CMU-CS-78-121

AN INTRODUCTION TO ALGORITHM DESIGN

Jon Louis Bentley
Departments of Computer Science and Mathematics
Carnegie-Mellon University
Pittsburgh, Pennsylvania 15213

ABSTRACT

The field of algorithm design is concerned with the development of efficient methods for solving computational problems. Although the field traces its roots to theoretical computer science, recent algorithmic advances have drastically reduced the costs of real computations. For this reason it is important that anyone involved with computing have at least a cursory knowledge of the area. This paper surveys the field of algorithm design in two ways: first by the study of a few problems in detail, and then by a systematic view of the field. The orientation of this paper is towards the practitioner of computing (in either software or hardware); the goals of the paper are to provide both an understanding of the field and a feeling for "what it can do for me".

An earlier version of this paper was given as an invited paper at the Computer Science and Statistics: Eleventh Annual Symposium on the Interface, and appears in the proceedings thereof.

This research was supported in part by the Office of Naval Research under contract N00014-76-C-0370

1. INTRODUCTION

"Algorithm design--that's the field where people *talk* about programs and *prove theorems* about programs instead of *writing and debugging* programs." Statements along those 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." In this paper we will investigate the field of algorithm design (which also is known as "Analysis of Algorithms" and "Concrete Computational Complexity", among other names) and better equip the reader to judge the field for himself.

The author trusts that anyone who has even the slightest love for mathematics burning somewhere inside his heart (however deeply), will continue to read this paper to see how mathematical tools can be applied to the problems of programming. But for the rest of the readers (whose interest in mathematics was probably squelched in freshman calculus) I would like to offer the same bait that drew me into this field. I can 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 had estimated (by counting the turning rate of the tapes) 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 all of the time was spent in scanning a one thousand 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 spent almost all of its time waiting for the tape!). Around that same time I was asked to help another programmer who had already spent one month of time and produced over a thousand cards of code for a particular program. A simple change in data structure and a few day's work (starting over from scratch) allowed us to redo the program in less than two hundred lines of code. The resulting 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 can sometimes be rewarding enough!

Throughout this paper we will refer only to the *discrete* aspects of algorithm design. We will not mention numeric problems such as stability, truncation error, error propagation and other issues that are in the domain of numerical analysts. Even with this restriction, we still 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 the field of discrete algorithm design have appeared recently. Hopcroft [1974] and Tarjan [1977] both give a broad and thorough picture of the field. Weide's [1977] survey concentrates on the techniques used for analyzing discrete algorithms, and accomplishes that task expertly. For those who are skeptical of sweeping surveys and prefer to see a couple of problems examined in detail, Knuth's introductions [1971, 1977] will prove enlightening and fascinating. And if one is ready to become a serious student of the field, the standard texts are provided by Aho, Hopcroft and Ullman [1974] (a one-semester, graduate level introduction) and Knuth [1968, 1969, 1973] who has

completed three volumes of his seven volume *definitive* work on computer algorithms. This paper attempts to supplement those works by providing a broad survey for the novice. The bibliography has been kept exceptionally short; both Tarjan [1977] and Weide [1977] contain excellent bibliographies for those interested.

This paper is divided into five sections. In Section 2 we will examine five problems and some algorithms for solving them. Having examined those concrete examples we turn to a systematic view of the field in Section 3. In Section 4 we will mention some of the current directions in which the field is now moving. Finally, we tie together the main points of this paper in Section 5.

2. EXAMPLES OF FAST ALGORITHMS

Sweeping generalizations without supporting examples are often content-free, so before we go on to sweeping generalizations in Section 3 we will study a few examples of fast algorithms. For each example we will specify a problem, mention some of its real-world applications, give an algorithm to solve the problem, analyze the efficiency of the algorithm, and then discuss interesting issues which have surfaced. We will study the "subset testing" problem of Section 2.1 in a fair amount of depth and then treat the other four problems at a more superficial level. After discussing these examples, and before we move on to the statements about the field of algorithm design in Section 3, we will summarize what all of our work bought us in Section 2.6.

But first a word on why we are examining these particular problems. The subset testing problem of Section 2.1 will raise a number of familiar issues and should cover some old ground for many; it also gives us a nice illustration of the tremendous time savings achievable with proper algorithms. The substring searching problem of Section 2.2 provides an extremely interesting blend of theory and practice. The Fast Fourier Transform of Section 2.3 is known to many, uses some important algorithmic techniques, and is eminently practical. In Section 2.4 we examine a very old problem (matrix multiplication) and a recent and remarkably counter-intuitive solution; we will also see some mysterious relations among very dissimilar problems. In Section 2.5 we will investigate algorithmic aspects of a public-key cryptosystem that has recently revolutionized the world of cryptography, and promises to have a substantial impact on "secure" computing.

