PART II: PERFORMANCE

A simple, powerful program that delights its users and does not vex its builders — that is the programmer's ultimate goal and the emphasis of the four previous columns.

We'll turn our attention now to one specific aspect of delightful programs: efficiency. Inefficient programs sadden their users with late output and big bills. These columns therefore describe several paths to performance.

The next column surveys the approaches and how they interact. The three subsequent columns discuss three methods for improving run time, in the order in which they are usually applied:

Column 6 shows how "back-of-the-envelope" calculations used early in the design process can ensure that the basic system structure is efficient enough.

Column 7 is about algorithm design techniques that sometimes dramatically reduce the run time of a module.

Column 8 discusses code tuning, which is usually done late in the implementation of a system.

To wrap up Part II, Column 9 turns to another aspect of performance: space efficiency.

There are two good reasons for studying efficiency. The first is its intrinsic importance in many applications. A software manager I know estimates that half his development budget goes to efficiency; a manager of a data processing installation has to purchase million-dollar mainframes to solve his performance problems. Many systems demand execution speed, including real-time programs, huge databases and machines dedicated to a single program.

The second reason for studying performance is educational. Apart from practical benefits, efficiency is a fine training ground. These columns cover ideas ranging from the theory of algorithms to common sense like "back-of-the-envelope" calculations. The major theme is fluidity of thinking; Column 5, especially, encourages us to look at a problem from many different viewpoints.

Similar lessons come from many other topics. These columns might have been built around user interfaces, system robustness, security, or accuracy of answers. Efficiency has the advantage that it can be measured: we can all agree that one program is 2.5 times faster than another, while discussions on user interfaces, for instance, often get bogged down in personal tastes.

Column 5 appeared in the November 1984 *Communications of the ACM*, Column 6 in March, Column 7 in September, Column 8 in February, and Column 9 in May.

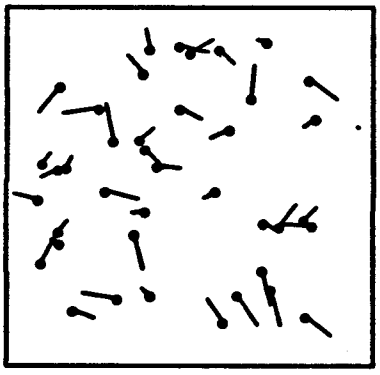
COLUMN 5: PERSPECTIVE ON PERFORMANCE

The next three columns describe three different approaches to run-time efficiency. In this column we'll see how those parts fit together into a whole: each technique is applicable to one of several *design levels* at which computer systems are built. We'll first study one particular program, and then turn to a more systematic view of design levels.

5.1 A Case Study

Andrew Appel describes "An efficient program for many-body simulations" in the January 1985 *SIAM Journal on Scientific and Statistical Computing 6*, 1, pp. 85-103. By working on the program at several levels, he reduced its run time from a year to a day.

The program solves the classical "*N*-body problem" of computing interactions in a gravitational field. It simulates the motions of *N* objects in 3-space, given their masses and initial positions and velocities; think of the objects as planets, stars or galaxies. In two dimensions, the input might look like



Appel's paper describes two astrophysical problems in which $N=10,000$; by studying simulation runs, physicists can test how well a theory matches astronomical observations.

The obvious simulation program divides time into small "steps" and computes the progress of each object at each step. Because it computes the

attraction of each object to every other, the cost per time step is proportional to N^2. Appel estimated that 1,000 time steps of such an algorithm with $N=10{,}000$ would require roughly one year on a VAX-11/780 or one day on a Cray-1.

The final program solves the problem in less than a day on a VAX-11/780 (for a speedup factor of 400) and has been used by several physicists. The following brief survey of his program will ignore many important details that can be found in his paper; the important message is that a huge speedup was achieved by working at several different levels.

Algorithms and Data Structures. Appel's first priority was to reduce the $O(N^2)$ cost per time step to $O(N \log N)$.† He therefore represents the physical objects as leaves in a binary tree; higher nodes represent clusters of objects. The force operating on a particular object can be approximated by the force exerted by the large clusters; Appel showed that this approximation does not bias the simulation. The tree has roughly $\log N$ levels, and the resulting $O(N \log N)$ algorithm is similar in spirit to the algorithm in Section 7.3. This change reduced the run time of the program by a factor of 12.

Algorithm Tuning. The simple algorithm always uses small time steps to handle the rare case that two particles come close to one another. The tree data structure allows such pairs to be recognized and handled by a special procedure. That doubles the time step size and thereby halves the run time of the program.

Data Structure Reorganization. The tree that represents the initial set of objects is quite poor at representing later sets. Reconfiguring the data structure at each time step costs a little time, but reduces the number of local calculations and thereby halves the total run time.

Code Tuning. Due to additional numerical accuracy provided by the tree, 64-bit double-precision floating point numbers could be replaced by 32-bit single-precision numbers; that change halved the run time. Profiling the program showed that 98 percent of the run time was spent in one procedure; rewriting that code in assembly language increased its speed by a factor of 2.5.

Hardware. After all the above changes, the program still required two days of VAX-11/780 run time, and several runs of the program were desired. Appel therefore transported the program to a similar machine equipped with a floating point accelerator, which halved its run time.

The changes described above multiply together for a total speedup factor of

† 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, and Section 7.5 illustrates the relevance of the notation to program design.

400; Appel's final program runs a 10,000-body simulation in about one day. The speedups were not free, though. The simple algorithm may be expressed in a few dozen lines of code, while the fast program required 1200 lines of Pascal. The design and implementation of the fast program required several months of Appel's time. The speedups are summarized in the following table.

DESIGN LEVEL	SPEEDUP FACTOR	MODIFICATION
Algorithms and Data Structures	12	A binary tree reduces $O(N^2)$ time to $O(N \log N)$
Algorithm Tuning	2	Use larger time steps
Data Structure Reorganization	2	Produce clusters well-suited to the tree algorithm
System-Independent Code Tuning	2	Replace double-precision floating point with single precision
System-Dependent Code Tuning	2.5	Recode the critical procedure in assembly language
Hardware	2	Use a floating point accelerator
Total	400	

This table illustrates several kinds of dependence among speedups. The primary speedup is the tree data structure, which opened the door for the next three changes. The last two speedups, changing to assembly code and using the floating point accelerator, were in this case independent of the tree. The tree structure would have less of an impact on a Cray-1 (whose pipelined architecture is well-suited to the simple algorithm), so we see that algorithmic speedups are not necessarily independent of hardware.

5.2 Design Levels

A computer system is designed at many levels, ranging from its high-level software structure down to the transistors in its hardware. The following survey is intended only as an intuitive guide to design levels, so don't expect a formal taxonomy.

Problem Definition. The battle for a fast system can be won or lost in specifying the problem it is to solve. On the day I wrote this paragraph, a vendor told me that he couldn't deliver supplies because a purchase order had been lost somewhere between my organization and my company's purchasing department. Purchasing was swamped with similar orders; fifty people in my organization alone had placed individual orders. A friendly chat between my management and purchasing resulted in consolidating those fifty orders into one large order. In addition to easing administrative work for both organizations, this change sped up one small part of a computer system by a factor of

fifty. A good systems analyst keeps an eye out for such savings, both before and after systems are deployed.

Sometimes good specifications give users a little less than what they thought was needed. In Column 1 we saw how incorporating a few important facts about the input to a sorting program decreased both its run time and its code length by an order of magnitude. Problem specification can have a subtle interaction with efficiency; for example, good error-recovery may make a compiler slightly slower, but it usually decreases its overall time by reducing the number of compilations.

System Structure. The decomposition of a large system into modules is probably the single most important factor in determining its performance. Here are two distinct organizations for a query-answering system.

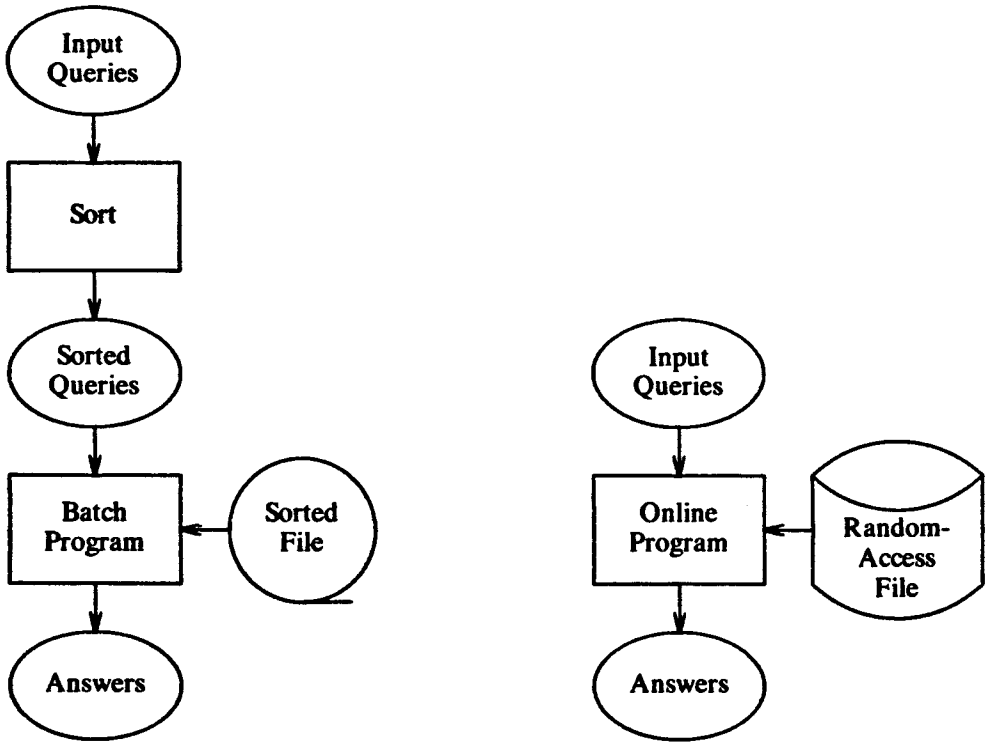


Figure 1. Two system structures.

These structures are two canonical ways of bringing together related information; they also arise in file updating programs and spelling checkers (we'll return to these structures in Column 13). Problem 1 shows that each organization is fast in some contexts and slow in others. After decomposing a system into modules, the designer should do a simple "back-of-the-envelope" estimate to make sure that its performance is in the right ballpark; such calculations are the subject of Column 6. Because efficiency is much easier to build into a new system than to retrofit into an existing system, performance analysis is crucial during system design.

Algorithms and Data Structures. The keys to a fast module are usually the structures that represent its data and the algorithms that operate on the data. The largest single improvement in Appel's program came from replacing an $O(N^2)$ algorithm with an $O(N \log N)$ algorithm; Columns 2 and 7 describe similar speedups.

Code Tuning. Appel achieved a factor of five by making small changes to code; Column 8 is devoted to that topic.

System Software. Sometimes it's easier to change the software on which a system is built than the system itself. Section 6.4, for instance, describes how replacing an interpreted language with a compiled language could increase performance by a factor of several hundred. Other available speedups include compiler optimizations (including a language's run-time system) and operating system and database system modifications.

Hardware. There are many ways that faster hardware can increase performance. General-purpose computers are sometimes fast enough; speedups are available through faster implementations of the same architecture, multiprocessors, and supercomputers. Sometimes it is more cost-effective to build a special-purpose box that handles just the particular problem at hand; special-purpose chips for speech synthesis, for example, enable inexpensive toys and household appliances to talk. Appel's solution of adding a floating point accelerator to the existing machine was somewhere between the two extremes.

5.3 Principles

Because an ounce of prevention is worth a pound of cure, we should keep in mind an observation due to Gordon Bell of Encore Computer Corporation.

