A Survey of Analysis Techniques for Discrete Algorithms*

BRUCE WEIDE

Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Pennsylvania 15213

This survey includes an introduction to the concepts of problem complexity, analysis of algorithms to find bounds on complexity, average-case behavior, and approximation algorithms The major techniques used in analysis of algorithms are reviewed and examples of the use of these methods are presented. A brief explanation of the problem classes P and NP, as well as the class of NP-complete problems, is also presented.

Keywords and Phrases Analysis of algorithms, computational complexity, combinatorics, NP-complete problems, approximation algorithms.

CR Categories 5.25, 5 30, 5.39

INTRODUCTION

We shall express our darker purpose
-William Shakespeare

It has long been recognized that the study of the behavior of algorithms plays a crucial role in intelligent algorithm design. Aho, Hopcroft, and Ullman [1] begin the preface of their recent book on algorithmic design and analysis by pointing out that "the study of algorithms is at the very heart of computer science." At the foundation of every computational discipline is a collection of algorithms. After a problem—for example, understanding speech, analyzing data, or compiling a program—is analyzed at a high level and design decisions are finalized, the algorithm must be implemented on a real machine. One job of the computer scientist is to isolate and study these algorithms, which abound in graph theory, statistics, operations research, and many other areas. Hence the pervasive nature of analysis of algorithms.

Despite (or perhaps because of) many significant new results in analysis of algorithms in the past few years, there is no current survey of the mathematical techniques used in algorithmic analysis. For the reader who wants to see the gory details of the analysis of many algorithms, and is willing to and capable of supplying many more details himself, the three volumes of *The Art of Computer Programming* by Knuth [43, 44, 46] are unsurpassed for completeness. Aho, Hopcroft, and Ullman [1] is an excellent text with many examples of both design and analysis but, like Knuth, does not provide an overview of the area.

For someone who wants an overview of techniques and a review of some important results, the literature is sparse. The articles by Knuth [45], Reingold [56], and Frazer [22] provide some relief, but concentrate heavily on presenting the details of one or two example algorithms or techniques. A fine paper by Borodin [6] treats primarily the theoretical aspects of computational complexity, reviewing the definitions and properties of complexity classes for various automata. Hopcroft's survey [29] of principles of algorithm design comes closest to fitting the bill, but

* This work was supported in part by a National Science Foundation Graduate Fellowship.

CONTENTS

INTRODUCTION
1 THE NOTION OF COMPLEXITY
Bounds on Complexity
Worst-case versus Average-Case Analysis
Models of Computation
Measuring Problem Size
2 LOWER BOUNDS
Trivial Lower Bounds
Information-Theoretic Bounds and Decision Trees
Oracles
Problem Reduction
Miscellaneous Tricks
3 UPPER BOUNDS
Identifying a Worst Case
Counting Operations
Recurrences
The Brute Force Method
4 THE AVERAGE CASE
Pros and Cons of Average-Case Analysis
Some Examples of Average-Case Analysis
5 APPROXIMATION ALGORITHMS
Problem Classes and Reducibility
"Guaranteed" Approximation Algorithms
Probabilistic Behavior of Approximation Algorithms
CONCLUSIONS
APPENDIX Glossary of Problems
ACKNOWLEDGMENTS
REFERENCES

new results since it appeared are abundant, especially in the area of approximation algorithms. In order to avoid duplication, the more detailed examples given here discuss algorithms other than the ones treated in those papers. This survey is an attempt to collect some of the important techniques used in algorithmic analysis and to list some of the results produced, and thereby (albeit temporarily) help fill the gap in this area. It is designed to be primarily a survey, with tutorial comments where appropriate.

The reader is assumed to be familiar with the notions of algorithms and data structures, to have been introduced to computational models such as Turing machines, and to know something about asymptotic bounds on functions and recurrence relations. He or she should also recognize some of the important discrete (combinatorial) problems. These include sorting, searching, graph problems, discrete optimization, set manipulation, and pattern matching. The Appendix is a glossary of problems mentioned in the text, which should help in this regard. Some familiarity with the notation used by algorithmic analysts would also be helpful, but is not essential. The paper is not intended as a review of algorithm or data structure design, but as a survey of analysis techniques. The reader is urged to consult the original sources for detailed algorithms and analyses which can only be briefly mentioned here.

1. THE NOTION OF COMPLEXITY

If you wish to converse with me, define your terms.
-Voltaire

I hate definitions.
-Benjamin Disraeli

The primary definition of complexity is running time. For a given problem, "time complexity" is a function that maps problem size into the time required to solve the problem. We will not consider other measures of problem difficulty, such as the amount of space required, nor will we attempt to measure comprehensibility of algorithms or the scores of other factors which must be considered when designing an algorithm. To make even this intuitively simple quantity, time, a useful measure, we need to consider such questions as upper and lower bounds on complexity, the model of computation, and the manner of problem representation. The model of computation is important because we count its operations when measuring time. The problem representation is important because it affects the problem size. These questions are considered in the following subsections.

Bounds on Complexity

Typically, we are interested in the (inherent) complexity of computing the solution to problems in a particular class. For example, we might want to know how fast we can hope to sort a list of n items, initially in an arbitrary order, regardless of the algorithm we use. In this case, we seek a "lower bound" $L(n)$ on sorting, which is a property of the sorting problem and not of any particular algorithm. This lower bound says that no algorithm can do the job in fewer than $L(n)$ time units for arbitrary inputs, i.e., that every sorting algorithm must take at least $L(n)$ time in the worst case. Some approaches to refining lower bounds are surveyed in Section 2.

On the other hand, we might also like to know how long it would take to sort such a list using a known sorting algorithm with a worst-case input. Here, we are after an "upper bound" $U(n)$, which says that for arbitrary inputs we can always sort in time at most $U(n)$. That is, in our current state of knowledge, we need not settle for an algorithm which takes more than $U(n)$ time, because an algorithm which operates in that many steps is known. For this reason, algorithms are normally analyzed to determine their worst-case behavior, in the hope of reducing $U(n)$ even further by demonstrating that some new algorithm has worst-case performance which is better than any previous algorithm. Techniques for doing this are reviewed in Section 3.

One way of seeing the distinction between lower and upper bounds is to note that both bounds are minima over the maximum complexity of inputs of size n. However, $L(n)$ is the minimum, over *all possible* algorithms, of the maximum complexity, while $U(n)$ is the minimum, over *all known* algorithms, of the maximum complexity. In trying to prove better lower bounds, we concentrate on techniques which will allow us to increase the precision with which the minimum, over *all possible* algorithms, can be bounded. Improving an upper bound means finding an algorithm with better worst-case performance. This difference leads to the differences in techniques developed in complexity analysis.

While there are apparently two complexity functions for problems, lower and upper bounds, the ultimate goal is to make these two functions coincide. When this is done, the "optimal" algorithm will have $L(n) = U(n)$. For most of the problems we will mention, this goal is not yet realized.

Worst-Case versus Average-Case Analysis

Traditionally, the worst-case complexity has been of major theoretical interest for the reasons just cited. Occasionally it is of practical importance to know the worst-case behavior of an algorithm. For example, an air-traffic control system with good expected performance might not be considered useful if occurrence of its worst case, however unlikely, could cause an accident. Recently, however, there has been greater effort in the analysis of the behavior of algorithms "on the average," since the average is often more useful in practice. For example, the simplex algorithm for linear programming is known to require an amount of time which is an exponential function of the problem size in the worst case, but for problems encountered in practice it almost always does extremely well.

This approach has difficulties, however. For one thing, averaging over many cases complicates the analysis considerably. Further, while the average alone might be of some value, finding the distribution of solution times or even the variance is an added burden, and therefore often neglected in practice. Perhaps the biggest objection of all is that the typical assumptions regarding the probability distribution over all possible inputs (usually simple ones to make the analysis tractable) are often unrealistic. The objections will be covered in more detail in Section 4. Despite the objections, average-case analysis is important and continues to grow.

Models of Computation

Does complexity measure the number of steps on a Turing machine, or the number of seconds on an IBM370/195? We are usually not interested in either of these figures exactly, although each is a legitimate measure of complexity in certain cases. The issue at hand is the "model of computation." Turing machine complexity is sometimes important, as we will see in Section 5 when examining the problem classes P and NP. A highly sophisticated machine like the IBM370/195 is probably a bad choice because it makes the analysis even more complicated than it needs to be for most purposes. Knuth [43, 44, 46] chooses an intermediate ground for his MIX machine, and derives running times for particular implementations of algorithms. Between the two extremes, results can differ by more than simply a constant scale factor. In choosing a model of computation, we try to balance realism against mathematical tractability. However, our experience is that results can be useful despite what appear to be overly simplified computational models.

Rather than finding exact running times on particular machines, most analyses count only certain "elementary" operations. The complexity measure reports the asymptotic growth of this operation count. Mathematicians have used various notations for such "order" results, as reported by Knuth [47], who suggests the following generally accepted version. To express the fact that a function $g(n)$ grows asymptotically no faster than another function $f(n)$, we write $g(n) = O(f(n))$, which reads, "of order at most $f(n)$". It is helpful to regard $O(f(n))$ as the set of functions with the property that for any function $g(n)$ in $O(f(n))$ there is a constant c such that $g(n) \leq cf(n)$. Similarly, $\Omega(f(n))$ ("of order at least $f(n)$") is the set of functions with the property that for any function $g(n)$ in $\Omega(f(n))$ there is a constant $c > 0$ such that $g(n) \geq cf(n)$. Finally, $\theta(f(n))$ ("of order exactly $f(n)$") is the set of functions with the property that for any function $g(n)$ in $\theta(f(n))$ there are constants $c_1 > 0$ and c_2 such that $c_1f(n) \leq g(n) \leq c_2f(n)$.

We commonly write, for example, $g(n) = O(f(n))$ rather than $g(n) \in O(f(n))$. In the present context, O-notation is used to describe upper bounds and Ω-notation is used for lower bounds. The θ-notation is used when it is possible to characterize the complexity to within a constant factor. Aho, Hopcroft, and Ullman [1] argue convincingly that such asymptotic analyses are especially meaningful in light of faster hardware. How much larger problems such machines can solve, and therefore their cost-effectiveness, depends on the growth rate of the computation time.

