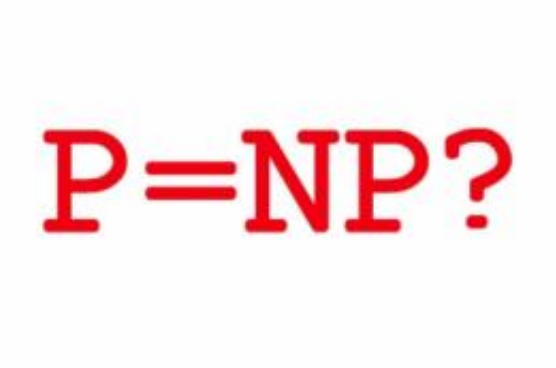
P vs. NP -- The most notorious problem in theoretical computer science remains open

October 29 2009, by Larry Hardesty



In the 1995 Halloween episode of *The Simpsons*, Homer Simpson finds a portal to the mysterious Third Dimension behind a bookcase, and desperate to escape his in-laws, he plunges through. He finds himself wandering across a dark surface etched with green gridlines and strewn with geometric shapes, above which hover strange equations. One of these is the deceptively simple assertion that P = NP.

In fact, in a 2002 poll, 61 mathematicians and computer scientists said that they thought P probably didn't equal NP, to only nine who thought it did — and of those nine, several told the pollster that they took the position just to be contrary. But so far, no one's been able to decisively answer the question one way or the other. Frequently called the most important outstanding question in theoretical computer science, the

equivalency of P and NP is one of the seven problems that the Clay Mathematics Institute will give you a million dollars for proving — or disproving. Roughly speaking, P is a set of relatively easy problems, and NP is a set of what seem to be very, very hard problems, so P = NP would imply that the apparently hard problems actually have relatively easy solutions. But the details are more complicated.

Computer science is largely concerned with a single question: How long does it take to execute a given algorithm? But computer scientists don't give the answer in minutes or milliseconds; they give it relative to the number of elements the algorithm has to manipulate.

Imagine, for instance, that you have an unsorted list of numbers, and you want to write an algorithm to find the largest one. The algorithm has to look at all the numbers in the list: there's no way around that. But if it simply keeps a record of the largest number it's seen so far, it has to look at each entry only once. The algorithm's execution time is thus directly proportional to the number of elements it's handling — which computer scientists designate N. Of course, most algorithms are more complicated, and thus less efficient, than the one for finding the largest number in a list; but many common algorithms have execution times proportional to N^2, or N times the logarithm of N, or the like.

A mathematical expression that involves N's and N^2s and N's raised to other powers is called a polynomial, and that's what the "P" in "P = NP" stands for. P is the set of problems whose solution times are proportional to polynomials involving N's.

Obviously, an algorithm whose execution time is proportional to N^3 is slower than one whose execution time is proportional to N. But such differences dwindle to insignificance compared to another distinction, between polynomial expressions — where N is the number being raised to a power — and expressions where a number is raised to the Nth

power, like, say, 2^N.

If an algorithm whose execution time is proportional to N takes a second to perform a computation involving 100 elements, an algorithm whose execution time is proportional to N3 takes almost three hours. But an algorithm whose execution time is proportional to 2^N takes 300 quintillion years. And that discrepancy gets much, much worse the larger N grows.

NP (which stands for nondeterministic polynomial time) is the set of problems whose solutions can be verified in polynomial time. But as far as anyone can tell, many of those problems take exponential time to solve. Perhaps the most famous problem in NP, for example, is finding prime factors of a large number. Verifying a solution just requires multiplication, but solving the problem seems to require systematically trying out lots of candidates.

So the question "Does P equal NP?" means "If the solution to a problem can be verified in polynomial time, can it be found in polynomial time?" Part of the question's allure is that the vast majority of NP problems whose solutions seem to require exponential time are what's called NP-complete, meaning that a polynomial-time solution to one can be adapted to solve all the others. And in real life, NP-complete problems are fairly common, especially in large scheduling tasks. The most famous NP-complete problem, for instance, is the so-called traveling-salesman problem: given N cities and the distances between them, can you find a route that hits all of them but is shorter than … whatever limit you choose to set?

Given that P probably doesn't equal NP, however — that efficient solutions to NP problems will probably never be found — what's all the fuss about? Michael Sipser, the head of the MIT Department of Mathematics and a member of the Computer Science and Artificial

Intelligence Lab's Theory of Computation Group (TOC), says that the P-versus-NP problem is important for deepening our understanding of computational complexity.

"A major application is in the cryptography area," Sipser says, where the security of cryptographic codes is often ensured by the complexity of a computational task. The RSA cryptographic scheme, which is commonly used for secure Internet transactions — and was invented at MIT — "is really an outgrowth of the study of the complexity of doing certain number-theoretic computations," Sipser says.

Similarly, Sipser says, "the excitement around quantum computation really boiled over when Peter Shor" — another TOC member — "discovered a method for factoring numbers on a quantum computer. Peter's breakthrough inspired an enormous amount of research both in the computer science community and in the physics community." Indeed, for a while, Shor's discovery sparked the hope that quantum computers, which exploit the counterintuitive properties of extremely small particles of matter, could solve NP-complete problems in polynomial time. But that now seems unlikely: the factoring problem is actually one of the few hard NP problems that is not known to be NP-complete.

Sipser also says that "the P-versus-NP problem has become broadly recognized in the mathematical community as a mathematical question that is fundamental and important and beautiful. I think it has helped bridge the mathematics and computer science communities."

But if, as Sipser says, "complexity adds a new wrinkle on old problems" in mathematics, it's changed the questions that computer science asks. "When you're faced with a new computational problem," Sipser says, "what the theory of NP-completeness offers you is, instead of spending all of your time looking for a fast algorithm, you can spend half your

time looking for a fast algorithm and the other half of your time looking for a proof of NP-completeness."

Sipser points out that some algorithms for NP-complete problems exhibit exponential complexity only in the worst-case scenario and that, in the average case, they can be more efficient than polynomial-time algorithms. But even there, NP-completeness "tells you something very specific," Sipser says. "It tells you that if you're going to look for an algorithm that's going to work in every case and give you the best solution, you're doomed: don't even try. That's useful information."

Provided by Massachusetts Institute of Technology (news : web)