Fast algorithms dramatically decrease run time for certain tasks. Designing them offers not only the promise of better performance but also a better understanding of the problems underlying all computational tasks.

SPECIAL FEATURE

An Introduction to Algorithm Design

Jon Louis Bentley
Carnegie-Mellon University

"Algorithm design—that's the field where people *talk* about programs and *prove theorems* about programs instead of *writing* and *debugging* programs." Statements along these lines have been uttered by applications programmers and academicians alike. But there are also some who say, "No! Proper algorithm design has helped us to save kilobucks at our installation every month." Here we will investigate the field of algorithm design (also known as "analysis of algorithms" and "concrete computational complexity," among other names) and better equip the reader to judge the field for himself.

I trust that anyone with even the slightest love for mathematics burning somewhere in his heart (however deeply) will want to see how mathematical tools can be applied to the problems of programming. But for the other readers (whose interest in mathematics was probably squelched in freshman calculus), I offer the same bait that hooked me on this field. I trace my interest in the design of efficient algorithms to the time when I was a business data processing programmer and had just finished reading an introductory text on data structures. A colleague of mine had just had his program cancelled—the operators, by counting the turning rate of the tapes, had estimated that it would take about three hours to process his one reel of data. The program itself was fairly short and a quick glance told us that the time was spent in scanning a 1000-element table. I suggested that instead of scanning we try a new-fangled technique I had just read about—binary search. We did, and the modified program processed the reel of tape in *five minutes*, and actually spent most of its time waiting for the tape! Around that time I was also asked to help another programmer who had already spent a month producing over a thousand cards of code for a particular program. A simple change in data structure and a few days' work allowed us to redo the program in less than two hundred lines of code. The redone program was faster than the original would have been, used far less code, and was much easier to understand. So even if you have no aesthetic interest in algorithm design (yet), please read on—the practical benefits alone are rewarding enough.

Throughout this article I refer only to the *discrete* aspects of algorithm design. I do not address problems in the domain of numerical analysis, such as stability, truncation error, and error propagation. Even with this restriction, I include some very numeric problems, such as the manipulation of sparse matrices, in which almost all elements are zero, and the fast Fourier transform.

A number of survey papers on discrete algorithm design have appeared recently. Hopcroft[1] and Tarjan[2] both give a thorough picture of the field. Weide's survey[3] concentrates on techniques for analyzing discrete algorithms, and accomplishes that task expertly. For those who are skeptical of sweeping surveys, Knuth's excellent introductions[4,5] examine a few problems in detail. And if one is ready to become a serious student of the field, the standard texts are Aho, Hopcroft, and Ullman's one-semester, graduate-level introduction[6] and Knuth's first three volumes[7-9] of his proposed seven-volume definitive work on algorithms. I hope to supplement those works by providing a broad survey for the novice. For this reason, I have kept my list of references short; both Tarjan and Weide, cited above, contain excellent bibliographies.

This survey discusses algorithm analysis from three viewpoints. We first examine five problems and some algorithms for solving them. Having con-

0018-9162/79/0020-0066$00.75 © 1979 IEEE

sidered these concrete examples, we next take a systematic tour of the field. We then investigate some current research directions and conclude with an assessment of the field's value to practitioners.

Some problems and fast algorithms for solving them

The five problems presented here serve as background to our general discussions of algorithm design. We will not only give an algorithm for solving each, but will also mention some real-world situations in which our problems could exist and to which our algorithms could be applied. We will analyze the algorithms' efficiency and discuss what this analysis shows. We will study the first problem—"subset testing"—in detail and then treat the remaining problems at a more superficial level. Before moving on to our overview of algorithm design, we will consider what all this attention to efficiency gets us.

But first let me offer an explanation of our concern with these examples. The subset testing problem will raise familiar issues and should cover some old ground for many; it also gives us a nice illustration of the tremendous time savings we can get by using proper algorithms. The next example—the substring searching problem—provides an extremely interesting blend of theory and practice. The fast Fourier transform—FFT—is the third example. It is known to many, uses some important algorithmic techniques, and is eminently practical. Fourth, we examine a very old problem—matrix multiplication—and a recent and remarkably counterintuitive solution. We conclude our examination of specific problems by investigating algorithmic aspects of a public-key cryptosystem. This system has recently revolutionized the world of cryptography and promises to have a substantial impact on "secure" computing.

We will see an application in which our first algorithm would require over six days of CPU time, while our final algorithm can solve the problem in four seconds.

Subset testing. Given a set A of size n and a set B of size $m \leqslant n$, is B a subset of A?[10] We can state this "subset testing" problem as a programming exercise: given an array A[1:n] and B[1:m], both of (say) 32-bit words, is every word in B also in A? Disguised versions of this problem arise in many contexts: set A could be an employee master file, set B a list of weekly transactions, and the problem a question of finding whether a master-file record exists for each weekly transaction. Or A might be a table of real numbers x with an associated table S containing sine x, B then being a set of x values at which the sine function is to be evaluated. But even though this problem does have many such practical applications, that is not our main motive in examining it here. Rather, it leads to many of the basic problems in sorting and searching, and points to interrelationships between those problems. It also exposes us to some common algorithm design methods.

We will examine three solutions to this problem. Finding the running time of each, by counting the number of comparisons between elements, will enable us to compare the solutions. The following may also encourage the reader as he follows the different solutions: our first algorithm will require over six days of CPU time to solve a sample problem, while our final algorithm will solve it in four seconds.

Brute force. The simplest way to accomplish this task is to compare every element in B to each of the elements of A until either its equal is found or we have examined all of A and determined that every element of B has no equal in A (in which case B is not A's subset); this approach gives a simple, two-loop program. If B is indeed contained in A, then each scan for an element that is B's mate in A takes $n/2$ comparisons on the average (you have to look halfway down the list). Since there are m such scans made, the total number of comparisons made by this program is about $m(n/2)$. So if m is very close to the size of n, then we will make about $n^2/2$ comparisons on the average.* Although this algorithm is exceptionally simple to understand and to code, its slow running time might prohibit its use in certain applications. We will now turn our attention to a faster algorithm.

Sorting. If you were given a randomly ordered list of phone numbers B (say a list of phone numbers in a town) and another randomly ordered list A (say all phone numbers in the county) and you were asked to check whether B was a subset of A (make sure every town phone number is included in the county list), then you might use the brute-force algorithm just discussed. If, however, you were handed a town phone book and a county phone book and asked to perform the same task, then your job would be much easier. Since the two phone books are already sorted by name, we can just scan through the two books together, ensuring that the county book contains all the town names. This of course immediately gives us another algorithm for subset testing: sort A, sort B, then sequentially scan through the two, checking for matches. To analyze the run time of this strategy we observe that the scan will take about n comparisons, and we heard somewhere that you can sort a list of size n in about $n \log_2 n$ comparisons, so the total running time is $(n \log_2 n) + (m \log_2 m) + n$ comparisons.

We could pull a sorting routine out of thin air, but it is not much more difficult to describe one called Mergesort. The basic operation of Mergesort is merging two sorted lists of numbers, say X and Y, with these lists either stored as arrays or linked together

*We won't try to analyze the case that B is not a subset of A; to do so we would have to say exactly how it is not a subset, and that is very dependent on the particular problem.

with pointers. To do this we compare the first element of X with the first element of Y and give the smallest as the first element of the new list, deleting it from its source. We repeat this remove-the-smallest step until both X and Y are empty. Since we used one comparison for each step, if there were a total of m elements in X and Y, we will have used about m comparisons. We can now use this tool of merging to Mergesort a set S of n elements. We start by viewing S as a set of n sorted one-element lists. We then merge adjacent pairs of one-element lists, giving $n/2$ two-element sorted lists. The next step is to merge adjacent pairs of those lists giving $n/4$ four-element lists, and the process continues. After $\log_2 n$ iterations we have one sorted n-element list, and our task is complete. To analyze this we note that we use about n comparisons for the merges at each of the $\log_2 n$ iterations, so the total number of comparisons used is the promised $n \log_2 n$.

