An Introduction to Algorithms

Professor Richard Harvey FBCS

26 October 2020

This is the first in series of lectures called “Great Ideas from Computer Science”. The idea behind the series is to pick-up ideas which are commonplace in IT and Computing and to, as we now say, “mainstream” them. This is a challenge as the words have already leached into everyday society, so these lectures will often start with a small linguistic battle in which I define the thing I am talking about and the audience will scream “but I am using that word to mean something different.” Well unlike Humpty-Dumpty[1] I shall at least try to give a reason for my definitions.

This lecture is about algorithms, and at least in the UK, this has been the year of algorithms – there have been defective algorithms for the computation of high-school students exams; there have been track-and-trace algorithms and there has been the scrapping of “racist” algorithms for UK visa decisions. The press uses the words, *computer system*, *program*[2] and *algorithm* interchangeably. As we shall see, compared to some of the more egregious errors found in the British press, this is easier to forgive, but an algorithm might be informally as:

> *a finite number of exact, unambiguous instructions that (i) always stops after a finite number of steps; (ii) produces the correct answer; (iii) could be followed by a human without any aids except pencil and paper and (iv) one needs to only follow the instructions rigorously to succeed (no ingenuity is required).*

Now, although this is not a very precise definition, we can see it is precise enough to rule out certain types of things from being called an algorithm. A cooking recipe is not an algorithm (ingenuity required) and a patent is not an algorithm (too descriptive). A mathematical formula is an algorithm (but algorithms cover things not easily written down in a single formula) and, like formulae, algorithms can be described in terms of other algorithms.

The origins of the word “algorithm” are not a mystery – it is a corruption of “Algorithmi” which was a latinsation of Al-Khwarizmi. Muhammd Al-Khwarizmi was one of the great Persian mathematicians who formalised algebra and, among other things, “invented[3]” what we know now as Arabic numerals (also formerly known as algorithms). Although a numbering system is not an algorithm, there are, as we shall see later, clear connections between a system labelling of quantities and what we know now as an algorithm and hence in around the 1800s the term seems to morphed from meaning a numbering system to a formal recipe for doing something.

More precisely, we are going to insist that an algorithm is what is called an *exact method* [1] which is just a fancy way of linking algorithms to systems for solving logic problems. However, an algorithm

---

[1] *“When I use a word,” Humpty Dumpty said in rather a scornful tone, “it means just what I choose it to mean — neither more nor less.”*
“The question is,” said Alice, “whether you can make words mean so many different things.”
“The question is,” said Humpty Dumpty, “which is to be master – – that’s all.”

[2] *Since Gresham College is in Europe, I should note that the word “program” which is a set of instructions for a computer to follow is not the same as a “programme” which is a schedule. Americans use the same word for both senses. New Zealanders tend to follow European practice. Australians cannot decide.*

[3] *By which I mean he nicked the idea from Indian mathematicians!*

is more general than a program – a program is often a list of instructions[4] that will work with a particular computer. An algorithm is usually stated in a way that is more general. This is a practice that has quiet ancient origins and dates from the time when the word "computer" meant a human who was undertaking the computing. Devising algorithms is the creative part of computer science and is a mental activity similar to solving puzzles or mathematical problems. Given that devising algorithms is quite tricky and you might think that each new problem requires a fresh algorithm, there seems to be a human bottleneck which is algorithm design.

Fortunately, many problems are similar so one can rely on "well known" algorithms[5]. For example, many systems require us to sort things: the Excel "sort by column" function; or the gocompare.com "sort by price"; or the less reliable ebay "sort by review score" option. Given sorting is such a common task for computers, there are indeed common algorithms. One of the popular ones at the moment is called Timsort (named after Tim Peters who implemented it for the programming language Python in 2002). Timsort is also used in Android, and in some more esoteric languages including Octave, Swift and Rust.

What differentiates a "good" from a "bad" algorithm is complexity. Complexity might mean time complexity, the number of steps to do something, or space complexity, the amount of storage required to run. You will hear people speak of $O(N)$ or "order N" algorithms. This means that if there are N things to process then the algorithm would take N steps to do so. Some algorithms, such as sort, depend on the data, so if the data are already ordered then sorting is obviously order N since we just need to run through the list of N items once and we are finished. So, in such cases, it is usual to speak of worst-case complexity or mean complexity. In the lecture I picked a particularly bad sorting algorithm known as bubble sort, and showed that it has a worst case complexity of $O(N^2)$.

