as we shall soon see.

- Semantic tells us what the program *means* (usually, what it *does*)

  It is far more difficult to define the semantics of a language. At least three different systems have been proposed. None have been accepted as appropriate to every language.

It is entirely possible to write programs which are syntactically correct but semantically meaningless, or incorrect.

These programs will usually compile without error but will either fall over with an error when run or will produce the wrong (or no) result.

[4] Newton and Leibnitz, for example, used very different notation for calculus, which they invented independently of one another. The notation used today is that of Leibnitz.

For an English-language analogy, here is a sentence[5] which is syntactically correct – i.e. conforms to the "rules"[6] for constructing English sentences – but semantically meaningless:

> Colourless green ideas sleep furiously.[7]

12.3.3 Context Free Grammars

Most programming languages' syntax can be defined as a *Context Free Grammar* (CFG).

Of the various ways of representing syntax the easiest and most commonly-used is *BNF* (*Backus-Naur Form*).

A working knowledge of BNF is a very useful skill for any computer technologist to acquire.

A CFG definition consists of:

- A set of *Terminal Symbols*: these are the elementary symbols permitted in the language, keywords, constants, etc. In python, for example, words like *while*, *for*, *def* etc. would be terminals, as would constants like 'spam', 5 and 3.14159260.
- A set of *Nonterminals*: these are the constructs permitted in the language. For instance, in a BNF description of python the *while statement* (not just the word while*)* would be considered a non-terminal.
- A set of *Productions: a production has two sides:*
  - The Left side consists of a single nonterminal.
  - The Right side consists of a sequence of zero or more terminals or nonterminals.
- A single *Starting Nonterminal*: representing the main construct of the language. In many languages this is called *program*.

12.3.4 BNF notation

For the plain, or "vanilla", version of BNF the rules are:

- Non-terminals are written inside angle-brackets < and >
- The exclusive or is written as |
- Production rules are written as

[5] Due to Noam Chomsky, the most influential language theorist of the second half of the 20th century.

[6] There are, of course, no such rules for English, despite what some teachers, particularly in the USA, think.

[7] Feel free to assume a poetic meaning for this sentence. Alternatively, substance abuse may also seem to give this meaning.

```
LHS ::= RHS
```

- The empty string written as <empty>

12.3.4.1 An example – a school week

```
<Week>       ::= <RestOfWeek>
<RestOfWeek> ::= <Day> <RestOfWeek> | <empty>
<Day>        ::= <RestOfDay>
<RestOfDay>  ::= <Session> <Day>
<Session>    ::= <Classroom> | <Lab> | <Free>
```

The first and third productions look a little odd at first, but there is now way, in "vanilla" BNF to specify repetition and so repeated items must be dealt with using recursion.

The second production, for example, says that[8]:

> The rest of the week consists of a day followed by the rest of the week – think of this as "the rest of the rest" if it helps – or it is empty.

So, a week composed of Monday, Tuesday, Wednesday, Thursday and Friday (you go to school on the weekend if you want to), would be analysed initially, as:

```
   <Week>
      |
      |
 <RestOfWeek>
     / \
    /   \
<Day> <RestOfWeek>                          (Monday)
         /\
        /  \
     <Day> <RestOfWeek>                     (Tuesday)
              /\
             /  \
          <Day> <RestOfWeek>                (Wednesday)
                   /\
                  /  \
               <Day> <RestOfWeek>           (Thursday)
                        /\
                       /  \
                    <Day> <RestOfWeek>      (Friday)
```

Note that:

[8] This is probably the best way to read it to yourself as well.

- this diagram would be a *lot* bigger if we included the breakdown of each day into sessions
- as we split off each day, the rest of the week changes – as it does in fact: on Monday the rest of the week begins with Tuesday; on Tuesday it begins with Wednesday.

12.3.5 Extended BNF

"Vanilla" BNF is quite sufficient, but it is often more convenient to be able to specify alternatives, groups of symbols and repetition (rather than recursion) – hence *Extended BNF*, which permits all of those things.

In Extended BNF:

- Non-terminals are written as words, with no < or > surrounding them.
- The exclusive or is still written as |.
- Production rules are still written as:

```
LHS ::= RHS
```

- Parentheses, ( ), are used for grouping constructs together–always used in conjunction with |, { } or [ ].
- Braces, { }, are used to indicate a sequence of *0 or more* occurrences of the construct inside the braces.
- Brackets, [ ], are used to indicate that construct inside them is optional.

12.3.5.1 The school week in Extended BNF:

As you can see, extended BNF can make our definitions – i.e. our set of productions – considerably shorter:

```
Week    ::= {Day}
Day     ::= {Session}
Session ::= Class | Lab | Free
```

12.3.6 Expressions

Arithmetic expressions – e.g. 2 + 3 * 5 – occur in virtually all programming languages.

Here is a BNF description of expressions, as used in mathematics and most programming languages:

```
<expr> ::= <expr> + <term> | <expr> - <term> |<term>
<term> ::= <term> * <fact> | <term> / <fact> | <fact>
<fact> ::= <number> | ( <expr> )
```

Note that we have reverted to plain BNF, at least in part because, by their very nature, expressions are recursive: as indicated by the last production, which says that one possibility for an expression is a parenthesised[9] expression, which takes us back to the first production.

Another way of looking at it is that a *subexpression* is merely an expression contained within another expression and that the subexpression can be arbitrarily complex.

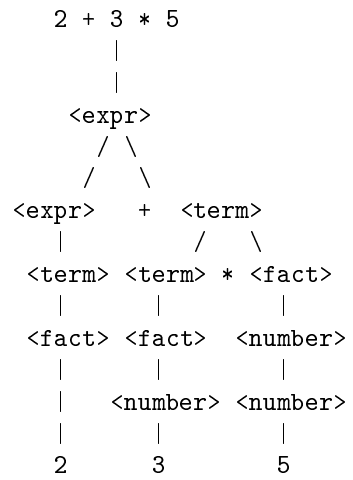
The BNF also captures the precedence of multiplication and division over addition and subtraction.

12.3.7 Parse Trees

A *Parse Tree* is created by a compiler using the syntax definition. We have already seen an extract from a parse tree in the discussion of the school week above.

The parse tree is a marvellously useful data structure which can be used to generate code or to evaluate the expressions.

Here is a very simple example, an expression and its parse tree, based upon the BNF in 12.3.6:



12.4 Turing Machines

The Turing machine is the primary formal model for a computing agent.

A Turing Machine:

- has a tape extending infinitely in both directions that is divided into cells, with each cell able to hold one symbol

[9] Bracketed, if you insist.

- has a read/write head that can read and write the contents of one cell, the read/write head can move one cell to the left or one cell to the right.
- has a set of internal states[10]
- has a finite number of cells on the tape that are not blank
- can only read and write a finite set of symbols on the tape
- can only assume one of a finite number of internal states.

The Turing Machine is usually denoted by a 5-tuple of:

```
(current state, read symbol, next state, write symbol, direction of move)
```

e.g.

```
(1, 0, 1, 2, R)
```

Note that the output and the next state depend on both the current input and state.

Some versions allow either a move or a write symbol, but not both. There is no loss of generality either way.

Is a Turing Machine an appropriate model for a computing agent? Well, it can

- accept input
- store information in and retrieve it from memory
- take actions according to algorithm instructions; action taken depends on present state and input item being processed
- produce output

Which conforms to our requirements above.

However, there are still a few potential difficulties to consider.

- What if there is more than one instruction that applies to the current configuration of (current state, current symbol)?
- What if there is no instruction that applies to the current configuration of (current state, current symbol)?
- What is the initial configuration?

By *convention*, a Turing Machine:

- starts at leftmost non-blank character on the tape, in state 1

[10]It should be pointed out that a Turing machine is a "finite state" device, but has far more computational power than a simple FSM.

- halts in state 2 (put another way, state 2 is the halt state)
- cannot move left from start without crashing

An algorithm is a collection of instructions for a computing agent to follow.

If a Turing machine is a good model of a computing agent, then instructions for a Turing machine can be a model for an algorithm.

Recall Knuth's list of properties of an algorithm, which we first saw back in chapter 2:

1. Finiteness: "An algorithm must always terminate after a finite number of steps ... a very finite number, a reasonable number".

   Clearly our TM will terminate after a finite number of steps.

2. Definiteness: "Each step of an algorithm must be precisely defined; the actions to be carried out must be rigorously and unambiguously specified for each case".

   This is also obviously the case.

3. Input: "...quantities which are given to it initially before the algorithm begins. These inputs are taken from specified sets of objects".

   The initial tape configuration is the input.

4. Output: "...quantities which have a specified relation to the inputs".

   The final tape configuration contains the output.

5. Effectiveness: "... all of the operations to be performed in the algorithm must be sufficiently basic that they can in principle be done exactly and in a finite length of time by a man[11] using paper and pencil[12]".

   Moving the tape head right or left, (re)writing cells are all this basic: this was, indeed, Turing;s original concept, to make the model as simple as possible.

It is clear, then, that a Turing Machine can serve as a model for an algorithm

12.4.1 An Example Turing Machine

Assume we have the following instructions (_ stands for a blank cell, H stands for the halt state):

```
(1, _, 1, _, R)
(1, 1, 3, 0, R)
(1, 0, 3, 1, R)
(3, 0, 3, 1, R)
(3, 1, 3, 0, R)
(3, _, H, _, L)
```

[11] Knuth was writing in 1968, no doubt today he would write "person".

[12] Paper and pencil might not be the appropriate vehicle for some of the algorithms in daily life discussed below.

And the following tape (input) to begin with (b indicates a blank cell):

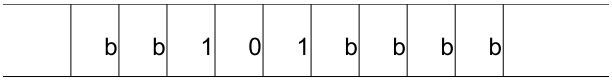


Figure 12.5: Input tape for Turing Machine No.1

This Turing Machine accepts any binary string as an input and produces the complement of the string as the output, when it halts.

