# How hard is too hard? An introduction to complexity

**Professor Colva Roney-Dougal OBE FRSE**

**11th May 2026**

## 1. A difficult problem

Imagine that you are hosting an enormous family wedding. The plan is to seat all 100 of your closest relatives around a giant circular table. Unfortunately, your family is rather dispute-prone, and if certain people are sat next to one another then fights will surely break out.

The **party-planning problem** asks you to find a seating arrangement

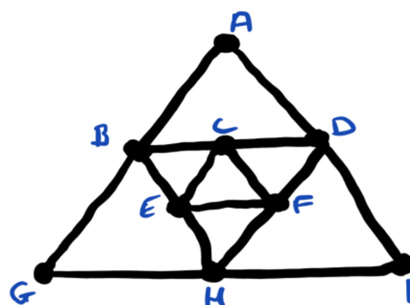
such that dinner remains fight-free.

For example, here's a smaller but still fractious family: the table below shows who is willing to sit next to whom.

| Person | Will sit next to | Person | Will sit next to | Person | Will sit next to |
|---|---|---|---|---|---|
| A | B, D | B | A, C, E, G | C | B, D, E, F |
| D | A, C, F, I | E | B, C, F, H | F | C, D, E, H |
| G | B, H | H | E, F, G, I | I | D, H |

Looking at this table can feel a little overwhelming, so what mathematicians do with this data to make it easier to understand is to represent it pictorially, with a drawing called a *graph*. We put one dot (called a *vertex*) for each person, and draw a line between two people (called an *edge*) if they are willing to sit next to each other.

Here's a graph representing the table above:




Managing to seat this family at dinner means finding a route around the graph which visits each vertex exactly once before returning to the start. Staring at the graph we see that person A *must* sit between B and C, so part of our table must look like D, A, B (either clockwise or anticlockwise). Similarly, person G *must* sit between B and H, so part of our table has to look like D, A, B, G, H. But now person I will only sit between H and D, and we have a problem! Person D can only sit in one place, and now there's no way sit C, E and F. We either need to call the wedding off, or persuade some family members to reconcile.

## 2. The birth of computing

To solve these types of problems, we use computers. In 1937, the genius Alan Turing came up with the idea of a programmable computer, long before any such machine was a reality. In his honour we call these imaginary machines *Turing machines*: they can solve any problem that a real computer can solve, even a quantum computer.

Alan Turing proved the remarkable result that there are problems, where the answer is just yes or no, which *cannot* be solved by any computer. There are lots of these problems, and some of them have great real-world importance.

- We are all used to seeing a timer appearing on our screen, or a little spinning wheel, and not knowing how long we might need to wait for the computer to stop doing whatever we just asked it to do, or indeed whether it will work at all. Turing proved that it is impossible to design a computer to test whether all other computer programmes will stop: any way we try to do it, there will be some programmes where it just can't tell.

Even if a problem *can* be solved on a computer, scientists quickly realised that we might care how *long* it will take to do so.

- For example, if we wish to **add** two numbers, like 123 + 456, this takes just three single digit additions (it might take more if there were carries). In general, to add two numbers we need to do roughly as many single digit additions as there are digits in the longest number, and then maybe take care of any carries. So the total work is proportional to the number of digits in the longest number.
- To **multiply** two numbers, like 123 x 456 is more work: we need to multiply each of the digits in one of the numbers by each of the digits of the other, and then do some additions. So in this case, the amount of work we need to do is proportional to the product of the number of digits of the two numbers.

In practice, both of these are *easy* problems to solve: if you tell me the number of digits of the biggest number, at worst I multiply it by itself to get an idea of how long it will take to find the answer. In general, we deem a problem to be *quick* to solve if the amount of time it will take is proportional to any given fixed power of the number: addition is to the power 1, multiplication is to the power 2, and so on.

**Class P** consists of all problems that can be solved in time bounded by some polynomial of the input size. **P** stands for *polynomial time*.

Let's compare a pretty big polynomial with one of the smallest exponentials, imagining my computer can do one operation per millisecond (it's actually much faster!)

| N | 2 | 10 | 30 | 100 |
|---|---|---|---|---|
| $N^5$ | 32 ms | 100 secs | 6.75 hours | 115 days |
| $2^N$ | 4 ms | 1 sec | 12.42 days | 17 centuries |

So although raising numbers to powers can get big, they do so much more slowly than taking exponentials.

## 3. Two more hard problems

### Problem 1: cryptography

Outside mathematical circles, Alan Turing is most famous as the man who broke the enigma code during World War II. The connection between these amazing achievements is that almost all ways of sending secret messages rely, in some way, on big numbers.

