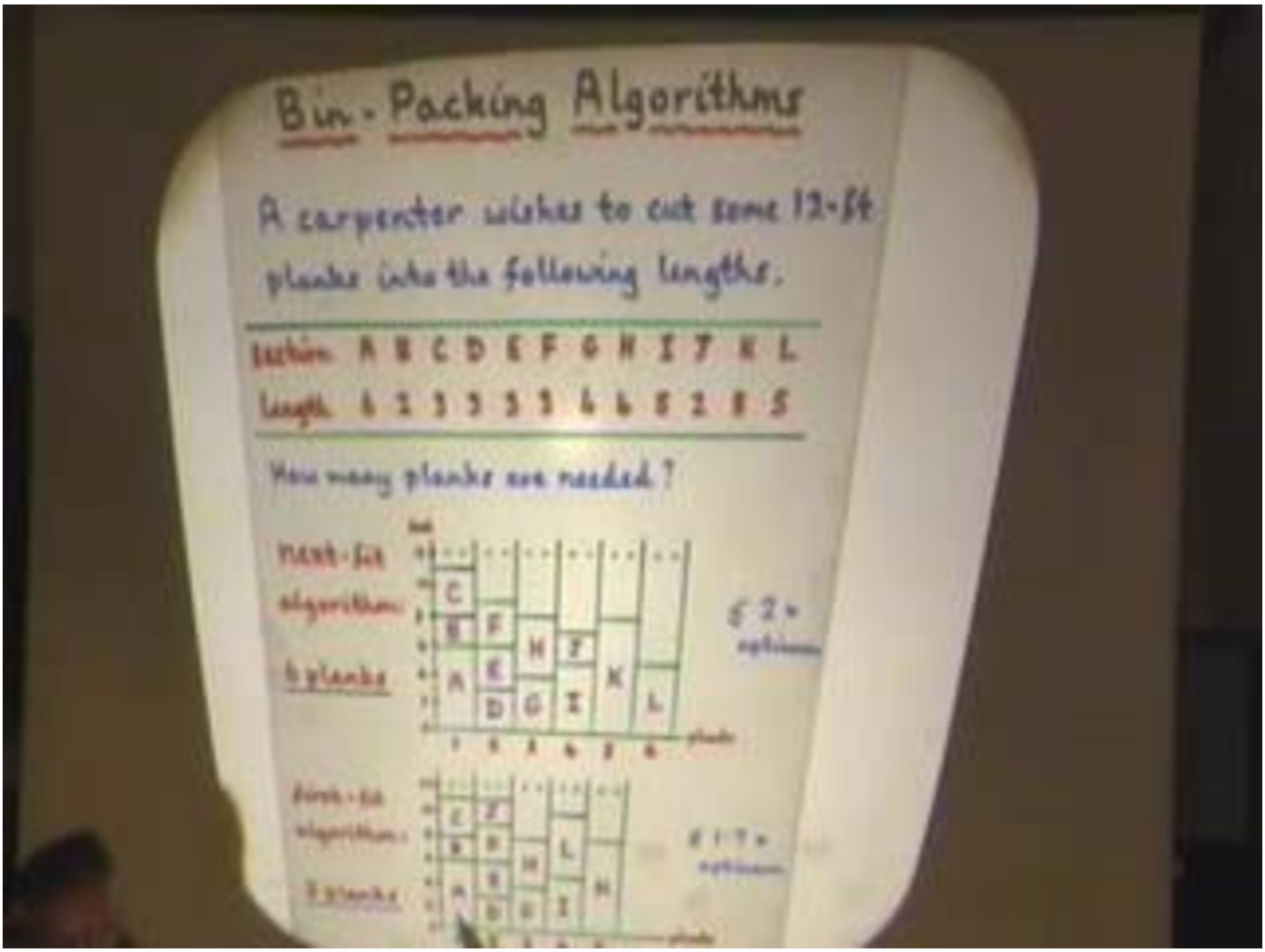
GRESHAM COLLEGE
Founded 1597

How hard is a hard problem? Transcript

Date: Wednesday, 9 March 2005 - 12:00AM

Location: Barnard's Inn Hall



HOW HARD IS A HARD PROBLEM?