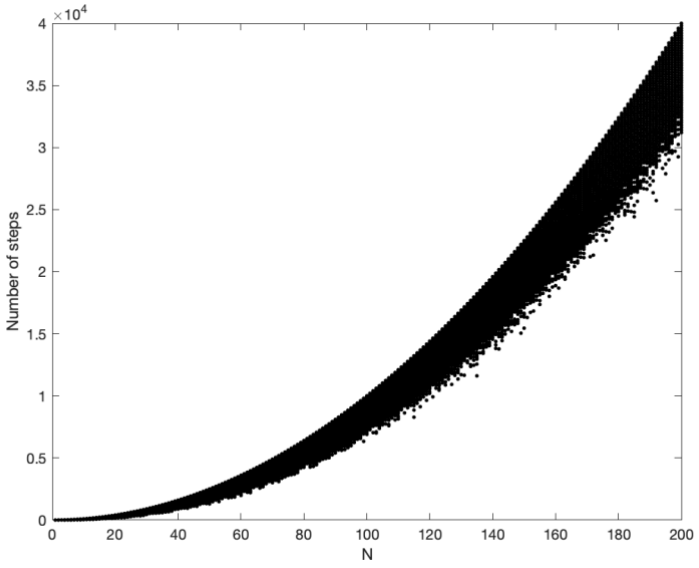


Figure 1: *Number of steps required to complete a bubble sort of a list of numbers 1 to N in random order.*

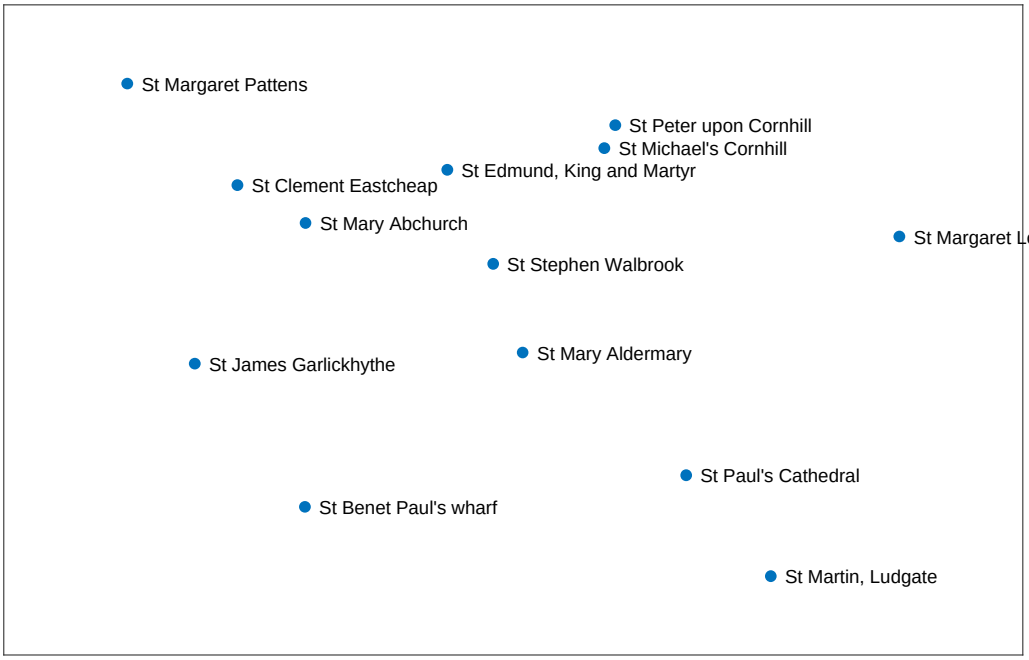
Figure 1 shows attempts to sort lists of N items – each dot is sort and I tried 1000 random lists for each N – the list is "bound" from above by N^2. Thus, bubble sort has a worst-case time complexity

[4] But not always – function and declarative languages merely state what things that want to be true and the computer goes away and tries to solve it.