> *The cheapest, fastest and most reliable components of a computer system are those that aren't there.*

Those missing components are also the most accurate (they never make mistakes), the most secure (they can't be broken into), and the easiest to design, document, test and maintain. The importance of a simple design can't be overemphasized.

But when performance problems can't be sidestepped, thinking about design levels can help focus a programmer's effort.

If you need a little speedup, work at the best level. Most programmers have their own knee-jerk response to efficiency: "change algorithms" or "tune the queueing discipline" spring quickly to some lips. Before you decide to work at any given level, consider all possible levels and choose the one that delivers the most speedup for the least effort.

If you need a big speedup, work at many levels. Enormous speedups like Appel's are achieved only by attacking a problem on several different fronts, and they usually take a great deal of effort. When changes on one level are independent of changes on other levels (as they often, but not always, are), the various speedups multiply.

Columns 6, 7 and 8 discuss speedups at three different design levels; keep perspective as you consider the individual speedups.

5.4 Problems

1. Figure 1 in Section 5.2 shows two possible organizations for a simple query-answering system. Assume that each of the one million records in the file is identified by a key and that each query refers (by key) to a single record. Further assume that both files are stored on disk in 100-record blocks, and that a random block can be read from disk in 50 milliseconds while blocks can be read sequentially every 5 milliseconds. The batch program reads the entire file, while the online algorithm reads only relevant blocks but may read some blocks many times. When is the batch method more efficient than the online program?
2. Discuss speedups at various design levels for some of the following problems: simulating Conway's "Game of Life", factoring 100-digit integers, Fourier analysis, simulating VLSI chips, and searching a large text file on disk for a given string. Discuss the dependencies of the proposed speedups.
3. Appel found that changing from double-precision arithmetic to single-precision arithmetic doubled the speed of his program. Choose an appropriate test and measure that speedup on your system.
4. This column concentrates on run-time efficiency. Other common measures of performance include fault-tolerance, reliability, security, cost, cost/performance ratio, accuracy, and robustness to user error. Discuss how each of these problems can be attacked at several design levels.
5. Discuss the costs of employing state-of-the-art technologies at the various design levels. Include all relevant measures of cost, including development time (calendar and personnel), maintainability, and dollar cost.
6. An old and popular saying claims that "efficiency is secondary to correctness — a program's speed is immaterial if its answers are wrong". True or false?
7. Discuss different solutions to problems in everyday life, such as injuries suffered in automobile accidents.

5.5 Further Reading

I learned the theme of this column from Raj Reddy and Allen Newell's paper "Multiplicative speedup of systems" (in *Perspectives on Computer Science*, edited by A. K. Jones and published in 1977 by Academic Press). Their work was motivated by problems in Artificial Intelligence; they conjecture that speech and vision systems might be sped up by a factor of a million. Their paper describes speedups at various design levels, and is especially rich in speedups due to hardware and system software.

Butler Lampson's "Hints for Computer System Design" appears in *IEEE Software 1*, 1, January 1984. Many of the hints deal with performance; his paper is particularly strong at integrated hardware/software system design.

COLUMN 6: THE BACK OF THE ENVELOPE

It was in the middle of a fascinating conversation on software engineering that Bob Martin asked me, "How much water flows out of the Mississippi River in a day?" Because I had found his comments up to that point deeply insightful, I politely stifled my true response and said, "Pardon me?" When he asked again I realized that I had no choice but to humor the poor fellow, who had obviously cracked under the pressures of running a large software shop within Bell Labs.

My response went something like this. I figured that near its mouth the river was about a mile wide and maybe twenty feet deep (or about one two-hundred-and-fiftieth of a mile). I guessed that the rate of flow was five miles an hour, or a hundred and twenty miles per day. Multiplying

$$1 \text{ mile} \times 1/250 \text{ mile} \times 120 \text{ miles/day} \approx 1/2 \text{ mile}^3/\text{day}$$

showed that the river discharged about half a cubic mile of water per day, to within an order of magnitude. But so what?

At that point Martin picked up from his desk a proposal for the computer-based mail system that AT&T developed for the 1984 Summer Olympic games, and went through a similar sequence of calculations. Although his numbers were straight from the proposal and therefore more precise, the calculations were just as simple and much more revealing. They showed that, under generous assumptions, the proposed system could work only if there were at least a hundred and twenty seconds in each minute. He had sent the design back to the drawing board the previous day. (The conversation took place in early 1983, and the final system was used during the Olympics without a hitch.)

That was Bob Martin's wonderful (if eccentric) way of introducing the engineering technique of "back-of-the-envelope" calculations. The idea is standard fare in engineering schools and is bread and butter for most practicing engineers. Unfortunately, it is too often neglected in computing.

6.1 Basic Skills

These basic reminders can be quite helpful in making back-of-the-envelope calculations.

Two Answers Are Better Than One. When I asked Peter Weinberger how much water flows out of the Mississippi per day, he responded, "As much as flows in." He then estimated that the Mississippi basin was about 1000 by 1000 miles, and that the annual runoff from rainfall there was about one foot (or one five-thousandth of a mile). That gives

$$1000 \text{ miles} \times 1000 \text{ miles} \times 1/5000 \text{ mile/year} \approx 200 \text{ miles}^3\text{/year}$$

$$200 \text{ miles}^3\text{/year} \;/\; 400 \text{ days/year} \approx 1/2 \text{ mile}^3\text{/day}$$

or a little more than half a cubic mile per day. It's important to double check all calculations, and especially so for quick ones.

As a cheating triple check, an almanac reported that the river's discharge is 640,000 cubic feet per second. Working from that gives

$$640{,}000 \text{ ft}^3\text{/sec} \times 3600 \text{ secs/hr} \approx 2.3\times10^9 \text{ ft}^3\text{/hr}$$

$$2.3\times10^9 \text{ ft}^3\text{/hr} \times 24 \text{ hrs/day} \approx 6\times10^{10} \text{ ft}^3\text{/day}$$

$$6\times10^{10} \text{ ft}^3\text{/day} \;/\; (5000 \text{ ft/mile})^3 \approx 6\times10^{10} \text{ ft}^3\text{/day} \;/\; (125\times10^9 \text{ ft}^3\text{/mile}^3)$$

$$\approx 60/125 \text{ mile}^3\text{/day}$$

$$\approx 1/2 \text{ mile}^3\text{/day}$$

The proximity of the two estimates to one another, and especially to the almanac's answer, is a fine example of sheer dumb luck.

Quick Checks. Polya devotes three pages of his *How To Solve It* to "Test by Dimension", which he describes as a "well-known, quick and efficient means to check geometrical or physical formulas". The first rule is that the dimensions in a sum must be the same, which is in turn the dimension of the sum — you can add feet together to get feet, but you can't add seconds to pounds. The second rule is that the dimension of a product is the product of the dimensions. The examples above obey both rules; multiplying

$$(\text{miles}+\text{miles}) \times \text{miles} \times \text{miles/day} = \text{miles}^3\text{/day}$$

has the right form, apart from any constants.

A simple table can help you keep track of dimensions in complicated expressions like those above. To perform Weinberger's calculation, we first write down the three original factors.

1000 miles	1000 miles	1 mile
		5000 year

Next we simplify the expression by cancelling terms, which shows that the output is 200 miles^3/year.

$$\frac{\cancel{1000}\ \cancel{\text{miles}} \mid \cancel{1000}\ \cancel{\text{miles}} \mid \cancel{1}\ \cancel{\text{mile}} \mid 200\ \text{mile}^3}{\mid \cancel{5000}\ \text{year} \mid}$$

Now we multiply by the identity (well, almost) that there are 400 days per year.

$$\frac{\cancel{1000}\ \cancel{\text{miles}} \mid \cancel{1000}\ \cancel{\text{miles}} \mid \cancel{1}\ \cancel{\text{mile}} \mid 200\ \text{mile}^3 \mid \text{year}}{\mid \cancel{5000}\ \text{year} \mid \mid 400\ \text{days}}$$

Cancellation yields the (by now familiar) answer of half a cubic mile per day.

$$\frac{\cancel{1000}\ \cancel{\text{miles}} \mid \cancel{1000}\ \cancel{\text{miles}} \mid \cancel{1}\ \cancel{\text{mile}} \mid \cancel{200}\ \text{mile}^3 \mid \cancel{\text{year}} \mid 1}{\mid \cancel{5000}\ \cancel{\text{year}} \mid \mid \cancel{400}\ \text{days} \mid 2}$$

These tabular calculations help you keep track of dimensions.

Dimension tests check the form of equations. Check your multiplications and divisions with an old trick from slide rule days: independently compute the leading digit and the exponent. There are several quick checks for addition.

```
  3142        3142        3142
  2718        2718        2718
 +1123       +1123       +1123
 -----       -----       -----
   983        6982        6973
```

The first sum has too few digits and the second errs in the least significant digit. "Casting out nines" reveals the error in the third example: the digits in the summands sum to 8 modulo 9, while those in the answer sum to 7 modulo 9 (in a correct addition, the sums of the digits are equal after "casting out" groups of digits that sum to nine).

Above all, don't forget common sense: be suspicious of any calculations that show that the Mississippi River discharges 100 gallons of water per day.

6.2 Quick Calculations in Computing

Card, Moran and Newell paint an ambitious picture of estimation on pages 9 and 10 of their *Psychology of Human-Computer Interaction* (published by Erlbaum in 1983).

> A system designer, the head of a small team writing the specifications for a desktop calendar-scheduling system, is choosing between having users type a key for each command and having them point to a menu with a lightpen. On his whiteboard, he lists some representative tasks users of his system must perform. In two

> columns, he writes the steps needed by the "key-command" and "menu" options. From a handbook, he culls the times for each step, adding the step times to get total task times. The key-command system takes less time, but only slightly. But, applying the analysis from another section of the handbook, he calculates that the menu system will be faster to learn; in fact, it will be learnable in half the time. He has estimated previously that an effective menu system will require a more expensive processor: 20% more memory, 100% more microcode memory, and a more expensive display. Is the extra expenditure worthwhile? A few more minutes of calculation and he realizes the startling fact that, for the manufacturing quantities now anticipated, training costs for the key-command system will exceed unit manufacturing costs! The increase in hardware costs would be much more than balanced by the decrease in training costs, even before considering the increase in market that can be expected for a more easily learned system. Are there advantages to the key-command system in other areas, which need to be balanced? He proceeds with other analyses, considering the load on the user's memory, the potential for user errors, and the likelihood of fatigue. In the next room, the Pascal compiler hums idly, unused, awaiting his decision.

Their book then goes on to develop a scientific base in psychology that is a necessary precursor to such a handbook.

That scenario shows how a few envelopes' worth of arithmetic might enable a system designer to make a rational choice between two appealing alternatives. That is a fundamentally different use than Martin's calculation for the Olympic mail system: his analysis of a single design uncovered a fatal flaw (similar calculations in Section 2.4 showed the folly of simple anagram algorithms). In both cases, a short sequence of calculations was sufficient to answer the question at hand; additional figuring would have shed little light.

Early in the life of a system, rapid calculations can steer the designer away from dangerous waters into safe passages. And if you don't use them early, they may show in retrospect that a project was doomed to failure. The calculations are often trivial, employing no more than high school mathematics. The hard part is remembering to use them soon enough.

6.3 Safety Factors

The output of any calculation is only as good as its input. With good data, simple calculations can yield accurate answers which are sometimes quite useful. In 1969 Don Knuth wrote a disk sorting package, only to find that it took twice the time predicted by his calculations. Diligent checking uncovered the flaw: due to a software bug, the system's one-year-old disks had run at only half their advertised speed for their entire lives. When the bug was fixed, the

sorting package behaved as predicted and every other disk-bound program also ran faster.