Most modern forms of cryptography use a version of a problem about numbers. To describe it, we first need a definition. A *prime* number is a whole number, bigger than 1, that is only divisible by itself and 1. For example the number 2 is prime, as it is only divisible by 1 and 2. So are 3, 5, and 10000000019 – although you might need a computer to check that last one! The number 4 is not prime, as it's divisible by 2. Similarly, 15 is not prime, as it's divisible by 5, and every though it's big we can see that 10000000000 is divisible by 10, so is not prime.

By dividing and dividing again, we can break every whole number down as a product of primes, for example 12 = 2 x 2 x 3.

Can you break 10101203027 into primes?[1]

Most modern encryption relies on the fact that multiplication is easy (lies in **P**), whilst factorisation seems to be difficult.

## Problem 2: Job scheduling

Suppose you run a factory with three identical machines. One morning at 9am, you get to work to see that overnight you have received jobs that will take this many minutes each:

| Job | A | B | C | D | E | F | G | H |
|---|---|---|---|---|---|---|---|---|
| Minutes | 200 | 250 | 220 | 90 | 190 | 50 | 130 | 200 |

You want to know whether it will be possible to run all of these jobs before you finish work that day at 5pm. That's a total of 480 minutes, so you need to get cracking!

The way that we solve problems like this on a computer is by an approach called backtrack search.

- Initially, assign the jobs in order to machine 1 until you can't give it more without staying late at work. That's jobs A and B, which take 450 minutes, as nothing takes only 30 minutes.
    - Now assign the remaining jobs to machine 2, until it's full. We try jobs C, D, and G.
        - Then we check whether machine 3 can do the rest. That will take too long. We **backtrack.**
    - We try the next possible assignment for machine 2. If we don't give it G then we can't give it anything else, so we try replacing job D, and instead assigning machine 2 jobs C and E.
        - Again we check whether machine 3 can do the rest. We fail and backtrack.
    - Next we try machine 2 with C and F.
    - Then we try machine 2 with C and G.
    - Then we try machine 2 with C and H. Again this fails. So the issue was assigning job C to machine 2. But job C must go somewhere! We need to **backtrack** all the way to the start.
- We try instead assigning machine 1 jobs A and C.
    - I'll leave you to check the details, but every possible assignment to machine 2 will fail.
- We try the next assignment to machine 1: jobs A, D, and E.
    - Now machine 2 can get jobs B and C.
        - And now machine 3 can do jobs F, G and H!

So the answer is yes, as long as we get started right away!

This method may seem cumbersome, and there are lots of little ways to speed it up, but it is ultimately the best approach we have for solving lots of hard problems.

# 4. P versus NP

[1] 10101203027 = 100003 x 101009, and both of these are prime.

A problem with a yes/no answer is in **class NP** if whenever the answer is ‘yes’ there is some kind of proof of that claim, that we can check in polynomial time.

- For example, any problem in **P** with a yes/no answer is automatically in **NP**: we can do the whole calculation in polynomial time, so that’s our ‘proof’.
- The party planning problem is in **NP**: if I just ask whether people can be sat down at table, if the answer is “yes” then as a proof you can give me the table plan, which I can quickly check.
- The factorisation problem is in **NP**, provided I turn it into a yes/no problem, for example by saying “this number is a product of exactly two primes”. If you give me the two factors I can multiply them together again.
- The job scheduling problem is in **NP**: you can check that an assignment of jobs to machines covers all of the jobs in the given amount of time.

For many problems in **NP** we know of no fast way of finding the ‘proof’, only a way of checking it.

In the year 2000, the Clay mathematics institute chose seven of the most important unsolved problems in mathematics, and offered a prize of a million dollars for the solution to each of them. One of them is

Is **P** equal to **NP**?

That is, if the answer to a problem is easy to check, is the problem easy to solve?

So far, only one of the Clay Prize problems has been solved: in 2003, Grigory Perelman proved the Poincaré conjecture (but refused to claim the money).

## The hardest problems in NP

A problem is **NP-complete** if it is in **NP**, and if the existence of a polynomial-time solution to it would mean that *every* problem in **NP** can be solved in polynomial time.

This may sound like a frankly stupid definition, why would we think such a thing exists?

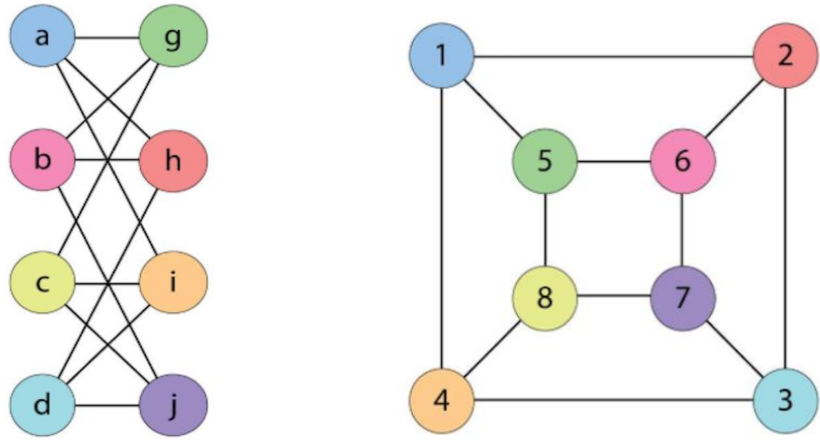
The reason is that in the 1970s, Stephen Cook and Leonard Levin both independently proved that such a problem *does* exist! The problem is a bit complicated to describe here, but it’s a problem in logic called 3-SAT.