We have thus shown how to solve the subset problem with $n(\log_2 n + 1) + m \log_2 m$ comparisons. If m is about the same size as n then our algorithm takes approximately $2n \log_2 n$ comparisons. Can we do better?

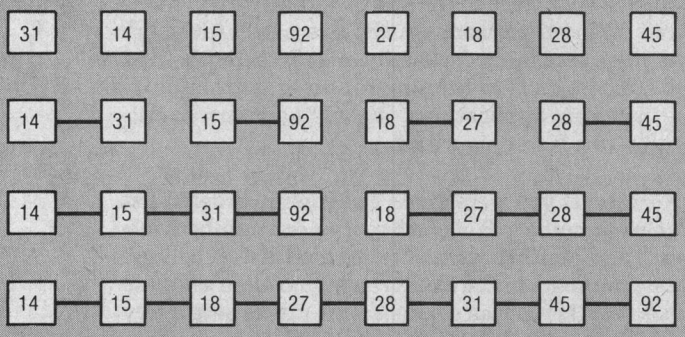
Hashing. Thinking about the phone book problem leads to an interesting sorting approach to the subset problem; if we rephrase the phone book problem then the "human" approach will lead to an even faster subset algorithm. Suppose that the county phone book (A) was sorted and the town phone list (B) was not; to ensure that A contains B we can "look up" in A each number in B by the name of the subscriber. For each of the m elements in B we would do a "binary search" among the n elements of A. It is not hard to see that a binary search in an n-element sorted table takes at most $\log_2 n$ comparisons, so this algorithm is easily analyzed: it takes $n \log_2 n + m \log_2 n$ comparisons, or approximately $2n \log_2 n$ if m is the same size as n. We therefore have a searching solution to the subset problem: store the elements of A in a table, then for each element of B ensure that it is in the table.

Although binary search is the best searching method for many problems, there is another strategy—hashing—even more appropriate for this problem. By hashing we can store an element in a table or check to see if it already is in a table in about two comparisons, on the average.* With this approach we will be able to do subset testing in $2n + 2m$ comparisons—$2n$ to store A and then $2m$ to look up each element of B. To store the n elements of A we will have to allocate a *hash table*—an array of length $(1.5)n$.** We then store the elements of A in the table one-by-one by the use of a *hash function.* This maps a data value into an integer in the bounds of the hash table. If that position in the hash table is empty, fine. We insert the element. If the position is occupied, however, we have a *collision* and must employ a *collision resolution strategy,* such as scanning up the elements of the array until a free position is found. Analysis shows that a proper collision resolution strategy allows us to find an empty spot very quickly—say, in two comparisons. When an empty spot is finally found the element is inserted. After inserting all of A's elements into the table we then look up all of B's elements. For any particular element we calculate its hash function and look in that position. If that position is empty then it is not in A; if the element is in the position then we have found it; otherwise we must employ the same collision resolution strategy to see where it should be. Hashing is something that a human would never use in searching (humans are much better at comparing things and then looking in one of two directions than at calculating weird hash functions), but it leads to a very efficient algorithm. If m is about the same size as n then the hashing approach uses only about $4n$ comparisons, on the average, to do subset testing.

*For pessimists, however, we note that the worst case of hashing is as bad as brute force—we might have to look at all of the elements in the table.

**We can even use a smaller array; $(1.1)n$ would probably work almost as well.

Mergesort



The above illustrates the operation of Mergesort applied to a list of eight elements. The top row shows the eight elements in the order they were presented for sorting. We then view each successive pair of elements (31 and 14, 15 and 92, etc.) as pairs of one-element lists and merge those lists—this amounts to placing the smaller element as the first element in the list of length two. The second row depicts this result. We now have four lists, each of length two. As the next step of our algorithm, we merge the two successive pairs of lists, giving two (sorted) lists of length four. As the final step of our algorithm, we merge that pair of lists, resulting in the list of all elements in sorted order as shown in the bottom row.

This example had three "levels" of merging, each requiring work proportional to the number of elements in the list. In sorting an n-element list we will have $\log_2 n$ levels, with work proportional to n on each level. Careful analysis shows that if n is a power of two, we will use exactly $n \log_2 n - n + 1$ comparisons. If we assume that in a program implementing this algorithm each comparison costs approximately one microsecond, then we can sort a million elements in approximately 20 seconds. Using the well-known bubble sort for the same problem could require up to *six days.* (We can be encouraged by the fact that bubble sort will only take three days on the average—practice saying that with a straight face to your DP manager!)

The subset testing problem, although a very simple one, led us straight to some of the fundamental issues in algorithm design. We very quickly arrived at searching—the scan of the brute force algorithm was just a naive search. From there we moved to sorting, then to binary search, and finally to hashing, which introduced us to a non-obvious data structure—the hash table.* The approaches that we used to solve the problem are some of the fundamental tools of algorithm designers. We have also touched on aspects of algorithmic problems such as time and space analysis and worst-case versus expected-time analysis, which we will study later.

But what has all this gained us? Although we may have a better understanding of some of the basic computational issues, does our understanding make any difference in practice? Let's assume that we are writing a program for subset testing where A and B both contain one million elements, and that one comparison takes, say, one microsecond of computer time. Under these assumptions, the $n^2/2$ comparisons required by brute force translates to 138 hours, or a little shy of six days, of machine time; the $2n \log_2 n$ for sorting will give 40 seconds; and the $4n$ of hashing will yield 4 seconds. Although we haven't calculated all the costs of implementation, this example shows how a simple analysis is sometimes all we need to make an informed choice.

Substring searching. Does a given *string* contain a specified substring *pattern*, and if so, where? This is the substring searching problem, one familiar to most who have used computer text editors. As I sat down to type this paragraph, for example, I told the editor to find the substring "Substring searching." in my text file so I would know where to begin my paragraph.** Information retrieval systems use this same operation as they identify abstracts which contain certain keywords. Similar problems are encountered in many text formatting and macro processing programs.

It is not hard to write a program to solve this problem. We first hold *pattern*'s leftmost character under *string*'s leftmost character and start comparing. If all the characters of *pattern* match the characters above them, fine—we have found the substring in position 1. If we find a mismatch then we slide *pattern* over one and do the same thing again. This continues until we either find a match or come to the end of *string*. The worst-case behavior of this algorithm is very slow—for each n positions of *string* we might have to compare all m positions of *pattern*. Thus in the worst case we might have to make mn comparisons. Strings and patterns that realize this worst-case behavior are fairly pathological and the performance of this algorithm in practice is fairly good, but the question still haunts us—can we design an algorithm that will always do better?