Often, though, sloppy input is enough to get into the right ballpark. If you guess about twenty percent here and fifty percent there and still find that a design is a hundred times above or below specification, additional accuracy isn't needed. But before placing too much confidence in a twenty percent margin of error, consider Vic Vyssotsky's advice from a talk he has given on several occasions.

"Most of you", says Vyssotsky, "probably recall pictures of 'Galloping Gertie', the Tacoma Narrows bridge which tore itself apart in a windstorm in 1940.† Well, suspension bridges had been ripping themselves apart that way for eighty years or so before Galloping Gertie. It's an aerodynamic lift phenomenon, and to do a proper engineering calculation of the forces, which involve drastic nonlinearities, you have to use the mathematics and concepts of Kolmogorov to model the eddy spectrum. Nobody really knew how to do this correctly in detail until the 1950's or thereabouts. So, why hasn't the Brooklyn Bridge torn itself apart, like Galloping Gertie?

"It's because John Roebling had sense enough to know what he *didn't* know. His notes and letters on the design of the Brooklyn Bridge still exist, and they are a fascinating example of a good engineer recognizing the limits of his knowledge. He knew about aerodynamic lift on suspension bridges; he had watched it. And he knew he didn't know enough to model it. So he designed the stiffness of the truss on the Brooklyn Bridge roadway to be *six times* what a normal calculation based on known static and dynamic loads would have called for. And, he specified a network of diagonal stays running down to the roadway, to stiffen the entire bridge structure. Go look at those sometime; they're almost unique.

"When Roebling was asked whether his proposed bridge wouldn't collapse like so many others, he said, 'No, because I designed it six times as strong as it needs to be, to prevent that from happening.'

"Roebling was a good engineer, and he built a good bridge, by employing a huge safety factor to compensate for his ignorance. Do we do that? I submit to you that in calculating performance of our real-time software systems we ought to derate them by a factor of two, or four, or six, to compensate for our ignorance. In making reliability/availability commitments, we ought to stay back from the objectives we *think* we can meet by a factor of ten, to compensate for our ignorance. In estimating size and cost and schedule, we should be conservative by a factor of two or four to compensate for our ignorance. We should design the way John Roebling did, and not the way his contemporaries did — so far as I know, none of the suspension bridges built by Roebling's

† For more information on the event, see Section 2.6.1 of Braun's *Differential Equations and Their Applications*, Second Edition, published in 1978 by Springer-Verlag.

contemporaries in the United States still stands, and a quarter of all the bridges of any type built in the U.S. in the 1870's collapsed within ten years of their construction.

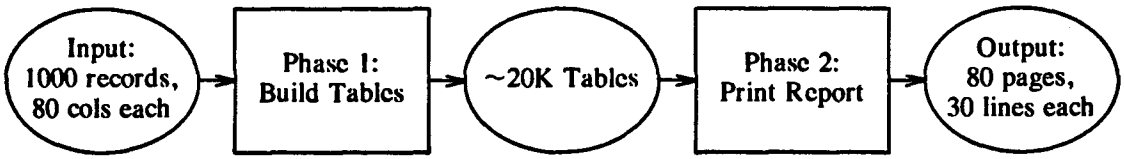
"Are we engineers, like John Roebling? I wonder."

6.4 A Case Study

To make the above points more concrete, I'll describe how I (almost) used them in a system I built for a small company in early 1982. The details can be found in Carnegie-Mellon University Computer Science Technical Report CMU-CS-83-108; I'll just sketch them here.

The system prepared several reports a day to summarize the data on one thousand eighty-column records; the reports were each about eighty pages long. The system's predecessor ran on a large mainframe; my task was to implement a similar system on a personal computer, using interpreted BASIC.

Early in the design of the system I did simple calculations to make sure that the personal computer was up to this application. The space analysis was simple: I calculated the size of the several largest tables and found that they used only half of the 48K bytes of the machine. The time analysis was centered around two main phases.



I didn't worry much about the time for Phase 1: a previous system did that task on an IBM System/360 Model 25 in a minute, and the microprocessor on the personal computer was more powerful than that old workhorse. Instead, I concentrated on Phase 2, which I thought would be limited by the sixty-lines-per-minute speed of the printer. Each page of the report contained about thirty lines, so the total time of forty minutes was well within bounds. After this short analysis, the company purchased three personal computers and I implemented the design.

The first implementation of the program was revealing. Storing the BASIC program required about twenty kilobytes of main memory that I had ignored in my calculation; the safety factor of two saved the day. The forty minutes of printing time was right on the mark. Unfortunately, I was way off in the time to read the records and build the table. Instead of taking a minute, it took *fourteen hours*, which made it awfully hard to prepare a few reports a day. The problem was that I had compared assembly code on the old System/360 with interpreted BASIC on the personal computer, ignoring the fact that interpreted BASIC usually runs several hundred times slower than assembly code.

At that point I did a more careful back-of-the-envelope calculation. Using the parameters described above (1000 records of 80 columns each) and ballpark

guesses at other parameters (50 BASIC instructions per column and one hundred BASIC instructions per second) gave the following.

1000 recs	80 cols	50 insts	1 sec
	rec	col	100 insts

Multiplying and cancelling shows that the job takes

~~1000~~ ~~recs~~	~~80~~ ~~cols~~	~~50~~ ~~insts~~	~~1~~ sec	40000
	~~rec~~	~~col~~	~~100~~ ~~insts~~	

or 40,000 seconds. Dividing by the identity 3600 secs/hr gives the estimate of about eleven hours. Alternatively, I knew that the old machine took one minute for the task and executed an instruction in about ten microseconds. The slowdown to ten milliseconds is a factor of one thousand, and one thousand times the previous value of one minute is about seventeen hours.

Had I known the expense of this approach before I built the program, I would have used a faster language. Instead, I had an existing 600-line program and no choice but to tune the code, using the techniques of Column 8. The 70 lines of code in Phase 1 accounted for over 90 percent of the run time, and just 3 lines accounted for 11 hours (less than one percent of the code took 75 percent of the time!). I spent forty hours replacing 70 lines of BASIC with 110 lines of BASIC and 30 lines of assembly code; that reduced the time of Phase 1 from fourteen hours to two hours and twenty minutes. That was good enough for this particular system, but more than it might have been had I done a quick calculation beforehand and then chosen a more efficient implementation language.

6.5 Principles

When you use back-of-the-envelope calculations, be sure to recall Einstein's famous advice.

Everything should be made as simple as possible, but no simpler.

We know that simple calculations aren't too simple by including safety factors to compensate for our mistakes in estimating parameters and our ignorance of the problem at hand.

6.6 Problems

1. At what distances can a courier on a bicycle with a reel of magnetic tape be a more rapid carrier of information than a telephone line that transmits 56,000 bits per second? Than a 1200-bps line?
2. How long would it take you to fill a disk by typing?

3. When is it cost-effective to supply a programmer with a home terminal?
4. Suppose the world is slowed down by a factor of a million. How long does it take for your computer to execute an instruction? Your disk to rotate once? Your disk arm to seek across the disk? You to type your name?
5. Which has the most computational oomph: a second of supercomputer time, a minute of midicomputer time, an hour of microcomputer time, or a day of BASIC on a personal computer?
6. Suppose that a system makes 100 disk accesses to process a transaction (although some systems need fewer, some systems require several hundred disk accesses per transaction). How many transactions per hour per disk can the system handle?
7. A programmer spends one calendar day and one hour of CPU time to speed up a program by ten percent on a machine that costs one hundred dollars per hour of CPU time. Individual runs of the program typically require a minute of CPU time. How long will it take to pay for the speedup if the program is run a hundred times a day? What if the speedup were a factor of two or a factor of ten?
8. [R. Pike] Many compilers have optimizers that, when enabled, produce more efficient code. If your compiler has such an option, measure how much more compile time it takes and how much faster the resulting object code is. When is it worthwhile to run your optimizer? When is it worthwhile to build such an optimizer?
9. I was once asked to write a program to transmit data from one personal computer to another PC of a very different architecture. Since the file had only 400 records of 20 numeric digits each, I suggested re-keying the data from a readily available listing. Estimate costs associated with each approach, including programmer time, hardware investment, and transmission cost.
10. An article on page 652 of the July 1984 *Communications of the ACM* states that "the system handles an average of 7,328,764 transactions a day". Any comments?
11. Use quick calculations to estimate the run time of designs described in this book.
    a. Evaluate the designs in Problems 1.7, 2.7 and 5.1 and in Sections 2.2, 2.4, 5.2, 13.1 and 13.3.
    b. "Big-oh" arithmetic can be viewed as a formalization of quick calculations — it captures the growth rate but ignores constant factors. Use the "big-oh" run times of the algorithms in Columns 5, 7, 10, 11 and 12 to estimate the run time of their implementation as programs. Compare your estimates to the experiments reported in the columns.

6.7 Further Reading

Douglas Hofstadter's "Metamagical Themas" column in the May 1982 *Scientific American* is subtitled "Number numbness, or why innumeracy may be just as dangerous as illiteracy"; it is reprinted with a postscript in his book *Metamagical Themas*, published by Basic Books in 1985. It is a fine introduction to ballpark estimates and an eloquent statement of their importance.

Physicists are well aware of this topic. After this column appeared in *Communications of the ACM*, Jan Wolitzky wrote

> I've often heard "back-of-the-envelope" calculations referred to as "Fermi approximations", after the physicist. The story is that Enrico Fermi, Robert Oppenheimer, and the other Manhattan Project brass were behind a low blast wall awaiting the detonation of the first nuclear device from a few thousand yards away. Fermi was tearing up sheets of paper into little pieces, which he tossed into the air when he saw the flash. After the shock wave passed, he paced off the distance travelled by the paper shreds, performed a quick "back-of-the-envelope" calculation, and arrived at a figure for the explosive yield of the bomb, which was confirmed much later by expensive monitoring equipment.

Edward Purcell edits a monthly column in the *American Journal of Physics* entitled "The back of the envelope". It is full of such delightful questions as "A 60-watt bulb lit for a year takes how many barrels of oil?" and "How long would it take to transmit over a video channel the information contained in the human genome, approximately 1 meter of DNA?"

6.8 Quick Calculations in Everyday Life *[Sidebar]*

The publication of this column in *Communications of the ACM* provoked many interesting letters. One reader told of hearing an advertisement state that a salesperson had driven a new car 100,000 miles in one year, and then asking his son to examine the validity of the claim. Here's one quick answer: there are 2000 working hours per year (50 weeks times 40 hours per week), and a salesperson might average 50 miles per hour; that ignores time spent actually selling, but it does multiply to the claim. The statement is therefore at the outer limits of believability.

Everyday life presents us with many opportunities to hone our skills at quick calculations. For instance, how much money have you spent in the past year eating in restaurants? I was once horrified to hear a New Yorker quickly compute that he and his wife spend more money each month on taxicabs than they spend on rent. And for California readers (who may not know what a taxicab is), how long does it take to fill a swimming pool with a garden hose?

Several readers commented that quick calculations are appropriately taught at an early age. Roger Pinkham of the Stevens Institute of Technology wrote

> I am a teacher and have tried for years to teach "back-of-the-envelope" calculations to anyone who would listen. I have been marvelously unsuccessful. It seems to require a doubting-Thomas turn of mind.
>
> My father beat it into me. I come from the coast of Maine, and as a small child I was privy to a conversation between my father and his friend Homer Potter. Homer maintained that two ladies from Connecticut were pulling 200 pounds of lobsters a day. My father said, "Let's see. If you pull a pot every fifteen minutes, and say you get three legal per pot, that's 12 an hour or about 100 per day. I don't believe it!"
>
> "Well it is true!" swore Homer. "You never believe anything!"
>
> Father wouldn't believe it, and that was that. Two weeks later Homer said, "You know those two ladies, Fred? They were only pulling 20 pounds a day."
>
> Gracious to a fault, father grunted, "Now that I believe."

