programming pearls

by Jon Bentley

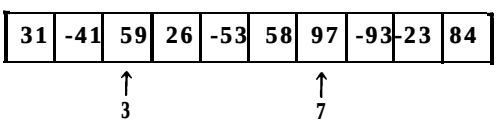
ALGORITHM DESIGN TECHNIQUES

The September 1983 column described the "everyday" impact that algorithm design can have on programmers: an algorithmic view of a problem gives insights that may make a program simpler to understand and to write. In this column we'll study a contribution of the field that is less frequent but more impressive: sophisticated algorithmic methods sometimes lead to dramatic performance improvements.

This column is built around one small problem, with an emphasis on the algorithms that solve it and the techniques used to design them. Some of the algorithms are a little complicated, but the complication is justified; while the first algorithm we'll study takes 39 days to solve a problem of size 10,000, the final algorithm solves the same problem in just a third of a second.

The Problem and a Simple Program

The problem arose in one-dimensional pattern recognition; I'll describe its history later. The input is a vector X of N real numbers; the output is the maximum sum found in any contiguous subvector of the input. For instance, if the input vector is



then the program returns the sum of X[3..7], or 187. The problem is easy when all the numbers are positive-the maximum subvector is the entire input vector. The rub comes when some of the numbers are negative. Should we include a negative number in hopes that the positive numbers to its sides will compensate for its negative contribution? To complete the definition of the problem, we'll say that when all inputs are negative the maximum sum subvector is the empty vector, which has sum zero.

The obvious program for this task is simple: for each pair of integers L and U (where $1 \le L \le U \le N$), compute the sum of $X[L..U]$ and check whether that sum is greater than the maximum sum so far. The pseudocode given in Algorithm 1 is short, straightforward, and easy to understand. Unfortunately, it has the severe disadvantage of being slow. On the computer I typically use, for instance, the code takes about an hour if N is 1,000 and 39 days if N is 10,000 (we'll get to timing details later).

Those times are anecdotal; we get a different kind of feeling for the algorithm's efficiency using "big-oh" notation? The statements in the outermost loop are executed exactly N times, and those in the middle loop are executed at most N times in each execution of the outer loop. Multiplying those two factors of N shows that the four lines contained in the middle loop are executed $O(N^2)$ times. The loop in those four lines is never executed more than N times, so its cost is $O(N)$. Multiplying the cost per inner loop times its number of executions shows that the cost of the entire program is proportional to N cubed, so we'll refer to this as a cubic algorithm.

Those simple steps illustrate the technique of "big-oh" analysis of run time and many of its strengths and weaknesses. Its primary weakness is that we still don't really know the amount of time the program will take for any particular input; we just know that the number of steps it executes is $O(N^3)$. Two strong points of the method often compensate for that weakness. "Big-oh" analyses are usually easy to perform (as above), and the asymptotic run time is often sufficient for a "back-of-the-envelope" calculation to decide whether or not a program is efficient enough for a given application.

The next several sections use asymptotic run time as the only measure of program efficiency. If that makes

```
MaxSoFar := 0.0
for L := 1 to N do
    for u := L to N do
        Sum := 0.0
        for I := L to U do
            Sum := Sum + X[I]
        /* Sum now contains the
             sum of X[L..U] */
        MaxSoFar := max(MaxSoFar, Sum)
```

Algorithm 1. The cubic algorithm

[1] The notation $O(N^2)$ can be thought of as "proportional to N^2"; both $15N^2 + 100N$ and $N^2/2 - 10$ are $O(N^2)$. Informally, $f(N) = O(g(N))$ means that f(N) $< cg(N)$ for some constant c and sufficiently large values of N. A formal definition of the notation can be found in most textbooks on algorithm design or discrete mathematics.

you uncomfortable, peek ahead to the section on "What Does It Matter?", which shows that for this problem such analyses are extremely informative. Before you read on, take a minute to try to find a faster algorithm for this problem.

Two Quadratic Algorithms

Most programmers have the same response to Algorithm 1: "There's an obvious way to make it a lot faster." There are two obvious ways, however, and if one is obvious to a given programmer then the other often isn't. Both algorithms are quadratic-they take $O(N^2)$ steps on an input of size N-and both achieve their run time by computing the sum of $X[L..U]$ in a constant number of steps rather than in the $U - L + 1$ steps of Algorithm 1. But the two quadratic algorithms use very different methods to compute the sum in constant time.

