Computing made Difficult

A C Norman

December 18, 2025

Chapter 1

Introduction

To make something properly complicated the best thing to do is to start by showing that is starts off very simple, and so the intent here is that the first part of every chapter is showing how simple some aspect of computing is. These initial sections cover a range of topics that are to do with the broad area that people sometimes refer to as “computing”, “informatics”, “computer science” or mix up with “information technology”. The middle part of each chapter then sets a sequence of puzzles, projects or challenges relating to the topic involved. In general these do not need much particular prior knowledge of computers or how to use them — and in particular they do not start with any assumption that you have done a lot of programming before or have special deep knowledge about particular makes or models of computer. They just suppose that you have read and thought about the initial and specifically easy part of the chapter, but the intent is that despite this the material covered will provide glimpses into the areas that Computer Scientists spend some of their time thinking about, and some of the more important results that exist in that area. Each chapter ends with a few case studies of a much more substantial nature, and in at least a few cases these will introduce challanges that have not yet been solved at all. The material earlier in the chapter should provide a good basis for understanding what the problems concerned are and why they might be both interesting and important to solve, but any serious attack on them would call for some more background reading!

A citation here — I like Dijkstra[1] as a base for rigorous work with computers. When I have put in a load more citations scattered through the chapters this sample one can be removed!

Chapter 2

Register Machines

2.1 Flowcharts

If you needed to explain about the sorts of step-by-step methods that are commonly used with computers to solve problems you would quite probably start by drawing some *flowcharts*. These are a traditional way of explaining how to build up a sequence of tests and actions so that by threading a path through the chard you trace out steps that solve your problem. Flowcharts are not just used with computers — indeed they grew out of schemes that were to formalise and describe business proceduces in days before computers had been invented. They commonly appear in pamphlets explaining how to use or fix problems with all sorts of domestic equipment, and using then to describe things that everybody already understands quite well is possible if a bit ridiculous. Traditionally different shaped cells are written join up using arrows. There ought always to be a cell to start at and one labelled **stop**.[1]

```
(start}
  |   <-----------------------
  /\                 |re-read |
understood this? --------------
  \/
   |
(stop)
```

[2]

[1]Making the pictures for this article will be something I want to delaY until I have the basic word drafted — so I will use CRUDE text-art for now.

[2]I want some trivial and obvious but slightly jokey exampe of an irrelevant flowchart here

2.2 Examples

1. Multiply a number by a constant
2. Divide by a constant,checking if division is exact
3. Multiply two integers
4. Test equality of two values
5. note consequences of above for packing several integers into one register and for letting a register machine simulate actions common in full-scale programming systems

2.3 A Harder Problem

The Busy Beaver problem is ...

(take home problem is to seek the second-busiest beaver for a register machine with n states, n = 2,3,...

Chapter 3

Logic Programming

The threshold concept here is that of separating concern about "what" from worry about "how". Well if you view it as a mathematical notation (ie Horn Clauses) Prolog has no syntax to mention and its meaning is merely (ha ha) logic, so it is trivial. I could cobble up an ultimately cut-down pico-prolog in pretty short order for use here. The other key words here are *logic*, *consequence*, *satisfy* and *back-track*.

1. Typical Sunday newspaper puzzles along the lines of *who owns the zebra?*
2. The use of a predicate "in reverse", so that eg *add* can also be used as *find all pairs of integers that sum to this value.*
3. The interpretation of clauses as rows in a database and computations as queries, applied (say) to a family-tree database or some other structured data where chaining is interesting.
4. Some sort of bin-packing or jigsaw-style puzzle
5. 8 queens

Chapter 4

Finite Automata

This is a topic where the problem solving part may fit happily as a pencil and paper exercise rather than needing computers. The lecture part can introduce state machines. Key ideas about them (and which of these can fit into a presentation to novices is not quite clear yet) are the relationship between a machine and a language, non-determinism and ways of composing automate to make larger or different ones. The word *Chomsky* can be mentioned maybe. One perhaps attractive way to go is then to look at cellular automata — ie rows or grids of finite state machines.