2.1 Subset Testing

Given a set A (of size n) and a set B (of size $m \leq n$), is B a subset of A?[1] This "subset testing" problem can be stated as a programming exercise: given an array A[1:n] and B[1:m],

1 This problem is discussed by Knuth [1973, p. 39[].

both of (say) 32-bit words, is every word in B also in A? Disguised versions of this problem arise in many contexts: A could be an employee master file, B a list of weekly transactions, and we want to find whether a master-file record exists for each weekly transaction. Or A might be a table of real numbers x and have an associated table S which contains sine x, then B would be a set of x values at which the sine function is to be evaluated. Although this problem does have some practical application, that is not our main motivation for examining it here. We will see that it leads to many of the basic issues in sorting and searching, and points to inter-relationships between those problems. We will also get an exposure to some of the common methods of algorithm design.

We will examine three ways of solving this problem. In order to compare the methods we will find the running time of each by counting the number of comparisons between elements. The following enticement might encourage the reader as he labors through the different methods: following our discussion of the methods we will see an application in which our first algorihm would require over six days of CPU time, while our final algorithm can 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 it 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[2]. 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, insuring 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

2 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.

scan will take about m+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 $(n \log_2 n) + (m \log_2 m) + m+n$ 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 (the lists could either be stored as arrays or linked together 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 2-element sorted lists. The next step is to merge adjacent pairs of those lists giving n/4 4-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 $n(\log_2 n + 1) + m(\log_2 m + 1)$ 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

Introspection as to how we would solve the phone book problem led 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"[3] 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 $n \log_2 n + m \log_2 m$ 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 searching strategy even more appropriate for this problem: hashing. Using hashing we can store an element in a table or check to see if an element is already in a table in

3 A binary search for a name in a phone book first compares that name to the middle name in the book. If that name is less than the middle we restrict our search to the first half of the book, otherwise we search the last half, and so on.

about two comparisons, on the average[4]. 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* which is an array of length (1.5)n.[5] We then store the elements of A in the table one-by-one by the use of a *hash function*. This function maps a data value into an integer in the bounds of the hash table. If that position in the hash table is empty, fine: insert the element. If the position was 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 has shown that a proper collision resolution strategy allows one 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. The technique of 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.

Summary

The subset testing problem is stated very simply but has 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 is 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).[6] The approaches that we used to solve these problems are some of the fundamental tools of algorithm designers. We have also touched on a number of interesting aspects of algorithmic problems such as time and space analyses and worst-case versus expected-time analysis. We will study these issues further in Section 3.

But what has all this gained us? We certainly have a more definite understanding of some of the fundamental computational issues involved, but does it make any difference in practice? To answer this question let's assume that we are writing a program for subset testing where A and B both contain one million elements, and for the sake of argument assume that one comparison takes one microsecond of computer time. By 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

---

4 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.

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

6 A more thorough examination of searching is contained in Knuth's [1977] survey.

will yield 4 seconds. Although we haven't calculated all the costs of implementation, this example shows how sometimes a simple analysis is all one needs to make an informed choice.

2.2 Substring Searching

Does a given *string* contain a specified substring *pattern*, and if so, where? This is the substring searching problem. This problem is familiar to most who have used computer text editors; as the author sat down to type this paragraph he told the editor to find the substring "2.2" in his text file so he would know where to insert this text![7] This same operation is used by information retrieval systems 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 the *string*. The worst-case behavior of this algorithm is very slow--for each of the 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 give an algorithm that will always do better?

Knuth, Morris and Pratt [1977] give 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* by their algorithm 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 string in position i, 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 [1977] recently used the basic idea of the Knuth, Morris and Pratt algorithm to give an even faster method of substring searching. Their method has the same worst-case performance (proportional to m+n), but is somewhat 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.

---

7 The text editor he uses looks at his file as one long string of text, sprinkled with special characters representing "carriage return".

The history of the substring searching problem provides an interesting insight into the relation of theory and practice in Computer Science. Knuth relates that he was led to his discovery of the algorithm by the use of a machine from automata theory called the "two-way deterministic pushdown automaton". The easiest way to understand the 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 talk about such diverse ideas as abstract automata and PDP-10 instructions, with a lot of combinatorial analysis in between!