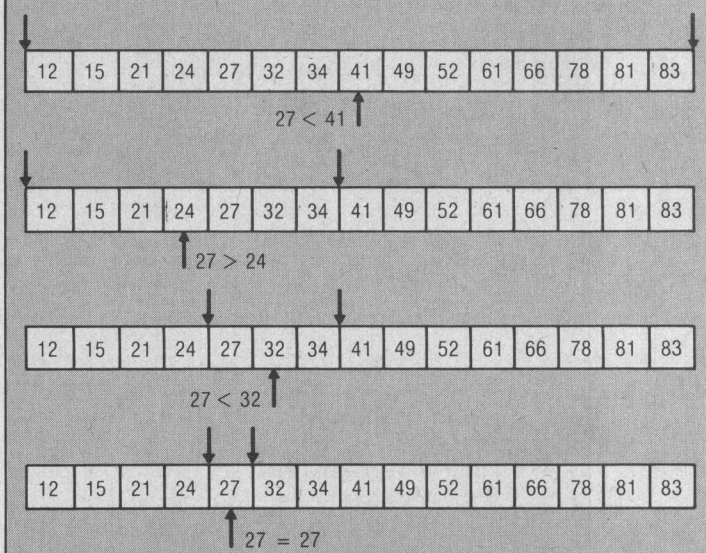
*A more thorough examination of searching is contained in Knuth.[11]

**The text editor I use looks at a file as one long string of text, sprinkled with special characters representing "carriage return."

Knuth, Morris, and Pratt[12] offer an algorithm that beats the mn performance. They preprocess *pattern* into a data structure that represents a program; that program then looks for *pattern* in *string*. Preprocessing *pattern* takes only m operations (where m is the length of *pattern*) and the "program" they produce looks at each character of *string* only once, so the total running time of their algorithm is proportional to $m + n$ instead of mn. (Of course if the *pattern* is in the ith position in *string*, then their algorithm takes time proportional to $i + m$.) This result is exceptionally interesting from a theoretical viewpoint, and also provides a faster substring searching algorithm in practice.

Boyer and Moore[13] recently used the basic idea of the Knuth, Morris, and Pratt algorithm to derive an

Binary search



This figure shows the operation of binary search in locating the position of the number 27 in a sorted table. At the start of binary search we only know that the element, if in the table at all, lies between 12 and 83, inclusive. This is denoted by the two "down-arrows" at the top row. We then compare the middle element of that sequence—41—to 27. Since 27 is less than 41 we can exclude positions 41 to 83 from our search, denoted by the down-arrows in the second row. Binary search continues by always examining the middle element of the sequence in which the "target" value is known to lie. We show that middle element by an "up-arrow" below it. The binary search terminates by either finding the desired element or showing that the desired element is not in the table.

To analyze binary search we note that each time it makes a comparison the size of the subtable containing the desired element is *cut by one-half*. For example, if we search a 1000-element table, after one comparison we are down to only 500 elements, after two comparisons to 250, and so on. After about ten comparisons we either find our man or discover he is not in the table. In general if there are n elements in the table, binary search requires approximately $\log_2 n$ comparisons. Thus, if we use it to search a one-million-element table, it will require only 20 comparisons, a substantial improvement over the *half-million* comparisons used by the average linear scan.

even faster method of substring searching. Their method has the same worst-case performance (proportional to $m + n$) but is faster on the average. They accomplish this by making it unnecessary to examine every element of *string*. They have implemented their algorithm on a PDP-10 so efficiently that when *string* contains typical English text and *pattern* is a five letter word in *string*, the number of PDP-10 instructions executed is less than $i + n$. This is at least an order of magnitude faster than the naive algorithm.

The history of the substring searching problem provides an insight into the relation of theory to practice. Knuth relates that his use of a machine from automata theory, the "two-way deterministic pushdown automaton," led him to his discovery of the algorithm. The easiest way to understand fast algorithms is through the use of finite state automata, which are commonly used in digital systems design. It is noteworthy that in this one problem we can talk about such diverse ideas as abstract automata and PDP-10 instructions, with a lot of combinatorial analysis in between!

The fast Fourier transform. The Fourier transform is often studied in mathematics and engineering. It can be viewed in a number of ways, such as the transformation of a function from the "time domain" into the "frequency domain" or as the decomposition of a function into its "sinusoidal components." The continuous Fourier transform has a discrete counterpart, which applies an operation to one set of n reals yielding a "transformed" set of n reals. This problem has applications in signal processing, interpolation methods, and many other discrete problems.

The naive algorithm for computing the Fourier transform of n reals requires approximately n^2 arithmetic operations (adds and multiplies). The fast Fourier transform of Cooley and Tukey[14] accomplishes this task in approximately $n \log_2 n$ arithmetics. It achieves this by doing about n arithmetics on each of $\log_2 n$ levels; in this sense it is quite similar to the Mergesort algorithm discussed earlier. There are many different expositions of the algorithm, such as those in Aho, Hopcroft, and Ullman[15] or Borodin and Munro.[16] It is interesting to note that in addition to being faster, many of the numeric properties of the FFT are better than those of the naive transform.

The fast Fourier transform has had a substantial impact on computing. It forms the backbone of many "numeric" programs. Practitioners in diverse fields use the FFT to find hidden periodicities of stationary time series. Signal processing uses it in filters to remove noise from signals and eradicate blurring in digital pictures. Numerical analysis utilizes it for the interpolation and convolution of functions. Applications of the FFT in such diverse areas as electrical engineering, acoustics, geophysics, medicine, economics, and psychology are listed by Brillinger.[17] Many special-purpose processors have been built which implement this algorithm; some are multiprocessors which operate in parallel. The FFT is also widely used in the design of "discrete" algorithms. It is the primary element in many algorithms which operate on polynomials, performing such operations as multiplication, division, evaluation, and interpolation. Not surprisingly, it is also employed in some of the fastest known algorithms for operating on very long integers, such as multiplying two 1000-bit integers. We will see an application of this problem later.

Matrix multiplication. One of the most common ways of representing many different kinds of data is in a matrix, and one of the most common operations on matrices is multiplication. How hard is it to multiply two n x n matrices? Using the standard high school method takes about $2n$ arithmetic operations to calculate each of the n^2 elements of the product matrix, so the total amount of time required by that

Matrix multiplication

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \times \begin{bmatrix} e & f \\ g & h \end{bmatrix} = \begin{bmatrix} p & q \\ r & s \end{bmatrix}$$

We can compute a product matrix in the following (non-straightforward) fashion. First we compute the seven "temporary" values by the following seven multiplications:

$$\begin{aligned} t1 &= (b-d) * (g+h) \\ t2 &= (a+d) * (e+h) \\ t3 &= (a-c) * (e+f) \\ t4 &= (a+c) * h \\ t5 &= a * (e-f) \\ t6 &= d * (g-e) \\ t7 &= (c+d) * e \end{aligned}$$

We can now combine these temporary values to yield the product matrix:

$$\begin{aligned} p &= t1 + t2 - t4 + t6 \\ q &= t4 + t5 \\ r &= t6 + t7 \\ s &= t2 - t3 + t5 - t7 \end{aligned}$$

The "high school" method of multiplying two-by-two matrices requires eight multiplications and four additions. This method requires only seven multiplications, but 18 additions and subtractions. This hardly seems like a worthwhile tradeoff!

Amazingly enough, however, we can use the above scheme *recursively* to produce an algorithm for matrix multiplication that is asymptotically more efficient than the $O(n^3)$ high school algorithm—it requires $O(n^{2.81})$ time. Although the savings it achieves are offset by the additional overhead in implementation (it is probably inferior to the high school algorithm until the matrix contains at least 10,000 elements), it is remarkable to find we can multiply faster than people have been doing it for the past hundred years.

algorithm is proportional to n^3. People have been multiplying matrices by this method for a century. Surely this must be the best possible way to multiply matrices—our intuition tells us that we just can't do any better.

