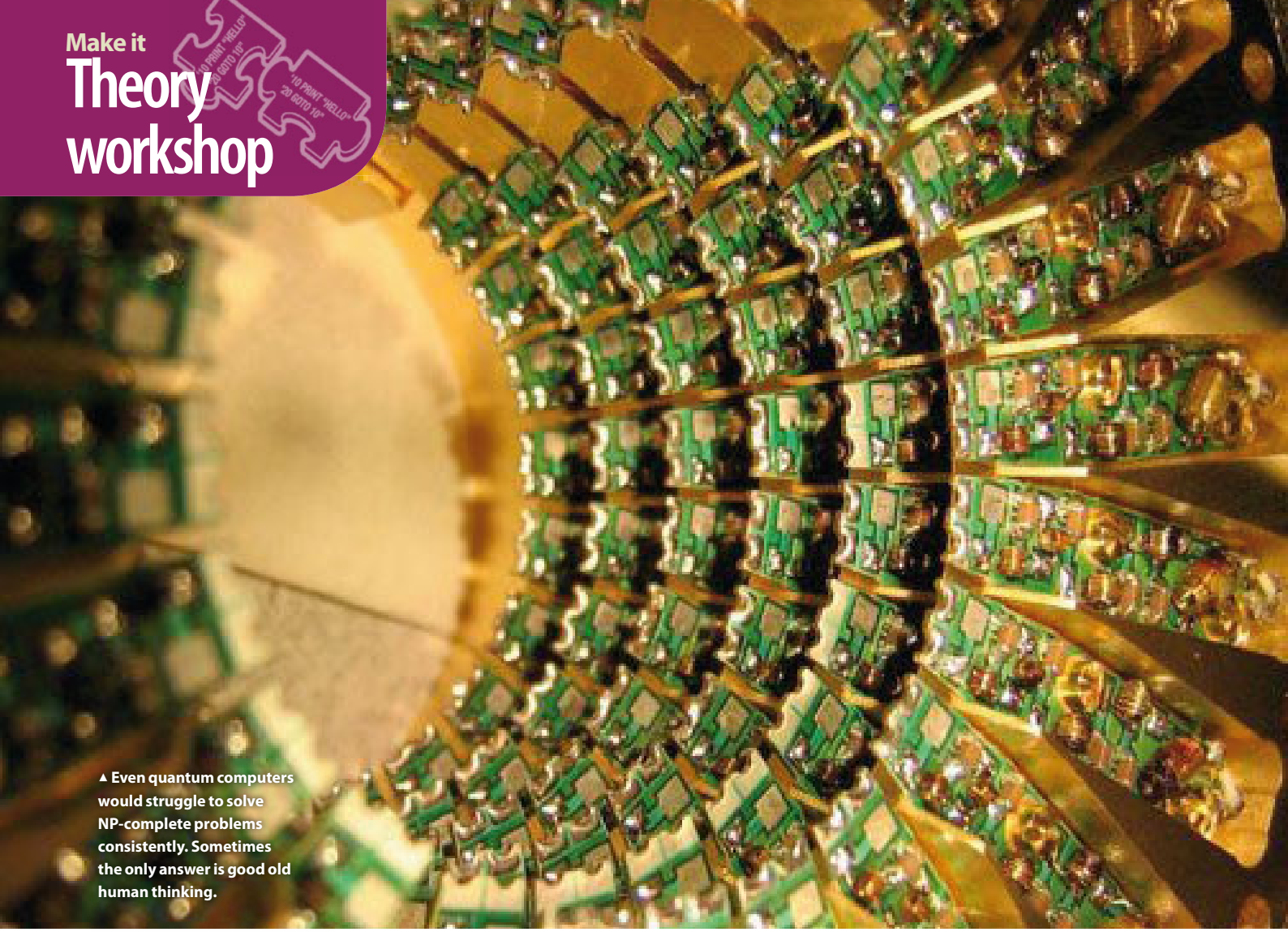


▲ Even quantum computers would struggle to solve NP-complete problems consistently. Sometimes the only answer is good old human thinking.

Answering difficult questions

When is the solution to a problem intractable instead of being merely hard? And how do you then solve it? You'll find the answers right here

Programming can be thought of as the art of providing solutions to problems. Spend any time developing applications and you begin to see problems for which finding the solution takes a long time – perhaps too long. These types of problems are of a different quality than the normal problems for which we have exact algorithms: they're known as NP-complete problems.