[5] I once worked with a mathematician who was presenting a paper of ours. "This is very well known," he said while referring to a standard result we had used, "By which I mean, there are three or four people in the world who know this!"

of N^2. Given there are better sorting algorithms, there are no reasons to use Bubble sort[6]. That said, polynomial complexity (N^2 is an example of polynomial complexity) is not too bad since are plenty of algorithms which appear to have exponential or even super-exponential complexity. Formalised, this is known as the Cobham-Edmonds [2] thesis – any algorithm which has polynomial complexity is manageable on real computers whereas anything greater is intractable. I sometimes think the Cobham-Edmonds thesis is the most misunderstood thesis on the planet since it is frequently taken to mean that any algorithm that has worse than polynomial complexity cannot be, or should not be, programmed. Nothing could be further than the truth – there are a great number of algorithms that have worse than polynomial complexity – Cobham-Edmonds warn us that, if we attempt to use those algorithms on larger problems we might expect the computation to take an unfeasibly long time or use an unfeasibly large amount of storage[7].

One of the classic non-polynomial algorithms is known as the Travelling Salesperson Problem or TSP. Actually, it is has become known as TSP since Julia Robinson used the term in 1949 [3]. Before that it was known as "the postal messenger problem" or the "Hamiltonian Game." Let's consider the problem of Sir Christopher Wren, a former Gresham Professor, wishing to visit all fifty-two of his London churches[8]. To simplify matters let's consider a tour of just the thirteen churches that are currently still standing unmodified: St Benet Paul's wharf; St Clement Eastcheap; St Edmund, King and Martyr; St James Garlickhythe; St Margaret Lothbury; St Margaret Pattens; St Martin, Ludgate; St Mary Abchurch; St Mary Aldermary; St Michael's Cornhill; St Paul's Cathedral; St Peter upon Cornhill and Sr Stephen Walbrook. The closest of these churches are 104m apart, according to Google Maps walking directions, and the farthest 2948 m.



[6] *Indeed, there are serious academic papers saying that we should never teach "bubble sort" as it seems bizarrely memorable.*

[7] *Incidentally, if there are any complexity theorists reading this, I have an informal theory that the number of steps and the amount of memory are interchangeable. One of the more pleasurable experiences of my life was coming across a paper by my father, who was an electronics engineer struggling to "fit" one of the classic algorithms, the Cooley-Tukey Fast Fourier Transform (FFT) into some hardware. He rejigged the Cooley-Tukey algorithm by increase time-complexity but reduce space-complexity [8]. To solve the problem my recollection is that he deliberately increased the number of multiplications to constrain the amount of storage required.*

[8] I'm classifying St Paul's as a church.

Figure 2: *Thirteen of Wren's surviving churches plotted as their latitude and longitude. Each path has a* weight *(not shown here) that is mean of the Google walking distances from and to the point.*

Sir Christopher is standing in St Paul's and is struggling to translate into Latin "if you seek his monument – look around you" so he decides to take tour around his 11 special churches[9] to gain inspiration. He decides to visit each church only once and, being a parsimonious person, he wants the shortest route landing back at St Paul's. This is TSP.

In Figure 2 I have illustrated this with a *graph*. Computer Scientists generally use the word graph to mean a set of *nodes* (Churches in this instance) connected with *edges*. From St Paul's he could travel to one of twelve churches, and from the next church we could visit eleven possibilities (St Paul's and one other having been now visited) and from then ten others and so on. Thus, the number of possibilities is

$$12\times11\times10\times9\times8\times7\times6\times5\times4\times3\times2\times1 = 479{,}001{,}600$$

This calculation $N\times(N\text{-}1)\times(N\text{-}2)\ldots\times2\times1$ arises so frequently in combinatorics that we give a special symbol $N!$ which we call N-factorial or (factorial(N)). Well although Sir Christopher Wren might be daunted by searching through 480,000 possible routes[10], this is something we can do fairly swiftly on a modern computer. Let's say I can do it in a microsecond (1×10^{-6} s). How long would the algorithm take if Sir Christopher decided on a more strenuous tour of all fifty-two of his choices. Well starting from St Paul's there are 51! routes. Now we can see why the factorial function is indicated by an exclamation mark. A naïve person might assume that a 52-element problem was around four times trickier that the 13 element which we solved in a microsecond. In fact, using the same computer it would take us 5×10^{32} years to solve the larger problem!