Several other readers discussed teaching this attitude to children, from the viewpoints of both parent and child. Popular questions were of the form "How long would it take you to walk to Washington, D.C.?" and "How many leaves did we rake this year?" Administered properly, such questions seem to encourage a life-long inquisitiveness in children, at the cost of bugging the heck out of the poor kids at the time.

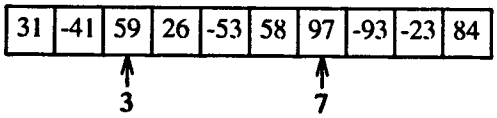
COLUMN 7: ALGORITHM DESIGN TECHNIQUES

Column 2 describes the "everyday" impact that algorithm design can have on programmers: an algorithmic view of a problem gives insights that can 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 the algorithms. Some of the algorithms are a little complicated, but with justification. While the first program we'll study takes thirty-nine days to solve a problem of size ten thousand, the final one solves the same problem in less than a second.

7.1 The Problem and a Simple Algorithm

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 iterates over all pairs of integers L and U satisfying $1 \le L \le U \le N$; for each pair it computes the sum of $X[L..U]$ and checks whether that sum is greater than the maximum sum so far. The pseudocode for Algorithm 1 is

```
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)
```

This code 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 1000 and thirty-nine days if N is 10,000; we'll get to timing details in Section 7.5.

Those times are anecdotal; we get a different kind of feeling for the algorithm's efficiency using the "big-oh" notation described in Section 5.1. 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.

This example illustrates 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 $O(N^3)$. That weakness is often compensated for by two strong points of the method. 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 you uncomfortable, peek ahead to Section 7.5, which shows that for this problem such analyses are extremely informative. Before you read further, though, take a minute to try to find a faster algorithm.

7.2 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.

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

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 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 the following code for Algorithm 2b.

```
CumArray[0] := 0.0
for I := 1 to N do
    CumArray[I] := CumArray[I-1] + X[I]
MaxSoFar := 0.0
for L := 1 to N do
    for U := L to N do
        Sum := CumArray[U] - CumArray[L-1]
        /* Sum now contains the sum of X[L..U] */
        MaxSoFar := max(MaxSoFar, Sum)
```

This code takes $O(N^2)$ time; the analysis is exactly the same as the analysis of Algorithm 2.

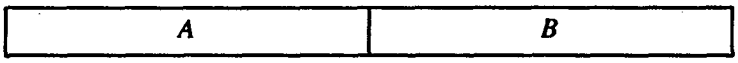
The algorithms we've seen so far inspect all possible pairs of starting and ending values of subvectors and consider the sum of the numbers in that subvector. Because there are $O(N^2)$ subvectors, any algorithm that inspects all such values must take at least quadratic time. Can you think of a way to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

7.3 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:

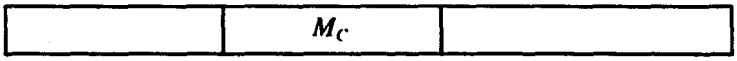
> *To solve a problem of size N, recursively solve two subproblems of size approximately N/2, and combine their solutions to yield a solution to the complete problem.*

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; we'll call that 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 the following code for Algorithm 3, which is originally invoked by the procedure call

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

```
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 + X[I]
            MaxToRight := max(MaxToRight, Sum)
    MaxCrossing := MaxToLeft + MaxToRight

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

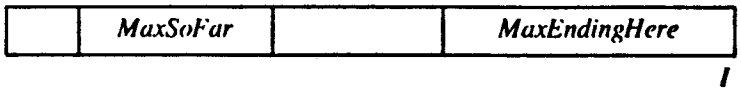
The code is complicated and easy to get wrong, but it solves the problem in $O(N \log N)$ time. There are a number of ways to prove this fact. An informal argument observes that the algorithm does $O(N)$ work on each of $O(\log N)$ levels of recursion. The argument can be made more precise by the use of recurrence relations. If $T(N)$ denotes the time to solve a problem of size N, then $T(1)=O(1)$ and

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

Problem 11 shows that this recurrence has the solution $T(N) = O(N \log N)$.

7.4 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 that 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.

```
MaxSoFar := 0.0
MaxEndingHere := 0.0
for I := 1 to N do
    /*  Invariant: MaxEndingHere and MaxSoFar
        are accurate for X[1..I-1] */
    MaxEndingHere := max(MaxEndingHere+X[I], 0.0)
    MaxSoFar := max(MaxSoFar, MaxEndingHere)
```

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 because the maximum subvector ending at I is the empty vector. Although the code is subtle, it is short and fast: its run 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 the journal *Science of Computer Programming 2*, pp. 207-214.

7.5 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 VAX-11/750, timed them, and extrapolated the observed run times to achieve the following table.

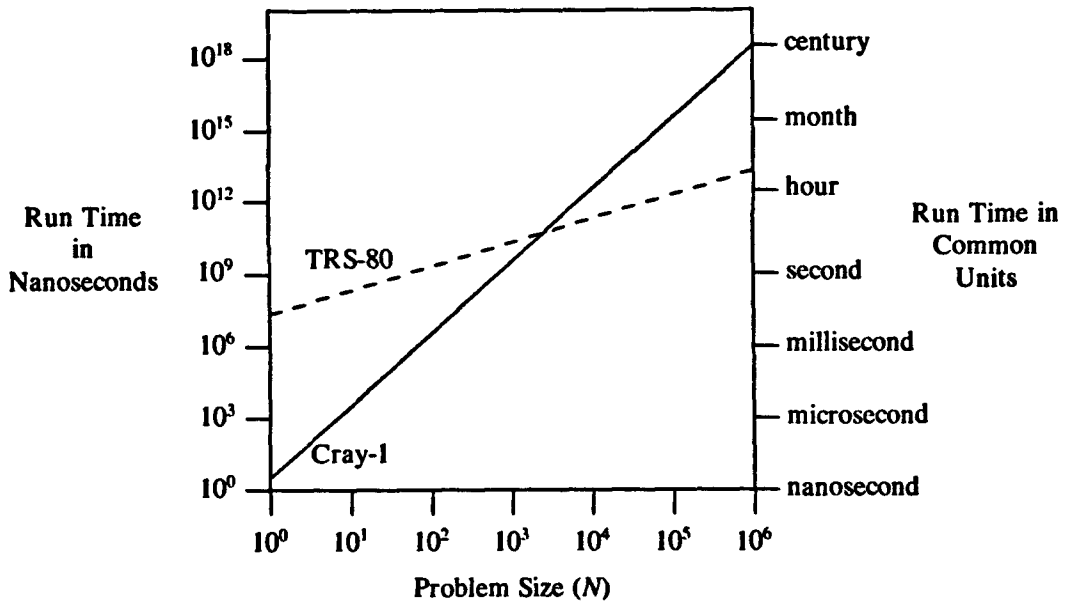
ALGORITHM		1	2	3	4
Lines of C Code		8	7	14	7
Run time in microseconds		$3.4N^3$	$13N^2$	$46N \log_2 N$	$33N$
Time to solve a problem of size	10^2	3.4 secs	.13 secs	.03 secs	.003 secs
	10^3	.94 hrs	13 secs	.45 secs	.033 secs
	10^4	39 days	22 mins	6.1 secs	.33 secs
	10^5	108 yrs	1.5 days	1.3 mins	3.3 secs
	10^6	108 mill	5 mos	15 mins	33 secs
Max size 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

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 last two rows show how increases in problem size are related to increases in run time.

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. (The discussion of the $O(N!)$ algorithm in Section 2.4 shows that constant factors matter even less in functions that grow faster than polynomially.) 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 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 $19.5N$ milliseconds, or $19{,}500{,}000N$ nanoseconds. This table shows how those expressions translate to times for various problem sizes.

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

The difference in constant factors of six and a half million allowed the cubic algorithm to start off faster, but the linear algorithm was bound to catch up. The break-even point for the two algorithms is around 2,500, where each takes about fifty seconds.



7.6 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) 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 Carnegie-Mellon seminar attended by statistician Jay Kadane, who designed Algorithm 4 within a minute. Fortunately, we know that there is no faster algorithm: any correct algorithm must take $O(N)$ time.

Even though the one-dimensional problem is completely solved, Grenander's original two-dimensional problem remained open eight years after it was posed, as this book went to press. 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!

The algorithms in this story were never incorporated into a system, but they illustrate important algorithm design techniques that have had substantial impact on many systems (see Section 7.9).

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. The *CumArray* structure in Algorithm 2b allowed the sum of a subvector to be computed 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 $X[1..I]$?" Algorithm 4 stores 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 they know their algorithms are the best possible; for this assurance, they must prove a matching lower bound. The linear lower bound for this problem is the subject of Problem 9; more complex lower bounds can be quite difficult.

7.7 Problems

1. Algorithms 3 and 4 use subtle code that is easy to get wrong. Use the program verification techniques of Column 4 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 *max* functions used by each algorithm as exactly as possible; does this exercise give any insight into the running times of the programs? How much space does each algorithm require?
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 the maximum subvector to be the value of the largest element; how would you change the programs?
4. Suppose that we wished to find the subvector with the sum closest to zero rather than that with maximum sum. What is the most efficient algorithm you can design for this task? What algorithm design techniques are applicable? What if we wished to find the subvector with the sum closest to a given real number T?
5. A turnpike consists of $N-1$ stretches of road between N toll stations; each stretch 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 with $O(N^2)$ entries. Describe 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. The method just sketched requires $O(N^2)$ time. Can you find a faster algorithm?
7. In the maximum subarray problem we are given an $N \times N$ array of reals, and we must find the maximum sum contained in any rectangular subarray. What is the complexity of this problem?
8. Modify Algorithm 3 (the divide-and-conquer algorithm) to run in linear worst-case time.
9. Prove that any correct algorithm for computing maximum subvectors must inspect all N inputs. (Algorithms for some problems may correctly ignore some inputs; consider Saxe's algorithm in Solution 2.2 and Boyer and Moore's substring searching algorithm in the October 1977 *CACM*.)

10. Given integers M and N and the real vector $X[1..N]$, find the integer I ($1 \le I \le N-M$) such that the sum $X[I]+...+X[I+M]$ is nearest zero.

11. What is the solution of the recurrence $T(N) = 2T(N/2) + CN$ when $T(1)=0$ and N is a power of two? Prove your result by mathematical induction. What if $T(1)=C$?

7.8 Further Reading

Only extensive study can put algorithm design techniques at your fingertips; most programmers will get this only from a textbook on algorithms. *Data Structures and Algorithms* by Aho, Hopcroft and Ullman (published by Addison-Wesley in 1983) is an excellent undergraduate text. Chapter 10 on "Algorithm Design Techniques" is especially relevant to this column.

7.9 The Impact of Algorithms *[Sidebar]*

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

Numerical Analysis. The standard example of the power of algorithm design is the discrete Fast Fourier Transform (FFT). 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 $N=1000$ or greater, the algorithm speeds up programs by factors of more than one hundred.

In Section 10.3.C of his *Numerical Methods, Software, and Analysis* (published in 1983 by McGraw-Hill), John Rice chronicles the algorithmic history of three-dimensional elliptic partial differential equations. Such problems arise in simulating VLSI devices, oil wells, nuclear reactors, and airfoils. A small part of that history (mostly but not entirely from his book) is given in the following table. The run time gives the number of floating point operations required to solve the problem on an $N \times N \times N$ grid.

METHOD	YEAR	RUN TIME
Gaussian Elimination	1945	N^7
SOR Iteration (Suboptimal Parameters)	1954	$8N^5$
SOR Iteration (Optimal Parameters)	1960	$8N^4 \log_2 N$
Cyclic Reduction	1970	$8N^3 \log_2 N$
Multigrid	1978	$60N^3$

SOR stands for "successive over-relaxation". The $O(N^3)$ time of Multigrid is

within a constant factor of optimal because the problem has that many inputs. For typical problem sizes ($N=64$), the speedup is a factor of a quarter million. Pages 1090-1091 of "Programming Pearls" in the November 1984 *Communications of the ACM* present data to support Rice's argument that the algorithmic speedup from 1945 to 1970 exceeds the hardware speedup during that period.

Graph Algorithms. In a common method of 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 1970's 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 *19th 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 $N=1000$ 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. "Plane sweep" or "scan line" algorithms have reduced the run time to $O(N \log N)$, so the designs can now be processed in a few hours. Szymanski and Van Wyk's "Space efficient algorithms for VLSI artwork analysis" in the *20th Design Automation Conference* describes efficient algorithms for such tasks that use only $O(\sqrt{N})$ primary memory (a later version of the paper appears in the June 1985 *IEEE Design and Test*).

Appel's program described in Section 5.1 uses a tree data structure to represent points in 3-space and thereby reduces an $O(N^2)$ algorithm to $O(N \log N)$ time. That was the first step in reducing the run time of the complete program from a year to a day.

COLUMN 8: CODE TUNING

Some programmers pay too much attention to efficiency; by worrying too soon about little "optimizations" they create ruthlessly clever programs that are insidiously difficult to maintain. Others pay too little attention; they end up with beautifully structured programs that are utterly inefficient and therefore useless. Good programmers keep perspective on efficiency: it is just one of many problems in software, but it is sometimes very important.

Previous columns discuss high-level approaches to efficiency: problem definition, system structure, algorithm design, and data structure selection. This column is about a low-level approach. "Code tuning" locates the expensive parts of an existing program and then makes little changes to the code to improve its performance. It's not always the right approach to follow and it's rarely glamorous, but it can sometimes make a big difference in a program's performance.

8.1 A Typical Story

Chris Van Wyk and I chatted about code tuning early one afternoon; he then wandered off to try it on a program. By 5:00 PM he had halved the run time of a three-thousand line program.

Given a textual description of a picture, his program produces commands to draw the picture on a phototypesetter. Although the run time for typical pictures was much shorter, the program took ten minutes to draw an extremely complicated picture. Van Wyk's first step was to *profile* the program by setting a compiler switch that caused the system to report how much time was spent in each procedure (like the second output in Solution 10). Running the program on ten test pictures showed that it spent almost seventy percent of its time in the memory allocation subroutine.

His next step was to study the memory allocator. A few lines of accounting code showed that the program allocated the most popular kind of record 68,000 times, while the runner-up was allocated only 2,000 times. Given that you knew that the majority of the program's time was spent looking through storage for a single type of record, how would you modify it to make it faster?

Van Wyk solved his problem by applying the principle of *caching*: data that is accessed most often should be the cheapest to access. He modified his program by caching free records of the most common type in a linked list. He could then handle the common request by a quick reference to that list rather than by invoking the general storage allocator; this reduced the total run time of his program to just 45 percent of what it had been previously (so the storage allocator now took just 30 percent of the total time). An additional benefit was that the reduced fragmentation of the modified allocator made more efficient use of main memory than the original allocator.

This story illustrates the art of code tuning at its best. By spending a few hours adding about twenty lines to a 3000-line program, Van Wyk doubled its speed without altering the users' view of the program or decreasing the ease of maintenance. He used general tools to achieve the speedup: profiling identified the "hot spot" of his program and caching reduced the time spent there.

8.2 A First Aid Quiz

We'll turn now from a beginning-to-end story to a quiz made of three composite stories. Each describes a typical problem that arises in several applications. The problems consumed most of the run time in their applications, and the solutions use general principles.

The first problem arises in text processing, compilers, macro processors, and command interpreters.

> *Problem One — Character Classification.* Given a sequence of one million characters, classify each as an upper-case letter, lower-case letter, digit, or other.

The obvious solution uses a complicated sequence of comparisons for each character. In the ASCII character code this approach uses six comparisons to determine that a particular character is of type "other", while the same result requires fourteen comparisons for EBCDIC. Can you do better?

One approach uses binary search — think about it. A faster solution views a character as an index into an array of character types; most programming languages provide a way to do this. For an example, let's assume that the character code has eight bits, so each character can be viewed as an integer in the range 0..255. *TypeTable*[0..255] is initialized with code in the spirit of the following fragment (although the wise programmer would try to do the job with loops).

