# Computing made Difficult

A C Norman and others I hope

February 27, 2026

# Chapter 1

# Intrduction

*Computers (including phones and all manner of smart devices) have been made so easy to use that often there is no need to think about exactly what is going on. Even programming can be mads to seem an almost trivial activity. Here the object is perhaps less to make computing difficult and more to uncover some of the depth that can be found in it when one looks just below the glossy surface.*

There are may ways in which the world tries to pretend that computing is easy. There are schemes that teach coding to children certainly starting from age 6. There are self-help books with titles along the lines of "Teach yourself programming in ¡so many days¿". Almost every new serious programming language or software package will trumpet that it represents the next step in rendering computer use accessible to all. Finally one of the claims for Artificial Intelligence is that it means that everybody can develop computer systems by merely giving an informal explanation of what they want achieved. A rather small amount of web search (which is of course really easy!) will back up all the above. But what is hidden in all that enthusiasm is that the behaviour of computers and software and the design and construction and analysis of programs has astonishing layers of difficulty just beneath the shiny and simple-looking surface.

There are basically two reasons for investigating this difficulty. The first can obviously arise if you are trying to build a computer-based product or solve some particular problem and you come face to face with the unhappy fact that the world is messy and that naive or simplistic techniques are not good enough. If you are an optimist this may come as a nasty surprise! The second which is the one emphasised here is when you understand that clever techniques, fairly intricate details and plain weird results can be fascinating –

and that coming face to face with some of them will let you build experience and understanding that lets you achieve more in the future.

So we start off here by noticing that many computing challenges had be presented in ways that do not need special skill or knowledge to appreciate. In plenty of cases there will be fairly obvious ways to start work towards resolving them. But then there are a dozen or more attitudes as well as problem categories that make it possible to unpick levels of difficulty seriously greater than were first apparent.

In some of the examples included here even the more complicated way to solve the problem will in fact be reasonably easy to grasp (once you have seen why it is necessary to go to all the trouble involved). In others it will involve somewhat messy data-structures or mathematics – but the cases we have chosen are intended to make these possible to understand and appreciate. Finally there are cases where there is no known solution or (worse) where it is known that there is no perfect solution. In such cases grasping how it is possible to demonstate that something is impossible is of itself a challenge worth facing up to.

So here is a sort of catalogue of ways that let you start with a simple task and uncover the challenges concealed beneath its simplicity. For each idea there is a reference to a section later on here that works through an example in reasonable detail:

**Look beneath the abstraction to understand how something is implemented or how it "really" works** There are an amazing number of instances of this. Any time you ask a computer to sort some data, do a database lookep, compile a program, create a file, fetch a web page or encrypt a message that is suposed to be kept secret there is a great deal of technology that happily most people can just take for granted. One can view this as rather like the situation with almost all technology from digital watches to aeroplanes – almost anybody can take advantage of them. Plenty of people will be able to present an overview of how and why they work. But the details end up almost unimaginably complex. So the attitude of this point will underpin almost all of the sections here! Here an analogy might be that leaping on a motorbike or into a car and driving off probably seems pretty straightforward. Understanding in full detail how the engine and transmission works to a level where you can see how to design the next version to further optimise power or fuel efficiency or cost of manufacture is a very different business! For computers one can consider details not just of the program being written, but also the compiler used to process it, the operating system that as installed, how the instruction set of the computer is implemented in terms of gates and other logic circuitry down to the device physics behind the transistors inside. Of course for *practical* purposes it is not generally necessary to have specialist understanding of all the levels, but gaining it reveals just how difficult it is to get a proper in depth understanding of computation. A happy though here is that the very earliest computers, while much less powerful and very much less convenient to use, were really a lot simpler and so looking back in history makes it possible to understand them from top to bottom much more easily.

**A product vs. a personal project** This is perhaps best explained by an analogy with feature films. One could dream that anybody with a modern phone could capture video and a few friends could take parts in film. The status of the cameras on current phones which are inexpensive (at least in comparison with professional gear!) and can deliver remarkably high quality is analagous to the way in which these days computers are cheap, powerful and can so an amazing range of things following just a few clicks on an app or a web-resource. By choosing the plot you would not need any elaborate scenery or props, and you can avoid the need for potentially dangerous stunts. Some films pretend to be constructed by pasting together "found footage" extracted from the phones of characters (who will normally have vanished or died) further boosting the illusion that almost no serious equipment is needed. To get a clearer view of reality look at the list of credits at the end of a film and see the size of team really involved in even a seemingly simple one. Check the budget involved and let that give another indication of the complexity of the full process – even before the issues of marketing and distribution are taken into account. So the reality is that in general the finished product almost conceals the fact that it relies on the work of hundreds anmd sometimes thousands of people: scouting for locations, make up artists, lighting technicians, foley artists who add sound effects and many many more down to the transport catering teams who support the people more on the front line. In general while being a member of an audience you are almost unaware of all of this – save that if it was missing that would be very noticable. This situation can seen as an "iceberg syndrome" where something that looks simple and where indeed an individual amateur could start to develop something in fact relies on such a wide range of techniques and skills that for any one person to master them all would represent a huge challenge. The message here is that making a a home-movie is amazingly easy, but the simpliciy there conceals the fact there is a great deal of difficulty in creatiing a block-buster. The same sort of things applies as between a bit of software created just for your own private use and a full-blown product.

**Consideration of edge and worst cases** There is a scheme called "Newton's Method" or "The Newton-Raphson Iteration" that provides a simple to implement way of obtaining numeric solutions to equations. As a concrete and rather easy example it can be used on the equation $x^3 - a = 0$ for some known value of $a$ to find the cube root of $a$. But even with a case as easy as this there are challenges. How fast will it get an accurate result? It needs an initial guess for its answer to start from – how much does that value matter. An especially jolly issue for this case is that if one accepts that in many areas of mathemetics and physics one is working with complex rather than real numbers it is necessary to accept that $a$ will have three cube roots – and the issue of which one Newton's method will deliver for you turns out to be a messier issue than you might have expected.

There are plenty of non-numeric instances of difficulty raised by asking about best and worst (and indeed average) cases in problems. For instance solving a Sudoku puzzle or planning how to return Rubic's cube to a tidy state may not be totally trivial, but trying to identify the hardest possible Sudoku board or the most awkward starting point for a Rubic's cube escalates the challenge sharply! Yet another example here come from "turtle graphics" – a computational model that has often been used in introductions of computers to the very young. In that world it can be quite easy to ask such questions as "If you continue with this pattern of movement will you ever find yourself exactly back where you began?" or "Might your turtle eventually fall off the end of the paper or indeed the world?" that make life tougher for yourself.

**Insist on total correctness in every case.** Pocket calculators and computers provide facilities to calculate trigonometric functions, logarithms and square roots of numbers. In normal circumstances one just truists the computer. But there are two ways to make your own life harder. One is to express a fear that the computer will occasionally get bad results. Well in 1997 there were significant sales of computers that did not even always get division correct! How do you test things like this?

To go further note that with a computer the answers will always have been clipped to some limited precision. On a pocket calculator this may for instance be 8 decimal digits. The full perfact answer to a calculation has digits beyond that - so for instance if $\pi$ is returned as 3.1415926 on a calculator (which can feel reasonable) there is an issue in that the true value is $3.1415626_5359\ldots$ and the value quoted should have been rounded up because if the $5359\ldots$ tail beyond where it ends. So the challenge is to evaluate all the elementary functions in such a way that for all possible inputs the result that is returned is correctly rounded – ie as accurate as is at all possible. And for this to be done first without intolerable extra cost and secondly with some proper scheme that can certify that the goal of perfection has been achieved. Amazingly there are people who have been arranging that! A few years ago one could almost always just assume that the computer's results would be more than precise enough for all reasonable purposes, but now people are using rather low precision floating point arithmetic for big parallel comutations in either graphics or artifical intelligence areas, and so these rather pedantic issues of precision come much closer to pracical reality.

**Portability** For small programs that are only intended for use over a fairly short perious of time and only by one person on their own computer it is not necessary to worry about portability. However larger projects raise more and more serious challenges. Windows, Macintosh, Android and Linux are amazingly not 100% compatible with each other, and it is often necessary to use techniques that apply to just one particular system. Intel and AMD processors, ARM and Risc-V are all different and are either at present in widespread use or may become so. There are areas where each of those needs custom treatment to achieve goals even as simple sounding as measuring time with the highest feasible precision. Use of a sufficiently high level language can conceal these issues by arranging that all the mess is embedded within the language and its associated library - but for our purposes that still leave curious people with a need to know exactly what is going on. And very frequently common solutions that paper over differences carry costs that oen might like to avoid.

**Demand the fastest (or most compact) solution that could ever exist.** The world of computer gaming is one where delivering the fastest frame-rate for the highest resoluation version of the action ia of serious commercial importance. And that is a matter of real directly measurable performance where there are basicelly few limits to what can be deployed to deliver it. That can lead to power-hungry and expensive video cards with amazingly elaborate driver software. Because much of that world is proprietary it is perhaps hard to get into, but it does illustrate that despite the fact that computers have become quite fast there are still areas where squeezing the best from them matters. This can involve both algorithm selection and coding style. For some application areas this aim to excel has the same sort of issues that mean that it will be different sets of athletes who dominate over 100 metres and over the 42195 metres of a marathon. In computing sometimes techniques that will be great for large problem instances will not show up well on smaller inputs. There are two classical illustrations of this to be found in books on computer algorithms. One related to finding all then shortest routes from a given starting point to other locations in a maze (which is generally referred to as a graph). The other is simply multiplying numbers together. We will summarise and discuss each of these and various other cases where optimising for speed or size turns a reasonable problem into a tough one.

One nice term sometimes used is "galactic algorithms". This is used to refer to ways of solving problems that in the long term would be better or faster than the simpler methods more commonly used, but achieve their closer approach to optimality at the cost of impressive overheads. Sometimes by way of a constant factor slowdown that would not by overcome by the fact that their cost grows a little more slowly than the standard schemes, and sometimes by such a level of implmentation complication and mess that it will basically never make sense for some rather minor improvement. A few examples of this style will be covered here. The term "galactic" is to suggest something of the magnitude of problem instances where some of these would necome competitive.

**Challenging underpinning theory behind a simple-looking problem** I do not have nice text here, just a list of a few

1. Simplify algebraic formulae.
2. Telling if it is possible to find values for all the variables to make a boolean formula evaluate to TRUE.
3. the 3n+1 challenge (ie Collatz).
4. Generate an unpredictable (ie random) sequence.

**Constraints that rule out obvious approaches** In some real sense all computer projects fall into this category because if you had a programming language or software package that aligned well enough with your task then everything would become easy. So to illustrate the point we will cover cases of extremely weak programming environments where it might not be obvious to start with that it will be possible to do anything much of interest, Each of the ones listed here and delved into in a bit more depth later can give insight into some particular aspect of computation:

1. Turing Machines and variants
2. Counter machines
3. Lambda-calculus and combinators
4. Cellular automata
5. Primitive recursive functions

Several of these start off seeming to be rather abstract and certainly not obviously practical, but for instance a highlight in one case is a report of how somebody has built a computer out of lego based on one of them such that in principle it is fully general purpose!

**Seek good partial solutions to intractable problems** Some – indeed many – of the challenges that arise in the real world turn out to be such that nobody knows how to solve them in general and there is serious reason to believe that that will always be the case. In a number of these it is even possible to prove that no general solution can be found. That opens the door to a lot of fun seeking either computer schemes that obtain approximate solutions or ones that sometimes or perhaps often get to the best answer, but are not guaranteed to complete the task. WQe provide several examples!

**Use wilfully perverse techniques that achieve your goal.** There can be great joy in exploring stupid ways of solving problems – and sometimes these emerge when a naive programmer submits their best efforts and you recognize that what they have achieved is a program that works but is magnificantly slower than one might have hoped. Even experienced software engineers can end up delivering solutions that in retrospect can be seen to be pretty well absurd. We can illustrate this with cases from simple tasks the like of which are set as exercises in an elementary programming class: reversing a list, sorting some data and the like.

**Challenges that make good puzzles** Would any reasonable person want to invent unrealistic or pointless tasks? Why yes - those involved in recruitment of any sort might want ”puzzle tasks” to set to candidates and while some questions they pose might merely test depth of knowledge, others want to look for inventiveness and the ability to think on the feet. Perhaps giving a preview of such cases undermines the joy in them, and maybe it will even be hard to tell which of the sections here have a component of this philosophy!

**Future-proofing** The landscape of computers has changed dramatically, and today there are still big new developments in prospect. Perhaps the two most visible are artificial intelligence and quantum computation, but one should really also note that the exploitation of the various forms of massive parallelism that is now available respresents a frontier. While some of the underpinning theoretical study of computation has remained valid for a long while, much practical software has a lifespan of a rather few decades. And segments of the theory that were central to all courses on computer science a generation back have much less clear-cut relevance than before. So we provide a few case studies that note both exciting ideas whose time seems to have passes and projects that against the odds have been kept alive.

**Emphasize what must not happen as much as what will** When specifying what a computer system will be for it is normal to describe what it will achieve. However the reality of things is that computer programs of any non-trivial size are liable to have flaws and will not always behave as intended. So there are cases where it becomes important to specify what must not happen as well as what should. And indeed in somc cases the negatives are actually the important constraints. We will cover two areas that highlight this: security and safety-critical systems. The first can usefully be partitioned into the consideratiuon of first encryption primitves (ie codes and the like) and then into protocols (how information is exchanged reliably between several parties). The counterpart to designing and building secure system is defeating same - and there is an amazing history of purportedly safe schemes being cracked open!

**Cope with the inevitability of human error** We all know that "To err is human", and we are all used to observing that computer systems have flaws. If you were concerned with safety critical applications (think of control of anti-locking brakes on a car, of the guidance system for a missile or for a device to be implanted in somebody's body) you may feel the need to reduce the chances of error as far as you possibly can. You may also wish to be resilient against all possibe forms of hardware malfunction. There is no simple silver bullet.

Both the hardware of computers and the software they run will be constructed by humans – or these days perhaps by an artificially intelligent agent. Anybody who claims that what they deliver has been built by some spcial and really reliable tool shoudl be asked about who created that tool and what basis there is for thinking its is perfact.

The inevitable result is that at least the initial version of any software must be presumed to be flawed. Testing – even very extensive testing – tends not to uncover all the defects. There are two approaches that aim toward perfection: (a) look for software building techniques that minimise (note ”minimise” rather than ”eliminate”) errors and (b) investigate ways to create formal proofs that the software ends up correct. We will have examples of or introductions to each of these.

**Make sense of abstract graphical presentations** Various schemes generate amazingly complicated images from rather simple recipes. Many people have come across the Mandelbrot Set which is one such instance, but we will point at a number more and consider how much understanding can be gleaned from reviewing the pictures.

Obviously these schemes for making life harder overlap in places, but it is also the case that several can all apply at the same time. The result is that challenges that started off seeming easy can end up causing quite severe headaches.

One might reasonably ask whether real people apply any of these principles in their lives. The next chapter sketches a particularly extreme case, but in reality a great deal of work starts with tasks that turn out to have depths that were not at all obvious at the start.

So it is very much the case that even though computing can be seen as really simple and we can keep telling ourselves “The computer only does what you tell it to”, just beneath that attractive simple veneer there are a great many wonderful, messy, complicated and difficult details.

# Chapter 2

# A case study

*It is easy to imagine that when an individual or a company sets out to build some computer software that the amount of work they will do can ce estimated by looking at the task and thinking about how complicated it seems. But a perfectionist can end up wanting to do so much more. . .*

This sketches one of the most impressive cases there has been of somebody starting a project and then going to town in the way they insisted that everything should be exactly to their taste.

In the early 1960s Donald Knuth began a project to write a book. It was a time of change in that somewhat before then serious books had been prepared by armies of compositors placing metel type in trays, but it was becoming clear that computer-aided publication was going to be the way forward. Knuth could have prepared his work the old fashioned way by writing it out by hand, getting a secretary to prepare a typed version from that and then letting a publishing house and printers loose to finalise things. That would have been the easy route.

However Knuth wanted greater control and knew that the standard route would not result in a publication up to his standards. So he diverted from his main book for a while and developed his own software to lay characters out on the page. There are multiple issues that have to be faced up to in that endeavour:

- Decide where to split lines and how to stretch text so that all lines end up neatly at the right margin;
- Kern letters propery – i.e. arrange the separation between individual letters based on their shapes so that the overall visual effect is of uniformity. This includes allowing for the way that letters in *italic sloping style* abut gracefully with upright text that surrounds.

- Present displayed mathematics well. Doing so leads to a need to handle Greek letters and a huge number of special symbols, to manage subscripts and superscripts, fraction bars, nested brackets of various styles and vertical alignment (for instance in tables or presentations of matrices);

- Schemes for the generation of an incorporation of diagrams, generation of cross references, indexes and for the formatting of a title page and chapter headers.

The resulting system, TEXended up a great succcess and since then it has been used a standard tool for scientific publication. Many would have viewed that as at least as large a task as writing the book that motivated it.

But that was not the end! Historically fonts had been designed by specialists and cut in metal. The transcription of those to computers was not in a very well developed state, and in particular Knuth felt that the computer fonts available to him were not good enough. One issue was that when you have a single style of lettering the exact shapes of small letters (for instance to use in a subscript) should not be simply scaled versions of the standard versions, and similarly huge versions for use in titles are not merely magnified copies of the original. So Knuth invented notations for describing the shaped of characters and how those shapes changed with size. And using that he developed the "Computer Modern" family of typefaces including all the symbols he would need in his book. This is another contribution that has lasted but that can be viewed as a deep diversion rendering his project of writing a book much harder than might have been expected.

It is still the case that this story is incomplete. In writing the programs that could lay out text and define fonts Knuth was very aware that reading a program written by somebody else can be really difficult because the justification behind all the choices they have made is not visible. So he developed a scheme known as "literate programming" where code and a careful textual explanation of what it was doing (and why) were woven into a single document. By running one decoder over that document he could recover code to pass to a compiler and use, but with a different decoder he extracted a TEXdocument by way of explanation. Within the single combined source the concept was that each fragment of code would be positioned adjacant to a careful explanation of it, and so as and when any changes were made it would be natural to keep the code and its documentation in step. Using this he was able to turn the commentary within the TEXsource code into a book "TeX the program" that could accompany his book that documented how to use it.

With all that in place he could get back to “The Art of Computer Programming” which expanded from an initial expectation of being a single volume into a sequence with each focussing a particular aspects of the subject. And these became standard issue to all aspiring and practising computer scientists amd TEXbecame the most common way for them to prepare papers and books.

This level of starting with a task that while large might have seemed reasonably straightforward and by demanding “better” escalating the projefct scope amazingly is a great illustration of the thoughts we present here. But for many practical reasons we will be concentrating on cases that do not demand quite the above levels of heroic enegy!

# Chapter 3

# Counter Machines

*One way to get a really deep understanding of programming is to strip away all the convenience features that modern systems provide and concentrate on the fundamentals. One interpretation says that flowcharts, which were originally invented to describe engineering and commercial activity and were commonly used with computers from the very start through the 1960s. In the day plastic templates to make it easy to draw the verious boxes were readily available. So this chapter goes back to basics by not just re-visiting flowchars but by discarding almost everything else.*

To fully understand computation it can be good to strip things down to absolute basics. Doing so may at first make it seem as if nothing useful can be achieved, or that it it could be that setting it up would be intolerably clumsy and laborious. Happily if you are prepared to take a few liberties with notation and if you are willing to view the time that a program would take to run as a total irrelevance things turn out to be less clumsy and less laborious then one might have expected. So in this chapter the emphasis will be on one particular way of looking at proogramming. It is the use of flowcharts, as often used when introducing computation to real beginners.

A text that aims to get people to understand what computers do and how they are driven might start with a version of instructions to make a cup of tea along the following lines:

1. Put water in the kettle;
2. Put tea in the teapot;
3. Switch kettle on;
4. wait a bit;

5. See if kettle is boiling: if not go back to step 4;
6. Pour (boiling) water into teapot;
7. Wait 3 minutes;
8. Done!

And having introduced the laborious step by step description of actions they then draw it out as a flowgraph with actions in square boxes, tests in squashed boxed sitting on their corner and loads of arrows that indicate the flow from box to box. One location is identified as the starting point and another and where to stop.

Well pretty well any program that does not involve defining and using functions can be rendered this way, and the chains of arrows provide a nice visual indication of what one would call “the flow of control”.