2.3 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 transforming 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 calls for applying 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 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 [1965] 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 of Section 2.1. There are many different expositions of the algorithm; see Aho, Hopcroft and Ullman [1974] or Borodin and Munro [1975]. (It is interesting to note that in addition to being faster to compute, 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. The FFT has been used in diverse fields to find hidden periodicities of a stationary time series. In signal processing it is used in filters to remove noise from signals and eradicate blurring in digital pictures. It is used in numerical analysis 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 [1975, Section 1.5]. Many special-purpose processors have been built which implement this algorithm; some of those are multiprocessors which operate in parallel. The FFT is also widely used in the design of "discrete" algorithms. It is the primary tool 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 one-thousand bit integers; we will see an application of this problem in Section 2.5).

2.4 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 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 8 multiplications and 4 additions. It is fairly counter-intuitive to learn that the product can be computed using only 7 multiplications at the cost of an increase to 15 additions. But if that is counter-intuitive, then it is absolutely mind-boggling to find that this fact alone allows us to construct an algorithm for multiplying n x n matrices that runs in less than n^3 time! This algorithm is due to Strassen [1969] and works by decomposing each n x n matrix into four (n/2) x (n/2) matrices. To find the product of the original matrices it does seven multiplications of (n/2) x (n/2) matrices and then fifteen additions on matrices of that size. 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 x n matrices, then T(n) satisfies the recurrence

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

which has the solution $T(n) = O(n^{2.81})$ (where 2.81 is an approximation to $\log_2 7$). Using the naive implementation of this algorithm proves less efficient than the high school algorithm until n is in the thousands; recent work, however, has shown 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 reign. 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, solving systems of linear equations, and calculating determinants. A number of problems which seemed to be totally unrelated to matrices were phrased in that language and $O(n^{2.81})$ 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!

2.5 Public-Key Cryptography

Communications systems which deal with the problem of transmitting a *message* from

a *sender* to a *receiver* across an insecure (possibly bugged) channel while protecting the privacy of the message are known as *cryptosystems*. Such coding problems arise often in military applications, and they promise to play an ever increasing role in computer systems such as electronic mail and electronic banking, among others. A cryptosystem is usually implemented by *encoding* and *decoding* algorithms which transform their inputs according to the *keys* they are given. To send person X message M we use the encoding algorithm and key to produce message M' and transmit M' across the (insecure) channel to X. When X receives M' he can use the decoding algorithm and key to determine M, and any "eavesdropper" on the line will be left with only M'. One difficulty with this system is that the appropriate keys must somehow be given to the various parties, and this must usually be accomplished by the use of expensive secure channels such as human couriers.

An alternative to such a system was recently invented by Hellman and Diffie and is called a *public-key* cryptosystem. In a public-key system each person has an encoding key and a decoding key, as before; to send a message to person X we encode it with his encoding key, and then he can decode it with his decoding key. The novel aspect of this system is that the encoding key can be made public without revealing the corresponding decoding key. The encoding key can then be viewed as the address of person X's mail box, and anyone can put mail into that box simply by encoding it. To actually unlock the box, however, requires the decoding key, which only X possesses. Such a system solves almost all of the difficulties of previous cryptosystems, but there is one major obstacle yet to overcome: 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 encoding/decoding algorithm, but is it possible for such a function to exist?

A suitable encoding/decoding method was recently developed by Rivest, Shamir, and Adleman [1978]. Their method is based on algorithmic issues in the theory of numbers. They view messages as multiprecision (long) integers, of (for example) 200 decimal digits. 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; encoding and decoding is then performed by fast multiprecision exponentiation algorithms. The security of the system is indicated by the fact that any method that "breaks" the system must (essentially) find the factors of a very large number. Although no one knows precisely how difficult this is, mathematicians have been working on this problem for many centuries, and no one yet knows of a fast method. To 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, while the best known method for breaking the system would require *ten million centuries* of CPU time. This method thus appears to be reasonably secure against code breakers!

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", or verifications of identity. This cryptosystem is another

interesting example of the interactions between theory and practice. The system is based on number theory (perhaps the purest of the areas of pure mathematics) and complexity theory (an area of theoretical computer science), yet it promises to revolutionize the practice of cryptography. The interested reader should refer to the article by Rivest, Shamir, and Adleman [1978] or the exposition in Gardner [1977]. Although the algorithms we have mentioned did not really solve existing computational problems, they solve a problem in a totally different area by casting it in a computational light.

2.6 So What?

We have now examined five cases in which proper algorithm design has led to a sophisticated algorithm which is much faster than a naive algorithm. A lot of work has been invested in developing these algorithms; what difference will all this work make in practice?