The first quadratic algorithm computes the sum quickly by noticing that the sum of $X[L..U]$ has an intimate relationship to the sum previously computed, that of $X[L..U-1]$. Exploiting that relationship leads to Algorithm 2. The statements inside the first loop are executed N times, and those inside the second loop are executed at most N times on each execution of the outer loop, so the total run time is $O(N^2)$.

An alternative quadratic algorithm computes the sum in the inner loop by accessing a data structure (called *CumArray*) built before the outer loop is ever executed. The I^{th} element of *CumArray* contains the cumulative sum of the values in $X[1..I]$, so the sum of the values in $X[L..U]$ can be found by computing $CumArray[U] - CumArray[L-1]$. This results in Algorithm 2b, which takes $O(N^2)$ time; the analysis is the same as for Algorithm 2.

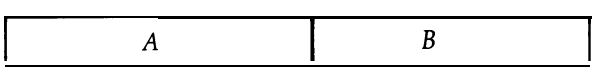
The algorithms we have seen so far inspect all possible pairs of starting and ending values of subvectors and consider the sum of the numbers in that subvector. Because there are $O(N^2)$ subvectors, any algorithm that inspects all such values must take at least quadratic time. Can you think of a way to sidestep this problem and achieve an algorithm that runs in less time?

A Divide-and-Conquer Algorithm

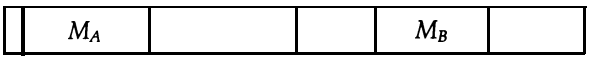
Our first subquadratic algorithm is complicated; if you get bogged down in its details, you won't lose much by skipping to the next section. It is based on the following divide-and-conquer schema:

> *To solve a problem of size N, recursively solve two subproblems of size approximately N/2, and combine their solutions to yield a solution to the complete problem.*

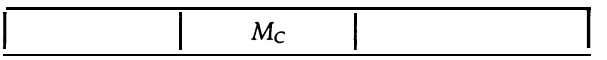
In this case the original problem deals with a vector of size N, so the most natural way to divide it into subproblems is to create two subvectors of approximately equal size, which we'll call A and B:



We then recursively find the maximum subvectors in A and B, which we'll call M_A and M_B:



It is tempting to think that we have solved the problem because the maximum sum subvector of the entire vector must be either M_A or M_B, and that is almost right. In fact, the maximum is either entirely in A, entirely in B, or it crosses the border between A and B (which we'll call M_C for the maximum *crossing* the border):



Thus our divide-and-conquer algorithm will compute M_A and M_B recursively, compute M_C by some other means, and then return the maximum of the three.

That description is almost enough to write code. All we have left to describe is how we'll handle small vectors and how we'll compute M_C. The former is easy: the maximum of a one-element vector is the only value in the vector (or zero if that number is negative), and the maximum of a zero-element vector was previously defined to be zero. To compute M_C *we* observe that its component in A is the largest subvector starting at the boundary and reaching into A, and similarly for its component in B. Putting these facts together leads to Algorithm 3, which is originally invoked by the statement

```
Answer := MaxSum(1, N)
```

Although the code is complicated (and easy to get wrong), it does yield a substantial decrease in run time:

```
MaxSoFar := 0.0
for L := 1 to N do
    Sum := 0.0
    for u := L to N do
        Sum := Sum + XII]
        /* Sum now contains the
             sum of X[L..U] */
        MaxSoFar := max(MaxSoFar, Sum)
```

Algorithm 2. The first quadratic algorithm

```
CumArray[O] := 0.0
for I := 1 to N do
    CumArray[I] := CumArray[I - I] + XII]
MaxSoFar := 0.0
for L := 1 to N do
    for U := L to N do
        Sum := CumArray[U] - CumArray[L - I]
        /* Sum now contains the
             sum of X[L..U] */
        MaxSoFar := max(MaxSoFar, Sum)
```

Algorithm 2b. An alternative quadratic algorithm

```
recursive function MaxSum(L, U)
    if L > U then    /* Zero-element vector */
         return 0.0
    if L = U then    /* One-element vector */
         return max(0.0, X[L])

    M := (L + U)/2    /* A is X[L..M], B is X[M + 1..U] */
    /* Find max crossing to left */
        Sum := 0.0; MaxToLeft := 0.0
        for I := M downto L do
            Sum := Sum + X[I]
            MaxToLeft := max(MaxToLeft, Sum)
    /* Find max crossing to right */
        Sum := 0.0; MaxToRight := 0.0
        for I := M + 1 to U do
            Sum := Sum + XII]
            MaxToRight := max(MaxToRight, Sum)
    MaxCrossing := MaxToLeft + MaxToRight

    MaxInA := MaxSum(L, M)
    MaxInB := MaxSum(M + 1, U)
    return max(MaxCrossing, MaxInA, MaxInB)
```