Switching a kettle on is not typically a primitive operation that a computer can perform, so in real programs the box contents will be individual statements valid in the programming language concerned. If we are trying to find a really deep understanding of computation it is natural to wonder how small a collection of different sorts of statement and different sorts of test it is possible to get away with and still have scope for interesting behavior. Counter machines[1] provide one extreme version. As considered here a counter machine has a (small) number of variables each of which can hold a non-negative whole number, i.e. $0, 1, 2\ldots$. In due course we may think of those numbers as codes for text using any of the standard ways in which characters can be coded as numbers. The program we will develop will have its input data provided in its first register (which I will call $A$), and all the other registers start off holding zero.

Apart from a box that ia labelled “stop” the only square action boxes that can be used have as their action statement that increments one of the registers. The only lozenge-shaped test-boxes that can be present check the value of one of the registers. If that value is zero they drop through to the next part of the flowgraph, otherwise they decrease the value in that register by 1 and go somewhere else. When the machine reaches its stop state the value in register $A$ is considered to be the result it has calculated. Although this is of course just an integer, just as was the case with the input it can be interpeted as character data. To make this point as clear as possible,

[1]There are a number of different names used for these primitive models of computation and a range of different sets of operations they can perform, but all the variations can be coaxed into modelling each of the others so the key results about them are robust. The version used here follows Minsky[**?**]

and computer file contaning text is stored on disc or transmitted over a network as some sequence of bits, and one frequently used scheme transmits everything in 8-bit chunks, with a scheme that means that the commonly used characaters (e.g. a-z, A-Z, 0-9 and various punctuation marks are fitted into one 8-bit unit (byte) while more exotic characters such as Greek $\alpha$, $\beta$, $\pi$ and the rest use two bytes and specialist symbols including many geometric shapes, lots of emoticons and pictures of the pieces for a chessboard use yet more bytes. One can then interpret this potentially rather long string of bits as the denotation of a binary number. In that way every file on your computer as a really natural interpretation as an integer and could be passed to a counter machine! Of course from a practical point of view this is totally absurd given that the only things we can do with numbers is to increment and decrement them. It would not take a very long input string to hit a situation where the number representing it was so large that counting it down to zero would involve more steps than there are atoms in the universe, and taking those steps would take more time than most people are prepared to wait. So this is to be viewed as a theoretician's model of computation. So the previous demand that you view computing time as an utter irrelevance really has pretty sharp teeth.

The big assertion to be made here is that for any computer program that can be provided with all its input at the start and deliver all its output when it stops, and subject only to the understanding that input and output will be encoded as big numbers, that it will be possible to devise a register machine implementation of whatever that program does. Is this going to be difficult? Well at some level yes it is – but once you have grasped how to attack the translation it is going to be less horrendous than it at first seemed. So the next few paragraphs show how the various key features in "ordinary" programming languages can be supported, Once that are in place transcribing the rest of the target program will be straightforward.

It is useful to start with better arithmetic then mere adding and subtracting one.

*Here I will show the bits of code in a programming-language like notation because preparing flowchart diagrams would pain me. But for the final version many need to be drawn out, perhaps especially the early ones.*

For addition consider setting up something that behaves like $A = B + C$ where $A$, $B$ and $C$ are registers and where $D$ is spare one. Well in fact it hardly deserves to be described as "difficuly".

```
while A!=0 do A--
while D!=0 do D--
while B!=0 do B--, A++, D++
while D!=0 do D--, B++
while C!=0 do C--, A++, D++
while D!=0 do D--, C++
```

This has risked destroying $B$ and $C$ along the way, so it carefully preserved their values in $D$ and then restored them. If you were willing to leave them as zero things could be simplified a little. But the overall idea is that the counter machine can do something $B$ times by counting down in $B$ so we sum $B$ and $C$ by counting up in $A$ first $B$ times and then $C$.

The big magic that makes setting up a counter machine a lot less difficult that it might first have seemed s that after having convinced yourself that the above does perform addition you can write boxes in your flowcharts with $A = B + C$ in them alongside the primitive ones that say just $A = A + 1$. You work on from there producing additional calculations that you can use in boxes but then expand out into the primitives if you are really forced to. This is not really cheating – it is just like program building in an ordinary computer language where you set up a collection of subroutines (which you sometimes call functions or procedures) and then use them freely in the higher level parts of what you do.

In what follows I will often assume that a target register starts off at zero and that there is no need to preserve anything but the trick of initially counding down in $A$ until it is zero and of saving values in $D$ shown above can be applied wherever it is necessary. And I will also suppose that the number of registers that my machine has, while finite, is large enough that I always have a spare one available.

Then of course multiplication can be coded as just repeated addition, so $A = B * C$ will be

```
while B!=0 do B--, A = A + C
```

and with that it will be clear that raising to a power, being merely repeated multiplication, is also straightforward. Well if expanded out fully the flowcharts concerned may start to look untidy. But if one views each operation that gets implemented as a nice block of nodes that can be packages and thought of and presented as a unit things are not so bad.

Subtraction involves a new issue because the numbers in a counter machine may never go negative. So the statement you first though of as being $A = B - C$ needs to be handled more like "if $B < C$ then drop through not changing anything, otherwise set $A = B - C$ and take the other exit from the lozenge". Given that it is easy to mechanise it by decreasing $B$ and $C$ in turn and noticing which one hits zero first. Then as necessary things are restored (using the "$D$ trick") or $A$ can be set. It should be pretty obvious that by using this that division can be coded up in a way that leaves both a quotient and a remainder.

It is perhaps useful to note that multiplication and division by known constants is rather easier. So for instance $A = 2 * A$ needs a single workspace register $D$ but is then

```
  while A!=0 do A--, D++
  while D!=0 do D--, A++, A++
\end{varbatim}
and halving $A$ if it is even amounts to
\begin{verbatim}
  if A!=0 then
    A--;
    if A!=0 then
      A--
      D++
      go back to start
    else               A was odd
      while D!=0 do D--, A++, A++
      exit reporting A odd and unchanged
   else
     while D!=0 do D--, A++
     exit reporting that A has been halved.
```

Given the above that multiply and divide by 2, and the fairly obvious small variations on them that multiply and divide by 3, 5,...one can in fact with a little bit of extra encoding of data get away with a counter machine that only has two registers, say $A$ and $D$. If you had really wanted say three registers $A$, $B$ and $C$ you handle that by putting the value of $2^A 3^B 5^C$ in the main register of the two-register setup. What would have been increment or decrement operations on $A$, $B$ and $C$ now expand into multiplications of (test) divisions by 2, 3 and 5. By using more primes there you can model as many registers as you feel you need. To do this properly you need to convince yourself that with one register that contains real data and one to use as temporary workspace you can manage the multiplications and divisions by 2, 3 etc., but those operations really are simple enough that that is not a severe challenge.

fundamental steps inside them this does not even make things seem much worse: you can write your code with blocks that say $A = A + 1$, $B = B + 1$ and so on for as many variables as you need and each just denotes a mess of lower level messing with the two real registers you have. This scheme is completely general save for one caveat. That is that if your machine needs input, say the number $K$, it will have to have that encoded into its register as $2^K$, and similarly when the machine stops its result will be an encoded version of the true answer.

So all is well and you can restrict yourself to using 2-register counter machines unless your resolution falters and you consider what it means in terms of the number of steps taken to perform some calculation. But it is proper to stress again that this is a game where you have agreed not to think about that!

It is now clear that simple integer arithmetic can be handled at least if you restrict yourself to positive values. The next thing to consider will be arrays, since they are a pretty frequent component of programs. Well in the same spirit that there was an explanation of how to encode a string of text as a number, here is a recipe for dealing with an arrany of $N$ integers with values $a_i$ for $i$ from 0 to $N-1$. Set up a string that starts with a 1 then has $a_{N-1}$ zeros before another 1, then $a_{N-2}$ zeros and so on down so that the string ends in $a_0$ zeros. So if the array was of length 3 and the values in it were 2, 4 and 6 we would set up 100000010000100. Now view this as the representation of a number in binary and view that number as an encoding of the state of the array. Well that step is simple - but it is now necessary to verify that the key operations of accessing the $j$th and updating it can be performed using a counter machine.

Actually those two operations are remarkably easy to arange. First note that counting the number of trailing zeros in the binary representation of a number just amounts to finding out how many times 2 divides into it evenly. And division by 2 is one of the things we have seen that counter machines can do. To trim off a trailing 1 from an encoding you just need the transformation $N \to (N-1)/2$ which is also straightforward. With those two operations in place it is then easy to trim $j-1$ entries from a packed array, leaving the item at position $j$ as the next to inspect, and it is then easy to read its value. To replace it all that is needed is to start by deleting $j$ items and then put back the replacement followed by everything that had been removed. To put a new value $k$ on the "front" of an array means just extending its binary representation by a 1 follow3ed by $k$ zeros. And that is $N- > 2^k(2N+1)$. Again the computation there is simple enough arithmatic that the counter machine can be set up to do it. The effect of all of this is that it becomes proper to write flowcharts with actions such as $A = B[C]$ that accesses the $C$th element of an array $B$.

The arrays set up as above could (of course?) be nested, and one way to cope with negative integers would be to represent a positive value $N$ as and arrray of length 1 $[0, N]$ and a negative value $-N$ as $[1, N]$. All the basic arithmetic operations would now need to extract and check the sign marker but that is “just a bit more programming”. And since we are interested in shoding how to make things difficult that can not be seen as a problem. Those who are serious massochists could look to the standard representation of floating point numbers as arrays of length 3 with one field for sign, an exponent represented by a number in the range 0..2027 and a 52 bit mantissa. All the basic arithmetic operations on floating point values amount to integer operations on the components of these triples.

Well with simple variables mapping onto counter machine registers and arrays packed up as explained above and strings represented as arrays of characters, with the individual characters held as integer codes (as they would be anyway in languages like C and C++) it should be clear that the flowchard for any program that does not perform input or output operations while running but is just presented with some input at the start and just generates results as it stops can be expanded into a flowchart for a counter machine. What may be more amazing will be that each textual expansion along the way when making this transformation only increases things by a constant factor, so the new flowchar based on an almost ultimately primitive model of computing will only be a constant factor larger then the natural one you started with. And when writing out your flowchart you can use essentially all the operations from the instruction set of a traditional computer in the boxes.

What about function definitions and function calls? Well even they are not a disaster. If you review the way that arrays have been introduced here you will observe that they do not have any predefined limit to the number of entried in them. In fact the representation used behaves more like flexible lists than rigid arrays and slightly different ways of looking a them allows one to perform operations that amount to pushing a new item onto the front of a list and at some later time popping it off. Those operations are just what you need to provide a stack structure that keeps track of procedure calls. And what is goin on there is really a fairly close relative on how compilers work when mapping procedures and their calls onto the hardware of real machines. Working through the full details here would become tedious but anybody who has followed this far should be able to sort it out for themselves if they really wanted to. The conclusion one ends up with is that counter machines can express pretty well any computation that an ordinary computer could be programmed to perform subject mainly to the constraint that all input data must be available from the start and no output should be inspected until the program terminates. For many purposes the limitation will not be severe

A good joke about counter machines is that one of the earliest transistorized desktop electronic computer was built at a time when hardware to do complicated things such as addition and multiplication was seriously challenging, so internally it worked by counting, and it got as far as offering a square root operation to its users. So perhaps the ideas here have more pracical impact than you might have expected.

Now some of the constructions shown here are unduly general and so it would be possible to model "real computers" in a more compact and possibly more efficient way. While we are only concerned with abstracion that is again not a big issue, but it does make one ask "Given a computation that is to be done, what is the most compact counter machine that will achieve it?" or in other words how much better that the direct modelling shown here can we do? This is not just a difficult problem. It is one that in general can not be solved in any systematic way. It is natrual to feel that if ons has a counter machine with $M$ nodes that solves a problem then if resource constraints are being ignored one can enumerate all the counter machine configurations with less then $M$ nodes (start by cataloguing all directed graphs with that size limit) and just check which if any behave in the same way as the original. By scanning from smallest opwards one would natrurally come across the best. The painful reality here is that given a counter machine (or indeed any other sufficiently general model of computation) there can be no algorithm that will in general certify that example setups will have terminating patterns of computation. A consequence of that is that it can not be possible to guarantee to be able to verify that two counter-machine configurations have the same behaviour. However it will be possible to spot some cases that agree, and to identify others that do not. So for a *really* difficult problem design software that does the best you can to take the description of a counter machine and optimise it.

There is another famous challenge-problem assocoiated with counter machines, Consider all possible machined with $M$ nodes (and $K$ registers). Consider their behaviours when started with all their registers zero. Among all of those which one halts and when it does has the largest possible value in its first register. Obviously there can be machines that never halt - perhaps the simplest version just increments a register and returns to its starting state, so it sits there counting up for ever. Such cases are to be discarded – it is just machines that actually terminate that are of concern.

With a truly tiny number of nodes this question is easy to answer. A machine with one node plus a stopping one can not do better than to make its first node increment its register and transfer to the helt node. So the answer there is 1. An alterative version of the challenge is to maximise the number of steps taken before the halt state is reached – both versions are remarkably tough. It is plausible that a register machine with just 2 registers and only 3 nodes (plus the stopping one) can compute for much longer than you would have expected before terminating, and I here leave that as a topic for enthusiasts to explore.

*Hmmm - google AI summary for "busiest minsky machine" seems to suggest that a 2-register 3-state machine might take thousands of steps and then stop. I have not spent time working out what design behaves that way and the AI summary feels slightly incomplete and in particular does not point me at "proper" papers or reports. So I hope that somebody better at web search than me can find proper detailed documentation!*

```
https://codegolf.stackexchange.com/questions/279153/
long-running-section-11-4-minsky-machine

‘‘Rick J. Griffiths in his dissertation in 2003 has adapted
the busy beaver problem to Minsky’s "register machines".
His program is written in Java.’’

and that is a Cambridge Part II dissertation and I may be
able to find a copy in the Computer Laboratory archives.
```

# Chapter 4

# Turing Machines

*One of the biggest difference between a calculator and a computer is that the computer has a serious amount of memory. When one talks about using a computer to solve some problem – say sorting – implicitly that should mean that you consider not just sorting a set of 10 values but that you look at processing as many as anybody could ever provide. This sets an agenda that the key thing that a model for computation must provide is memory. Turing machines do this by having as their most important feature a "tape" where data lives on it as a sequence of symbols. The tape is assumed to be long enough for whatever task is to be performed. This is one of the most important basic abtractions of what computers are and hence a model in which it is possible to explore what they can and can not do and how much time or space they may need to do it.*

There have been two or three big reasons for looking for minimal schemes that are able – in some sense – to compute.

1. If you want to build a computer it is very reasonable to try to make things easy for yourself by designing the simplest possible device that you can. Even though that might make life impressively harder for those who want to write programs for it. There are modest applications of this idea that have led to very successful commercial designs the successors of which are in widespread use today – but the extremist version of it may provide a great basis for a fun practical construction project. The assertion "I have made by own computer from scratch" is obviously a good one to be able to make;

2. If you are serious about wanting to develop a robust theory that *really* lets you understand what computers can do and what they can not it

is rational to start with something simple. What you will be trying to do will be difficuly enough without needing to worry whether all the special capabilities built into modern computers and programming languages make big differences;

3. Those who are concerned with how long it should take to solve some problem will find there are huge and shifting complications if they consider hours. minutes and seconds on a range of desktop and laptop machines as well as mobile phones and embedded controllers. By looking at timings on truly reduced hardware their results will be less of immediate relevance but can be ones that will remain valid as this year's computers are replaced by next years ones that are posisbly from a quite different manufacturer.

The best known minimalist sort of computer is the Turing Machine. This emphasises the fact that a computer of any sort will have memory and if viewed from a distance all that it does can be seen as taking steps that each inspect and change a single item within that memory. In focussing on how data is stored it does not take any steps to make it particularly easy to coax it into solving the problem that interests you.

The storage provided by a Turing Machine is a tape which is marked out in cells. Each cell can hold one of a modest number of symbols. The number of symbolc allowed in one of the parameters that describe exactly what sort of Turing machine is being considered. The tape is made long enough for whatever task you are happening to try to process. Some people would characterise that as a tape that is infinite in length, or would use the word "unbounded", but in any particular computation that the Turing Machine takes only a finite segment of it will be used. So for practical experimentation it can be acceptable to provide a limited length "tape" and view an attempt to go beyond its end as merely reaching a limitation of the physical approximation to the abstract machine.

The Turing machine starts with whatever input data it needs ready on the tape. It then works step by step: it has a read/write head positioned over some call of the tape and a cycle it takes will inspect the symbol there and based on an internal state (from a limited number) it will write back a possbly changed symbol and move the tape left or right. It also transfers into some internal state. One state will be special in that entering it causes the maching to halt. At that stage it is expected that it will be written its result onto the tape. The number of distinct states that the machine can be in is the second parameter alongside the size of the alphabet of symbols on the tape that characterise it.

Programming a Turing machine has to involve setting up a table that is indexed by which symbol has just been read and which state the machine is in. When that information is used to inspect a row in the table you can read off the symbol to be written, the tape movement to apply and the identify of the next state that the machine should be in. A reasonable expectation is that designing tables like that to perform even modestly elaborate computations will be a bit painful.

One compelling reason that theoreticians like Turing Machines is that any such machines has its entire behaviour defined by this table, and it is easy to write out such tables on paper, set them up in computer programs for simulation and to set them out as sequences of symbols written on a tape.

One way to prove that it is really worthwhile setting up this fairly clumsy looking model of computation is a proof by construction that it makes it feasible to build a mechanism that follows its behaviour patter. In LEGO!

`https://beta.ideas.lego.com/product-ideas/`
`10a3239f-4562-4d23-ba8e-f4fc94eef5c7`

is a concrete realisation of this using under 3000 LEGO parts. While that is quite a lot, it can be put in perspective by comparing with the official LEGO kit to make a model of the Star Wars Death Star, which comes with 9023 pieces – but rather fewer gearwheels. And an astounding thing is that the only limitation at all on what thie Lego Turiing machine could do is set by the length of its “tape” and at least in imagination it would be easy to make that really long.

When considering building a real working Turing Machine it is reasonable to ask just how much mechanism it needs to have before it can actually perform any useful calculations. The LEGO version has 8 states and handles an alphabet of 4 symbols on its tape and that feels as if it might be quite limiting, however the astounding thing is that if you had a long enough tape (and seeing how to make the tape a *bit* longer is surely not a terribly tricky technical challenge even if giving it a capacity of thousands or millions of cells starts to seem absurd) this is enough to allow the machine to perform **any** computation that any other computer can. To be more specific it will be possible to set up initial contents on the tape where the first part amounts to “program” that documents what is to be done and the rest is the “input data”. A very special case of this is that the program part can explain how to behave as if the rest of the tape is being used by a Turing machine with a larger number of states and a bigger alphabet. For instance if one wanted to have 8-bit characters on the tape the machine that would be emulated would systematically use the contents of four consecutive cells on the physical tape to represent a single byte on the bigger system. The details of making one Turing Machine emulate another even if the latter is large really amount to nothing more than messy programming. If necessary multiple cells on the real tape will be grouped together so they can be treated as if they were single cells using a larger range of symbols. Some part of the tape will be used to record the current state of the emulated machine, and a significant part of the fixed data on the tape will be dedicated to a table recording how transitions happen. To orchestrate all this the head of the baseline Turing Machine will be very busy zooming backwards and forwards between the location of the active point on the simulated tape, the record of the state of the simulated machine and the transition table. It will be possible for it to find each of those by marking their locations with special symbols that are not used elsewhere. Details are not given here because setting it up it is closer to boring details programming work than to anything especially exciting.

However designing, understanding and using Turing machines that are really close to being mimimal can be huge fun. The sort of fun that there so often can be in seeking the most compact way to achieve a task! But really small universal Turing Machines can sometimes demand ugly ways of encoding the data on their tape or suffer from needing unreasonably large numbers of steps to reach results. Subject to that issue it is amazing how small a machine can be while still being capable of emulating larger machines and hence solving any problems that those larger ones could.