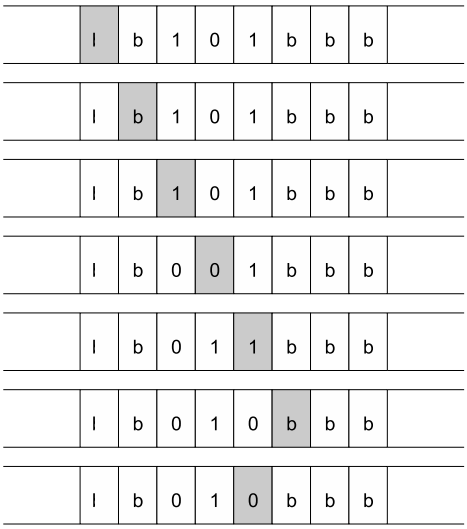


Figure 12.6: Progress of the inverting Turing Machine

Figure 12.6 shows the progress of execution of the machine. The shaded cells indicate where the tape head is situated.

12.4.1.1 The Turing Machine's power

In the hierarchy of formal models of computation mentioned above (see section 12.2.3.1) the Turing Machine is at the very top.

In fact, a Turing Machine is actually more powerful than a real computer in the sense that there is no limit to its memory size.

12.4.2 Finite State Machine Diagrams

We can construct FSMs as visual representations of Turing Machine algorithms[13].

[13]Only in a limited number of cases can we construct an equivalent FSM for a given Turing Machine. We can only if the Turing Machine does not rewrite characters on the tape and moves in only one direction.

The difference between these and the FSMs we saw earlier, is that the Turing Machine is capable of output and movement, so each transition arrow must also be labelled with the output value (new symbol written on the tape) and the direction (if any) of the tape head movement.

12.4.3 Unary Addition Machine

Unary arithmetic is arithmetic using a base of 1, i.e. there is only one numerical symbol – 1.

A number is represented in unary by writing *that number* of 1s in succession. Addition is achieved by concatenating two numbers:

```
111 + 1111 = 1111111
```

i.e.

```
3 + 4 = 7
```

The *alphabet* (i.e. he set of allowed symbols) for this machine is the set

```
{0, 1}
```

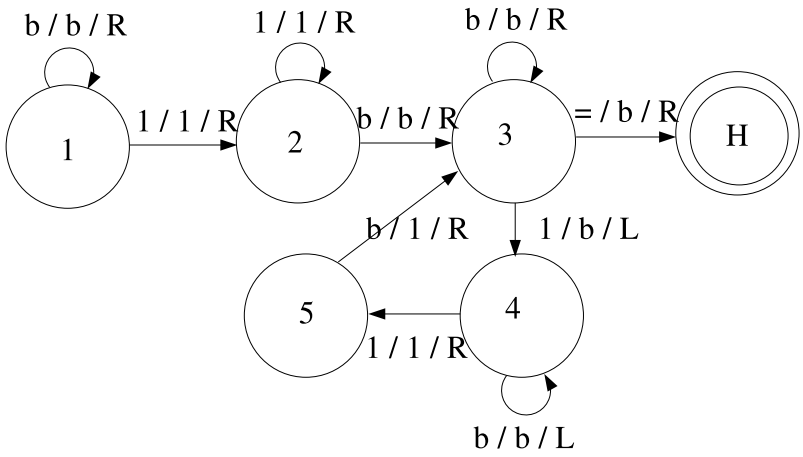


Figure 12.7: FSM for the unary addition TM

Figure 12.7 shows the state transition diagram for a unary addition machine.

Two numbers are represented as strings of 1s with (any number of) blanks in between and an = afterwards (this is to make the machine simpler, it is not necessary).

The basic strategy is simple:

```
while cell does not contain the '=':
   Move to the right until a blank is encountered   (state 1)
   Move to the write until a 1 is encountered (state 2)
   Replace the 1 with a blank (state 3)
   Move to the left until a 1 is encountered (state 4)
   Move one cell to the right and write a 1 (state 5)
```

12.4.4 Binary Incrementing Machine

We are now going to see a TM for incrementing (adding 1 to) a binary number – once we can add 1 we can add any number, simply by repeating the add 1.[14]

At first glance, even this might seem difficult: after all, we know that the TM can read and write symbols and move the tape head, but arithmetic?

Perhaps we can get an idea of how to proceed by examining a few actual binary increments:

0010110	0010111	1111110	0111111
+ 1	+ 1	+ 1	+ 1
- - - -	- - - -	- - - -	- - - -
0010110	0011000	1111111	1000000
(a)	(b)	(c)	(d)

and remember Alan Turing's oft-quoted remark that computers do not *do* arithmetic, they *simulate* arithmetic.

So we are going to build our TM based on the *patterns* of 0s and 1s and totally ignore any numerical meaning they might have.

A binary number either:

- Ends with a 0 or
- Ends with a 1 (to be precise a series of 1s)

There are no other possibilities.

Now, if the number ends with a zero – as in examples (a) and (c) above, then all our TM has to do is replace the final zero with a one and we're finished.

If the number ends with a one – examples (b) and (d) – things aren't actually that bad. Examine the examples and you will realise that what the TM needs to do is:

- Replace the final 0 with a 1
- Move left, replacing 1s by 0s until a 0 is found
- Replace the 0 with a 1

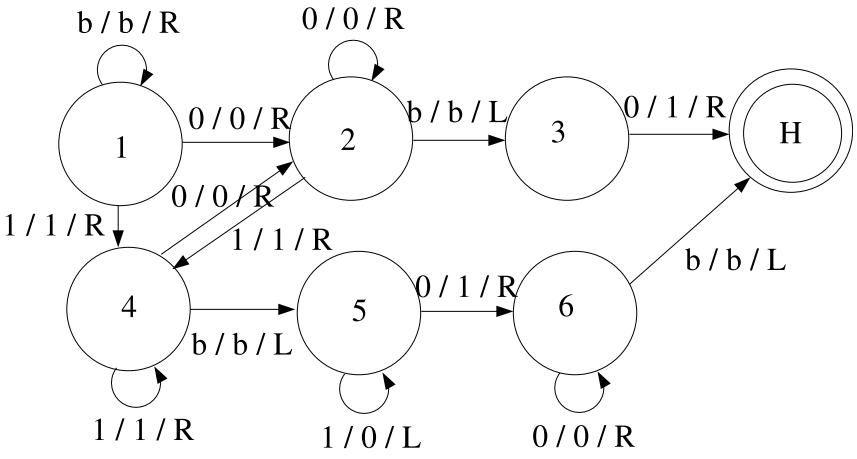


Figure 12.8: Binary incrementer FSM

Clearly the alphabet for this TM is the set of binary digits {0, 1}. The FSM is shown in figure 12.8.

Once again, it is worth remarking that there is absolutely nothing in the definition of the TM which refers to the concept of addition. We have completely defined binary incrementing in terms of the patterns of zeros and ones, with no reference to their (supposed) numerical values.

12.4.5 The Universal Turing Machine

Although individual Turing Machines are of great interest, Turing's real insight was to produce a definition of a *Universal Turing Machine*.

For this, he showed how a "normal" TM can be encoded onto the tape – along with its data – which is far from obvious, although not especially difficult. He then showed how the UTM can then run the program from the tape against the data, also on the tape.

As the tape is the TM's storage, we can see that the UTM has the *program and data stored together in memory*.

It is for this reason that Turing is credited with inventing the stored program computer.

The UTM is the theoretical equivalent of the modern electronic stored-program digital computer – the world's first *general-purpose* machine. Turing himself realised that the UTM was truly universal:

> The importance of the universal machine is clear. We do not need to have an infinity of different machines doing different jobs. A single one will suffice. The engineering problem of producing various

[14] Nobody ever claimed that TMs were *efficient*.

> machines for various jobs is replaced by the office work of "programming" the universal machine to do these jobs.

The UTM is a fascinating topic, but far too deep for this book.

For further reading see appendix XXX.

12.4.6 The Church-Turing Thesis

A Turing Machine does not always halt – which is the TM equivalent of an infinite loop in a "normal" program.

Essentially any input on the tape gives one of three situations:

1. The TM accepts the input (halts)
2. The TM rejects the input (stops in a non-halt state)
3. The Turing Machine does not stop, it loops

   The unary add machine we saw in section 12.4.3 will loop if we omit the = from the input

If there is an algorithm to do a symbol manipulation task, then there is a Turing Machine to do that same task.

This implies that Turing Machines define the limits of computability.

Thus, if we can find a symbol manipulation problem for which we can prove that no Turing Machine exists, then no algorithm exists, and thus the problem is uncomputable.

This concept is now known as the *Church-Turing Thesis*, after Alan Turing and Alonzo Church, a Princeton Logician who was also Turing's PhD supervisor.

Church had already formulated a theory of functions called the λ-calculus (lambda calculus); Turing's PhD thesis demonstrated the equivalence of TMs and the λ-calculus.

Turing's original thesis, as expressed by him, was that:

> LCMs[15] can do anything that could be described as "rule of thumb" or "purely mechanical".

Church himself acknowledged the utility of Turing's great invention:

> computability by a Turing machine ... has the advantage of making the identification with effectiveness in the ordinary (not explicitly defined) sense evident immediately.

The term *Church-Turing Thesis* seems to have been introduced in 1967 by Stephen Kleene – yes, he of the Kleene Star:

[15] Logical Computing Machines was Turing's expression for Turing Machines. He went on to add:

> This is sufficiently well established that it is now agreed amongst logicians that "calculable by means of an LCM" is the correct accurate rendering of such phrases.

"Such phrases" meaning phrases such as "rule of thumb".

> So Turing's and Church's theses are equivalent. We shall usually refer to them both as Church's thesis, or in connection with that one of its...versions which deals with 'Turing machines' as the Church-Turing thesis.

12.4.7 The "Halting" Problem

One of the great goals of computer science is "automatic program proving": the ability to prove a program correct *without actually running it.*

As a preliminary to this, it would be extremely useful if we could write a general program which would tell us whether another program running on specific data, will stop or loop forever.

This is called the *Halting Problem* and can be formulated in terms of Turing Machines.

A solution to the Halting Problem could, given any collection of Turing Machine instructions together with any initial tape contents, tell us whether that Turing Machine will ever halt if started on that tape.

Alas, it turns out that this is an uncomputable problem: no Turing Machine exists to solve this problem.

First a few simple examples to show what we mean.

