Comp. Lang. Feb/87

THEORY & PRACTICE

Eight diversions to keep your Cray out of mischief on a cold winter's night

By Brian Hayes

Musing at the keyboard, trying to decide between "which" and "that" or between "while . . . do" and "repeat . . . until," I sometimes think of the little forty-legged prodigy deep inside the machine, and I suffer a twinge of guilt and shame. While I go off into a daydream, it twirls in a tightly wound loop, killing time at a frantic pace, asking again and again, perhaps half a million times per second, "Has he made up his mind *yet*?" What a waste.

Until quite recently, CPU cycles were a precious resource. With the cycles I fritter away in the course of a year's woolgathering, the hackers of the 1960s could have found another Mersenne prime or computed several thousand zeros of the Riemann zeta function. They would have been appalled at my profligacy. Some years ago I was present at a reunion of two such legendary code-tuners. Said one to the other: "Getting any lately?" The answer was: "Two MIPS in the daytime, more at night."

Computing has changed since then. With the next round of machines from Atari and Commodore you may well be able to buy two MIPS at K-Mart. We have cycles to burn. No one ever really gets enough, of course, but many of us get more than we know what to do with. Perhaps we are headed for some dreadful Malthusian catastrophe. The world population of computers is doubling every year or so. Will the supply of computing problems be able to keep up?

Twenty questions

The awful prospect of CPUs starving for data has long worried certain alert and public-spirited members of the computing community. For example, in 1972 a list of 20 "proposed computer problems, in order of increasing running time" was offered as a means of postponing the crisis. The list was compiled by Michael Beeler, Bill Gosper, and Rich Schroeppel, who were all then working in the AI Laboratory of the Massachusetts Institute of Technology, Cambridge, Mass. They published the list as part of a miscellaneous document called HAKMEM,[1] for "hackers' memorandum."

The HAKMEM list is biased toward puzzles, games, and mathematical recreations. For instance, it calls for solutions to four variations on the game of chess, the simplest of these being minichess, played on a 5-by-5 board. In this context "solving" a game means finding an unbeatable sequence of moves or else proving no such sequence exists. Solving the full game of chess is the 19th problem on the list; the authors note that there are about 10^{40} possible positions. The 20th problem is solving the game of go, which is said to have 10^{170} positions.

These last and most difficult challenges carry the warning "considered unfeasible," which seems a safe judgment. If we put a billion computers to work on chess, each examining a billion positions per second, it would still take more than 10,000 times the present age of the universe to finish the job. Go would take 10^{130} times longer still, which should be long enough to boggle any mind, human or mechanical.

But the rest of the list is not so reassuring. I happen to know that at least three of the problems have been solved, and others may have fallen without my knowledge. Problem 4 was to find the smallest squared square—a square completely filled with smaller squares, no two of which are allowed to be the same size. In 1978 the Dutch mathematician A.J.W. Duijvestijn found a solution with 21 squares and proved it is the smallest possible.

Problem 5 asked for a count of the magic squares of order five; that is, it asked how many ways the integers from 1 to 25 can be arranged in a square array so that all columns, rows, and major diagonals have the same sum. Schroeppel himself dispensed with this one just a year after the list was compiled. He found 68,826,306 distinct arrangements.

The third problem that must now be crossed off the list is the most disturbing because it was ranked among the most difficult—16th out of 20. The challenge was to solve three-dimensional tic-tac-toe on a 4-by-4-by-4 board. Oren Patashnik, using a computer dictionary of several thousand opening plays, proved that the first player can always win.

Given this depletion of the world's reserve of hard problems, it seems only prudent to extend the list as quickly as we can. I have taken it upon myself to suggest eight new problems. Readers are invited to nominate more.

Hard choices

Since the HAKMEM group so thoroughly covered mathematical games, I have deliberately avoided them and looked for diversion elsewhere. In selecting problems I kept four criteria in mind.

First, I looked for problems and tasks that can be stated with at least fair precision; it should be possible to decide without great controversy when the problem has been solved or the task has been accomplished. "Write a program that composes music better than Mozart's" is not a well-formulated problem. Who's to judge?