In the above argument I checked every combination – this is a method called "brute force" or "exhaustive search". Some people would argue that brute force is so naïve it is not an algorithm at all. In fact, when I solved the problem on my local computer I used the Google optimisation toolbox (OR-Tools) [4] which tries a variety of algorithms. For small problems I imagine it uses the Bellman Held-Karp algorithm which dates from pre-history (1962). It has order complexity of $O(N^2 2^N)$. At this order complexity, if solving the 13-church problem took a microsecond then solving the full problem would take 102 days of computing time – rather longer than the walk itself but manageable on a supercomputer I expect. For large problems, OR-Tools uses the best-known approximate algorithm which is called the Christofides-Serdyukov algorithm which also dates from prehistory (or 1976). The algorithm is useful because it has been proved to produce solutions that are no worse than 1.5 times of the optimal tour and it has order complexity of $O(N^4)$ which means solving the 52 church problem would take a miserly 256 microseconds.

The situation with TSP is not untypical – for large problems we can have either an exact solution but have to wait for an unfeasible length of time, or we can have an approximate solution in realistic time. Good approximate algorithms are bounded in their errors, so we know the maximum badness of the solution. Sometimes those bounds have to be computed – on the slides I show how a different approximate algorithm, the nearest neighbour tour, is also an upper bound. In other words, if you ever solve a TSP and your route is longer than nearest neighbour (which consists of always walking to the nearest church that you have not yet visited) then your route is not a good route.

TSP has proven to be rather resistant to new ideas – it is pretty unusual that the best algorithms known today are at least 40 years old. At least that was true until a few months ago when a new algorithm was published which appears to have the same order complexity as Christofides-

[9] Like all Gresham Professors he has the power of foresight so he knows which thirteen churches will survive the remodelling efforts of Londoners and the Luftwaffe.

[10] And you can halve that if you do not care if you traverse the route clockwise or anti-clockwise.

Serdyukov but is closer to optimum by a smaller amount (specifically the authors claim that the tour is better than 3/2 - ε. Mathematicians usually use to mean a very small number. In this case it is proven to be 10^{-36}. Such is progress on TSP …modest.

So far, I have introduced a couple of algorithms and the idea of complexity which is not really about how space and algorithm uses nor about how many steps it takes, but about how space and time scale as the problem grows. But I skipped a whole host of practical issues such as standard algorithms and how we should write down algorithms. As we shall see later complexity theorists rely on the fact that the solution to problems often boil down to implementing standard algorithms. We don't have enough time to list all the "standard" algorithms – the Wikipedia list is huge and in whichever field you are working everyone always claims there is a standard well-known algorithm which turns out to be completely unknown to anyone else! More annoyingly there are not even wide agreement on standard ways of writing these things down. Two methods that are quite widely adopted however are Structured English and flowcharts.

Flowcharts are the more venerable of the algorithm description methods and seem to date from Frank and Lillian Gibreth who were time-and-motion specialists who developed flow charts to describe human work processes in around 1921[11]. The famous computer scientist John von Neumann [5] appears to have used a chart but they look closer to what we would now call signal flow graphs and focussed on mathematical functions so I'm discounting that attempt. The earliest computer flow chart that resembles modern usage is in the papers of Grace Hopper, a computer pioneer based in the US Navy. Hopper includes in her notes a flow chart from Margery K League who was of the early programmers of BINAC which the USA's first stored program computer. Flowcharts were very popular up the early eighties and many of us had little plastic stencils that allowed us to draw the things. The in a bizarre twist of fate along came cheap computers which were widely adopted in the home and offices but the first generation of these "personal computers" were terrible at graphics so graphical representations such as flowcharts were killed by the computer. Their replacements were English descriptions of algorithms "…do this" … "and then do that" or what is sometimes called *pseudocode*. Neither is very well specified, but algorithms specialists seem to muddle along without worrying about this paradoxical situation – specialists who spend their lives worrying about precise definitions of classes of problem cannot seem to devise precise definitions for their own language. The situation is in stark contrast to the business of programming where the languages are not only well specified but there a meta-languages, such as Z, which allow one to specify systems and languages.