12.4.7.1 Examples:

Our first example is very simple, a two-line loop:

```
while x != 1:
   x = x - 2
```

The legal input (original value of x) for this algorithm is the set of positive integers (1,2,...).

It is fairly trivial to show that the algorithm will halt for *odd* inputs, and will run forever for *even* numbers

Our second example does not *look* that much more complicated:

```
while x != 1:
   if x % 2 == 0:
      x = x/2
   else:
      x = 3*x+1
```

This algorithm has been tested on many positive integers and has always terminated.

However, nobody has been able to *prove* that it will always terminate.

Try it with x = 7. The sequence of x is:

```
7,22,11,34,17,52,26,13,40,20,10,5,16,8,4,2,1
```

At which point the algorithm halts.

12.4.7.2 The Insolubility of the Halting Problem

The full proof that the Halting Problem cannot be solved is rather more complex that we have space for, however here is the outline of the proof.

This is an example of what mathematicians call *Proof by contradiction*: we make an assumption and then show that the assumption leads inevitably to a contradiction. The original assumption, therefore, must have been false.

In this case our assumption is that the Halting Problem can be solved, i.e. that we can construct:

P, a Turing Machine that takes as input:

- the encoding, T^* – as per the UTM, we do not need the details here – of a Turing Machine T and an
- input tape t for that Turing Machine

The TM solves the halting problem; in particular, P always halts and its outputs are:

0	If T does not halt with input t
1	If T does halt with input t

We now use P to build another Turing Machine, Q, that also takes as input:

- the encoding T^* of a Turing Machine T and
- the input tape input t for that Turing Machine

Q halts only when P outputs 0.

The result of executing Q is:

Halts	If T does not halt with input t
Does not halt	If T does halt with input t

Finally, we use Q to build a Turing Machine, S, that:

- takes as input the encoding T^* of a Turing Machine T
- makes a copy of T^*
- feeds (T^*, T^*) to Q.

In other words, Q tells us what would happen if we were to run T with its own encoding as its input.[16]

The result of running S is:

[16] This is not quite such a strange idea as it might seem: many compilers are written in their "own" language – e.g. most, if not all, C compilers are written in C.

To compile the compiler we use the source code as input to the compiler (i.e. the executable binary) itself. Although we might not normally think of it that way, the source code is an encoding of the compiler algorithm, but not, of course, as a turing machine.

Halts	If T does not halt with input $T*$
Does not halt	If T does halt with input $T*$

You may want to go back and re-read this section before the final, mid-boggling step:

What happens when S is given $S*$ as input?

Well, from the definition of S, it will initially make a copy of $S*$ and then feed ($S*$, $S*$) to Q and Q will halt if, in this particularly instance, S (encoded as $S*$) does not halt on input $S*$ (i.e. its own encoding).

This means that S running on input $S*$:

Halts	If S does not halt with input S*
Does not halt	If S does halt with input $S*$

So we have a contradiction: S halts on $S*$ when S does not halt on $S*$ and vice-versa.

As every step followed logically from our initial assumption – that we could build the TM P – we must conclude that the assumption was false, therefore:

P cannot exist and therefore the Halting Problem is unsolvable.

12.4.7.3 Implications

The implications of the insolubility of the Halting Problem are legion.

If we can show that a particular problem is equivalent to the Halting Problem, then it, too, cannot be solved algorithmically.

Some examples of insoluble problems include:

- Given a Turing Machine T, determine whether it ever halts on any input.
- Given two Turing Machines, T1 and T2, determine whether they solve the same problem.
- Given a formal logical statement about algebra, determine whether it is a theorem or not.
- A variety of puzzles that involve tiling the plane with regular shapes.

As computer scientists we must realise that there are many things we cannot do.[17]

- No program in any language can be written to decide whether any given program always stops eventually no matter what the input.

[17] The 1984 cover writers of Time were evidently *not* computer scientists.

- No program can be written to decide whether any two programs are equivalent – i.e. that will produce the same output for all inputs. Here are two functions that we can clearly see are equivalent – and indeed prove it for this specific case – yet when the functions become much more complex we have no way of proving their equivalence

```
def double1 (x):
    return 2 * x

def double2 (x):
    return x + x
```

- No program can be written to decide whether any given program on any given input will ever produce some specific output. This case is equivalent to stating that it is impossible to write a general testing program.

In conclusion, it is worth remarking that is may very well be the generality of such tasks which is making life difficult, or impossible.

12.4.7.4 Research and Undecidable Problems

Finally a somewhat pessimistic end note.

When designing and analysing algorithms you should remember that problem you are trying to solve might:

- be equivalent to the Halting Problem – i.e. not solvable at all
- only have an *intractable* – i.e. exponential – algorithm. As we have seen these are only useful for very small data sets. To all intents and purposes intractable problems are effectively insoluble algorithmically.

However, with luck and a following wind, your problem will have a tractable algorithmic solution.

Now all you have to do is program it.

Part VII

Computers and Society

Chapter 13

Computers, Society and Ethics

> It should be noted that no ethically-trained software engineer would ever consent to write a *DestroyBaghdad* procedure. Basic professional ethics would instead require him to write a *DestroyCity* procedure, to which *Baghdad* could be given as a parameter.
>
> Nathaniel Borenstein

13.1 Computers and Society

It is clear that the arrival and continued improvement of the computer has had a massive effect upon society: initially this was limited to First World countries, but as computers have dropped in price and increased in capacity, so their presence has spread to virtually every corner of the globe.

There are perhaps three major areas of study:

1. The effect of computers on society in general. This includes such topics as the computerisation of the workplace, privacy and the surveillance society, and computer crime, to mention but three.

2. The ways in which the law deals (and fails to deal) with computer-related issues. This includes privacy legislation, copyright legislation and much more.

3. The way in which those directly involved with computers (programming, running, administering them) should behave with respect to their fellow citizens and society in general.

Entire books have been written on each of these areas[1]. In this section we are

[1] Two that I have on my shelves are entitled *Computer Ethics* and *Morality and Machines*.

going to concentrate on the last of the three topic areas – and, even here, we shall do little more than skim the surface.

The basic aim of this chapter is to make you think about your actions when using the computer and make you aware that there is an ethical dimension to the use of computer.

13.2 Computer Ethics

What are (is?) "ethics".

Let us begin by examing three related, possibly overlapping, but not identical terms: legal, moral, ethical.

We all know what is meant by "legal", although precisely what is and is not legal will change from jurisdiction to jurisdiction – not just between countries, but also inside countries.

Unfortunately, the dictionary definitions are not too helpful when it comes to the other two term.

The Concise Oxford Dictionary, for example, offers the following:

> moral: adj. 1. **a** concerned with goodness or badness of human character or behaviour, or with the distinction between right and wrong.

Which seems fair enough (in fact the definition has several more flavours, all amounting to the same thing).

But when it comes to "ethical", we find

> ethical: adj. relating to morals esp. as concerning human conduct.

Which seems to confuse the issue, if anything.

13.2.1 An Example

Perhaps if we take a concrete, non-computer-related example, we shall be able to better see the difference between the terms[2].

Imagine the following situation: two people, neither of them married, check into a hotel and spend the night together.

> Question: is their behaviour moral, legal and/or ethical?
>
> Answer: It depends[3].
>
> Let us look at each in turn.

[2] I am taking no particular stand on the issues discussed here.

[3] Don't you just love answers like this?

13.2.1.1 Morality

Even though we all believe that morality is (or should be) universal, the fact remains that each of us tends to have his or her own version of morality. What is moral for one person may well be immoral for another.

Some people's morality is partially or completely received: from their religious beliefs, for example.

Other people have their own personal morality, derived from wherever they feel appropriate.

Oddly – or perhaps not – everybody believes that their morality should be universal, i.e. should apply to everybody. When I say, for example, that murder is immoral, I do not simply mean that it would be immoral for *me* to commit murder.

As to our example: again, the question of whether the behaviour of these two people is moral or not will depend entirely on who is being asked.

13.2.1.2 Legality

Legality and morality are not the same thing: as has often been observed, you cannot legislate morality. Which is, presumably, why lying is not a crime.

Moreover, laws change and what was legal last year may be illegal this year and vice-versa.

In the case of our example, the further question which must be answered is: where did this assignation take place.

In some countries, such liaisons, even between consenting, single adults, are illegal[4].

In some countries the *gender* of the people in question is relevant: if they are of the same gender then the act is illegal.

In certain of the United States the actual details of what transpired between the individuals is relevant to its legality. Some actions are illegal in some states[5].

13.2.1.3 Ethicality

The law typically does not refer to *ethics*. How then do they enter into this particular example?

Consider the following refinement: one of the two people involved is a doctor or psychiatrist and the other is his or her patient.

Although this does not affect the legality of the situation, most codes of medical *ethics* – regardless of jurisdiction – consider sex between doctor and patient seriously unethical behaviour, which can even result in the offending physican's being struck off.

We can see, then, that ethics often apply to situations which the law does not.

---

[4] These countries tend to be the ones where religion plays a significant role in government. Typically, adultery – i.e. liaisons where one or both partners is married to somebody else – is also illegal in these countries.

[5] And if you want details, you're going to have to hunt them out for yourself.

As for morality, it is often the case that ethics are a form of legislated morality, legislated by a particular group of people (e.g. the Canadian Medical Association) and intended to apply to a limited subset of society in general.

Many professional organisations adopt Codes of Ethics which their members must agree to adhere to at the risk of censure (or possible expulsion, as in the case of the philandering physician).

13.2.2 The Ten Commandments of Computer Ethics

The notion of Computer Ethics – i.e. a set of rules covering the behaviour of people who work with computers – was first touted in the 1960s.

In 1992, Ramon C. Barquin of the Computer Ethics Institute ("a nonprofit research, education, and public policy organization focused on the issues, dilemmas, and challenges of advancing information technology within ethical frameworks", founded in 1985 and based at the Brookings Institution in Washington, DC) offered the following "Ten Commandments"[6].

1. Thou shalt not use a computer to harm other people.
2. Thou shalt not interfere with other people's computer work.
3. Thou shalt not snoop around in other people's computer files.
4. Thou shalt not use a computer to steal.
5. Thou shalt not use a computer to bear false witness.
6. Thou shalt not copy or use proprietary software for which you have not paid.
7. Thou shalt not use other people's computer resources without authorization or proper compensation.
8. Thou shalt not appropriate other people's intellectual output.
9. Thou shalt think about the social consequences of the program you are writing or the system you are designing.
10. Thou shalt always use a computer in ways that ensure consideration and respect for your fellow humans.