Algorithm 3. A divide-and-conquer algorithm

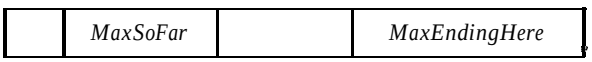
it solves the problem in O(N log N) time. There are a number of ways of proving this fact. An informal argument observes that the algorithm does O(N) work on each of O(log N) levels of recursion. The argument can be made more precise by the use of recurrence relations; if T(N) denotes the time to solve a problem of size N, then we can show that $T(1) = O(1)$ and that

$$T(N) = 2T(N/2) + O(N).$$

Most textbooks on algorithm design show that this recurrence has the solution $T(N) = O(N \log N)$.

A Scanning Algorithm

We'll now use the simplest kind of algorithm that operates on arrays: it starts at the left end (element $X[1]$) and scans through to the right end (element X[N]), keeping track of the maximum sum subvector seen so far. The maximum is initially zero. Suppose we've solved the problem for $X[1..I-1]$; how can we extend that to a solution for the first I elements? We use reasoning similar to that of the divide-and-conquer algorithm: the maximum sum in the first I elements is either the maximum sum in the first $I-1$ elements (which we'll call *MaxSoFar),* or it is that of a subvector that ends in position I (which we'll call *MaxEndingHere):*



Recomputing *MaxEndingHere* from scratch (using code like that in Algorithm 3) yields yet another quadratic algorithm. We can get around this by using the technique that led to Algorithm 2: instead of computing the maximum subvector ending in position I from scratch, we'll use the maximum subvector that ends in position $I-1$. This results in Algorithm 4.

The key to understanding this program is the variable *MaxEndingHere.* Before the first assignment statement in the loop, *MaxEndingHere* contains the value of the maximum subvector ending in position I – 1; the assignment statement modifies it to contain the value of the maximum subvector ending in position I. The statement increases it by the value $X[I]$ so long as doing so keeps it positive; when it goes negative, it is reset to zero (that is, the maximum subvector ending at I is the empty vector). Although the code is subtle, it is short and fast: its running time is O(N) (so we'll refer to it as a linear algorithm). David Gries systematically derives and verifies this algorithm in his paper "A Note on the Standard Strategy for Developing Loop Invariants and Loops" (in *Science of Computer Programming 2,* pp. 207–214).

What Does It Matter?

So far I've played fast and loose with "big-ohs"; it's time for me to come clean and tell about the run times of the programs. I implemented the four primary algorithms (all except Algorithm 2b) in the C language on a Digital

```
MaxSoFar := 0.0
MaxEndingHere := 0.0
for I := 1 to N do
    MaxEndingHere := max(0.0,
        MaxEndingHere + X[I])
    MaxSoFar := max(MaxSoFar,
        MaxEndingHere)
```

Algorithm 4. The linear algorithm

TABLE I. Summary of the Algorithms

Algorithm		1	2	3	4
Lines of C Code		8	7	14	7
Run time in microseconds		$3.4N^3$	$13N^2$	46N log N	33N
Time to solve problem of size	10^2	3.4 secs	130 msecs	30 msecs	3.3 msecs
	10^3	.94 hrs	13 secs	.45 secs	33 msecs
	10^4	39 days	22 mins	6.1 secs	.33 secs
	10^5	108 yrs	1.5 days	1.3 min	3.3 secs
	10^6	108 mill	5 mos	15 min	33 secs
Max problem solved in one	sec	67	280	2000	30,000
	min	260	2200	82,000	2,000,000
	hr	1000	17,000	3,500,000	120,000,000
	day	3000	81,000	73,000,000	2,800,000,000
If N multiplies by 10, time multiplies by		1000	100	10+	10
If time multiplies by 10, N multiplies by		2.15	3.16	10–	10

Equipment Corporation VAX-11/750,[2] timed them, and extrapolated the run times to achieve Table I.

