Computing made Difficult

A C Norman and others I hope

January 13, 2026

Chapter 1

Intrduction

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 twoo 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 ens up almost unimaginably complex. So the attitude of this point will underpin almost all of the sections here!

Seek a full analysis of what is going on, including identification and characterisation of best 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 π 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.

Investigate tasks where the underpinning theory is much deeper that one might have expected from the fairly simple problem

statement. I do not have nice test 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.

Work subject to constraints that render the obvious approaches unavailable. 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!

Attack problems where there is no complete solution to try to deal with interesting cases in a practical way. 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!

Apply wilfully perverse techniques to 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.

Set up challenges that are not realistic but that are 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!

Try to arrange software or algorithms or protocols that will remain relevant despite future changes in the computer landscape. 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.

Face up to 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 when software is designed or built. 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.

Understand graphical output. 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

1.1 Speculation about further chapters...

Suppose that the examples we cover are grouped by theme, here are some possibile clumps. I am not certain that all these will be winners but think there are enough that enough will be!

- Images - create and understand them.
- Geometry
- Deduction and boolean algebra
- General numerics
- Floating point arithmetic
- Other arithmetic
- Turing and Counter Machines
- Lambda-calculus and combinators
- recursive functions and types
- very high level descriptions of computation
- very low level techniques and issues
- Proving results about computation
- Ridiculous ways to compute things
- analysing and predicting costs
- Optimising space or time
- The tower of levels of abstraction
- Ingenious data structures and algorithms
- Puzzles
- Miscelaneous

Chapter 2

A case study

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

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, 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 α, β, π 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

[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[**?**]

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 jth 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 $N \rightarrow (N-1)/2$ 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 Cth 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

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

Chapter 4

Turing Machines

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 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-f4fc94eef5c`

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.

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) 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. Designing, understanding and using Turing machines that are really close to being mimimal can be huge fun however they can sometimes demand really ugly ways of encoding the data on their tape or suffer from needing unreasonably large numbers of steps to reach results.

A variant using just 2 states and 3 symbols exists and is characterised as "weakly universal". It will not even halt when it has completed its work and it also needs its tape initialised to a particular pattern in all regions beyond the input data, while standard machines will not read from the tape beyong the region that is explicitly data - it also tends to take a great many steps to get anywhere, while modestly larger machines are actually quite efficient.

Once one has demonstrated that a basic and fairly small Turing Machine can be use dto 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. The treatment of those tapes and those in a multi-tape Turing machine are closely analagous to one another, so that sort of TM provides a really solid model for analysing that sort of algorithm while avoiding the need to worry about detailed chatacteristics of any physical computer.

The overall message here is that if you really want toa 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.

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 sepatate 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

Lambda calculus and Combinators

Chapter 6

Primitive Recursion

Chapter 7

Solving a quadratic equation

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 1.8×10^{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 1.8×10^{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 4.94×10^{-3242} but once values get lower than 2.23×19^{-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 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

[1] Since it is working in binary internally it will actually be more like 0.01010101...
[2] 2^{-1917}

in every case where that is possible. Part of that might involve finding a scale factor σ and replacing a with σa and c with c/σ^3. Then a final result can be generated by multiplying or dividing the solutions to the scaled version by σ, 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 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.

[3]with σ a power of 2 to avoid introducing corruption through rounding

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 8

Turtle Graphics

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?

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 9

Ingenious Data Structures

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.

9.1 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 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

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

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.

9.2 What next?

I have not decided yet!

Chapter 10

Simple Pattern Matching

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. 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 `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. One needs some scheme to distinguish use of `(` as a literal character to be matched or a grouping marker, just as care is needed with the vertical bar. This pattern will match either `pet` or how you might treat one, i.e. `pat`.

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

- $P*$ – This is tge 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 ^ 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 o 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 two 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. 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 dreadfil in practise.

Consider a regular expression built up as follows and over an alphabet that consists of just the two letters `a` and `b`. If one sets up a fairly messy example of such an expression it might not be instantly obvious what inputs it will match. The question that gets tricky is "Is there any input at all that it will **not** match?". This may seem a frivolous and artificial question, because what on earth would be the point of a pattern that matched absolutely all input? Well actually it is not as stupid as it at first sounds! The components used to build up the regular expressiion may be modelling the behaviour of some machine or the progress through some program as it inspects the input. Succeeding in making match could correspond to successful processing, and so input that is not matched could amount to input data that causes the machine or program to fail. If that sounds a bit fanciful it is in fact almost exactly the insight used when considering how hard it will be to answer our question! And regular expressions and a certain simple class of machines or programs are very closely linked.

There are two major parts to understanding where the difficulty comes from. The first is to show the way in which a regular expression can be "compiled" into a nice program that will check for things that match it. The way of doing this in effect builds a sort of flowchart for the program, but it is not quite the normal sort. This one consists of blobs referred to as states and arrows fromn one to another where the arrows are laballed with characters from the input alphabet. One state is identified as where one starts, and

any of the states can be marked with a "success" tag. The concept is that from where it starts the arrows are followed and when as a result one is positioned in a success state that means that the text absorbed to date has been matched. So for instance one of the earlier very easy cases ends up as

```
                 /-a->\
(start)  -p-> (.)    (.) -t-> (success)
                 \-e->/
```

which should be very easy to follow and to understand how it relates to the original regular expression.