A machine using just 2 states and 3 symbols exists and is characterised as “almost weakly universal”. It will not even halt when it has completed its work and it also needs its tape initialised to a particular but non-repeating pattern in all regions beyond the input data, while standard machines will not read from the tape beyond the region that is explicitly data - it also tends to take a great many steps to get anywhere, while modestly larger machines can actually be quite efficient. As benchmarks for what is possible it is known that with just 2 states but an alphabet of 18 symbols universality can be attained, and similarly fpor just 2 symbols and 19 states. Between those 5 symbolc and 5 states is also sufficient – much work has gone into charting the boundary and discussing exactly what rules should apply.

Once one has demonstrated that a basic and fairly small Turing Machine can be used to emulate a machine with more states or a larger alphabet it can be reasonable for subsequent work to feel free to relax those constraints to make machine design simpler. and in particular at least from a theoretical perspective it is acceptable to imagine that the “state” part of the machine can do anything that an “ordinary computer” with no unbounded memory can. The tape is then just needed to provide the unbounded storage that it is good to have when analysing algorithms. To slightly simplify the proper and general result, if the "ordinary computer" completes a run within $N$ steps it can not possibly have touched more than $N$ distict memory locations. If its memory is modelled by data on the tape then the most remote bit of data ever accessed can be no more than about $N$ cells away. Well we may aggregate raw tape cells so that some symbols are stored spanning across say $K$ of them, but then the furthest accessed data is only $KN$ away, and the Turing machine should snly take time proportional to that to access it. So we find that each of the $N$ steps of the ordinary machine gets emulated in time bounded by something of the form $KN^2$. This quadratic overhead would of couse be calamitous in practise, but is modest enough for a great many theoretical studies.

Rather simple extensions and generalisations to Turing Machines allow for yet better efficiency for many problems. One can consider a device wither with more than one tape or with just one tape but more than one read/write head. As a concrete example of how this makes things easier, Merge Sort was a solid solution to sorting vast amounts of data when computer memory was small and the data has to live on magnetic tapes. One version of it worked by starting with the unsorted data split between two tapes and at each step it compared the two items next visible on those tapes and transferred the larger to an output tape. By the end of the pass the data originally on the two tapes had been merged. Well a proper version would handle not just a pair of items at once but as much as it could reasonably fit into its memory. With that sort of scheme if the total amount of data was about $N$ times larger than the computer's memory capacity the sorting could be completes after only $K \log N$ passes for some fairly small value of the constant $K$. The treatment of those tapes and those in a multi-tape Turing machine are closely analagous to one another, so a multi-tape TM provides a really solid model for analysing that particular algorithm while avoiding the need to worry about detailed characteristics of any physical computer. Similar reasoning applies to enough other tasks that Turing machines pretty much set the gold standard for properly pedantic discussion of assymptotic growth rates in cost.

The overall message here is that if you really want to make things hard for yourself try to design the most compact Turing Machine that can do the calculation you are interested in, seeing how you can make trade-offs between the number of states, the size of the alphabet you allow on the tape, the ugliness of how data has to be coded for use and just how many compoutational steps will be taken to obtain your solution. That is a wild and confusing space to search within. A variation on this is just to look for Turing machines that run for a seriously long time but then stop when they are started on an empty tape. For really small machines bounds are known. For instance if the tape can only hold one of two symbols in each call and the machine has 2, 3, 4 or 5 states the number of steps it might take before termination can be 6, 21, 107 and 47176870. With more states the number of steps becomes infeasible to write using commonly-used notations! These results show rather clearly that very small systems can have extraordinarily complex behaviour and is systems as small as these can behave in such extraordinary ways the detailed understanding of larger and less artifial ones will be difficult in the extreme.

Perhaps the most important resully regarding Turing Machines is that if you are given one there is no systematic procedure you could ever have that would guarantee to tell you if it was going to terminate. This result emerges from the techniques that allow one machine to emulate another in that thet task starts by representaing the second machine in the form of data that cen be processed by the first. It imagines there was a "systematic procedure" that could take such a description and judge whether it represented a machine that woudl terminate or not. It is pretty obvious that it could just emulate the machine and if that led to a termination state it has an answer. The problem will be that it has no bound on how long it might need to run the simulation for and as a result it is really hard for it to be certain that the emulated machine is never going to terminate rather than it just having a very long run-time.

One big trick is used! Given the imagined "will this machine terminate" machine (it is perhaps useful to think of it as a program and the patterns on the tape that describe it as its source code) one makes a trick version. This (if it worked) would decide if its input was a terminating program and in that case it would go into a boring infinite loop. If on the other hand it determined that its input was a machine that was going to run for ever it reports that fact by halting. You give a description of this new machine to inself as its input data. Now because of the trickery upi have something that will terminate only if it does not, and vice versa. Ooops that is impossible! And the only assumption that had been made was that there existed a Turing Machine that could determine if another one was going to halt. The inescapable conclusion is that such a machine can not exist. Phrased in the way this is normally spoken on "The Halting Problem is Undecidable". Given this result it is possible to deduce that many other questions about programs can not always be answered. That does not mean that in particular instances it may not be possible to resolve them – just that no effective procedure that canm cope with every case can exist. Well if (in general) one can not tell if a Turing Machine (or program) is ever going to terminate it is going to be impossible to tell if two programs always compute the same output (in particular one might sometimes halt when the other does not). In turn that means that guaranteeing to find that smallest program that achieves just what a reference one would can not be done automatically. These results pretty well leave resolving those sorts of questions in particular cases as begin "difficult".

To finish this chapter it is proper to mention another variation on these machines that has enough witty consequences that it will form the basis of a whole separate chapter. Ordinary Turing machines are very much mechanical and deterministic devices that always behave in unambiguous and predictable ways. An entertaining variation will be the class of machines where the state transition is not quite deterministic. In some cases the machine will be able to choose for itself from two different next states, with no pre-programmed or external guidance. If the choice was made by notionally tossing a coin in each such case you would have a randomised machine, and exploring the capabilities there would be entertaining. But the most important case here is where the decision between the alternate paths is made as if some good fairy waves a magic wand and the computation proceeds such that if your calculation could at all possibly succeed it now will. This is obviously a delightful fantasy, and the resulting model of computation is referred to as a "non-deterministic Turing Machine". It seems obvious that machines like this could never exist in reality – but in fact if you abandon all concern for timings they could be emulated, and to date nobody has been able to prove that there is no way of building a rather efficient simulation of one.

# Chapter 5

# Galactic algorithms

*Some procedure are not just difficult to apply: they can have costs that dwarf numbers that are merely astronomical. Can such grotesquely infeasible schemes ever be worthy of study? Well there is a resounding yes to that question where the algorithm – even if somewhat ridiculous – can be shown to be at least theoretically better than any other at least in the lng run, or even better if it can be proved to be optimal.*

Galactic algorithms are ones that have costs that grow slowly with the size of their input to an extent that for large enough cases thay will do better than simpler methods, but that suffer from such large overheads that they will never become practically useful for any plausible cases. The finest ones will provably achieve the slowest possible growth rate for the problem that they solve. Some of the most impressive have overheads that can be expressed as a constant factor multiplier in their cost formula with that constant exceeding astronomical proportions. There will obviously be case that come close to fitting into this category. Because such methods are not of a lot of practical use one cound be tempted to view them as frivolous, but where they represent a better growth rate than any other known method or are provable optimal in that respect they can both be of intellectual interest and can provide ideas to guide work towards schemes that can be useful in the real world.

A number of galactic algorithms illustrate how much more complicated things get if you aime for the absolutely best possible method, but here we start off explaining one that has been known for some time and is neither terribly complicated nor terribly expensive, but that despite it giving a theoretical speedup will onlt achieve that in cases pushing against the size limits of the biggest available computers, and even then its advantage will be slight.

## 5.1 Matrix Multiplication

The straightforward scheme for multiplying a pair of square matrices of size $n \times n$ has cost that grows proportional to $n^3$. Our first galactic algorithm represented a breakthrough discovery that this growth rate was not the best possible, and that a tolerably straightforward scheme could achive a cost that grows as $n^{\log_2 7}$ which is about $n^{2.80735}$. To show how this is done it is sufficient to consider the very simple case of forming the prodict of a pair of $2 \times 2$ matrices $A$ and $B$. The four elements of their product as evaluated in the classical manner aee just $A_{11}B_{11}+A_{12}B_{21}$ and three rather similar looking values and a total of 8 multiplications and 4 additions are used. A paper by Strassen[**?**] instead managed to show that using just 7 multiplcations that worked on carefully chosen sums and differences (e.g.$(A_{11} + A_{22})(B_{11} + B_{22})$) it was possible to express all the values needed in the product matrices as sums and differences involving these 7 values. Here we leave finding exactly how to do that as either a puzzle to work on or a topic to research! A variety of different schemes have been found, all ending up with just 7 multiplications but some with more and some with fewer additions.

Once the base case of $2 \times 2$ matrices has been handled large case can be treated by partitioning each matrix into four blocks (padding out with zeros if necessary to make those blocks all square and all the same size) and coping there by using just 7 multiplications (which are now matrix ones) on those half size blocks. An analysis of the growth rate in time spent using this scheme fairly clearly suggests that doubling the side of the matrices multiplies the amount of work by a factor of 7, hance the $n^{\log_2 7}$ result. This is clever and quite elegant, and not even too hard to program. But as against the classical method it struggles even to break even until matrices are larger than almost all applications, amd it certainly does not deliver truly valuable speedup for some way beyond that. A further concern with this approach is its impact on how rounding errors propagate through the calculation.

But this is not the end. Since the Strassen work others have sought to find schemes that are at lease assymptotically faster, with a long sequence of reports og gradually reduction of the exponent in the cost function. As of 2026 the best growth rate that has been reported is around $n^{2.37134}$ which looks like a distinct improvement over Strassen. So far nobody has been able to show what the ultimate limit for solving this problem will be. Most of these improved schemes work by splitting large matrices into more then 4 blocks and all of them are even further from practicality then the Strassen method.

## 5.2 Integer multiplication

Multiplying integers is a pretty fundamental operation, but here to make things harder the emphasis will be on large integers – typically ones with from hundreds to millions of decimal digits. Things start off in a style almost parallel to the matrix multiplication one, with first the simple "schoolbook" method that will form the product of a pair of $N$ digit numbers using around $N^2$ operations. The first improvement on this is due to Karatsuba[**?**] and works by observing that for 2-digit numbers one can form the partial products $a_1b_1$, $a_1b_2 + a_2b_1$ and $a_2b_2$ using 3 rather than the obvious 4 multiplications by calculating $a_1b_1$, $(a_1 + a_2) \times (b_1 + b_2)$ and $a_2b_2$ and doing a couple of subtractions. By taking large numbers are splitting each in half this can be applied recursively leading to a method where costs grow like $N^{\log_3 2}$. This scheme can become proper to use when numbers exceed say $10^{300}$, but the exact break even point will be quite sensitive to details of the computer used and just how everything was coded. So this not a galactic algorithm.

The next scheme of note may start to be competitive when multiplying numbers of magnitude around $10^{5000}$. The exact point where it becomes realistic can vary quite a lot and 5000 digits is perhaps better characterised as stellar rather than galactic, so this is a method that is at least relevent in some slightly specialist cases. It is however much harder to explain in elementary terms, and so what will be presented here is an overview perhaps uses enough technical language to show that it starts to count as "difficult" or at the very least "reasonably advanced". So for those who do not recognise the words used here this will just show how much messy technology a simple-seeming task such as integer multiplication can involve. For those who then want to follow through by re4ading textbooks or scouring the web it may give a broad overview so they know how the things they read up on fot together. And those who already understand this well can just smugly skip to the next section.

Schönhage-Strassen works by first padding its integer inputs so that each have a suitable power of two bits. At least as an abstract algorithm if expresses its operations in terms of bits not digits so it is kept fully honest and does not hide costs within arithmetic on digit-sized units. Then if the numbers are of length $N$ it clumps the bits into clumps of size around $\sqrt{N}$ and views each clump as a digit, and so there are around $\sqrt{N}$ digits. Well there can be some pedantry about the exact size of digits that is needed but the decomposition as described here is close enough for use in this overview.

The next issue is that it views each collection of digits as a vector and it takes a Fourier Transform of it. So what is a Fourier Transform? Well it amounts to multiplying a vector of length $K$ by a special form of matrix where the entries in the matrix are all powers of a value $\omega$ where $\omega^K = 1$ but no lower power of $\omega$ has that value. In the normal world one would say that $\omega$ is a complex $K$th root of unity, and that it is primitive, so at first it looks as if complex numbers are involved. However to avoid them (and any risk of creeping rounding error that could hurt if floating point arithmetic was used), Schönhage and Strassen used modular arithmeic. In other words they have some number $P$ so that after any operation on integers they just retain the remainder when the natural result is divided by $P$. A big part of their cleverness is that by choosing $P = 2^M + 1$ for a suitable $M$ two wonderful things can be achieved. First it becomes possible to use 2 as the value for $\omega$ and that means that multiplication by powers of $\omega$ can be mechanised as simple cheap shift operations. Secondly the remaindering operation where a double length integer is to be reduced modulo $P$ can be performed by just subtracting the top half of the number from its bottom half.

One interpretation of what a Fourier Transform does is that it views its input vector as the corfficients in a polynomial and evaluates that polynomial at all the powers of $\omega$. If that is done to two polynomials and the resulting values are pairwise multiplied the result will be the values of the product polynomial at all those points. An inverse Fourier Transformation will act as interpolation and recovers the coefficients of the product polynomial. This can be re-interpreted as an integer following some simple carry operations. A further key feature of Fourier Transforms is that on using vectors of size $K$ can be completed using around $K \log K$ arithmetic operations. Of course here each of those arithmetic operations is using modular arithmetic modulo $P$ where $P$ is a pretty huge multi-digit number – but because of all the powers of 3 involved that is not in fact seriously painful. In fact the worst part of all of this is the pairwise multiplication of elements of the transformed vectors and that generally needs to recurse into a further layer of use of the whole procedure – at least until things get small enough that classical methods will win.

The "polynomial product" produced by the Fourier Transform scheme has been computed modulo $P$ but we want exact and correct integer results. Well provided $P$ was greater than any value that could legitimately arise in that product all is well and the calculated values will be the perfect integer results as needed.

The above explanation know it is a sketch. It has omitted the need to pad various things up to powers of 2 and the proper care about splitting the inputs up so that the value of $P$ used is big enough to ensure that results are correct. Those details of course matter in a proper formal explanation of the algorithm and in any implementation of it, but perhaps they repesent details best defered to a second reading about the method.

The Schönhage-Strassen scheme for multiplying $N$ bit numbers has a cost that grows proportional to $N \log N \log\log N$. Because it is concerned with truly gigantic input numbers (i.e. values of $N$) it becomes proper to worry that if digits making of numbers are stored in computer arrays of length $N$ then index arithmetic used to work out which digit to touch next can become beyond th4e scope of single computer operations. For instance a modern 64-bit computer will not cope trivially with arrays with more than $2^{64}$ elements! So the proper analysis of costs here needs to be done in terms of some suitable abstract machine that does not have any limita at all on the bulk of data it can store and maniplulate. Variations on Turing machines are commonly used, and amazingly some of them can deliver the cost growth rate noted above.

This scheme, however messy, is still used in some practical applications. But it is not the end of the road. A succession of authors have described yet more elaborate schemes that reduce the $\log\log N$ term in the cost function, culiminating one[**?**] where the cost is precisely proportional to $B \log N$. This is a great theoretical result in that it is conjectured (but not yet proved) that this is the best growth rate that can be reached. The technology applied is significantly higher powered than the scheme covered above and so there is not even an attempt to explain it here, but those who really enjoy difficulty can read the 45-page paper that introduced it! One of the authors say that for this method to be cost competitive the numbers involved would need to be rather large: "Even if each digit was written on a hydrogen atom, there would not be nearly enough room available in the observable universe to write them down." That surely qualifies it to count as galactic.

## 5.3 Regular expressions

The separate chapter here on pattern matching also presents a problem that might have seemed tame but is in fact galactic.

## 5.4 Primality checking

It is easy to check if a small number $N$ is prime – just check each potential factor up to $\sqrt{N}$. For larger numbers there are fine and practially very sensible methods some of which rely on a supply of (genuine) random numbers are deliver a result with arbitrarily small probability of error, or that rely on currently unproven results in number theory. But a scheme introduces by Agrawal, Kayal and Saxen[**?**] runs in time bounded by a polynominal in the number of digits it takes to express the number and guarantees a correct response without needing to make any qauestionalble assumptions. This is difficult in two senses! The mathematics behind explaining and justifying it are somewhat tough, and if it was implemented the suggestion is that its cost would would be significantly slower than using probabilistic methods for all numbers with no more than $10^{1000}$ digits. It is so labotious to use that even for numbers of a few thousand bits (as used in many cryptographic contexts) is is not really feasible at all as well as it being hugely slower than other methods. So here we can report that this exists, that it is a fine example of the levels of difficulty you get when trying to get the very best solution to a problem, but that those who want full details may first need to attend courses on number theory!

## 5.5 Minimum Spanning Trees

A graph here is a collection of vertices with edges that join some of them. A spanning sub-tree is a setset of the edges such that it is possible to get from any vertex to any other one only traversing edges in that subgraph. A graph may have weights (or costs) associated with each edge, and then a minimum spanning tree is a spanning sub-tree such that the sum of the weights on all its edges is as small as possible. Often in university courses on datastructures and algorithms procedures by Prim and Kruskal[**?**] that find such minimum trees are presented. In the more advanced versions of such courses improvements on the basic versions of those using forms of priority queues known as Fibonacci Heaps[**?**] get discussed. If the graph hash $m$ edhes and $m$ vertices these schemes can guarantee to find a minimum spanning tree within time proportioanl to $m + n \log n$ and that is typically good enough for practical purposes. However a fancier method by Chazelle[**?**] has a growth rate that scales as $m\alpha(m, n)$ where $\alpha$ is an inverse Ackerman function and grows exceptionally slowly such that for any conceivably feasible practical graph its value will be no greater than 4. So until we are really at or beyond galactic scale the costs of this method grow linearly with the number of edges regardless of the number of vertices. As with the previous examples in this chapter this represents a breakthrough such that for large enough graphs the method will beat all the ones that can been developed before. As with other examples in this chapter the details of the extreme method defy compact explanation, but the overview is that they replace the Fibonacci heaps (which are themselved somwhat messy!) with data structures known as “soft heaps” that mostly provide priorty queue operations in a really cheap way but that do not always yield the correct result. However they do offer bounds on the frequency with which errors can be introduces, and the Chazelle method as a while allows for that by making checks as it goes and where necessary back-tracking to correct for mistakes. The presence of the inverse Ackermnann function in the eventual cost bound gives clear warning that this is all far from straightforward even though right here we are not documenting just what that function is or how it arises!

## 5.6 Perfect data compression

When data of any sort is to be transmitted over a slow network or sored in an archive it may well be good to produce a reduced size version of it such that when the original data is eventually required it can be reconstructed. The case considered here is where the original data must be recovered perfectly – that is in contrast to data compression often used with images (`.jpg`) and sound (`.mp3`) where ending up with a version that is in some sense "close enough" to the original will suffice. There are many commonly used schemes for compressing data and a numbers of then use the word `zip` in their names. But none of those even pretend to achieve the best possible reduction in size.

There is a scheme that is in a sense optimal, but the big problem with it is that it is even worse than the previous galactic algorithms just considered, in that it can be shown that it is and will always be impossible to write a program to implement it!

This scheme is known as Kolmogorov compression and the idea behind it is really simple. A body of data is represented by the most concise program that, when run, will regenerate it. Well that definition raises an obvious issue: "what notation should this program expressed in?". From a theoretician's perspective the response is that any notation at all may be used. If then one wanted a version of the data to be decoded on a different computer or one that provided implementations of different languages all you do is construct code in the notation that will suit this different computer that will emulate the one the data was compressed by. This simulation code will be of some definite finite size and so sticking it on the front of your compressed data will only alter its simply-measured size by some additive constant.

The unsolvable problem that makes this scheme one only to dream of is that there is no way to guarantee to find a shortest program to generate some given output. This is a consequence of the fact that given a program there is no algorithm that can guarantee even to tell if it will terminate in finite time, far less give information about its output. But this is only a practical limitation! One might expect that something that was impossible to implement would not have many uses, but amazingly that is not the case.