```
for I := 0 to 255 do TypeTable[I] := Other
TypeTable['a'] := ... := TypeTable['z'] := LCLetter
TypeTable['A'] := ... := TypeTable['Z'] := UCLetter
TypeTable['0'] := ... := TypeTable['9'] := Digit
```

The type of the character *C* can then be found in *TypeTable*[*C*], which replaces a complicated sequence of character comparisons with a single array access.

This data structure typically reduces the time required to classify a character by an order of magnitude.

The next problem arises in applications ranging from set representation to coding theory.

> *Problem Two — Counting Bits.* Given a sequence of one million 32-bit words, tell how many one bits are in each word.

The obvious program uses a loop to do thirty-two shifts and logical ANDs; can you think of a better way?

The principle is the same as before: a table whose I^{th} entry contains the number of one bits in the binary representation of I. Unfortunately, most computers don't have enough room to store a table with 2^{32} entries, and the time to initialize the four billion entries would also be prohibitive. We can get around this by trading a few additional operations for a smaller table. We'll set up a table giving the counts for all eight-bit bytes, and then answer the 32-bit question by summing the answers to four eight-bit questions. The count table is initialized with two loops that have the effect of the following assignment statements; see Problem 2.

```
CountTable[0]   := 0;     CountTable[1]   := 1
CountTable[2]   := 1;     CountTable[3]   := 2
   ...
CountTable[254] := 7;     CountTable[255] := 8
```

We could use a four-iteration loop to count the bits in all bytes of word *W*, but it is just as easy and probably a little faster to unroll the loop into

```
WordCount :=   CountTable[ W              and 11111111B]
             + CountTable[(W rshift  8) and 11111111B]
             + CountTable[(W rshift 16) and 11111111B]
             + CountTable[(W rshift 24) and 11111111B]
```

The code isolates the bytes by shifting and then ANDing off all but the eight low-order bits; this operation is language- and machine-dependent. While the original solution would use over a hundred machine instructions, the above approach can usually be implemented in about a dozen instructions; it is not uncommon for this change to result (again) in a speedup of an order of magnitude. (Several other approaches to counting bits are discussed by Reingold, Nievergelt and Deo in Section 1.1 of *Combinatorial Algorithms: Theory and Practice*, published in 1977 by Prentice-Hall.)

The final problem is typical of applications that deal with geographic or geometric data.

> *Problem Three — Computing Spherical Distances.* The first part of the input is a set *S* of five thousand points on the surface of a globe; each point is represented by its latitude and longitude. After those points are stored in a data structure of our choice, the program reads the second

> part of the input: a sequence of twenty thousand points, each represented by latitude and longitude. For every point in that sequence, the program must tell which point in S is closest to it, where distance is measured as the angle between the rays from the center of the globe to the two points.

Margaret Wright of Stanford University encountered a problem similar to this in the preparation of maps to summarize data on the global distribution of certain genetic traits. Her straightforward solution represented the set S by an array of latitude and longitude values. The nearest neighbor to each point in the sequence was found by calculating its distance to every point in S using a complicated trigonometric formula involving ten sine and cosine functions. While the program was simple to code and produced fine maps for small data sets, it required several hours of mainframe time to produce large maps, which was well beyond the budget.

Because I had previously worked on geometric problems, Wright asked me to try my hand on this one. After spending much of a weekend on it, I developed several fancy algorithms and data structures for solving it. Fortunately (in retrospect), each would have required many hundreds of lines of code, so I didn't try to code any of them. When I described the data structures to Andrew Appel of Carnegie-Mellon University, he had a key insight: rather than approaching the problem at the level of data structures, why not use the simple data structure of keeping the points in an array, but tune the code to reduce the cost of computing the distance between points? How would you exploit this idea?

The cost can be greatly reduced by changing the representation of points: rather than using latitudes and longitudes, we'll represent a point's location on the surface of the globe by its x, y and z coordinates. Thus the data structure is an array that holds each point's latitude and longitude (which may be needed for other operations) as well as its three Cartesian coordinates. As each point in the sequence is processed, a few trigonometric functions translate its latitude and longitude into x, y and z coordinates, and we then compute its distance to every point in S. Its distance to a point in S is computed as the sum of the squares of the differences in the three dimensions, which is usually cheaper than computing one trigonometric function, let alone ten. This method computes the correct answer because the angle between two points increases monotonically with the square of their Euclidean distance.

Although this approach does require additional storage, it yields substantial benefits: when Wright incorporated the change into her program, the run time for complicated maps was reduced from several hours to half a minute. In this case, code tuning solved the problem with a couple of dozen lines of code, while algorithmic and data structure changes would have required many hundreds of lines.

```
I := 512; L := 0
if X[512] < T then L := 1000+1-512
while I ≠ 1 do
    /* Invariant: X[L] < T and X[L+I] >= T and I = 2**j */
    I := I div 2
    if X[L+I] < T then
        L := L+I
/* assert I = 1 and X[L] < T and X[L+I] >= T */
P := L+1
if P > 1000 or X[P] ≠ T then P := 0
```

Although the correctness argument for the code still has the same structure, we can now understand its operation on a more intuitive level. When the first test fails and L stays zero, the program computes the bits of P in left-to-right order, most significant bit first.

The final version of the code is not for the faint of heart. It removes the overhead of loop control and the division of I by two by unrolling the entire loop. In other words, because I assumes only a few distinct values in this particular problem, we can write them all down in the program, and thereby avoid computing them over and over again at run time.

```
L := 0
if X[512]   < T then L := 1000+1-512
   /* assert X[L] < T and X[L+512] >= T */
if X[L+256] < T then L := L+256
   /* assert X[L] < T and X[L+256] >= T */
if X[L+128] < T then L := L+128
if X[L+64]  < T then L := L+64
if X[L+32]  < T then L := L+32
if X[L+16]  < T then L := L+16
if X[L+8]   < T then L := L+8
if X[L+4]   < T then L := L+4
if X[L+2]   < T then L := L+2
   /* assert X[L] < T and X[L+2]   >= T */
if X[L+1]   < T then L := L+1
   /* assert X[L] < T and X[L+1]   >= T */
P := L+1
if P > 1000 or X[P] ≠ T then P := 0
```

We can understand this code by inserting the complete string of assertions like those surrounding the test of $X[L+256]$. Once you do the two-case analysis to see how that `if` statement behaves, all the other `if` statements fall into line.

I've compared the binary search of Section 4.2 with this fine-tuned binary search on several systems, with the following results.

array, then its first occurrence is in position U; that fact is stated more formally in the `assert` comment. The final two statements set P to the index of the first occurrence of T in X if it is present, and to zero if it is not present. (The final statement must use a "conditional `or`" that does not evaluate the second clause if the first clause is true; see Problem 10.2.)

While this binary search solves a more difficult problem than the previous program, it is potentially more efficient: it makes only one comparison of T to an element of X in each iteration of the loop. The previous program sometimes had to test two such outcomes.

The next version of the program uses a different representation of a range: instead of representing $L..U$ by its lower and upper values, we'll represent it by its lower value L and an increment I such that $L+I=U$. The code will ensure that I is at all times a power of two; this property is easy to keep once we have it, but it is hard to get originally (because the array is of size $N=1000$). The program is therefore preceded by an assignment and `if` statement to ensure that the range being searched is initially of size 512, the largest power of two less than 1000; thus L and $L+I$ are either 0..512 or 489..1001. Translating the previous program to this new representation of a range yields this code.