Consider the proble of finding the maximum element in a list of n items from a linearly ordered set. An appropriate choice for an elementary operation is a comparison between two items of the list, because the items might be records for which comparisons are nontrivial, while bookkeeping operations such as loop control and pointer management are usually proportional to the number of comparisons. By counting dominant operations, the asymptotic complexity measure is off by only a constant factor from the true number of operations performed.

Some analyses use the "uniform cost criterion," where memory references, comparisons, arithmetic operations, and all other instructions each take unit time. Still others employ the "logarithmic cost criterion," in which the charge for accessing a piece of information is proportional to the number of symbols needed to represent it. Normally, the choice of operations to be counted is not crucial to asymptotic analysis, assuming that the dominant operation is among them.

Measuring Problem Size

Problem size is another vague concept. It could be made exact by letting n be the number of symbols required to encode the problem for a particular Turing machine or for some other computational model. This encoding is extremely important (see [1], Section 10.1). Suppose that an algorithm takes as input a single integer, and that the input k requires ck time for some constant c. Let $n(k)$ refer to the length of the encoding of the integer k (i.e., $n(k)$ is the problem size when the input is the integer k). If k is encoded in unary notation, then $n(k) = k$, so the algorithm runs in cn time. However, if k is encoded in binary, then $n(k) = \log_2 k$, so that the running time of the algorithm is $c2^n$. Notice that for any radix representation, the size of the representation is within a constant factor of the size of the binary encoding. Because of this, along with the fact that in practice we nearly always use a radix representation for integers, unary notation is not appropriate.

Problem representation can be of major

importance in graph algorithms as well. Determining if a graph with n nodes is planar can be done in $\theta(n)$ time if the graph is represented by adjacency lists, but requires $\Omega(n^2)$ time if the representation is an adjacency matrix [72]. In contrast with the unary/binary choice for the representation of integers, both of these graph representations seem like reasonable choices, and indeed each is appropriate for various problems.

These points illustrate that the problem size must be explicitly defined in each case in order for results to have any meaning. For example, in sorting problems, n is the number of items to be sorted; for graph problems, it may be the number of vertices. Describing the problem size may even be more convenient if two or more parameters are used, for example the number of edges and the number of vertices in a graph. If a graph has V vertices and E edges, then it is clear what is meant by "an algorithm which requires $O(V + E)$ steps."

For the results to be of practical interest, definitions of both the measure of problem size and the measure of computing time should be closely related to the well-defined meanings which these terms have for actual machines. Thus, the random-access machine model is more commonly used than the Turing machine model for calculations on most computers. On the other hand, the Turing machine is useful in theoretical studies and in modeling computations which are tape-bound. Aho, Hopcroft, and Ullman [1] present a good account of the similarities and differences between these two models of computation.

2. LOWER BOUNDS

. . . abounding in intuitions without method . . .

-George Santayana

The more difficult of the bounds on problem complexity is the lower bound. There is no algorithm to analyze, few general principles to apply; proofs of results in this area often require outright cleverness. The results must apply to any algorithm, including undiscovered ones. Still a few techniques have been found useful, and others are promising.

Trivial Lower Bounds

The most obvious, and also the weakest, method produces what are appropriately called trivial lower bounds. The method consists of simply counting the number of inputs that must be examined and the number of outputs that must be produced, and noting that any algorithm must, at least, read its inputs and write its outputs.

There are many examples of the use of such a technique. One interesting graph problem is the single-source shortest-path problem: given a directed graph G with nonnegative edge weights, and a distinguished vertex v, find the minimum-weight path from v to each other vertex of G. Dijkstra [14] gives an algorithm for this version of the problem, where all edge weights are known to be nonnegative. A more interesting variation allows negative-weight edges but no negative-weight cycles; by definition of the problem, a correct algorithm must be able to detect negative-weight cycles. Let n be the number of vertices of G. Then there may be as many as $n(n - 1)$ edges in G, and any algorithm for solving the modified problem must inspect each of them. If some edge were ignored by any algorithm, we could change its weight so that a shortest path or a negative-weight cycle would be missed and force the algorithm to give a wrong answer. Hence there are inputs which require $\Omega(n^2)$ time for any algorithm to solve the modified single-source shortest-path problem.

Similarly, multiplication of a pair of $n \times n$ matrices requires that n^2 outputs be produced, and is therefore $\Omega(n^2)$. This says nothing about the number of multiplications required to solve the problem, but only that *some* operation (namely output) must be performed $\Omega(n^2)$ times; therefore, the dominant operation must be performed at least that many times.

Trivial lower bounds are generally easy to come by and, therefore, are of less interest than sharper bounds which can sometimes be proved by more sophisticated methods. In many cases, however, trivial bounds are the only lower bounds available. Because they are usually easy to prove, they should be tried before more difficult techniques are applied.

Information-Theoretic Bounds and Decision Trees

Several authors have used arguments from information theory to show that any algorithm for solving some problem must do some minimal amount of work. The most useful principle of this kind is that the outcome of a comparison between two items contains at most one bit of information (where "bit" denotes the values 0 or 1). Hence, if there are m possible input strings, and an algorithm purports to identify which one it was given solely on the basis of comparisons between input symbols, then $\lceil \log m \rceil$ comparisons are needed. This is because $\lceil \log m \rceil$ bits are necessary to specify one of the m possibilities (in standard binary notation, for example; for this reason, all logarithms in this paper are to the base 2).

The best-known example of a lower bound on computational complexity from information theory is for the problem of searching an ordered table with n elements for the position of a particular item. There are n possible outcomes, so unique identification of an index in the table requires $\lceil \log n \rceil$ bits. Hence, at least $\lceil \log n \rceil$ comparisons are required according to the principle mentioned above. A similar argument applied to the problem of sorting a linearly ordered set gives a lower bound of $\Omega(n \log n)$ for that problem.

The same basic principle often appears in a different guise in lower bound proofs for comparison-based problems. For example, Knuth [46] uses a "decision tree" model for the sorting problem, in which any sorting algorithm can be viewed as a binary tree. Each internal node of the tree represents a comparison between two input elements, and its two sons correspond to the two possible comparisons which are made next, depending on the outcome of the previous comparison. Each leaf node specifies an input permutation that is uniquely identified by the outcomes of the comparisons along the path from the root of the tree. Since there must be $n!$ leaf nodes, some path of the tree must be of length at least $\lceil \log n! \rceil$ This follows because the number of nodes at any level of a binary tree can be at most twice the number on the previous level. The worst case of any sorting algorithm must therefore be $\Omega(\log n!) = \Omega(n \log n)$.

Reingold [57] has extended this approach to allow for comparisons between functions of the inputs, rather than simply between input values themselves. He shows, for example, that deciding whether two sets of n real numbers are identical requires $\Omega(n \log n)$ comparisons, even if comparisons between linear functions of the inputs are allowed. Similarly, Dobkin and Lipton [15] show that the "element uniqueness" problem (determining among n real numbers whether any two numbers are equal) requires $\Omega(n \log n)$ steps, even if comparisons between linear functions of the inputs are allowed.

Oracles

Knuth [46] points out that a bound can be obtained for the problem of merging two ordered lists by another technique which he calls the construction of an "oracle", or "adversary". An oracle is a fiendish enemy of an algorithm which at every opportunity tries to make the algorithm do as much work as possible. Consider comparison-based algorithms for merging the two lists $A_1 < A_2 < \cdots < A_n$ and $B_1 < B_2 < \cdots < B_n$. The oracle will provide the result of any comparison on the basis of some rule; in this case, a useful rule is $A_i < B_j$ iff $i < j$. Of course, this rule applies only for certain inputs, but the algorithm does not know which input it has, nor does it know the rule, and must therefore ask the questions anyway. If comparisons are resolved by this oracle, merging must end with the configuration:

$$B_1 < A_1 < B_2 < A_2 < \cdots < B_n < A_n$$

since this is the only ordering consistent with the oracle's rule, and the algorithm must produce this output if it works properly.

Suppose that one of the comparisons between adjacent elements from this final list had not been made during the course of execution of the algorithm; say, $A_1{:}B_2$. Then the configuration:

$$B_1 < B_2 < A_1 < A_2 < \cdots < B_n < A_n$$

would also be a legitimate possible outcome, being indistinguishable from the correct answer on the basis of the comparisons which were made. Hence, all $2n - 1$ comparisons between adjacent elements of the final list must be performed for the algorithm to produce the correct output. An algorithm with this performance is easily constructed. We therefore know that for the problem of merging two ordered lists of n elements, $L(n) = U(n) = 2n - 1$.

For searching an ordered table, inquiries are of the form: "Is the key element less than this element?" The obvious oracle simply responds to inquiries in such a way that the key item is always in the larger of the two portions of the list. Thus at most half the table can be eliminated from consideration with each comparison, and at least $\lceil \log n \rceil$ comparisons are required. Hyafil [33], among many others, has used an oracle to prove a lower bound for the selection problem (finding the kth-largest of n elements), where the trivial bound is simply n. In this case, both upper and lower bounds are known to be $\theta(n)$, and the oracle provides a way of refining the lower bound to permit comparison with precise upper bounds.

Problem Reduction

One of the most elegant means of proving a lower bound on a problem P_1 is to show that an algorithm for solving P_1, along with a transformation on problem instances, could be used to construct an algorithm to solve another problem P_2 for which a lower bound is known. The power of this approach is substantial.

Shamos and Hoey [66] use problem reduction to show that an algorithm for finding the Euclidean minimum spanning tree of n points in the plane can be used to solve the element uniqueness problem, and must therefore take time $\Omega(n \log n)$. The reduction is quite simple. Suppose that we want to determine whether any two of the numbers $x_1, x_2, \cdots x_n$ are equal. We can solve this problem by giving any Euclidean minimum spanning tree algorithm the points $(x_1, 0)$, $(x_2, 0)$, $\cdots$ $(x_n, 0)$. The two closest points are known to be joined by one of the $n - 1$ spanning-tree edges, so we can simply scan these edges and, in linear time, determine if any edge has zero length. Such an edge exists if and only if two of the x_i are equal. Therefore, if the spanning tree algorithm could operate in less than $O(n \log n)$ time, the element uniqueness problem could be solved in less than $O(n \log n)$ time, contradicting the $\Omega(n \log n)$ lower bound mentioned earlier.