1. Wolfram's "New kind of science" can be raided for examples

2. Miller's Periodic Forests of Stunted Trees may be fun to draw, but their relationship to the factors of polynomials over GF(2) may be beyond GCSE level!

3. The *firing squad* problem might be a good take-home one? And providing a download-from-web applet to animate things and help visualise may be good

4. One can invent automate design puzzles easily by specifying a regular language (eg via either a regular expression or a non-deterministic automata) and asking people to design a DFA for it. Asking for the smallest DFA raises the bar further.

Chapter 5

Regular Languages

This is obviously related to the previous topic, but is here because it is an area where there are few prerequisites, where results are maybe fun and where puzzles abound. It starts with an explanation of what regular expressions are (so easy!) and probably talks through the Pumping Lemma (that many will find it hard to absorb all at once) as the "gee-whiz, Computer Science has some real clever stuff in it" bit.

1. The puzzles/problems set can be styled after *here is a language described in words — can you find a regular expression for it or can you convince yourself that there is not one.*

2. Here is a two regular expressions. Can you produce regular expressions that are for the complement of each and the the intersection of the two languages?

3. Here is a regular expression, are there any strings that it does not match?

4. Here are two machines, or two regular expressions, or one of each. Do they accept the same language?

5. Especially if you permit intersection and complementation as extra constructors, try to find a regular expression written in n symbols where that shortest non-empty string it can accept is as long as possible. Eg `(aaa*)&(aaaaa)*` matches strings of lengths that are both multiplies of 3 and of 5, hence only multiples of 15. Can you do better [the answer is yes, if your RE is of length k you can make it match strings of length over $2^{2^{2^{\cdots 2^k}}}$ for any height tower of exponents you like, so this seemingly harmless problem is not quite elementary!]

Chapter 6

Phrase-structure grammars

Continuing in the same vein we can have

```
sentence ::= subject verb object
```

and the like as an introduction to context-free grammars. But issues of *is it context free?*, *is it ambiguous?* and *does it generate all but only all the sentences we wanted?* arise. We get a chance to mention terminal and non-terminal symbols and precedence.

1. Here is a baby grammar, is it ambiguous?

2. Create a grammar to exactly characterise the following...

3. Create a grammar that informally and approximately models the following real worldish situation

4. Start work on a grammar that describes your favourite computer language

Chapter 7

String re-writing

This starts with a world that is close to that of the cellular automata mentioned earlier. Your formalism is a set of rewrites such as

```
abaa -> axba
xxx -> ya
```

etc., where the re-writes are just of strings of characters, they may not preserve the length of the overall string and you keep applying them for so long as one of the left hand sides of a rule is present as a sub-string of your initial input. Key ideas are universal capability, non-determinacy and confluence and Knuth-Bendix style completion.

1. What is the consequence of these rewrites?
2. Create a set of rewrites to achieve this effect
3. Is the following set of rewrites confluent?
4. Extent these to make them confluent

Chapter 8

(metafont) The formation of characters

This is something I am not really familiar with, but we could unquestionably find a suitable expert! The fact that the shape of a letter *g* differs from **g** and that there are ways of describing it must provide plenty of scope for fun.

1. I am not going to plan exercises to go with this, but I am confident that there are plenty of opportunities and that the topic can transform people's view of what Computer Science concerns itself with. And while it is "graphic" it sends a message different from the "computing is about video games" one. It also gives a chance to chant the name *Knuth*.

Chapter 9

Iterated Functions

I do not have the exact agenda for this topic worked out, but if I list a bunch of things that fall under the general heading, or technology we understand that it relates to I hope you can see that there is scope for it turning into a nice day with both exposition and practical work. It may be that the spread of ideas I have bundled together here mean it is really two or more topics!

1. $f(2n) \to n, f(n) \to 3n+1$. What is the trajectory of the sequence of iterates of f starting at various starting points $k = 1, 2, \ldots$?

2. The logistic map $x \to \lambda x(1-x)$ for various values of the parameter λ and perhaps various starting values of x. This is all to do with the onset of chaos and cries out for practical work! And it is so much less technically messy than the Mandelbrot set and so much more really keyed into academic empirical study, as in what is the sequence of critical values for λ where the behaviour changes…

3. numerical iterations, and can you predict the value that they converge to? Newton Raphson to compute square roots and reciprocals etc.