This table makes a number of points. The most important is that proper algorithm design can make a big difference in run time; that point is underscored by the middle rows. The table also shows something of the different character of cubic, quadratic, $N \log N$ and linear algorithms: the last two rows show how the problem size and run time vary as a function of each other.

Another important point is that when we're comparing cubic, quadratic, and linear algorithms with one another, the constant factors of the programs don't matter much. To underscore this point, I conducted an experiment in which I tried to make the constant factors of two algorithms differ by as much as possible. To achieve a huge constant factor I implemented Algorithm 4 on a BASIC interpreter on a Radio Shack TRS-80 Model III microcomputer. For the other end of the spectrum, Eric Grosse of AT&T Bell Laboratories and I implemented Algorithm 1 in fine-tuned FORTRAN on a CRAY-1 supercomputer. We got the disparity we wanted: the run time of the cubic algorithm was measured as $3.0N^3$ nanoseconds, while the run time of the linear algorithm was $19{,}500{,}000N$ nanoseconds. Table II shows how those expressions translate to times for various problem sizes (the same data is displayed graphically in Figure 1.)

The difference in constants (a factor of six and a half million) allowed the cubic algorithm to start off faster, but the linear algorithm was bound to catch up. In this case, the break-even point for the two algorithms is around 2,500, where each takes about 50 seconds.

Principles

The history of the problem sheds light on the algorithm design techniques. The problem arose in a pattern-matching procedure designed by Ulf Grenander of Brown University in the two-dimensional form described in Problem 7. In that form, the maximum sum subarray was the maximum likelihood estimator of a certain kind of pattern in a digitized picture. Because the two-dimensional problem required too much time to solve, Grenander simplified it to one dimension to gain insight into its structure.

Grenander observed that the cubic time of Algorithm 1 was prohibitively slow, and derived Algorithm 2. In 1977 he described the problem to Michael Shamos of UNILOGIC, Ltd. (then of Carnegie-Mellon University)

TABLE II. The Tyranny of Asymptotics

N	Cray-1, FORTRAN, Cubic Algorithm	TRS-80, BASIC, Linear Algorithm
10	3.0 microsecs	200 millisecs
100	3.0 millisecs	2.0 secs
1000	3.0 secs	20 secs
10,000	49 mins	3.2 mins
100,000	35 days	32 mins
1,000,000	95 yrs	5.4 hrs

[2] VAX is a trademark of Digital Equipment Corporation.