```
I := 512
if X[512] >= T then
    L := 0
  else
    L := 1000+1-512
while I ≠ 1 do
    /* Invariant: X[L] < T and X[L+I] >= T and I = 2**j */
    NextI := I div 2
    if X[L+NextI] < T then
        L := L + NextI; I := NextI
      else
        I := NextI
/* assert I = 1 and X[L] < T and X[L+I] >= T */
P := L+1
if P > 1000 or X[P] ≠ T then P := 0
```

The correctness proof of this program has exactly the same flow as the proof of the previous program. This code is usually slower than its predecessor, but it opens the door for future speedups.

The next program is a simplification of the above, incorporating some optimizations that a smart compiler might perform. The first `if` statement is simplified, the variable *NextI* is removed, and the assignments to *NextI* are removed from the inner `if` statement.

```
I := 512; L := 0
if X[512] < T then L := 1000+1-512
while I ≠ 1 do
    /* Invariant: X[L] < T and X[L+I] >= T and I = 2**j */
    I := I div 2
    if X[L+I] < T then
        L := L+I
/* assert I = 1 and X[L] < T and X[L+I] >= T */
P := L+1
if P > 1000 or X[P] ≠ T then P := 0
```

Although the correctness argument for the code still has the same structure, we can now understand its operation on a more intuitive level. When the first test fails and L stays zero, the program computes the bits of P in left-to-right order, most significant bit first.

The final version of the code is not for the faint of heart. It removes the overhead of loop control and the division of I by two by unrolling the entire loop. In other words, because I assumes only a few distinct values in this particular problem, we can write them all down in the program, and thereby avoid computing them over and over again at run time.

```
L := 0
if X[512]   < T then L := 1000+1-512
   /* assert X[L] < T and X[L+512] >= T */
if X[L+256] < T then L := L+256
   /* assert X[L] < T and X[L+256] >= T */
if X[L+128] < T then L := L+128
if X[L+64]  < T then L := L+64
if X[L+32]  < T then L := L+32
if X[L+16]  < T then L := L+16
if X[L+8]   < T then L := L+8
if X[L+4]   < T then L := L+4
if X[L+2]   < T then L := L+2
   /* assert X[L] < T and X[L+2]   >= T */
if X[L+1]   < T then L := L+1
   /* assert X[L] < T and X[L+1]   >= T */
P := L+1
if P > 1000 or X[P] ≠ T then P := 0
```

We can understand this code by inserting the complete string of assertions like those surrounding the test of $X[L+256]$. Once you do the two-case analysis to see how that `if` statement behaves, all the other `if` statements fall into line.

I've compared the binary search of Section 4.2 with this fine-tuned binary search on several systems, with the following results.

MACHINE	LANGUAGE	OPTIMI-ZATIONS	UNITS	SLOW CODE	FAST CODE	SPEEDUP FACTOR
MIX	Assembly	Extreme	Tyme	18.0	4.0	4.5
TRS-80	BASIC	None	millisecs	43.6	14.6	3.0
PDP-10 (KL)	Pascal	None	microsecs	16.4	5.5	3.0
	Assembly	Extreme		4.5	0.9	5.0
VAX-11/750	C	None	microsecs	33.8	13.3	2.5
		Source code		22.5	12.2	1.8
		Compiler		29.2	12.8	2.3
		Both		19.7	12.2	1.6

The first line says that Knuth's assembly language implementation of the code in Section 4.2 runs in roughly $18.0 \log_2 N$ MIX Tyme units, while his implementation of the fast program in this section is 4.5 times faster. The speedup factor is dependent on many variables, but it is significant in all the above cases.

This derivation is an idealized account of code tuning at its most extreme. We replaced the obvious binary search program (which doesn't appear to have much fat on it) with a super-lean version that is several times faster.† The program verification tools of Column 4 played a crucial role in the task. Because we used them, we can believe that the final program is correct; when I first saw the final code presented without verification, I looked upon it as magic for months.

8.4 Principles

The most important principle about code tuning is that it should be done rarely. That sweeping generalization is explained by the following.

The Role of Efficiency. Many other properties of software are as important as efficiency, if not more so. Don Knuth has observed that premature optimization is the root of much programming evil; it can compromise the correctness, functionality and maintainability of programs. Save concern for efficiency for when it matters.

Profiling. When efficiency is important, the first step is to profile the system to find out where it spends its time. The solution to Problem 10 shows the output of two profilers. Such output usually shows that most of the time is going to a few hot spots and that the rest of the code is almost never executed (in Section 5.1, for instance, one procedure accounted for 98

† This program has been known in the computing underground since the early 1960's. Some of its history is sketched on page 93 of "Programming Pearls" in the February 1984 *Communications of the ACM*.

percent of the run time; in Section 6.4, 10 percent of the code accounted for 90 percent of the run time). Profiling points to the critical areas; for the other parts we follow the wise maxim of, "If it ain't broke, don't fix it."

Design Levels. We saw in Column 5 that there are many ways to solve efficiency problems. Before tuning code, we should make sure that other approaches don't provide a more effective solution.

The above discussion considers whether and when to tune code; once we decide to do so, that still leaves the question of how. I tried to answer that question in my book *Writing Efficient Programs* with a list of general rules for code tuning. All the examples we've seen can be explained in terms of those principles; I'll do that now with the names of the rules in *italics*.

Van Wyk's Drawing Program. The general strategy of Van Wyk's solution was to *Exploit Common Cases*; his particular exploitation involved *Caching* a list of the most common kind of record.

Problem One — Character Classification. The solution with a table indexed by a character *Precomputes A Logical Function*.

Problem Two — Counting Bits. The table of byte counts is closely related to the previous solution; it *Stores Precomputed Results*.

Problem Three — Computing Spherical Distances. Storing Cartesian coordinates along with latitudes and longitudes is an example of *Data Structure Augmentation*; using the cheaper Euclidean distance rather than the angular distance *Exploits An Algebraic Identity*.

Binary Search. *Combining Tests* reduced the number of array comparisons per inner loop from two to one, *Exploiting An Algebraic Identity* changed representations from a lower and upper bound to a lower bound and an increment, and *Loop Unrolling* expanded the program to remove all loop overhead.

So far we've tuned code to reduce CPU time. One can tune code for other purposes, such as reducing paging or increasing a cache hit ratio. Perhaps the most common use of code tuning beyond reducing run time is to reduce the space required by a program. Problem 4 gives a taste of that endeavor, and the next column is devoted to the topic.

8.5 Problems

1. The character classification program in the text assumed that the character classes were disjoint. How would you write a routine to test membership in overlapping character classes, such as lower case letters, upper case letters, letters, digits and alphanumerics?
2. Write a code fragment that given N, a power of two, initializes *CountTable*[0..$N-1$] as described in the text.

3. How do the various binary search algorithms behave if they are (against specification) applied to unsorted arrays?
4. In the early days of programming, Fred Brooks faced the problem of representing a large table on a small computer. He couldn't store the entire table in an array because there was room for only a few bits for each table entry (actually, there was one decimal digit available for each entry — I said that it was in the early days!). His second approach was to use numerical analysis to fit a function through the table. That resulted in a function that was quite close to the true table (no entry was more than a couple of units off the true entry) and required an unnoticeably small amount of memory, but legal constraints meant that the approximation wasn't good enough. How could Brooks get the required accuracy in the limited space?
5. The typical sequential search to determine whether T is in $X[1..N]$ was given in Problem 4.9.

```
I := 1
while I <= N and X[I] ≠ T do I := I+1
```

   A common example of code tuning speeds that up by placing T in a "sentinel" position at the end of the array.

```
X[N+1] := T
I := 1
while X[I] ≠ T do I := I+1
```

   Eliminating the test $I<=N$ typically reduces the run time of the program by twenty or thirty percent. Implement the two programs and time them on your system.
6. How can sentinels be used in a program to find the maximum element in an array (see Problem 4.9)? How can sentinels decrease the search time in sets represented by linked lists, hash tables, and binary search trees?
7. Because sequential search is simpler than binary search, it is usually more efficient for small tables. On the other hand, the logarithmic number of comparisons made by binary search implies that it will be faster than the linear time of sequential search for large tables. The break-even point is a function of how much each program is tuned. How low and how high can you make that break-even point? What is it on your machine when both programs are equally tuned? Is it a function of the level of tuning?
8. D. B. Lomet of IBM Watson Research Center observes that hashing may solve the 1000-integer search problem more efficiently than the tuned binary search. Implement a fast hashing program and compare it to the tuned binary search; how do they compare in terms of speed and space?
9. In the early 1960's, Vic Berecz of the United Technologies Corporation found that most of the time in a simulation program at Sikorsky Aircraft was devoted to computing trigonometric functions. Further investigation

showed that the functions were computed only at integral multiples of five degrees. How did he reduce the run time?

10. Use a profiler to gather statistics on the performance of a program.
11. One sometimes tunes programs by thinking about mathematics rather than code. To evaluate the polynomial

$$y = a_n x^n + a_{n-1}x^{n-1} + \cdots + a_1 x^1 + a_0$$

the following code uses $2N$ multiplications. Give a faster routine.