To be honest, most of the time a fast algorithm makes no difference at all. Knuth has gathered empirical evidence which shows that most of the run time of a program is spent in just three percent of the code (a similar result is often mentioned by statisticians: twenty percent of the population accounts for eighty percent of the beer consumed). If the problem to be solved is not in the critical three percent of the code (as about 97 percent of the problems are) then it makes little difference if that algorithm is fast or not. A more complicated algorithm can often be 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 the computation being performed is indeed the bottleneck in the system flow, then an algorithm of half the running time almost doubles system throughput. In many text editors the vast majority of the time is spent in string searching; the fast algorithm of Section 2.2 can speed up many text editors by a factor of five. The author's experience with the searching program mentioned in the introduction (when the running time of a program was reduced from three hours to five minutes) is another classic example of an appropriate use for a fast algorithm. In the inner loops of many programs, proper algorithm design is critical.

An analogy will perhaps clarify these issues. It is fairly easy to walk, it is more complex to drive, and it is even more complex yet to learn to fly a modern jet airplane. Walking is the best way to get from one room of a house to another, driving is superior for getting from one town to another, and flying is hard to beat for getting from one part of the country to another. There is no "best" mode of transportation--the best mode in a particular case depends strongly on that case. For most of us the time we spend travelling in jet airplanes is very small compared to the time we spend walking--but it sure is nice to know about jets when we need them!

Although the effort of fast algorithm design only occasionally gives us large financial savings, it 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 not only knows how to search fast 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.

3. A SYSTEMATIC VIEW

In Section 2 we saw a number of specific problems and a number of specific solutions; in this section we will show that there is more to the field than isolated examples. In Section 3.1 we will discuss the concepts one needs to define a computational problem, and in Section 3.2 we will use those concepts to describe the kinds of problems for which fast algorithms have been designed. In Section 3.3 we will peek into the algorithm designer's tool bag.

3.1 Dimensions of a Problem--A Microscopic View

The subset testing problem of Section 2.1 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. In this subsection we will discuss 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) used, and these are therefore the two dimensions of a problem most frequently studied. The running time was the primary subject we examined in the examples of Section 2. Most of the algorithms we examined use very little extra space after storing the inputs and outputs; the hashing algorithm of Section 2.1 was the only exception. In large computer systems huge quantities of extra space (megawords) can be had for the asking and the paying; for that reason the space requirements of algorithms have often been ignored. With the rise in popularity of mini- and microprocessors with very small memories, however, space analysis is once again an extremely important issue.

Model of Computation

Throughout Section 2 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 will 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 be analyzing the expertise of the implementor of the algorithm more than the algorithm's intrinsic merit, and second, once we have completed such an analysis using the IBM 650 we still know very little about the algorithm's behavior on a PDP-10. One way of dealing with this difficulty is to invent a representative computer and then compare the performances of competing algorithms on that machine. Knuth [1968] has described one such machine which he named the MIX computer; it has much in common with most existing machines without many idiosyncracies of its own. If algorithm A is faster than algorithm B when implemented on MIX, then it is very likely to be faster on most real machines, too.

Another solution to the model of computation problem is not to analyze the implementation of the algorithm on any particular machine at all, but to count only the number of times some *critical operation* is performed. 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[8]. 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 [1974, Chapter 1].

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

8 Though we must be careful not to ignore certain "bookkeeping" operations that may become critical in implementations.

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 one's model must express this fact; the particular model employed will vary with the multiprocessor architecture[9]. 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 the performance of an algorithm 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 only the asymptotic growth rate of the functions is needed. We saw in the subset testing problem an example in which the run time for one program for a task was 138 hours while another program took just 4 seconds. 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 larger and larger problems are being solved 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 current, using an $O(n \log_2 n)$ 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 we will only be able to increase the problem size by a factor of ten. Thus the asymptotic growth rate of a function alone is usually enough to tell us how much an increase in problem size will cost.

Average or Worst-Case Analysis

Many algorithms perform a sequence of operations independent of their input data; the FFT and matrix multiplication algorithms of Section 2 are both data-independent. The analysis of a data-independent algorithm is straightforward--we simply count the number of operations used. The operation of other algorithms (such as the sorting and substring algorithms) are dependent on their input data; one algorithm can have very different running

9 The interested reader should refer to Kung [1976] for a discussion of some of these issues from an algorithmic viewpoint.

times for two inputs of the same size. How do we describe the running time of such an algorithm? Pessimists would like to know the worst-case of the running time over all inputs and realists would like to know the average running time. (We are rarely concerned with the best-case running time, for there are very few optimists involved with computing.)