Other applications include the reduction of context-free language recognition to matrix multiplication [74] and the mutual reductions between Boolean matrix multiplication and transitive closure [18]. In these cases, the reductions provide the potential for proving lower bounds, but nontrivial lower bounds are not known for these particular problems.

Many other examples of this technique are found in transformations between so-called NP-complete problems, where the purpose is not to show lower bounds but to demonstrate membership in the equivalence class of NP-complete problems (see Section 5). Note that it is not always clear how to identify the problem P_2, which is of course a requirement for using this approach.

Miscellaneous Tricks

Among the newer approaches for proving lower bounds is the use of graph models of algorithms. Hyafil and Kung [34] show tradeoffs between the depth and breadth of trees describing the parallel evaluation of arithmetic expressions to show that the possible speedup using k processors is bounded by $(2k+1)/3$. This result is counter to the intuition that having k processors available would allow a speedup of k. In fact, for certain computations the speedup is even less; for adding up a list of k numbers, it is only $k/\log k$. The lack of a good model for parallel computation has hindered further development of ways of decomposing problems for parallel solutions and proving bounds such as these, even though the prospect of inexpensive parallel hardware compels us to study such algorithms. Lawler [51] expresses confidence that such graph-theoretic arguments will continue to prove useful in demonstrating lower bounds, and recent results (see, for example, [75]) show this optimism to be well-founded.

An interesting lower bound which relates to problem representations is the so-called Aanderaa-Rosenberg conjecture [59]. Simply stated, it asserts that detecting any nontrivial monotone graph property[1] requires $\Omega(n^2)$ steps if the graph is represented by an $n \times n$ adjacency matrix. A proof of this conjecture by Rivest and Vuillemin [58] is based on a decision tree model for the evaluation of Boolean functions and on properties of permutation groups.

Another new approach uses theorems from complex analysis. Shamos and Yuval [67] show that finding the average of all the interpoint distances for n points in the plane requires $\Omega(n^2)$ square-root operations. Their proof is based on the ambiguity of the square-root function. The primary significance of this result is that while it had previously been almost impossible to obtain lower bounds except for the four common arithmetic operations and for comparisons, the new approach applies to any multiple-valued function such as square root, inverse trigonometrics, and logarithms.

It remains to be seen whether these and other tricks will be applicable to enough problems to be called "methods" for proving lower bounds. At present, nontrivial results and general techniques are quite sparse.

[1] A graph property is nontrivial if at least one, but not all, graphs have it. It is monotone if adding new edges to the graph does not change the property. Thus, for example, nonplanarity is a nontrivial monotone property.

3. UPPER BOUNDS

Method is good in all things.
Order governs the world.
–Jonathan Swift

There are two powerful methods for proving upper bounds by analyzing the worst-case behavior of an algorithm: counting instructions, or solving recurrence relations. Either approach may require identification of a worst-case input. A third alternative, using brute force computing power to find both a worst-case and an associated optimal algorithm, sometimes works.

Identifying a Worst Case

In seeking an upper bound on problem complexity, the first task is to identify a "worst case", i.e., an input of size n which maximizes the amount of work the algorithm must do for that value of n. The algorithm with this input then defines $U(n)$, provided that no other algorithm's worst-case behavior is better.

When the algorithm's work is the same for all inputs of size n, finding the worst case is easy. This phenomenon is easily recognized when the flow of control does not depend on the data. Then every case is a "worst" one, because all cases are the same from the standpoint of the analysis. For example, a straightforward algorithm for finding the larget element in a set S is:

```
procedure largest (S);
  begin
    big := first element in S,
    for each remaining element x of S do big :=
      max(big, x);
    return(big)
  end;
```

Clearly, for every set S with n elements, the algorithm makes $n - 1$ comparisons (and this is optimal since the lower bound is also $n - 1$). Similarly, multiplication of two $n \times n$ matrices in the classical way takes $\theta(n^3)$ steps regardless of the data. There are many more examples of such algorithms for which identifying the worst case is easy because every case is the same.

Sometimes data-dependent decisions obscure the fact that every case is the same. Many search algorithms have this feature; for example, the binary search algorithm searches through a sorted array A consisting of n elements with indices i through j (where $n = j - i + 1$), for the position of a particular "key" item. For simplicity, assume that the key item is in the table. The procedure below returns the index in A of the key item:

```
procedure binsearch ($A, i, j, key$);
  if $i = j$
    then return($i$)
    else
      begin
        integer $middle = \lfloor (i+j)/2 \rfloor$ ;
        if $key \leq A[middle]$
          then return(binsearch($A, i, middle, key$))
          else return(binsearch($A, middle+1,$
            $j, key$))
      end;
```

It is easy to see that the basic idea here is "divide-and-conquer": one comparison detemines which half of the array to search next. A worst-case input is one for which the part of the table which remains to be searched is always at least as large as the part eliminated, which happens if the key item is in the first position of the array. We will see later (in the discussion of recurrences) that the algorithm takes $\lceil \log n \rceil$ comparisons in this case, which is optimal because a lower bound of $\lceil \log n \rceil$ is obtained from an information-theoretic argument.

A worst case which is only slightly harder to manufacture is one for Quicksort, an ingenious sorting algorithm which was proposed by Hoare [28]. The algorithm is very simple:

```
procedure quicksort($S$);
  begin
    if $|S| \leq 1$ then return($S$),
    choose some element $x$ from $S$,
    partition $S$ into these elements less than $x$
      ($S_1$), those equal to $x$ ($S_2$) and those greater
      than $x$ ($S_3$),
    return(quicksort($S_1$) followed by $S_2$ followed
      by quicksort($S_3$))
  end,
```

Quicksort is probably the best practical sorting algorithm known, primarily because on the average it uses $O(n \log n)$ comparisons. One might guess that this is true by noting that if each partitioning step divided the remaining lists into approximately equal parts, there would be about $\log n$ partitioning stages, each costing $O(n)$ time. A rigorous proof of this result is not difficult. Further aspects of the average case of Quicksort are discussed in Section 4.

This instance of Quicksort has poor worst-case performance. If all elements of S are distinct and the algorithm by poor luck picks x as the smallest element of S at every stage, then S_1 is empty, S_2 contains one element, and S_3 contains only one element fewer than S. This case requires about n partitioning stages, with the k-th stage examining about $n - k$ elements; thus the complexity is $\theta(n^2)$. This is not optimal, since sorting algorithms which never require more than $O(n \log n)$ steps are known, and the best lower bound is $\Omega(n \log n)$. A conceptually easy modification to Quicksort (choosing x as the median element of S) produces one such algorithm which is within a constant factor of being optimal; however, this algorithm is quite impractical. Heapsort [78, 19] is a practical sorting algorithm with complexity $O(n \log n)$.

For more complex algorithms, particularly those for graph problems and discrete optimization problems, finding a worst case can be more difficult. Scheduling problems have this feature (see [9]). Another interesting case is the modified single-source shortest-path problem (i.e., negative-weight edges are allowed). Edmonds and Karp [16] mention in passing that a modified version of Dijkstra's [14] algorithm runs in $O(n^3)$ time for any directed graph satisfying the conditions of the modified problem (described in Section 2). Their one-sentence justification is superficially convincing, but D. B. Johnson [36] shows an entire family of directed graphs for which the algorithm requires $\Omega(n2^n)$ time! This example demonstrates that it is easy to be misled by faulty identification of the worst case.

Counting Operations

Each of the above examples leads directly to a way of counting the number of steps required in the worst case. Often, however, this counting stage is at least as difficult as finding a worst-case input. In fact, it seems very similar to the problem of finding invariant relations for proofs of program correctness.

Consider an algorithm for finding the convex hull of n points in the plane. A correct solution (by definition of the convex hull) requires that the vertices of the hull be output in the order they would be encountered by tracing the convex hull in, say, a clockwise direction. Shamos [65] shows that a lower bound is $\Omega(n \log n)$, and Graham [26] has given an algorithm which achieves this. Graham's method operates in three stages. First, all n points are converted to polar coordinates about some point which is interior to the hull. Since any point which lies within the triangle formed by three noncollinear points of the set serves as such an origin, the conversion step can be done in $\theta(n)$ time. Next, the points are sorted by polar angle in $O(n \log n)$ time.

For the final phase, each point is on either one or two lists: the UNCONSIDERED list and the (circular) TENTATIVE list. The UNCONSIDERED list is the output of the sorting stage, while the TENTATIVE list is initially empty. A scan is then made through the UNCONSIDERED points. For each new point, the algorithm determines whether the path through the last two points on the TENTATIVE list and then through the new point turns to the right or the left. If it turns to the right, the algorithm adds the new point to the TENTATIVE list and continues scanning the UNCONSIDERED list. If it turns to the left, then the middle point of the three cannot be on the hull, so the algorithm deletes the last point of the TENTATIVE list from that list. Again, a path through the last two TENTATIVE points and the new point is checked, and this continues until a right turn is discovered or until there are fewer than two points on the TENTATIVE list. When all points on the UNCONSIDERED list have been scanned, the TENTATIVE list contains the points on the convex hull in the proper order.

It is not immediately clear how long this step may take, since for each new point the scan can apparently back up to the beginning of the TENTATIVE list. A reasonable (but pessimistic) guess for the complexity of the final phase would then be $O(n^2)$, which is not sufficient for our purposes. It is not particularly easy to identify a worst-case input because of the tradeoffs involved.

Graham noticed that all the work is done in creating and testing paths between three consecutive points, and that the total time is proportional to the number of times this must be done. Every time such a path is created and tested, either a new point is added to the TENTATIVE list, or a point is deleted from it. The number of points added cannot exceed n, and the number deleted also cannot exceed n because once deleted, a point is never considered again. Hence, a new three-point path is created at most $2n$ times.

The argument is tantamount to finding an invariant relation among the relevant quantities: P, the number of paths considered so far; A, the number of points which have been added to the TENTATIVE list so far; and D, the number which have been deleted from the TENTATIVE list so far. The quantity $P - A - D$, calculated after any path is considered, is constant. Hence, $P = O(A + D)$, and since $A + D \leq 2n$, $P = O(n)$. The running time of the entire algorithm is therefore dominated by the sorting stage, and is $O(n \log n)$.

