

It has become the lingua franca of document-preparation languages for the sciences—I would guess that close to 75% of all computer science papers are written in LATEX. I'm much better known as the author of LATEX than for my real work. People are surprised that I'm not devoting my life to LATEX.

Redefining time for distributed systems, taming traitorous processes, verifying safety-critical code, creating document-preparation languages—Lamport has played a pivotal role in all these fields. Why him and why these problems? His explanation is modest, but comprehensive.

When I look back on my work, most of it seems like dumb luck—I happened to be looking at the right problem, at the right time, having the right background.

Stephen Cook and Leonid Levin


A GOOD SOLUTION IS HARD TO FIND

The idea that there won't be an algorithm to solve it—this is something fundamental that won't ever change—that idea appeals to me.

—STEPHEN COOK

*Sometimes it is good that some things are impossible.
I am happy there are many things that nobody can do to me.*

—LEONID LEVIN

 An average 15-year-old can understand a proof of the Pythagorean theorem, but the Greek geometers burnt offerings to the gods when they discovered it. Millions of people can whistle parts of Beethoven's *Ninth Symphony*, but few can aspire to his musical genius. There is a basic asymmetry in every good idea—it is easier to recognize than to discover. In computational theory, this asymmetry is a central fact of life, or so it appears.

Consider the well-known traveling salesman problem. Suppose our salesman, Willy Loman, has a sales route, a travel budget, and a booklet of airline prices. He needs to travel from Boston to the nine other cities shown in Figure 1 and back to Boston, staying within his



Figure 1 The traveling salesman problem. The task is to determine whether Willy the salesman can travel from Boston to all the other cities and return to Boston within a certain budget.

budget. He asks you to figure out the route he should take. If his budget is large, then your task is easy. If his budget is small, you may have to work very hard. It might not even be possible to stay within his budget. Either way, you may have to consider every possible ordering of cities, and for even this small number of cities, there are approximately 100,000 possible routes!

If there are 3 cities A, B, and C having direct flights between every pair, then there are 6 possible orderings (or sequences for visiting each city): ABC, ACB, BAC, BCA, CAB, and CBA. If there are 4 cities, then there are 24 orderings: the fourth city D can be in four positions with respect to each of the six possible orderings of A, B, and C. For example, with respect to the ordering BCA, we can have the following four orderings: DBCA, BDCA, BCDA, and BCAD.

The number of orderings is known as the factorial (expressed by the ! symbol). Factorials get very big, very fast: 4 factorial or $4! = 4 \times 3 \times 2 \times 1 = 24$; $5! = 5 \times 4!$ which is $5 \times 24 = 120$. $6! = 720$. And so it goes. Computers are good at handling many possibil-

ities, of course, but the sheer number of possibilities overwhelms any amount of computational resources. As Stephen Cook points out:

If there are 100 cities you have to evaluate 100 factorial tours. No computer is going to be able to try out 100 factorial tours. It's hard for people to understand that. If you do some simple calculations, you realize that if you had all the electrons in the solar system working on it at frequencies comparable with their spins, it would still take until the sun burnt out to find it. The basic point to get across is that there are things you just can't do in practice.

The asymmetry in the salesman problem is this: If someone gives Willy a particular route, it's easy for him to check whether or not it meets his budget constraints. A good solution for Willy may be hard to find, but it is easy to verify.

In the 1950s and 1960s many problems in design, operations research, and artificial intelligence seemed to have this hard-to-find, easy-to-verify property. Many members of the computer science community suspected that these problems belonged to a common mathematical family.

Two scientists working simultaneously and independently, the American Steve Cook and a Russian, Leonid Levin, described this family in the early 1970s. Now known as *NP-complete* problems (for reasons you'll see below), their numbers are huge and growing. For working computer scientists, proving that a problem belongs to this family is tantamount to demonstrating that finding an exact solution for it is impossible—the customer must be willing to accept an approximate solution.

In the Western computational tradition, the notion of the difficulty of computation has roots in logic and Alan Turing's "uncomputability" results. Michael Rabin first formalized the notion of the inherent difficulty of computable problems in 1959. When he defined NP-complete problems in 1971, Steve Cook was following this tradition.

Starting in the late 1950s, Russia's operations research community had informally characterized certain optimization problems as requiring *perebor*, which in Russian means "brute force," or exhaustive. Russian scientists put into the category of *perebor* search those

problems which required a search of all or nearly all alternatives before one could be sure of finding the best solution. For example, trying all alternatives for even a modest-sized version of problems like the traveling salesman problem can become infeasible.