Professor Robin Wilson ***Introduction*** As in my last lecture, I'll be telling you today how you can earn a million dollars. Last time I offered you the chance to earn this amount by solving the *Riemann hypothesis*, a problem relating to prime numbers – sadly, none of you has yet come forward with a solution. This time, my problem is concerned with *algorithms*, which are procedures for solving problems. It's known as the '*P* versus *NP* problem', and is concerned with how hard it is to solve problems. Its solution would be of great theoretical interest to both mathematicians and computer scientists, and also of enormous practical importance in operations research and business economics. Let me remind you of where the million dollars come in. In May 2000 in Paris, the Clay Mathematics Institute, set up with the mission 'to increase and disseminate mathematical knowledge', hosted a meeting called *A celebration of the universality of mathematical thought*. The central theme of this meeting was the announcement of seven so-called 'millennium problems', with a prize of one million dollars for the solution to each, in order to celebrate mathematics in the new millennium. The timing of their announcement was deliberate: it celebrated the centenary of the most famous lecture ever given in mathematics – David Hilbert's lecture at the International Congress in 1900, also in Paris, when he listed the 23 problems that he most wanted to see solved in the twentieth century. Some of these were indeed solved, while others have remained open to this day. These seven millennium problems are the problems considered by the mathematical community to be the most difficult and important in the subject – the ' Himalayas of mathematics'. They're all highly technical and some are extremely difficult to describe, but very roughly they're as follows:

- the *Riemann hypothesis* (relating to prime numbers);
- the *P = NP? problem* (on the efficiency of algorithms for solving problems);
- the *Poincaré conjecture* (on surfaces in four-dimensional space) – this is the only one of the seven that may be nearing a solution;
- the *Navier-Stokes equation* (solving a partial differential equation arising from the motion of fluids);
- *Yang-Mills theory* (relating to the mathematical treatment of some quantum physics);
- the *Birch-Swinnerton-Dyer conjecture* (another problem from number theory);
- and finally the *Hodge conjecture* (a highly technical problem from geometry and topology).

As I mentioned last time, a general description of them appears in Keith Devlin 's popular book *The Millennium Problems: the Seven Greatest Unsolved Mathematical Puzzles of our Time*. So what is an algorithm? It's basically *a finite step-by-step procedure for solving a problem*. You can think of it as a sort of recipe – like cake-making, where you input sugar, flour and eggs, carry out each step of the recipe, and the output is a perfect cake. For a general algorithm you input some data, carry out each step in turn, and produce an output that's the solution to your problem. The word *algorithm* derives from the ninth-century Persian mathematician al-Khwarizmi, about whom I'll be telling you on 5 October in my lecture on Islamic mathematics. He wrote an influential book on arithmetic, on what we now call the Hindu-Arabic numerals, and also a book which included the word *al-jabr* in its title, giving us our word *algebra*. When this book was later translated into Latin, its title became *Ludus algebrae et almucgrabalaeque*, so perhaps we got off lightly. In its Latin form, the opening words of the arithmetic book became *Dixit algorismus* and the word *algorism* came to be used in the Middle Ages for 'arithmetic'. So the word *algorithm* is named after al-Khwarizmi – even though my American friends like to claim that it really celebrates Al Gore. ***Examples of algorithms*** In this talk I'll look at many different algorithms – chosen from several different areas – algebra, number theory, arithmetic, graph theory and computer science. The first one you met in my earlier talk on Mesopotamian mathematics. It comes from a clay tablet, with several other problems of a similar nature. I have subtracted the side of my square from the area: 14, 30. You write down 1, the coefficient. You break off half of 1. 0; 30 and 0; 30 you multiply. You add 0; 15 to 14, 30. Result 14, 30; 15. This is the square of 29; 30. You add 0; 30, which you multiplied, to 29; 30. Result: 30, the side of the square. This tablet tells us how to solve what we now call a quadratic equation, and although not explicitly given in general form, the description is clearly a step-by-step recipe for finding a solution. If we work through it, we can easily see that it corresponds to completing the square, or to using the quadratic equation formula. Another algorithm concerns writing a given decimal number in binary form. Recall that in our decimal notation, a number such as 675 represents (6 × 10 2) + (7 × 10 1) + (5 × 10 0), where 10 0 is 1. In the same way, a binary number can be written in powers of 2: the decimal number 13 can be written as (1 × 2 3) + (1 × 2 2) + (0 × 2 1) + (1 × 2 0), or 1101 in base 2. It's said that there are three kinds of mathematician – those that can count and those that can't. It's also said that there are 10 kinds of mathematician – those that can count in binary and those that can't. Let's look at a couple of algorithms that we can use to turn decimal form to binary form. In the first, we find the largest power of 2 less than the number, subtract it, and repeat the process. From example, from 13 we first subtract 2 3 (= 8), putting 1 in the 2 3 place, and then repeat the process with 13 – 8 = 5. We next subtract 2 2 (= 4), putting 1 in the 2 2 place, and then repeat the process with 5 – 4 = 1. But this is just 2 0, so we have 1 in