4. We also know about the relationship between primitive recursive functions and recursive functions (the latter being iterations of the former subject to a minimisation operator) and we know about the Y operator that iterates a function.

5. Find a function f such that $f(f(x)) = x^2 - 2$.

Chapter 10

Functional Programming

ML is already a notably small language, but one could cut it down yet further so the only utterances available were

```
fun f 0 = value       fun f nil = value
fun f n = ...n...     fun f (a::b) = ...a...b...
```

plus multi-argument variants. All function definitions are global so you have no scope issues at all. So this is really just recursive functions! Via currying, map and a few more bits of cleverness I can start to write worthwhile but mind-bending programs using no more syntax than that! And providing infrastructure that could do just that is close to trivial.

1. append, reverse
2. Use of map, filter, fold
3. Check Larry's ML tickable problems and ones from earlier generations of the course to find further exercises — there are plenty of opportunities! Including ones that will burn out the brains of the smartest customers!

Chapter 11

Lazy Programming

Rather than cut-down ML this is thinking cut-down Miranda or Haskell. And it introduces a list comprehension notation, as in

```
{x^2 | x in {1,2,...}}
```

that is used in place of map and that copes with non-terminating lists. See all the early Miranda-related stuff from Turner for witty examples of what this can achieve.

1. Describe the list `{2,3,5,7,11,13,17,...}` of primes.
2. Interpret a list as the digits in a number expressed in some radix. So the in decimal the reciprocal of 7 is `{1,4,2,8,5,7,1,...}`. Do basic arithmetic on the infinite precision representation that this gives you.
3. Develop code to convert from one radix to another. What is the reciprocal of seven in octal?
4. Introduce the idea of a mixed radix where the ration between the weights of columns is not constant. Now `{1,1,1,1,1,...}` with radix `{1,2,3,4,...}` is a way of representing the mathematical constant e, and if you apply radix conversion you get that in decimal to unlimited precision.

Chapter 12

How the Clique problem and Hamiltonian Circuit problems are related

Complexity is all about transforming instances of problem A into instances of problem B, and very often both problems are easy to explain, often at least one will involve graphs (with vertices and edges) and often the transformation will be bob-obvious but easy once explained. This may make it a natural area for gee-whizz in the presentation and lead to scope for puzzling exercises finding or applying transformations. The fact that that in the end we are interested in polynomially bounded transformations does not need stressing.

1. Scan both existing Complexity notes here, corresponding note son the web for courses given elsewhere and all the textbooks and raid the most entertaining examples. Avoid all technical discussion of P, NP etc!

Chapter 13

Spreadsheets as serious computation

People may see spreadsheets early, but they are liable to be presented as tools for simple summing of columns, or perhaps for sorting rows. I.e. as just slightly animated mete tables of numbers that are not too interesting. They may also be seen as natural tools for a statistician who needs to reduce a pile of numbers to something. But through the eyes of a Computer Scientist they provide more opportunities!

1. Given a tabulation of an unknown function, taking successive differences lets you detect if it is in fact a polynomial and find its degree, and you can the very easily extrapolate it.

2. Given the tabulation of a function, put a small error into one value in the column. How can you find it? Take differences again and after a while it will be very visible!

3. Given a slowly converging sequence you may fear it will take a lot of work to evaluate enough terms that you can find its limit. But I have seen Geoff Miller use simple convergence accelerating techniques to compute ? to around 10 significant figures doing all the working by hand on the blackboard. I was impressed then! Spreadsheets are needed now for those whose mental arithmetic and accuracy do not match those displayed by JCPM.

4. The topics covered in Iterated Functions provide further possibilities here.

Chapter 14

Combinators

The simple (if not instantly obvious) rewrite system of combinators has long featured in our Foundations of Functional Programming course, and the fact that just two symbols (S and K) with simple rules for how they behave gives you a computationally universal system is so very neat. Some of us also like the fact that the polymorphic types one gives to S and K correspond to the axioms of propositional calculus, so well-typed combinations and theorems in logic are directly related! But that maybe is not GCSE-level!