Although not without their critics – and we shall look at some of these criticisms – these "commandments", which have been used by, *inter alia*, the International Information Systems Security Certification Consortium, as a basis for their own codes of ethics, provide as useful starting point for discussion of the whole topic.

Let us, then, take a look at each of the "commandments".

[6] They are clearly inspired, certainly in terms of language, by the Biblical commandments, particularly as translated in the "King James" version of the bible.

And, for those who care about these things, like the original ten, these commandments have an 8-2 split between proscriptive ("thou shalt not") and prescriptive ("thou shalt").

In addition to looking at examples of behaviour that violates each commandment, we shall look at criticisms of some of the commandments from Ben Fairweather of the UK-based Centre for Computing and Social Responsibility. Fairweather consider the commandments to be "*possibly* a useful starting point for computer ethics". These criticisms will be referred to by the Centre's abbreviation[7], CCSR.

We shall also consider the criticisms of the "hacking" community[8], specifically those addressed by the hacking group Xanatomy, in response to a course based upon them and given by Dr. Ahmet Celal Cem Say of the University of Bogazici in Turkey. These criticisms will be referred to by the term "Hacker" and come,

[7] No, it's *not* an acronym. An acronym is a *pronounceable word* made up of an abbreviation. I defy anyone to come up with a reasonable pronunciation of ""CCSR".

[8] Here I am using the term "hacker" in its more recent flavour: a "hacker" being one who breaks into computer systems without authority. Originally the term referred to one who was particularly good at "hacking" out code; these "hackers" refer to the break-in artists as "crackers" – i.e. people who "crack" systems, not people who are insane.

according to Xanatomy, from the hackers' "constitution"[9], although a Google search does not provide any obvious source for this claim.

Note that many of the criticisms are in the form of specific exceptions; this is rather like the specific exceptions that have been issued to Christian soldiers in wartime, who clearly would undermine their side's fighting if they insisted on obeying the "Thou shalt not kill" commandment.

This points towards another difficulty: are there – can there be – blanket, universal rules that apply to *every* society in *every* situation? Or must we always take the specific society, context or situation into account.

Here we see the difference between *absolutist*, *relativist* and *situationist*

[9] The "Hackers' Constitution":

1. We believe: That every individual should have the right to free speech in cyber space.
2. We believe: That every individual should be free of worry when pertaining to oppressive governments that control cyber space.
3. We believe: That democracy should exist in cyber space to set a clear example as to how a functioning element of society can prosper with equal rights and free speech to all.
4. We believe: That hacking is a tool that should and is used to test the integrity of networks that hold and safe guard our valuable information.
5. We believe: Those sovereign countries in the world community that do not respect democracy should be punished.
6. We believe: That art, music, politics, and crucial social elements of all world societies can be achieved on the computer and in cyber space.
7. We believe: That hacking, cracking, and phreaking are instruments that can achieve three crucial goals:
   (a) Direct Democracy in cyber space.
   (b) The belief that information should be free to all.
   (c) The idea that one can test and know the dangers and exploits of systems that store the individual's information.
8. We believe: That cyber space should be a governing body in the world community, where people of all nations and cultures can express their ideas and beliefs has to how our world politics should be played.
9. We believe: That there should be no governing social or political class or party in cyber space.
10. We believe: That the current status of the internet is a clear example as to how many races, cultures, and peoples can communicate freely and without friction or conflict.
11. We believe: In free enterprise and friction free capitalism.
12. We believe: In the open source movement fully, as no government should adopt commercial or priced software for it shows that a government may be biased to something that does not prompt the general welfare of the technology market and slows or stops the innovation of other smaller companies' products.
13. We believe: That technology can be wielded for the better placement of mankind and the environment we live in.
14. We believe: That all sovereign countries in the world community should respect these principles and ideas released in this constitution.

morals or ethics:

- Absolutist morals insist that a moral rule must be obeyed regardless of any other information.

  According to this stance an anti-abortionist[10] would maintain that abortion is wrong under any circumstance (including rape, incest and pregnancies which endanger the life of the mother).

- Relativist morals say that each society and each era have their own morals.

  For example, relativists would point out that some societies (again, typically those where religion still has considerable influence) consider abortion immoral, but others do not.

- Situationist morals would examine each case individually.

  Frequently, even in jurisdictions where abortion is not normally legal, exceptions are made for extreme circumstances – rape, incest, etc. – but often there is a time limit attached (e.g. first trimester).

13.2.2.1 Thou shalt not use a computer to harm other people.

Examples This does not, of course, refer to hitting somebody over the head with your keyboard, or strangling them with your mouse cable no matter how tempting.

How can people be harmed by computer?

A few ways:

- Money can be transferred from their bank account
- Incorrect data could be inserted into a hospital's patient system, resulting in the wrong medial procedure being carried out
- Incorrect data could be inserted into a police database, resulting in the person's being arrested and possibly facing criminal proceedings

(Actually, many of the remaining commandments involve harm to others of a more or less specific nature.)

Perhaps the most obvious and currently "fashionable" example of harm done to others by computer, is *identity theft*. By stealing another person's identification information the thief can get credit and much more besides. Identity theft is an increasingly common problem in all heavily-computerised societies[11].

[10] The author is taking no stance on what is undoubtedly a particularly controversial topic, merely using it as illustration.

[11] You can even purchase identity theft insurance in the USA.

Discussion

CCSR Is it just people that we should not harm? What about the environment and animals (the environment is clearly harmed by the production and use of computers, and by the disposal of waste computers)?

These are clearly valid points.

Hacker Happily, the hacker community sees no particular problem with this commandment. However, they also proclaim that "We believe: In free enterprise and friction free capitalism" and there are those who would argue that capitalism (whether "friction free", whatever that means, or not) involves the exploitation of people and therefore harm. This is a political, not ethical argument.

13.2.2.2 Thou shalt not interfere with other people's computer work.

Examples The clearest example of this is the computer virus (and the computer worm)[12]. That the interference is more than simply an annoyance is attested by a recent report from the City of Manchester in the UK.

In February, 2009, the city's computer systems became infected with the "Conficker" worm. The initial and obvious cost resulted from the 1,609 traffic tickets which could not be issued.

More seriously, it is estimated that the infection will cost the city a total of some £1.5 million, including

> £600,000 on external IT consultants, including Microsoft staff, £169,000 on staff to process a backlog of benefits claims and council tax bills, and compensation payments to families awaiting benefits.

You might like to reflect upon the profit made by Microsoft here: arguably it was their defective operating system security which allowed the worm in in the first place. (The worm, incidentally, was believed to have been brought into work by an employee on a "thumb drive"; the city has now forbidden their use and disabled all USB ports on desktop computers.)

Discussion

Hacker "Hacking is a tool that should and is used to test the integrity of networks that hold and safe guard our valuable information" also "hacking, cracking, and phreaking are instruments that can achieve three crucial goals:

1. Direct Democracy in cyber space.
2. The belief that information should be free to all.

[12] Essentially, the difference is that a virus does harm to its host, whereas the worm's only point is to reproduce, harm only being done as a side effect, rather than as an intent.

3. The idea that one can test and know the dangers and exploits of systems that store the individual's information."

We might ask: but what does the hacker community do when they do discover a network lacking "integrity"? Should information about, for example, benefit recipients (or medical information) truly be "free to all"? And what is meant by "Direct Democracy" in this context? Breaking into, infecting or otherwise subverting a computer is hardly a "democratic" action, if by democratic we assume the usual definition, involving the will of the majority. "Hacking, cracking and phreaking"[13] are usually the actions of individuals, or small groups.

You might also like to consider the contradiction inherent in the two hacker statements above ("safe guard our valuable information" and "information should be free to all").

13.2.2.3 Thou shalt not snoop around in other people's computer files.

Examples It is generally considered at the very least immoral – if not, as in many jurisdictions, actually illegal – to open and read another person's mail. Why then should it be acceptable to read another's email? Or any other information resident on their computer system?

CCSR What if the "other people" are using the computer to do harm? Should we still refrain from interfering? Should computer files be private even if they are being used as part of a criminal conspiracy?

Again, excellent points. It is probably worth bearing in mind that the world's first electronic, digital computer, Colossus, was designed and built *precisely* to read somebody else's electronic messages.

In this case, as the "other people" were the genocidal Third Reich, most of us would probably consider this an ethical application of computer technology.

13.2.2.4 Thou shalt not use a computer to steal.

Examples This may, at first sight, appear to be what is often referred to as a "no brainer". Stealing is illegal and generally considered illegal, so this does not appear to be much of a "stretch".

However, the fact of digitised information has undermined the concept of "theft" in a novel and interesting way.

Consider: I buy the latest Britney Spears CD, download a song from iTunes[14], or rent a movie from Blockbuster. The using my computer, I make a copy for a friend.

Is this theft?

[13] The term "phone phreaking" signifies obtaining telephone services without paying for them. Clearly the "hacker" community is similiarly confused about the "hacker/cracker" dichotomy.

[14] I have *never* done nor will I ever do either of these things, they are only an examples.

This is a topic which can be debated endlessly (and has, in my classes, on more than one occasion).

On the "pro" side, our friend has clearly acquired something of (presumable) value without paying for it.

On the "con" side, the company which produced the original medium has not been deprived of anything (except the potential of a future sale, they will argue).

Discussion

CCSR What if stealing is the only way to prevent someone from doing a much greater harm?

13.2.2.5 Thou shalt not use a computer to bear false witness.

Examples How can a computer "bear false witness" (i.e. lie)?

Consider that data held solely in a computer is easier to change without leaving a trace than virtually any other format. The maintaining of two sets of books – one for the taxman and one containing the truth – has never been easier.

Another example: the old adage "the camera never lies" is clearly out of date. Even without resorting to a digital camera, scanned photographs can easily be manipulated by software. The same can be done (with a little more computer power) to moving images.