the 2 0 place. This gives us 1101, the required binary form. Another algorithm is to keep on dividing by 2, writing 1 whenever the remainder is 1. So 13 > 6 (rem. 1), > 3 > 1 (rem. 1) > 0 (rem. 1). The remainders, when read backwards, give us 1101, the required binary form. Binary numbers also underlie a form of calculation which is sometimes called 'Russian multiplication', although it has appeared in various places. To multiply two numbers such as 23 and 89, we write them in two columns and successively halve the numbers in the first column (ignoring any remainders) while doubling the numbers in the second column. We then delete any rows where the left-hand number is even, and add all the other numbers in the right-hand column. This gives the required answer. For example, if we write 23 on the left, we successively get 23, 11, 5, 2 (which we cross out) and 1. The corresponding numbers on the right are then 8, 178, 356, 712 (crossed out) and 1424. Adding these numbers gives 2047, the answer. Alternatively, if we write 89 on the left, we successively get 89, 44 (which we cross out), 22 (which we cross out), 11, 5, 2 (which we cross out) and 1. The corresponding numbers on the right are then 23, 46 (crossed out), 92 (crossed out), 184, 368, 736 (crossed out) and 1472. Adding these numbers gives 2047, as before. One of the oldest algorithms is the *Euclidean algorithm*, which appears in Euclid 's *Elements*, Book VII. It is used to find the greatest common divisor, or highest common factor, of two given numbers. Here are two methods: To find the greatest common divisor of 1155 and 1575, we factorise each number as a product of prime number – so 1155 = 3 × 5 × 7 × 11 and 1575 = 3 2 × 5 2 × 7. Taking the lower power of each prime gives the greatest common divisor 3 × 5 × 7, which is 105. Unfortunately, this is very inefficient for large numbers, as we'll see. The algorithm that Euclid proposed, around 300 BC, is successively to divide the larger number by the smaller one and record the remainder. We obtain successively: 1575 = (1 × 1155) + 420, 1155 = (2 × 420) = 315, 420 = (1 × 315) + 105, 315 = 3 × 105. This tells us that the greatest common divisor is 105. This appears in Euclid, Book VII, Propositions 1 and 2. The first talks about two unequal numbers being set out and the less being continually subtracted from the greater, as we did here. If a unit (that is, 1) is left, the original numbers are prime to one another – that is, the greatest common divisor is 1. The second uses this method to find the greatest common divisor of two numbers not prime to one another. How quick is this method? The worst case happens for the so-called *Fibonacci numbers* 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, …. These numbers arose in Fibonacci's *Liber Abaci* of 1202 in connection with a problem involving pairs of rabbits: the rabbits produce a new pair of rabbits each month, with each new pair starting to reproduce after just one month and no rabbits ever dying. The numbers of pairs of rabbits after each month are given by the numbers in the sequence (a hare-raising problem?). If we carry out the Euclidean algorithm on two terms of the Fibonacci sequence: 34 = (1 × 21) + 13; 21 = (1 × 13) + 8; 13 = (1 × 8) + 5; and so on, we see that each successive remainder is a term of the Fibonacci sequence. This is a worst possible case, and it can be shown in general that whenever we apply the Euclidean algorithm the number of steps is about 5 × (the number of digits in the larger number), so for a 100-digit number there are at most 500 steps. Compare this with the earlier prime factorisation method which grows exponentially: for a 100-digit number, the number of steps can be up to 10 20, a number larger than the number of atoms in the universe. ***Computer science*** A problem that arises in computing science, and in many practical situations (such as when you're trying to pack the boot of your car) is the *bin-packing problem*. Here's an example of it: *A carpenter wishes to cut some 12-foot planks into the following lengths: section A B C D E F G H I J K L length 6 2 3 3 3 3 4 4 5 2 8 5 How many planks are needed?