Most of the mathematical analysis of algorithms has been done for the worst case. Even in data-dependent algorithms it usually easy to identify the worst possible occurrence, and then analyze that as in a data-independent algorithm. In certain applications (Air Traffic Control is often cited) it is very important to have an algorithm with which we are never surprised by a very slow case. For most applications, however, we are more interested in what will usually happen; expected-time analysis provides us with this information. Relatively little work has been done on expected-time analysis. The two major stumbling blocks appear to be in the choice of a realistic and tractable probability model of the inputs and the instrinsic difficulty of dealing with expectations instead of single cases. It would be very desirable to have a single algorithm that is very 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; in Section 2.1 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 it can be shown 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 and is described nicely by Aho, Hopcroft, and Ullman [1974]. 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 on a problem one need only give a particular algorithm and then analyze it. For a lower bound, however, one must show that *in the set of all algorithms for solving the problem*, there are none which are more efficient than the lower bound. There are some trivial lower bounds which can be achieved easily: 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.

In proving lower bounds it is important to be very precise about the model of computation. In Section 2.1 we gave three algorithms that can be used for testing set

equality[10]: brute force, sorting, and hashing. The importance of computational model becomes clear when we learn that each of those algorithms can be proved optimal under different computational models! The $O(n^2)$ performance of brute force is optimal if only equal/not-equal comparisons can be made between elements of the two sets. If the model of computation inludes only less-than/not-less-than comparisons then the $O(n \log_2 n)$ comparisons of sorting are optimal. If the model is a random access computer (such as MIX), then the average-case linear performance of hashing is provably best.

Exact and Approximation Algorithms

There are many problems for which the best-known algorithms 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 which are either all solvable in polynomial time 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 Travelling 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. There are other problems which have not been proved to be hard, yet no one has been able to design fast algorithms for them. When we have a problem which we do not 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, one can build an algorithm that will produce an approximation to the exact answer. So instead of finding a minimal tour for the Travelling Salesman, we might provide him with a tour which we know to be no more than fifty percent longer than the true minimum. Or if someone asks us to determine 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." Examples abound in which the best known exact algorithms for a problem require exponential time, but approximate solutions can be found very quickly. Garey and Johnson [1976] examine these issues.

Summary

These problem dimensions are the categories in which algorithm designers think. When someone brings a problem to an algorithm designer, the algorithm designer's first task is to understand the abstract problem. His second task is to understand what kind of solution the person wants, and he uses these dimensions to describe the desired solution. Using the vocabulary of this section it is easy to describe concepts such as a "fast expected time and low worst-case storage approximation algorithm for task X which is to be run on a

---

10 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.

multi-processor machine". There are other infrequently used dimensions which we have not covered (such as code complexity--how long is the shortest program to solve this problem?) but these dimensions are adequate to describe most algorithmic results.

3.2 Problem Areas--A Macroscopic View

In Section 3.1 we developed a vocabulary which we can use to describe a particular algorithmic problem at a very precise level of detail. In this section we will change our perspective and examine large classes of problems, using the terminology of the last section. We will describe each area by a brief summary and one or two illustrative problems.

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; in other cases we define a "less than" relation between character strings (JONES is less than SMITH). The problems which arose in Section 2.1 are all problems on ordered sets--these include sorting, searching, merging, and subset testing. The algorithms of that section are appropriate if the elements of the sets to be processed are numbers, character strings, or any other type of "orderable" object. Knuth [1973] 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 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. An $O(n \log_2 n)$ algorithm is given by sorting the elements and then reporting the middle of the sorted list. A median algorithm with linear expected time was first described by C. A. R. Hoare in 1962. For over ten years it was not known if there was an algorithm that had linear worst-case time; one was finally given by Blum *et al.* [1973]. Much additional work has been done on this problem, exploring such facets as minimal storage, detailed analysis of worst-case and expected running times (both 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 of such a problem; the fast algorithm can be described (and appreciated) without reference to any of its numeric properties. The FFT can also be viewed "non-numerically". Another example of a numeric problem that can assume a purely discrete character is the manipulation of sparse matrices (matrices in which almost all elements are

zero); we return to this problem in our discussion of graph problems. Borodin and Munro [1975] give many applications of the principles of discrete algorithm design to numeric problems such as polynomial manipulation, extended precision arithmetic, and multiprocessor implementations of numeric problems.