The high school algorithm for multiplying two-by-two matrices uses eight multiplications and four additions. It is fairly counterintuitive to learn that the product can be computed using only seven multiplications at the cost of an increase of 15 additions. But if that is counterintuitive, then it is absolutely mind-boggling to find that this fact alone allows us to construct an algorithm for multiplying $n \times n$ matrices that runs in less than n^3 time! This algorithm, devised by Strassen,[18] decomposes each $n \times n$ matrix into four $(n/2) \times (n/2)$ matrices. It does seven multiplications of $(n/2) \times (n/2)$ matrices and then fifteen additions on matrices of that size to find the product of the original matrices. Notice, however, that the cost of those additions is proportional to n^2. If we let $T(n)$ be the time required to multiply $n \times n$ matrices, then $T(n)$ satisfies the recurrence

$$T(1) = 1,$$
$$T(n) = 7T(n/2) + O(n^2)$$

having the solution $T(n) = O(n^{2.81})$ (where 2.81 is an approximation to $\log_2 7$). The Strassen algorithm as originally presented is less efficient than the high school algorithm until n is in the thousands. Recent work, however, shows that it can be practical when n is as small as 40. But practice aside, who can help but be amazed by the fact that we can multiply matrices faster than we thought we could?

The fast matrix multiplication algorithm provided the basis for one of the all-time great revolutions in the history of "theoretical" algorithm design, during which a number of "best" known algorithms were toppled from their thrones. Many of these were n^3 matrix algorithms which we can now do in $O(n^{2.81})$ time; among these are matrix inversion, LU decomposition, solutions to systems of linear equations, and calculations of determinants. A number of problems which seemed to be totally unrelated to matrices were phrased in terms of them. As a result $O(n^{2.81})$ algorithms followed for such diverse problems as finding the transitive closure of a graph, parsing context-free languages (an important problem in compilers), and finding distances between n points in Euclidean n-space. All of these algorithms stem from the fact that two-by-two matrices can be multiplied with seven multiplications.

Public-key cryptography. Communications systems which deal with the problem of transmitting a message from a sender to a receiver across an insecure and possibly bugged channel, while protecting the privacy of the message, are known as *cryptosystems*. Such security needs often arise in military applications; they also promise to play an ever increasing role in applications such as electronic mail and electronic funds transfer. A cryptosystem is usually implemented by *encoding* and *decoding algorithms* which transform their inputs according to *keys* they are given. To send person X message M we use the encoding algorithm and key to produce message M', transmitting M' to X across the possibly insecure channel. When X receives M' he uses the decoding algorithm and key to determine M, while any "eavesdropper" has only M'. One difficulty with this system is the distribution of keys to the appropriate parties. This must usually be accomplished by the use of expensive secure channels such as human couriers.

An alternative, called a *public-key* cryptosystem, was recently invented by Hellman and Diffie.[19] In their system each person has an encoding key and a decoding key, as before. To send a message to person X we encode it using X's encoding key, and then X can decode it with his decoding key. The novel aspect of this system is that we can make the encoding key public without revealing the corresponding decoding key. We can then view the encoding key as the address of X's mail box, and anyone can put mail into that box simply by encoding the message. Actually unlocking the box, however, requires the decoding key, which only X possesses. Although such a system solves almost all of the difficulties of previous cryptosystems, one major obstacle remains: for most codes knowledge of the encryption key immediately reveals the decryption key. Thus all we need to complete our public-key cryptosystem is an appropriate en-

O-notation

Algorithm analysts frequently employ the "big-oh" notation of mathematics. In this notation we can write an expression like

$$T(n) = 6n^2 + 9.3n = O(n^2)$$

to show the running time—$T(n)$—of an algorithm. The more accurate expression for $T(n)$ is much more complicated to express than the simple $O(n^2)$; we therefore see a tradeoff between the accuracy of the statement and the ease of expressing the formula (as well as the ease of deriving the expression). There is a well-developed mathematical theory of "order arithmetic" that allows derivations to be made without regard to the constant factors of the leading terms. Most calculations in this arithmetic are very simple.

"Big-oh" notation is a useful tool for analyzing algorithms. The running times of most algorithms to within "big-ohs" are much more easily obtained than are the exact analyses of the algorithms. If we know that algorithm A takes $O(n^2)$ time and that algorithm B takes $O(n \log n)$ time, then regardless of the constants "hidden" in the notation, algorithm B will be faster than algorithm A for large enough values of n. The "big-oh" analysis will therefore be of great help in choosing an algorithm that is to process a huge amount of data, as long as we are careful not to be caught unaware by the constants hidden in the notation.

coding/decoding algorithm, provided such an algorithm is possible.

A suitable encoding/decoding method was recently developed by Rivest, Shamir, and Adleman.[20] Their method is based on algorithmic issues in the theory of numbers. They view messages as multiprecision (long) integers of 200 decimal digits, for example. The coding procedure then transforms these messages by sophisticated use of modular arithmetic and prime number theory. The transformations require many sophisticated algorithms. For instance, the process of key selection is based on fast algorithms for multiplying multiprecision integers and testing such integers for primality; fast multiprecision exponentiation algorithms then perform encoding and decoding. The fact that any method that "breaks" the system must essentially find the factors of a very large number is the basis of the system's security. Although no one knows precisely how difficult this factoring is, mathematicians have been working on the problem for many centuries, and no one yet knows of a fast method. Let me put this in perspective: if we deal with 200 decimal digit integers, then the key selection, encoding, and decoding algorithms require only a few seconds of CPU time; the best known method for breaking the system, however, would require *ten million centuries* of CPU time.

We have only scratched the surface of the fascinating field of public-key cryptography. In addition to use in cryptosystems, these methods can also be used to provide "electronic signatures," i.e., verifications of identity. This cryptosystem is another interesting example of the interactions between theory and practice. It is based on number theory (perhaps the purest of the areas of pure mathematics) and complexity theory, yet it promises to revolutionize the practice of cryptography. The interested reader should refer to the article by Rivest, Shamir, and Adleman, cited above, or the exposition in Gardner.[21] Although the algorithms we have mentioned do not really solve an existing computational problem, they solve a problem in a totally different area by casting it in a computational light.

Most of the time, a fast algorithm makes no difference. Sometimes, it drastically increases throughput. But it always gives us an understanding of our computational problems.

We have now examined five cases in which proper algorithm design has led to a sophisticated algorithm that is much faster than a naive algorithm. Algorithm designers have invested a lot of work in developing these algorithms—but what difference will it all make in practice?

To be honest, most of the time a fast algorithm makes no difference at all. Knuth has gathered empirical evidence showing that most of the run time of a program is spent in just three percent of the code. If an algorithm for a problem is not in the critical three percent of the code, it makes little difference if the algorithm is fast or not. A complicated algorithm is often a liability rather than an asset. It will usually mean more coding and more debugging time, and can sometimes even increase the run time (when the overhead of "starting up" a fancy algorithm costs more than the time it saves).

Sometimes, however, a fast algorithm can make all the difference in the world. If a certain computation is indeed the bottleneck in the system flow, then an algorithm of half the running time almost doubles system throughput. Many text editors spend a vast majority of their time in string searching; the fast algorithm for substring searching can speed up many text editors by a factor of five. My experience with the searching program I mentioned in the introduction, where we reduced the running time from three hours to five minutes, is another good example of an appropriate use for a fast algorithm. In the inner loops of many programs, proper algorithm design is critical.