Algorithmic specialists see close connections between algorithms and algebra. It is rather beyond the scope of this lecture to draw those connections but a key development in the formalisation of what we mean by computer was made by Alonzo Church, who developed a general language for describing computation (and algebra) known as lambda calculus, and his student Alan Turing, who conceived an imaginary machine known as a Turing machine [6]. Turing devised several machines. The simplest, called an a-machine, had a table of instructions that allowed it manipulate symbols on an infinite strip of tape. Turing then generalised this idea to form the universal machine in which the rules were themselves printed on the tape. This is the origin of the stored program computer and according to the Church-Turing thesis the problems that the universal Turing machine can solve are exactly those solvable by an algorithm. Any algorithm is equivalent to a Turing machine and visa versa. Given that Turing was not actually at this moment intending to invent the whole of computer science, his paper was about a related problem posed by David Hilbert which is known as the decidability problem, which is given a valid formula in a logical system can one decide if it can be derived from the axioms, then, given that, the paper is astonishing. The answer to the decidability problem, by the way, turned out to be a surprising "No".

[11] To my annoyance I cannot find any versions of their original charts.

The Turing machine was an important step in being more precise about complexity of algorithms. By algorithm we mean something that could be executed on a universal Turing machine. Another important step is being more precise about the type of problem we are solving, the *class* of problems. One common class is called decision problems – those that have a yes/no answer. Decision problems often have a lower complexity than their full-fat alternative. Thus, while TSP might have non-polynomial complexity, the decision variant, which is "is there a tour which has a distance less than some number" is …well we will look at that in a moment.

Earlier I mentioned the set **P**, which I blandly defined as algorithms that run in polynomial time. More precisely I should have said it is the set of all decision problems that can be solved by a universal Turing machine in polynomial time.

What about Sudoku? Figure 3 shows a Sudoku grid.

	1	2		3	4	5	6	7
	3	4	5		6	1	8	2
		1		5	8	2		6
		8	6					1
	2				7		5	
		3	7		5		2	8
	8			6		7		
2		7		8	3	6	1	5

Figure 3: *A minimal Sudoko grid with 40 clues*[12]*.*

The idea is to fill in the blank squares such that each row and column contains a permutation of [1,2,3,4,5,6,7,8,] and each 3 by 3 grid also contains all the 1…9 digits with no repeats. You might imagine that finding a solution to Sudoku is in general a bad problem (imagine solving a much bigger puzzle and how much harder it would get as the grid expands). But what about verifying a solution? That sounds much easier – essentially one has to iterate along the rows (*N* steps) *N* times =N^2 and then down the columns (N^2 again) and then to check each square (9 *N*). So, the overall complexity of verification is a manageable O(N^2). This leads us to a new class of problems **NP**. Here **N** stands for Non-deterministic[13] but we shall merely think of **NP** as the class of problems that have solutions or guesses at solutions which can be verified in polynomial time. Now obviously **P** is a subset of **NP** but here we come to one of the cruxes of modern computer science – there seem to be an awful lot of problems in **NP** (TSP for instance) that, so far, we have not been able to find algorithms for the decision problem in **P**. There are occasional victories, the famous 2004 paper "Primes is in P" [7] was pretty staggering when it came out but mostly we have this collection of knotty problems which we can verify in polynomial time but we cannot solve in polynomial time.

This leads us to the famous question in algorithmic complexity. **P** = **NP**? The Cley Mathematics institute has a $1M prize for the first person to resolve the question but the money does not seem to have led us to a solution. If **P**=**NP** then the we would conclude that with more ingenuity we might be able to find manageable algorithms for a variety of problems. For many this would be excellent

[12] A minimal Sudoku grid is one in which the removal of any clue would make it ambiguous. This is a 40-clue grid which the currently largest known number of clues in a minimal grid.
[13] Another annoying use of terminology from the complexity boffins.

but, as we shall see in a moment there are a number of systems that are gambling that **P≠NP**. Indeed **P≠NP** is what most theorists believe to be true.