The theory behind *perebor* problems, however, was inspired not by logic, as in the Western tradition, but by Andrei Kolmogorov's notion of the relationship between the randomness of a sequence of characters and the difficulty in describing it. This approach drew upon the American Claude Shannon's information theory of the late 1940s. After that time, communication in mathematical sciences between the Soviet Union and the West was sporadic at best.¹

In talks given in 1971 at Soviet universities and in a short journal paper in 1973, Leonid Levin showed the relationship between *perebor* and Kolmogorov's ideas. One of the results was a characterization of NP-complete problems.

This merely started the discussion. Neither Cook nor Levin nor any scientist since then has actually proved that problems in this family are extremely difficult. They merely showed that if any problem in the family is difficult, then they all are. They thus posed the major open problem in theoretical computer science today: Do the NP-complete problems require exhaustive search or don't they? To put it another way: An algorithm that could solve problems like the traveling salesman problem while exploring only a small fraction of the possible routes would change computer science history.

Stephen Cook: Logic and the Western Tradition

Stephen Cook was born in Buffalo, New York, in 1939. Cook's father, a chemist for Union Carbide, also taught at the University of Buffalo. He had always dreamed of living in the country, and when

¹The extent of concurrent invention resulting from this mutual isolation is astounding. This chapter is about the independent discoveries of Levin and Cook, but two other independent results come to mind immediately. While Rabin worked out the foundations of complexity in 1959, Gregorii Samuilovich Tseitin was doing similar work in Russia. In 1969 Gregory Chaitin in New York defined a measure of randomness similar to Kolmogorov's, but a few years later.

Stephen was ten, the family moved to a dairy farm in Clarence, New York. They rented the land to a farmer, but kept a cow (which Steve milked) and some other animals.

I had a normal interest in chess and so on—nothing special. I did well in math in school but it was just an ordinary rural high school. I certainly never thought about being a mathematician. My mother's uncle Arthur was a mathematics professor in Wichita. That's the only mathematics ancestor that I know of.

Clarence, New York, was also the hometown of Wilson Greatbatch, the inventor of the implantable pacemaker. He inspired Cook's teenage ambition to become an electrical engineer.

I worked for him in the summer in the attic of his barn where he had a little shop. Transistors were new—this was in the fifties—and he was experimenting with transistor circuits. And I would help him solder up the circuits. I found it quite fascinating.

When I started out at the University of Michigan my major was what they called science engineering, but my interest was electrical engineering. My freshman year [1957] I took a one-credit programming course with Bernard Galler.

Cook and a friend concocted a program to test Goldbach's famous conjecture that every even integer greater than 3 is the sum of two primes. As far as they could compute, the conjecture held. (The conjecture remains open, because it is very hard to prove universal properties about prime numbers; on the other hand, nobody has found a counterexample.)

Cook completed a major in mathematics, but he also learned enough computational theory to know about inherently impossible problems such as Turing's halting problem (discussed in the chapter on Rabin). Cook then entered Harvard's Ph.D. program in mathematics with the intention of studying algebra. But his most influential teacher turned out to be Hao Wang of the applied science division. Trained in mathematical logic and philosophy, Wang worked on automatic theorem proving—the discovery of proofs by the computer itself.

Another influence was complexity theory, which had just been given a mathematical foundation by Michael Rabin's 1959 paper.

Many of complexity theory's seminal figures, including Rabin, Juris Hartmanis, and R. E. Stearns, delivered talks to eager graduate students like Cook.

It seemed like a very natural and basic question. Obviously there are problems that are solvable in principle by algorithms but not in practice, because the sun burns out before you solve them. So, it's just a very natural question to ask about the inherent difficulty of problems.

Before real computers existed, you couldn't execute algorithms except by hand. The process was so tedious that the question of complexity was less interesting. Now that we had these powerful machines to help us and they seemed like an enormously powerful tool—thousands of operations per second—it was very natural to ask just what sorts of problems could you really solve.

Cook's advisor, Hao Wang, added a logical perspective to these considerations.

I was very aware of his [Wang's] ideas and his techniques. My result for NP-complete problems is an analog of his. Turing and Wang were talking about the predicate calculus. I was talking about propositional calculus.

