

Computing made Difficult

A C Norman and others I hope

January 7, 2026

Chapter 1

Introduction

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

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

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

In some of the examples included here even the more complicated way to solve the problem will in fact be reasonably easy to grasp (once you have seen why it is necessary to go to all the trouble involved). In others it will involve somewhat messy data-structures or mathematics – but the cases we have chosen are intended to make these possible to understand and appreciate. Finally there are cases where there is no known solution or (worse) where it is known that there is no perfect solution. In such cases grasping how it is possible to demonstrate 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 lookup, compile a program, create a file, fetch a web page or encrypt a message that is supposed to be kept secret there is a great deal of technology that happily most people can just take for granted. One can view this as rather like the situation with almost all technology from digital watches to aeroplanes – almost anybody can take advantage of them. Plenty of people will be able to present an overview of how and why they work. But the details end up almost unimaginably complex. So the attitude of this point will underpin almost all of the sections here!

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 mathematics 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 Rubik'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 Rubik'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 trusts 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 perfect 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 3.1415626₅359... and the value quoted should have been rounded up because of the 5359... 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 computations in either graphics or artificial intelligence areas, and so these rather pedantic issues of precision come much closer to practical reality.

Portability. For small programs that are only intended for use over a fairly

short period 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 one 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 resolution version of the action is of serious commercial importance. And that is a matter of real directly measurable performance where there are basically 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 the 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 than 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. We 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 magnificently 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 represents 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 passed 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

some 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 consideration of first encryption primitives (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 possible 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 special and really reliable tool should be asked about who created that tool and what basis there is for thinking it is perfect.

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 possible 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
- Miscellaneous

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 metal 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 properly – 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, \TeX ended up a great success 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 \TeX document by way of explanation. Within the single combined source the concept was that each fragment of code would be positioned adjacent 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 \TeX source 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 and \TeX became 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 project 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 energy!

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 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 than one might have expected. So in this chapter the emphasis will be on one particular way of looking at programming. 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 machines¹ 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. $0, 1, 2, \dots$. 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 is 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 interpreted as character data. To make this point as clear as possible, and computer file containing 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 characters (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

¹There 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 .

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 counting down in A until it is zero and of saving values in D shown above can be applied wherever it is necessary.

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 thought 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{verbatim}
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. And Provided you are drawing out the flowchart with the sub-blocks for known operations written as blobs rather than by revealing all the fundamental steps inside them this does not even make things seem much worse. Well that is unless your resolution falters and you consider what it means in terms of the number of steps taken to perform some calculation, so 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 array of N integers with values a_i for i from 0 to $N - 1$. Set up a string that starts with a 1 then has a_{N-1} zeros before another 1, then a_{N-2} zeros and so on down so that the string ends in a_0 zeros. So if the array was of length 3 and the values in it were 2, 4 and 6 we would set up 100000010000100. Now view this as the representation of a number in binary and view that number as an encoding of the state of the array. Well that step is simple - but it is now necessary to verify that the key operations of accessing the j th and updating it can be performed using a counter machine.

Actually those two operations are remarkably easy to arrange. 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 followed by k zeros. And that is $N \rightarrow 2^k(2N + 1)$. Again the computation there is simple enough arithmetic that the counter machine can be set up to do it.

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 an array 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 showing how to make things difficult that can not be seen as a problem. Those who are serious masochists 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 flowchart 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 flowchart based on an almost ultimately primitive model of computing will only be a constant factor larger than the natural one you started with.

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 entries in them. In fact the representation used behaves more like flexible lists than rigid arrays and slightly different ways of looking at 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 going 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 practical 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 abstraction 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 than 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 natural to feel that if one 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 than 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 upwards one would naturally 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 associated with counter machines, Consider all possible machines 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 halt node. So the answer there is 1. An alternative 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>

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 difficult 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 possibly 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 symbols 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 cell 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 possibly 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 machine 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 pattern. 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.

blah blah blah including the remark that Wolfram suggests that there is a TM with just 2 states and 3 symbols that is “weak universal” and one with 2 states and 5 symbols that is universal in the usual sense. And the 2 state 5 symbol version is surely very much within the scope of a variation on the LEGO build...

The other things to note are that (sharply in contrast to counter machines) Turing machines can solve problems “efficiently”. Well if a problem can be solved in $O(N)$ using ordinary programming it can be done in $O(N^2)$ on a TM.

Furthermore rather simple extensions and generalisations to TMs allow for yet better efficiency for many problems. One can consider a TM-like device with more than one tape or with just one tape by more than one read/write head. As a concrete example of what could be done with them, 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 analogous to one another, so that sort of TM provides a really solid model for analysing what can be done.

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 challenge 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 IEEE 754 and the program that is written will use a representation they call “binary64” which is commonly referred 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 have 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 furthermore 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 2.23×10^{-308} the representation starts to work with them at 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 getting 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

¹Since it is working in binary internally it will actually be more like $0.01010101\dots$
_{2²⁻¹⁹¹⁷}

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. Suppose that the value of $-b$ and $\sqrt{b^2 - 4ac}$ are rather similar in absolute value. 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 $1.0000123x^2 - 2 * x + 0.99991234 = 0$. Here $b^2 = 4.0000000$ and if it were calculated precisely $4ac = 3.999698555687128$. However it is only possible to keep the first 8 digits of the latter, so it will be taken to have the value 3.9996986 with the last digit that is kept rounded up. When these two values are subtracted one gets 0.0003014 which gets normalised up to 3.014×10^{-4} . But a truer result would have been 0.00033014443 to 8 significant digits. The discrepancy between these means that the result you end up with will only have around 4 correct digits rather than the 8 that you had hoped for. While we can all accept that the example shown here had awkward numbers deliberately chosen to give this severe cancellation of leading digits, there is no fundamental reason why such cases might not arise in real life.

Happily (for those who like to deliver accurate answers) 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 because 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.

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 calls for much more understanding than 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

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

with all cases, not just the ones you think about first!

Chapter 8

Lessons that have been learned

Bibliography