Expressions involving the * operator introduce loops in a rather straightforward way. The harder issue thart arises in the "`Bananananana`" example is that one ends up with a state where two arrows emerge from it each corresponding to the same symbol.

```
               /-a-> (success)
(start)  -B-> (.)
             / \-a-> (.) --|
             ^             v
             \----<--- n --|
```

This is referred to as being nondeterministic in that there is no clear way to know which `a` arrow to follow. The understanding is that the flowchart is deemed to recognise a string if at least one way of making those arbitrary-seeming transition leads to it ending up in a success node at the end of the input.

There are tricks that arrange that if one has a flowcharts corresponding to two regular expressions P and Q that one can construct ones for PQ, $P|Q$, $P\&Q$ and $P*$. The one for $!P$ is especially easy since all it has to do is swap which states are or are not tagged as success. Along the way these tricks can expand a nondeterministic scheme into one where each choice is unambiguous. As before the fine details of this are skipped here because none of it is difficult enough to match the book's title.

An important observation is that the flowchart you end up with has a finite number of nodes, and if there is an input that can be presented that is not going to be matched that means there must be a non-success node present and a path from the starting state to it. Viewing the flowchart as a sort of maze and checking for this is very much something that will not correspond to a hard program at least provided you are not worried too much about performance. So phrased the way we need it here we claim that the question as to whether there are any strings not matched by a given regular

expression is one where it is possible to write a program that answers it in a finite amount of time. To be specific, the program starts by elaborating the regular expression into a flowchart and it then maze-searches that to see if a non-succes state can be reached from the start position.

Well the normal language used for this would be to say that the question is "decidable" and can be resolved by mapping the regular expression onto a "deterministic finite state machine"[2] and checking if a "non-accepting state" is reachable from the start.

At this stage it may be useful to point out that a deterministic flowchart maps onto a really simple computer program that has a simple loop and a look-up table:

```
state = start_state
while more_input do
   state = transition_table[state, next_input_character]
return result_table[state]
```

and this fact is part of why this sort of pattern matching is so popular in software.

This has set up a problem that we can guarantee to solve, and despite the fact that setting up the `transition_table` as used above involves a bit of effort[3] it really does not look too bad. But now we can lead into the pay-off in terms of difficulty. By designing a regular expression cunningly and using all the extended notations we have it is possible to arrange that if the regular expression is made up of K symbols then the size of the flowchart and hence the time and effort needed to answer the initial question can end up greater than $2^{2^{\cdots^{2^2}}}$ where the tower of exponentiation has height K. This easily gets way beyond astronomical, as can be confirmed by reviewing the numbers that represent the side of the galaxy or known universe. On that basis it is probably fair to declare that obtaining the answer is difficult, even though one can have written a program that would deliver it if it could ever run to completion. And that it is absolutely known that the program would complete if give long enough – the only problem is that "long enough" might exceed the lifetime of the universe.

The assertion that something is that hard deserves some justification, and so here is a sketch of how it arises. All sorts of awkward details will be

[2]Well a non-determistic one might be as good, ane different authors us different names such as "finite state machine" or they abbreviate things to terms such as "NDA" for non-deterministic accepter. But the concept being discussed is the same whatever language is used when discussing it

[3]. . . and that good implementations seek ways of representing it in more compact ways than just a simple big rectabgular block

skimmed over!

Chapter 11

Lessons that have been learned

The comments here are just ACN rambling a bit more

I have just picke dup a copy of Wolfram's New Kind of Science for not a huge amount on eBay. That is a much bulkier volume than I had expected, not having checked it before. But what is maybe more terrifing is that Wolfram now has a "20 years on" book that he explains basically as "When I write ANKoS I thought it was a real breakthrough for the world, but 20 years on I see it is way better than even the extreme level of importance I saw in it back them".

I have only just started reading it and the main think that comes across is how utterly Wolfram wants to make a point that everybody from before the ancient Greeks has just skipped past the motherload of overwhelmingly important stuff that he and he alone has discovered. He is asserting that what he has discerned upends every scientific and many other disciplines totally and provides a way to discern the true nature of everything. Gosh it is amazing in that way. Wow – what a guy.

However for the purposes of what this book wants to do it is perhaps rather nice in that I think a major point he is wanting to hammer on is that something that follows very simple rules can have astonishingly complicated behaviour, and that this applies not to just one sort of "simple thing" but rather generally.

It is less clear to me (as yet) whether he can them do anything interesting with the complexity apart from show it off in loads of pictures. And some of us sort of believed that the not-too-bad equations of fluid dynamics could lead to very messy turbulence and not just smooth flow, that looking at multiplication and division dumped one into the quasi regularity of the distribution of prime numbers and great depth, and that the investigation simple problems like "boolean satisfiability" could tell you about all the other NP-complete problems. But that some of these are discrete and some continuous so he will

have to do a really merry dance to convince me that they are all the same even if all show complicated behaviours. But still reviewing all he has to say is liable to reveal a range of very find examples of things where the starting point is simple and the end-point really is not.

A lot of what he talks about is cellular automata.

A different thnk to consider is "solve your problem by first designing and building your computer, then the software stack...". This is of course just what people had to do in the 1940s. And indeed Babbage/Lovelace had a go there, and there was Konrad Zuse and his relay-based computer where one can even at least imagine constructing the relays....

In yet a different direction I think of steam engines. The pistons must slide nicely into their cylinders so they have to move in straight lines, but if you support them with sliding supports that migh introduce friction you do not like. So how do you make a pin-jointed linkage so that the end-point moves in exactly a straight line? One answer is Hart's Inversor. Now having invented that if you are a massochist you set up all the simultaneous equations that characterise the way that the other end of a rod that has a fixed pivot at one end lies on a circle, and so on. You then see if you can simplify and solve all those quations to prove that the key endpoint lies on a straight line. This is a horrid thing to try Groebner Bases on.