Here are two that illustrate the power of this concept:

### 5.6.1 Testing for randomness

A funadamental result is that if data is truly random then the most concise why to express it is basically to put it in a simple print statement. For instance this shows that a string of digits such as those from 10000 to 1000000 within the decimal expansion of $\pi$ are not really random, because this very sentence can be viewed as a recipe from which they could be reconstructed. And if desired the sentence could be preceeded by a program for calculating digits of $\pi$. So the fact that the string of digits that starts off 85667227966198867... which may have no pattern discernable to the casual eye could be spotted by Kolmogorov compression as being extracted from the list of digits of $\pi$.

So suppose you have a sequence of values that are supposed to be truly random – such as numbers drawn each week in some lottery – the though here is that if somehow you could find a scheme that could generate exctly that sequence where the description of your scheme is notably shorter tham merely listing the numbers themselves then you have good reason to believe that they were not random after all. At least in some sense all the techniques used to try to assess procedures that generate pseudo-random numbers are modelled on this in that they look for patterns that represent traces left by the mechanisms actually used to generate the numbers. But Kolmogorov compression would uncover the implementation of the generating program and in so doing show that the sequences were pseudo-random rather than] genuine.

### 5.6.2 How common are primes?

The result sketched here uses the fact that almost all data must be impossible to compress. This result emerges from counting how many different possible byte-sequences of length $N$ exist and concluding that any loss-free compression can only possibly map that number of longer strings onto them. So all the other longer strings can not be squashed. This line of reasoning does not indicate exactly just which strigs will end up being mapped onto shorter ones and which will not! But good practical compressions methods try to arrange that the ones that are allocated short representations are ones more liable to arise in practise.

Using this principle that most data is not compressible it is possible to derive a result that shows that primes must be reasonably common. This result is not going to be anything like as good as the ones that “proper number theory" comes up with, but the fact that it uses data compression as its main tool is perhaps witty!

So consider a really huge number that is written using $N$ digits. The principle that most data can not be compressed says that in general it will not be possible to find a way to describe it using significanly fewer than $N$ digits. Well one alternate way of specifying the number would be to list its prime factors, so for instance the integer 856672 would be presented as $2 \times 2 \times 2 \times 2 \times 2 \times 19 \times 1409$. and now we imagine numbering the primes, so 2 is the first, 3 is the second and so on, Then this representation of 856672 might be written as $(1, 1, 1, 1, 1, 8, 223)$ because 19 is the 8th prime and 1409 turns out to be the 223rd.

Now if you imagine that primes are really very uncommon then the index of a prime (eg 233) will be very much less than the value of that prime (eg 1409) and so writing out the index will use less space. That could potentially give a way of finding a way to create a compressed representation of the number, and while that can happen occasionally that has to be rare. This establishes that primes can not be hugely rare.

Rather that attempting to get the sharpest possible result out of this, pretent for a moment that the density of primes was such that the $n$th prime had value comparable with $n^2$ which would mean that its simply written value used about twice as many digits as were needed for writing out its index. Then at least a simplistic reading is that the compressed representation of a number a slist of the indices of its factors will use half the number of digits plus a number of commas based on how many factors are present. for big enough numbers this will pretty well always be significantly shorter than the representation that memly writes out the number simply, and this being impossible establishes that primes must be more common that had been assumed.

This illustrated that even without being able to exhibit perfect compression a consideration of it can be used as a basis for reasoning about probability distributions and the like. But it remains the fact that optimal compression is not merely galacticm it is uncomputable.

# Chapter 6

# Intractable problems

*The chapter about galactic algorithms looked at extraordinarily laborious ways of solving problems, but various of the problems considered could be resolved in sensible amounts of time in practical-sized cases using an alternative method which would lose out to the galactic algorithm in the long run, but only the very very very long run. This chapter considers problems where in some cases all we can say is that the current best solution known is obnoxiously time consuming and in others where it is possible to prove that an algorithm, including all the ones not yet invented, will be infeasibly slow.*

# Chapter 7

# Practical Performance

*Sometimes if you attack a task in a simple or obvious way that will turn out not to be as efficient as you would have liked. Other chapters here have focussed on mathematical or theoretical limits to performance, often with a total disregard for reality. Practical computer systems can need to seek enlightenment from the theoreticians, but then seek solutions that work best in real world circumstances. Where they get to may change as new computer hardware becomes available, and the tricks they use may not always please purists, but delivering the best in a realistic context is far from easy and deserves celebration.*

Galactical algorithms is all about pushing theory and assymptotic performance estimates to the limit regardless of whether the result would be practical. So a scheme that ran in $10^{100}N$ when faced with a problem of size $N$ would be prefereed to one that needed $0.001N^{1.0001}$ because for large enough $N$ it would be the winner. This chapter considers the other end of a spectrum and considers cases where absolute timings measured in minutes and seconds or space use in bytes is to be optimized, but in the pursuit of perfection all other constraints and limitations are ignored. This can certainly involve use of significantly messy and elaborate code to achieve even small improvements over straightforward implementation.

## 7.1 Abandoning portability

## 7.2 Special hardware support

## 7.3 Composite methods

## 7.4 Dynamic code generation

# Chapter 8

# If you See What I Mean

*Elsewhere you will find attempts to get to the root of what computing is really about by focussing on either data storage (Turing machines) or the flow of control as you perform a computation (Counting machines). This chapter can be seen as yat anther way to look at things by considering "functions" as primitive. It lends itself to presentation in the form of reasonably accessible (family of) programming languages where the most limited of those has nothing except functions to work with but higher up varieties can be genuinly practical and convenient to work with.*

Turing machines provide a model for computation that (perhaps unexpectedly) can give really good insight into costs. But setting up programs for them is not really anything like conventional programming and they do not adapt well to day to day use. Minsky or Counter machines have their behaviour expressed by flowcharts and come much closer to being amenable to casual use, but they exhibit totally absurd run-times that render them quite unsuitable for anything other than theoretical purposes. That leaves a gap in the market to be filled by a formal model of computation that can be reasonably easily adapted for use as a practical programming system. A key candidate for this is called "lambda calculus".

The fundamental idea here ia that functions should be seen as the basic building block for everything else. When first introduced this was accepted as a neat theoretical idea and it led to what I will impolitely describe as niche programming languages, but over recent years it has been recognised as such a good idea that basically all modern programming languages incorporate features that are directly derived from it. A *lambda expression* denotes a function and is seen as freestanding entity. For instance in earlier usage one could only introduce a function by a notation that gave it a name, as in

```
    f(x) = 3*x + 1
```

as a definition of a function called `f`. With lambda calculus the name f is not needed and we can write just this function as

```
    λ x . 3*x + 1
```

where the name just after the $\lambda$ indicates what the formal argument is and the section after the dot is the body of the function. About the only action that can apply to a lambda expression is that it can be applied to an argument, and the consequence of that is just about the same as when the previous function called `f` is applies to an argument. It is almost as simple as that!

Well there are two issues that need discussion that arise when several lambda expressions arise together. The first rule is just to give a pedantic confirmation that things happen the way you would naturally hope, and it insists that when the body of a function has a value for the function's formal parameter substituted in nothing improper happens. One version of it starts by noting that in any lambda expression one could rename the variable that is bound to anything else and so the example shown above is to be viewed as entirely equivalent to

```
    λ y . 3*y + 1
```

and the mapping from one text string to the other is known as alpha conversion. Now if you applied out first lambda expression to an argument that was the literal symbol `x` rather than a number there would be a potential source of confusion about substituting for `x` in the body. All possibiity of uncertainty can be overcome if one used alpha conversion on the function just before applying it to make its formal parameter a new symbol that could not conflict with anything. Another part of what is essentially the same issue arises when the body of a lambda expression includes further nested lambdas and variable names are reused. It is proper to apply scoping rules so that inner lambda bindings take priority over outer ones, but even here use of enthusiastic alpha conversion can keep all names separate and avoid risk.

The second issue is that if one has a reasonably large nest of lambdas (and some examples will arise later) it could be that there are several places in the whole where a lambda could be applied to an argument. When there are several options for the order to do these operations does it matter which is selected and if so what is the proper strategy? On this there is some divergence of opinion! From a theoretical point of view it is considered proper that where their is a choice one should select the leftmost outermost application. This is known as normal order reduction and if there is any chance at all of continued application of lambda expressions to arguments ever terminating with a form where there are not more available than this guarantees to reach that state. The alternative is that innermost lambdas are processed first.

This can sometimes risk unnecessary exploding or looping sequnces of operations, but is generally seen as easier and cheaper to implement. So many but not all programming lamngauges with lambda support follow the second path.

It is perfectly feasible to write out everything that is to be discussed in this chapter using loads of $\lambda$ symbols, But for many people it will be easier to work with a notation that feels more familiar (but that can be expanded directly into the raw lambdas). A paper by P. J. Landin[?] has as its title "The Next 700 Programming Languages" and introduced `ISWIM` (If You See What I Mean) as a sketch, and so rather than latching onto any particular real or modern notation that exposition used here will be based on that.

ISWIM starts by using a keyword **where** to give names to things. To be a little pedantic here are a couple of examples and how that would expand onto raw lambda expressions:

```
3*x+2  where x = a + b       : (λ x . 3*x+2) (a + b)
f(a+b)  where f(x) = 3*x + 2 : (λ f . f(a + b)) (λ x . 3*x+2)
```

The hope here is obviously that the ISWIM versions are un-threatening and so will make things easy to understand. As well as **where** that puts a definition after the use of something it will also be possible to use **let** to put a definition first, and here we will often use that with the intent that the definition provided should be available to all subsequent input. What is more likely to cause some concern to start with is that in this world functions are first class citizens with all the rights and privileges entailed. In particular they can be passed as arguments and returned as values. A major lesson that lambda-calculus teaches is that this is liberating an powerful and so it is used a very great deal.

The ISWIM examples above use function definitions and **where** clauses to provide the structure of a program, and then have numbers and arithmetic operations. As a prototype for 700 variant languages ISWIM can imagine a wide variety of built-in data types and functions that can apply to them, so that it can be tuned for particular application areas. I will allow for the creation of user-specified data types. But it tends to de-emphasise sequential programming, with its documentation noting that in Algol 60 it was already possible to avoid that quite a lot by virtue of allowing conditionals as expressions as well as statements. It can view a sequence of several values as in `A, B, C` to be viewed as a single item, and that item could be returned as the value of a conditional expression or passed to provide three arguments for a function that needed that many. The variant or variants of it used here will be ones tuned to be good for describing things that can be done with lambda expressions!

From the tiny examples shown above it is possible to push explanations either up or down in terms of the level of abstraction involved. Here we will first show that lambda calculus using nothing more than function definitions and even restricting those to the one-argument case is in fact capable of modelling all computation. And that amazingly the steps it needs to take to do this are all remarkably concise and may even not be terribly inefficient. After that it will be explained how lambda expressions can act as a convenient and well defined intermediate stage to be used by compilers while mapping from source code down to some machine-specific level. And When that has been done it is easy to observe that any programs in the languages that those compilers support can be rendered as lambda expressions. A final section will comment on a somewhat archaic but thoroughtly practical language that was explicitly built to illustrate this but that has been used to build large and complicated programs.

To build up an understanding of computation from lambda calculus we start by noting that a lambda expression only has a single argument. In much of what follows it is going to be convenient to think in terms of functions with two or more arguments. These can be rendered using a technique called "currying" after H. B. Curry[**?**] who popularised it. We will write function definitions as if they used several arguments as in

**let** `f a b c = a+b+c+2`

using spaces between the multiple parameters rather than enclosing the collection of them in parentheses. We write in parentheses to show how we intend or expect to view this as grouped. In this case both the description of the pattern for the function (i.e. the bit to the left of the `=`) and the additions in the body associate to the left. Then there is a bit of rather dubious renaming and a use of (where) and tha name `F` that is really not quite proper but that turns the main definition into one with a single argument which is then easy to transform into a lambda.

**let** `((f a) b) c = ((a+b)+c)+2`

$\Longrightarrow$

**let** `F c = a+b+c+2`
**where** `F = (f a) b`

$\Longrightarrow$

**let** `(f a) b =` $\lambda$ `c . a+b+c+2`

$\Longrightarrow$

**let** `f =` $\lambda$ `a .` $\lambda$ `b .` $\lambda$ `c . a+c+c+2`

Even if the last line might look slightly awkward to start with it shows that the ISWIM-style notation that defines a function that in a certain sense has three arguments does correspond to respecable lambda calculus.

Given that the first thing to be done is to show how easy it is to provide truth values (i.e. *true* annd *false*) and a conditional that tests them. Annd instantly from that **and**, **or** and **not**:

```
let true a b = a
let false a b = b
let if p X Y = p X Y
let and a b = a b false
let or a b = a true b
let not a = a false true
```

These can also look a bit strange to start with, but observe how very concise they all are and then just verify them by writing down combinations of them and using the funcction definitions as rules to check how things behave. However there is one big caveat in all this – it only works nicely provided you use normal order reduction, because that manages to arrange that for instance when you write `if p X Y` that the evaluation of `X` and `Y` gets deferred until it can be determined that only one actually needs to be processed. With the evaluation strategy that deals with innermost expressions first both `X` and `Y` would get expanded there and often that would lead to a loop or other disaster.

Note that for the above to behave the value `p` tested by `if` must evaluate to either `true` or `false`. In a bit there will be a discussion about how that might be enforced.

Given truth values and conditionals the next feature of "real programming" to consider will be data structures. A useful building block from which pretty well anything else can be constructed will be the "ordered pair", where we will have a function `pair` that creates on and then `left` and `right` that retrieve one or other of the two included items. Again the lambda forms that implement this are amazingly concise and you just have to try following through what happens when you use them to convince yourself that for instance `left (pair A B)` will recover `A`:

```
let pair A B f = f A B
let left P = P true
let right P = P false
```

There are two things that can be done instantly once you have pairs. The first is to use clusters of them to build triples, 4-tuples and larger structures with any fixed number of components. The second is to pair an object with a tag (which might be a truth value) so that as a while it can have two (or more) possible shapes. As an illustration of this consider a way of handling lists that can have and number of elements (including zero):

```
let emptylist = pair false <anything>
let node A B = pair true (pair A B)
let isempty L = not left P
let first L = if (isempty L) <error case> (left (right L))
let rest L = if (isempty L) <error case> (right (right L))
```

which exposes the fact that dynamic structures like this have to carry extra information so that it is possible to detect when the end of a list is reached. A similar scheme could describe trees. The sections in the above that are written in angle brackets are parts that ought never to be used so anybody confident that their code would not contain any errors could use absolutely anything to fill those gaps.

Next comes a representation of numbers and arithmetic. Well one scheme that could handle positive integers tolerably well would be to express them all in binary and use a list of the bits, with `true` used for `1` and `false` for `0`. With that representation arithmetic operations would have costs driven by the bit-length of the numbers, and really that is a proper cost even though we tand to be used to thinking of arithmetric as having unit cost per operation. This modelling comes much closer to revealing how many electronic components will be activated in performing an operation.

But coding up binary arithmetic is a bit tedious, so it is fun to show that it is seriously easy to make pure lambda calculus explain the behaviour of integer arithmetic[1] really neatly. Rather than thinking in terms of zero, one, two, three and so on it is good here to thing in terms of never, once, twice, thrice and the like, and a "number" can then be a function that applies a second function multiple times:

```
let never f a = a
let once f a = f a
let twice f a = f (f a)
let thrice f a = f (f (f x))
```

and this pettern allws you to have a function corresponding to any number of copies of `f` you could want to set up. With this representation of numbers a scheme to do basic arithmetic is almost frightendingly simple!

```
let increment n f a = n f (f a)
```

[1]To be more pedantic natural number arithmetic, i.e. only using positive values

```
let add m n f a = m f (n f a)
let multiply m n f a = m (n f) a
let power m n f a = n m f a
```

and also a test for zero is remarkably easy to arrange

```
let iszero n = n f true where f x = false
```

Again one can appreciate how compact all the above are even without (at first) understanding how they work. And to get insight into the working it is possible to start with simple expressions such as `power twice thrice f x` and follow through the definitions to observe it expanding until it ends up as (in this case) `f(f(f(f(f(f(f(f(a))))))))` which used `f` $8 = 2^2$ times. Or `iszero twice = twice f true = f(f true) = false`

There is one thing to be done here that is slightly more tricky, and one issue that has so far been swept under the carpet. The tricky issue is a function `decrement` to find the predecessor of a n. It should be expected that this will not be quite so straightforward because `never` does not have a proper predecessor. There are however several ways of managing this. The one shown here is not the standard one but is perhaps the most compact – but it relies on the precise way in which `increment` has been coded and it achieves its result by handing that an argument that is rather improperly not a number but that has been chosen so that the resuly returned is `never`. See how that can be done, starting with a letter X standing for what we will invent:

```
increment X f a = never
increment X f a = X f (f a) = never
let X a b = never
```

and the three lines above that intoduce a function `X` that in just this one contexe serves as $-1$ we end up with an implementation

```
let decrement n f a = n f X a where X a b = never
```

Concise it possibly confusing[2]

Any reasonable programmer will see that given decrement it will now be pretty trivial to code up subtraction and then division. If negative numbers are needed that can be handled using a sign-and magnitude representation by using `pair` to glue an explicit sign only every number.

[2]A scheme that does not cheat by passing things to the increment function that are not numbers works by having a function that accepts an ordered pair as its argument: `let inc (a,b) = (a+1,a)`. If this is applied $n$ times to a starting value (0,0) the result is $(n, n-1)$ so the second component is the predecessor of $n$

Now for the issue that had been avoided that far (but when you look back at all the fragments of code to date it does not impact them). In raw lambda calculus the functions do not have names. That means there is no obvious way for the definition of a function to refer back to the function itself. In ISWIM you could have felt tempted to write things rather like

```
let factorial n = if n=1 then 1 else n*factorial(n-1)
```

as a simple recursive definition. On expanding that to lambda calculus the use of the word `factorial` within the body of the function will be improper and it certainly has no simple way to relate to the name you happen to be using for the function. So it looks as if recursion is going to be a problem! Happily this can be resolved. Often the way of sorting it is thought of as a bit strange. The presentation here is based on thinking of recursive definitions as just a notation for programs of infinite length that unwind it:

```
let f1 n = if n=1 then 1 else n*f2(n-1)
  where f2 n = if n=1 then 1 else n*f3(n-1)
    where f3 n = if n=1 then 1 else n*f4(n-1)
      where f4 n = if n=1 then 1 else n*f5(n-1)
        ...
```

and provided that chain of nested definitions is continued to a depth that is at least as great as the value of an argument $n$ we are about to use this will do just what is needed. This sort of expansion can be applied to any recursively defined function[3].

The magic that makes it possible to create unboundedly nested things in raw lambda calculus is a function traditionally called `Y`. Its behaviour can be characterised with the identity

```
 Y f = f (Y f) = f(f(f(...
```

but because this is an explicitly recursive presentation it can not be used directly. However there are many ways in which ISWIM or lambda calculus can provide a `Y` function without needing recursion, and the simplest is

```
let Y f = g g where g h = f (h h)
```

In ISWIM notation this is really concise and rather easy to understand! Just simply following the steps we have `Y f = g g = f (g g) = f (Y f)`. ands the ISWIM definition did not mention `Y` within the right hand size of its definition so it translated into 100% valid lambda calculus.

[3]ISWIM insists on the use of `let rec` when a recursive function is being defined to highlight that something special is going on.

The general use of `Y` is that recursive function definitions can be expanded using the following rule:

```
  let f x = ... f x' ...
```
$\Longrightarrow$
```
  let f = Y G
    where G f =
      F where F x = ... f x' ...
```

and the key thing here is that the recursive reference to `f` now has a meaning because it is talking about the argument to G, and if you trace through expanding the definitions out a few times you can see that this ends up being exactly what `f` was to end up as.

There really only one further issue that needs to be addressed – having multiple functions which are mutually recursive. In fact this is rather easy to cope with. It will be sufficiant to explain what to do with just two such, since larger numbers can just use the same technology "written large".

If there are a two functions say `f` and `g` echo of which depends on the other one just considers putting them as the two components of a `pair`. Then the definition in ISWIM form might look something like

```
  let P = (pair F G) where f = first P and g = second P
```

where `F` and `G` are the two definitions written as lambda expressions so that there are no indications of arguments on the left hand side. This is now merely a definition of P in terms of itself and the expansion to use `Y` shown before can be used to turn it into a respactable lambda form.