Predicate calculus and propositional calculus are two languages used in mathematical logic. Wang studied the complexity of the "satisfiability problem" for the predicate calculus. Cook later became interested in satisfiability for the propositional calculus. To understand the distinction, consider the following examples.

Predicate calculus makes statements about groups of individuals. For example, the assertion "All Olympic athletes are fit" becomes: for all x OlympicAthlete(x) implies fit(x).

Predicate calculus permits you to substitute particular individuals for x . For example, if you know that Achilles is an Olympic athlete, that is, that "Olympicathlete(Achilles)" is true, then the above formula leads to the conclusion that "fit(Achilles)" is also true.

By contrast, propositional calculus, which Cook studied, is a simpler language that only allows you to make assertions about individuals. For example, "Tweety is a bird." The rules of proposi-

tional calculus allow you to infer new propositions from old ones. For example, if the propositional sentences "Either Tweety is a bird or Luke is a gazelle" and "Luke is not a gazelle" are both true, then the rules allow you to infer that "Tweety is a bird" is true.

Satisfiability

An important question for both logical languages is whether there is an assignment of true and false values that makes a given formula true. If there is, then the formula is *satisfiable*; otherwise it is *unsatisfiable*. For example, " x and not y " is satisfiable because it is true whenever x is assigned the value true and y is assigned the value false.

On the other hand, " x and not y and not x " is unsatisfiable because either x or not x must be false under any assignment of truth values, making the whole assertion false.

Alonzo Church and Alan Turing had shown that determining whether certain formulas in predicate calculus were satisfiable was computationally impossible even in infinite time. Cook would show that satisfiability for propositional calculus may require trying a large percentage of the possible truth assignments.

How many possible assignments exist? If there is one propositional variable (such as x), then there are two possible assignments (x is true or x is false). When there are two variables (x and y), then there are four possible assignments: x true, y true; x true, y false; x false, y true; and x false, y false. In general, if there are n variables, then there are 2^n possible assignments. This comes to one thousand for 10 variables, one million for 20 variables, one billion for 30 variables, one trillion for 40 variables. It keeps growing by a factor of 1000 for every 10 additional variables.

The number of possibilities is said to rise exponentially with the number of variables n , because n is in the exponent. If an algorithm tests each possibility separately, then the time to execute the algorithm is also exponential and the algorithm is said to take exponential time. By contrast, in a polynomial-time algorithm, time requirements rise according to an expression of n raised to some fixed power, where n is the size of the problem.

For example, the polynomial time expression n^3 has the value 1000 when n is 10. When $n = 20$, the value is only 8000; it's 27,000 for $n = 30$ and 64,000 for $n = 40$. Compare this with the exponential growth of one trillion possible assignments for 40 variables!

Defining NP-Completeness

After completing his doctoral thesis at Harvard, Cook spent a short period at the University of California at Berkeley before moving to the University of Toronto.

In 1971, his ideas about satisfiability coalesced in a paper for the Third Annual Symposium of the ACM (Association for Computing Machinery) on the Theory of Computing. Cook discussed problems for which a possible solution—a “candidate”—could be checked in polynomial time. Since it is not always possible to decide which candidate is a good one to try, a program may have to guess a solution. For this reason, Cook called these problems nondeterministic polynomial, or NP, problems. The guessing part is nondeterministic and the checking part is polynomial.

The traveling salesman problem and satisfiability both have this property, since you can quickly check whether a candidate travel plan satisfies the salesman's budget or whether a candidate truth assignment yields a true formula. The big question is whether it is possible to find the appropriate candidate in polynomial time.

Cook showed that the satisfiability problem is among the hardest problems in NP. Specifically, he showed that if satisfiability has a polynomial-time algorithm, then so does any problem in NP. That made satisfiability an NP-complete problem. In practice, then, showing that a problem is NP-complete implies that it is hard to solve, though you could recognize a solution fast if you found one. And if by some lucky quirk of fate someone found an efficient algorithm for one NP-complete problem, then that algorithm could be used for all NP-complete problems.

Shortly after Cook's paper appeared, Richard Karp of the University of California, Berkeley, demonstrated that twenty-one other problems are NP-complete, including a problem closely related to the traveling salesman problem. He argued by a method known as

USING GENES TO FIND SHORTEST PATHS

Recently, Leonard Adleman of USC discovered a way to solve the traveling salesman problem using strands of DNA. DNA consists of two strands of chemical constituents known as nucleotides. There are four kinds, conventionally labeled A, C, T and G. Certain pairs of these bond together (specifically, A with T and C with G) giving DNA its characteristic double-stranded shape.

