Computing made Difficult

A C Norman and others I hope

January 3, 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.

Demand the fastest (or most compact) solution that could ever ex-

ist. 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.

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

Bibliography