Second, I considered only specific instances of problems, not entire classes. There is no question the traveling-salesman problem is "hard" in that the computing time needed to plan a tour grows exponentially with the number of cities to be visited. But the difficulty can be measured and expressed in absolute terms only when the number of cities is specified.

Third, for a problem to be included here it should be primarily a computing problem, not an intellectual one. The boundary between these categories is fuzzy, of course, and occasionally a problem jumps the fence. (In the 1950s machine translation of natural-language text was considered difficult because we lacked the computing resources; now the reason is that we lack the techniques.) Nevertheless, the distinction can generally be made. A problem is very likely intellectual if the main difficulty lies in writing a program to solve it; it is probably a computing problem if the hard part is finding a machine on which to run the program.

Finally, I favored "interesting" problems. Recognizing membership in this class is itself an intellectually hard problem, but there are some basic principles

on which most people seem likely to agree. Calculating the billionth digit in the decimal expansion of pi is not very interesting because there is no reason to think the billionth member of an infinite series will have any distinguishing traits. Discovering that the first billion digits of pi are not randomly distributed would be much more interesting, but the search for such patterns is a quest, not a problem. It very likely has no end.

I attempted to arrange the problems in order of increasing difficulty, but I would not attempt to defend the ranking in detail. It would surprise me to learn that the first problem had turned out to be the hardest or the last one the easiest, but between those extremes much shuffling is possible.

In some cases I attempted a quantitative estimate of the amount of computing needed to solve a problem. The unit of measurement is the Cray-year: the number of instructions a Cray 1 can execute in a year. This number is itself quite difficult to determine because it depends crucially on what the instructions are. I have taken it to be 10^{15}, a very crude estimate based on the Cray 1 clock rate of roughly 100 MHz.

Here are the eight problems. If you solve any of them, please send me a postcard (c/o Editor, *COMPUTER LANGUAGE*).

Problem 1

Decompile a large file of machine code to produce a corresponding program in a high-level language such as Pascal or LISP. The test of success is to recompile the source code created in this way: the result should be machine code that is functionally equivalent (although not necessarily identical) to the original.

One may be tempted to respond that this task is trivially easy. After all, disassemblers, which produce an assembly-language listing from machine code, are commonplace and fast. Another response is that the task is impossible, because the transformation from source code to machine code is noninvertible. Information is irretrievably lost in the process of compilation, and you can no more decompile a program than you can unadd two numbers.

I take the safe (and dull) position that the truth lies somewhere between these extremes. Decompiling is not as easy as disassembly because there is no one-to-one mapping between machine-code instructions and high-level language statements (as there generally is between machine code and assembler mnemonics). On the other hand, although it is true that some aspects of the source code cannot be reconstructed—there is no way to restore comments, for example, or variable names—I suspect that all the essential elements of a program's structure can be recovered.

The problem as I have stated it does not ask for a perfect reversal of the compilation process, merely for a source file that can be recompiled to yield a program of equivalent function. The original programmer might have written a *case* statement that compiled to a series of test-and-branch instructions; the decompiler might then translate these instructions into a series of nested *if . . . then . . . else* statements. The change is of no consequence provided the two expressions are semantically equivalent.

Why is this a computationally hard problem? Basically because high-level languages are carefully designed to be easily compiled whereas machine languages are not designed with decompilation in mind. A compiler can march straight through a source program, never backtracking and looking ahead only a limited distance. A decompiler, on the other hand, might have to repeatedly revise its hypothesis about the structure and function of a program; the last instruction in a file could conceivably alter the interpretation of all those that precede it.

A practical decompiler would be a valuable tool in debugging, in porting programs from one machine to another, and in various forms of software larceny. As far as I know, a successful decompiler has never been written. Some interesting related work is being done by Valentin F. Turchin of the City College of New York, who is exploring the idea of what he calls a supercompiler.[2] Instead of translating a source program, the supercompiler paraphrases it, reading the text, digesting its meaning, and writing a new program in the target language.

Problem 2

Create a concordance to the human genome.