We now have boolean logic, data structures, arithmetic and recursion and with those it is pretty easy to code up anything else that can be imagined[4]. And the mapppings from them down onto the extraordinarily simple basic rules of lambda calculus are concise and in general not too unpleasant to work with, so if anybody *really* wants to reason about the behaviour of code starting from first principles it can be reasonable to select lambda expressions and their behaviour as those fundamental principles. But now numbers and all the rest can (when necessary) be explained in this rather gory manner it is perhaps reasonable to consider a version of ISWIM that has all such features as built in primitives that can be implemented using traditional computer hardware and reasoned about at their own leval of abstraction. The fundamentalist view of lambda calculus then just sits in the background providing assurance that the foundations for all that is built are really secure.

[4]Well interactive input and output, file-system access and the like are not parts of the world of concern just here!

Even if some of the code fragments above look unusual it will be easy to see that all are rather concise and none hide particularly terrible costs. Well modelling the integers in this simple way will make them slow but for reasoning about things and proofs that is not a real concern, while for realistic execution of code one could either use lists that provided binary arithmetic or more plausibly build in genuine integers as a primitive type. But the gap between a really fundamental and primitive model (i.e. lambda calculus with nothiing beyond functions to work with) and a world in which computing could be realistic is rather small.

Well having looked downwards from practical computing to see how it maps onto a nice theoretical model the next thing to do is to look sideways. This will be a consideration of the use of lambda expressions as part of a compiler. Specifically the idea is to use lambda expressions or at least something very much like them as an intermediate code that the user's original programs are coverted into before things get mapped down as far as machine code. Having a clean intermediate form is generally valuable because it is liable to be simpler to analyse then the original complicated language things started off in, and transformations and optimisations done there can be independent of the precise machine that will eventually be targetted.

For the discussion here it will again be convenient to use a version of ISWIM notation to show the lambda-calculus stuff. It will not make sense to cover every single part of the expansion from C++ or any other high level language[5] but representative important challenges can be covered.

The first thing to note is that this far ISWIM has not included assignment statments or `while` loops – far less `goto` statements. But a "real" programming language might! A feature of such a language will be that programs are made up of sequences of statements that are usually obeyed one after the other. If one is using the lambda calculus as an intermediate representation for code within a compiler than it will not matter too much of that version looks a bit ugly! Given that attitude we can take each line of the original user code and make it into a function all on its own. All the variables the user has will be passed as arguments to this. The function we have just invokes the one that corresponds to the following line with arguments adjusted to capture changes made to the values of variable. Suppose we had some C++ code that read

```
<start here>
int r = 0;
for (i=0; i<10; i++)
  r = r + i;
```

[5]Some people would object to calling C++ high level.

```
  <end here>
```

this might expand into something along the lines of

```
  let start () = f1 0   % call f1 with r=0
  let f1 r = f2 r 0     % introduce i starting at 0
  let f2 r i = if i < 10 then f3 r i else f4 r
  let f3 r i = f2 (r+i) (i+1) % the loop body
  let f4 r = ...
```

This is perhaps suprisingly not that dramatically bulkier that the code that it started with, but being in lambda calculus is in a form where reasoning about it and transforming it is liable to be safer than trying to work with the original.

Variations on the above cope (fairly) gracefully with many local features of programs. Function calls (and perhaps especially the interaction between them and exception handling raise further challenges. Before explaining how lambda calculus can help it will be useful to review how compilers tend to map procedure calls onto sequences machine code instructions.

I many cases arguments to a procedure will be passed in machine registers, so to call a function the compiler will arrange to have argument values loaded into the registers that they will belong in. It then needs to do the procedure call itself. That involves saving a return address somewhere and then jumping to the procedure entrypoint. The code for the procedure they knows where to find its arguments. It does what it needs to and puts its result into another agreed register. It then recovers the return address that it had been provided with and jumps to the instruction that is identified. This explanation has skipped discussion of stacks and of arranging that the procedure does not corrupt registers or data that the caller needed to have preserved, but that needs to be arranged as well.

Well an insight that emerged from lambda-calculus enthusiasts was that the return address could be viewed as nothing different from just an additional argument. It is a value put in a register before the procedure is entered. And similary the return step looks very much the same as the process of viewing that return value as the entry point of a function and calling it giving it an argument that was your “return value”. Thiw view feels fairly natural if you are ready to accept functions as first class objects rather than as something seriously different from numbers and arrays. So now we explain the code by saying that to call a function you enter it having placed in agreed locations all of its arguments together with a function that you call a “continuation” (and that you used to call a “return address” and view as something much more special). Then when the function has done its stuff it just invokes its continuation. Well an odd consequence of this is that nothing ever truly returns – all code works by chaining from one continuation to another, but given that this is merely a semantically clean way of describing what was going to happen anyway there is not need for it to be expensive. And because it means that argument registers and return address ones are not now treated as separate sorts of thing it may simplify optimisation.

A further great joy of this is that it provides a way to explain exceptions and `catch/throw` behaviour. A `catch` just sets up a continuation that would proceed into the part of its code that responds to an exception. This is put somewhere same or passed along to functions that are called. If a `thow` has to be performed all that happens is that this alternative continuation is triggered and the one that would normally have applied just gets ignored. What a neat and elegant way to explain something that previously might have seemed a bit messy!

The final section in this chapter is an explanation that the roots and the legacy of ISWIM and of lambda calculus permeate quite a lot of the world of computers at the practical level. Historically the language LISP[**?**] had lambda calculus as one of its explicit inspirations, and over the years many serious bodies of code were implemented in it. Those includes the operating systems for special purpose computers that were in their day prime ones for development of artificial intelligence code and packages that can perform symbolic algebra in ways that can help students, scientists and engineers. More recently SML[**?**] (used to develop heavy duty theorem provers), Haskell[**?**] (a language that features normal order reduction) and OCaml[**?**] can all be seen as having lambda calculus at their core and as being among the 700 languages that the ISWIM paper talked about. So this is an topic that spans from foundational theory all the way up to large scale practical application, but with the gulf between those two much less than is observed with other theoretical models.

# Chapter 9

# Primitive Recursion

*Turing machines etc are HORRID because you can not solve the halting problem, and that means in effect that formal justification of stuff is too hard. So is there a model of computation that is powerful enough to be useful, straightfoward enough to leave at least some prospect for analysis b ut that is tame enough that the biggest blockages of Turing machines are not included. Step forward "primitive recursion". Its key features are that any computation expressed using it is guaranteed to terminate (possibly after a time so long that you will have given up waiting), And it can express a remarkable proportion of the problems you might set a computer.*

# Chapter 10

# Solving a quadratic equation

*Enthusiastic educators can set tasks in introductory computing couses that give their students a taste of program creation but are not too compilicated or difficult. Until you demand that the answers really appoach perfection. The example given here is just to give a taste of this!*

There are introductions to programming that give as one of their earliest examples the challange of creating an application that reads in three numbers, $a$, $b$ and $c$ and then prints out the two solutions to the equation $ax^2+bx+c=0$. The clear expectation is that this will be done using the well-known formula $\frac{-b\pm\sqrt{b^2-4ac)}}{2a}$. When done by just writing out use of that formula the task is indeed easy enough to be in an introductory section about use of computers.

However if one looks at the task more carefully and start to insist on getting as correct a result as possible for all legitimate inputs things become rather different. I will suppose that since this is a task set for beginners it is to be solved using the default way in which your computer handles numbers. In almost all cases this will follow an international standard called IEE 754 abd the program that is written will use a representation they call "binary64" which is commonly referrred to as "double precision". The task in hand here highlights several ways in which the computer arithmetic that is thereby provided can do things that a naive user might not havr thought about. So let's take these in turn and see how the equation solver will need to be made more elaborate to avoid them hurting.

The (IEEE) floating point representation in a computer can not handle arbitrarily large numbers. Specifically it runs out of steam when values exceed around $1.8\times10^{308}$. Beyond that it stores values that represent "infinity". Now of course sensible people will not tend to work with problems that lead to numbers so absurdly large, and so it is common not to think much about it! However if one seeks perfection then the quadratic solver should deliver accurate results for any inputs where the results are sensible, and it should report failure in exactly the cases where the inputs do not lead to answers that can be represented in the floating point format that the computer uses.

So now consider the case when some user provides the input $a = 1, b = 1, c = -1$. Substituting these into the formula leads to a pair of very sensible roots with values about 0.618034 and $-1.618034$. Everything seems good. But now a rather less helpful user enthusiastically offers $a = b = 2 \times 10^{154}$ and $c = -2 \times 10^{154}$. This fairly obviously has exactly the same two roots. However the with this and with various fairly closely related cases involving huge numbers it is possible to arrange that in the computation of $b^2 - 4ac$ that either $b^2$ overflow or $4ac$ overflow or both, or that neither overflow but their difference does. It is furthemore possible for one or both of the terms to exceed the $1.8 \times 10^{308}$ maximum and hence overflow even though when they are combined the result is in range. This is all a mess! However this particular case can be tidied up by starting the calculation by dividing all of $a$, $b$ and $c$ by some large value so as to reduce them to sensible size numbers. At which point an additional issue comes into play. If you divide a floating point number by a general scale factor doing so can introduce rounding errors. Consider for instance the calculation of just 1/3 and the computer will produce something in the style of 0.3333333333333333[1] which is not precisely the same since it terminates after a finite number of digits. Continuing the calculation with this corrupted value will naturally lead to a (small) error in the final result. Happily with IEEE floating point it is legitimate to divide by any power of 2 and in that case no precision will be lost. So it will be necessary to identify a power of 2 that is about the right size so that it can be used to rescale the input to avoid overflow.

The situation with very very small numbers is in fact even more curious although the way to sort things out is basically the same. The smallest floating point value that can exist (with smaller values being flushed to zero) is about $4.94 \times 10^{-324^2}$ but once values get lower than $2.23 \times 19^{-308}$ the representation starts to work with them an lower than normal precision. So to preserve as much accuracy as possible the scaling has to keep things away from not the point where underflow maps values to zero, but from some rather earlier-encountered threshold.

For now and to prevent things geting quite out of hand the issues of rounding errors that might arise when multiplying $b$ by itself and so on up to

[1] Since it is working in binary internally it will actually be more like 0.01010101...

[2] $2^{-1917}$

and including those that could be introduced by the square root calculation are going to be ignored, but anybody who really wants to push for perfection regardless of the cost in difficulty can explore those paths. Also it would be proper to review cases where the true results are going to be very close to or beyond the overflow or underflow points so that good answers are produced in every case where that is possible. Part of that might involve finding a scale factor $\sigma$ and replacing $a$ with $\sigma a$ and $c$ with $c/\sigma$[3]. Then a final result can be generated by multiplying or dividing the solutions to the scaled version by $\sigma$, and any overflow or underflow will then be captured in and limited to that final re-scaling operation.

However there is a further and distinct way in which the naive use of the standard formula can generate seriously inaccurate results even after overflow is avoided and regardless of minor rounding errors. Supose that the value of $-b$ and $\sqrt{b^2-4ac}$ are rather similar in absolute value. This easily gappens when $b$ is rather larger than either $a$ or $c$. Then one of their sum and their difference will add two similar values and give a good result, while the other can lead to massive loss of precision as leading digits match and cancel out. To illustrate this consider the use of 8-digit decimal floating point on a pocket calculation and look at the equation $x^2-4003*x-1=0$. Here $-b$ is obviously just 4003.0000 when written so that the 8 digits of precison are made explicit. A careful calculation of $\sqrt{b^2-4ac}$ gives its value as 4003.000499625249859... and to 8 digits that is 4003.0005. When the computer subtracts these it has no access to ant of the lower digits so the only result it can offer is $-0.0005$ while the ideal result would have been $-0.00049962524$, So although the arithmetic had been being performed keeping 8 significant digits thoughout the cancellation of leading digits in the subtration means that our final result has at best only 4 of its leading digits correct. Using a number larger than 4003 would make this precision loss worse to an extent that even with full 64-bit IEEE arithmetic results can be unsatisfactory.

While we can all accept that the example shown here might have been chosen with awkward numbers deliberately picked to give this severe cancellation of loading digits, there is no fundamental reason why such cases might not arise in real life and in fact it can occur so easily that it should be considered a common risk.

Happily (for those who like to deliver accurate anwers) or painfully (for those who see this adding yet an extra layer of complication and difficulty to the task) it is possible to avoid this calamity, because it is known that the product of the two roots of a quadratic will always be equal to $c/a$ and bacause whenever calculating one of the two roots does a subtraction that can

[3]with $\sigma$ a power of 2 to avoid introducing corruption through rounding

lead to leading digit cancellation the other will be found by doing an addition that gives full precision results. So in effect one should calculate the more delicate root as $2c/(-b - \sqrt{b^2 - 4ac}))$ which will now be very respectable. OF course it will be necessary to judge which of the plus or minus cases is the one to use and which the one to avoid.

So overall the program that solves a simple quadratic and that does not even worry about cases where there is no (real) solution is dramatically messier and calla for much more understanding that those novice programmers will have been ready to deploy. But if you are writing a library or application that is to be used by others you have a responsibility to cope with all cases, not just the ones you think about first!

# Chapter 11

# Turtle Graphics

*One of the (apparently) easiest introductions to computational thinking is Turtle Graphics, which has been introduces to children as young as 4. Is it really that simple, or might it let one raise questions that are a bit more difficult?*

One of the schemes often proposed for getting the very young into computation involved letting them draw pictures using a "turtle". This moves around the world leaving a trail that shows where it has been (so why is it not described as a snail?). It is possible to instruct it to move directly forward by some number of steps or to turn left or right by an angle that is usually specified in degrees. Combinations of these two operations can be repeated. So two very easy initial examples of what can be done are

```
  repeat 4 times
    move forward by 10
    turn left by 90 degrees
```

end

```
  repeat 5 times
    move forward by 10
    turn left by 90
    move forward by 10
    turn right by 90
```

where one of these draws a square and the other a zig-zag. By giving instructions that are less repetitive it will be possible to draw a house or other interesting outlines. It can be useful to be able to say "pen up" and "pen down" so that the drawing being produced does not have to use a single continuous line and then perhaps the turtle can be used to create any drawing then could be made using a pencil. That sets one path towards difficulty: set out the instructions for a turtle to approximate some of the pencil work from Leonardo da Vinci or Albricht Durer! That would probably just ends up as a hugely long list of movements and although the output would be spectacular the text of the sequence of instructions to the turtle would not be very interesting or informative! So here we will concentrate on examples where the sequence of instructions is reasonably tidy. It was easy to understand what the square and zigzag scripts would lead to, but the point of this chapter is that with fairly harmless-looking extensions to the set of operations that a turtle can be asked to perform it gets remarkably harder to predict what will emerge or reason about it in detail. So what we provide here are a selection of more or less difficult questions and challenges ragarding turtle behaviour and we will not spoil them all by giving all the answers!

1. Start with something that is not too hard. For exactly what angles of turn will a sequence rather like the one that draws a square return to its starting point, and how many steps will that take? Angles do not need to be whole numbers of degrees.

2. If the turtle position is computed using computer arithmetic that is only precise to say around 16 or 17 significant figuures, for a pattern that would close up in an ideal world how far from joining up can it be in reality? What are the consequences if the turtle keeps following the same pattern of activity for a really long time? In other words why might it be that when I use a turtle to draw pentagons and try tracing all the way round many many times in some cases I keep following exactly the same path on each cycle, while in others I drift very very slowly away from where I started?

3. What rule will lead to the turtle following a nice spiral path, and how does it behave beyond the time it reaches the centre (of it ever does)?

4. Suppose that at each step the turtle moves forward by unit distance and then spins so that the next direction is utterly at random. That could include it keeping on in its original direction, totally backtracking or anything in between. After $N$ steps about how far from its starting point is it likely to be? Will it ever return to its starting point? Does anything change if you move to work in 3 rather than 2 dimensions?

5. As above, but the new random direction is limited, say to corespond to turning right by an angle uniformly chosen between 0 and 180 degrees? How much does this impact things as against the fully random turn?

6. Suppose the turtle has a home and it makes a random turn that is almost fully random but that has a rather small bias toward pointint it homewards. How big does this bias be to give it a good change of getting within a reasonable range of its island? This case can be interpreted as a reasonable first attempt to model real bird or animal long range migration skills by exploring just how much navigational precision they actually need. And the traces of movement of several turtles trying this out can make nice pictures!

7. For parameters $x$ and $N$ consider the turtle instructions:

```
a = b = c = 0
repeat N times
  a = a + x
  b = b + a
  c = c + b
  move forward by 1
  turn left by c
```

   This has introduced some arithmetic and is a generalisation of the challenge to understand what drawing will emerge from "move 1;turn 1;move 1;turn 2; move 1; turn 3;. . . " where the angle turned at each step grows. Note that turning by angles over 360° is perfectly respectable in that you just spin all the way around once (not having any overall effect!) and the turn by the specified angle less 360. The challenge here is to understand what values of $x$ lead to closed paths, how long the paths are before they join up (i.e. how large should $N$ be to make this neat), and what symmetries there will be in the picture created. For some values of $x$ one gets a 3-fold symmetry. Just what values of $x$ lead to that? Can one get a 5-fold symmetry ever? And why are the pictures so decorative?

# Chapter 12

# Reduce to range 0 to $\pi$

*Your calculator or computer can "obviously" not just add and subtract. It can compute square roots, logarithms and all the standard trigonometric functions. These feel as if they are built in and so can be taken for granted. But somebody will have needed to design the hardware or software that evaluates each of them. So if you are going to understand computation from top to bottom it makes sense to look into some of what is involved in that.*

When needing to compute a trigonometric function such as sine or cosine of some number (and when you are working in radians rather than degrees) it will normally be right to start by reducing the argument to by subtracting off multiples of $2\pi$. That is because these functions are periodic so altering the argument by any multiple of $2\pi$ does not change the value to be returned. It might in fact be better to reduce the range to $0 \ldots \pi/2$ or possibly $0\pi/4 \ldots \pi/4$ but the principles explained here apply in all such variations.

A naive way of doing the reduction would be something along the lines of

```
n = integer_part_of(x/(2*pi))
x = x - 2*n*pi
```

and if computer arithmetic was in perfact agreement with mathematical ideas of numbers this would be enough to do the job. However floating point numbers on a computer have limited precision. An immediate consequence of that is that the mathematical value of $\pi$ can not be represented exactly there. So computationally (as distict from mathematically) the subtraction of `2*n*pi` is going to subtract something that is not exactly what was ideally needed. A fine illustration of this arises when you simply ask the computer to show you the value of `sin(2*pi)`. To illustrate this imagine that the computer works to a precision of 4 digits in decimal (while double precision would be 52 binary digits which is messier to present here). Then the computer's idea of `pi` will be 3.142 precisely, and its model for `2*pi` will be 6.283. Nots for a start that the second of these is not just twice the first! The computer can not do any better within the 4-digit precision we are working with. If you calculate the true value of `sin` of that exact value and keep just 4 significant figures you do not get zero. You obtain $-0.0004073$. If the computer had started the calculation by subtracting its idea of `2*pi` and ended up getting zero it would have been wrong.

Some would try to claim that the point being made here is aa bit pedantic and a result of zero is what the user "expected" so would be better. Here the view is that a good implementation of elementary functions should treat input values as standing for exactly what they say (which will always be values with only a limited number of digits) and that results should match the mathematical result based on those values. And that anything else is sub-standard.

The issue has just been illustrated with a rather small argument value. It becomes much sharper if you consider large values. With 4-decimal floating point you could consider `sin(6.283*10^50)` where multiplying by a power of $10$[1] leaves the number still exact within its precision limit. Here the true result is $-0.9725$. The jolly thing is that this arises because `6.283e50` needs not $10^50*(2\pi)$ subtracting to get it reduced, but the implausible value $99997050744637839463589668526950773666250995418728*(2\pi)$. That is the integer part of $6.283e10/(2\pi)$ calculated precisely.

Looking at the above case it appears that to calculate accurate values of trig functions for large arguments (perhaps stupidly large arguments that sensible people do not use a lot – but perfectionists want to cope well with even awkward cases) is going to need serious multiple-precision arithmetic, as suggested by the need for 50 digits here. And for proper computer floating point the largest valid double precision float has value around $1.810^{308}$ and that starts to suggest that it may be necessary to work with precision equivalent to over 300 decimals. That would start to be expensive – a.k.a. "difficult".