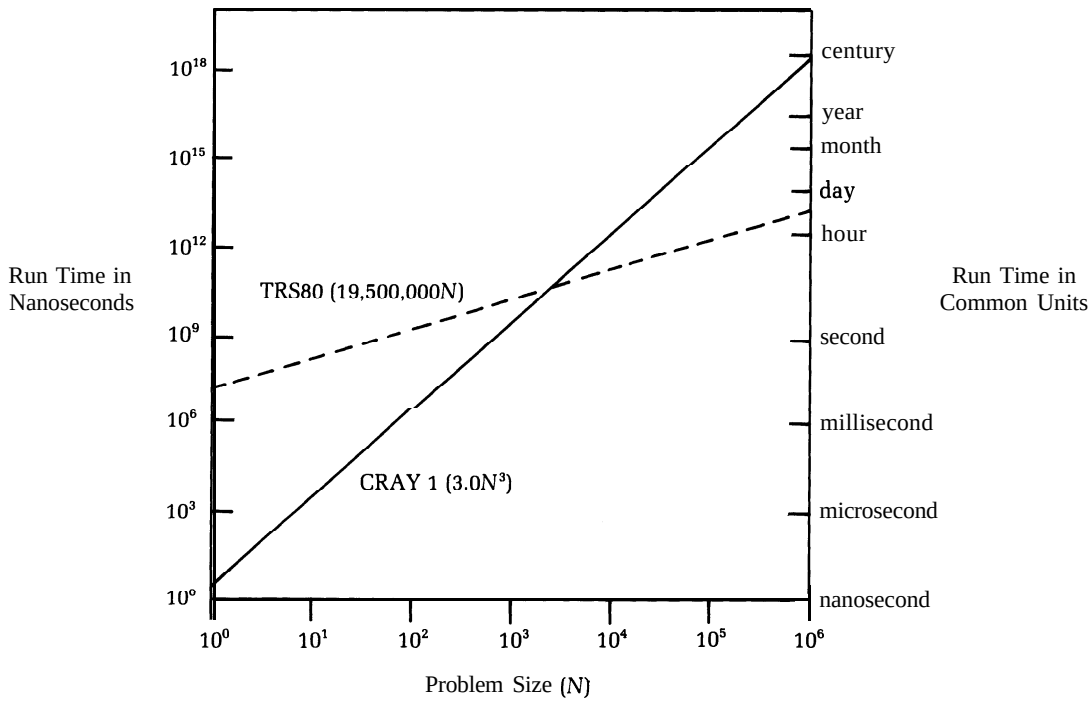


FIGURE 1. The Run Times of Two Programs

who overnight designed Algorithm 3. When Shamos showed me the problem shortly thereafter, we thought that it was probably the best possible; researchers had just shown that several similar problems require time proportional to N log N. A few days later Shamos described the problem and its history at a seminar attended by Jay Kadane (a statistician at Carnegie-Mellon University), who designed the linear-time Algorithm 4 within a minute. Fortunately, we know that there can be no better algorithm: any algorithm must look at all N inputs,

Even though the one-dimensional problem is now completely solved, Grenander's original two-dimensional problem remains open. Because of the computational expense of all known algorithms, Grenander had to abandon that approach to the pattern-matching problem. Readers who feel that the linear-time algorithm for the one-dimensional problem is "obvious" are therefore urged to find an "obvious" algorithm for Problem 7!

Although the algorithms in this story were never incorporated into a system, they do illustrate several important algorithm design techniques that have had substantial impact on many systems (see the sidebar on page 870):

Save state to avoid recomputation. This simple form of dynamic programming arose in Algorithms 2 and 4. By using space to store results, we avoid using time to recompute them.

Preprocess information into data structures. This was the technique underlying Algorithm 2b: the *CumArray* structure allowed the sum of a subvector to be accessed in just a couple of operations.

Divide-and-conquer algorithms. Algorithm 3 uses a simple form of divide-and-conquer; textbooks on algorithm design describe more advanced forms.

Scanning algorithms. Problems on arrays can often be solved by asking "how can I extend a solution for $X[1..I-1]$ to a solution for $X[1..I]$?" In Algorithm 4 we had to remember both the old answer and some auxiliary data to compute the new answer.

Cumulatives. Algorithm 2b uses a cumulative table in which the I^{th} element contains the sum of the first I values of X; such tables are common when dealing with ranges. In business data processing applications, for instance, one finds the sales from March to October by subtracting the February year-to-date sales from the October year-to-date sales.

Lower bounds. Algorithm designers sleep peacefully only when their algorithm is known to be the best possible because they have proved a matching lower bound. The linear lower bound for this problem was easy; more complex lower bounds can be difficult.

Problems

1. Algorithms 3 and 4 use subtle code that is easy to get wrong. Use the program verification techniques described in the December 1983 column to argue the correctness of the code; specify the loop invariants carefully.
2. Our analysis of the four algorithms was done only at the "big-oh" level of detail. Analyze the number of additions and comparisons done by each algorithm as exactly as possible; does this exercise give any insight into the behavior of the algorithms?

The Impact of Algorithms

Although the problem studied in this column illustrates several important techniques, it is really a "toy"-it was never incorporated into a system. We'll now survey several very real problems in which algorithm design techniques proved their worth.

Numerical Analysis. The standard example of the power of algorithm design is the Fast Fourier Transform. Its divide-and-conquer structure reduced the time required for Fourier analysis from $O(N^2)$ to O(N log N). Because problems in signal processing and time series analysis frequently process inputs of size N = 1,000 or greater, the algorithm speeds up programs by factors of more than 100. The Fast Fourier Transform has opened whole new areas of engineering.

Graph Algorithms. In a common approach to building integrated circuitry, the designer describes an electrical circuit as a graph that is later transformed into a chip design. A popular approach to laying out the circuit uses the "graph partitioning" problem to divide the entire electrical circuit into subcomponents. Heuristic algorithms for graph partitioning developed in the early 1970s used $O(N^2)$ time to partition a circuit with a total of N components and wires, Fiduccia and Mattheyses describe "A linear-time heuristic for improving network partition" in the *29th Design Automation Conference.* Because typical problems involve a few thousand components, their method reduces layout time from a few hours to a few minutes.