Adleman represented each city by a sequence on a single strand. If $X^{\text{in}} X^{\text{out}}$ is the strand for city X and $Y^{\text{in}} Y^{\text{out}}$ is the strand for city Y, then the flight from X to Y would be represented by $x^{\text{out}} dy^{\text{in}}$. Here x^{out} pairs with X^{out} (each A in one is at the same position as a T in the other and similarly for C and G); y^{in} pairs with Y^{in} . The d has a length that is proportional to the cost to go from X to Y.

Mixing the city strands and trip strands together, he used well-established biological lab techniques to find the shortest double strands that included all cities. This is the path the salesman should take.

This remarkably clever approach allows one to think of extremely small, low-power computing devices. Does this approach offer a general solution to all NP-completeness problems? Unfortunately not. Solving the traveling salesman problem for 1000 cities would require far more molecules than there are atoms in the known universe.

reducibility: if any of these problems could be solved fast, then so could satisfiability. Therefore, these problems were at least as hard as satisfiability—and vice versa. Unfortunately for science, none of this proves that these problems are in fact difficult.

A (Nearly) Perpetual Discussion Machine

So, you might ask, if no problem was really proved to be hard, what was accomplished? An analogy may help. The patent office will immediately reject a patent application if it promises perpetual motion. The patent examiners use a reducibility argument: if the machine worked as promised, then the hypothesis of energy conservation would be false. The examiners have a lot of faith in that hypothesis.

The perpetual motion machine is analogous to an NP-complete problem. If any such problem could be solved fast (in polynomial time), then all could, so the working hypothesis that NP-complete

problems require exhaustive search would be false. Computer scientists have a lot of faith in the hypothesis that NP-complete problems are really difficult. Therefore if a computer scientist encounters a problem for which he can't give a fast algorithm, he may try to show that the problem is NP-complete, implicitly invoking the hypothesis that such problems really are difficult. Cook sums up that position.

NP-Complete problems probably do not have polynomial-time solutions. We know that all NP problems have exponential-time solutions. They may also have polynomial-time solutions, but that's the big open question. If one NP-complete problem such as satisfiability has a polynomial-time solution, then they all do. They're all interreducible—that's the importance of the question.

Since Karp's work, researchers around the world have shown thousands of problems to be NP-complete. Typical examples are the optimal geometric layout of telephone networks, or the best way to play a game like checkers. Cook was taken aback by the number of NP-complete problems.

I thought NP-completeness was an interesting idea—I didn't quite realize its potential impact.

Leonid Levin: The Kolmogorov Tradition

While Cook was contemplating complexity theory at Harvard in the 1960s, a young high school student was learning about *perebor* and Kolmogorov complexity in the Soviet Union.

Leonid Levin was born in 1948 in Dnepropetrovsk, an industrial city in the heartland of Ukraine. His father, Anatoly Levin, first taught Russian language and literature in high school, then completed a Ph.D in education to teach at the university level. Leonid's mother, Anna Erenburg, was an industrial architect who designed bridges. Early on, Levin became interested in science and mathematics.

I was frustrated when I learned Mendeleyev's table [the periodic table of elements] It was not quite as regular as I would like it to be. I would work hard and rewrite and rewrite the table. I mostly would put it closer to how I later saw it in America.



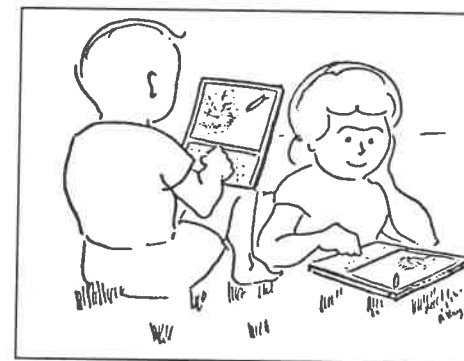
John Backus helping his daughter compile a sand castle in 1961.



John Backus thinking early thoughts about form and functional programming in 1969.



A young John McCarthy around the time of his invention of LISP.



Kay's original Dynabook drawing.



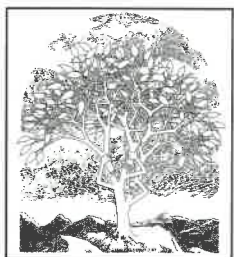
A more recent photograph of John McCarthy. (Courtesy Department of Computer Science, Stanford University.)