Once we know that there exists a single **NP**-complete problem, it becomes surprisingly easy to find other ones: we now know dozens and dozens of them. Both the party planning problem and the job scheduling problem are **NP**-complete, for example.

We saw in the section on Turing’s marvellous machines that quantum computers cannot solve any problems that can’t be solved by classical computers. They can, however, solve some problems much more quickly than classical machines. At the moment, we don’t know of any fast quantum algorithms for any **NP**-complete problems, and there are good reasons to believe that maybe none exist. However, Peter Shor in 1994 cane up with an algorithm for quantum computers that can factorise numbers in polynomial time. This means that many current encryption techniques, on the internet especially, could be broken by quantum computers. Rapid progress is currently being made on building practical quantum computers, and Google have recently announced that they are making all of their systems change to new, hopefully quantum-secure, cryptography by 2029.

# 5. Symmetry and graphs

Recall the idea of a graph, from Section 1. The **graph isomorphism problem** asks whether two graphs are the same, just with different vertex names.

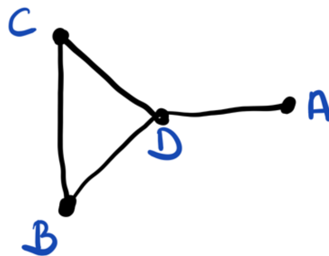



For example, the two graphs above are "the same": the colours show you how to do the re-naming.

The graph isomorphism problem is in **NP**: if the answer is "yes", then a proof of that is just a description of which letter corresponds to which number, as you can then check that the edges in the graph are correct.

Graph isomorphism is **not** believed to be **NP**-complete. So the question arises as to how hard it is?

One useful way to attach the problem is using *symmetries*: ways of moving the graph around so it looks like same. Consider the graph below:




This has just two symmetries: we can swap B and C, or we can leave everything in place.

Symmetries sometimes make it easier to solve problems in **NP**, but sometimes make it harder.

- Our graph representing the troublesome family for the party-planning problem in Section 1 has a lot of symmetries, in particular it has all of the symmetries of an equilateral triangle. That made it **easier** to see that there was no way to seat everyone around a table.
- The job scheduling problem becomes **easier** if some jobs take the same amount of time, and **easier** if (as in our example) some of the machines are identical.

In the 1980s, Luks showed that to solve the graph isomorphism problem, it's enough to design a fast algorithm to find all of the symmetries of a graph: we don't need to think about pairs of graphs. Finding the symmetries of a graph is much easier if some of the vertices have more edges than others. Luks showed that if we put an upper limit on the number of edges each vertex has, then the graph isomorphism problem is in **P**.

If **P** is not equal to **NP**, we should expect some problems that are in **NP** but are not **NP**-complete to be solvable in time that is *bigger* than a polynomial, but *smaller* than exponential. In a dramatic breakthrough in 2016, Babai showed that the graph isomorphism problem has exactly this property: he found a *quasipolynomial time* algorithm to solve it.

## 6. Conclusion: How to win a million dollars

First you should decide whether you're going to prove that **P** is equal to **NP**, or prove that it's not!

**To prove P = NP**: To do this, all you need to do is find a polynomial-time solution to your favourite **NP**-completely problem. You might for example find an algorithm for the party-planning problem that is guaranteed to run in polynomial time for all parties.

If you do this, take steps to protect yourself! You might have broken most internet cryptography as a side effect, and many national security agencies will be very interested in speaking with you.

**To prove P is not equal to NP**: this is the statement that most mathematicians think is probably true. To do this, you must show that some problem in **NP** has **no** fast solution. It doesn't need to be one of the **NP**-complete problems, graph isomorphism would do.

## Good luck!

© Professor C. M. Roney-Dougal 2026

## References and Further Reading

Scott Aaronson, *Quantum Computing Since Democritus* (Cambridge University Press, 2013).

William J. Cook, *In Pursuit of the Traveling Salesman: Mathematics at the Limits of Computation* (Princeton University Press, 2012).

Lance Fortnow, *The Golden Ticket: P, NP, and the Search for the Impossible*(Princeton University Press, 2013).

Dennis Shasha, *Out of their Minds: The Lives and Discoveries of 15 Great Computer Scientists* (first published 1995; Springer, 2008).