The scheme that somewhat amazingly makes the calculation cheap is to stop trying to divide by $2\pi$ and instead multiply that by $1/(2\pi)$ where that value is

```
0.159154943091895335768883763372514362034459645740456 4...
```

So now we need to multiply that number by the big integer

[1]With binary computer floating point one would multiply by a power of 2 here

```
62830000000000000000000000000000000000000000000000
```

which is the proper way of viewing the number whose sine we are trying to find. Well firstly that is multiplying the 50-digit number by just 4 digits which starts to feel not too bad. But better yet the the result is going to be of the form 999 . . . 8728.7867 . . . where the bit before the decimal point is the huge integer multiple of $2\pi$ that was reported earlier. And we are just not interested in its value. So when you are doing the multiplication there is no point at all about forming partial products that will only add into that part. Also any partial products that contribute much further down that the four digits immediately after the decimal point are not relevant unless they lead to carries up into the important four digits. The result of that is that you can get the required digits of the fractional part of your input divided by $2\pi$ doing little more than a single length multiplication! The only cost is that youi need to have stored rather a lot of digits of the value of $2/\pi$.

The above shows that by doing something just slightly tricky it is possible to turn something that looked at first as if it would either be impossible, or at best that it would be grimly expensive, into a fairly short calculation.

But for perfectionists this is not the end of the difficulty! Note that in the imaginary 4-decimal precision arithmetic being used here to illustrated what is done[2] every floating point number bigger than `999.9` in fact describes a whole number. This is in the spirit that for instance $6.233e3$ has the integer value 6283 and the next floating point value up, i.e. $6.284e3$ is then 6284. That is a simple observation to make. The challenging question is “How close to an exact multiple of $2\pi$ can an a non-zero whole number less than $10^308$ be?” The reason this is important is that when we find, for some $x$, the fractional part of $x/(2\pi$ some number of digits immediately after the decimal point night be zeros. So in fact it will not be good enough as suggested by the previous paragraph to collect just four digits there. It is necessary to collect enough that there will be four *significant* digits following any leading zeros[3].

A useful observation is that $\pi$ is a transcendental number and that means that the remainder can never be zero, so we always have a finite number of leading zeros there, and if we limit the values of interest to the range up to $10^{308}$ (a value picked as an an approximation the the range supported by standard computer arithmetic) there must be a worst case. For almost all values of $x$ there are no or very few leading zeros, but the concern here has to be to get every single case correct and that involves seeking the most extreme behaviour.

[2]Using real 52 or 53-bit machine floating point does not add any significant extra issues

[3]. . . and equivalent cases when the fraction has many leading 9 digits

# Chapter 13

# Ingenious Data Structures

*If challenges to set up a computer system to perform some action or data transformation it can be natural to stary by asking "How would I do this by hand?". Often that woudl let you write a simple and direct program to do the job. However there are many many cases where more alaborate procedures will do a much better job. Especially when you then ask just how far can these improvements go it becomes possible to find amazingly ingenious and complicated ways to solve the original problem. Sometimes these even give the best practical results!*

Textbook of algorithms are often full of explanations of clever ways to do things that would not at first have been at all obvious. These have typically been invented because the more straightforward ways of solving the problems concerned can be improved on, often by huge amounts once you get to big enough test cases. So in this chapter a few of these schemes are presented as an ilustration of the way in which seeking the most efficient scheme can escalate difficulty quit a lot.

## 13.1 Binomial Heaps

A problem that can have plenty of real-world application is the maintainance of a priority queue. With such a queue anybody joining the queue has an associated weight or priority. The queue is processed by insisting that whenever the server finishes with one customer they next look after whichever one in the queue has highest priority. Clearly this scheme arranges that a new customer but very important customer can arrive and jump much of the queue. There are two main operations whose cost needs to be considered:

inserting a new customer into the queue and identifying and removing the next to be served.

The most naive scheme will be to keep all customers in the queue in a list arranged such that adding a new one has unit cost – and not at that stage worrying about the priorities. Then to pick the next one to serve it may be necessary to scan the whole of that list to identify the most worthy member of it, and excise them from the list regardless of whether they are its is head, tail or somewhere in the middle. That may have optimised adding new customers to the queue but it makes selecting the next one to be served have a cost proportionate to the queue length. If this was in fact the best that coule be achieved all would be simple, but a datastructure called a "heap" improves on it sharply making the cost of both queue activities proportional to the logarithm of the queue length. For long queues this gives a very good saving.

The explanation of heaps given here is not going to go into all the details and tricks that proper textbooks do because the main payoff of this section is something that builds on the general idea. So for now a heap is a structure arranged as a (binary) tree. Each node of the tree holds a customer and references to two sub-trees. Two key rules ar applied: the customer in each node is one who will have priority over every customer in either of the two sub-trees. And things are arranged so that the number of customers in each sub-tree are close to the same. The first of these means that server has immediate access to the most important customer – they are the one in the top node of the tree. The second ensures that the tree is nicely balanced and that if there are $N$ customers in all the the height of the tree is only $\log_2(N)$[1]. The management of such a heap is based on needing to be able to add new items to it while preserving its properties in time proportionate to its height, and equally being able to repair it when its top item is removed in a simmilarly fast way. Go and read the books to discover how that can be achieved, because the interest here is in making the problem slightly harder yet in a way that calls for further ingenuity.

Imagine you now have priority queues all sorted, and your setup now happens to have two servers each with their own separate queue. You are not allowed to make any assumption about which arrriving customers joined which queue. Now one server finishes their shift, and the remaining one is left to handle all the work. This immediatly calls for the two separate priority queues to be consolidated. One could do that by asking each customer who has been abandoned by the server they were waiting for to join the other

[1]Well that logarithm usually has a value that is not a whole number, so we need to round it up to get one!

queue one at at time, but since the queues we are now thinking about represent the queues as trees it is likely that this has a significant cost. The scheme sketched next reduces that almost as much as is possible!

Priority queues that may need to be merged can usefully be represented by a data structure known as a "Binomial Heap". All operations on such heaps have costs no worse that the logarithm of the number of items stored. The explanation here will start from the top level so that it can motivate the data structure used by showing why it is good.

The key clever idea here is that if we have any number $N$ we can look at how it would be written in binary notation and express it as the sum of a bunch of powers of 2. So for instance 39 is 100111 in binary and that means $39 = 2^0 + 2^2 + 2^2 + 2^5$. If the number is $N$ then we can saay that there are $\log N$ bits in its binary representation.

If a Binomial Heap (which is going to act as a priority queue) has $N$ items stored in it then they are arranged in a buunch of sub-heaps each of which has size that is a power of 2. It is not yet clear just how these will be represented, but it will be arranged that the higest priority item in each sub-heap is instantly accessible. That means that the top item in the whole heap can be found by checking each of the (up to) $\log N$ sub-heaps.

The clever part now emerges when you wish to consolidate two such heaps into one. The steps taken follow exactly the pattern used in performing addition on binary values. It can consider each possible power of 2 in turn. If neither input has a sub-heap that size then the output will not. If just one has then that will appear in the result. But if both do then those two sub-heaps get consolidated in a way that will be descrribed soon into a single one that corresponds to the next power of 2 up. In terms of the binary addition this is a "carry". Provided that the sub-heaps are kept with the $2^0$ one first and provided consolidating a pair of sub-heaps on size $2^j$ into a single one of size $2^{j+1}$ is cheap this manages to add (or pergaps we say form the union) of the two binomial heaps in logarithmic time.

Now what about that consolidation step? Well a good way to represent a sub-heap of size $2^j$ is to have the highest priority item in it picked out and sitting at the top, and the remaining $2^j - 1$ items kept in a list of smaller heaps of size $2^{j-1}, 2^{j-2}, \ldots 4, 2, 1$. Happily this satisfies our hope that the top item in the sub-heap would be easy to find, and it means that the double size sub-tree can be formed very easily by comparing the top items in the two trees to me merged and just pushing the smaller one onto the list held by the larger.

The task of removing the top item from a heap is equally straightforward. We alreadyu know we can identify which sub-heap had the desired element at its head. Remove that whole sub-heap from the top-level list. Now if you lop the top item from the bit of structure you have just retrieved you have a nice list of sub-heaps each of whose size is a power of two. Gosh that is just the shape of a geberal Binomial Heap and you can re-insert all its data into the main one using just the binary addition process already described.

Those who are properly pedantic will observe that in each of the various sub-heaps you will wany to have stored not just the top element and not just a reference to the list of sub-sub heaps, but something to explain how many items are present (i.e. which power of 2 is involved) and probably also a reference to the tail end of the chain of sub-sub-heaps so that tagging a new item on the end is really cheap. Doing all that carries some overhead but for cases with enough customers the savings by having costs that grow only logarithmically with the size of the queues is so much more valuable that it is not a big issue.

The tricks and the elaboration of data representation here may seem extreme enough that it would be natural to expect it was the best that could be achieved. But massochists can look in the bext chapter of their Big Book of Algorithms to learn about "Fibonacci Heaps" that are yet more bizarre – and which may only rather rarely be useful in practise because although from a theoretical analysis its costs grow slowly in pracise the overheads mean that simpler schemes test to win. But for anybody keen to see how difficult computing can be made looking at the them, and at the Brodal Queue and all other options for priority queue implementation can provide a fine collection of rabbit holes to dive into.

## 13.2 Continued Fractions

Any rational number $P/Q$ will have a decimal representation that either terminates or that after some starting digits continues into a repeating pattern. In contrast numbers such as $e$ and $\pi$ lead to non-repeating patterns of digits. There is another way of expressing numbers – continued fractions. This has the property that rational numbers always have a terminating representation, but values that are roots of quadratic equations (that have whole number coefficients) end up with repeating patterns. sectionBit counts Consider the three sequences

```
(1)   1 2 4 8 16 32 64 128 256 ...
(2)   3 5 6 9 10 12 17 18 20 24 ...
(3)   7 11 13 14 19 21 22 25 26 ...
```

where the first sequence is easy to recognize but the next two may at first be less obvious. However once there are displayed in binary things start to become clear:

```
(1)   1 10 100 1000 10000 100000 1000000 ...
(2)   11 101 110 1001 1010 1100 10001 10010 ...
(3)   111 1011 1101 1110 10011 10101 10110 ...
```

and it is now easy to see that each sequence collects the numbers whose binary representations have a certain number of 1 bits. Suppose that following one of these sequences (including the follow-on ones that cover 4, 5 and so on bits) can allow one to enumerate all the ways that $r$ things can be selected from a set of size $n$. Often in the mathematics of combinatorics the greatest concern will be how many ways of picking $r$ from $n$ there will be and looking at just what all the cases are is not terribly important – just jowning that a binomial coefficient lets us count is enough. But with computers it is entirely plausible that it will sometimes be useful to work through all the individual cases. This generating the above sequences and their friends might be useful.

Implementing code that will step through the values at first looks a bit messy, but amazingly if one mixes up oridinary arithmetic operations with `&`, `|` and `^` for bitwise and, or amd exclusive or it can be done an an amazingly concise way, as reported in Hakmem[**?**] and the Hacher's Delight[**?**].

```
[start off with a number in n]
w1 := n & -n;
w2 := n + w1;
n := (((w2 ^ n) >> 2) / w1) | w2;
[n is now the next value up with the same bit-count]
```

It is also sometimes useful to note that the division (which might be a bit slower than other basic arithmetic on some conputers) is dividing by a power of 2, and so when the computer provides a way to discover the position on the single bit that will be present in `w1` it will be possible to replace the division by a right shift (`>>`) of the bits involved. So a first challenge set here is to understand exactly why the 3-line program fragment there works! But then the real issue is going be to sort out just what else can be achieved if arithmetic and logical operations on machine binary numbers are combined. The Hacker's Delight has a range of examples where at least if remainder calculations are allowed demonstrate how counting the bits in a word, reversing them and other things are possible.

## 13.3 What next?

An INCOMPLETE section...

*The idea here is to go all the wey back to HAKMEM and report on performing arithmetic on continued fractions. Along the way II want to present the fact that many values depending on e have continued fractions that interleave an arithmetic progression with other simple patterns. Eg e has partial quotients 2,1,2,1,1,4,1,1,6,1,1,8,1,1,8,...*

# Chapter 14

# Images

*Well one can generate very pretty pictures related to eigenvalues of Bohemian matrices. These pictures must surely capture some useful facts about the matrices concerned! Can one discern it! Where else can a graphical presentation abstract up from some problem and make it possible to gain extra insight? This is an area I do not know a lot about!!!!*

# Chapter 15

# Simple Pattern Matching

*Almost every user interface provides some sort of "search" facility, and a powerful search needs to cover more than merely looking for fixed strings. It needs to seek text that matches some sort of user-specified pattern. A mimimum would be to allow wild-cards within the target to be sought. But across a wide range of computer applications there are variations on a form of pattern known as a "regular expression". Using those can feel easy and natural. But as is the case with some much in the computational world, these have more depth that is at first obvious.*

Sometimes you might want to search within some text but what you want to find is not just a fixed string. Perhaps it can allow options or repetition of sub-parts. Perhaps you want to put some sort of wild-cards into the pattern that is your target. There is a very well established scheme for setting up patterns for use in cases like this, and variations on it. Very many programming languages and even dialog-boxes in user interfaces use at least subsets of it. So for instance the pattern `*.jpeg` may be used to let you look for all files with the "`jpeg`" suffix, while at least in a Linux shell the pattern "`*.\{cpp,h\}` will match names that end in either `.cpp` or `.h`. A fuller scheme used for pattern matching as part of the language PERL and available through libraries in almost all other programming languages as a bit more formal. A pattern is built up starting with the very simplest: patterns that consist of and match just one letter[1]. These simple patterns are combined using three constructions, If P and Q are existing patterns then one can write

- *PQ* – this is a pattern that matches anything that can start with a sequence that matches P and follows that with one that matches Q.

[1]It can also in fact be useful to have a basic pattern that matches an empty string.

Obviously the very easiest use of this is that it means that you can write a sequence of individual letters and they form a word to be spotted;

- $P|Q$ – here we accept anything that matches either $P$ or $Q$. So for instance `cat|dog|rabbit` matches strings that name creatures suitable as pets, and `p(e|a)t` illustrates that it is sometimes useful to have parentheses to group things. This pattern will match either `pet` or how you might treat one, i.e. `pat`. If you need one of the characters `(`, `)` or `|` in the alphabet you are matching over some way of distinguishing raw characters from punctuation used to build up the pattern will be called for.

- $P*$ – This is the big one, It indicates an arbitrary repetition of the pattern $P$. So it is in effect equivalent to $(|P|PP|PPP|\ldots)$. Note there the initial option of no instances of $P$, i.e. of this matching the empty string. A really simple instance of this would be `B(an)*a` which matches `Ba, Bana, Banana, Bananana` and so on.

There are two viewpoints that can be taken about this. One is a practical one that adds a number of shorthand notations for things one might frequently want to do. A particular instance of this arises because these patterns (which are referred to as "regular expressions") provide an excellent way of characterising the ways in which tokens or symbols can be written in programming languages, and there are software tools that take a list of patterns and create a program that splits textual input up based on the. A first exxtension to notation that is used there is being able to give a name to a pattern fragment and then use it later. In the programs `lex` and `flex` one can name a fragment and then to refer to it you put the name in braces. You also enclose literal text in your pattern in double quotes. So for instance:

```
digit    "0"|"1"|"2"|"3"|"4"|"5"|"6"|"7"|"8"|"8"
number   {digit}{digit}*
```

gives a pattern for any (non-empty) string of digits and calls it "number". This example motivates two further expensions which clerly do not alter the range of patterns that can be expressed but that can make the presentation of the regular expressions concerned much more compact. Enclosing a collection of characters in square brackets and allowing character ranges is a help. If the opening square bracket is followed by ˆ then the expression is treated as if was a square bracket form enclosing all letters in your character set except the ones actually shown. WIth this the tabluation of digits becomes just `[0-9]`. The second expension allows for the fact that `*` can indicate zero or more uses of the pattern that preceeds it and sometimes as here you want at least one. Replacing the `*` with `+` does that. Hence you can now write

```
digit    [0-9]
number   {digit}+
```

Note that this could be textually expanded to the slightly clunky but basic for for regular expressions, so the extended notations are in general a matter of convenience rather than things that bring genuine new capabilities. And in that spirit here are a few more useful extensions that similarly do not alter the range of patterns that can be expressed but that may make it easier to specify them.

```
P & Q
~ P
.
```

The idea is that the first of these will match every pattern provided that both `P` and `Q` do, while the second matches any input that `P` would fail to accept. The single dot (`.`) will match any single character, and so of course `.*` matches either anything at all (including nothing).

While it is fairly straightforward to show that adding these constructs does not add any ability to match new sorts of pattern – all they do is make it easier and more flexible to specify them – the notes here are not going to explain the details there. Head for a suitable textbook if you need to know exactly how it can be done! But the typical places on your computer that provides support for regular expressions will typically not support these last two because in fact they unlock levels of practical difficulty that are hard to comprehend.

Thus far the use of regular expressions to provide patterns that you can try to match against input text seems really rather easy despite the pessimistic statement above. So here to give some insight into just why adding those two last capabilities is so bad let's state what problem it turns from tolerable into being solvable in principle and theory but utterly dreadful in practise.

The sections here will next work through showing that it is possible to set up a regular expression of reasonable size that such that the only strings it can match will be absurdly long. Once that is done it becomes possible to argue that while answering various questions about them and their relatives is theoretically posisble, the cost of doing so will be at least as great as the length of the shortest string they match, and hence is beyond all feasiblity now and for ever.

The way of doing this will be by using the term "ruler" to characterise an expression that only matches strings of some given lenght $L$, and showing that if you are given a rules for $L$ you can derive one for a length of the order of $2^L$ such that the new expression is only some constant factor lerger than the original one. By iterating this process you can obtain rulers of length $2^{2^{\cdot\cdot L}}$ for any height tower of powers of 2 that you want, and that leads to truly huge values rather rapidly.

To illustrate the process we will start with a ruler of length 3, and the simple regular expression `x x x` consisting of just 3 characters does the job. The general concept that will be set up is based on counting in binary with (in this case) 3 bit numbers, which may be presented to start with on two lines as

```
   #000#001#010#011#100#101#110#111#
...#xxx#xxx#xxx#xxx#xxx#xxx#xxx#xxx#...
```

Here the upper line can be seen to be counting and it uses the hashes to keep the separate binary values neatly apart. The lower line uses our ruler and will help to enforce the regular patter of where the hashes appear. It can be rendered as the simple expression `#(xxx#)*` which just takes our ruler and sets up repeats of it separated by `#` characters . Well this looks like two strings, one for counting and one for ruler. So then we can use the language of regular expressions we interleave the two so we have a single string where odd characters are from the top line and even from the bottom:

```
##0x0x0x##0x0x1x##0x1x0x##0x1x1x##1x0x0x##1x0x1x##1x1x0x##1x1x1x##
```

This may be less easy for a human reader to decode and so in the following presentation we will sometimes use the 2-line format, but what it will always mean will be the flattene out version. Note that with our ruler of length 3 the string we have here is of length $2 \times (3+1) \times 2^3 + 2$ where the $3+1$ and the final $_2$ come from counting the hashes and the extra factor of 2 is because there were two lines or text merged. Calling this just $2^3$ is something of an understatement but that does not alter the main thrust of the presentation, which is that this will allow us to build extraordinarily long rulers.

Initially it might seem that setting up regular expressions that constrain text strings to count in binary is going to be hard. The way of doing so is is to consider what strings are *not* counting sequences. So we will establish a set of regular expressions exact of which characterises somenthing that we do not want to see, and using negation and the "and" operator ensure that none of them apply. What remains will only be able to be our counting sequence. This of course is the just what Sherlock Holmes taught us: "When you have eliminated all which is impossible, then whatever remains, however improbable, must be the truth.".

To start with it makes sense to enforce the block format of the string, which amounts to demanding that hashes in the top line can only be in places where they lie above hashes in the bottom line. Well any string that violated that rule will have either a hash above a non-hash on a non-hash above a hash somewhere. If just deal with the first of those it can be desribed using a pattern that carefully uses pairs of characters everywhere

```
(..)*  (# (~#)) (..)*
```

This little expression and its friend that matches the bottom to top will form part of the expression we are building to describe our long string. As written above it is only 17 tokens long. If you needed to use more primitive regular expressions each wildcard charactter . might expand into something like `a|b|c|d...0|1|#` and have a size proportional to the number of characters in your alphabet, but that again us just a constant.