Graphs

Graphs are used to represent many different kinds of relations, from the interconnections of an airline system to the configuration of a computer system. Tarjan's [1977] survey discusses many computational problems on graphs. One important problem calls for determining if a given graph can be imbedded in the plane without any edges crossing. This might be used to check if the connections of a given circuit could be imbedded 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 on the part of many researchers had been spent on the problem, Hopcroft and Tarjan finally gave a linear-time planarity algorithm in 1974. Another graph problem is to construct the minimal spanning tree of a weighted graph, which is a minimal-weight set of edges connecting all nodes. A wide variety of algorithms have been proposed and analyzed for this problem; some are superior for very dense graphs, others for relatively sparse graphs, and still others for graphs which are planar. Efficient graph algorithms have been given for problems such as the flow analysis of computer programs and finding maximal flows in networks. A sparse matrix is usually represented by a graph; the algorithms for manipulating matrices are then graph algorithms.

Geometry

Shamos' [1975] paper is an outstanding introduction to the field of *Computational Geometry*, which is concerned with developing optimal algorithms for geometric problems. Many applications are inherently of a geometric nature (such as laying out circuits on a board) and other problems can be viewed geometrically (such as looking at a set of multivariate observations as points in a multidimensional space). Shamos has described an important structure called the Voronoi diagram which allows many geometric problems dealing with n points in the plane to be solved in $O(n \log_2 n)$ time. Among these problems are determining the nearest neighbor of every point and constructing the minimal spanning tree of the point sets (both of these are important tasks of many data analysis procedures, and previously required $O(n^2)$ time). Many other important problems have been solved after being cast in a geometric framework. One result obtained by this effort is 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 in the two problems of $O(n^2)$ and $O(n^3)$, respectively; an $O(n \log_2 n)$ method has been given and proved optimal for both the two- and three-variable case.

Other Areas

In this section we have glimpsed a few fields that have been studied by algorithm designers. The results in many other areas must go unmentioned; these include algorithms for compilers, operations research problems, data base management, statistics, and problems on character strings. These results have led to both fast algorithms for solving real problems and to a new, algorithmic, understanding of the various fields.

3.3 Fundamental Structures

Wandering through a computer room one can not help but be impressed by the complexity of a large-scale computing system, and the novice might find it hard to believe that a human mind could design anything so complicated. The novice is not too far from the truth, yet many undergraduates are able to understand the basics of the organization of a computer 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" of which computers are made. A similar experience awaits the novice algorithm designer. The algorithms mentioned in Section 3.2 deal with many problem areas, but are rather simple to comprehend once one understands the "building blocks" of algorithm design. In this section we will describe three important classes of these fundamental structures.

Data Structures

Algorithms deal with data, and data structures are the tools the algorithm designer uses to organize his data. In Section 2 we saw simple data structures such as arrays and matrices, and a fairly complex data structure, the hash table. There are many more exotic types of data structures, such as linked lists, stacks, queues, priority queues, and trees, to name a few. Each of these provides an appropriate way to structure data for a particular task. Tarjan [1977] gives a brief description of many of these structures; a detailed description of a large number of interesting structures is provided by Knuth [1968].

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 a basic, well understood method is applied to a well defined problem. Good programmers used this technique long before it was vocalized; good algorithm designers use a similar strategy even though they infrequently discuss it. The constructs available to the algorithm designer are similar to those in structured programming languages, though somewhat more powerful. We will describe some of these constructs very briefly; more detail can be found in Tarjan [1977] and Aho, Hopcroft, and Ullman [1974, Chapter 2].

We have already seen many common algorithmic techniques in Section 2. Most of the algorithms we described used *iteration* in one form or another--this strategy 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*, which gives us a way to express recursive problem solving in programming languages. To define a recursive solution to a problem one says (essentially), "to solve a problem of a certain size, solve the same problem of a smaller size." We used recursion to describe binary search: to binary search a table of size n we binary searched a table of size n/2. A particular application of recursion is usually called *divide-and-conquer*, and says, "to solve a problem of size n, 1) divide it into subproblems each of size only a fraction of n, 2) solve those subproblems recursively, and 3) combine the subsolutions to yield a solution to the original problem." Mergesort can be viewed as a textbook example of divide-and-conquer: to sort a list of n elements we 1) break the list into two sublists each of n/2 elements, 2) sort those recursively, and 3) merge those together. The Fast Fourier Transform and the $O(n^{2.81})$ matrix multiplication algorithms are other application of the divide-and-conquer technique. Once one understands the fundamental principles of divide-and-conquer algorithms, each of these instances becomes rather easy to grasp.