```
Y := A[0]; XToTheI := 1
for I := 1 to N do
    XToTheI := X*XToTheI
    Y := Y + A[I]*XToTheI
```

12. Apply the techniques of this column to real programs. Try, for instance, the sorting program in Section 1.4 and the anagram program in Section 2.8.

8.6 Further Reading

Although I am willing to admit to a certain personal bias, my favorite book on code tuning is my own *Writing Efficient Programs*, published by Prentice-Hall in 1982. The heart of the book is a fifty-page discussion of the efficiency rules mentioned above; each is stated in general terms and then illustrated by application to small code fragments and by "war stories" of its application in real systems. Other parts of the book discuss the role of efficiency in software systems and the application of the rules to several important subroutines.

8.7 Tuning the Federal Government's COBOL Code *[Sidebar]*

The role of code tuning in data processing is discussed in the General Accounting Office report "Improving COBOL Applications Can Recover Significant Computer Resources" (1 April 1982, Order Code PB82-198540, National Technical Information Service, Springfield, Virginia 22161). These application systems are from the Department of Housing and Urban Development.

CPU Time Reduction (Percent)	Dollar Savings Per Year	Cost To Optimize
82	$37,400	$5,500
45	45,000	1,200
30	4,400	2,400
19	9,000	900
9	7,000	9,000

The optimizations cost a total of $19,000; over the four-year minimum life of

the systems, they will save almost $400,000. In an Army application, one staff-day of optimization reduced the typical elapsed time of a program from 7.5 hours to less than two hours; this change saved $3,500 in its first year and resulted in the output being delivered to the user more reliably.

The report warns that code should not be tuned haphazardly; other considerations such as correctness and maintainability must be given their rightfully high priority. It points out that there is usually a point of diminishing returns in tuning either an individual program or a set of programs on a system: work beyond that point will be very difficult and have little positive impact.

The recommendations in the report include the following: "Heads of Federal agencies should require periodic review of the machine resource consumption of COBOL applications at their installations, and, where feasible, require action to reduce the consumption of the expensive applications."

COLUMN 9: SQUEEZING SPACE

If you're like several people I know, your first thought on reading the title of this column is "How old-fashioned!" In the bad old days of computing, so the story goes, programmers were constrained by small machines, but those days are long gone. The new philosophy is "a megabyte here, a megabyte there, pretty soon you're talking about real memory". And there is truth in that view — many programmers use big machines and rarely have to worry about squeezing space from their programs.

But every now and then, thinking hard about compact programs can be profitable. Sometimes the thought gives new insight that makes the program simpler.† Reducing space often has desirable side-effects on run time: smaller programs are faster to load, and less data to manipulate usually means less time to manipulate it. Even with cheap memories, space can be a critical. Many microprocessors have 64-kilobyte address spaces, and sloppy use of virtual memory on a large machine can lead to disastrously slow thrashing.

Keeping perspective on its importance, let's survey some techniques for reducing space.

9.1 The Key — Simplicity

Simplicity can yield functionality, robustness, speed and space. Fred Brooks observed this when he wrote a payroll program for a national company in the mid 1950's. The bottleneck of the program was the representation of the Kentucky state income tax. The tax was specified in the law by the obvious two-dimensional table (income as one dimension, number of exemptions as the other). Storing the table explicitly required thousands of words of memory, more than the capacity of the machine.

† In their paper about the UNIX operating system, which was developed on a small machine, Ritchie and Thompson remark that "there have always been fairly severe size constraints on the system and its software. Given the partially antagonistic desires for reasonable efficiency and expressive power, the size constraint has encouraged not only economy but a certain elegance of design." See page 374 of the July 1974 *Communications of the ACM*.

The first approach Brooks tried was to fit a mathematical function through the tax table, but it was so jagged that no simple function would come close. Knowing that it was made by legislators with no predisposition to crazy mathematical functions, Brooks consulted the minutes of the Kentucky legislature to see what arguments had led to the bizarre table. He found that the Kentucky tax was a simple function of the income that remained *after* federal tax was deducted. His program therefore calculated federal tax from existing tables, and then used the remaining income and a table of just a few dozen words of memory to find the Kentucky tax.

By studying the context in which the problem arose, Brooks was able to replace the original problem to be solved with a simpler problem. While the original problem appeared to require thousands of words of data space, the modified problem was solved with a negligible amount of memory.

Simplicity can also reduce code space. Column 3 describes several large programs that were replaced by small programs with more appropriate data structures. In those cases, a simpler view of the program reduced the source code from thousands to hundreds of lines and probably also shrank the size of the object code by an order of magnitude.

9.2 Data Space

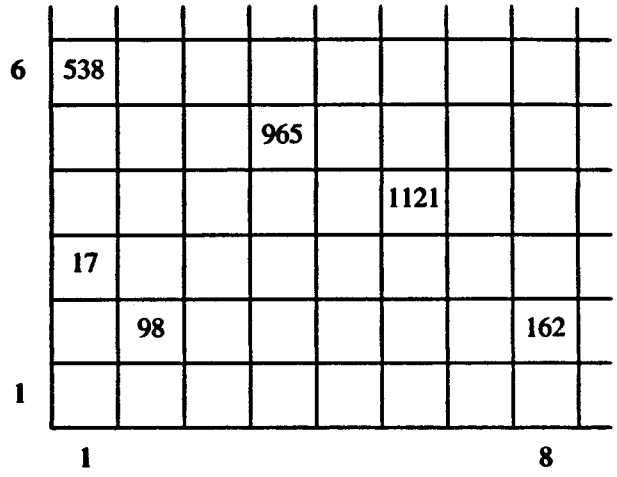
Although simplification is usually the easiest way to solve a problem, some hard problems just won't yield to it. In this section we'll study techniques that reduce the space required to store the data accessed by a program; in the next section we'll consider reducing the memory space used to store the program as it is executing.

Don't Store, Recompute. The space required to store a given object can be dramatically reduced if we don't store it but rather recompute it whenever it is needed. In the early days of computing, some programs stored large tables of, for instance, the *sine* function. Today, virtually all programs compute trigonometric functions by calling a subroutine. This greatly reduces space requirements, and because of advances in numerical analysis and floating point hardware design, it is only slightly more expensive than interpolating from a large table. Similarly, a table of the prime numbers might be replaced by a subroutine for testing primality. This method trades more run time for less space, and it is applicable only if the objects to be "stored" can be recomputed from their description.

Such "generator programs" are often used in executing several programs on identical random inputs, for such purposes as performance comparisons or regression tests of correctness. Depending on the application, the random object might be a file of randomly generated lines of text or a graph with randomly generated edges. Rather than storing the entire object, we store just its generator program and the random seed that defines the particular object. By

taking a little more time to access them, objects that have many megabytes can be represented in a few bytes.

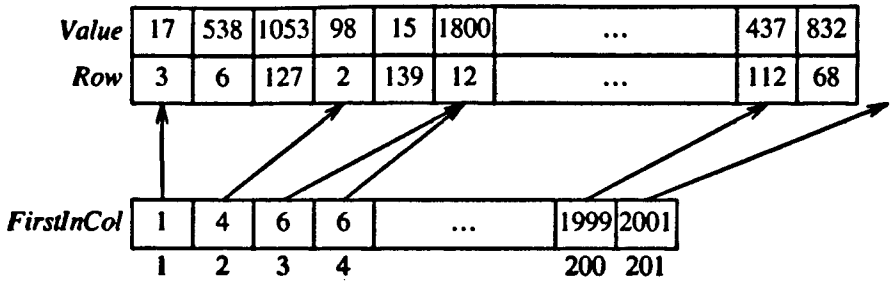
Sparse Data Structures. Replacing a data structure can drastically reduce the space required to store given information. I once encountered a system that allowed the user to access any of two thousand points on a map by touching an input pad. The program converted the physical location selected to a pair of integers with x in the range 1..200 and y in the range 1..150 — the board was roughly four feet by three feet and the program used quarter-inch resolution. It then used that (x,y) pair to tell which, if any, of the two thousand points the user had chosen. Because no two points could be in the same (x,y) location, the programmer represented the map by a 200×150 array of point identifiers (an integer in the range 1..2000, or zero if no point was at that location). The bottom left corner of that array might look like this, where zero point identifiers are represented by empty squares.



In the corresponding map, point 17 is in location (1,3), point 538 is in (1,6), and the four other visible locations in the first column are empty.

Although the array was easy to implement and gave rapid access time, its 200×150=30,000 16-bit words consumed over ten percent of the memory on the half-megabyte machine. When the system started to run out of space, the programmer hoped that we could reduce the storage in this structure. What would you suggest?

I'll describe our solution in FORTRAN terms because that was the language we used; if you use a language with more powerful data structuring facilities, take a minute to think about how you would express this solution. Here are the three arrays in our solution, with the integer indices in the bottom array also depicted as arrows.



The points in column I are represented in the *Row* and *Value* arrays between locations *FirstInCol*[I] and *FirstInCol*[$I+1$]-1; even though there are only 200 columns, *FirstInCol*[201] is defined to make this condition hold. In the above picture there are three points in the first column: point 17 is in (1,3), point 538 is in (1,6), and point 1053 is in (1,127). There are two points in column 2, none in column 3, and two in column 200. To determine what point is stored in position (I,J) we use the following pseudocode.

```
for K := FirstInCol[I] to FirstInCol[I+1]-1 do
    if Row[K] = J then
        /* Found it in position K */
        return Value[K]
/* Unsuccessful search; (I,J) empty */
return 0
```

This method uses much less space than its predecessor: two 2000-element arrays and one 201-element array give 4201 16-bit words instead of 30,000. Although it is a little slower than its predecessor (in the very worst case an access costs 150 comparisons, but it uses half a dozen on the average), the program had no trouble keeping up with the user. Because of the good module structure of the system, this approach was incorporated in a few hours by changing a few subroutines. We observed no degradation in run time and gained fifty sorely needed kilobytes.

This solution illustrates several general points about data structures. The problem is classic: sparse array representation (a sparse array is one in which most entries have the same value, usually zero). The solution is conceptually simple and easy to implement using only arrays. Note that there is no *LastInCol* array to go with *FirstInCol*; we instead use the fact that the last point in this column is one before the first point in the next column. This is a trivial example of recomputing rather than storing. Similarly, there is no *Col* array to go with *Row*; because we only access *Row* through the *FirstInCol* array, we always know the current column.

Many other data structure techniques can reduce space. In Section 3.1 we saved space by storing a "ragged" three-dimensional table in a two-dimensional array. If we use a key to be stored as an index into a table, then we need not store the key itself; rather, we store only its relevant attributes, such as a count of how many times it has been seen. Applications of this *key indexing*

technique were discussed in Sections 1.4 and 8.2 and Problems 1.7 and 1.8. In the sparse matrix example above, key indexing through the *FirstInCol* array allowed us to do without a *Col* array. Storing pointers to shared large objects (such as long text strings) removes the cost of storing many copies of the same object, although one has to be careful when modifying a shared object that all its owners desire the modification. This technique is used in my desk almanac to provide calendars for the years 1821 through 2080; rather than listing 260 distinct calendars it gives fourteen canonical calendars (seven days of the week for January 1 times leap year or non-leap year) and then a table giving a calendar number for each of the 260 years.

Data Compression. Insights from information theory reduce space by encoding objects compactly. In the sparse matrix example, for instance, we assumed that elements of *Row* were 16-bit integers. Because each row value is an integer in the range 1..150, that array could instead be represented by 8-bit bytes, which would save another kilobyte. In a microcomputer-based business system I encoded the two decimal digits A and B in one byte (instead of the obvious two) by the integer $N=10\times A+B$. The information was decoded by the two statements

```
A := N div 10
B := N mod 10
```

This simple scheme squeezed a file of numeric data onto one floppy disk instead of two.† Such encodings can reduce the space required by individual records, but the small records usually take more time to process because they must first be decoded.

Information theory can also compress a stream of records being transmitted over a channel such as a telecommunications line or a disk file. Such compression techniques are typically subtle to implement, but they can lead to substantial savings: Column 13 sketches how a file of 30,000 English words was squeezed into 26,000 16-bit computer words. Details on these techniques can be found in the references.

Allocation Policies. Sometimes how much space you use isn't as important as how you use it. Suppose, for instance, that your program uses three different types of records, *X*, *Y*, and *Z*, all of the same size. In some languages your first impulse might be to declare, say, one hundred objects of each of the three types. But what if you used 101 *X*'s and no *Y*'s or *Z*'s? The program could run out of space after using 101 records, even though 200 others were completely unused. *Dynamic allocation* of records avoids such obvious waste by allocating records as they are needed. Most modern languages provide such

† Several readers suggested using the encoding $N=(A \text{ lshift } 4)$ or B; the values can be decoded by the statements $A:=N$ rshift 4 and $B:=N$ and 1111_2. John Linderman observes that "not only are shifting and masking commonly faster than multiplying and dividing, but common utilities like a hex dump could display the encoded data in a readable form".

a mechanism, but even in primitive languages such as FORTRAN, a programmer can implement such a policy in a user-level routine.

Dynamic allocation says that we shouldn't ask for something until we need it; the policy of *variable-length records* says that when we do ask for something, we should ask for only as much as we need. In the days of eighty-column records it was common for more than half the bytes on a program library disk to be trailing blanks. Variable-length files denote the end of lines by a "new line" character and thereby double the storage capacity of such disks. I once tripled the speed of a microcomputer program by using tape records of variable length: the maximum record length was 250, but only about 80 bytes were used on the average.

More advanced allocation techniques are described in the references. *Garbage collection* recycles discarded storage so that the old bits are as good as new. The Heapsort algorithm in Section 12.4 *overlays* two logical data structures used at separate times in the same physical storage locations. For another approach to *sharing* storage, Brian Kernighan once wrote a traveling salesman program in which the lion's share of the space was devoted to two $N \times N$ matrices, where N was 150. The two matrices, which I'll call A and B to protect their anonymity, represented distances between points. Kernighan therefore knew that they had zero diagonals ($A[I,I]=0$) and that they were symmetric ($A[I,J]=A[J,I]$). He therefore let the two triangular matrices share space in one square matrix, C, one corner of which looked like

0	B[1,2]	B[1,3]	B[1,4]	
A[2,1]	0	B[2,3]	B[2,4]	
A[3,1]	A[3,2]	0	B[3,4]	
A[4,1]	A[4,2]	A[4,3]	0	

Kernighan could then refer to $A[I,J]$ by the code

```
C[max(I,J), min(I,J)]
```

and similarly for B, but with the *min* and *max* swapped. This representation has been used in various programs since the dawn of time. The technique made Kernighan's program somewhat more difficult to write and slightly slower, but the reduction from two 22,500-word matrices to just one was significant on a 30,000-word machine. And if the matrices were 900×900, the same change would have the same effect today on a four-megabyte machine.

9.3 Code Space

Sometimes the space bottleneck of a program is not its data but rather the size of the program itself. For instance, I subscribe to hobbyist magazines that publish graphics programs with page after page of code like