A similar argument finds its way into the analysis of a number of other algorithms for very different problems. For instance, it is used to show that Tarjan's algorithms [72] which use depth-first search of graphs to solve various problems (such as finding biconnected components; see also [30]) run in an amount of time which is linear in the number of edges E. In this case, each edge is pushed onto a stack and later popped off and discarded. The total number of times this can be done equals the number of edges. A proof that Knuth, Morris, and Pratt's fast pattern matching algorithm [49] runs in linear time uses the same approach.

Another version of the argument is used by Hopcroft and Ullman [32] to analyze the behavior of a data structure and associated algorithms for set manipulation. A sequence of n UNION and FIND instructions is input; a UNION instruction requests that two disjoint sets be merged, and a FIND asks for the name of the set containing a particular element. Such a sequence can be processed in $O(nG(n))$ time, where $G(n)$. is the least k such that the k-th iterated logarithm of n does not exceed one. The proof of this bound relies on an "accounting trick" in analyzing the FIND algorithm. The total cost of all FIND instructions is divided between the instructions themselves and the elements inspected by the instructions. Each account is then analyzed separately and shown to be charged $O(nG(n))$ units; thus the total is $O(nG(n))$ for n FIND instructions, which dominates the time for the UNION instructions. The reader is urged to consult Hopcroft and Ullman [32] or Aho, Hopcroft, and Ullman [1] for more details.

Recurrences

Although analysis of worst-case behavior by directly counting the number of steps is greatly simplified by a concrete description of the algorithm, it is not always necessary to be so explicit. In deriving recurrence relations for solution times, it is sometimes more convenient to think in abstract terms about what the algorithm does. This is especially true when the algorithm itself is not written recursively.

In accordance with established usage, let the running time be denoted by $T(n)$. It is sometimes possible to find a recurrence relation (difference equation) for $T(n)$ and to solve it exactly—or even just approximately, concluding that $T(n)$ is $O(f(n))$, for example—to discover the worst-case behavior of an algorithm.

Recall the first example of the previous subsection, where the problem was to find the largest element in a set S of n elements. Although the algorithm is not written as a recursive procedure, it can nevertheless be viewed as finding the largest element of a set S' consisting of the first $n - 1$ elements, then comparing the result to the nth element of S. The recurrence obtained is

$$T(n) = T(n-1) + 1 \qquad \text{for } n > 1,$$
$$T(1) = 0$$

where the initial condition is zero because no comparisons are needed to find the maximum element of a singleton set. The solution to this recurrence is clearly $T(n) = n - 1$, the same result as before.

Next consider the binary search algorithm, where it is true that

$$T(n) \le T(\lceil n/2\rceil) + 1 \qquad \text{for } n > 1,$$
$$T(1) = 0.$$

Again, the initial condition is zero because the process is finished without any more comparisons when only one element remains in the array. The recurrence is discovered by recognizing that in the worst case, one comparison is used to determine which remaining part to search recursively, so that the total number of comparisons $T(n)$ is the sum of this comparison plus the number $T(\lceil n/2\rceil)$ required to find the key in an array essentially half as large. Since the remaining part can never be larger than $\lceil n/2\rceil$ whether n is odd or even, we use '$\le$' rather than '$=$'. The solution $T(n) \le \lceil \log n\rceil$ can be discovered by computing the value of $T(n)$ for the first several values of n, and can then be proved by induction. As mentioned before, the binary search algorithm is optimal, since this solution coincides with the lower bound.

A more complicated recurrence results from analyzing the worst case of Quicksort. Here, the equation is

$$T(n) = T(n-1) + P(n) + C(n) \qquad \text{for } n > 1,$$
$$T(1) = T(0) = 0,$$

where $C(n)$ is the number of comparisons required to choose an element x from S, and $P(n)$ is the number needed to partition S on the chosen element x. Since we are counting only comparisons, $C(n) = 0$ and $P(n) = n - 1$; thus,

$$T(n) = T(n-1) + n - 1 \qquad \text{for } n > 1,$$
$$T(1) = T(0) = 0,$$

for which the exact solution is $T(n) = n(n - 1)/2$. Of course, it is clearly not sufficient to choose x arbitrarily if a worst case of $\theta(n^2)$ must be avoided. Rather, x should partition S into approximately equal parts. Aho, Hopcroft, and Ullman [1] call this the "principle of balancing." It can be accomplished by choosing x as the median element of S, whereupon the recurrence becomes

$$T(n) \leq 2T(\lfloor n/2 \rfloor) + P(n) + C(n) \quad \text{for } n > 1,$$
$$T(1) = T(0) = 0.$$

As in the case of binary search, '$\leq$' rather than '$=$' is used because n may be odd and because the partitioning element is already in place and need not be considered in the recursive step. $T(n)$ still provides an upper bound since neither remaining list can contain more than half the elements. It is convenient to think of T as being defined not only for integer arguments, but on the entire real line. Since T is a nondecreasing function, $T(\lfloor x \rfloor) \leq T(x)$, so the inequality remains valid if $\lfloor n/2 \rfloor$ is replaced by $n/2$. This step greatly simplifies the task of solving the recurrence.

Now, $P(n) = n - 1$ as before; however, $C(n)$ is no longer zero but the number of comparisons necessary to find the median of n elements. Blum et al., [5] present an algorithm which finds the median in at most $5.43n$ comparisons, and Schonhage, Paterson, and Pippenger [63] have an algorithm which uses at most $3n$ comparisons. Taking $C(n) \leq 3n$, we have

$$T(n) \leq 2T(n/2) + 4n - 1 \quad \text{for } n > 1,$$
$$T(1) = T(0) = 0,$$

for which the solution is $T(n) \leq 4n \log n - n + 1$, so that $T(n)$ is $O(n \log n)$. This algorithm is not practical, however, because the $3n$ median algorithm is extremely complicated, and because the expected behavior of Quicksort is very good without such a modification (see Section 4).

As a final example, consider Strassen's algorithm [71] for matrix multiplication. The algorithm is based on recursive application of a method for multiplying 2×2 matrices with elements from an arbitrary ring, using seven multiplications and 18 additions (as opposed to the textbook method which uses eight multiplications and four additions). Aho, Hopcroft, and Ullman [1] explain how the number of additions can be reduced to 15, which is optimal.

Suppose $n = 2^k$. Strassen's algorithm begins by partitioning each of the original matrices into four submatrices of size $2^{k-1} \times 2^{k-1}$ (to which the algorithm is applied recursively), then multiplies the 2×2 matrices (which have $2^{k-1} \times 2^{k-1}$ matrices as elements) using the seven-multiplication 15-addition algorithm. The recursive application of the algorithm to perform the seven multiplications is the key to its efficiency. Even though the total number of scalar operations used to multiply 2×2 matrices of scalars is 22, as opposed to 12 using the classical method, the number of multiplications is reduced by the new algorithm. When the elements of the 2×2 matrices are themselves matrices, this fact becomes important, because matrix multiplications are more costly than matrix additions.

An analysis of the tradeoffs involved in sacrificing 11 additions in order to save one multiplication begins with the recurrence describing the number of scalar operations used to multiply a pair of $n \times n$ matrices:

$$T(n) = 7T(n/2) + 15(n/2)^2 \quad \text{for } n > 1,$$
$$T(1) = 1.$$

Here, the recursive step requires seven applications of the algorithm to $(n/2) \times (n/2)$ matrices, plus 15 additions of such matrices. Each matrix addition takes $(n/2)^2$ scalar operations. The initial condition is obvious, since multiplication of two 1×1 matrices consists of a single scalar multiplication. The solution is $T(n) = 6n^{\log 7} - 5n^2$; thus $T(n)$ is $\theta(n^{\log 7}) = \theta(n^{2.81})$, compared to $\theta(n^3)$ for the classical method. Because of the factor of 6, though, the classical algorithm (which takes $2n^3 - n^2$ operations) is still faster for n less than about 300. By using a hybrid scheme which uses Strassen's algorithm for large matrices and the classical algorithm for smaller ones, this crossover point can be reduced (for a real implementation) to about $n = 40$ [11].

Simply finding a recurrence is only part of the problem; the other half, of course, is solving it. It is relatively easy to find an upper bound on the solution by simply guessing a solution and then trying it. For example, given the recurrence

$$T(n) = 2T(n/2) + n \log n$$

with some initial condition $T(1)$, we might guess that $T(n)$ should be no larger than $O(n^2)$. If we assume that $T(n) = cn^2$ and can show that the right-hand side of the recurrence is at most cn^2 + lower-order terms, then $O(n^2)$ is an upper bound on $T(n)$. That this is true for the present example is easily verified.

A better guess in this case is that $T(n)$ is $O(n \log^2 n)$; this means that our guess of the value of $T(n)$ is $cn \log^2 n$, resulting in:

$$\begin{aligned} T(n) &= 2T(n/2) + n \log n \\ &= cn \log^2(n/2) + n \log n \\ &= cn \log^2 n + (1 - 2c)\, n \log n + cn \end{aligned}$$

so that $O(n \log^2 n)$ is an upper bound for $T(n)$. In fact, since the coefficient of $n \log^2 n$ is the same on both sides of the equation, $T(n) = \theta(n \log^2 n)$.

This computation can be extended to find the exact solution. Suppose that $T(n)$ is a linear combination of linearly independent functions, the dominant one of which is $n \log^2 n$. Substituting $cn \log^2 n$ into the recurrence gives rise to terms in $n \log n$ and in n, which appear on the right-hand side but not on the left. Consequently, $T(n)$ must also have terms $an \log n + bn$, which, when expanded on the right, produce no new lower-order terms. Now it is a simple matter to equate coefficients of like functions to get the solution: $T(n) = (n \log^2 n + n \log n)/2 + T(1)n$.

More powerful techniques must sometimes be applied. Generating functions (Z-transforms), which are also of value in solving problems associated with average-case analysis, are among the most useful of these tools. Knuth [43], Liu [52], and Kleinrock [42] give excellent accounts of how to use this method. Some relatively easy recurrences can also be solved by referring to standard formulas (see, for example, [13] on difference equations), while at least references to others can be found by iterating the recurrence to find the first few terms and then looking up the sequence in Sloane [68].