Given that law enforcement bodies depend increasingly on computerised data (cell phone records, CCTV camera footage – especially in the UK – and the like) there are more and more opportunities to use a computer to "bear false witness".

Discussion

CCSR What if stealing or bearing false witness is the only way to prevent someone from doing a much greater harm?

13.2.2.6 Thou shalt not copy or use proprietary software for which you have not paid.

Examples These are almost too numerous even to need mention. When I ask the average class who amongst them has never, ever used a piece of proprietary software which has not been paid for, there is usually not a single hand raised.

Why should this be? Particularly when many, if not most, of these people would never contemplate stealing money, for example.

For starters, obviously the chances of getting caught[15] are considerably less; and many people can convince themselves that this is a "victimless crime", insofar as nobody who owned the software before has been deprived of it now.

[15] The 11th Commandment: Thou shalt not get caught.

It also seems likely that, at least subconsciously, people feel that the cost of software is too high.

Discussion

CCSR This is too simplistic. Many of us use software on University or business computer systems where somebody else has paid for us to use the software. Beyond this, though, what if the software house that produced the software has used immoral methods to gain an excessively large share of the software market, which thus prevents competition, and enables it to over-charge for software? Under these circumstances is it wrong to use or copy software without paying the software house?

Interestingly, despite the fact that "pirated" software is so commonly used, the hackers' constitution has nothing to say on this subject.

13.2.2.7 Thou shalt not use other people's computer resources without authorization or proper compensation.

Examples Given the global convergence of various forms of communication, we could today extend this to include the telephone and television systems.

One of the most famous cases of unauthorised use of computer resources was that uncovered by Clifford Stoll in the 1980s and documented in his book *The Cuckoo's Egg*. A group of hackers in Germany was being paid by the Eastern Bloc to break into computers in the US and steal classified information.

The relevance of the case to this commandment is that the hackers used many computers not for the information stored directly on them, but as a means to access other computers.

CCSR What if it is an emergency, and the only way to stop a great harm is to use computer resources without authorization?

Clearly the hackers disagree with this.

In fact, "hacking" (in this sense) is precisely using "other other people's computer resources without authorization or proper compensation" and would seem to be the hacker's entire *raison d'etre*.

13.2.2.8 Thou shalt not appropriate other people's intellectual output.

Examples Intellectual Property (IP) rights have become a hot topic in the early 21st century.

Discussion

CCSR Even here, it is possible that somebody has a brilliant idea that can produce great social benefit, but which will not be taken seriously if the true author is known. By appropriating their intellectual output, society as a whole will gain substantially.

This seems like a bit of a stretch. (Perhaps he is thinking of the girl in the cafe in Rickmansworth, in Douglas Adams's *Hitchhiker's Guide to the Galaxy*, who discovers the secret to world peace, freedom from poverty, etc. only to die seconds later, along with the rest of the human race[16] when the Vogon Destructor Fleet destroys the Earth.)

More to the point, the (according to many people) broken US patent system allows companies to take out patents on ideas (the original intent of the patent system was to protect *inventions*) which are extremely broad (one company in the UK attempted to patent the idea of clicking on a link) and, perhaps, extremely obvious (the recent case involving Microsoft Word and the storing of a document and its attributes in XML).

13.2.2.9 Thou shalt think about the social consequences of the program you are writing or the system you are designing.

Examples This would seem, at first sight, to be entirely obvious. And yet there are many programs in use today whose entire purpose is, in some people's eyes, malicious – the telemetry control software for nuclear missiles, for instance, or bulk spam mailing software – and these systems were written by *somebody*.

Discussion

CCSR Thought, unaccompanied by action, is pointless. They must act upon those thoughts. Further, it is not just in writing of software that thought of social consequences and action should follow: although both are necessary in the writing of software.

Interestingly, the hackers seem to have nothing to say about this topic.

13.2.2.10 Thou shalt always use a computer in ways that ensure consideration and respect for your fellow humans.

Examples One only has to observe certain Usenet news groups, or internet mailing lists, or political blogs to see this commandment not so much broken as smashed to smithereens.

Is there a free speech issue here?

Discussion

CCSR There may be situations in the world where more good can be done by not showing respect for all, and the possibility of doing such good should not be dismissed out of hand.

Should we all, for example, have been considerate and respectful towards Saddam Hussein?

[16] Yes, yes, except for Arthur Dent and, as it later transpires, Tricia McMillan.

Part VIII

Appendices

Appendix A

Annotated Example Python Programs

The example programs in this appendix are simply for illustration. Many of them are also discussed in the main text.

A.1 Simple Algorithms

Most of these programs are simple enough to need no explanation other than the comments. Each program is followed by a short illustration of running it.

A.1.1 Averaging Three Numbers – sequence

This program asks the user for three numbers and prints the average (integer version).

```
N1, N2, N3 = input ()                # get numbers from user
average = (N1+N2+N3)/3               # calculate average
print average                        # and display it

$ python al1.py
5,7,7
6
```

And here is the non-integer version:

```
N1, N2, N3 = input ()                # get numbers from user
average = (N1+N2+N3)/3.0             # calculate average as float
print average                        # and display it
```

```
python al1a.py
5,7,7
6.33333333333
```

A.1.2 Testing Values – if

This program prompts the user for a number and tells him/her if that number is zero (I know).

```
A = input ('Enter a number ')
if A == 0:
   print "That number is zero!"
else:
   print "That number is not zero!"

python al2.py
Enter a number
0
That number is zero!

python al2.py
Enter a number
14
That number is not zero!
```

A.1.3 Averaging with validation – if

The following program is a rather simple-minded Grade Point Average calculator. Given three grades, which must be between one (1) and nine (9), print the GPA.

```
G1, G2, G3 = input ('Enter three grade point values ')
if G1 < 1 or G1 > 9 or G2 < 1 or G2 > 9 or G3 < 1 or G3 > 9:
   print "Bad data"
else:
   average = (G1+G2+G3)/3.0
   print average

python GPA.py
5, 9.5, 7
Bad data

python GPA.py
8, 7, 8
7.66666666667
```

Note the complexity of the if condition; the reason is simple, we have three input values and each could be wrong in either of two ways (too small or too large), so we have six subconditions to test, although the above examples just show a single error.

A.1.4 Printing squares – while

This program prompts for a number and then prints numbers and their squares from the entered number up to and including 10:

```
count = input ('Enter count')
while count <= 10:            # Loop until > 10
   square = (count * count)   # calculate square
   print count, square        # print number and its square
   count = count + 1          # next number

$ python while1.py
Enter count 6
6 36
7 49
8 64
9 81
10 100
```

A.1.5 Adding a series of numbers – while

This program prompts the user to enter a series of numbers

```
number = input ('Enter numbers - negative value to quit ')
                           # get first number
total = 0                  # total so far

while number >= 0:         # until negative value entered
   total = total + number  # add this to running total
   number = input ('Enter next number, < 0 to quit ')
                           # get next number

print "total", total       # display results

$ python while2.py
Enter numbers - negative value to quit 5
Enter next number, < 0 to quit 7
Enter next number, < 0 to quit 33
Enter next number, < 0 to quit 14
Enter next number, < 0 to quit 99
```

```
Enter next number, < 0 to quit -1
total 158
```

A.1.6 Averaging a series of numbers – while

We can easily extend the previous program to count the numbers as they are entered and then average them at the end.

Note that we now use zero as our terminator – although zeros can come into average calculations, this is not an entirely unreasonable approach.

Note also the use of the python builtin *float* function for converting the count into a floating point value before doing the average division – essential unless we are going to get all-integer, truncated results.

```
number = input ('Enter numbers - zero to quit ')
                            # get first number
total = 0                   # total so far
count = 0                   # count of numbers entered
while number != 0:          # until zero entered
   count += 1               # count this non-zero number
   total = total + number   # add to running total
   number = input ('Enter next number, 0 to quit ')
                            # get next number
print "total", total        # display results
print "count", count
print "average", total/float(count)
```

```
$ python while3.py
Enter numbers - zero to quit
2
Enter next number, 0 to quit
7
Enter next number, 0 to quit
-5
Enter next number, 0 to quit
99
Enter next number, 0 to quit
-6
Enter next number, 0 to quit
3
Enter next number, 0 to quit
0
total 100
count 6
average 16.6666666667
```

A.2 More Interesting Algorithms

A.2.1 Finding the largest number

Given a list of numbers (entered by the user) we need to find the largest number in the list and the position where it (first) occurs.

```
A = input ('Enter a list of numbers ') # the list to process
largest = A[0]                         # first is largest so far
location = 0                           # and here it is
i = 1                                  # start checking from second number
while i < len (A):                     # as long as there are more to check
   if A[i] > largest:                  # this is a new largest
      largest = A[i]                   # remember it
      location = i                     # and where we found it
   i = i + 1                           # check next item in list

print "largest, location", largest, location # print results
```

```
$ python largest.py
Enter a list of numbers
[5, 7 ,4, 3, 2, 9, 8, 12, 9, -5, 14, 2, 14, 7, 6]
largest, location 14 10
```

Notice that the largest value – 14 – occurs twice in the list, at position 10 (as reported) and at position 12.

As an exercise, modify the program to find the position of the *last* occurrence of the number rather than the first.

A.2.2 The Data Cleaning Problem

A.2.2.1 Shuffle Left

```
#
# The shuffle-left version of the data cleanup algorithm
#
L = input ('Enter list to cleanup ') # read in list
position = 0                         # start at beginning
size = len (L)                       # assume all list is non-zero
while position < size:               # until we've done the whole list
   if L[position] == 0:              # found a zero
      pos = position                 # remember where
      while pos < size-1:            # shuffle rest of list
         L[pos] = L[pos+1]           # shuffle one entry left
```

```
         pos = pos + 1               # move along (sub)list
      size = size - 1                # one fewer non-zero entries
   position = position + 1           # move along list
print 'cleaned up list', L[:size]    # print non-zero entries

$ python cleanup1-1.py
Enter list to cleanup
[0, 12, 0, 99, 5, 0, 66, 13, 0]
cleaned up list [12, 99, 5, 66, 13]
```

And here is the version with the shuffle hived off into a separate function:

```
#
# The shuffle-left version of the data cleanup algorithm
#

#
# Shuffle sub-list left given the list and the position to
# begin the shuffle.
#

def shuffle (L, pos):    # shuffle given list 1 position left
   while pos < len(L)-1: # until the end of the list
      L[pos] = L[pos+1] # move 1 entry left
      pos = pos + 1 # go to next entry

#
# Main program
#

L = input ('Enter list to clean up ') # read in list
position = 0 # start at beginning
size = len (L) # current count of valid entries
while position < size:                # as long as there is more to do
   if L[position] == 0:               # found a zero
      shuffle (L, position)          # get rid of it
      size = size - 1                # list has 1 fewer valid entry
   position = position + 1           # move along list 1 position

print L[:size]                        # print non-zero entries

$ python cleanup1-3.py
Enter list to clean up
[0, 12, 0, 99, 5, 0, 66, 13, 0]
[12, 99, 5, 66, 13]
```

A.2.2.2 Copy Over

Here is the copy over version of the cleanup algorithm.