Fast algorithm design, although only occasionally yielding large financial savings, almost always gives us something of a different value—a fundamental understanding of our computational problems. This is usually reflected in cleaner programs; but even more important is the understanding of how difficult it is to compute something. After a student has spent a month or two investigating the problem of searching, he knows not only how to search faster, but also why he can do it that fast and why he can't do it any faster. Such a student has learned something of the foundations of his field.

Algorithm design—a systematic view

In the section above we saw a number of specific problems and specific solutions; here we will show that there is more to the field than isolated examples. Algorithm design is a coherent discipline—one needs a specific set of concepts to define a computational problem and a specific set of tools to design an optimal algorithm to solve it.

Problem dimensions—a microscopic view. The subset testing problem showed that there can be many different algorithms for solving a particular problem. In order to say which one is best in a particular application we have to know certain *dimensions* along which to measure properties of the algorithm. For example, in one application we may need a subset algorithm that must be very space-efficient and have good worst-case running time; in another context we might have a lot of available space and only require good expected running time, not caring if we infrequently must take a lot of time. We have thus identified three dimensions of a computational problem: time analysis, space analysis, and expected vs. worst-case analysis. We will now investigate these and other dimensions of computational problems.

Time and space analysis. The two most important resources in real computational systems are time (CPU cycles) and space (memory words). These are therefore the two dimensions of a problem most frequently studied. Running time was the primary subject we examined in the examples cited earlier. Most of the algorithms we discussed used very little extra space after storing the inputs and outputs—the hashing algorithm we used for subset testing was the only exception. Since in large computer systems we can have huge quantities of extra space for the asking and the paying, we have often ignored the space requirements of algorithms. With the rise in popularity of mini- and microprocessors with very small memories, however, space analysis is once again an important issue.

Model of computation. In covering the examples earlier in this article, we were able to make reference to the time and space requirements of various algorithms without reference to their implementation on any particular computer. Our intuitive notions were robust enough to lead to sophisticated algorithms that would certainly beat their naive competitors on any existing machine. But to analyze an algorithm in detail we must have a precise mathematical *model* of the machine on which the algorithm will run.

We could choose as our model a particular computer, such as an IBM 650 or a DEC PDP-10, and then ask how many microseconds of time or bits of storage a particular algorithm requires. There are two problems with this approach. First, we will probably analyze the expertise of the algorithm's implementer more than we will the algorithm's intrinsic merit; and second, once we have completed an analysis using an IBM 650 we still know very little about the algorithm's behavior on a PDP-10. One way of dealing with these difficulties is to invent a representative computer and then compare the performances of competing algorithms on that machine. Knuth[22] has described one such machine which he named the MIX computer. It has much in common with most existing machines without many idiosyncrasies of its own. If algorithm A is faster than algorithm B when implemented on MIX, then it is likely to be faster on most real machines, too.

Another solution is not to analyze the implementation of the algorithm on any particular machine at all, but simply to count the number of times the algorithm performs some *critical operation*. For the analysis of the FFT and matrix multiplication we chose to count the number of arithmetic operations. We know that the FFT uses exactly $n \log_2 n$ multiplications. To estimate its running time for a given implementation we can look up the execution speeds of the instructions around the multiplication instruction, sum those, and then multiply by $n \log_2 n$ to get an estimate for the running time. It is usually easy to determine the running time of a particular program if we know the number of times the critical operation is to be performed.* Once we have chosen a critical operation to count it is very easy to specify a model of computation. To count arithmetic operations we usually employ the "straight-line program" model in which an algorithm for a particular value of the problem size (n) is represented by a sequence of statements of the form

$$X_i \leftarrow X_j \ \text{OP} \ X_k$$

where OP is add, subtract, multiply, or divide. If the sequence for a particular value of n is m instructions long then we say that the execution time of our program is $T(n) = m$. If our critical operation were comparison, then we would probably choose the "decision tree" model. These and other models are described by Aho, Hopcroft, and Ullman.[23]

The above models allow us to analyze algorithms for their suitability as "in-core" programs on single-processor machines. If a program has very little main memory available and must store most of its data on tape, then some tape-oriented model such as the Turing machine is the most accurate model of the computation. If a program is to be run on a multiprocessor machine then our model must express this fact, and the model employed will vary with the multiprocessor architecture.* Many other models of computation have been proposed to describe diverse computing devices. The two important things in choosing a model are that it be *realistic*, so the results will apply to the situation it purports to model, and that it be mathematically *tractable*, so we can derive those results.

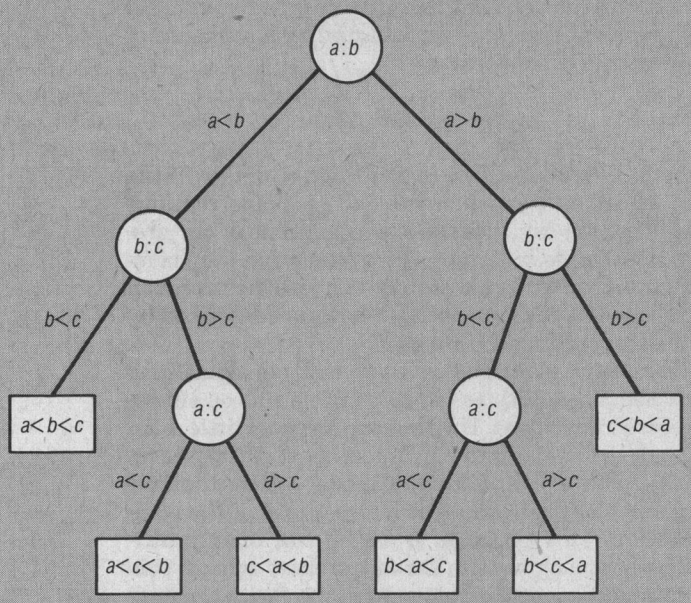
Exact or approximate analysis. Once we have chosen a model of computation we can analyze an algorithm's performance by counting the resources (time or space) it uses as a function of n, the problem size. How accurately should we do that counting? We could be very precise, calculating the answer exactly, or we might settle for an approximate answer. There are levels of approximation, all the way from the first two terms of the answer to rough upper and lower bounds. It is certainly desirable to get the exact answer, but this is sometimes very difficult. The first one or two terms of the cost function are adequate for most purposes, and in many cases we need only the function's asymptotic growth rate. We saw in the subset testing problem an example in which the run time for one program was 138 hours while another program took just 4 seconds to carry out the same task. Even if our analysis had missed a factor of ten, that could not affect our choice for large problems.

We often use the "big-oh" notation to describe the complexity of a problem. No matter what the respective constants are, an $O(n \log_2 n)$ algorithm will be faster than an $O(n^2)$ algorithm for large enough n. As we solve larger and larger problems by computer, we are more and more frequently in the domain of "large enough n." Asymptotically fast algorithms also have another advantage. If we get a new machine one hundred times faster than our old one, using an $O(n \log^2 n)$

*We must be careful, however, not to ignore certain "bookkeeping" operations that may become critical in implementations.

*The interested reader should refer to Kung[25] for a discussion of some of these issues from an algorithmic viewpoint.

Decision trees



This figure illustrates a decision tree program to sort three numbers. The program "starts" at the top node and compares the value of *a* with the value of *b*. If *a* is the lesser it goes down the left branch of the tree and if *a* is greater it goes down the right branch (we assume for clarity that all elements are distinct). It continues until it reaches a "leaf" (leaf nodes are depicted by rectangles); when we arrive at a "leaf" we know the total ordering of *a*, *b*, and *c*. I encourage the reader to study this example to see that it correctly sorts the three elements.