The next consaint can be expressed positively and is that the string we are going to match must start at `000` anmd stop at `111`. Again this is easy:

```
#. (0.)* #. (..)* #. (1.)* #.
```

where this version demands that our initial ruler is of length at least 2 so there are at least two hashes between the first and last segmnents, This wors because `# 0* #` insiats that there are only zeros in that region, and similarly for `1`s at the end. And the previous rule has made cartain that the `#` marks all line up and that forces those strings of zeros and ones to be the desired length. Note that all the messy things like `(0.)*` and indeed all the dots here are just arranging that we only paty attention to top-line (eg odd position) chartacters. because that is really just a bit of technicality in future we will present things in the form as if they were only applied to the top line and the extra mess to ignore the interleaved bottom line will be assumed to be applied:

```
# 0* # .* # 1* #
```

Looking ahead it will make sense te enesre that we only count through the binary sequnce once, so that it is not possible to have a second section consisting all of zeros. This is also rather easy if we make it a pattern that we will say must not apply that puts two separating hash marks ahead of a block of all zeros. It is written here ignoring the issue of the lower ruler line...

```
.* # .* # 0* # .*
```

which is anything then #, more anything and then a block like #000#. By disallowing that we will count just once.

That leaves what may feel like the hard part which is to ensure that sequential blocks of digits count. It feels easiest to first explain what must happen, but the fragments of pattern we end up using will all need to be set up to disallow anything else. To increment a binary number one looks for the rightmost 0 and of course that means any digits to the right of if must alll be 1s. The next number up must preserve all bits to the left of that key zero, flip the zero into a one, and turn all the trailing ones into zeros. A way of handling this is to imagine that corresponding bits in the number and its successor (which must be a distance apart set by our ruler) are underline and displayed in bold, then four cases apply which are shown a bit informally here:

...$\underline{\mathbf{0}}$ ... 0 ... # ... $\underline{\mathbf{0}}$ ...
...$\underline{\mathbf{1}}$ ... 0 ... # ... $\underline{\mathbf{1}}$ ...
...$\underline{\mathbf{0}}$ 1* # ... $\underline{\mathbf{1}}$ ...
...$\underline{\mathbf{1}}$ 1* # ... $\underline{\mathbf{0}}$ ...

The first two cases have at least one zero after the particular character and insist that the character remains unchanged, while the second two are when there are only 1s to the right in which case the bit is flipped. These cases are mutually exclusive but between them cover all possibilities.

The key trick is clearly to get the effect of the underlining set up in our petterns, and that can be explained in a "two row" presentation again. where for that first the above four cases what must be avoided is

```
.* 0 .* 0 .* # .* 1 .*
.* #      xxx.     # .*
```

which can be read out as the lower line using the ruler xxx. to insist that the two # marks are the length of our ruler plus one (to allow for the # in the main pattern) apart. The top line looks for zero which is followed by at last one zero before a #, and that after the # there is a one that has been forced to be just the right distance away. Matching a 1 there captures the case that is wrong, so saying that this pattern does not match leaves only the good cases. The ruler line used here is not the same as the one used before in that it puts in only two markers that are properly separated, and the leading and trailing .* parts say that it can match and indicate that separation anywhere. Ah well – adding this in will mean interleaviung these two patters again, and the interleaving with the previous repeating ruler will have to remain in place. The result is that characters are going to end up being worked on in blocks of 3 or perhaps 4 rather than individually. This just makes the big ruler that we are creating with our counting scheme a bit longer yet and is not a big cause for worry.

Each of the fragments of regular expression introduced here are rather small even when all the interleaving stuff is allowed for, and in the end we have shown how to create a pattern whose size is some modest multiple of the size of the original ruler, but that will only match strings that are exponentially larger.

So why might this be a worthwhile exercise? Apart of course from it being a demonstration that pattern matching can do more that was first apparent. Well it can be part of the process of showing that understanding particular patterns can be hard. Begin with a fundamental result about these patterns, which is that any regular expression will have a corresponding deterministic finite automaton that can recognise exactly the strings that the pattern will match. Well to rephrase that in less tecnical language for any pettern there is a really simple program that can be used to apply it to input strings. The program is basically of the form:

```
int state = 0;
for each character in the input
  state = transitionTable[state, character];
return isAcceping[state];
```

where the transition table is used to adjust the state as each fresh character is presented and the `isAccepting` vector reports at the end whether the string matched. It is clear that the work done for each input character is rather small – just accessing the array, and so this is liable to be really efficient way of performing pattern matching. There are readily available programs[2] that can build the tables and they will tend to be a bit cleverer than using a simple 2 dimensional array for the transitions would be because many entries in the full rectangular block would never be used.

Both for practical and theoretical purposes it is wothwhile to consider how many distinct states will be required to build a program of the above form that will correspond to a given pattern. One bound that is pretty clear cut is that there must be at last as many states and the shortest string that matches the patterm. To see that imagine the sequence of states encountered in the process of a succesful match of such a shortest string. If tere were to be fewer atates than the length of the string them one of the along the way would have to be repeated, as in

```
ABCDEFDGHIJ
```

[2]notably lex and flex

where state `D` has been entered twice. Well if that happened it would be possible to provide input that ran up to that situation and then continued ona home run, omitting all the steps between the first and second encouter with the repeating state.

```
ABCDGHIJ
```

This is a shorter string that has just been accepted, contradicting the assertaion that we had started by looking at the shortest matching input. What we can read off from this observation is that provided we allow extended regular expressons with a negation operator there are tolerablly concise patterns that would necessarily lead to super-galactically large numbers of states. This is a pretty convincing reason not to allow negation in practical applications of these patterns!

So next consider a challenge that if it was readily solvable would be of great use: given some pattern expressed as a regular expression is there another more concise pattern that matches exactly the same set of inputs? From the perspective of a theoretician this can be addressed by emulerating all possible regular expressions with all the ones using $k$ characters before the ones with $k+1$ characters, amd with ones that are the same length in alphaberic order. Over any fixed alphabet of the characters you are working with this is clear-cut. Now for each such you want to test if it happens to match the same inputs that your original one did. Well if I have two regular expressions $P$ and $Q$ then the language $(P\&\ Q|Q\&\ P)$ will match any things that $P$ does but $Q$ does not and vice versa. So if the two expressions do the same thing this new one I have just set out will not match anything at all. But that is something that there is a clear-cut way of checking for. Make the transaition table for this new composite expression and supposing it turns out that there are $N$ states then if there is any string that it accepts at all there must be one of length at most $N$. This is just the flip-side of the previous remark about lengths of matched strings and the number of states. If your alphabet if of size $k$ then there are only(!) $k^N$ strings of length $N$ to check and if none of those match then nothing will. The explanation here is a bit needlessly inefficient but it demonstrates that in a finite amount of work one can tell of two patterns match the same set of strings and that then by exhaustive search in a finite amount of time it would be possible to exhibit the most concise pattern for any particular behaviour. But although tis shows that the challengs is decidable in the sense that one can have a computer program that would guarantee to address it and eventually complete its work with a correct result, the resources needed to achieve that might well be truly ridiculous.

The explanations given here show how huge rulers and hence regular expressions that necessarily load to transition tables with huge numbers of states csn arise. They do not complete the proof that there can not be some clever way of reasoning about e.g. equivalence between patters that manages to sidestep generating the associated transitionm table. And details of that will be left as a futher literature search or research project for the reader, But a key step that it may by now be easy to recognise is that the sorts of techniques explained here that block a string into segments each representing the next binary number on from the previous can be modified so that the string is split into blocks such that each represents the next state of a fairly arbitrary computer one step on from the previous. It would be normal to set this up describing a Turing machine with a tape whose length is limited to the size of our ruler. Given that and the large amount of theory known about the problem of telling of a Turing machine will halt you can imagine that formalising a statement that there can be no amazing short cuts and that something very much like building the full state transition table for the description is going to be necessary to tell if certain messy patterns will match anything at all.

In writing this the authors here want to leave the reader understanding that there is always more to be looked into!

# Chapter 16

# Parallelism and Concurrency

*Gosh this can be difficult with both avoiding deadlock and arranging that the overhead of all those semaphores etc does not slow things down badly. And for a kicker there are memory access semantics such as acquire and release, and all the joy of lock-free methods.*

# Chapter 17

# Puzzles

*Here is a chance to dump things that only exist for fun. It will at least include the Quine, but could consider rubic's cube and Sudoko as well as logic puzzles like "who owns the zebra?"*

# Chapter 18

# Floating Point

*Numerical monsters! How to read and print floats accuretely. double-double to gain higher precision and the way that relies on precise detail of the underlying representation. Directed rounding and interval arithmetic – error estimation!*

# Chapter 19

# Formal Proofs

*Operational vs. Denotational semantics. How are you sure your proof checker is not bug-ridden? Looking back to Dijkstras mantra that testing can not demonstrate correctness, only incorrectness.*

# Chapter 20

# Iteration

*One thing that computers are really good at expressing is repeating some simple action. One might have hoped that when that action really is simple that the consequences of doing it over and over again would hold few surprizes. However that is very much not the case, and repetition or iteration can lead to all sorts of complications and difficulties. This chapter starts with a couple of famous examples.*

## 20.1 Collatz

Perhaps the best known illustration of the fact that repeating a small calculation can have tricky consequences is due to Collatz[**?**]. Given a positive integer the step taken in to halve it if it ie even or to multiply by 3 and add 1 it it is odd. This step should be repeated until the number 1 is reached, or until it can be shown that this will never happen. So for instance if you start with 27 you get the sequence

```
27 82 41 124 62 31 94 47 142 71 214 107 322 161
484 242 121 364 182 91 274 137 412 206 103 310
155 466 233 700 350 175 526 263 790 395 1186 593
1780 890 445 1336 668 334 167 502 251 754 377
1132 566 283 850 425 1276 638 319 958 479 1438
719 2158 1079 3238 1619 4858 2429 7288 3644
1822 911 2734 1367 4102 2051 6154 3077 9232 4616
2308 1154 577 1732 866 433 1300 650 325 976 488
244 122 61 184 92 46 23 70 35 106 53 160 80 40
20 10 5 16 8 4 2 1    4 2 1 4 2 1 4 2 1 4 2 1...
```

If you do reach 1 then after that the pattern would repeat `4 2 1` from there onwards. The Collatz conjecture is that for all initial numbers the sequence eventually falls down into the little `1 4 2 1` loop. I.e. there are neither any other cycles involving larger numbers nor any starting points from which the values increase without limit. As of 2026 it seems that compouter tests have checked ever starting number uf to $2^71$ which is around $2.3610^{21}$ and shown that all those sequence terminate at `1`. This challenge has been addresed my many mathematicians with computers being used to provide emirical background and sometimes to assist in formaizing proofs of sub-results, but to date while it is generally believed thet the conjecture holds no full proof of it has been found. Variations and generalizations of the simple rule (for instance to allow negative as well as positive numbers or to extent it to real or complex arithmetic) have been tried. For such a very simple statement it is a really hard problem!

## 20.2 Adding up

For a complete change of theme consider the following obviously trivial little program (so simple that it does not need to be given in any particular real programming language):

```
r = 0.0;
for i := 1:100000000
  r := r + 0.000001;
print r;
```

This adds up a hundred million copies of a millionth, and so the result printed at the end should obviously be just 100.0. Well of course this is computer (floating point) arithmetic and so if we should expect rounding errors to intrude so that the final result is not quite correct. If we use single precision (i.e. 32-bit) arithemtic we expect each arithmetic step to introduce a little bit of error, keeping around 7 decimal places correct. Taking a pessimistic view maybe each addition introduces and error that is about $10^{-7}$ and we have just done that 100 million times, so it will not be too astonishing if our final result is wrong by about the product of these two numbers, in other words 10. That was of course an informal (and incorrect!) estimate. What in fact happens is that the result printed is 32.0 rather than the ideal 100.0 so it is grossly wrong. And things get worse because if you change the code to add up a thousand million numbers (so the ideal answer would be 1000.0) the calculated one remains unchanges at 32.0!

When this is explained it makes very good sense, and furthermore it becomes possible to understand in rather fine detail just how rounding errors will build up all through the calculation. Although the actual sume here will be being done by the computer using binary arithmetic much of it can be explained by working in decimal with numbers stored to 7 siognificant figures. Suppose then that you have done that as far as the value in `r` having reached the value 10. Adding in the next copy of a millionth will look rather like

```
     10.00000
   +  0.000001
```

and you can now see that the value a millionth has been shifted beyond the 7 digits we will be prepared to work with whe it is aligned ready to be added in. So in this decimal case the result will get stuck at 10 and it happens that with standard single precision floating point the stopping point is 32.

A futher point is that the number “a millionth” looks really nice in decimal but in binary it is `10000110001101111011110...` with a suitable exponent. So now let's consider what adding that to 1 is going to look like...

```
      1.0000000000000000000000
   +                        10000110001101111011110...
```

and this shows that many of the low bits of the increment are going to be discarded. Well a reasonable approximation to the story of this calculation is that for the very first addition all bits of the small number will be taken account of. For the next two all one bit will be lost from the end. For the next four there will be two discarded bits and so on until after what would be just $2^24$ (apart for the consequences of these truncation errors) **all** the bits are lost. That all means that anybody who wanted to be really careful could analyse just what happened over each power of 2 range and predict just what the partial results were going to be basaed on knowledge of the binary representation of a millionth!

It is reasonable to ask if this matters or if anything like it ever arises in sensible rather than utterly artificial calculations. Well it can happen. A natural way to obtain numerical solutions to differential equations involves taking small steps in time (or whatever the independent variable happens to be) and based on that adding a small adjustment into values. The simplest case is Euler's method for a single simple equation $y' = f(t, y)$ and to take a step of length $h$ it will use $y_{t+h} = y_t + hf(t, y_t)$ and it will keep doing that to creep along from the starting value of $t$ as far as it needs to go. To keep accuracy high a small value of $h$ may be needed, so this may involve rather a lot of steps. Pretty well all the more sophisticated schemes end up with the same tendancy to obtain a final answer by adding a lot of small adjustments to some starting values. So the bad things that can happen can apply! Of course using double rather than single precision is liable to do rather a lot to sweep the issue under the carpet, but even there the results from adding up many small values may not be as accurate as anybody would naively expect.

Is it possble to add up very many numbers and not suffer loss of precision in this manner? Well if you have all the numbers avaible from the start the answer is yes and a good technique is easy to explain if not obviously cheap. If you start with $N$ numbers begin by identifying the pair of them whose sum is smallest. Pick them out, add them together and put the result back so you now have only $N - 1$ to work on. Keep doing that until you eventually end up with only a single value, in other words your result. If that feels too expensive and if there is some reason to believe that all your numbers are about the same size it can be good to take a pass over all $N$ numbers replacing each consecutive pair by their sum, so you now have $N/2$ remainining. Again repeat this process until you have just one number left.

So something as straightforward as merely getting the computer to add up a bunch of floating point values can be messier than might at first be apparent.

## 20.3 Population dynamics and Chaos

Collatz sequences show that a simple transformation on integers can lead to deep questions. The above discussion about the consquences of rounding in floating point arithmetic show that that can be a bit messy, but what about real-number arithmetic if one pretends that computer-based finite precision does not intrude? Well start with some parameter $r$[1] and a starting value $x_0$ strictly between 0 and 1, and investigate the iteration

```
  x := r*x*(1-x);
```

This might be interpreted as a rule for judging the size of a population from generation to generation, where at each step `x` is the number of individuals scaled so it lies on the range from 0 to 1. If `x` is small the `(1-x)` term hardly makes a difference and `r` sets a reporoduction-based growth rate. However as `x` grows it starts to reach a limit set perhaps by limited on the food supply or by overcrowding, and as it gets close to 1 it will eventually start to decrease even if `r` is sunstantial. Thus exploring this iteration provides a simple model of many ecologies, including economic ones where some product is really

[1]It will be normal to restrict `r` to the range 0 to 4

attractive when rare (so many fresh customers buy it) but when the market saturates it becomes uninteresting. Pleasingly although this model is pretty simplistic the sorts of behaviours that emerge from it are observed in the real world. And these behaviours are a curious mixture between understandable and plain weird.

The behaviours depend on `r` as follows:

**r is less than 1** Here things are simple - the population, wherever it starts, dies away towards zero.

**1 to 2** The behavior is again simple, even if slightly less obvious. Regardless of its starting value `x` tends ta a stable fixed value. The mathematics that shows what this value is and that explains how rapidly the fixed point is approached is not hard, but since this book is about computation not mathematics it is skipped here!

**2 to 3** In the long term here the behaviour is similar and the population stablises, but before doing so it jumps around above and below its final value for a significant time.

**3 to** $1 + \sqrt{6} = 3.449...$ This is where things start to become messier, as parhaps indicated by the fact that the upper limit set for `r` is not a number that would have been obvious to guess. So the existence of that limit helps justify this as starting to get difficult! Here the trajectory taken by values of `x` ends up in alternation between two values and those values are given by $(r+1\pm\sqrt{r-3(r+1)})/2r$! The population enthusiastically reproduces and grows beyond suatainability and then has to collapse, but things end up with a stable pattern of that behaviour. Amazingly something of this sort of behaviour can be observed in the United States market for hogs. In years when pork-raising is profitable formers expand production. Eventually that saturates the market and prices drop leading to farmers withdrawing from the market. Over a many decades the records show cyclic behaviour with repetitions of the change in pricing happening over on average around 3 years. The exact behaviour is not quite as neat as the simple iteration we are looking at, but it is pretty clear that it is related.

**3.448 to 3.544 end eventually 3.57** . The number shown as 3.544 there is more precisely 3.54409035955192285361596598660480454058. . . which is the root of a certain 12th degree polynomial. And up to there the values of `x` end up bounding between 4 rather then 2 values. Just beyond there it will be 8, values then 16, 32 and so on. If this seems bad things get worse when you reach the end point 3.56994567187094490184200515138649893676. . . by there the repeating pattern has run through cycles of all power of 2.

**3.56994 and up** Here the behaviour of the iteration descends into chaos. Minute changes in the initial value of `x` lead to wildly different sequences which do not repeat values at all. Well that statement is not quite true – there are some special (small) ranges of values of `r` that still lead to cyclic behaviour even beyond the critical point where chaos is the general effect. In general the behaviour is truly complicated with traces of pattern visible nestling within the otherwise semingly random behaviour.

There are plenty of adjustments to the iteration considered here. For inatance a version extended to complex numbers uses just

```
  x := x^2 + c
```

for some complex constant `c` and asks the simple question "If you start with `x`=0 what values of $C$ lead to $x$ eventually becoming var large". Plotting a map of of the answer to this question as `c` ranges over the complex plane leads to the well-known but rather decorative Mandelbrot Set[**?**].

## 20.4 Intentional random-like behaviour

The chaotic behavour of the above process, which is known as the Logistic Map, might seem pretty randomish, but it has just enough residual regularity that relying on it as a sort of randomness is not safe. There has been a huge amount of study of methods that can used to judge if sequences of values will count as "random". For many years it was commonplace for the library functions provided with computer systems to claim to generate random sequences but to fail many of these tests. Sometimes one would like a "random" sequence of characters, sometimes integers in a specified but larger range and sometimes floating point values. And for practical reasons it will be desirable that the values are generated rapidly. That means thay must emerge from iterating some simple calculation. The ones noted here are not going to be the best known. They are instead presented to show how concise fragments of code can behave in manners hard to predict and somethimes difficult to analyse.

### 20.4.1 RC4

This scheme generates a sequence of 8-bit bytes and for many years was a heavily used component of many security schemes, based on a hope that adversaries would not be able to predict which byte values it would generate. These days it is not considered secure, but it is still fast and concise and illustrated that iterating something that is easy to describe can have hard to understand consequences. Start with an byte array called `v` of length 256 filled with some permutation of the values 0 to 255. If unpredictable values are required this needs to be an unpredictable permutation, but for now pretend that that issue is solved. The code used to generate a sequence of bytes is just

```
i := 0;
j := 0;
repeat
  i = i+1 mod 256;
  j = j+v[i] mod 256;
  swap v[i] with v[j] in the array;
  emit v[v[i] + v[j] mod 256]
```

All the mention og `mod 256` merely indicates keeping only the low 8 nits from the arithmetic – it is not anything very sophisticated. The verb `emit` indicates a value which will be the next pseudo-random byte that is to be provided for use. It is perhaps astonishing that something so compact and that does not do any particularly tricky steps generates sequences that for some years were viewed as unpredictable enough te secure wirelss networks and be useful for file encryption. So if you want to delve into difficulty do some research into the ways in which it was in the end shown to be insecure!