The Brute Force Method

Even though the method to be described here is not often practical, it is interesting because it is possible at all only with the aid of high-speed computers and, therefore, could only recently have been attempted. The question of how to sort using a minimum number of comparisons is considered in detail by Knuth [46], who points out that the merge-insertion algorithm of Ford and Johnson [21] is optimal for $n < 12$ and for $n = 20$ and 21. That is, the number of comparisons is exactly $\lceil \log n! \rceil$ for these cases.

The question of the optimality of merge-insertion for $n = 12$ was settled by Wells [76, 77] by using brute-force computing power to demonstrate that no algorithm can sort 12 items using fewer than 30 comparisons, so that merge-insertion (which uses 30) is optimal even though $\lceil \log 12! \rceil = 29$. In a sense, he refined the lower bound on sorting 12 elements by effectively bounding the worst-case performance of every possible algorithm! One can imagine finding worst cases in a similar manner to demonstrate upper bounds, either by proving the existence of an algorithm or by explicitly producing one.

4. THE AVERAGE CASE

What is normal is at once most convenient, most honest, and most wholesome.

—Frederic Amiel

The normal is what you find but rarely.

—W. Somerset Maugham

Recent efforts in algorithmic analysis have been largely directed toward analyzing expected behavior, i.e., finding the complexity of a computation averaged over some distribution of inputs. Generally, the techniques reported in the previous section are still applicable, although some of the recurrences are tougher to handle and therefore stronger solution methods may need to be used.

Pros and Cons of Average-Case Analysis

The primary reason for analyzing the behavior of algorithms "on the average" is, of course, that a worst case may arise so rarely (perhaps never) in practice that some other complexity measure would be more useful. An alternative to worst-case analysis that immediately comes to mind is some sort of average-case analysis. Rather than try to define and analyze a particular case which is somehow "average," the approach is to simultaneously analyze all cases and to weight the individual case complexities with the appropriate probabilities of each case occurring.

Obviously, this complicates the mathematics considerably. If this were the only objection to doing average-case analysis, all that would be required would be more sophisticated tools, which could be discussed in detail here. However, more serious questions have been raised which tend to cast considerable doubt on the entire venture; this is the main reason for not going into more detailed description of the methods used in average-case analysis. Rather, the reader is urged to consult the original sources.

The most important objection is that, typically, there is no way to identify the probability distribution over all problem occurrences. While it may be reasonable in some situations to assume that every possible abstract problem instance is equally likely (for example, that every item is equally likely to be the key in a binary search, or that every permutation is equally likely to be the input to Quicksort, or that every n-vertex graph is equally likely), this assumption really only makes sense if the problem space is finite. It clearly makes no sense, for example, to say that every integer program is equally likely. Furthermore, even when it does have meaning, the assumption of a uniform distribution over all possible inputs may not be at all realistic if we have prior knowledge about the likelihood of various inputs. While a uniform distribution is not the only possible assumption, it is the one most often encountered. Exceptions are Spira [70] on the expected time for finding all shortest paths in a graph, and Bentley and Shamos [4] on the expected behavior of some geometric algorithms. In both of these cases, the only important assumptions deal with independence of the random variables involved and not with their distribution. The assumptions made in analyzing scheduling algorithms or parallel computations often include exponentially distributed or Erlangian distributed services times for tasks [8, 2].

In one attempt to answer this objection, Yuval [79] has suggested that algorithms might "randomize" their inputs in order to make the assumption appear valid. He has pointed out that if suitable random steps were being taken at certain stages, an algorithm could have good expected behavior for every input, and thereby assure good expected-case solution times regardless of the probability distribution being assumed (see also [55]). For example, Quicksort could choose the partitioning element randomly (this idea was offered by Hoare [28] in his original paper on Quicksort). Even though this might seem a case of the tail wagging the dog, there is some justification for such an approach in this instance. In order to make the analysis of the algorithm tractable, Sedgewick [64] assumes that the files to be sorted are random, and presents evidence that the algorithm works better if they are! Clearly, not all algorithms can be modified in this manner. The key element in binary search cannot be "randomized"; it is given as the sole input. Similarly, the array in which the search is to be made is certainly fixed during the search, so there is no room to manipulate the algorithm in this way.

Despite the shaky basis for assuming random inputs, the results of analysis using this assumption may be reasonable approximations for other distributions. Moreover, nonuniform distributions complicate the recurrences.

Knowing the average behavior of an

algorithm is helpful, but knowing the variance as well would be even more so. Few average-case analyses have considered higher moments of the solution time. One notable exception is Sedgewick's analysis of Quicksort [64] (see also Knuth [46]), in which he shows that the average number of comparisons is about $\mu = 1.39n \log n$, and the standard deviation is approximately $\sigma = 0.65n$. Since σ/μ diminishes with increasing n, our confidence that the algorithm will be efficient grows with n. This explains in part why Quicksort works so well in practice. Spira [70] also finds the variance of the time required by his all-shortest-paths algorithm.

Some Examples of Average-Case Analysis

There is a large and growing number of algorithms that have been subjected to analysis of average-case complexity. While only a few will be discussed here, many others can be found in the literature.

Surely the single most comprehensive analysis of any algorithm is presented for Quicksort by Sedgewick in his PhD thesis [64]. He analyzes the average number of comparisons, exchanges, and partitioning stages through the use of recurrences, just as for the simple worst-case analysis of the number of comparisons described in Section 3. Along with Sedgewick's Appendix B, Knuth [43, 46] is a good reference for these techniques.

Obviously, not all recurrences are easy to solve. In analyzing radix exchange sorting, Knuth [46] uses properties of the gamma function and complex-variable theory to derive asymptotic results for a recurrence that looks fairly simple, but cannot be solved with traditional techniques.

One algorithm that has been analyzed in at least three different ways is the alpha-beta search algorithm for game trees. The problem which the algorithm solves is a search through a game tree. There are two players who alternately make moves, the first trying to maximize some function of the position, the other trying to minimize it. The classical example is the game of chess; all sophisticated chess-playing programs use alpha-beta search. Good descriptions of the algorithm can be found in Fuller, Gaschnig, and Gillogly [23] and in Knuth and Moore [48]. Fuller, et al., assume that the game tree is a complete tree with branching factor N and depth D, and that each permutation of the ranks of the values of the leaf nodes is equally likely. They derive expressions for the probability of expanding individual nodes, and the expected number of evaluated bottom positions. While the answers in this case look simple enough because of concise notation, the authors point out the computational infeasibility of calculating these quantities for any but very small values of N and D. However, they surmise from simulation results that the average number of nodes examined is about $O(N^{0.72D})$.

Knuth and Moore [48] make the same assumptions about the tree and the random ordering of leaf-node values, and show that the average behavior of the algorithm is $O((N/\log N)^D)$. They suggest that the simulation results by Fuller, et al., result in a fit to $N^{0.72D}$ because N is so small.

Newborn [53] uses a model in which the branch (rather than the node) values are randomly ranked; he obtains results for the cases $D = 2, 3, 4$, and in each case they differ from those of Knuth and Moore. This is yet another example of the dependence of results on the assumptions of the model being used in the analysis: the three different analyses for $D = 2$ give complexities of $O(N^{1.44})$, $O((N/\log N)^2)$, and $O(N \log N)$, respectively.

Other examples of average-case analysis include Guibas and Szemeredi [27] on double hashing, O'Neil and O'Neil [54] on Boolean matrix multiplication, and Knuth [45] and Floyd and Rivest [20] on selection.

5. APPROXIMATE ALGORITHMS

Trouble creates a capacity to handle it.
-Oliver Wendell Holmes, Jr.

Until very recently, the focus of attention in algorithmic analysis has been on "tractable" combinatorial problems such as searching, sorting, and matrix multiplication, which have been mentioned above. These are among the "easy" problems (which in current terminology means that their complexity is bounded by a polynomial in n); on the other hand, many optimization and graph problems are "hard" (i.e., their complexity is apparently not bounded by any such polynomial). Since so many important problems are, unfortunately, in the latter category, an entire new group of algorithms that find approximate solutions to hard problems has been developed. They are known as "approximation" or "heuristic" algorithms. With these algorithms have come new measures of "goodness" as well as techniques for their design and analysis. Two classes of approximation, one guaranteeing a near-optimal solution always, and the other producing an optimal or near-optimal solution "almost everywhere," are discussed.

Problem Classes and Reducibility

For the discussion that follows we require some concepts regarding Turing machines (TM) and formal languages. It is somewhat artificial, but convenient, to pose a problem in terms of a language-recognition task. This is done by formulating it to require a yes-no solution; for example, "Does this traveling salesman problem have a solution with cost less than k?" A TM can be asked to accept the input problem description if the answer is "yes." To solve the problem, such a TM then accepts only input strings from some language L which comprises precisely those problem instances with "yes" answers. It is helpful to refer to the original problem and the language L interchangeably in the context of the classes P and NP (see [1] for more details).

Formally, the class P (for "Polynomial") is the set of languages which are recognized by some deterministic TM that always halts in a number of steps which is bounded by a fixed polynomial in the length of the input (i.e., in "polynomial time"). Similarly, the class NP (for "Nondeterministic Polynomial") is the set of languages which are recognized by some *non*deterministic TM in polynomial time. A nondeterministic TM operates in polynomial time if all sequences of choices of moves are of polynomial bounded length; a string is accepted by such a machine if there exists any such sequence of steps which leads to an accepting state (see Hopcroft and Ullman [31]).

That $P \subseteq NP$ is clear from the definitions. Undoubtedly the most intriguing open question in the complexity area is whether $P = NP$, or whether there are problems in NP which cannot be solved in polynomial time by a deterministic TM. Problems known to be in P include the "easy" problems previously discussed. Other problems in NP which might also be in P are most of the so-called optimization and graph problems, such as 0/1 integer programming, certain scheduling problems, finding Hamiltonian circuits, graph coloring, and many others (see Karp [39, 40]). So far as is known, there is no deterministic polynomial-time algorithm for solving any of these "hard" problems; all known algorithms have a worst-case complexity that is not bounded by a polynomial function of the input size.

One method of proving the equivalence $P = NP$ might be to demonstrate a deterministic polynomial-time simulation of a nondeterministic TM. Since this would amount to a systematic search through a tree of move sequences of polynomial depth, the total number of nodes (and hence the time for the simulation) could be exponential. Hence, it appears that it is not possible to simulate an arbitrary nondeterministic TM operating in polynomial time by a deterministic TM operating in polynomial time. Of course this does not by any means show that $P \neq NP$; there might well be other approaches, not resembling a deterministic simulation, which could solve one of the hard problems in NP in polynomial time.