A number of biologists in the U.S. and Europe have proposed a major project to determine the sequence of base-pairs in all the DNA that makes up the genetic endowment of a human being.[3-5] The information would be stored in a computer data base that would be large but by no means beyond the bounds of current practice. There are an estimated three billion base-pairs (combinations of the four nucleotide bases designated *A*, *T*, *G*, and *C*) in the human genome. The most efficient possible encoding would use two bits per base-pair, which yields a data base of six billion bits or about 750MB.

The problem is not in storing the information but in making effective use of it. Consider what happens when the last few sequences are being added to the almost complete data base. It is important to learn whether each new sequence is related to any other sequence already on file. At first this appears to be a simple matter of string searching, for which there are efficient algorithms: just check to see if the same pattern of *A*s, *T*s, *G*s, and *C*s appears anywhere in the 750MB of data.

A conventional string search, however, will find only an exact match, which is most unlikely for any sequence longer than about a dozen base-pairs. What the biologist wants to know is whether any known sequence is similar to the new one, allowing for random changes from one base to another and for random insertions and deletions. Measuring resemblance in this way makes the problem much harder.

There is another complication. In many cases what is really of interest is not the similarity of two DNA sequences but the similarity of the proteins they encode. Each triplet of base-pairs in DNA specifies one amino acid unit in the protein, but there is much redundancy in the code: 64 possible triplets correspond to 20 amino acids. Thus both *CGT* and *AGA* specify the amino acid arginine, and this redundancy would have to be recognized by the search software. To make matters still worse, the DNA can be interpreted in six "reading frames," depending on where the grouping of base-pairs into triplets begins on either of the two strands of the double helix. All six reading frames must be checked for a match.

Roughly one-tenth of one percent of the genome is already on file in a facility at the Los Alamos National Laboratory in New Mexico. They use a Cray to do sequence searches and comparisons. A recent report mentions that one such computer run took a little under three hours. Linear extrapolation suggests that searching a data base 1,000 times larger would take about four Cray-months, but the rise in search time as a function of data base size may in fact be far worse than linear.

Problem 3

Given a detailed description of weather conditions at noon on Wednesday, predict whether or not it will rain on Saturday's picnic.

"That's done already," you object. "There's a three-day forecast every night on the TV news."

But do you believe what the weatherman tells you? The issue here is what is meant by the verb "predict." If an astronomer says the sun will rise at 6:33 tomorrow morning or the moon will eclipse the sun at a certain moment 50 years hence, few skeptics dare to doubt. If a meteorologist assures us the clouds will scatter by lunchtime, prudent people take along

an umbrella. What this problem asks for is reliable prediction.

Weather forecasting is one of the classic CPU-intensive applications of computers. It has a longer history than most people realize. During World War I the British meteorologist Lewis Fry Richardson[6] made an extraordinary, premature attempt to predict the weather by numerical methods based entirely on hand calculation (some of it done in a rest billet just behind the lines in France). His prediction was a flop— largely because of errors in his initial data—but his methods were sound. Richardson also proposed a marvelous parallel computer for weather prediction, in which the 64,000 computing elements were people equipped with adding machines.

The basic technique of numerical weather prediction is to define the state of the atmosphere at some initial moment and then apply the laws of fluid dynamics to determine the state at some future time. The initial data come from measurements of temperature, barometric pressure, wind velocity, and so forth made at a grid of points covering much of the earth's surface and extending to altitudes in the stratosphere. The accuracy of the prediction depends strongly on the density of the grid; unfortunately, so does the computational burden.[7,8]

At the European Center for Medium Range Weather Forecasts in Reading, England, most calculations are based on a grid with stations spaced about 200 km apart, with measurements made at 15 altitudes—a total of 273,630 points. The state of the atmosphere is computed for 15-minute intervals, so that a three-day forecast entails 288 calculations for each point and not quite 80 million calculations in all.

The center uses a Cray for this work, which takes from 15 to 20 minutes per forecast day. If the density of the grid were doubled in all three dimensions and the time interval were halved, a three-day forecast would require 1.2 billion calculations and would take 12 hours of CPU time. With another doubling of the model resolution, Saturday's weather report would not be ready until the following Thursday.