Note that, for simplicity, we allocate a new list (all zeros initially) the *same size* as the old list, then count the entries as we insert them.

```
#
# The copy-over version of the data cleanup algorithm
#
L1 = input ('Enter list to cleanup ')
L2 = [0] * len (L1)         # new list same size
inpos = 0                   # start copying from zero
outpos = 0                  # start copying to zero
count = 0                   # new list size so far

while inpos < len (L1):      # process entire list
   if L1[inpos] != 0:        # must copy this item
      L2[outpos] = L1[inpos] # copy it
      count = count + 1      # and count it
      outpos = outpos + 1    # move along new list
   inpos = inpos + 1            # move along old list

print L2[:count]             # display result
```

```
$ python cleanup2.py
Enter list to cleanup
[0, 12, 0, 99, 5, 0, 66, 13, 0]
[12, 99, 5, 66, 13]
```

A.2.2.3 Converging Pointers

Note that this is the final, bugs-removed, version discussed in chapter 4.

```
L = input ('Enter list to cleanup ') # get list
left = 0                             # start left at beginning
right = len (L) - 1                  # and right at end
count = 0                            # nothing done yet
while left <= right:                 # until pointers collide
   if L[left] == 0:                  # a zero to zap
      if L[right] != 0:              # and we're not zapping it with another
         L[left] = L[right]          # move right number to left position
         count = count + 1           # count this entry
         left = left + 1             # move left pointer
      right right = right - 1        # move right pointer left
   else:                             # not looking at zero
      count = count + 1              # count non-zero entry
      left = left + 1                # move right along list
```

```
print L[:count]                      # print result

python cleanup3.py
Enter list to cleanup
[0, 12, 0, 99, 5, 0, 66, 13, 0]
[13, 12, 66, 99, 5]
```

A.3 Searching Algorithms

A.3.1 Linear Search – the address book

Here is the program we first saw in chapter 3. A realistic implementation would allow the address book data to be entered – either online or, more likely, read from a file.

```
N = ['bill', 'amanda', 'arthur', 'jane', 'ron', 'kathy', 'tom', 'gillian']
A = ['1223 Any Street', '666 Revelation Avenue', '932 Main Street',
     '3472 Baseline', '17 Government Street',
     '4432 Douglas Street', '4461 Interurban Road',
     '3327 Old Island Highway']

name = raw_input ('Enter name to look for ')
                          # name to search for
found = 'No'              # not found it yet
i = 0                     # start from beginning of book
while found == 'No' and i < len (N):
                          # loop until we find name or run out of book
   if N[i] == name:       # this is the one!
      print A[i]          # print address
      found = 'Yes'       # remember we found it
   else:                  # not this one
      i = i + 1           # look at next entry

if found == 'No':         # name not in book
   print 'Name not found' # tell user
```

A.3.2 Binary Search

In this example we are merely searching through a list (not a python list) of names and either reporting on the position where we found the desired name or reporting that it is not in the book.

This example *relies* on the user's entering the names in order (i.e. sorted).

This is *not* a good idea.

```
#
# Binary search program
#

#
# Search function
# Given book (a sorted python list) and
# a name to search for
# uses a binary search and either returns
# the position where the name was found (a number) or
# returns the python value None to indicate not found

def search (book, name):
   left = 0                    # interval spans
   right = len (book) - 1      # entire list
   found = False               # not found yet
   while left <= right and not found:
                               # either run out of list or find it
      mid = (left + right) / 2 # calculate mid point
      if name == book[mid]:    # we lucked out - it's there
         found = True          # remember it's found
         position = mid        # here
      elif name < book[mid]:   # go left young man
         right = mid - 1       # new (left) interval
      else:                    # go right
         left = mid + 1        # new (right) interval

   if found:                   # it was there - tell caller
      return position          # where it was
   else:                       # not found
      return None              # tell caller

if __name__ == '__main__':     # command line
   Book= input ('Enter names: ')
                               # Get names
   Name = raw_input ('Enter name to search for, enter to quit: ')
                               # Get first search name
   while Name != '':           # Until user hits just enter
      position = search (Book, Name)
                               # search for name
      if position == None:     # not there
         print Name, 'not found'
      else:                    # it's there
         print Name, 'found at', position
      Name = raw_input ('Enter next name, enter to quit: ')
                               # get another name
```

Before seeing this code in action, a quick word about one thing that may be puzzling you, the line:

```
if __name__ == '__main__':
```

which seems to be followed by most of the program.

The idea here is so that the program before this line – which consists entirely of the definition of the search function – could be useful to another program. The function can be made available using the *import* statement, which we shall be using later.

The special name *__name__*, which is defined by python, will tell the code whether it is being imported (in which case *__name__* will contain the name of the module) or being run "at the top level" either by typing

```
python programname
```

or simply

```
programmname
```

at the shell prompt, or by dropping the program file icon onto the python icon or (double) clicking on the program file icon in a windowing system.

This feature is also useful for writing *self test code*, code which won't normally be used – because the module which contains it will normally be imported for use by other programs – but which is sufficient, when run in standalone fashion, to test the module code.

```
$ python binsearch.py
Enter names: ['aardvaark', 'arthur', 'beryl', 'deryk', 'dorothy', 'zak']
Enter name to search for, enter to quit: fred
fred not found
Enter next name, enter to quit: beryl
beryl found at 2
Enter next name, enter to quit: arthur
arthur found at 1
Enter next name, enter to quit: zak
zak found at 5
Enter next name, enter to quit: deryk
deryk found at 3
Enter next name, enter to quit: dorothy
dorothy found at 4
Enter next name, enter to quit: aardvaark
aardvaark found at 0
Enter next name, enter to quit:
```

What happens, though, if the user enters the names out of order?

```
$ python binsearch.py
Enter names:  ['aardvaark', 'arthur', 'beryl', 'deryk', 'dorothy', 'alan', 'bill', 'cha
Enter name to search for, enter to quit: frank
frank not found
Enter next name, enter to quit: fred
fred not found
Enter next name, enter to quit: zak
zak found at 8
Enter next name, enter to quit: beryl
beryl found at 2
Enter next name, enter to quit: alan
alan not found
Enter next name, enter to quit: charlie
charlie not found
Enter next name, enter to quit: ethelred
ethelred not found
Enter next name, enter to quit:
```

As you can see, some names can be found, others cannot.

Fortunately python (and several other languages) offer us an easy way to sort the list and make everything work.

Rather than repeat the entire program, we'll simply repeat the last part:

```
if __name__ == '__main__':     # command line
   Book= input ('Enter names: ')
                               # get names
   Book.sort ()                # make sure they're in order
   Name = raw_input ('Enter name to search for, enter to quit: ')
                               # Get first search name
   while Name != '':           # Until user hits just enter
      position = search (Book, Name)
                               # search for name
      if position == None:     # not there
         print Name, 'not found'
      else:                    # it's there
         print Name, 'found at', position
      Name = raw_input ('Enter next name, enter to quit: ')
                               # get another name
```

```
$ python binsearchsort.py
Enter names: ['aardvaark', 'arthur', 'beryl', 'deryk', 'dorothy', 'alan', 'bill' , 'cha
Enter next name, enter to quit: aardvaark
aardvaark found at 0
Enter next name, enter to quit: arthur
arthur found at 2
Enter next name, enter to quit: alan
```

```
alan found at 1
Enter next name, enter to quit: bill
bill found at 4
Enter next name, enter to quit: charlie
charlie found at 5
Enter next name, enter to quit: ethelred
ethelred found at 8
Enter next name, enter to quit: deryk
deryk found at 6
Enter next name, enter to quit: dorothy
dorothy found at 7
Enter next name, enter to quit: zak
zak found at 9
Enter next name, enter to quit: ethelred
ethelred found at 8
Enter next name, enter to quit: frank
frank not found
Enter next name, enter to quit: aaaaaa
aaaaaa not found
Enter next name, enter to quit: zzzzzz
zzzzzz not found
Enter next name, enter to quit:
```

Of course, since the list has been sorted by the program, the positions reported back by the search are not going to be of much interest – or use – to the user.

A.4 Sorting

Both of the sort algorithms presented here are – unfortunately – $\mathbf{O}(n^2)$. There are more efficient algorithms, the best of which – such as Hoare's *quicksort* – are $\mathbf{O}(n \ (\lg n))$, but these are too complex for us to go into them here.

A.4.1 Bubble Sort

The bubble sort works by repeatedly traversing the list, exchanging out-of-order pairs, until there are no more pairs to swap.

```
def sort_list (L):
    swaps = True;                       # force into loop 1st time
    while swaps:                        # until the list is in order
        swaps = False;                  # None so far
        for i in range (len (L) -1):
            if L[i] > L[i+1]:           # swap this pair
                swaps = True;           # remember we found something to swap
                L[i], L[i+1] = L[i+1], L[i]
```

```
L = input ('Enter list to sort:')

sort_list (L)

print  L
```

Note that the code uses that python builtin *range* function to generate a list of values which *for* then cycles through:

```
>>> L = range(5)
>>>
>>> print L
[0, 1, 2, 3, 4]
>>> for i in L:
...    print i
...
0
1
2
3
4
```

and the extremely useful parallel assignment feature, which most languages do *not* possess.