Fortunately, it is sufficient to consider solving these problems in polynomial time on a normal random-access computer rather than on a TM; the problems in P remain the same [1], because each machine can simulate the other in polynomial time. Roughly speaking, problems that seem to require some sort of backtrack-searching through a tree of polynomially bounded depth are the difficult problems of NP. While there is overwhelming circumstantial evidence that such problems are not also in P, no proof of this conjecture has yet been produced.

A language L_1 is "polynomially reducible" to L_2 if there is a deterministic polynomial-time algorithm which transforms a string x into a string $f(x)$ such that x is in L_1 iff $f(x)$ is in L_2. Among the consequences of reducibility is the fact that if there is a polynomial-time deterministic algorithm to recognize L_2 and L_1 is reducible to L_2, then a polynomial-time deterministic algorithm to recognize L_1 can be constructed. It consists of applying the transformation f to the input x and then checking whether $f(x)$ is in L_2, all of which can be done deterministically in polynomial time.

The key to the argument that $P \neq NP$ is a remarkable theorem by Cook [12] stating that every problem in NP can be polynomially reduced to Boolean satisfiability. This problem is very simply stated: Is there an assignment of truth values to the literals of a Boolean expression which makes the expression true? This means that every problem which can be solved in polynomial time on a nondeterministic TM can also be solved by subjecting the input string to a transformation (done deterministically in polynomial time) that converts it to an instance of satisfiability, and then solving the resulting satisfiability problem. Many other problems have subsequently been shown to have the same property. A problem such as satisfiability is called "NP-complete." One way to prove that a problem is NP-complete is to demonstrate that the problem is in NP, and that another NP-complete problem is reducible to it. All NP-complete problems are also "P-complete" (see Sahni [60]), i.e., in the class of problems solvable in deterministic polynomial time iff $P = NP$. The distinction between these definitions has all but disappeared in the literature, and the term "NP-complete" has come to be used for either one. Under either definition, if there is a polynomial-time deterministic algorithm for any NP-complete problem, then $P = NP$.

Karp [39] shows reducibilities among problems which demonstrate that the class of NP-complete problems is quite large, and includes all the optimization and graph problems mentioned above. This fact forms the basis for believing (even if one cannot prove) that $P \neq NP$, since none of the hundreds of algorithms for the scores of problems that are NP-complete runs in polynomial time. If any of these problems could be solved quickly, all of them could be; the fact that so far none of them can be is a convincing argument (although not a proof) that they never will be.

Whether $P = NP$ is not the only open question in this area. Some problems remain unclassified. For example, deciding whether two regular expressions are equivalent, or whether a string is in a given context-sensitive language, are problems at least as hard as any problems in NP, but are not known to be in NP themselves (see Aho, Hopcroft, and Ullman [1]). Such problems are called "NP-hard" because their inclusion in P would imply that $P = NP$. On the other hand, deciding whether two given graphs are isomorphic, whether a given integer is a prime, or whether a given integer is not a prime, have not been shown to be NP-complete, even though they are in NP and cannot currently be solved in polynomial time. Hence, these may be problems in NP but not in P, whose solutions will not help solve the other problems in NP. Ladner [50] has shown that $P \neq NP$ implies the existence of such "intermediate" problems, but it is not known whether the above problems are in this class.

"Guaranteed" Approximation Algorithms

All known algorithms for solving NP-complete problems run in nonpolynomial deterministic time. It is therefore impractical to solve very large instances of these problems, even though some of them are among the most important problems of graph theory and operations research. Fortunately many applications requiring solution of such problems do not require exact solutions. To take advantage of this fact, algorithm designers have developed many new approximation methods. Most of these algorithms seek near-optimal solutions to all instances of a problem.

A generally accepted error measure for these algorithms is the maximum relative error over all problem instances [37]. Normally, the quantities from which the relative error is computed are obvious: an integer program's objective function value, a graph's chromatic number for the graph coloring problem, or a schedule's length. Sometimes, as for the satisfiability problem, the original problem must be rephrased as an optimization problem in order to study approximate solutions. The ratio of the absolute error to the exact solution value is given the symbol ϵ.

In designing approximation algorithms of this type, one is concerned with guaranteeing that the relative error ϵ is never larger than some prescribed maximum, which essentially means that ϵ is taken as given. The goal is to develop an algorithm that always solves the problem to within a factor of $1 \pm \epsilon$. For this reason, many of these algorithms tend to resemble the patterns commonly encountered in calculus proofs, with strange functions of ϵ appearing, as if by magic, in the early stages of the algorithm in order to assure that the final solution will be within a factor of $1 \pm \epsilon$ of optimal. Recognizing this takes much of the mystery out of what can appear to be very complicated algorithms.

Because error analysis is part of the design of such algorithms, understanding the analysis is tantamount to understanding the operation of the algorithm. Sahni [62] nicely summarizes the techniques in this class, dividing the methods into three categories. These are especially useful for the knapsack problem, packing problems, and certain scheduling problems. The worst-case complexities they produce are usually like $O(n^a/\epsilon^b)$ for constants a and b which depend on the problem.

An example of one of Sahni's techniques, called interval partitioning, is a simple algorithm for solving the 0/1 knapsack problem:

$$\begin{aligned} &\textit{maximize: } \mathbf{c}\cdot\mathbf{x} \\ &\textit{subject to: } \mathbf{a}\cdot\mathbf{x} \le b \\ &\qquad\qquad\quad \mathbf{x} \in \{0, 1\}^n \end{aligned}$$

Here, $\mathbf{c}$ and $\mathbf{a}$ are n-vectors of positive "utilities" and "weights", respectively, and b is a scalar representing the capacity of a knapsack. The objective is to fill the knapsack with a subset of the n items so that the total utility $\mathbf{c}\cdot\mathbf{x}$ is maximized without violating the capacity constraint $\mathbf{a}\cdot\mathbf{x} \le b$. Item i is to be included iff $x_i = 1$.

The problem can be solved by a straightforward tree-search method. For any partial assignment to the variables $x_1, x_2, \cdots x_i$ which we may have at level i of the tree, there correspond two more assignments (letting x_{i+1} be either 0 or 1) at level $i + 1$. A feasible solution is one for which $\mathbf{a}\cdot\mathbf{x} \le b$, and a partial assignment at level i consists of the fixed variables x_1 through x_i, with the remaining ones set to 0. The key observation is that the infeasibility of a partial assignment implies the infeasibility of every completion; accordingly, we may prune the tree at that point. However, even with such pruning rules, the total number of solution candidates generated may be exponential in n.

An approximation scheme using interval partitioning is constucted as follows: let P_i be the maximum total utility of partial solutions on level i. Divide the interval $[0, P_i]$ into subintervals of size $P_i\epsilon/n$, and discard all candidates with total utility in the same interval except that one with the least total weight. There are now at most $\lfloor n/\epsilon \rfloor + 1$ nodes on each level, and therefore only $O(n^2/\epsilon)$ nodes in the entire tree. The errors introduced at each stage are additive, so the total error is bounded by ϵ. This means that in $O(n^2/\epsilon)$ time we can solve the 0/1 knapsack problem to within ϵ, for every $\epsilon > 0$. By another approach, Ibarra and Kim [35] have shown that this can be reduced to $O(n \log n) + O((3/\epsilon)^4 \log(3/\epsilon))$.

Shamos and Yuval [67] have demonstrated an interesting approximation algorithm for a problem which is relatively easy. It finds an ϵ-approximation to the mean distance between n points in the plane in $\theta(n)$ time although an optimal solution requires $\theta(n^2)$ time.

Not all NP-complete problems lend themselves equally well to approximation, even though they are, in a certain sense, of the same time complexity. D. S. Johnson [37] points out that for some problems, including those to which Sahni's techniques can be applied, the relative error ϵ can be bounded by a constant independent of problem size. However, for others (such as the maximum clique problem) no algorithm has yet been found for which ϵ does not grow at least as fast as n^c for some $c > 0$. Garey and Johnson [24] show that approximating the chromatic number of a graph to within a factor of two is NP-complete; in fact, no known polynomial-time algorithm solves the problem to within any bounded ratio. Similarly discouraging is another result reported in the same paper: if there is a polynomial-time algorithm for approximating the size of the maximum clique of a graph to within *some* bounded ratio, then there is a polynomial-time algorithm for approximating it to within *any* bounded ratio.

Examples of approximation algorithms and analysis of errors and complexities have also been given by Johnson [38], Sahni [61], and Coffman and Sethi [10], among many others. Garey and Johnson [25] have compiled a fine annotated bibliography for this area.

Probabilistic Behavior of Approximation Algorithms

It is not easy to guarantee good approximate solutions to certain NP-complete problems. This fact has motivated a search for alternatives to the guaranteed approximation approach. One alternative is to make such a sophisticated guess about the solution that the likelihood of error is negligibly small. If we define p_n to be the probability that such an algorithm gives an unacceptably bad answer to a randomly chosen problem of size n—some distribution of problem instances is assumed—then the algorithm is said to work correctly "almost everywhere" if $\sum p_n$ is finite, where the sum is over all problem sizes $n = 1, 2, \cdots$.

Optimality or near-optimality "almost everywhere" is theoretically a strong condition. Were we to randomly choose one problem instance of each size n, for $n = 1, 2, \cdots$, and run the algorithm on each of them, then not only would the algorithm give good answers infinitely often, but (with probability one) it would fail to give good answers only finitely often.

Karp [41] illustrates such algorithms. He demonstrates an $O(n \log n)$ algorithm for solving random Euclidean traveling salesman problems which, for every $\epsilon > 0$, finds a solution within a factor of $1 + \epsilon$ of being optimal almost everywhere. The analysis depends heavily on an interesting theorem by Beardwood, Halton, and Hammersley [3], stating that for n points chosen at random in a plane figure of area A, there is a constant c such that the length of the shortest tour is within ϵ of $(cnA)^{1/2}$ almost everywhere.