Alan Kay around the time of inventing Smalltalk.



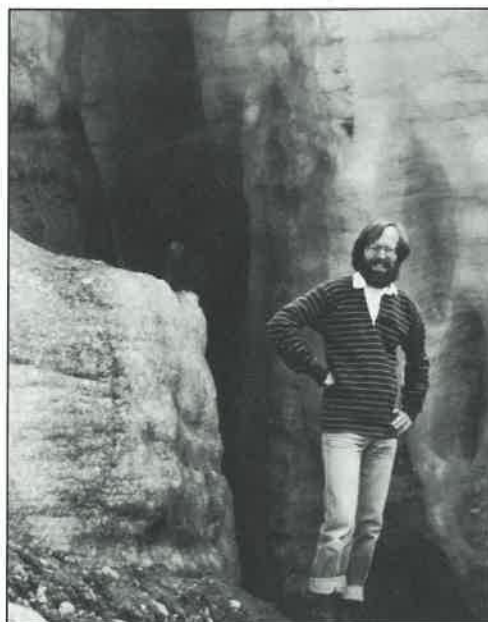
A young Bob Tarjan in southern California.



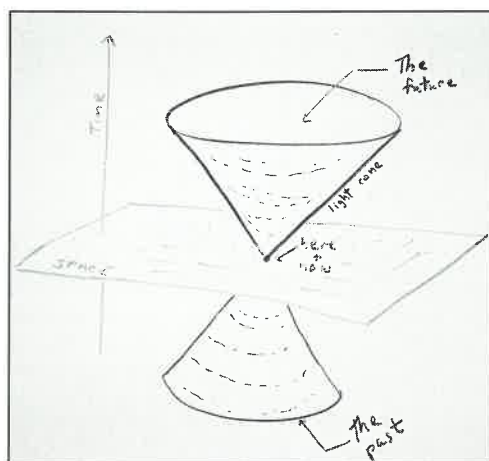
An allegorical Tarjan-style self-adjusting tree. (From J. Stolphi, 1987. *Communications of the ACM*, Vol. 30, No. 3, March 1987, p. 204. © 1987. Reprinted by permission.)



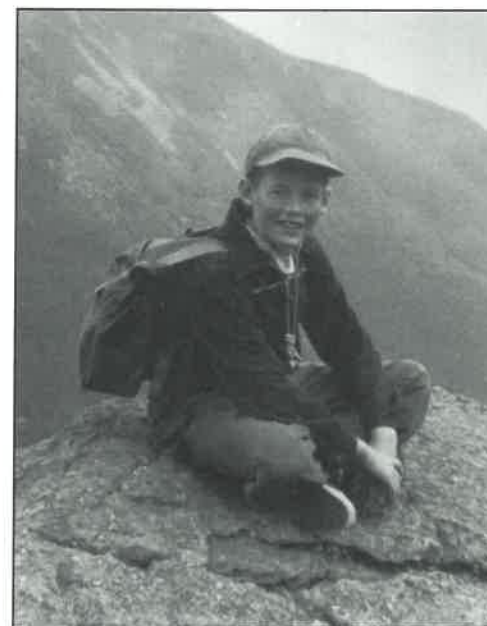
Leslie Lamport, from a recent ID picture.



Bob Tarjan on a hike through a nonplanar landscape.



The relativistic light cone drawing that illustrates Lamport clocks. (Courtesy Leslie Lamport.)



Steve Cook sitting at a local maximum in the Adirondacks.



Leonid Levin, after a perebor (brute force) haircut in 1967.



Levin, during one of his yearly visits to Jerusalem, in 1993.



Cook making a conjecture about P vs. NP on the white board.



Edward Feigenbaum in his office at Stanford in front of a modern-day Monroe calculator. (Courtesy Department of Computer Science, Stanford University.)

Doug Lenat with daughter Nicole.



Lenat strolling in Antarctica, 1992.

Levin also experimented with chemistry by mixing concoctions in the bathroom of the family's one-bedroom apartment. His father bought him a series of books by the popular Russian science writer Yakov Perelman and also encouraged him to participate in the Olympiads. Olympiads were competitions designed by the Soviet government to locate students with special talent in math and science. Winners advanced from local to regional levels and then were selected for boarding schools specializing in the sciences.