Decision trees accurately model computations based on comparisons. Theoretically, we can represent any comparison-based program for sorting n numbers with a corresponding decision tree. For example, we can "translate" the above tree into a set of nested IF statements in a high-level programming language. We can use the mathematical properties of decision trees to prove lower bounds on the complexity of certain computations. Notice that the height of the decision tree (i.e., the maximum distance from the root to one of the leaves) corresponds to the maximum number of comparisons ever used by the "decision tree program." It is a well-known mathematical fact that if a binary tree has m leaves, it must have height of at least $\log_2 m$. It is also known that there are $n!$ (i.e., $n*(n-1)*\ldots*1$) different permutations of n elements. Since a sorting program must correctly distinguish between all permutations of its input variables, its decision tree must have at least $n!$ leaves. Combining these two facts, we see that any decision tree for sorting must use at least $\log_2 n!$ comparisons, which is approximately $n \log_2 n$. We thus see that the Mergesort algorithm for sorting is very close to the best possible algorithm in terms of number of comparisons used.

We must be very careful in interpreting lower bounds—they only apply in the given model of computation. It might be possible to sort numbers faster if we use more powerful operations than comparisons. (In fact, it is!) The lower bound, however, still serves a useful purpose. It tells us that if we want to do something faster, we had better adopt a *fundamentally different approach* to solving the problem.

algorithm will allow us to solve a problem almost one hundred times larger in the same period of time. Using an $O(n^2)$ algorithm will permit the new machine to solve a problem only ten times larger in that time. Thus, a function's asymptotic growth rate is usually enough to tell us how much an increase in problem size will cost under each type of algorithm.

Average or worst-case analysis. Many algorithms perform a sequence of operations independent of their input data; the FFT and matrix multiplication algorithms are both data-independent. The analysis of a data-independent algorithm is straightforward—we simply count the number of operations used. The operation of algorithms such as those for sorting and substring searching, however, are dependent on their input data. One algorithm can have very different running times for two inputs of the same size. How do we describe the running time of such an algorithm? Pessimists want to know the worst-case running time over all inputs and realists want to know the average running time. (We are rarely concerned with the best-case running time, for there are very few optimists in computing.)

Most mathematical analysis of algorithms has been done for the worst case. Even in data-dependent algorithms it is usually easy to identify the worst possible occurrence and analyze it as in a data-independent algorithm. In certain applications (air traffic control is often cited) it is important to have an algorithm that never surprises us with a very slow case. For most applications, however, we are more interested in what will usually happen; expected-time analysis provides this information. Relatively little work has been done on expected-time analysis. The two major stumbling blocks are in the choice of a realistic and tractable probability model of the inputs and in the intrinsic difficulty of dealing with expectations instead of single cases. It would be desirable to have a single algorithm efficient in both expected and worst-case performances.

Upper and lower bounds. Most naive sorting algorithms (such as "bubble sort") require $O(n^2)$ comparisons in the worst case. Earlier we investigated Mergesort, which never uses more than $O(n \log_2 n)$ comparisons. Should we continue our search, hoping to find an algorithm that uses perhaps only $O(n)$ comparisons? The answer to this question is no, for we can show that *every sorting algorithm must take at least* $O(n \log_2 n)$ *comparisons in the worst case.* The proof of this theorem uses the "decision tree" model of computation, described nicely by Aho, Hopcroft, and Ullman.[24] The Mergesort algorithm gave us an *upper bound* of $O(n \log_2 n)$ on the complexity of sorting; this theorem gives us a *lower bound.* Since the two have the same growth rate, we can say that Mergesort is *optimal* to within a constant factor under the decision tree model of computation. Notice that we have now made the important jump from speaking of the complexity of an algorithm to speaking of the complexity of a problem.

Lower bound results are usually much more difficult to obtain than upper bounds. To find an upper

bound we need only give a particular algorithm and analyze it. For a lower bound, however, we must show that in the set of algorithms for solving the problem there are none more efficient than the lower bound. There are some trivial lower bounds we can easily achieve: most problems require examination of all their inputs, so we usually have an easy lower bound of the input size. The number of nontrivial lower bounds discovered to date is very small.

A precise model of computation is important in proving lower bounds. Previously we gave three algorithms we can use for testing set equality:* brute force, sorting, and hashing. The importance of the computational model becomes clear when we learn that we can prove each of those algorithms optimal under different computational models! Brute force's $O(n^2)$ performance is optimal if only equal/not-equal comparisons can be made between elements of the two sets. If the model of computation includes only less-than/not-less-than comparisons, sorting's $O(n \log_2 n)$ comparisons are optimal. If the model is a random access computer such as MIX, hashing's average-case linear performance is provably best.

Exact and approximation algorithms. The best-known algorithms for many problems are quite slow, requiring, say, $O(2^n)$ time. A very few of these problems have actually been proved to have exponential lower bounds. Others belong to a fascinating class called the NP-complete problems, either all solvable in time polynomial in problem size, or all of exponential complexity. Unfortunately, nobody yet knows which, but most of the money is on exponential. Examples of NP-complete problems include the traveling salesman problem (finding a minimal-length tour through a set of cities), bin packing, and the knapsack problem. Literally hundreds of problems are known to be NP-complete. Lewis and Papidimitriou[26] provide an excellent introduction to this class. There are other problems, not proven to be hard, for which no one has yet been able to design fast algorithms. So when we have a problem we don't know how to solve efficiently, what can we do?

The answer is amazingly simple: don't solve it. Solve a related problem instead. Instead of designing an algorithm to produce the exact answer, we can build one to produce an approximation to the exact answer. So instead of finding a minimal tour for the traveling salesman, we might provide him with a tour we know is no more than 50 percent longer than the true minimum. Or if someone asks if a number is prime or composite, instead of providing the true answer we might respond "I don't know, but I'm 99.999999 percent sure that it's prime." Problems abound for which the best known exact algorithms require exponential time, but for which we can quickly find approximate solutions. Garey and Johnson[27] examine these issues.

These problem dimensions—such as time analysis, space analysis, and expected vs. worst-case analysis—are the categories in which algorithm designers think. When someone brings a designer a problem, his first task is to understand the abstract problem. Next he must understand what kind of solution the person wants, and he uses the dimensions to describe the desired solution. Using the vocabulary introduced here it is easy to describe concepts such as a "fast expected time and low worst-case storage approximation algorithm for task X, to be run on a multiprocessor machine." There are other infrequently used dimensions we have not covered, such as code complexity—how long is the shortest program to solve a problem? But the dimensions we have examined can describe most algorithmic results.

Problem areas—a macroscopic view. The vocabulary above can describe a particular algorithmic problem at a precise level of detail. Here we change our perspective and examine large classes of problems, but use the same terminology.

Ordered sets. There are many problems on sets that depend only on a "less than" relationship being defined between the elements of the set. In many cases the set contains integers or real numbers, while in other cases we define a "less than" relation between character strings (JONES is less than SMITH). The problems discussed earlier in the section on subset testing all deal with ordered sets. The algorithms in that section are appropriate if the elements of the sets are numbers, character strings, or any other type of "orderable" object. Knuth[28] provides an excellent introduction to the applications of, and algorithms for, ordered sets.

The "median problem" is another problem defined for ordered sets: given an n-element set we are to find an element which is less than half of the elements and not less than the other half. A naive algorithm would count for each element the number of elements less than it, and then report the median as the element with exactly half the others less than it. This algorithm makes approximately n^2 comparisons. We can devise an $O(n \log_2 n)$ algorithm by sorting the elements and then reporting the middle of the sorted list. In 1962 C. A. R. Hoare described an algorithm for the median problem, with linear expected time. For over ten years no one knew if there was an algorithm with linear worst-case time; Blum et al. finally offered one in 1973.[29] Much additional work has been done on this problem, exploring facets such as minimal storage, detailed worst-case vs. expected running time analysis (upper and lower bounds), and approximation algorithms.