Karp's example, which uses results from geometry and probability, is typical of these algorithms. Results and techniques from a large number of fields of mathematics seem to apply in the area of probabilistic algorithms. More examples can be found in the theory of random graphs [17], which contains many useful theorems regarding connectedness, maximum cliques, and chromatic numbers. Rabin [55] and Solovay and Strassen [69] describe a different kind of probabilistic algorithm for testing whether a number is prime; the probability of error (guessing that a composite number is prime) is halved at each step regardless of the size of the number being tested.

Many probabilistic algorithms are remarkably simple, yet have been shown to work extremely well. However, the probabilistic approach suffers from the same basic objections as average-case analysis, especially with regard to the question of assuming a meaningful underlying distribution over problem instances. Even so, the approach does give some assurances that an approximation algorithm will not fail very often; the user may confidently expect few "surprises."

CONCLUSIONS

He that takes up conclusions on the trust of authors . . loses his labour, and does not know anything, but only believeth.
-Thomas Hobbes

There is a clear need for development of more sophisticated tools for proving lower bounds. Techniques for proving upper bounds will not change significantly, and algorithm design will continue to advance with more unified principles. Average-case analyses must be extended to include estimates of the variance and the nature of the distribution of solution times, and should be made more robust by requiring fewer unreasonable assumptions.

An area which has hardly been explored is that of realistic models for complexity analysis of parallel programs. Most prior work has dealt with parallel computation at the single-instruction level, and then essentially only for algebraic or numerical problems; see, for example, Traub [73] and Borodin and Munro [7]. There are practically no parallel algorithms and associated analysis techniques for combinatorial problems. The proliferation of parallel hardware makes this a particularly attractive area for exploration.

Possibly the most exciting and potentially rewarding area for research in the near future will be in designing and analyzing algorithms for very hard problems. Of course, the interesting theoretical questions include the now-infamous P = NP problem. Since presumably P ≠ NP, approximate algorithms will predominate. For those problems which are not amenable to guaranteed approximate solutions, the probabilistic approach may be preferred. However, the "almost everywhere" concept is too restrictive and says too little about problems of a particular size; a new definition of what constitutes an acceptable approximation algorithm might result in a new family of practical algorithms for some NP-complete problems. Such results can be expected to heavily apply concepts from probability and statistics.

APPENDIX

Glossary of Problems

Biconnected components: Given a graph G, determine whether G remains connected when any vertex and its incident edges are removed. (Sometimes, find all such vertices.)

Context-free language recognition: Given a context-free language L and a string x, determine whether x is in L.

Context-sensitive language recognition: Given a context-sensitive language L and a string x, determine whether x is in L.

Convex hull: Find the smallest convex set containing n given points.

Element uniqueness: Determine whether any two of n given elements are equal.

Equivalence of regular expressions: Determine whether two given regular expressions denote the same set.

Graph coloring (chromatic number): For a graph G, find an assignment of "colors" to the vertices of G such that no adjacent vertices are the same color, and such that the number of different colors used is a minimum. The number of colors used is called the "chromatic number" of G.

Graph isomorphism: Determine whether two given graphs are isomorphic.

Hamiltonian circuit: Determine whether a given graph contains a cycle passing through each node exactly once.

Integer programming: Given an n-vector $\mathbf{c}$, an $m \times n$ matrix A, and an m-vector $\mathbf{b}$, find an n-vector x (of integers) which maximizes $\mathbf{c} \cdot \mathrm{x}$ subject to $\mathrm{Ax} \leq \mathbf{b}$ and $\mathrm{x} \geq \mathbf{0}$. For the 0/1 version, each x_i must be either 0 or 1.

Knapsack problem: Given n-vectors $\mathbf{c}$ ("utilities") and $\mathbf{a}$ ("weights"), and a scalar b ("capacity"), find an n-vector x (of integers) which maximizes $\mathbf{c} \cdot \mathrm{x}$ subject to $\mathbf{a} \cdot \mathrm{x} \leq b$ and $\mathrm{x} \geq 0$. For the 0/1 version, each x_i must be either 0 or 1.

Linear programming: Given an n-vector $\mathbf{c}$, an $m \times n$ matrix A, and an m-vector $\mathbf{b}$, find an n-vector x (of reals) which maximizes $\mathbf{c} \cdot \mathrm{x}$ subject to $\mathrm{Ax} \leq \mathbf{b}$ and $\mathrm{x} \geq \mathbf{0}$.

Matrix multiplication: Given matrices A and **B**, compute the matrix product **AB**.

Maximum clique: Determine the size of the largest complete subgraph of a given graph.

Maximum element: Find the largest of n elements of a linearly ordered set.

Mean distance: Find the average of all the pairwise distances between n points in space.

Median: Find the median of a list of n elements of a linearly ordered set. (The median is one which is at least as large as half the elements, and as small as the others).

Merging: Coalesce two ordered lists into one.

Minimum spanning tree: For a given graph G with edge weights, find the minimum-cost set of edges which connects all the vertices of G without forming any cycles. For the Euclidean version, the vertices are points in space and the edge weights are Euclidean interpoint distances.

Nonprimes: Determine whether a given integer is a composite number (i.e., can be factored).

Planarity of a graph: Determine whether a given graph can be drawn in the plane without any crossing edges.

Primes: Determine whether a given integer is a prime number.

Satisfiability: For a given Boolean expression, determine whether there is any assignment of values to the literals for which the expression is true.

Scheduling: Given n tasks and some constraints on the order in which they may be done, find the best sequence in which to perform them. There are many variations which involve number of processors, processing times, precedence, deadlines, and penalties for late completion.

Search: Given a set of n elements and a "key" element, determine whether the key element is in the given set. Many variations of this problem have been defined.

Selection: Find the kth-smallest element from a given list of n linearly ordered elements.

Set identity: Determine whether two given n-element sets are identical.

Set manipulation (UNION-FIND): A UNION operation forms the union of two sets; a FIND operation determines which set contains a particular element. Perform a sequence of n such operations.

Shortest path: Given a graph G with nonnegative edge weights, and a distinguished vertex v, find the minimum-weight path from v to every other vertex of G. A modified version of this single-source problem allows negative edge weights. The all-pairs version requires minimum-weight paths between all pairs of vertices of G.

Sorting: Arrange a list of n elements $\{x_i\}$ from a linearly ordered set so that $x_1 < x_2 < \cdots < x_n$.

Transitive closure: Given a Boolean matrix A, compute $A + A^2 + A^3 + \ldots$.

Traveling salesman: Given a graph G with nonnegative edge weights, find the minimum-cost cycle which passes through each vertex exactly once. For the Euclidean version of this problem, the vertices are points in space, and the edge weights are Euclidean interpoint distances.

ACKNOWLEDGMENTS

The author would like to thank Michael I. Shamos, H T Kung, Samuel H Fuller, Jon L. Bentley, and Richard M Karp for their helpful critiques, and Peter Denning for also suggesting the problem glossary.

REFERENCES

[1] AHO, A. V ; HOPCROFT, J. E ; AND ULLMAN, J D *The design and analysis of computer algorithms,* Addison-Wesley, Reading, Mass , 1974

[2] BAUDET, G *Numerical computations on asynchronous multiprocessors,* Carnegie-Mellon Univ , Dept Computer Sci., Pittsburgh, Penn , April 1976

[3] BEARDWOOD, J., HALTON, J. H.; AND HAMMERSLEY, J M. "The shortest path through many points," *Proc. Cambridge Philosophical Soc.* 55, 4 (Oct. 1959), 299-327.

[4] BENTLEY, J L , AND SHAMOS, M. I. *Divide and conquer for linear expected time,* Carnegie-Mellon Univ , Dept. Computer Sci., Pittsburgh, Penn., March 1977; to appear in *Inf. Process. Lett*

[5] BLUM, M.; FLOYD, R. W.; PRATT, V ; RIVEST, R. L , AND TARJAN, R E. "Time bounds for selection," *J Comput. Syst. Sci.* 7, 4 (Aug. 1973), 448-461.

[6] BORODIN, A. "Computational complexity theory and practice" in *Currents in the theory of computing,* A. V. Aho, [Ed.], Prentice-Hall, Inc., Englewood Cliffs, N.J., 1973, pp. 35-89.

[7] BORODIN, A.; AND MUNRO, I. *The computational complexity of algebraic and numeric problems,* American Elsevier, N.Y., 1975

[8] CHANDY, K M , AND REYNOLDS, P. F. "Scheduling partially ordered tasks with probabilistic execution times," in *Proc. 5th Symp. Operating Systems Principles,* ACM, N Y , 1975, pp 169-177

[9] COFFMAN, E. G. [Ed.] *Computer and job shop scheduling theory,* John Wiley and Sons, Inc., N.Y., 1976.

[10] COFFMAN, E. G.; AND SETHI, R. "A generalized bound on LPT sequencing," in *Proc. Int. Symp. Computer Performance Modeling, Measurement, and Evaluation,* ACM, N.Y., 1976, pp. 306-310.

[11] COHEN, J.; AND ROTH, M. "On the implementation of Strassen's fast multiplication algorithm," *Acta Inf.* 6, 4 (1976), 341-355.

[12] COOK, S. A. "The complexity of theorem proving procedures," in *Proc. 3rd Ann. ACM Symp. Theory of Computing,* ACM, N.Y., 1971, pp. 151-158.

[13] DAHLQUIST, G.; AND BJORCK, A. *Numerical methods,* Prentice-Hall, Inc., Englewood Cliffs, N.J., 1974.

[14] DIJKSTRA, E. W. "A note on two problems in connexion with graphs," *Numer. Math.* 1, (1959), 269-271.

[15] DOBKIN, D.; AND LIPTON, R., *On the complexity of computations under varying sets of primitives,* Tech. Rep no. 42, Yale Univ., Dept Computer Sci., New Haven, Conn., 1975.

[16] EDMONDS, J.; AND KARP, R. M. "Theoretical improvements in algorithmic efficiency for network flow problems," *J. ACM* 19, 2 (April 1972), 248-264.

[17] ERDOS, P.; AND SPENCER, J. *Probabilistic methods in combinatorics,* Academic Press, N Y , 1974.

[18] FISCHER, M. J.; AND MEYER, A. R. "Boolean matrix multiplication and transitive closure," in *Conf. Record, IEEE 12th Ann Symp Switching Automata Theory,* IEEE, N.Y., 1971, pp. 129-131.

[19] FLOYD, R.W. "Algorithm 245· Treesort 3," *Commun ACM* 7, 12 (Dec. 1964), 701.