Intractable problems

Open any computer science book and you'll see algorithms by the bucketload. Do you want to store and retrieve items by key in key order? Have a go at using a sorted array, or a binary search tree, or a skip list, or a red-black tree, or a splay tree, or one of several other data structures. Do you want to sort data? Try bubble sort, shaker sort, insertion sort, selection sort, heapsort, merge sort, quicksort, or a few of the other more esoteric sorts that are available. How about compressing some data? You can use Huffman encoding, splay encoding or dictionary compression, including LZ77, Zip and many others. Encryption? There are lots of options.

All of these algorithms have one defining quality: we can analyse them and draw certain conclusions about their execution speed and memory requirements.

For instance, let's take the sorting problem as an example. With merge sort we can say that its worst-case scenario is as good as its average-case scenario (and its best-case scenario): that it'll take a period of time that's proportional to the number of items multiplied by the log of the number of items.

We say that the run time is $O(n.\log n)$, avoiding a bit of mathematical rigour as to what the big-Oh notation means. So, if it takes time t milliseconds to sort 100 items, that means it'll take roughly $150t$ milliseconds to sort 10,000.

Bubble sort, the sort everyone seems to learn first (for no discernible reason at all – it's a quite awful algorithm), is $O(n^2)$; that is, the time taken will be proportional to the square of the number of items. So if it takes t milliseconds to sort 100 items, it'll take $10{,}000t$ milliseconds to sort 10,000 items.

However, that's the average- and worst-case scenario. For the best-case scenario – there's nothing to sort since the items are already sorted – bubble sort is linear. In other words, sorting 10,000 items will only take 100 times longer than sorting 100.

Nevertheless, using bubble sort is still feasible. We could be sure that we wouldn't have that many items in our application or ▸

► the items would be mostly sorted and so the slightly easier implementation might be the way to go.

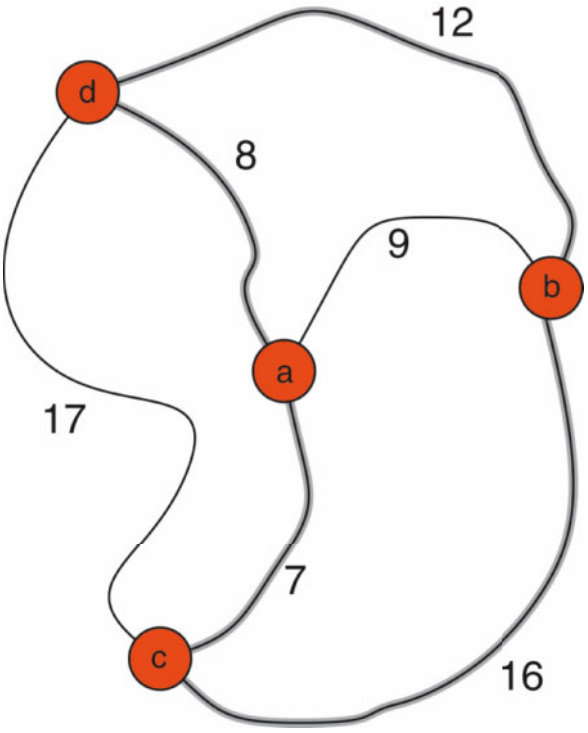
A way to go

Here's a thought experiment for you. Another way of sorting a set of items would be to generate all permutations of the items and select the one permutation that is sorted. Let's call this PermSort. How long would that take?

To calculate the permutations, we would in effect choose one item to be first (out of n, the number of items, so there are n possible choices to be first), one item to be second (n-1 possible items), one to be third (n-2 possible items) and so on. The total number of permutations we'd have to generate would be $n * (n\text{-}1) * (n\text{-}2) * \ldots * 2 * 1$, or $n!$ (n factorial).

This number gets staggeringly huge very quickly. If it took one nanosecond (a millionth of a second) to generate the next permutation and check it to be sorted – and note that I haven't described *how* to do this, so I'm almost certainly being way too optimistic – then it would take 3.6 seconds to sort 10 items.

To sort 100 items by PermSort using my highly efficient permutation generator? 3 * 10,143 years. The current age of the universe is 13.75 * 109 years. To put it mildly, if all we had for sorting was this hypothetical sorting algorithm, we'd have all got well used to unsorted lists by now. But the point remains. PermSort is guaranteed to provide a sorted list, but we don't have to use it because we've discovered better (faster) algorithms for sorting. PermSort is called an exponential algorithm: the time taken is exponential in the number of items.



▲ A simple travelling salesman problem with four cities.

Let's take a look at another issue: the travelling salesman problem, usually known as TSP. This problem supposes a certain number of cities that a salesman has to visit and the length of the roads that link those cities (if there's no road between two cities, we can suppose that the length of the road between them is some very large number).

Calculating the shortest path

To solve the problem, we have to find the shortest distance the salesman has to travel in order to visit each city exactly once and end up at the starting city. The diagram above shows an example with four cities – a, b, c, and d – with the shortest path highlighted.