Problem 4

Find the prime factors of an arbitrary 100-digit number.

Readers who have kept up with developments in cryptography over the past 10 years will immediately recognize the significance of this problem. Several recent cipher systems derive their strength from the computational difficulty of factoring large numbers.[9,10]

The factoring problem is closely related to the testing of a number for primality (that is, showing that the number has no factors other than 1 and itself). There are efficient methods for finding primes (much better than the well-known sieve of Eratosthenes) and for distinguishing them from the nonprimes or composite numbers. Proving that a number can be factored, however, gives no clue whatever to the identity of the factors.

The brute-force approach to factoring is to attempt division by all possible prime factors, starting with 2, 3, 5, 7, 11, etc., and continuing up to the largest prime less than the square root of the number. If any division yields a remainder of 0, the divisor is a factor. The square root of a 100-digit number is approximately 10^{50}, and there are roughly 10^{48} primes less than 10^{50}. A few tricks are known to speed up the calculation somewhat, but none of them make enough of a difference to bring 100-digit numbers into range.

The odd and disarming thing about factoring is that no one has yet proved it to be an intrinsically hard problem. Someone may come up with a quick and efficient algorithm tomorrow.

Because of the recent practical interest in factoring for cryptography, several mathematicians have published estimates of the time needed to find the factors of large numbers. The estimates are not consistent with one another and cover a wide range—from 10^{16} years for a 126-digit number to a billion years for a 200-digit number down to 8,200 years for a 100-digit number. Although I am a little skeptical of this last value, the conservative course is to accept the smallest estimate and rate the problem at a few thousand Cray-years.

Incidentally, if you solve this problem, I don't want to hear about it. Tell it to the National Security Agency. As the old joke goes, there's no need to look up their number; just pick up the phone and start talking.

Problem 5

Given the amino acid sequence of a protein made up of at least 50 amino acids, predict the molecule's three-dimensional structure.

A protein is a linear polymer: an unbranched chain of amino acids. In most cases the chain folds up spontaneously to form a tangled, globular mass whose shape has everything to do with the molecule's biological function. Determining the amino acid sequence of the protein is comparatively easy, but figuring out the folding pattern is an arduous process that can take years or even decades. It would certainly make life—and the study of life—easier if the three-dimensional shape could be predicted from the sequence.

The basic principles that control the folding are straightforward enough.[11-13] The forces acting on each amino acid are the electrical attractions and repulsions of all the other amino acids and of the surrounding water molecules. The protein will tend to adopt whatever shape minimizes the energy associated with these forces. For example, amino acids that are repelled by water will tend to congregate in the interior of the tangled skein, where they are protected by the rest of the chain from the aqueous environment. Amino acids that attract each other will have a lower energy if they are close together.

The plan, then, is simple: calculate the energy for each possible configuration and pick the lowest. After all, that's what the protein itself does. But the protein is a very fast analog computer. For a chain of 50 amino acids there are an estimated 10^{50} distinguishable conformations. Even if the energy of each folding pattern could be calculated in a single machine cycle, the computation would take a vast stretch of Cray-eons.

To make predictions at all, various approximations and simplifications must be adopted. A common approach is to consider only short stretches of the chain at one time and to predict their local folding into sheets and helices (called the molecule's secondary structure). Even this makes prodigious demands on the computer, and the success of the results is subject to dispute.

In 1973 Georg Schulz of the Max Planck Institute for Medicine in Heidelberg, Germany, solved the full structure of a medium-size protein by experimental methods. Before publishing his results he invited several chemists and biologists to predict the secondary structure by computational means. All of the predictions were wrong, although some were less wrong than others. What is most discouraging is that when the procedure was repeated a year later with another protein, a different set of programs performed best.

Problem 6

Confirm or refute Catalan's Conjecture, namely that 8 and 9 are the only two consecutive perfect powers among the integers.