In many languages, if you want to swap the contents of two variables, you need a third one to hold the intermediate value:

```
temp = x
x = y
y = x
```

But not in python:

```
>>>
x = 5
>>> y = 6
>>> x, y = y, x
>>> print x
6
>>> print y
5
>>>
```

You can actually extend this to any number of items and the "right hand side values" don't simply have to be variable names (or references to list items, as in this algorithm):

```
x, y, z = y, z * 5, x - y + z
```

Note that it is the *original* values for *x*, *y*, and *z* which are used:

```
>>> x = 3
>>> y = 4
>>> z = 6
>>> x, y, z = y, z * 5, x - y + z
>>> print x, y, z
4 30 5
```

Note that in the output:

- 4 is the original *y* value
- 30 is 6 (*z* value) times 5 and
- 5 is 3 (*x*) - 4 (*y*) + 6 (*z*)

And, just in case you need convincing, here is the bubble sort in action:

```
$ python bubble.py
Enter list to sort:[5, -4, 99, 14, -52, 101, 3]
[-52, -4, 3, 5, 14, 99, 101]
```

A.4.2 Selection Sort

The selection sort uses the *find_largest* algorithm we first saw in chapter 3and which appears again in section A.2.1.

The algorithm partitions the list into an unsorted (front) and sorted (back) part. Every element in the sorted part is

- in its correct place and
- greater than any element in the unsorted part.

Initially the sorted part is empty and the position marker is set to the end of the list.

The algorithm then repeatedly exchanges the *last* element in the *unsorted* part for the *largest* element in the unsorted part. This effectively extends the sorted part of the list one element to the left and the position marker is set accordingly.

Eventually the position marker reaches the front of the list, which is now completely sorted.

```
def find_largest (L):              # find largest item in List
    largest = L[0]
    lpos = 0
    i = 0
    while i < len (L):                  # rest of list
```

```
            if L[i] > largest:
                largest = L[i]
                lpos = i
            i += 1
        return lpos

    def sort_list (L):                      # use selection sort on list L
        sorted = len (L)
        while sorted > 0:
            lpos = find_largest (L[:sorted])
            L[lpos], L[sorted-1] = L[sorted-1], L[lpos] # swap
            sorted -= 1

    L = input ('Enter list to sort:')

    sort_list (L)

    print  L
```

This algorithm uses the python *slice* operation (:) to pass only the unsorted part of the list to *find_largest*:

```
lpos = find_largest (L[:unsorted])
```

As *unsorted* marks the position of the beginning of the sorted list, *L[:unsorted]* is that part of *L up to but not including* the item at position *sorted* – i.e. the unsorted part.

The code also uses the abbreviated assignment + operation syntax:

```
sorted -= 1
```

which is simply shorthand (borrowed from the C family of languages) for

```
sorted = sorted - 1
```

And here is the selection sort, operating on the same data as the bubble sort:

```
python select-sort.py
Enter list to sort:[5, -4,  99, 14, -52, 101, 3]
[-52, -4, 3, 5, 14, 99, 101]
```

A.5 Some examples using python libraries

Realistically we shall want to write programs which do more than read numbers and/or text from the keyboard.

A.5.1 Processing the command line

Handling parameters from the command line is simple in python. Here is a program which simply spits back its command line arguments, numbered.

```
#!/usr/bin/python
#
# Print back command line arguments.
#
# Deryk Barker, August 24, 2007
#

import sys

i = 1
for arg in sys.argv[1:]:
    print 'arg', i, arg
    i += 1
```

The first line is important in the unix/linux world: when we simply invoke a script file by name, the unix shell program (equivalent of the DOS/Windows *command.com*) reads the first line of the program and if it begins with the special sequence #! then the program specified on that line is used to process the rest of the script file.

The module *sys* provides us with many useful features to interface to the system In this example we are interested in *sys.argv* a list of all the command line parameters.

Our program – *args.py* – simply echoes back each of the parameters prefixed with a number. Note that we do not print out parameter 0, as that is the name of the script file itself.

$ python args.py first second third last

```
arg 1 first
arg 2 second
arg 3 third
arg 4 last
```

If we have used the #! convention described above, then the same effect can be had by simply using the script name:

```
$ args.py first second third last
arg 1 first
arg 2 second
arg 3 third
arg 4 last
```

A.5.2 Files

Python makes it very easy to read from disk files, as in this simple example, written while writing this book[1]: in order to import python files cleanly into the text processor (*klyx*) it was useful to be able to display them doublespaced, as it were: with an extra blank between every line.

That is what the *nler* (NewLinER) program does.

Let's see the code[2], then discuss it.

```
#!/usr/bin/python
#
# Print files with an extra newline after each line
#
# Handy for importing stuff into lyx
#
# Deryk Barker, August 21, 2007
#

USAGE = 'Usage: nler file [file...]'
```

The usage message will be printed out when the user gets the command line parameters wrong. Defining it at top level makes it available throughout the program.

```
#
# Function to print lines of a single file, each with an extra NL appended
#

def nler (name):
    f = open (name, 'r')        # open the file for reading
    for line in f.readlines (): # cycle through all lines
        print line              # print line *and* trailing NL
```

The function which does the work. Given a file name it uses the standard python file interface to open the file for reading.

The *readlines* function, defined for every open file, reads the *entire* file, line by line, into a list. Each line still has the NL (newline) character, which is why the print will add an extra newline – *print line* prints the contents of the variable name (the next element of the *readlines* array) and then an extra newline.

[1] As an indication of the power and ease of use of python, it took about five minutes from beginning to enter the program code to having it work.

[2] You will no doubt have realised from the discussion above that the program was used to print out its own code:

```
python nler.py nler.py
```

Notice that the *for* statement uses the result of the *f.readlines()* function call on-the-fly: we do not need to assign the list and give it a name, although we could:

```
all_lines = f.readlines()
for line in all_lines:
```

Is exactly equivalent, although unnecessary.

Our top level code will check the validity of the command line – we need at least one file name to print, for example.

Assuming we have at least one filename, we simply loop though the command argument list (the slice of *sys,argv* from 1 onwards, i.e. omitting the command name) passing each file name to the *nler* function.

```
if __name__ == '__main__':      # running at top level
    import sys                  # need this module
    if len (sys.argv) < 2:      # not provided any file names, silly
        print USAGE             # this is how you should use me
        sys.exit (1)            # but not this time - bye
    for arg in sys.argv[1:]:    # cycle through all file names provided
        print arg, ':'          # print the name
        nler (arg)              # and the file
```

To see it in action, let's construct a small file:

```
$ cat txtfile
line 1
line 2
line three
line the last

$ python nler.py txtfile
txtfile:
line 1

line 2

line three

line the last
```

A.5.3 Interacting with the file system

Our final simple example was also written during the creation of this textbook, but not for it.

The problem is fairly simple: when download or ripping audio files, one frequently ends up with files numbered out of range (when merging two CDs together on the hard drive for example).

So, you might get a directory containing files with names such as:

- 01_CD1_track1.wav
- 02_CD1_track2.wav
- 01_CD2_track1.wav
- 02_CD2_track2.wav

etc.

In order for our media player to play the tracks in the right order we need to resequence them.

The idea of *reseq.py* (which took perhaps half an hour to get to work) is this: we give it an *increment* (positive or negative) and a series of file names. The program must then apply the increment to the numerical prefix of each file name.

In the case of the list of files above: assuming that CD1 goes up to track 10, then we need to add 10 to every prefix for CD2, like this:

```
reseq 10 *CD2*
```

Where **CD2** is the unix "wildcard" expression that matches every file name which contains *CD2*.

This initial version assumes that each file name has a two digit numerical prefix (or it quits).

Here is the program:

```
#!/usr/bin/python
#
# Resequence a collection of files.
# Current assumption is that each filename
# begins with 2 digits.
#
# Deryk Barker, August 23, 2007
#

USAGE = "Usage: reseq increment file [file...]"
```

Once again we have a function which does most of the work.

In this case it also does some validation: does every file name provided have a numeric prefix?

This is why we loop through the *filelist* list twice: the first time we simply try to convert the first two characters of each name to a number, if it fails then the entire program quits.

If none of them fail, then we loop through the list again, this time doing the change.

```
#
# Do the resequencing.
# filelist is already in the correct order, i.e.
#       ascending for a negative increment
#       descending for a positive increment
#
# Grab the first two characters of each filename and make numeric.
# Add the increment
# Replace the first two characters of each filename with the 2-character
# ASCII version of the number.
#
# Rename the file.
#

import os
def reseq (filelist, increment):
    for name in filelist:               # Check they have numeric prefixes
        try:
            prefix = int (name[:2])
        except:
            print "Non-numeric prefix for file %s" % name
            sys.exit (4)
    for name in filelist:               # OK, now can do the work
        prefix = int (name[:2]) + increment # We know this will not fail
        newname = ('%2.2d' % prefix) + name[2:]
        print "renaming %s to %s" % (name, newname)
        try:
            os.rename (name, newname)
        except:
            print "Cannot access " + name

#
# Check the arguments.
# Need at least 2 (plus program name)
# increment must be numeric and
# >= -98, <= 98 but non-zero.
#
# The 98s are because the largest and smallest valid values for CD track
# numbers are 99 and 1 respectively and that's what this program is
# basically for.
#

if __name__ == '__main__':
```

```
import sys
if len (sys.argv) < 3:              # not enough
    print USAGE                     # tell user how
    sys.exit (1)                    # and bail out
```

The first argument should be the increment which must be:

1. Numeric
2. In the range -98 <= increment <= 98
3. Non-zero (ask yourself what would be the point?)

```
try:
     increment = int (sys.argv[1])
 except:                            # well THAT'S no good
     print "Illegal increment: %s" % sys.argv[1]
     sys.exit (2)
 if increment == 0 or increment < -98 or increment > 98:
     print "Invalid increment value %d" % increment
     sys.exit (3)
```

Now we know the increment is valid, we must make sure that the list of file names is in order.

For a positive increment we are renaming files "upwards", so we need to start with the highest-numbered file and work backwards, to avoid number clashes.

Similarly, for a negative increment, we need to work forwards.

As we cannot – for technical reasons – directly manipulate the *sys.argv* array, we make a copy of it, delete the first two items (the program name and the increment) and then sort what is left – the filenames – using the python *sort* method defined for all lists.