In Russia in general in the late fifties and early sixties there was a great popular enthusiasm about science. The Olympiads were very popular. Most kids from around the country participated. And then if you won and you were not Jewish, you could go to the international Olympiad as well.

Levin placed first in the Kiev City Olympiad in physics and was sent to the boarding school for physics and math at Kiev University. One day, with all the pomp and ceremony accorded to Soviet scientific "royalty," Kolmogorov himself visited the school. Levin was 15.

It was a great event when Kolmogorov visited. He met with all the bosses and then he met with the kids. It was in a huge room. He would talk and give a problem and we were told to raise our hands and quite quickly it turned out to be a competition between me and one other boy.

The problems Kolmogorov posed to the young Levin were good preparation for those he would later encounter in computer science. For an example of one of Levin's solutions, see the box on p. 150.

Kolmogorov didn't forget Levin's ability to answer the puzzles and later invited Levin to his own school at Moscow University.

He taught a small group of two dozen kids. He would make a musical concert for us—he liked music very much. He analyzed poetry. He knew very many things and lots of them on a professional level.

Kolmogorov started his career as a historian and I think he mentioned somewhere that he proved certain conjectures and then he presented his work and they said it's excellent but we need more proofs of that. Then he decided to become a mathematician where one proof was enough.

LEVIN'S SOLUTION TO KOLMOGOROV'S PUZZLE

THE PROBLEM

Suppose all words, all sequences of characters, are of one of two classes—those that are fit to print (decent) and those that are not (indecent). Now given an infinite sequence of characters, can you always break it into finite sequences so all words except perhaps the first one belong to the same class?

THE SOLUTION

Yes. Call a sequence of letters *prefix-decent* if all its initial segments are decent. Suppose an infinite sequence $A = a^1, a^2, \dots$ has an upper limit n , so that cutting off a prefix of more than n letters always leaves a non-prefix-decent infinite suffix of A (i.e. at least one prefix of that suffix is indecent). Then cut off the first segment with $n + 1$ letters. The remaining (not prefix-decent) sequence has an indecent prefix: cut it off. Repeat the previous sentence forever and you have broken A into finite prefixes all of which (except, maybe, the first one) are indecent. To see that you can do this infinite repetition, suppose you could not. Then after the k th indecent prefix, any subsequent prefix of the remaining sequence would have to be decent, so you could then cut a prefix until that point and get (contrary to the assumption) a prefix-decent sequence.

If no such n exists, cut off a prefix leaving a prefix-decent infinite suffix. You can repeat the previous sentence forever, otherwise the combined length of the first several prefixes would be the limit n which we assume does not exist. This does not mean that subsequent cuts can be arbitrary, because you might leave a prefix-indecent suffix in that case. But it must be possible to cut it off so that there is a prefix-decent suffix each time (because no such n exists). Now A is broken into finite prefixes all of which (except, maybe, the first) are decent, since they are prefixes of prefix-decent sequences.

Levin immersed himself in mathematics to the exclusion of nearly everything else. He played chess, but refused to study piano.

I did like to read Russian fiction. I had a very good memory and I would memorize huge pieces of poetry by heart just for sport. Later on, I lost this memory because I was afraid it would limit my imagination.

Although the Soviet system nurtured young scientists in mathematics and physics, it had a history of hostility toward computer science.

In the early fifties, computer science was kind of an illegal topic in the U.S.S.R. where everything was supposed to be consistent with the teachings of Marx (often interpreted with even greater stupidity than is native to them).

I think some of the Soviet philosophers were irritated by Norbert Wiener—he was a great mathematician when he was young but in his old age he wrote a lot of nonsense about computer science. One of the “pop-science” ideas of his late period was cybernetics, a buzzword which got to denote computer science in Russia. Even though this was kind of harmless nonsense, the Soviet philosophers somehow found it at odds with Marxism.

By the 1960s, however, the Soviet military insisted that computer science be taught because of its evident applications. They even began to train young students in the use of computers, and Levin's first exposure to computers came when he served in the quasi-military units that all undergraduates had to join.

We . . . worked with some very ancient computers, [using] punch tapes, lots of lights, and panels. It was very impressive. In addition, we learned some kind of probabilistic things. Suppose a rocket is flying. What is the probability it would hit this or that?

I hated the military of course. But the mathematics we were studying were more interesting than our regular university computer course on Algol—just the language nothing else.