So far, I have implied that algorithms that take a long time to solve something are a bad thing. However, there is at least one area, cryptography, where huge sums of money are riding on the idea that the verification algorithm is quick but the construction algorithm is very complex. An early cypher system, the Merkle-Hellman knapsack cryptosystem, used a standard problem known as a knapsack problem. The knapsack refers a hiker or possibly a thief who for various reasons wishes to maximise the value of items in their knapsack but the knapsack can hold only a fixed volume of items – imagine going into a pub and making an order for £13.48 without specifying what you want. The bartender must work out how many pints, peanuts, whiskies and whatnot are needed to make up exactly the right total. Not easy. However, the verification problem is trivial – given the items we merely sum up their values. The Merkle-Hellman knapsack problem took a message of binary digits (b_1,b_2,b_3,…,b_n) and set of secret "keys" which were positive integers known to the sender and the receiver (k_1,k_2,k_3,..). If b_i is 1 then we add in key, k_i if b_i is zero then we add zero. The coded message is *S* a sum of unknown keys, indeed an unknown number of unknown keys. Imagine eavesdropping a message of 1034 – how many ways can you *partition S* into its components? So here we have a system where the receiver merely has to verify a number – polynomial in time whereas an eavesdropper has to solve a horrendous problem. As it happens there was a vulnerability in the way the keys were chosen in the Merkle-Hellman code and it was shown in 1984 that it could be cracked in polynomial time. Needless to say, if **P**=**NP** and if the proof is constructive, then much of cryptography will collapse and the million dollar prize will be negligible compared to the societal costs – kids in their bedrooms will be able to crack ecommerce encryption.

At this point some clever clogs usually points out that so far we have been talking about only a very restricted class of problems: **P** is merely the class of decision problems that can be solved in polynomial time on a universal Turing machine. What about the other classes? These form what Scott Aaronson calls the "Complexity Zoo" – 417 classes and counting. To an outsider, the map of these classes is very intricate, not to say messy, and jam-packed with unproven theorems[14]. Possibly it is tricky because we know that the business of proving something such as **P**=**NP** is itself an **NP** problem.

In this lecture we have refined what we mean by algorithm and we have noted that a precise definition of algorithms leads into the fascinating topic of complexity which connects with algebra and fundamental theorems in mathematics. But algorithms are only a part of computer science and to create programs we need to consider not only the methods for manipulating data but also the way we store data. Computer scientists call those things data structures and that is the topic of the next lecture.

References

1. *Language, Proof and Logic 2nd Edition*, David Barker-Plummer, Jon Barwise and John Etchemendy, 2011, Center for the Study of Language and Information, ISBN-13: 978-1575866321

2. *The Intrinsic Computational Difficulty of Functions*, by Alan Cobham. Proc. of the 1964 International Congress for Logic, Methodology, and the Philosophy of Science, North-Holland Publising Co. Amesterdam, pp 24-30.

3. *On the Hamiltonian Game (The Travelling Salesman Problem),* Julia Robinson, Dec 1949, Report RM-303, The Rand Corporation, Santa Monica

[14] For example the no-one knows if **co-NP** (the class of decision problems in which we must verify if something is a wrong answer) is identical to **NP**.

4. *OR-Tools*, Version 7,2, Laurent Perron and Vincent Furnon, 19th July 2019, Google, https://developers.google.com/optimization/

5. *Planning and coding of problems for an electronic computing instrument, Report on the Mathematical and Logical aspects of Electronic Coding Instrument Part II*, Herman H Goldstine and John von Neumann, Institute for Advanced Study, Princeton, New Jersey, 1947.

6. *On computable numbers, with an application to the Entscheidungsproblem*, Nov 1963, Proceedings of the London Mathematical Society, s2-42: 230-265. doi:10.1112/plms/s2-42.1.230

7. *PRIMES is in* ***P***, Manindra Agrawal, Neeray Kayak and Nitin Saxena, Annals of Mathematics, **160**, (2004), pp 781—793.

8. *Fast Fourier Transform and its implementation*, D. Butler and G Harvey, Proceedings Signal Processing Conference, Loughborough, Aug 1972, Academic Press, pp 165-181.