If the increment is negative this ascending sequence is fine. If the increment is positive we use the python builtin *reverse.*

```
namelist = sys.argv[:]              # real copy
del namelist[:2]                     # remove program name and increment
namelist.sort ()                    # make sure at least ascending sequence
if increment > 0:                   # positive, need to work backwards, so
    namelist.reverse ()             # make it so
reseq (namelist, increment)         # now do the actual work
```

Finally, when the list is in order, we pass it across the the *`reseq` function.*

A.6 A Serious Example – The TASS Virtual Machine

This code is somewhat advanced, although in concept it is simple: it emulates the simple example CPU used in chapter 9 in our discussion of assembly programming.

This module actually implements the *fetch* the *decode* and the *execute*, but these are called (actually they are all in a single function) from another module, the debugger.

Here is the code, with explanations interspersed, rather than left until the end.[3]

Remember the most important features of the machine:

- 16-bit words
- 12-bit addresses – so a maximum of 2^{12}, i.e. 4096 words of memory
- 4-bit opcodes – so a total of 2^4, i.e. 16 instructions

The instructions are formatted as:

4-bit opcode	12-bit address field

```
#
# the tass virtual machine
#
# $Log: tsim.py,v $
# Revision 1.2  2003/10/29 04:42:43  dbarker
# Version: 0.91
# Change: Add symbol capability to file assembly
# Author: dgrb
#
#
```

The rather strange-looking comment beginning *$Log* is automatically inserted by the CVS revision control system.

A complete log of changes made to the code is an important aspect of the initial comments.

```
import texc
import tidy
```

These two modules are also part of the TASS system and will be referenced here.

```
INPROMPT = 'in: '
MAXOPCODE = 15
MAXADDRESS = 4095                       # 12-bit addresses
MAXOPERAND = 65535                      # 16-bit data
OPCODEMASK  = 0xf000
OPERANDMASK = 0x0fff
```

[3]This really is quite a complex example, if you don't understand it, don't worry. You might want to come back to it, though, if you ever take a course on Computer Architecture.

Some handy "constants" (python doesn't actually support non-updateable constants) to be used. The various *MAX* values are defined here so that we could easily "upgrade" the machine by, for example increasing the word size.

```
OPCODES = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
OPNAMES = ['load', 'store', 'clear', 'add', 'inc', 'sub', 'dec', 'cmp',
           'jump', 'jumpgt', 'jumpeq', 'jumplt', 'jumpneq', 'in', 'out',
           'halt']
```

These two lists go together[4]: the *OPCODES* list is simply the valid opcode values in order; the *OPNAMES* list contains the instruction names in *the same order*.

```
#
# Here is the actual machine itself:
#
MaxMem = 4096                           # Memory size
PC = 0                                  # Program Counter
Acc = 0                                 # Accumulator (register)
Mem = [0] * MaxMem                      # Main memory
```

Given our 12-bit addresses, the maximum memory is 4K. This is also where we define the physical attributes of the machine: the Program Counter and Accumulator registers and the memory, which is simply defined to be a list of zeroes, *MaxMem* entries long.

Accessing memory, then is simple: the contents of memory location L are *Mem[L]*.

There now follow functions for the individual instructions, one function per instruction. These constitute the execute phase and are called on individually by the *execute_instruction* function below.

```
#
# LOAD address -> Acc
#
def load (address):
   global Acc
   global Mem
   Acc = Mem[address]
```

The keyword *global* indicates to python that the name specified will not be found – and should not be created – within the function, but is to be found at the "top level" of the module.

As you can see, this function is simple – as, indeed, are they all: given the address, we simply transfer its contents to the accumulator.

[4]There is at least one other way to keep these in sync, using *tuples*.

```
#
# STORE Acc -> address
#
def store (address):
   global Acc
   global Mem
   Mem[address] = Acc

#
# CLEAR 0 -> address
#
def clear (address):
   global Mem
   Mem[address] = 0
```

The store and clear functions are similar in that they are given an address in memory; store copies the accumulator to that address and clear zeroes the address.

```
#
# ADD Acc += Mem[address]
#
def add (address):
   global Acc
   global Mem
   Acc += Mem[address]
```

Here we add the contents of the specified address to the accumulator.

```
#
# INC Mem[address] += 1
#
def inc (address):
   global Mem
   Mem[address] += 1
```

The *inc* function adds one to the *contents* of the specified memory location -- this is necessary to allow for *indexing* of arrays in memory, as this machine has no index registers for modifying addresses.

```
#
# SUB Acc -= Mem[address]
#
def sub (address):
   global Acc
   global Mem
```

```
    Acc -= Mem[address]
#
# DEC Mem[address] 1= 1
#
def dec (address):
    global Mem
    Mem[address] -= 1
```

The *sub* and *dec* are the subtractive equivalents of *add* and *inc*.

```
GT = EQ = LT = 0                          # flags
```

These are the flags or condition codes established by the compare instruction and used by the conditional jumps.

```
#
# COMPARE cmp Mem[address]: Acc, set flags
#
def compare (address):
    global Acc
    global Mem
    global GT, EQ, LT
    if Mem[address] > Acc:
        GT, EQ, LT = 1, 0, 0
    elif Mem[address] == Acc:
        GT, EQ, LT = 0, 1, 0
    if Mem[address] < Acc:
        GT, EQ, LT = 0, 0, 1
```

Note that the *compare* function *always* sets all three flags, although (obviously) two of them will be zero. Note also the use of parallel assignment.

The jumps all work in essentially the same way: a *target address* is provided and – if the condition is true (or it is the unconditional JUMP instruction) then that value is stored in the program counter register.

So, when the next fetch stage runs it uses the new PC value. This is why the automatic updating of the PC is done immediately after the fetch and not after the execute.

```
#
# JUMP address -> PC
#
def jump (address):
    global PC
    PC = address
#
# JUMPGT if GT address -> PC
#
```

```
def jumpgt (address):
   global GT
   global PC
   if GT:
      PC = address
#
# JUMPEQ if EQ address -> PC
#
def jumpeq (address):
   global EQ
   global PC
   if EQ:
      PC = address
#
# JUMPLT if LT address -> PC
#
def jumplt (address):
   global LT
   global PC
   if LT:
      PC = address
#
# JUMPNEQ if not EQ address -> PC
#
def jumpneq (address):
   global EQ
   global PC
   if not EQ:
      PC = address
```

As we are not running on a real single-input-device machine, the IN and OUT instructions are somewhat complex.

Most of the complexity is caused by the possibility of the user – from whom the input will come – typing in a non-numeric (or out of range for 16-bits) value.

Hence the *try/except*, which is wrapped around a loop which is at first sight apparently infinite. But look closer: we can, in fact get out of the loop in one of two ways:

1. The user types in a valid, in-range number, in which case the *break* jumps out of the loop, to the assignment statement

2. The user hits just Enter with no number. In this case the exception *texc.EndOfLine* will have been raised and caught; the code then exits the entire virtual machine by raising the exception *texc.ProgramEntered* which is caught by another module (these two exceptions are also defined in another module, *texc.py*).

```
#
# IN input -> Mem[address]
#
def inp (address):
   global Acc
   try:
       while 1:
           try:
               operand = int (raw_input (INPROMPT))
               if operand > MAXOPERAND:
                   print 'maximum allowed value (%d) exceeded' % MAXOPERAND,
               else:
                   break
           except ValueError:
               print 'invalid input, must be a number'
   except texc.EndOfLine:
       raise texc.ProgramEntered
   Mem[address] =  operand
```

The output instruction is rather simpler, as there is no user interaction to worry about. We simply print out the contents of the specified address.

```
#
# OUT Mem[address] ->
#
def out (address):
   global Acc
   print "out: %d" % Mem[address]
```

The module which called this one will be prepared to catch the *texc.MachineHalt* exception and stop the CPU running. Which is what the HALT instruction does.

```
#
# HALT - stop the machine
#
def halt (address):
   raise texc.MachineHalt
```

Now here comes on of the trickier parts of the code.

We already have lists of the instruction opcodes and their corresponding names.

Here is a list (in the same order) of the *functions* which implement the instructions.

The *execute_instruction* function actually does the fetch, decode and execute of the instruction contained in the address specified by PC.

First we get the instruction – from *Mem[PC]* – and isolate the opcode, which is the first four bits.

This is done (all in one statement) by:

1. Shifting the instruction 12 bits to the right (which puts the 4 opcode bits at the end) and then
2. ANDing out anything left before these last four bits[5].

The next statement using an AND to isolate the operand part of the instruction – the last twelve bits.

PC is then incremented – don't forget, the execute phase may well update PC, if we have a successful JUMP instruction.

Finally we execute the appropriate function, passing it the operand.

This is probably the trickiest single line in this entire program:

```
Execute[opcode] (operand)     # execute
```

Remember, Execute is a list of the functions which correspond to the opcode, so *Execute[opcode]* means the function which will implement the *opcode* instruction.

Given the "name" of a function, we call it by placing parentheses with an optional list of arguments inside – this is the *(operand)*.

So, this statement does indeed call the right function for the opcode and passes it the right operand.

```
#
# List (ordered by opcode) of virtual machine instruction routines.
#
Execute = [load, store, clear, add, inc, sub, dec, compare, jump, jumpgt, jumpeq,
           jumplt, jumpneq, inp, out, halt]
def execute_instruction ():
   global Mem
   global PC
   opcode = (Mem[PC] & OPCODEMASK) >> 12  # fetch and decode
   operand = Mem[PC] & OPERANDMASK
   PC += 1
#         print 'PC', PC, 'opcode', opcode
   Execute [opcode] (operand)     # execute
```

Finally, the *print_status* function is called when the program stops for any reason. It prints out the current PC value, Accumulator contents and the flags.

[5]The python shift right will "trail the sign bit" of any number – such as our instruction word – which begins with a 1 bit, i.e. would be a negative number in the 2's complement system. Half our opcodes start with a 1 and so we need to do this "masking out".

```
def print_status ():
   print 'machine stopped at %4.4d, R = %6.6d' % (PC, Acc)
   if LT:
      print 'LT',
   elif EQ:
      print 'EQ',
   elif GT:
      print 'GT',
   print
```