* Alternatively, if you want to place objects of these sizes into bins of capacity 12, how many bins do you need? There are three algorithms you might use to solve this problem. The simplest is the *next-fit algorithm*, where you go along the list cutting sections from the planks in turn. Thus, from plank 1, we cut sections *A*, *B*, *C* (with length 11) from plank 2, we cut sections *D, E, F* from plank 3, we cut sections *G, H* from plank 4, we cut sections *I, J* from plank 5, we cut section *K* from plank 6, we cut section *L*, using *six* planks in total. It can be proved that this method always gives an answer that is at most twice the optimum solution. A slightly better method is the *first-fit algorithm*, where we go along the list as before, but cut each section from the first plank available. Thus, the first three planks are as before, but when we come to section *J*, we cut it from plank 2, and we cut section *L* from plank 4. This gives the following solution: from plank 1, we cut sections *A*, *B*, *C* from plank 2, we cut sections *D, E, F, J* from plank 3, we cut sections *G, H* from plank 4, we cut sections *I, L* from plank 5, we cut section *K* using up *five* planks in total. This method gives an answer that is at most 1.7 times the optimum solution. These methods are both rather simple-minded. After all, if you're trying to pack cases into your car, you'd probably try to fit in the largest ones first and then pack the others around them. This leads to the *first-fit decreasing algorithm*, where we first list the items in decreasing order of size before trying to pack them. So here we first cut off section *K* (length 8). We then cut the next largest one (*A*) from plank 2, and also the following one (*I*). We obtain the following solution: from plank 1, we cut sections *K, G* from plank 2, we cut sections *A, I* from plank 3, we cut sections *L, H, C* from plank 4, we cut sections *D, E, F, B* from plank 5, we cut section *J.* So again we use up *five* planks in total. This method always gives an answer that is at most 1.22 times the optimum solution. In this example, none of the above methods gives the optimum solution, which uses just four planks: from plank 1, we cut sections *A, C, D* from plank 2, we cut sections *B, E, F, G* from plank 3, we cut sections *H, K* from plank 4, we cut sections *I, J, L.* Another type of algorithm involves the sorting of items in a list. Suppose that we want to sort the following seven animals into alphabetical order: *lion – tiger – aardvark – zebra – camel – dingo – gorilla.* There are several algorithms that do this – we'll use one called the *bubble-sort algorithm*, where we let the earlier words rise like bubbles to higher in the list. At the first stage *aardvark* bubbles up above *tiger*, and *camel*, *dingo* and *gorilla* all bubble up above *zebra*, giving *lion – aardvark* – tiger – *camel – dingo – gorilla* – zebra. At the next stage *aardvark* bubbles up above *lion*, and *camel*, *dingo* and *gorilla* all bubble up above *tiger*, giving *aardvark* – lion – *camel – dingo – gorilla* – tiger– *zebra.* At the next stage *camel*, *dingo* and *gorilla* all bubble up above *lion*, giving *aardvark – camel – dingo – gorilla* – lion – *tiger – zebra.* Any further bubbling makes no difference – the items are now in alphabetical order. ***Network algorithms*** Given an algorithm for solving a particular problem, we need to know how efficient it is. To illustrate the ideas involved, let's look at two