[20] FLOYD, R. W.; AND RIVEST, R. L. "Expected time bounds for selection," *Commun ACM* 18, 3 (March 1975), 165-172.

[21] FORD, L. R.; AND JOHNSON, S. M. "A tournament problem," *Am Math. Monthly* 66, (1959), 387-389.

[22] FRAZER, W D. "Analysis of combinatory algorithms—a sample of current methodology," in *Proc. AFIPS 1972 Spring Jt. Computer Conf.,* Vol. 40, AFIPS Press, Montvale, N.J., pp. 483-491.

[23] FULLER, S. H ; GASCHNIG, J. G.; AND GILLOGLY, J. J *Analysis of the alpha-beta pruning algorithm,* Carnegie-Mellon Univ , Dept Computer Sci., Pittsburgh, Penn., July 1973.

[24] GAREY, M. R.; AND JOHNSON, D. S. "The complexity of near-optimal graph coloring," *J. ACM* 23, 1 (Jan. 1976), 43-49.

[25] GAREY, M. R.; AND JOHNSON, D. S. "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

[26] GRAHAM, R L. "An efficient algorithm for determining the convex hull of a finite planar set," *Inf. Process Lett.* 1, (1972), 132-133.

[27] GUIBAS, L. J.; AND SZEMEREDI, E. "The analysis of double hashing," extended abstract in *Proc. 8th Ann. ACM Symp Theory Computing,* ACM, N.Y., 1976, pp. 187-191.

[28] HOARE, C. A. R. "Quicksort," *Comput. J* 5, (1962), 10-15.

[29] HOPCROFT, J E. "Complexity of computer computations," in *Proc. IFIP Congress 74,* Vol. 3, North-Holland Publ. Co., Amsterdam, The Netherlands, 1974, pp. 620-626.

[30] HOPCROFT, J. E.; AND TARJAN, R. E. "Efficient algorithms for graph manipulation," *Commun. ACM* 16, 6 (June 1973), 372-378.

[31] HOPCROFT, J. E.; AND ULLMAN, J. D *Formal languages and their relation to automata,* Addison-Wesley, Reading, Mass , 1969

[32] HOPCROFT, J E.; AND ULLMAN, J. D. "Set merging algorithms," *SIAM J. Comput.* 2, 4 (Dec. 1973), 294-303.

[33] HYAFIL, L. "Bounds on selection," *SIAM J. Comput.* 5, 1 (March 1976), 109-114.

[34] HYAFIL, L.; AND KUNG, H. T "The complexity of parallel evaluation of linear recurrences," in *Proc 7th Ann. ACM Symp. Theory Computing,* ACM, N.Y., 1975, pp. 12-22.

[35] IBARRA, O.; AND KIM, C. "Fast approximation algorithms for the knapsack and sum of subsets problems," *J. ACM* 22, 4 (Oct. 1975), 463-468

[36] JOHNSON, D. B. "A note on Dijkstra's shortest path algorithm," *J. ACM* 20, 3 (July 1973), 385-388.

[37] JOHNSON, D. S. "Approximation algorithms for combinatorial problems," in *Proc. 5th Ann. ACM Symp. Theory Computing,* ACM, N.Y., 1973, pp. 38-49; also in *J. Comput. Syst. Sci* 9, 3 (Dec. 1974), 256-278

[38] JOHNSON, D. S. "Fast algorithms for bin packing," *J. Comput. Syst Sci.* 8, 3 (June 1974), 272-314.

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

[40] KARP, R M "On the computational complexity of combinatorial problems," *Networks* 5, 1 (Jan 1975), 45-68.

[41] KARP, R. M "The probabilistic analysis of some combinatorial search algorithms," in *Algorithms and complexity New directions and recent results,* J F. Traub, [Ed.], Academic Press, N.Y., 1976, 1-19.

[42] KLEINROCK, L. *Queueing systems, Vol. I: Theory,* John Wiley and Sons, Inc., N Y., 1975.

[43] KNUTH, D. E. *The art of computer programming, Vol. I· Fundamental algorithms,* Addison-Wesley, Reading, Mass., 1968

[44] KNUTH, D. E. *The art of computer programming, Vol. II· Seminumerical algorithms,* Addison-Wesley, Reading, Mass., 1969.

[45] KNUTH, D E "Mathematical analysis of algorithms," in *Proc IFIP Congress 71,* Vol 1, North-Holland Publ Co., Amsterdam, The Netherlands, 1971, pp. 135-143.

[46] KNUTH, D E *The art of computer programming, Vol. III Sorting and searching,* Addison-Wesley, Reading, Mass , 1973

[47] KNUTH, D. E. "Big omicron and big omega and big theta," *SIGACT News* 8, 2 (April/June 1976), 18-24.

[48] KNUTH, D. E.; AND MOORE, R W. "An analysis of alpha-beta pruning," *Artif Intell* 6, (1975), 293-326.

[49] KNUTH, D. E , MORRIS, J H.; AND PRATT, V R. *Fast pattern matching in strings,* Tech. Rep STAN-CS-74-440, Computer Sci Dept., Stanford Univ., Stanford, Calif., Aug. 1974, also Knuth, D. E.; Morris, J. H.; and Pratt,

V. R. "Fast pattern matching in strings," *SIAM J. Comput.* **6**, 2 (June 1977), 323-350.

[50] LADNER, R E. "On the structure of polynomial time reducibility," *J ACM* **22**, 1 (Jan. 1975), 155-171.

[51] LAWLER, E. L. "Algorithms, graphs, and complexity," *Networks* **5**, 1 (Jan 1975), 89-92.

[52] LIU, C L. *Introduction to combinatorial mathematics*, McGraw-Hill, N.Y., 1968.

[53] NEWBORN, M. M. "The efficiency of the alpha-beta search on trees with branch-dependent terminal node scores," abstract in *Algorithms and complexity. New directions and recent results*, J. F. Traub, [Ed.], Academic Press, N.Y , 1976, p. 483.

[54] O'NEIL, P. E ; AND O'NEIL, E J. "A fast expected-time algorithm for Boolean matrix multiplication and transitive closure," *Inf Control* **22**, 2 (March 1973), 132-138.

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

[56] REINGOLD, E M. "Establishing lower bounds on algorithms· a survey," in *AFIPS 1972 Spring Jt. Computer Conf*, Vol 40, AFIPS Press, Montvale, N J , 1972, pp 471-481.

[57] REINGOLD, E M "On the optimality of some set algorithms," *J ACM* **19**, 4 (Dec. 1972), 649-659.

[58] RIVEST, R. L.; AND VUILLEMIN, J "A generalization and proof of the Aanderaa/Rosenberg conjecture," in *Proc 7th Ann. ACM Symp Theory Computing*, ACM, N.Y., 1975, pp. 6-11.

[59] ROSENBERG, A L. "On the time required to recognize properties of graphs. a problem," *SIGACT News* **5**, 4 (Oct. 1973).

[60] SAHNI, S. "Computationally related problems," *SIAM J Comput.* **3**, 4 (Dec 1974), 262-279

[61] SAHNI, S. "Approximate algorithms for the 0/1 knapsack problem," *J ACM* **22**, 1 (Jan 1975), 115-124.

[62] SAHNI, S. *General techniques for combinatorial approximation*, Tech. Rep. 76-6, Dept. Computer Sci., Univ Minnesota, Minneapolis, Minn., June 1976.

[63] SCHONHAGE, A.; PATERSON, M.; AND PIPPENGER, N. "Finding the median," *J Comput. Syst Sci.* **13**, (Oct 1976), 184-199

[64] SEDGEWICK, R. "Quicksort," PhD Thesis, Stanford Univ., Stanford, Calif., Tech. Rep. STAN-CS-75-492, May 1975; for a summary see Sedgewick R., "The analysis of Quicksort programs," *Acta Inf.* **7**, 4 (1977), 327-355.

[65] SHAMOS, M I "Geometric complexity," in *Proc. 7th Ann. ACM Symp Theory Computing*, 1975, ACM, N.Y., pp. 224-233.

[66] SHAMOS, M. I ; AND HOEY, D. "Closest point problems," in *Proc 16th Ann. IEEE Symp. Foundations Computer Sci* , IEEE, N.Y , 1975, pp. 151-162.

[67] SHAMOS, M. I; AND YUVAL, G. "Lower bounds from complex function theory," in *Proc 17th Ann. IEEE Symp. Foundations Computer Sci.*, IEEE, N.Y., 1976, pp. 268-273.

[68] SLOANE, N. J. A. *A handbook of integer sequences*, Academic Press, N.Y., 1973

[69] SOLOVAY, R.; AND STRASSEN, V. "A fast Monte-Carlo test for primality," *SIAM J Comput* **6**, 1 (March 1977), 84-85

[70] SPIRA, P. M. "A new algorithm for finding all shortest paths in a graph of positive arcs in average time $O(n^2\log^2 n)$," *SIAM J. Comput.* **2**, 1 (March 1973), 28-32.

[71] STRASSEN, V "Gaussian elimination is not optimal," *Numer Math.* **13**, (1969), 345-356.

[72] TARJAN, R. E. "Depth first search and linear graph algorithms," *SIAM J. Comput* **1**, 2 (June 1972), 146-160.

[73] TRAUB, J. F. [Ed], *Complexity of sequential and parallel numerical algorithms*, Academic Press, N.Y., 1973.

[74] VALIANT, L. G "General context-free recognition in less than cubic time," *J. Comput Syst. Sci.* **10**, 2 (April 1975), 308-315.

[75] VALIANT, L. G "On non-linear lower bounds in computational complexity," in *Proc. 7th Ann ACM Symp. Theory Computing*, ACM, N Y , 1975, pp 45-53.

[76] WELLS, M. B. "Applications of a language for computing in combinatorics," in *Proc IFIP Congress 65*, Vol. 2, Spartan Books, Washington, D. C., 1965, pp. 497-498.

[77] WELLS, M. B *Elements of combinatorial computing*, Pergamon Press, Elmsford, N.Y., 1971.

[78] WILLIAMS, J. W. J "Algorithm 232: heapsort," *Commun ACM* **7**, 6 (June 1964), 347-348.

[79] YUVAL, G personal communication, 1975.

RECEIVED NOVEMBER 1976; FINAL REVISION ACCEPTED AUGUST 1977