1. I like the exam question that asks e.g. what S S S S S ... (for n copies of S) would reduce to. If might even be accessible in a single-day session!
2. What is S K K x
3. if I = S K K, what is S I I x
4. what about (S I I) (S I I)
5. we are by now amazingly close to being able to define the fix-point operator Y
6. Could one get to Church numerals?

Chapter 15

Mathematical logic and Hardware logic

Via simulators it should be feasible to do some digital electronics. Showing the linke between logic in the mathematical sense and the gates used in computers could represent fun. I am going to suggest that a first session concentrates solely on pure logic (i.e. No flip-flops or other “state”).

1. Gate minimisation via Karnaugh maps
2. Conversion of a 4-wire representation on 0-9 to drivers for a 7-seg display
3. Binary numbers and a half-adder
4. Ripple carry in an adder
5. Fast look-ahead carry blocks

Chapter 16

Analog aspects of digital electronics

A different sort of simulation would be needed here, or even access to real breadboards. But a point to stress is that Maplin's one-off prices (easy to buy if not the cheapest) put nand gates and inverters at pence per gate, so experimentation is not an expensive hobby!

1. Making 3 inverters into a ring oscillator
2. Race conditions
3. Hazards
4. Slowly changing inputs, and making a schmitt trigger using a bit of positive feedback, thereby making things safe and debounced
5. Cross-talk from a driven line into a gate input that is left open circuit?
6. Whatever else some hardware people like to propose.

Chapter 17

Lisp (Scheme)

Ages and ages ago I prepared a range of illustrations of how Lisp (at that time on the BBC micro) could be used to model small versions of a range of applications, including

1. text adventure dungeon
2. tiny mini-compiler
3. evaluation of expressions
4. tree-sort
5. bignum arithmetic, e.g. with the number a million represented as the list (0 0 0 0 0 0 1) with the least significant digit first.
6. an *animal guessing game*
7. route-finding

Chapter 18

PIC microcontrollers

The 18F2550 PIC microcontroller costs around a couple of quid on ebay and presumably at educational prices in bulk they are “cheap as chips;;. If we could stick one on a board with a USB socket, 2 push-buttons, 2 leds and a header of some sort so that most pins were just available for off board use (plus a 4-pin connector for use with a dedicated programmer) we could put a USB bootloader in and what you then have is something much like the US$25 commercial offering you can see at `http://www.sparkfun.com/commerce/product_info.php?produc`
If this could be offered to participants for 10 quid (which almost sounds feasible) and we hand a bunch here it could form the basis for in introduction to a wide range of jolly things. The pragmatics here may be harder than for the earlier examples I have given but the scope opened up is huge.

1. Machine code programming of a PIC, with all the good games one can play there
2. interfacing the host computer to other devices, both analog and digital

Chapter 19

Term-rewrites for Algebra

First try $D(x + y) \to Dx + Dy$ etc to describe differentiation and try to build on that to see how well you can simplify the result of (for instance) $DDDDD(1/x)$.

Then try $q^2 \to 2$ and wonder if it gives you a way to computer with the square root of 2. If you assume you have polynomial simplification and introduce rewrites like this then with one simple rewrite life is easy. With two (for univariate polynomials) the key result is that you end up as if you have found the GDD of those two rewrites, while for more than 2 you need Groebner bases.

Chapter 20

Additional Ideas

I am certain that there is also scope for activities related to particular concepts in algorithms (e.g. the idea of the greedy approach etc. etc.) and I am also certain that some of the proposals I have shown above will turn out to be hard to sort out all the final details and make fully accessible to a young audience. But I have given about twice as many suggestions as a year-long series would need!

Chapter 21

Conclusions

Regardless of whether my suggestions here could turn into a series of master-classes does anybody have the stamina and energy to turn my list of suggestions 1-17 above into 17 chapters in a book on introductory and somewhat recreational computer science with me? My bet is that if I tried on my own I would stall, but surely each chapter above would easily turn into 5-10 pages (let's say 6 pages of words and 4 of pictures and listings and sample output) which adds up to 170 pages and a length that would not feel too silly?

Bibliography

[1] E. W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, 1976.

Index