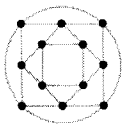
Accidental Algorithms

Brian Hayes

WHY ARE SOME computational problems so hard and others easy? This may sound like a childish, whining question, to be dismissed with a shrug or a wisecrack, but if you dress it up in the fancy jargon of computational complexity theory, it becomes quite a serious and grownup question: Is P equal to NP? An answer—accompanied by a proof—will get you a million bucks from the Clay Mathematics Institute.

I'll return in a moment to P and NP, but first an example, which offers a glimpse of the mystery lurking beneath the surface of hard and easy problems. Consider a mathematical graph, a collection of vertices (represented by dots) and edges (lines that connect the dots). Here's a nicely symmetrical example:



Is it possible to construct a path that traverses each edge exactly once and returns to the starting point? For any graph with a finite number of edges, we could answer such a question by brute force: Simply list all possible paths and check to see whether any of them meet the stated conditions. But there's a better way. In 1736 Leonhard Euler proved that the desired path (now called an Eulerian circuit) exists if and only if every vertex is the end point of an even number of edges. We can check whether a graph has this property without any laborious enumeration of pathways.

Now take the same graph and ask a slightly different question: Is there a circuit that passes through every *vertex* exactly once? This problem was posed in 1858 by William Rowan Hamilton, and the path is called a Hamiltonian circuit. Again we can get the answer by brute force. But in this case there is no trick like Euler's; no one knows any method that gives the correct answer for all graphs and does so substantially quicker than exhaustive search. Superficially, the two problems look almost identical, but Hamilton's version is far harder. Why? Is it because no shortcut solution exists, or have we not yet been clever enough to find one?

A strange new family of algorithms probes the boundary between easy and hard problems

Most computer scientists and mathematicians believe that Hamilton's problem really is harder, and no shortcut algorithm will ever be found—but that's just a conjecture, supported by experience and intuition but not by proof. Contrarians argue that we've hardly begun to explore the space of all possible algorithms, and new problem-solving techniques could turn up at any time. Before 1736, the Eulerian-circuit problem also looked hard.

What prompts me to write on this theme is a new and wholly unexpected family of algorithms that provide efficient methods for several problems that previously had only brute-force solutions. The algorithms were invented by Leslie G. Valiant of Harvard University, with extensive further contributions by Jin-Yi Cai of the University of Wisconsin. Valiant named the methods "holographic algorithms," but he also refers to them as "accidental algorithms," emphasizing their capricious, rabbit-from-the-hat quality; they seem to pluck answers from a tangle of unlikely coincidences and cancellations. I am reminded of the famous Sidney Harris cartoon in which a long series of equations on a blackboard hinges on the notation "Then a miracle occurs."

The Coffee-Break Criterion

For most of us, the boundary between fast and slow computations is clearly marked: A computation is slow if it's not finished when you come back from a coffee break. Computer science formalizes this definition in terms of polynomial-time and exponential-time algorithms.

Suppose you are running a computer program whose input is a list of n numbers. The program might be sorting the numbers, or finding their greatest common divisor, or generating permutations of them. No matter what the task, the running time of the program will likely depend in some way on n, the length of the list (or, more precisely, on the total number of bits needed to represent the numbers). Perhaps the time needed to process n items grows as n^2. Thus as n increases from 10 to 20 to 30, the running time rises from 100 to 400 to 900. Now consider a program whose running time is equal to 2^n. In this case, as the size of the input grows from 10 to 20 to 30, the running time leaps from a thousand to a million to a billion. You're going to be drinking a lot of coffee.

The function n^2 is an example of a polynomial; 2^n denotes an exponential. The distinction between these categories of functions marks the great divide of computational complexity theory. Roughly speaking, polynomial algorithms are fast and efficient; exponential algorithms are too slow to bother with. To speak a little less roughly: When n becomes large enough, any

Brian Hayes is Senior Writer for American Scientist. *Additional material related to the "Computing Science" column appears in Hayes's Weblog at http://bit-player.org. Address: 211 Dacian Avenue, Durham, NC 27701. Internet: bhayes@amsci.org*