problems from operations research – the *minimum connector problem* and the *travellingsalesman problem*. In the *minimum connector problem*, we wish to connect a number of cities by links, such as canals, railway lines, or air routes. There must be enough links so that we can get from any city to any other (not necessarily in one step), but building canals is costly, so we need to keep the total cost to a minimum. How can we do this? Here's an example with five cities, *A*, *B*, *C*, *D*, *E*, where the links have the following lengths, or costs: *AB* : 6; *AC*: 4; *AD*: 8; *AE*: 2; *BC*: 5; *BD* : 8; *BE*: 6; *CD*: 9; *CE*: 3 and *DE*: 7. How do we find our minimum connector? We can experiment: taking *BC*, *CD*, *DE* and *EA* we link all the cities, with total length 2 + 7 + 9 + 5, which is 23; taking *AB*, *AC*, *AD*, and *AE* we again link all the cities, but with total length 6 + 4 + 8 + 2, which is 20 – quite an improvement. But is this the best we can do? Notice that each of these solutions is a *tree structure* – it contains no cycles: for if there were a cycle, we could decrease the total length by removing any one link from the cycle. So how do we find the right tree structure? There are 125 possible trees joining these cities, so we can't use trial and error. Worse still, the number of trees increases dramatically with extra cities: over 1000 trees for six cities, and 100 million for ten cities. Since most practical examples have hundreds of cities, we need an algorithm to solve this problem. The algorithm we use is called the *greedy algorithm*, since at each stage we're 'greedy' and choose the shortest possible link available. So here we start with link *AE* (length 2). Then *CE* (length 3). We can't now choose *AC* (length 4), since that gives a cycle, so we use *BC* (length 5). What next? Each of the links of length 6, *AB* and *BE*, give a cycle, so we go to DE (length 7). This completes the tree, so we stop – the cities are now all linked by a tree of total length 2 + 3 + 5 + 7 = 17. This is a great improvement on what we had before, and is the best possible solution. In fact, the greedy algorithm *always* gives us a minimum connector – however many cities there are. We just model the situation by a network, and program a computer to make the choices. Moreover, it's an *efficientalgorithm*, taking very little computer running time – it's at worst a multiple of the *square* of the number of cities, so if the number of cities increases ten-fold, the computer time goes up by a factor of only 100, which is very efficient. The other problem I mentioned is the *travelling salesman problem*. Here, a salesman wishes to sell his wares in a number of cities, so he visits them all and then returns to his starting point. How can he plan his route so as to minimise the total distance? One route is to go round the outside: $A \to B \to C \to D \to E$ and back to *A*, with total distance 29. Or he could follow the star-shaped route $A \to C \to E \to B \to D \to A$ – still distance 29. The route $A \to C \to E \to D \to B \to A$ is slightly better, with length 28. The best possible route is $A \to D \to B \to C \to E \to A$, with length 26, but to find it I had to use trial and error: no known efficient algorithm is guaranteed to produce it – but neither has anyone ever proved that no such efficient algorithm can exist. There are many other problems involving such network-type diagrams, which we now call *graphs*: this is nothing to do with the usual meaning of that word. The travelling salesman problem involved visiting all the cities and returning to the starting point – and we can ask a similar problem for other graphs – for example, *is there a cycle that visits all the corners of a cube and returns to the starting point?* (Such a cycle is called a *Hamiltonian cycle*, after the Irish mathematician and physicist Sir William Rowan Hamilton.) A little experimentation shows that indeed there is. More generally, given a graph, is there an algorithm for finding whether it has such a Hamiltonian cycle exists? – no such algorithm is known. Another problem, known as the *graph isomorphism problem*, involves trying to decide whether two graphs are the same. Given two different network drawings, how can we decide whether they are different drawings of the same graph, or whether they arise from different graphs? *[Several examples of this problem were illustrated in the lecture.]* No efficient algorithm is known – in general, we may need to check *all possible* ways of re-labelling the points of the graph, and that's extremely inefficient. Even for graphs with just ten points, there are over three million re-labellings that we'd need to check. Yet another type of graph problem involves trying to draw a given graph diagram in the plane in such a way that none of the linking lines (edges) cross each other. For some graphs this can be done *[this was illustrated in the lecture]*, whereas for others (such as the one in the minimum connector and travelling salesman problems) no such drawing exists. Is there an efficient algorithm for deciding whether a given graph or network can be drawn in the plane with no crossing edges? ***P = NP?*** It's time that we stepped back and looked at such problems from a more general point of view, but first let's look at a short video sequence in which Keith Devlin explains the basic problem we're dealing with. In his famous *Essay on population* in 1798, Thomas Malthus contrasted the linear growth of food supplies with the exponential growth in population. He concluded that however well one can cope in the short term, the exponential growth would win in the long term, and there would be severe food shortages – a conclusion that has been borne out in practice. A similar distinction can be made for algorithms used to solve problems. Each problem has an *input size*, which we'll call *n* – it may be a number you want to factorise, or the number of cities in a network. The algorithm also has a *running time*, which we'll call *T*, which may be the time that a computer needs to carry out all the necessary calculations, or the actual number of such calculations involved. It any case, the running time *T* depends on the size *n* of the input. Particularly important, because they are the most efficient, are the *polynomial-time algorithms*, in which the maximum running time is proportional to a power of the input size – say, to *n* 2 or *n* 7. The collection of all polynomial-time algorithms is called *P*. In contrast, there are algorithms that are known not to take polynomial time – such as the *exponential-time algorithms* in which the running time may grow like 2 *n* or faster. The collection of all algorithms that are known to have no polynomial algorithm is called *not-P*. (This is not the same as *NP*, which we explain below.) In general, algorithms in *P* are efficient, and problems that can be solved using them are called *tractableproblems*. Algorithms in *not-P* are not efficient, and the corresponding problems are called *intractable problems*. To see the difference, let's draw up a table comparing the running times for input sizes *n* = 10 and *n* = 50, for various types of polynomial and exponential algorithm and for a computer performing one million operations per second: *n = 10 n = 50 n* 0.00001 seconds 0.00005 seconds *n* 2 0.0001 seconds 0.0025 seconds *n* 3 0.001 seconds 0.125 seconds *n* 5 0.1 seconds 5.2 minutes 2 *n* 0.001 seconds 35.7 years 3 *n* 0.059 seconds 2.3 × 10 10 years There are many polynomial algorithms around: their running time is at worst some power of the input – usually the square, or the cube. Examples include:

- the greedy algorithm for the *minimal connector problem*, whose running time is proportional to the square of the number of cities;

- the algorithms for determining whether a graph with n points is *planar* – the one I described has running time proportional to n 3, but there are improvements that are proportional to n;

- the *bubble-sort algorithm* – for a list of n words, the running time is proportional to n 2.

On the other hand, there are several problems for which no polynomial algorithm has ever been found. These include the *travelling salesman problem*, the *bin-packing problem*, the problem of deciding whether a given graph has a *Hamiltonian cycle*, and the *graph isomorphism problem*. At this point we at last introduce *NP*. You will recall that *P* is the set of 'easy' problems, that can be solved with a polynomial-time algorithm. As far as we know, the Hamiltonian cycle problem is not one of these – but if you give me an attempted solution, then I can *check* in polynomial time that it *is* a solution. In general, *NP* is the set of 'non-deterministic polynomial-time problems' – those for which a solution, when given, can be checked in polynomial time. Clearly *P* is contained in *NP*, since if a problem can be solved in polynomial time, the solution can certainly be checked in polynomial time – checking a solution is far easier than finding it in the first place – or are they actually the same? That's the big question – is $P = NP$? It seems very unlikely – indeed, few people believe that they are the same – but it's never been proved. For example, to solve the travelling salesman problem, I need to find a cyclical route of minimum length, and no polynomial algorithm is known. But if you give me a possible route for the salesman, and a fixed number k, then I can check in polynomial time whether the salesman it's a cyclical route, and whether it has total length less than k. So it's in *NP*. ***NP-complete problems*** In 1971 Stephen Cook, of the University of Toronto, wrote a short but fundamental paper, *The complexity of theorem-proving procedures*, in which he considered a particular problem in *NP* called the *satisfiability problem*. He proved the amazing result that every other problem in *NP* can be transformed into it in polynomial time. That means that, going via the satisfiability problem, *any problem in NP can be transformed to any other problem in NP in polynomial time*. So if the satisfiability problem is in *P*, then so is everything in *NP* – so $P = NP$. But if the satisfiability problem is not in *P*, then $P \neq NP$. Thus the whole $P = NP$? question relies on whether we can find a polynomial algorithm for just one problem. But not just one – there are thousands of such problems! We say that a problem is *NP-complete* if its solution in polynomial time means that *every* problem in *NP* can be solved in polynomial time. These include the satisfiability problem, and all your favourite problems. The travelling salesman problem is NP-complete – and so are the graph-isomorphism problem, the bin-packing problem, the Hamiltonian cycle problem, and thousands of others. If you could find a polynomial algorithm for just one of them, then polynomial algorithms would exist for the whole lot, and *P* would be the same as *NP*. On the other hand, if you could prove that just one of them has no polynomial algorithm, then none of the others could have a polynomial algorithm either, and *P* would then be different from *NP*. To conclude, let me just remark that many of the problems we're dealing with here are of highly practical importance. There's an enormous amount of money at stake in determining whether $P = NP$ – and it's not just the million-dollar prize that we started with!