Despite many very intelligent people spending an awfully long time over the years looking at and analysing this problem, no one has come up with an algorithm to find the guaranteed shortest path. Or, to be more precise, the only algorithm people have found is the equivalent of PermSort: generate all the permutations of the cities and, for each, calculate the distance travelled.

Select the permutation that has the shortest distance. The table on the next page shows the exhaustive list of 24 possible paths, together with their distances. Notice that there are eight with the shortest distance of 43: you can start at any of the four cities to do the round trip, and proceed either clockwise or anticlockwise for a possible eight paths.

We've already seen that generating all permutations is exponential, so finding the solution to the TSP will take exponential time. (Of course, someone could find an exact algorithm for solving TSP tomorrow, in which case this argument is null and void, but I would say that that person would be fêted by every single mathematician and computer scientist alive so I wouldn't feel too bad, as we'll soon see.)

As an aside, with PermSort, at least we can cut the search short if we manage to find a permutation that has the items in sorted order. With the exhaustive search of the solution to the TSP, we have to evaluate all permutations. After all, the next permutation we generate could be the shortest.

Notice another subtle point: although it may take an awfully long time to generate all the permutations, finding out whether a given permutation fits the problem is very quick. For PermSort, all we need to do for a particular permutation is to walk through the items and check that the next item is greater than (or equal to) the current one.

This is a linear process whose execution time is proportional to the number of items. For TSP, finding out if a permutation is a

Simplex algorithms

A problem is only known as intractable if no one has devised a polynomial algorithm to solve it. Nevertheless, there are some NP problems that have been solved quickly in practice with exponential-time algorithms. An example is the simplex algorithm for solving linear programming problems. A linear programming problem is a model of some real-world process with some set of variable quantities. These quantities have some constraints imposed on them by the model, and the problem is to find the smallest cost, or the maximum output, or something similar.

In 1947, George Dantzig devised the simplex algorithm for solving such a problem (an algorithm that is polynomial in nature), but in 1972, Klee and Minty proved that solving the worst-case problem takes exponential time. In other words, although the simplex method works quickly for the majority of linear programming problems found in the real world – we've been lucky so far – it does have a bad side that is exponential. Linear programming problems are NP. To date, no one has devised a truly polynomial time algorithm that works for all linear programming problems. ■

Spotlight on… Approximation algorithms

The argument in the article is that there is no exact algorithm to an NP-complete problem, apart from the obvious one of generating all solutions and selecting the right one.

That doesn't mean we can't find a 'good enough' answer to a subset of these NP problems using some other polynomial algorithm. By good enough, I mean a solution that is obtained that is 'close enough' to the true solution that spending more time trying to refine it further is counter-productive.

In general, these approximation algorithms are only applicable to optimisation problems where you're trying to find the smallest, the biggest, the shortest or the longest solution to a problem. For pure problems, such as the satisfiability problem, such approximation algorithms are worthless.

Take the TSP as an example. Suppose that we find a solution to the problem that's within 10 per cent of the optimal value in a tiny fraction of the time taken by an exhaustive search. Our mythical travelling salesman would be able to complete his journey using the 'good enough' path well before the exhaustive search even finished.

There are many algorithms that attempt to find these 'good enough' solutions. For the TSP, an excellent approximation is the ant colony algorithm. Other problems are amenable to algorithms such as genetic algorithms, simulated annealing and so on. ■

Other NP-complete problems

There are many NP-complete problems that have all been shown to be reducible to the Boolean satisfiability problem.

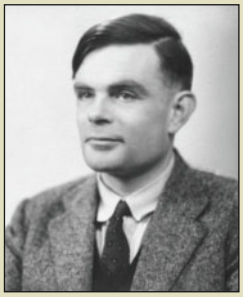
Graph and network theory is absolutely full of NP-complete problems, but they tend to be more mathematical than we have space for. For more 'real-world' examples, there's the bin-packing problem (packing a set of items of different sizes into a set of bins), the knapsack problem (packing part of a set of items of different sizes and values into a knapsack, such that the total value of the knapsack is maximised), and the timetable problem (given a set of courses, teachers and pupils, draw up a valid timetable).

Many games are also NP-complete. Some examples include Kakuro, FreeCell, Sudoku, and the solutions to various Tetris problems when you already know the sequence of pieces that you will be given. ■

Spotlight on… Undecidability

Although the NP problems I've described so far in this feature are intractable, I've also shown that they are 'decidable'. Given a solution to an NP problem, we can show in polynomial time that it satisfies the original problem. For example, verifying that a solution to PermSort is valid takes linear time.