Many other algorithmic techniques have been identified and studied. *Dynamic programming* is a technique from operations research that has found many applications in algorithm design. Search strategies such as *breadth-first search* and *depth-first search* have been used to yield efficient graph algorithms. *Transformation* allows us to turn an instance of one problem into another; we saw in Section 2.4 that there are many transformations to turn almost totally unrelated problems into instances of matrix multiplication. Perhaps the single most important algorithmic technique is to use optimal tools to solve the subproblems we create for ourselves in designing a new algorithm. To do so an algorithm designer must keep abreast of the current results in his field.

Proof Techniques

Once an algorithm designer has given an algorithm and "knows in his heart" that it has certain properties, he must prove that it does. (Perhaps it is this step which separates practitioner from theorist.) His first task is to prove that his algorithm indeed computes what it purports to; he will use many of the tools of program verification in this step. Next he must analyze the resource requirements of his algorithm, during which he will use many different mathematical tools. Finally he can prove his algorithm optimal by giving a lower bound proof. The different methods of analysis used in these various steps are discussed by Weide [1977].

4. CURRENT DIRECTIONS

The field of algorithm design has experienced a meteoric rise in the past decade. Essentially unknown as a field ten years ago, it is now one of the most active areas in theoretical computer science and has seen widespread use in applications. Although the field has come a long way, it has much further to go. In this section we will examine some of the directions in which the field is currently moving.

One constant direction of the field 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 twenty percent differences in average running time. Along with these efforts much has been done recently on approximation algorithms, since many applications do not require exact answers. On the more 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. Though many individual results have been achieved to date, 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 that will allow us to develop the (provably) best possible structure for a given situation.

An important outgrowth of this work will be the development of "Algorithmic Engineering". This field will supply the programmer with tools similar to those Electrical Engineering gives the circuit designer. Before Algorithm Design turns into Algorithmic Engineering we will need to develop many more particular results and give a theoretical basis 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."

5. CONCLUSIONS

In this paper we have looked at the field of algorithm design from a number of different viewpoints. In Section 2 we investigated particular computational problems and their algorithmic solutions. We saw interesting techniques used to solve the problems, learned of many counter-intuitive results, and glimpsed some of the practical benefits of algorithm design. We turned from "war stories" to a systematic view of the field in Section 3. In Section 3.1 we developed a set of terms which can be used to define a computational problem, in Section 3.2 we used those terms to sketch some of the results achieved in the field to date, and in Section 3.3 we mentioned some of the tools used to achieve the results. Having looked at what has already been done in Sections 2 and 3, we turned to the dangerous task of prophecy in Section 4.

In summary I would like to describe what algorithm design has to offer to various

individuals. The mathematician and theoretical computer scientist can view the field as a rich source of problems that need precise mathematical treatment; these problems are mathematically fascinating and require the use of some of the most powerful tools of discrete mathematics. The applications programmer with little interest in beautiful theorems can also benefit from this work, for the proper application of its products can occasionally be very rewarding financially. Finally, I feel that anyone involved with computing, regardless of his position on the practical-to-theoretical continuum, should be at least somewhat familiar with this field. The study of algorithms is the study of computing, and through it we gain a fundamental understanding of what computers are all about.

ACKNOWLEDGMENTS

The presentation of this paper has profited from the helpful criticism of many people; the comments of Dave Jefferson, Len Shustek, and Bruce Weide have been particularly helpful.

REFERENCES

Aho, A. V., J. E. Hopcroft, and J. D. Ullman [1974]. *The design and analysis of computer algorithms*, Addison-Wesley, Reading, Mass.

Blum, M., R. Floyd, V. Pratt, R. Rivest, and R. Tarjan [1973]. "Time bounds for selection," *Journal of Computer and Systems Sciences 7*, 4 (August 1973), 448-461.

Borodin, A. and I. Munro [1975]. *The computational complexity of algebraic and numeric problems*, American Elsevier, N. Y.

Boyer, R. S. and J S. Moore [1977]. "A fast string searching algorithm," *Communications of the ACM 20*, 10, (October 1977), 762-772.

Brillinger, D. R. [1975]. *Time series: Data analysis and theory*, Holt, Rinehart, and Winston, N.Y.

Cooley, J. M. and J. W. Tukey [1965]. "An algorithm for the machine calculation of complex Fourier series," *Math. Comp. 19*, pp. 291-301.

Gardner, M. [1977]. "Mathematical games," *Scientific American 236*, 8 (August 1977), pp. 120-125.

