A Survey of Analysis Techniques for Combinatorial Algorithms

Bruce Weide

September 1976

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

Abstract - 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 explained. A brief explanation of the problem classes P and NP, as well as the class of NP-complete problems, is also presented.

Acknowledgements - The author would like to thank Profs. S.H. Fuller, H.T. Kung, and M.I. Shamos for their helpful suggestions.

I. Introduction

A. MOTIVATION

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 [1974] 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 (speech understanding, picture processing, compiling, etc.) is analyzed at a high level and design decisions are finalized, algorithmic decisions must be made in order to implement the design 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 [1968, 1969, 1973] are unsurpassed for completeness. Aho, Hopcroft, and Ullman [1974] is an excellent text with many examples of both design and analysis but, like Knuth, does not provide an overview of the area.

For one who does not have the months or years to spend on details, however, but wants simply an overview of techniques and a list of some important results, the literature is sparse. The articles by Knuth [1971], Reingold [1972], and Frazer [1972] provide some relief, but concentrate heavily on presenting the detailed analysis of one or two example algorithms. Also, new results since 1972 are abundant and are obviously not contained in those papers. A fine paper by Borodin [1973] treats primarily the theoretical concepts of computational complexity, reviewing the definitions and properties of complexity classes for various automata.

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 notion of an algorithm and with the nature of some of the important discrete (combinatorial) problems: sorting, searching, graph problems, discrete optimization problems, etc. For those who are not, Knuth [1968, 1973] and Aho, Hopcroft, and Ullman [1974] provide explanations. These sources should in fact be referenced at practically every major point; to avoid this inconvenience, the default references are to the three volumes of Knuth [1968, 1969, 1973] and to Aho, Hopcroft, and Ullman [1974].

B. THE NOTION OF COMPLEXITY

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

-Voltaire

I hate definitions.

-Benjamin Disraeli

Numerous questions immediately come to mind regarding the very definition of complexity, which can be described as a function mapping problem size into the time required to solve the problem. (Note that we have already limited ourselves to consideration of time complexity, ignoring questions of the amount of space required, comprehensibility of the algorithm, and the literally scores of other factors which must be taken into account when designing an algorithm. Time complexity, while important, is certainly only one dimension of algorithm-space.) Among these questions, which are discussed in this section, are the notions of upper and lower bounds on complexity, models of computation, and problem representation. Clearly, for any measure of problem difficulty to be precise enough to be useful, a particular model of computation (including permissible operations and their associated costs) and a problem representation (making a definition of problem size possible) must be specified unambiguously.

DOES COMPLEXITY PERTAIN TO A PROBLEM OR TO AN ALGORITHM?

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 sorting takes time at least L(n).

On the other hand, we might also like to know how long it could take us to sort such a list 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). Such an upper bound can be thought of as being associated with the problem, but is most often proved by demonstrating an algorithm which solves the problem in at most U(n) time. Hence, algorithms are normally analyzed primarily to determine their worst-case behavior.

One way of seeing the distinction between lower and upper bounds is to note that L(n) is the minimum (over all possible algorithms) of the maximum complexity (over all inputs), whereas U(n) is the minimum (over all known algorithms) of the maximum complexity (over all inputs). In trying to prove 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), it is the ultimate goal in computational complexity to make these two functions coincide and provide a single complexity measure for the problem class by finding an "optimal" algorithm; however, for most of the problems we will mention, this remains a goal and is not yet a reality.

DOES COMPLEXITY HAVE TO REFER TO THE WORST CASE, OR COULD IT MEASURE THE AVERAGE CASE?

Traditionally, the worst-case complexity has been of major theoretical interest for the reasons just cited. Recently, however, there has been greater effort in the analysis of the behavior of algorithms "on the average", since (intuitively, at least) this seems to provide the kind of information which would be more useful in the application of complexity results to real-world problems. For example, the simplex algorithm for linear programming is known to perform miserably (require 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.

There are problems with this approach, however. This first is simply that averaging over many cases complicates the analysis considerably. Secondly, 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 too often neglected in practice. And of course there is the biggest objection of all: the typical assumptions which must be made regarding the probability distribution over all possible inputs (usually simple ones to make the analysis tractable) are normally either grossly unrealistic or even meaningless. Despite these objections, which will be covered later in more detail, work on average-case behavior has continued to expand to the point where it has now made its mark on the design of algorithms for very hard problems, as well as on algorithmic analysis.

DOES COMPLEXITY MEASURE THE NUMBER OF STEPS ON A TURING MACHINE, OR THE NUMBER OF SECONDS ON AN IBM-370/195?

Again, 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", and except for the most theoretically-oriented results, Turing complexity is not appropriate. Likewise, a particular machine like the IBM-370/195 is probably a bad choice because it makes the analysis even more complicated than it needs to be for most purposes (although Knuth [1968, 1969, 1973] defines his own machine and proceeds to derive functions giving the run times for some particular implementations of algorithms on that machine). In choosing a model of computation, we try to achieve a balance between realism and mathematical tractability, and this trade-off typically results in both being short-changed. However, in order to get any results whatsoever, this compromise must be made; experience indicates that useful results can be obtained despite what appear to be overly simplified computational models.