Algebraic and numeric problems. Many aspects of algebraic and numeric problems have a discrete flavor, and discrete algorithm design can play a significant role in such problems. Matrix multiplication is perhaps the clearest example: we can describe (and appreciate) the fast algorithm without reference to any of its numeric properties. We can also view the FFT "non-numerically." Another numeric problem

*We gave them originally for subset testing, but recall that two sets are equal if and only if each is a subset of the other.

that can assume a purely discrete character is sparse matrix manipulation (in a sparse matrix almost all elements are zero). Borodin and Munro, cited earlier, describe applications of discrete algorithm design to numeric problems such as polynomial manipulation, extended precision arithmetic, and multiprocessor implementations of numeric problems.

Graphs. Graphs can represent many different kinds of relations, from the interconnections of an airline system to the configuration of a computer system. Tarjan's survey[30] discusses many computational problems relating to graphs. One important problem involves determining if we can imbed a given graph in a plane without any edges crossing. We might use this to check if we can imbed the connections of a given circuit on a printed circuit board or integrated circuit. The first algorithms for testing planarity ran in $O(n^3)$ time on n-node graphs; after much effort by many researchers, Hopcroft and Tarjan finally designed a linear-time planarity algorithm in 1974.[31] Another graph problem is constructing a minimal spanning tree of a weighted graph, i.e., a minimal-weight set of edges connecting all nodes. Researchers have proposed and analyzed a wide variety of algorithms for this problem. Some are superior for very dense graphs, others for relatively sparse graphs, and still others for planar graphs. Efficient graph algorithms exist for problems such as computer program flow analysis and finding maximal flows in networks. And since a sparse matrix is usually represented by a graph, graph algorithms can apply to sparse matrix problems.

Geometry. Shamos' paper[32] is an outstanding introduction to computational geometry, the field concerned with developing optimal algorithms for geometric problems. Many application areas are inherently geometric (such as circuit layout) and other problems can be viewed geometrically (such as treating a set of multivariate observations as points in a multidimensional space). Shamos describes an important structure, the Voronoi diagram, which allows many geometric problems dealing with n points in a plane to be solved in $O(n \log_2 n)$ time. Among these are determining the nearest neighbor of every point, and constructing the minimal spanning tree of the point sets, both important tasks in many data analysis procedures and previously requiring $O(n^2)$ time. Many other important problems have been solved after being cast in a geometric framework. One result, for example, revealed that the standard simplex method of linear programming is not optimal for two- and three-variable programs with n constraints. The simplex method has worst-case running times of $O(n^2)$ and $O(n^3)$, respectively; an $O(n \log_2 n)$ method has proved optimal for both the two- and three-variable case.

Other areas. We have glimpsed a few fields studied by algorithm designers. The results in many other areas must go unmentioned. These include algorithms for problems in compilation, operations research, data base management, statistics, and character string handling. These results have led both to fast algorithms and to a new *algorithmic* understanding of the various fields.

The "building blocks" of algorithm design. Wandering through a computer room we cannot help but be impressed by the complexity of a large-scale computing system. The novice might find it hard to believe that the human mind could design anything so complicated. Although he is not too far from the truth, many undergraduates are able to understand the basics of computer organization after only one or two semesters. They are able to comprehend the complexity not by sheer force of concentration, but rather by understanding the "building blocks" from which computers are made. A similar experience awaits the novice algorithm designer. The algorithms surveyed above deal with many problem areas, but are rather simple to comprehend once we understand the "building blocks" of algorithm design. Here we will describe three important classes of these "blocks."

Data structures. Algorithms deal with data, and the algorithm designer uses *data structures* as tools to organize data. Earlier we saw simple data structures such as arrays and matrices, and a fairly complex data structure, the hash table. There are more exotic types, such as linked lists, stacks, queues, priority queues, and trees. Each provides an appropriate way to structure data for a particular task. Tarjan[33] briefly describes many of these structures; Knuth[34] provides detailed descriptions of a large number of interesting structures.

Algorithmic techniques. Structured programming demands that a programmer express a complicated sequence of commands as a series of refinements by which the program can be understood at different levels. In each of these refinements the programmer applies a basic, well understood method to a well defined problem. Good programmers used this technique long before it was verbalized; good algorithm designers use a similar strategy even though they infrequently discuss it. The constructs available to the algorithm designer are similar to those available in structured programming, though somewhat more powerful. We will describe some of these constructs very briefly; more detail can be found in Tarjan[35] and Aho, Hopcroft, and Ullman.[36]

We have already seen many common algorithmic constructs. Most of the algorithms we described use *iteration* in one form or another—this says "do x over and over until the task is accomplished." Iteration is present in almost all programming languages as DO and WHILE loops. A more powerful construct is *recursion*, giving us a way to express recursive problem solving in programming languages. To define a recursive solution to a problem we say (essentially) "to solve a problem of a certain size, solve the same problem of a smaller size." We used recursion to describe a binary search: to binary search a table of size n we binary searched a table of size $n/2$. A particular application of recursion, *divide-and-conquer*, says "to solve a problem of size n, first divide it into sub-

problems each of size only a fraction of n, then solve those subproblems recursively, and finally combine the subsolutions to yield a solution to the original problem." We can view Mergesort as a textbook example of divide-and-conquer: to sort a list of n elements we first break the list into two sublists each of $n/2$ elements, then sort those recursively, and finally merge those together. The FFT and the $O(n^{2.81})$ matrix multiplication algorithms are other applications of the divide-and-conquer technique. Once we understand the fundamental principles of divide-and-conquer algorithms, we can easily grasp each of these instances.

Researchers have identified and studied many other algorithmic techniques. *Dynamic programming*, a technique from operations research, has found many applications in algorithm design. Search strategies such as *breadth-first search* and *depth-first search* have yielded efficient graph algorithms. *Transformation* allows us to turn an instance of one problem into another; we saw earlier that there are many transformations to turn almost totally unrelated problems into instances of matrix multiplication. Perhaps the single most important algorithmic technique is the use of optimal tools to solve the subproblems we create for ourselves in designing a new algorithm.

Proof techniques. Once an algorithm designer has devised an algorithm and "knows in his heart" that it has certain properties, he must prove that it does. (Perhaps this separates practitioner from theorist.) His must first prove that his algorithm indeed computes what it purports to; he will use many of the tools of program verification here. Next he must analyze his algorithm's resource requirements, using many different mathematical tools. Finally he must prove his algorithm optimal by giving a lower bound proof. Weide[37] discusses the different methods of analysis used in these steps.

Current directions

Algorithm design has grown rapidly in the past decade. Essentially unknown as a field ten years ago, it is now one of theoretical computer science's most active areas and sees widespread use in applications. Continued emphasis on performance ensures even more algorithm research. Here we will examine some of the directions in which the field is moving.

One constant trend has been from "toy" problems to "real" problems. This involves many detailed analyses and expected-resource analyses, for in applications we are often seriously concerned about 20-percent differences in average running time. Much recent work has centered on approximation algorithms, since many applications do not require exact answers. On the theoretical side, the outstanding question is the complexity of the NP-complete problems—are they exponential or not? Another important theoretical problem is the search for some underlying theory of algorithm design; we still have no theoretical explanation for what makes a class of problems easy or hard. Tarjan has mentioned the need for a "calculus of data structures"—a set of rules leading to the best possible structure for a given situation.