Mathematics is full of problems where computing machines *might* be useful. In recent years there have been two celebrated cases—the proof of the four-color conjecture and the classification of the finite simple groups—where computers contributed materially to understanding. For the most part, however, the computer is quite useless as a tool for establishing mathematical truth. You cannot prove most theorems about numbers by testing individual cases because there are infinitely many cases to be tested.

There is one conspicuous exception to this rule. The 19th-century mathematician Eugène Charles Catalan pointed out that 8 and 9 are the only known consecutive integers that are perfect integer powers (8 is equal to 2^3 and 9 is equal to 3^2). Catalan speculated that there are no other consecutive perfect powers but was unable to prove it. This conjecture would be like many others in mathematics—where computational experiments are futile because there are infinitely many cases— except for a result proved in 1974 by Robert Tijdeman.[14] I find the result extremely weird. Tijdeman showed that if there is another pair of consecutive powers, they are less than some finite value.

Tijdeman's result can be expressed more formally as follows. A pair of consecutive powers would provide a solution to the equation:

$$x^m - y^n = 1$$

in which x, y, m, and n all have integer values. Tijdeman proved that all four numbers, if they exist, must be less than some constant C. Thus Catalan's Conjecture can be settled simply by checking all possible solutions to the equation using integers less than C.

There is a catch, of course. C, although finite, is huge; indeed, as far as I know its value has not even been calculated. It is a mark of the difference between mathematicians and the rest of us that most of them quite lost interest in the Catalan Conjecture after Tijdeman published his proof. The question is considered settled; even though we don't know the answer, we know how to get it. All that's needed is perhaps a few Cray-millenia of grunt work.

Problem 7

Predict, within 20%, the change in the Dow-Jones industrial average over a period of one week. Consolation prize: merely predict whether the average will rise or fall in the course of a week.

I suspect most readers will classify this an impossible challenge. I don't disagree, yet it's not immediately obvious why the

problem is so hard. There are fewer stocks traded on Wall Street than there are base-pairs in the human genome. And the main economic force at work—the law of supply and demand—does not seem any more complicated than the electrostatic force that guides the folding of a protein molecule. (People are attracted to wealth and repelled by poverty.)

Models of entire economies, such as the input-output matrices of Wassily Leontief, have on the order of 10,000 variables; they are not very complex compared with the other systems considered on this list. One such model, which is run monthly by the Federal Reserve Bank of Minneapolis, makes modest demands on the time of a Cray-1 at the University of Minnesota.[15] If the entire economy can be simulated, why not the stock market, which is a small part of that economy? What do you say we spend a year coding up a fancy simulation, connect the Cray to an on-line ticker service, then sit back and get rich?

Some economists might argue that the basic constraint is lack of input data. The stock market is not a closed system, they would point out, and you cannot predict tomorrow's prices from today's trading in the same way you can (or might) predict tomorrow's weather from the state of today's atmosphere. Personally, I think there is a more fundamental problem. The most important elements of the simulation would not be stock shares and dollars but the people who trade them. Even when the forces acting on people have a simple character, their responses to those forces can be baffling.

There is also a cruel twist to the entire undertaking. Suppose you wrote a perfect simulation that consistently predicted tomorrow's closing prices to the penny. You would shortly grow to be as rich as Jay Gould. Furthermore, like Jay Gould, you would become a significant force in the market, able to alter prices by your own trading alone. Could your program take into account the effects of its own decisions? And if *you* can write such a program, so can many other readers of *COMPUTER LANGUAGE*. At that point second guessing the market is no longer enough; you must *n*th guess it.

Problem 8

Prove the correctness of a large software system such as the set of programs proposed for the Strategic Defense Initiative.

Writing the 10 million lines of software for SDI—the Star Wars weapon system—might in itself be considered a candidate for inclusion on this list.[16,17] It is not included simply because it is a human problem rather than a computer problem and will remain so no matter how much design automation and AI magic are brought to bear on it.

Proving the correctness of all that software is an inhuman problem. That it needs to be proved is not in much dispute. Nothing that big has ever worked the first time before (and there cannot be a second time); since the whole point of building the system is to create a "credible counterthreat," some evidence that it just might work would seem to be required.