### 20.4.2 Pseudo-random numbers

If a sequence of numbers is generated by a program it will be improper to call then "random" since the program that generates them of itself specifies a pattern that they follow. But for many computational purposes it is reasonable to use pseudo-random sequences generated by algorithm provided they they do not have serious bias. One of today's winners based on a combination of unpredictability and low cost is the Mersenne Twister[**?**]. The version of this normally used works based on the fact that $2^{19937} - 1$ is a prime, which when it was found in 1971 was for a while the largest known prime. One can hope that the fact that this scheme's robustness is based on the primality of such a large value suggests that there is some serious theory undepinning it, and its details are a bit too messy to document here. So instead here is a scheme that is truly simple to code and that has been widely used. It maintains a state S that is a 64-bit integer and at each step merely performs

```
S := 6364136223846793005*S + 1;
```

where only the low 64-bits of the product are kept. Clearly terms in the sequence that this generates alternate between being odd and even, so the .last significant bit can hardly be viewed as usefully random, and similar issues impact further low bits, so it would be usual to only view about the top half of the valies of S as useful. That multiplier is has not been picked arbitrarily – a great deal of study and hard work has gone into choosing it so that the sequence of values generated have as few patterns in them as is possible. The real difficulty here is thus in understanding the analysis that leads to the choice there. A scheme following this pattern but with integer state and multiplier using yet more than 64 bits could still count as fully respectable. But one should take care not to rely on more than $2^32$ successive values from this 64-bit generator, and with modern fast computers it is easy to imagine circumstances in which random sequences that long might be used.

## 20.5 Life

John Conway's Game of Life[?] runs on grid where each position is either filled on blank. These states may be described as the call there being alive or dead. At each clock tick the following rules are folowed:

- Any live cell with fewer than two live neigbours and any with four or more live neighbours dies (isolation or overcrowding);
- Any dead cell with exactly three live neighbours comes alive.
- In other cases cells retain their state.

The neighbours here refer to the eight positions surrounding a cell. Even though these rules are not at all complicated, the evolution of the patterns of live cells can be quite elaborate. It would be possible to give many examples of witty behaviour, and in fact with suitable initial configurations running the rules of Life can act as a rather slow computer. But about as amazing illustration of how difficult it is going to be to predict just how a population evolves as can be imagined is shown in `https://www.youtube.com/watch?v=xP5-iIeKXE8ON` where an initial population has been set up so that from a distance the pattern looks like a grid with some of the squares shared in. When it is set to run it evolves in a way where after a number of steps it again shows a grid with filled in squares, but the new choice of which squared are shaded follows the rules of Life itself on the big board. The really simple original set of rules have been coaxed into modelling their own behaviour! There had been several previous designs for this sort of self-simulation, but this one has the visual advantage that it is really easy to discern which of the moddeled cells are alive and which dead. Hmmm I think that setting yourself the challenge to recreate that pattern or better it with a smaller or faster emulation would count as difficult!

## 20.6 ...and so?

The examples here show that in a range of ways rather simple fragments of code can lead to behaviour that can be remarkably hard to predict or reason about. But basically any worthwhile program will have some loops in it, and if we know that even trivial loop bodies lead to difficulties the same has to be doubly true when the iterated operation geta a little messier. Looking at this means that we have to face up to something of a contradiction. On the one hand the computer will behave systematically and deterministically. It obly does what it has been told to. So the results it will generate are totally determinated by inputs and the code itself. On the other hand for almost any reasonable program it is going to take serious good fortune to be able to predict its eventual behaviour, which can be complicated beyond rerasonable imagination and truly hard to analyse.

# Chapter 21

# Random and Nondeterministic models of computation

*This certainlly covers NP. It might things like Arthur-Merlin protocols and zero-knowledge proofs.*

# Chapter 22

# Hardware

*An enormous amount could be said about hardware, with such diverse issues as silicon and device physics, the technology of potential quantum computers, the practicalities of avoiding things overheating as well as all the variations on instruction set design. The first issue addressed here illustrates how something that can seem really trivial in fact conceals multiple layers of complication. That is followed by tracing enough of the early history of computing engines to point out that at least the earlier ones are at least conceptually reasonably straightforward and while building a system based on those ideas would be a serious size project it is far from unimaginable.*

## 22.1 USB

There are some things related to computers that seem so commonplace and straightforward that it is hard to imagine that anything is at all complicated, far less difficult. A good example of this is using USB cables to charge phones and other devices and to transfer data. What could possibly be a problem with that? There are basically a few pairs of wires and either power or data can be shipped along them with little fuss.

Well actually there is quite a lot of fuss at a number of different levels. Three issues in particular make this messy:

**Sharing one cable between several uses** A naive view is that when you plug in a cable it connects between two separate devices that then can exchange information. But a moment's thought reveals that your computer might end up connected to a hub that that supports several separate peripherals that each wish to interact with it, and these days that same USB cable might be being used to provide power to the computer. And when you plug in a new device ...

**What should be permitted when you plug a cable in** There are some very simple cables that are plugged into wall warts or power banks that deliver a fixed voltage but are utterly uninterested in data. There are other setups where the power to be provided will be at a higher voltage and with a higher current capability – but when used those ought not to overload and damage simple devices. Some devices will only be capable of running at low speed, but others can and should be used for the fastest possible data transfer.

To cope with all this the USB standards mandate that when a link is first made there has to be a negotiation phase where each end indicates what is wants and what it is capable of accepting. As part of the full process of establishing a link a device should be ready to identify itself to an extent that if the other end is a computer it will be able to select drivers – either generic to the device type or as provided by the device manufacturer.

**Supporting the speeds and power levels**

**etc etc**

## 22.2 Computers built from scratch

Obviously these days most people buy their computer in a ready-to-run state. But abybody really concerned about how things work ought to like to understand (at least in principle) how they could build one for themselves. While a substantial undertaking it may be that until there is concern for ultimate performance and lowest cost this is more feasible than it would at first seem. A way to underatand this is to look at the sequence of developments that led to modern machines:

### 22.2.1 Mechanical arithmetic

Arithmetic has of course been around for a long time, and for both scientific and commercial purposes it coudl get laborious. So in the mid 1600s Pascal[?] created a machine using gearwheels that could add and subtract (coping with carry operations) and which could assist with multiplication. So machanisation of the logic behind arithmetic has a long history and is really well understood. To took around 200 years until calculating devices became reliable, commercialised and widely used, but over time the designs were improved so that multiiplication and division were easily performed and electric motors sometimes replaces the handles that were at first used to wind the gears. The message from all this is that by the time anobody even started to think about computers as a whole the machanisation of arithmetic was well understood.

### 22.2.2 The start of programmability

In the 1820s Babbage[?] started work on a "difference engine" which had it been completed would have consisted of around 25000 component parts. This amounted to a number od adding units cascaded such that between them they could calculate values of a polynomial. This was of huge interest for the generation of tables of mathematical functions. A key part of Babbage's work there was integrating both direct printing on paper and the creation of type that could be used for subsequent large scale reproduction of the results – this all mattering because once the machine was set up it could avoid human error noth on the calculations an in transcription of results to the versions of the results that would be distributed. All this showed how multiple arithmetic steps could be combined to good effect. Later Babbage ideas considered controlling a (mechanical) arithmetic engine using the punched cards that had been developed to mechanise the weaving of complex patterns on Jacquard looms, and while at the time it was not feasible to build all he designed this showed at least in principle jow much might be achieved with no electronics at all. But with quite a lot of steam power. Anyone wanting to build their own computer can look back at all this early work and see how everything really can be mechanised with enough gear-wheels and levers!

### 22.2.3 Electromechanical calculation

An electomagnet is made by winding a coil of wire around a soft iron core that becomed a magnet when current flows. If this magnet is caused to attract another piece of iron and thereby throw some switches what one has is a relay. Wiring switches in series or on parallel give **and** and **or** operations and it then becomes almost straightforward to transliterate the design of a fully mechanical calculator into one built out of relays. While it is possible to do this and keep on using decimal arithmetic one can note that a relay has two states – energised and not energised. This leads to an insight that performing arithmetic in binary will be easier than using decimal. In the 1930s Konrad Zuse[?] had built a fully mechanical calculating machine where punched tape was read to indicate each step that it was to take. The Z1 was built in Zuse's home and was privately funded. It had 16 words of memory which were of course just for its working date since the program was on the punched tape. A little later a succesor, the Z2, was built using the mechanical memory scheme from the Z1 but around 600 relays for arithmetic and contol. This now took under a second to perform a 16-bit fixed point addition. This was succeeded by a Z3 and Z4 which improved speed and reliability and very much demonstrated that electromechanical computation was realistic. And probably that building and maintaining it was less stressful than having a fully machanical design, since Zuse's machines were an order of magnitude less heavy and bulky than the Babbage designs.

### 22.2.4 Memory as a key technology

The systems considered so far can be fairly general purpose but they take their instructions from punched tape or cards. The concept of putting instructions and data in the same or similar sorts of storage existed, but building suitable memory was not easy. So early calculating machines either used cards or punched tape, or had their operations set up on hand-rewired units that beeded very laborious re-work for each new task. So anybody considering building their ownb computer needs to worry about implementing memory! Two competing schemes were used in the earliest fully programmable general purpose machines. One used a steerable electon beam to record information as a pattern of dots on the screen of a cathode raw tubs[?]. The electric charge in the dots could be sensed, although it faded fairly rapidly and so the information had to be read and re-written constantly. The alternative was to send pulses of sound through a medium long enough to give a useful delay, and listen for the pulses at the other end and recyle them. The use of tanks of mercury for this was technology that built on work done to declutter radar images by avoiding display of reflections from static objects, so in essence it was not invented just for use in computers. With mercury delay lines it was quite delicate – the mercury was kept at a constant temperature and use at 40°C to get a good accoustic match between it and the transducers at each end. And of course mercury is heavy, expensive and poisonous! Both of these technologies could store of the order of 1000 bits.

A subsequent variant on delay lines used torsional waves in steel wires, and the waves generated by applying small twists to the end of the wire were reasonably robust and the wire did not need to be kept laid out in a straight line so the unit could be kept compact. So while this did not improve speed or capacity much it did simplify construction and move towards compact and reliable designs. It seems reasonable to imagine constructing a version of it for a personal project!

Perhaps the big breakthrough for computers was the development of core memory. That is based on little torroids of magnetic material with wires threaded. By pulsing current through the wires the torroids can ve magnetised and re-magnetised either clockwise or counter-clockwise and it is possible to sense when they change. This scheme made it feasible to build memories of sizes that grew from kilobytes of a megabyte or so. While conceptually simple, the practical construction involves threading wires through all the cores involved with one core for each bit to be stored, and of couse it is necessary to source suitable magnatic material. That all seems like a quite substantial project!

### 22.2.5 Vacuum tube to Transistors and beyond

The earliest fully electonic computing devices were built using vacuum tubes otherwise known as valves. These are conceptually simple! A heated cathode will emit electrons. A anode reasonably close to it and collected to a positive voltage will collect those so a current will flow even though there is no direct physical collection. A fine wire mesh (or just spiral) placed between the rwo electodes is called the grid. If that is much closer to the cathode than the anode is and if a negative voltage is put on it that cuts off the current flow. And the voltage changes on the grid to achieve this can be much smaller than those used on the anode. To make all this behave it has to be in a vacuum (and quite a good one at that!) so that there is no gas present that could ionise and confuse matters. As well as all their uses in radio receivers and the like, valves cen be used to create switch circuits and hence implement logic. For that everything that had been done by way of mechanical and then electromechanical calculation can be replicated but it can run a lot faster. However with all those red-hot cathodes and the need to preserve the high vacuum by the times a computer had been built using several thousand valves the run-time between failures could be fairly short, and for a number of years relay based hardware was more reliable even if it was slower.

After a while transistors were invented. Again the early versions worked really well and made the transistor radio a household staple, but in the numbers needed to build computers the big challenge was to boost reliability. But still the reduction in size and power consumption that they enabled was a big step forward. For a while the issues of cost and reliability meanst that a scheme called the parametron[?] based on pumping energy into resonant nonlinear circuits by stimulating them at twice their natural frequency provided a practical if somewhat slower solution, and a significant number of computers based on that technology were built. Just as building a model showing how core memory to store just one bit is a jolly home project, building circuitry that shows how a parametron can act as a majority-of-three gate (and given that as **and** or **or**) is thoroughtly deasible.

With the development of integrated circuits including ones that provided kilobytes of memory it suddently again became feasible for amateures as well as smaller companies to design and build their own computers from scratch, although soon after the ready built mictoprocessors would be the more attractive option for many.

Soldering many integrated circuits onto a prototyping board may be fun, but a more modern way of conducting experiments in computer design (apart from using simulation which could be seen as cheating!) is to use a field-programmable gate array. These tend to come with a sea of basic gates of various sorts and a scheme whereby the connections between those can be set up electronically in very much the way that the information stored in a memory card can be written and re-written. Fairly cheap teaching, hobbyist and developer boards for these are available and those can be perfecatly capable of making it possible to re-create all the early steps in caclulator and computer development. Is it difficult? Well there are quite a few skills to learn and technologies to master but it is utterly feasible.

One might look at current computers and fear that the only ones of any interest are going to be mind-numbingly complicated and so a personal design will not be of great relevance. Well there was a time (in the 1980s) when each newly announced computer model had more bells and whistles that its predecessor – this generally with a view to making it easier to build programs for them. Their internal complication was such that very often design flaws meant that they did not do what they were intended to. A fine example of that was that as late as the 1990s Intel issued a processor where certain divisions of floating point numbers returned incorrect answers[**?**]. This was particularly high profile, but other manufactrurers also issues hardware with odd and behaviours, which could often by masked by having compilers avoid certain combinmations of instructions. Performance was also sometimes strange, with cases where sequences of several primitive instructions ending up executing faster than a specialised instruction presumably intended for use as an optimisation. The RISC[**?**] movement pushed the idea that having a really simple design would make it possible to build a computer that executed each of its somewhat primive instructions fast, and that the more complicated and "powerful" instructions and operations that were commonplace elsewhere were in reality used somewhat infrequently. As a result doing the simple stuff really fast and accepting the fact that some uncommon steps might be a bit inconvenient could lead to an overall improvement. And of course having a simple basic design for an instruction set meant that either the computer built on it would end up smaller and cheaper then the competitation, with fewer errata in its manual and brought to market sooner. Or that the simple fundamentals could then be implemented using every single clever technique that had emerged through the history of computer to make it especially fast – and that applying all those techniqes would be a lot more feasible when building on a simple model than when starting from a state of complexity. So there has been a time when a fresh design emphasising simplicity could be a winner.

So if you want to be the one that makes the next breakthrough you need to spot what is broken (and then have quite a lot of good luck!). Perhaps it will be related to security because with existing computers it is clear that it is horribly easy for even carefully constructed systems to end up compromised. Or maybe some alternative to the tendancy of those pusing the limits of artificial intelligence to wan to use more silicon and more electricity that many nation states.

The message that this section wants to convey is that while designing and even constructiog your own computer from the very base upwards is a bit challenging and is certainly a lot of work, it is in the end something that an individual or small group can achieve. So it counts as difficult rather than impossible.

# Chapter 23

# Networks and other interconnects

*DNS servers and the Byzantine Generals Problem. The ISO 7-leyer model (and its limitations). The extreme complications of USB. long-distance links: latency. error correction. security and resilience.*

# Chapter 24

# AI

*Well a chapter on this is needed even if I do not really either know much about it or like it!*

# Chapter 25

# Unusual Computer Language

*Why was Babel inevitable? Aspects to focus on*

- *communication (eg Erlang)*
- *avoiding common bugs (eg Rust)*
- *avoiding more subtle bugs (eg SML)*
- *handling low level machine features (eg C)*
- *memory constrained situations (eg Forth)*
- *text data (eg PERL)*
- *typography (eg TeX)*
- *boosting an author's ego (eg Aldor) [ha ha ha]*
- *expressing proofs (eg Lean)*

*No doubt plus a bunch more!*

# Chapter 26

# Undecidable problems

*Well obviously the halting problem… But I think that the fun to be had here is showing collections of other problems that have the feature that if they were solved then the halting problem would fall. Hmm I need to do some research on this topic!*

# Chapter 27

# Lessons that have been learned

*Each of the chapters here has presented a challenge or a way of viewing things that reveals a level of difficulty. But in each case the informal introduction here at best provides an introduction to the topics covered. If you seek real "difficulty" you can take almost any of the issues raised here and dive into the serious literature on it, where the results will be derived in detail, results properly proved and often many more astonishing facts get presented. So really this has been at best and introduction to why computing is not always easy.*

Consider the simplest uses of computers and technology and observe that beneath the surface there is a great deal of rather elaborate technology in play:

**Send a text message** Encoding your message as a radio signal. Ensuring that your radio waves do not unduly interfere with those of all the others in your neighbourgood. Finding a route – potentially well acound the world – for your message to be sent to the recipient. Checking that the destination phone is switched on and available among the millions of other phones. Playing a nice user-selected sound to indicate that it has arrived. Keeping track so that you can have this charged against the allocation for your account.

**Word process a letter** First an operating system that lets you open your word processor, with that using the hardware of your computer, screen, keyboard and all the rest. The major software of the word processor positioning characters on the screen and in the document, spacing everything nicely. Spelling and syntax checking with "helpful" suggestions about what you might have meant to write. The detailed design of the shapes of characters. Support where relevant for international lettering, mathematical notation, diagrams and pictures. A scheme able to capture all you have done in a format that can be stored on a file or transmitted over networks.

**Create a web-site** Most websites will be created not by direct manual laying out of HTML but by use of some helpful package which may be either local or online. This should consider all the things that word processor does, but beyond that it ought to take into account the fact that the generated site may be viewed on everything from small-screen phones to large monitors. Ideally the site should still make sense on a monochrome monitor ot for those who are vidually impaired. Certainly for commercial use navigation and accessibility need to allow for disabilities, so even colour schemes must be designed with red/green ambiguity avoided. Different browsers may not all behave the same way, so coping with that will also be important. Sometimes security of access will be important, so integration with some robust underpinning authentication will be important. Maintaining counts, logs, histories of hoe particular individuals use the site can also add to the work!

Observe that in each case the simple-feeling activity only feels simple because it is able to rely on a great deal of technology. Somebody has had to invent and build that technology, today there need to be teams who understand it in detail and can maintain it, while tomorrow it could be you working on the "next big thing" which will undoubtedly similarly built on top of all that exists today.

Beneath the technology lies some science and mathematics. Directing internet traffic through the network where that network will constantly change as links fail and are restored, and parts of it become swamped and over-busy and as satellites move in and out of range will involve large scale use of techniques that the "find a path through this maze" style question poses in fairly elementary algorithms courses. Proper encryption often depends on rather high powered results from number theory. Building faster computer chips involves a lot of device physics.

One of the most challenging practical challenges in software development is getting things correct. Any scheme that tries to help with that must start with a really careful understanding of exactly what the underlying computer will do and the consequences of that. A whole field of study has grown up around program verification. It is also very natural to want to get computers to deliver their results as rapidly as possible, and that leads to a tower of challenges where looking at the ultimate limits is important since it sets a target for practical work and makes it possible to avoid trying to achieve the impossible. So even the most extreme impractical work towards the very best can inform the real world in useful ways!

Steven Wolfram in his book “A New Kind of Science”[**?**] goes to great lengths to show that very very simple sets of rules can lead to astonishingly complicated behaviour. He particularly emphasises this with studies of cellular automata, but Chaos Theory[**?**] also tells us that arithmetic can, when iterated, behave in ways that are not at all obvious. From those two background one can read off a result that might be viewed as terrifying: the end results of running even rather short programs may be really hard to predict in detail.

Large programs are of course made up from a collection of smaller components, and if it is hard to tell what those building blocks might do it will be yet harder to understand the whole. Well natutally one hopes that all the modules going into a big application have clear and sensible intentions and that these have all been designed to cooperate to a useful end. But above some not terribly large threshold the reality is going to be that typos and mental slips to say nothing of misunderstandnings and over-optimism mean that the world is not quite that tidy. This means that full-scale computing will for the foreseeable future have awkward rough edges.

The lesson that emerges is that even when computing looks easy and even when it behaves well a lot of the time, getting a full and proper understanding ot it is difficult, but that difficult can be fascinatind and fun.