```
for I := 17 to 43 do Set(I,68)
for I := 18 to 42 do Set(I,69)
for J := 81 to 91 do Set(30,J)
for J := 82 to 92 do Set(31,J)
```

where *Set*(*X*,*Y*) turns on the picture element at screen position (*X*,*Y*). Appropriate subroutines, say *Hor* and *Vert* for drawing horizontal and vertical lines, would allow that code to be replaced by

```
Hor(17,43,68)
Hor(18,42,69)
Vert(81,91,30)
Vert(82,92,31)
```

This code could in turn be replaced by an interpreter that read commands from a text file like

```
H 17 43 68
H 18 42 69
V 81 91 30
V 82 92 31
```

If that still took too much space, each of the lines could be represented in a 32-bit word by allocating two bits for the command (H, V, or two others) and ten bits for each of the three numbers, which are integers in the range 0..1023. (The translation would, of course, be done by a program.) This hypothetical case illustrates several general techniques for reducing code space.

Subroutine Definition. Replacing a common pattern in the code by a subroutine simplified the above program and thereby reduced its space and increased its clarity. This is a trivial instance of "bottom-up" design. Although one can't ignore the many merits of "top-down" methods, the homogeneous world view given by good primitive routines can make a system easier to maintain and simultaneously reduce space.

Featurecide reduces code space by removing functionality from subroutines. Sections 9.6 and 9.7 give references on this delicate topic.

Interpreters. In the hypothetical case above we were able to replace a long line of program text with a four-byte command to a special-purpose interpreter. Section 3.2 describes an interpreter for producing form letters; although its main purpose is to make a program simpler to build and to maintain, it incidentally reduces the program's space.

Translation to Machine Language. One aspect of space reduction over which most programmers have relatively little control is the translation from the source language into the machine language. For instance, some minor

compiler changes reduced the code space of early versions of the UNIX system by as much as five percent. As a last resort, a programmer might consider coding a large system into assembly language by hand, but this is an expensive, error-prone process that usually yields only small dividends. In an unpublished experiment, Lynn Robert Carter (then of Motorola Software Technology) found that spending ten weeks recoding 1500 lines of Pascal into 2900 lines ot assembly code reduced the size of the object module from 16,000 to 12,000 bytes. Those four kilobytes were crucial in the application (the system had run a few kilobytes over the 128K on the processor board and there was no room in the chassis for a memory board), but the effort works out to ten bytes per hour, or roughly five dollars per byte.

9.4 Principles

Now that we've seen the techniques for reducing space, let's consider the attitude we should have as we approach the problem.

The Cost of Space. What happens if a program uses ten percent more space? On many systems such an increase will have no cost: previously wasted bits are now put to good use. On small systems the program might not work at all: it runs out of memory. On large time-sharing systems the cost might rise by exactly ten percent. On virtual-memory systems the run time might increase dramatically because the program that previously fit in physical memory now makes many disk accesses — Problem 2.4 describes how increasing the problem size of a program by a few percent increased its run time by two orders of magnitude. Know the cost of space before you set out to reduce it.

The "Hot Spots" of Space. Section 8.4 described how the run time of programs is usually clustered in hot spots: a few percent of the code accounts for much of the run time. The opposite is true in the space required by code: whether an instruction is executed a billion times or not at all, it requires the same space to store. Data space can have hot spots: a few common records may account for most of the storage. In the sparse matrix example, for instance, a single data structure accounted for more than ten percent of the storage used on a half-megabyte machine. Replacing it with a structure one-tenth the size had a substantial impact on the system; reducing a one-kilobyte structure by a factor of a hundred would have had negligible impact.

Tradeoffs. Sometimes a programmer must trade away performance, functionality or maintainability to gain space; such engineering decisions should be made only after all alternatives are studied. Several examples in this column showed how reducing space can sometimes have a positive impact on the other dimensions. In Section 1.4, a bitmap data structure allowed a set of records to be stored in internal memory rather than on disk and thereby reduced the run time from minutes to seconds and the code from hundreds to dozens of lines. This happened only because the original solution was far from optimal, but we

programmers who are not yet perfect often find our code in exactly that state. We should acknowledge that fact and look for techniques that improve all aspects of our solutions before we trade away any desirable properties.

Work with the Environment. The programming environment can have a substantial impact on the space efficiency of a program. Important issues include the representations used by a compiler and run-time system, memory allocation policies, and paging policies. When you're almost out of space, make sure that you know enough about these issues to ensure that you aren't working against the system.

Use the Right Tool for the Job. We've seen four techniques that reduce data space (Recomputing, Sparse Structures, Information Theory, and Allocation Policies), three techniques that reduce code space (Subroutine Definition, Interpreters, and Translation) and one overriding principle (Simplicity). When space is critical, consider all your options.

9.5 Problems

1. In the late 1970's Stu Feldman built a FORTRAN 77 compiler that barely fit in a 64-kilobyte code space. To reduce space he had packed the elements of several kinds of records into four-bit fields. When he unpacked the records by storing the fields in eight-bit bytes, he found that although the data space had increased by a few hundred bytes, the overall size of the program went down by several thousand bytes. What happened?
2. Questions about the sparse matrix example: Are there other simple but space-efficient data structures for the task? How would you write a program to build the data structure described in the text? How would you change the structure so that it could be searched more quickly without using too much additional storage? Suppose storage became even more scarce — how much further can you reduce the space?
3. Study data in non-computer applications such as almanacs or mathematical tables for examples of squeezing space.
4. A BASIC program in a hobbyist magazine contained DATA statements that together define a sequence of about four thousand one-byte integers. The first few integers in the sequence are

   128, 128, 128, 128, 128, 128, 128, 128, 128, 128, 128, 128, 128, 128, 128, 128,
   128, 128, 128, 128, 128, 128, 128, 128, 128, 128, 128, 128, 128, 128, 128, 128,
   128, 128, 128, 128, 128, 128, 152, 166, 172, 153, 164, 128, 128, 128, 128, 128,
   128, 128, 128, 128, 128, 128, 128, 128, 128, 128, 128, 128, 128, 128, 128, ...

   The program listed the numbers in four pages of double-column text. What are more appropriate representations?
5. The discussion of "Data Compression" in Section 9.2 mentioned decoding $10 \times A + B$ with a *div* and a *mod* operation. Discuss the time and space tradeoffs involved in replacing those operations by table lookups.

6. Design a linked list that has only one "pointer" per node yet can be traversed in forward or reverse order.
7. In a common type of program *profiler*, the value of the instruction counter is sampled on a regular basis; see, for instance, Solution 8.10. Design a data structure for storing those values that is efficient in time and space and also provides useful output.
8. The (remarkably handsome) face in Section 3.3 is stored in 48×48 bytes. Suppose that you had to store many faces with similar characteristics; how far could you reduce the space requirements?
9. COBOL programmers typically allocate six bytes for a date in the twentieth century (MMDDYY), nine bytes for a social security number (DDD-DD-DDDD), and 21 bytes for a name (12 for last, 8 for first, and 1 for middle initial). If space is critical, how far can you reduce those requirements?
10. [A. Appel] Compress an online dictionary of English to be as small as possible. When counting space, measure both the data file and the program that interprets the data.

9.6 Further Reading

Many data structures texts discuss techniques for reducing data space. Standish's *Data Structures Techniques* (published in 1980 by Addison-Wesley) is particularly strong in this area. Relevant sections include 2.5 (managing several stacks), 5.4 (storage reclamation), 5.5 (compacting and coexistence), 6.2 (dynamic storage allocation), 7.2 (string representation), 7.4 (variable-length string representation), and 8.3 through 8.5 (array representation).

Chapter 9 of Fred Brooks's *Mythical Man Month* (published by Addison-Wesley in 1975) is entitled "Ten pounds in a five-pound sack"; it concentrates on managerial control of space in large projects. He raises such important issues as size budgets, module function specification, and trading space for function or time.

9.7 Two Big Squeezes *[Sidebar]*

The body of this column surveys various techniques; let's now inspect two space-critical systems to see how the techniques come into play.

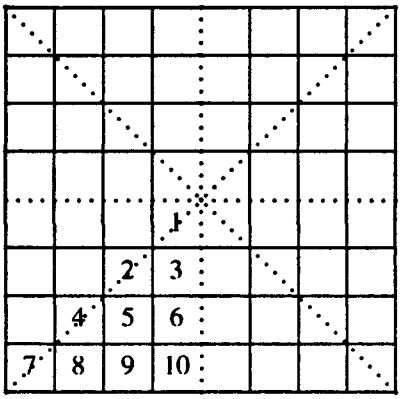
Ken Thompson has developed a two-phase program that solves chess endgames† for given configurations such as a King and two Bishops matched against a King and a Knight. The learning phase of the program computes the distance to checkmate for all possible chessboards (over the given set of four or five pieces) by working backwards from all possible checkmates; computer scientists will recognize this technique as dynamic programming, while chess

† This program is distinct from the Belle chess machine developed by Joe Condon and Thompson.

experts know it as retrograde analysis. The resulting database makes the program omniscient with respect to the given pieces, so in the game-playing phase it plays perfect endgames. The game it plays is described by chess experts with terms like "complex, fluid, lengthy and difficult" and "excruciating slowness and mystery", and it has already upset established chess dogma.

Explicitly storing all possible chessboards was prohibitively expensive in space. Thompson therefore used an encoding of the chessboard as a key to index a disk file of board information; each record in the file contained 12 bits, including the distance to checkmate from that position. Because there are 64 squares on a chessboard, the positions of five fixed pieces can be encoded by five integers in the range 0..63 that give the location of each piece. The resulting key of 30 bits implies a table of 2^{30} or about 1.07 billion 12-bit records in the database, which exceeded the capacity of the disk.

Thompson's key insight was that chessboards that are mirror images around any of the dotted lines in the following figure have the same value and need not be duplicated in the database.



His program therefore assumed that the white King was in one of the ten numbered squares; an arbitrary chessboard can be put into this form by a sequence of at most three reflections. This normalization reduces the disk file to 10×64^4 or 10×2^{24} 12-bit records. Thompson further observed that because the black King cannot be adjacent to the white King, there are only 454 legal board positions for the two Kings in which the white King is in one of the ten squares marked above. Exploiting that fact, his database shrunk to 454×64^3 or about 121 million 12-bit records, which fit comfortably on a single (dedicated) disk.

For a second space-critical system, we'll consider Apple's Macintosh computer; details on the machine and its design history can be found in the February 1984 issue of *BYTE*. The space constraint in that system was the fact that an enormous amount of system functionality had to be squeezed into a 64 kilobyte Read-Only Memory (ROM). The design team achieved this by careful subroutine definition (involving generalizing operators, merging routines and featurecide) and hand coding the entire ROM in assembly language. They

estimate that their extremely tuned code, with careful register allocation and choice of instructions, is half the size of equivalent code compiled from a high-level language.

These two systems differ in many ways. The Mac design team was motivated to squeeze 128 kilobytes of code into a 64K ROM because they anticipated shipping millions of copies. Even though Thompson knew there would be just one copy of his program, he had to squeeze a file onto one disk. The Mac team tuned assembly code to reduce ROM space; Thompson exploited symmetry in data structures to reduce disk space.

The two efforts do share some attributes, though. Both cases reduced space by a small constant factor (2 for the Mac ROM, 8 for the endgame disk) that was critical for the success of the entire system. Squeezing space also reduced the run time of both systems. The tight assembly code in the Mac can perform input/output at a megabit per second, and decreasing the number of positions in the endgame program reduced the time of its learning phase from a year to a few weeks.