An outgrowth of this work will be the development of "algorithm engineering." This field will supply the programmer with tools similar to those electrical engineering gives the circuit designer. Before algorithm design turns into algorithm engineering we will need to develop many more specific algorithms and provide a theoretical base for the field. We will know that the field has become an engineering discipline as soon as theoretical computer scientists assert that designing algorithms is no longer bona fide research because "it's such a well-understood process."

Algorithm design offers much to individuals involved in computing-related activities. The mathematician and theoretical computer scientist can view the field as a rich source of problems needing precise mathematical treatment. These problems are mathematically fascinating and call for some of the most powerful tools of discrete mathematics. The applications programmer with little interest in elegant theorems can also benefit, since proper algorithm design occasionally yields significant financial savings. But even placing aside benefits to specific prac-

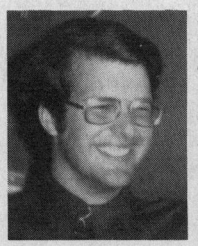
titioners, I feel that anyone involved with computing, regardless of his position in the practical-to-theoretical continuum, should have some familiarity with the field. The study of algorithms is the study of the very heart of computing, and it provides us with a new way of thinking about our computational problems. ■

Acknowledgments

This paper has profited from the helpful criticism of many people; the comments of David Jefferson, Len Shustek, and Bruce Weide have been particularly helpful. The research described here was supported in part by the Office of Naval Research, under Contract N000014-76-C-0370.

References

1. J. E. Hopcroft, "Complexity of Computer Computations," *Proc. IFIP Congress 74,* Vol. 3, 1974, pp. 620-626.
2. R. E. Tarjan, "Complexity of Combinatorial Algorithms," *SIAM Review,* Vol. 20, No. 3, July 1978, pp. 457-491.
3. B. Weide, "A Survey of Analysis Techniques for Discrete Algorithms," *Computing Surveys*, Vol. 9, No. 4, Dec. 1977, pp. 291-313.
4. D. E. Knuth, "Mathematical Analysis of Algorithms," *IFIP Congress 71*, Vol. 1, 1971, pp. 135-143.
5. D. E. Knuth, "Algorithms," *Scientific American*, Vol. 236, No. 4, Apr. 1977, pp. 63-80.
6. A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *The Design and Analysis of Computer Algorithms,* Addison-Wesley, Reading, Mass., 1974.
7. D. E. Knuth, *The Art of Computer Programming, Volume 1: Fundamental Algorithms,* Addison-Wesley, Reading, Mass., 1968.
8. D. E. Knuth, *The Art of Computer Programming, Volume 2: Seminumerical Algorithms,* Addison-Wesley, Reading, Mass., 1969.
9. D. E. Knuth, *The Art of Computer Programming, Volume 3: Sorting and Searching,* Addison-Wesley, Reading, Mass., 1973.
10. Knuth, *Art of Computer Programming, Volume 3,* p. 391.
11. Knuth, *Scientific American*, Vol. 236, No. 4, Apr. 1977, p. 63ff.
12. D. E. Knuth, J. H. Morris, and V. R. Pratt, "Fast Pattern Matching in Strings," *SIAM Journal of Computing*, Vol. 6, No. 2, June 1977, pp. 323-350.
13. R. S. Boyer and J. S. Moore, "A Fast String Searching Algorithm," *CACM*, Vol. 20, No. 10, Oct. 1977, pp. 762-772.
14. J. M. Cooley and J. W. Tukey, "An Algorithm for the Machine Calculation of Complex Fourier Series," *Mathematics of Computation,* Vol. 19, 1965, pp. 297-301.
15. Aho, Hopcroft, and Ullman, *Design and Analysis. . .*, pp. 252-276.
16. A. Borodin and I. Munro, *The Computational Complexity of Algebraic and Numeric Problems,* Elsevier-North Holland Pub. Co., N.Y., Theory of Computation Series No. 1, 1975.
17. D. R. Brillinger, *Time Series: Data Analysis and Theory,* Holt, Rinehart, and Winston, N.Y., 1975.
18. V. Strassen, "Gaussian Elimination is Not Optimal," *Numerische Mathematik*, Vol. 13, 1969, pp. 345-356.
19. W. Diffie and M. Hellman, "New Directions in Cryptography," *IEEE Trans. Information Theory*, Vol. IT-22, No. 6, Nov. 1976, pp. 644-654.
20. R. L. Rivest, A. Shamir, and L. Adleman, "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems," *CACM*, Vol. 21, No. 3, Feb. 1978, pp. 120-126.
21. M. Gardner, "A New Kind of Cipher That Would Take Millions of Years to Break" (Mathematical Games), *Scientific American*, Vol. 236, No. 8, Aug. 1977, pp. 120-125.
22. Knuth, *Art of Computer Programming, Volume 1,* pp. 120-160.
23. Aho, Hopcroft, and Ullman, *Design and Analysis. . .*, pp. 2-41.
24. *Ibid.*, pp. 24-25.
25. H. T. Kung, "Synchronized and Asynchronous Parallel Algorithms for Multiprocessors," in *Algorithms and Complexity: New Directions and Recent Results,* J. F. Traub, ed., Academic Press, N.Y., 1976, pp. 153-200.
26. H. R. Lewis and C. H. Papidimitriou, "The Efficiency of Algorithms," *Scientific American,* Vol. 238, No. 1, Jan. 1978, pp. 97-109.
27. M. R. Garey and D. S. Johnson, "Approximation Algorithms for Combinatorial Problems: An Annotated Bibliography," in *Algorithms and Complexity: New Directions and Recent Results,* J. F. Traub, ed., Academic Press, N.Y., 1976, pp. 41-52.
28. Knuth, *Art of Computer Programming, Volume 3.*
29. M. Blum, R. Floyd, V. Pratt, R. Rivest, and R. Tarjan, "Time Bounds for Selection," *Journal of Computer and Systems Sciences,* Vol. 7, No. 4, Aug. 1973, pp. 448-461.
30. Tarjan, *SIAM Review,* Vol. 20, No. 3.
31. J. Hopcroft and R. Tarjan, "Efficient Planarity Testing," *JACM*, Vol. 21, No. 4, Oct. 1974, pp. 549-568.
32. M. I. Shamos, "Geometric Complexity," *Proc. Seventh ACM Symposium on the Theory of Computing,* 1975, pp. 224-233.
33. Tarjan, *SIAM Review,* Vol. 20, No. 3.
34. Knuth, *Art of Computer Programming, Volume 1.*
35. Tarjan, *SIAM Review,* Vol. 20, No. 3.
36. Aho, Hopcroft, and Ullman, *Design and Analysis. . .*, pp. 44-74.
37. Weide, *Computing Surveys,* Vol. 9, No. 4, pp. 291-313.



Jon Louis Bentley is an assistant professor of computer science and mathematics at Carnegie-Mellon University, Pittsburgh, Pennsylvania. He worked as a research intern at the Xerox Palo Alto Research Center from 1973 to 1974. During the summer of 1975 he was a visiting scholar at the Stanford Linear Accelerator Center. From 1975 to 1976 he was a National Science Foundation graduate fellow. His current research interests include the design and analysis of computer algorithms (especially for geometrical and statistical problems) and the mathematical foundations of computation.

A member of the ACM and Sigma Xi, Bentley received the BS in mathematical sciences from Stanford University in 1974, and the MS and PhD in computer science from the University of North Carolina in 1976.