Kolmogorov Complexity

For his 1971 doctoral thesis, Levin wrote about Kolmogorov complexity. Kolmogorov served as his advisor and approved the thesis as did Levin's review committee, which included other great Russian mathematicians. Nevertheless, Levin was denied his Ph.D.

At that time, almost all Soviet youth belonged to Comsomol—an organization like the Communist party but for kids. It was teaching us, through various activities, to love our government. I would sabotage some of this.

For the things I did, there were punishments but it would be bad for them as well: it would show the higher-ups that there was something wrong. So they would try to pretend that nothing happened and I could get away with this.

But I didn't notice that the times had changed after Czechoslovakia—somehow I didn't want to recognize this.

Noisy and arrogant, I was an excellent scapegoat which the university Communist authorities needed at the time.

The biggest blow was not the denial [of the Ph.D.] itself but the quite rare use of explicit political words in the formal justification of the decision. This phrasing prevented me from trying again to get my Ph.D. and eventually led to my emigration.

Before he emigrated to the United States, Levin published a paper in 1973 on Universal Sequential Search Problems in a Soviet journal, *Problemy Peredachi Informatsii*. He outlined a formal relationship between *perebor* problems and Kolmogorov information theory. Kolmogorov information theory explored the relationship between the “randomness” of a sequence of characters (say 0s and 1s) and the length of its description. For example, it is easy to describe a string of a million consecutive 1s in a few words. (We just did so.) Similarly, one can describe any repetitive pattern in a short way, e.g., 00111 repeated a million times.

In Kolmogorov's terminology, long patterns with short descriptions are not random. A fundamental conclusion from Kolmogorov's work was that the amount of randomness depends little on the mathematical language used. For any definition, most sequences are “random”—they require a description at least as long as the sequence itself.

(You can prove this for yourself by a counting argument. How many numbers are there consisting of n binary digits? 2^n . How many different numbers can be described by a program that can be encoded in $m < n$ binary digits? Only 2^m . If $m = n - 10$, for example, then 2^m is only 1/1000 as large as 2^n , so 99.9 percent of all numbers of length n require more than $n - 10$ binary digits to describe them.)

Levin looked at the problem of how difficult it would be to find programs that could describe long strings.

This amounted to the inherent difficulty of the problem of finding short, fast programs producing a given string. I got an idea to re-

duce tiling [a problem of spatial packing] to it but failed. I succeeded, though, in proving the universality of a similar-looking problem: finding small depth-2 circuits for partial boolean functions [a problem in circuit design]. I also did it for five other problems including satisfiability, set-cover, graph onto mapping, and embedding.

Isolated in the Soviet Union, Levin was unaware that Cook and Karp had independently shown most of the same problems to be “universal”—or, as they put it, NP-complete.

The Cook and Karp papers were not known in Russia for several years since the proceedings they appeared in were not received by any Russian library or institution. Restrictions on travel and currency prevented this work from reaching Russia by private channels. Later, I was indeed surprised by Karp's work since I did not expect so many wonderful problems were NP-complete.

Compared with the reaction to Cook's work, the reaction to Levin's paper in the Soviet Union was positive but restrained.

Some people were interested. I didn't know if this was a genuine interest or sympathy to my political troubles: people would sometimes be more gentle to me than I deserved. There was no publicity. Then in the late 1970s in Russia people began to get excited after hearing about Karp's work.

Cook and Levin—After NP-Complete

In the mid-1970s, while Levin was sorting out his problems with the Soviet authorities, Cook began looking at a new problem: What is the trade-off between time and memory?

Anybody with a cluttered office or room knows how hard it can be to find things or to organize a collection of papers in a small space. With A. Borodin of the University of Toronto, Cook studied the analogous problem in computers.

Think of the Internal Revenue Service trying to sort a file of 100 million tax returns. Fast methods are known which require a lot of computer memory and slow methods are known which require little

computer memory. We proved that no method can be simultaneously time- and memory-efficient.

[For mathematically adept readers: the product of the time it takes and the memory space used must be at least proportional to $(n^2)/(\log n)$, where n is the number of returns to be sorted.]

To the lay mind, if a scientist states that a problem is impossible or infeasible to solve, that's tantamount to a hopeless situation. Cook reports that his mentor from adolescence, the inventor Wilson Greatbatch, a devout Presbyter, often teased him about his chosen field.