I put this problem at the end of my list, implying that I consider it the hardest of them all. I cannot substantiate that judgment of computational complexity, although I think it is fair to say that doing anything at all with 10 million lines of code—reading it, printing it, compiling it, much less proving theorems about its logical structure—is a formidable undertaking. The problem earns last place, however, simply because I believe the task impossible.

Suppose we ran the 10 million lines of source code through some hypothetical and fully automatic program-proving program, and after churning away for some unpredictable number of Cray-years the verifier issued its answer: Q.E.D. Could we believe it? Who verifies the verifier? Moreover, the proof would not consist of a single line of output. Instead it would be another 10 million lines of logic, fully as opaque to human readers as the original program. Do we then devise another program to check the proof and another to check the output of the proofchecker? Where, if anywhere, does this regress stop?

The largest software proof I know of was constructed by a group from Honeywell, the University of Texas at Austin, and the University of Minnesota.[18] Their aim was to verify the security of a program rather than its correctness, but the problems and techniques are very similar. The system being verified was the kernel of an operating system, far smaller and less complex than the SDI software, but still much bigger than the toy programs used in most studies of program verification.

The group completed the proof to their own satisfaction. They also wrote: "The proofs of complex systems tend to be extremely long and tediously detailed. Indeed, a long computer-generated proof may not be read even by the person supervising the proof. Consequently claims of correctness are often underlain by confidence in the verification system rather than by confidence in the proof the verification system has generated. Yet, in the end, a proof that cannot be understood proves nothing."

Q.E.D. ■

References

1. Beeler, M., R.W. Gosper, and R. Schroeppel. "HAKMEM." Artificial Intelligence Memo No. 239, Massachusetts Institute of Technology, Cambridge, Mass., 1972.
2. Turchin, Valentin F. "The Concept of a Supercompiler." *ACM Transactions on Programming Languages and Systems* (July 1986): 292-325.
3. Wade, Nicholas. "The Complete Index to Man." *Science* (Jan. 2, 1981): 33-35.
4. Goad, Walter B. "GenBank." *Los Alamos Science* (Fall 1983): 53-63.
5. Friedland, Peter, and Laurence H. Kedes. "Discovering the Secrets of DNA." *Communications of the ACM* (Nov. 1985): 1164-1186.
6. Richardson, Lewis F. *Weather Prediction by Numerical Process.* Dover Publications, 1965.
7. Williamson, David L., and Paul N. Swartztrauber. "A Numerical Weather Prediction Model—Computational Aspects on the Cray-1." *Proceedings of the IEEE* (Jan. 1984):56-67.
8. Kerr, Richard A. "The Race to Predict Next Week's Weather," *Science* (Apr. 1, 1983): 39-41.
9. Rivest, R.L., A. Shamir, and L. Adelman. "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems." *Communications of the ACM* (Feb. 1978): 120-126.
10. Hellman, Martin E. "The Mathematics of Public-Key Cryptography." *Scientific American* (Aug. 1979): 146-157.
11. La Brecque, Mort. "Protein Folding." *Mosaic* (May-June 1983): 2-9.
12. Seibel, George. "Computational Chemistry of Biomacromolecules." *Cray Channels* (Spring 1985): 2-7.
13. Sternberg, Michael J.E., and Janet M. Thornton. "Prediction of Protein Structure from Amino Acid Sequence." *Nature* (Jan. 5, 1978): 15-20.
14. Wei, Julie. "Pure Mathematics: Problems and Prospects in Number Theory." *University of Michigan Research News* (Mar. 1979).
15. "Econometric Supercomputing." *Cray Channels* (Spring 1986): 20-23.
16. Parnas, David Lorge. "Software Aspects of Strategic Defense Systems." *Communications of the ACM* (Dec. 1985): 1326-1335.
17. Lin, Herbert. "The Development of Software for Ballistic-Missile Defense." *Scientific American* (Dec. 1985): 46-53.
18. Young, W.D., W. E. Boebert, and R. Y. Kain. "Proving a Computer System Secure." *Scientific Honeyweller* (July 1985): 18-27.