Geometric Algorithms. Late in their design, integrated circuits are specified as geometric "artwork" that is eventually etched onto chips. Design systems process the artwork to perform tasks such as extracting the electrical circuit it describes, which is then compared to the circuit the designer specified. In the days when integrated circuits had $N = 1{,}000$ geometric figures that specified `100` transistors, algorithms that compared all pairs of geometric figures in $O(N^2)$ time could perform the task in a few minutes. Now that VLSI chips contain millions of geometric components, quadratic algorithms would take months of CPU time. "Plane sweep" (or "scan line") algorithms have reduced the running time to O(N log N), so the designs can now be processed in a small number of hours. The paper "Space efficient algorithms for VLSI artwork analysis" by Szymanski and Van Wyk in the *20th Design Automation* Conference describes efficient algorithms for such tasks that use only small amounts of primary memory.

Other Problem Domains. These stories just scratch the surface of the application of algorithm design techniques in real systems; for more examples, see the further reading. If readers know of additional case studies that have not been published, I would appreciate learning about them.

3. We defined the maximum subvector of an array of negative numbers to be zero (the sum of the empty subvector). Suppose that we had instead defined it to be the value of the element closest to zero; how would you change the programs?
4. Suppose that we wished to find the subvector with sum closest to zero rather than that with maximum sum. What is the most efficient algorithm you can design for this task? What design techniques are applicable? What if we wished to find the subvector with sum closest to a given real number T?
5. A turnpike consists of N − 1 stretches of road between N toll stations; each stretch of road has an associated cost of travel. It is trivial to tell the cost of going between any two stations in O(N) time using only an array of the costs or in constant time using a table of $O(N^2)$ elements. Is it possible to build a data structure that requires O(N) space but allows the cost of any route to be computed in constant time?
6. After the array $X[1..N]$ is initialized to zero, N of the following operations are performed

```
for I := L to U do
    X[I] := X[I] + v
```

where L, U, and V are parameters of each operation (L and U are integers satisfying $1 \le L \le U \le N$ and V is a real). After the N operations, the values of $X[1]$ through X[N] are reported in order. This problem arose in gathering statistics in a simulation program; the method just sketched requires $O(N^2)$ time. Can you find a faster algorithm?

7 *[Research Problem.]* In the maximum subarray problem we are given an N x N array of reals, and we must find the maximum sum contained in any rectangular subarray. What is the complexity of this problem?

Solution to July's Problem

This is the straightforward program for finding the maximum value in the array X of N values:

```
Max:=X[1]; I:=1
while I<N do
      /* Invariant: Max holds
           for X[1..I] */
      I := I + 1
      if X[I]>Max then
           Max:=X[I]
```

It makes a total of 2N comparisons: N in the while loop

compare I to N, and N in the if statement compare $X[I]$ to *Max*.

In his unpublished note "Finding the Maximum", R.G. Dromey of the University of Wollongong (P.O. Box 1144, Wollongong N.S.W. 2500, Australia) shows how to reduce the number of comparisons by a factor of almost two to $N + \log_e N$. His code uses $X[N+1]$ as a *sentinel* element that is always equal to the variable *Max*.

```
I:=1
while I<=N do
    Max:=X[I]; X[N+1] := Max; I:=I+1
    while X[I]<Max do I:=I+1
```

Because of the sentinel, the program compares I to N only when it finds a new maximum; in Section 12.10 of his *Art of Computer Programming,* Knuth shows that roughly $\log_e N$ maxima are found, on the average.

In his note, Dromey derives the program by asking how one could efficiently confirm that Max indeed contains the maximum, under the "best-case" assumption that it is initialized that way. He concludes, "The design lesson from this problem is that investigating a 'best-situation' can provide insight into a more efficient solution. The second implementation, in the terminology of Jackson's *Principles of Program Design,* arrived at a closer match between the structure of the data and the dynamic behaviour of the algorithm. Such matching frequently leads to more efficient implementations."

Further Reading

Only extensive study can put these algorithm design techniques at your fingertips; most programmers will get this only from the serious study of a textbook on algorithms. *Data Structures and Algorithms* by Aho, Hopcroft, and Ulfman is an excellent undergraduate text; Chapter 10 on "Algorithm Design Techniques" is especially relevant to this column. If you'd like more of an introduction before jumping into a textbook, you may enjoy my survey article "An Introduction to Algorithm Design" in IEEE *Computer* Magazine, Volume 12, Number 2, February 1979.

For Correspondence: Jon Bentley, AT&T Bell Laboratories, Room 2C-317, 600 Mountain Avenue, Murray Hill, NJ 07974.