Since I got my reputation proving impossibility, he was always very skeptical. He would say, "Making any progress proving things impossible?" [But] it's not that the problem you're trying to solve is impossible. There are always ways around it. You settle for less. You don't really want a perpetual motion machine. You're willing to cheat; you have to bring in an energy source, that's all.

With NP-complete problems, you're going to use heuristics, shortcuts, and approximations and all kind of ways around it. People have studied those. I think the effect of showing NP-complete is to direct people's energies to solving problems that are going to work. I think it's positive and constructive. The thing to emphasize is that although the technology is changing rapidly, there are underlying principles that remain and there are limits.

Cook has continued his fundamental work in algorithms for propositional logic. His current area of interest is finding short (polynomial-length) proofs for propositional logic formulas.

Levin is now a professor of computer science at Boston University, where he has helped develop a theory of "transparent" proofs in collaboration with Mario Szegedy, Laszlo Babai, and Lance Fortnow of the University of Chicago.

These mathematicians have developed a method for writing proofs with the following strange property: if the proof has an error, then a mathematical test on any tiny portion of the proof will reveal an error a certain proportion of the time. For the sake of discussion, let's say it will reveal an error with probability $1/2$ each time. Suppose

now that you have just written a proof in this way and want to test its correctness. If you do 40 tests on different parts and find no error, then the probability you have written an incorrect proof is less than 1 in a trillion.

It's a little like holographic photography. Every part of the photograph contains information about every other part. So does every part of the proof.

The Big Question

Recently, Andrew Wiles of Princeton came up with a promising approach for resolving Fermat's last theorem, a problem that has plagued scholars since the eighteenth century, when the French mathematician scribbled a teaser in the margin of one of his proofs. Could the same thing happen to prove that $NP = P$? In other words, could a problem whose solution can be *checked* in polynomial time also be *solved* in polynomial time? (See Figure 2.)

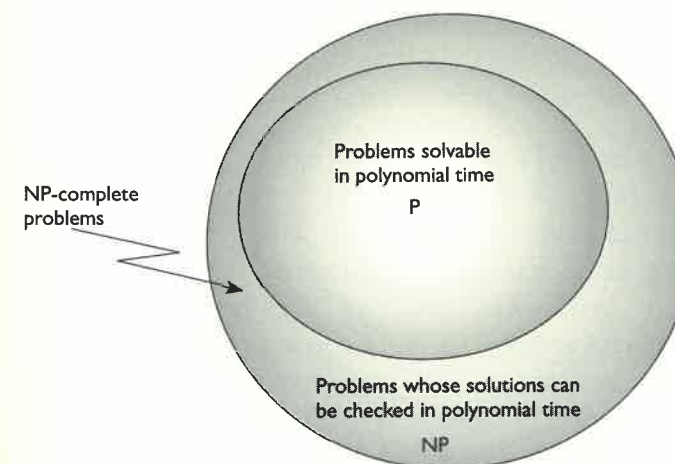


Figure 2 If a problem can be solved in some time, then its solution can be checked in that time. So P is included in NP . The major open theoretical problem in computer science is whether the thousands of important problems that are NP-complete are in fact solvable in polynomial time or do they require exponential time? That is, does P equal NP ?

Cook: The sad state of the mathematics of this field is that we can't prove these things. We can't prove P not equal to NP . So it may turn out that someone will invent some clever parallel algorithm that will solve an NP -complete problem in polynomial time.

It's hard to say what progress is in a case like this. It's not just around the corner, but there are partial results; people are attacking it from lots of different areas. And by the way it is possible that $P = NP$.

Levin: The fact that almost all mathematical conjectures that have been famous conjectures for many centuries have been solved is strong evidence that the solution is polynomial not exponential. Mathematicians often think that historical evidence is that NP is exponential. Historical evidence is quite strongly in the other direction.



A RCHITECTS

HOW TO BUILD BETTER MACHINES

Directing a huge project can be a headache. You must inspire confidence even when you have doubts. You must explore ideas that no one has ever tested. If you are too conservative, the market will sweep you by. If you are too radical, you will risk getting nothing out the door. If it works out, no success is sweeter.

For computer system architects—*system* here means a computer consisting of one or more processors, a network, and the software to run it—even success can have a short life. Other products may incorporate better technology, consigning your organizational ideas to a niche in the folklore of computer construction. Industries depend as much upon folklore as literature does, but authorship is often forgotten when a newer product repackages older ideas. Part 3 of this book presents three computer designers whose ideas are central to the present and future philosophies of computer construction.