Garey, M. R. and D. S. Johnson [1976]. "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., pp. 41-52.

Hopcroft, J. E. [1974]. "Complexity of computer computations," In *Proceedings IFIP Congress 74*, vol. 3, North-Holland Publishing Company, Amsterdam, The Netherlands, pp. 620-626.

Karp, R. M. [1972]. "Reducibility among combinatorial problems," In *Complexity of computer computations*, R. E. Miller and J. W. Thatcher, [Eds.], Plenum Press, N. Y., pp. 85-103.

Knuth, D. E. [1968]. *The art of computer programming, vol. 1: Fundamental algorithms*, Addison-Wesley, Reading, Mass.

Knuth, D. E. [1969]. *The art of computer programming, vol. 2: Seminumerical algorithms*, Addison-Wesley, Reading, Mass.

Knuth, D. E. [1971]. "Mathematical analysis of algorithms," in *Proceedings IFIP Congress 71*, vol. 1, North-Holland Publishing Company, Amsterdam, The Netherlands, pp. 135-143.

Knuth, D. E. [1973]. *The art of computer programming, vol. 3: Sorting and searching*, Addison-Wesley, Reading, Mass.

Knuth, D. E. [1977]. "Algorithms", *Scientific American 236*, 4, (April 1977), 63-80.

Knuth, D. E., J. H. Morris, and V. R. Pratt [1977]. "Fast pattern matching in strings," *SIAM Journal of Computing* 6, 2 (June 1977), 323-350.

Kung, H. T. [1976]. "Synchronized and asynchronous parallel algorithms for multiprocessors," in *Algorithms and complexity: New directions and recent results*, J. F. Traub, [Ed.], Academic Press, N. Y., pp. 153-200.

Rabin, M. O. [1976]. "Probabilistic Algorithms," in *Algorithms and complexity: New directions and recent results*, J. F. Traub, [Ed.], Academic Press, N. Y., pp. 21-39.

Reingold, E. M. [1972]. "Establishing lower bounds on algorithms: A survey," in *AFIPS 1972 Spring Joint Computer Conference*, vol. 40, AFIPS Press, Montvale, N. J., pp. 471-481.

Rivest, R. L., A. Shamir, and L. Adleman [1978]. "A method for obtaining digital signatures and public-key cryptosystems," *Communications of the ACM 21*, 3 (February 1978), pp. 120-126.

Shamos, M. I. [1975]. "Geometric Complexity," in *Proceedings of the Seventh ACM Symposium on the Theory of Computing*, ACM, N. Y., pp. 224-233.

Strassen, V. [1969]. "Gaussian elimination is not optimal," *Numerische Mathematik 13*, pp. 345-346.

Tarjan, R. E. [1977]. *Complexity of Combinatorial Algorithms*, Stanford Computer Science Department Report STAN-CS-77-609. To appear in *SIAM Review*.

Traub, J. F., [Ed.] [1976]. *Algorithms and complexity: New directions and recent results*, Academic Press, N. Y.

Valiant, L. G. [1975]. "General context-free recognition in less than cubic time," *Journal of Computer and Systems Sciences 10*, 2 (April 1975), 308-315.

Weide, B. [1977]. "A survey of analysis techniques for discrete algorithms," *Computing Surveys 9*, 4 (December 1977), pp. 291-313.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER CMU-CS-78-121	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) AN INTRODUCTION TO ALGORITHM DESIGN		5. TYPE OF REPORT & PERIOD COVERED Interim 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Jon Louis Bentley		8. CONTRACT OR GRANT NUMBER(s) N00014-76-C-0370
9. PERFORMING ORGANIZATION NAME AND ADDRESS Carnegie-Mellon University Department of Computer Science Pittsburgh, Pennsylvania 15213		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Office of Naval Research Arlington, VA 22217		12. REPORT DATE May 1978 13. NUMBER OF PAGES 24
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) same as above		15. SECURITY CLASS. (of this report) UNCLASSIFIED 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for Public Release; Distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

The field of algorithm design is concerned with the development of efficient methods for solving computational problems. Although the field traces its roots to theoretical computer science, recent algorithmic advances have drastically reduced the costs of real computations. For this reason it is important that anyone involved with computing have at least a cursory knowledge of the area. This paper surveys the field of algorithm design in two ways: first by the study of a few problems in detail, and then by a systematic view of the field. The orientation of this paper is towards the practitioner of computing (in either

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

software or hardware); the goals of the paper are to provide both an understanding of the field and a feeling for "what it can do for me".