Some NP problems are actually worse than that. In the 1930s, Alan Turing proved that there exist problems that are not only intractable but also undecidable. The classic example is devising an algorithm that, given a computer program and a set of inputs to that program, will decide whether the program will halt or not. Turing proved that it was impossible to specify such an algorithm – you could invent one, but you could never prove that it worked. ■



▲ **Alan Turing proved that undecidable, intractable problems also exist.**

possible solution is equally as quick: since nonexistent city links have some very large distances associated with them, if the total distance is greater than this, the permutation is not valid. Of course, testing to see if a particular path is the shortest is exactly equivalent to the original problem that we faced: we have to generate all of the possible valid paths and see that it is the shortest.

From this brief argument, you can see that there are algorithms whose execution time is proportional to some polynomial in the number of items – we've seen O($n2$), for example, and O(n.log n) can also be counted as polynomial, since it's always less than O($n2$). And we've seen others where the whole algorithm is defined as essentially working out all possible solutions in exponential time, with the check for a valid solution being much quicker and, in fact, polynomial.

> **"Much work was done trying to convert one given NP problem into another"**

We say that 'polynomial time' algorithms belong to the class P (for polynomial), and all the others belong to the class NP (for non-polynomial). Although this definition wouldn't throw out a O($n1000$) algorithm from the class P – an algorithm we would be somewhat hesitant to implement – in reality, the maximum power of n tends to be two or three.

Although implementing the P algorithms may be non-trivial or hard, they don't hold any terrors for us with regard to execution – we understand how fast they work with regard to the number of items. In fact, the earliest researchers into this discussion of algorithm and computational complexity would name polynomial time algorithms as 'good' and the NP algorithms as, well, 'not so good'.

The general term that has come into use is 'intractable'. NP problems are known as intractable problems, and the algorithms to solve them exactly tend to be some variation of 'search through all the permutations for the right answer'.

Finding patterns

Given that there are intractable problems, are they related or are they all different? Much work during the 1950s and 1960s was spent in trying to convert one given NP problem into another. These conversions were known as reductions: people tried to reduce one problem to another using a polynomial algorithm in the hopes that if they could devise a polynomial algorithm to solve the second, they had, in effect, devised a slightly more complicated polynomial algorithm to solve the first.

Finally, in 1971, Stephen Cook presented a paper called *The Complexity of Theorem Proving Procedures* that laid down the theory of NP-completeness. In this paper, he first emphasised the importance of polynomial reducibility. Second, he concentrated on decision problems (that is, problems solvable with yes/no or true/false) that belonged to the class NP. Third, he showed that there was one problem in this set, known as the satisfiability problem, that was the 'ancestor' of all NP decision problems – all other NP problems could be reduced to it. This is, if you like, the archetypal NP problem. In other words, the NP class is complete and we talk about NP-complete problems.

If you can find a polynomial solution to the satisfiability problem, you will have automatically found a polynomial algorithm to all other NP problems. In mathematical terms, you will have shown that P = NP.

Richard Karp, a year later, proved that the TSP (at least the decision problem version of it) was reducible to the satisfiability problem and so, again, if you solve that, you've solved TSP.

So what is this archetypal problem? Imagine a Boolean formula – an expression containing several Boolean variables combined with logical operators such as AND, OR, XOR, NOT and the like. This expression will either have the value 'false' or 'true' (yes or no, if you like) given some values for its Boolean variables. Is there a set of values for these variables that will produce true? This is the satisfiability problem (known as SAT), and it is NP hard. As an example, suppose we use one of the simplest Boolean expressions: x AND y. We can quickly determine that if both x and y are true, the result of the expression is true. This is pretty much equivalent to finding the shortest path for TSP given two cities in the map – from A go to B, and then return to A.

However, once we have a Boolean expression with many variables, there is no known algorithm shorter than enumerating all possible permutations of true/false for those variables to find a combination that evaluates to true for the whole expression. Despite this, over the years, various methods have been devised to quickly solve SAT problems, even with thousands of variables. Nevertheless, SAT is NP-complete and a polynomial time algorithm has not been found.

Find one, and the world of mathematics will beat a path to your door. PCP

The exhaustive solution for our TSP problem

Path taken	Distance travelled	Path taken	Distance travelled
abcd	50	cdab	50
bacd	45	dcab	45
cabd	45	acdb	45
acbd	43	cadb	43
bcad	43	dacb	43
cbad	50	adcb	50
dabc	50	bcda	50
adbc	43	cbda	43
bdac	43	dbca	43
dbac	45	bdca	45
abdc	45	cdba	45
badc	50	dcba	50

Julian M Bucknall has worked for companies ranging from TurboPower to Microsoft and is now CTO for Developer Express.
feedback@pcplus.co.uk