What is more appropriate than finding exact run times on particular machines (since it factors out many implementation details) is a model of computation in which only certain important "elementary" operations are counted, and the complexity is reported to be $O(f(n))$, read "of order $f(n)$". Mathematically, $g(n)$ is $O(f(n))$ if there is a constant $c>0$ such that as $n\rightarrow\infty$, $g(n)/f(n)\rightarrow c$. Intuitively, the number of elementary operations required, $g(n)$, is approximately some constant times $f(n)$ for large values of n. Other definitions are also found in the literature. Knuth [1968] uses $g(n) = O(f(n))$ to mean that $cf(n)$ is an upper bound on $g(n)$ for all sufficiently large values of n. What has been called "big-O" notation here is one instance of what Knuth [1976] would call "big-theta" notation, which means that $g(n)/f(n)$ is bounded above by some constant and is also bounded below by some constant, as $n\rightarrow\infty$. The reason for using this variation rather than the version which only describes an upper bound is that it contains more information, and that information can be useful in comparing algorithm behavior. Also, it is inappropriate to bound a lower bound from above, which is what Knuth's definition of $O(f(n))$ does.

Consider as an example the problem 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 non-trivial, even while loop control, pointer management, etc. remain easy. Another natural choice is the "uniform cost criterion", where memory references, comparisons, additions, etc. all take unit time. If the operation which must be done most often (as n gets large) is chosen, then counting auxiliary operations would only change the constant c and would not affect the asymptotic behavior of the algorithm.

Normally, the choice of operations to count is not crucial to the analysis, assuming that the dominant one is among them. However, there are examples of extremely clever algorithms which almost surely resulted from careful consideration (by the algorithm designer) of which operation is most important. One particularly instructive case is Strassen's algorithm for multiplying $n \times n$ matrices (see Strassen [1969]). He shows how to multiply two 2×2 matrices, with elements from an arbitrary ring, using 7 multiplications and 18 additions (as opposed to the obvious method using 8 multiplications and 4 additions). His algorithm for multiplying two n x n matrices, where $n = 2^k$, begins by partitioning each of the original matrices into four sub-matrices of size $2^{k-1} \times 2^{k-1}$ (to which the algorithm is applied recursively), then multiplying the 2 x 2 matrices (which have $2^{k-1} \times 2^{k-1}$ matrices as elements) using the 7 multiplication, 18 addition algorithm.

At first glance, this looks like a losing proposition on real machines (if we count addition and multiplication as elementary operations), since the obvious method for multiplying 2 x 2 matrices uses only 12 operations as opposed to 25. This is a very bad choice of elementary operations, though, because the multiplications and additions are of matrices, and matrix multiplications really do cost more. A careful analysis (covered later) reveals the trade-offs involved, and leads to the conclusion that the total number of operations is not $O(n^3)$ as in the obvious method, but rather $O(n^{\log 7}) = O(n^{2.81})$ (all logarithms in this paper are base-2). This is true even though the obvious algorithm takes significantly fewer elementary operations for the case of 2 x 2 matrices of scalars.

The main point is that even while the end result, that the algorithm is $O(n^{\log 7})$, applies whether we count scalar multiplications or both additions and multiplications, the key to understanding the algorithm's efficiency (and to designing it in the first place!) is realizing that multiplication of matrices is much more costly than addition of matrices. Therefore, because the algorithm is applied recursively to matrices whose elements are matrices, for sufficiently large values of n we should be quite willing to perform many more additions in order to save just one multiplication. Except for that crucial observation, one would be very unlikely to even try to find a new algorithm for multiplying 2 x 2 matrices.

HOW IS THE "SIZE" OF A PROBLEM MEASURED?

The measure of problem size is another vague concept. It can 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. We must be careful that clever encoding tricks are not used which drastically affect our results, however. For example, if we encode integers in binary representation for an algorithm which then takes $O(2^n)$ steps (where n is the number of bits used to represent the input), the same algorithm would require only O(n) steps if we could encode the input in unary notation (since a number which could be represented using k bits in binary requires about 2^k "marks" in unary). For this reason, along with the fact that in practice we use anything but unary notation, the latter representation is not appropriate.

In most cases, though, the problem size is simply some natural measure which, of course, 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 in the list; 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 results which are to be of interest in real-world situations, 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 in all but primarily theoretical analyses. Aho, Hopcroft, and Ullman [1974] contains a good account of the similarities and differences between these two models of computation.

II. Lower Bounds

... abounding in intuitions without method...

-George Santayana

It is generally agreed that 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 conceivable algorithm, including undiscovered ones. Still, a few techniques have shown themselves to be applicable to many problems, and others seem to have promise.

A. TRIVIAL LOWER BOUNDS

Among those in the former category is the most obvious (and also the weakest) method, which 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 for solving the problem must do enough work to accomplish these tasks.

There are many examples of the use of such a technique. One interesting graph problem is Dijkstra's [1959] single-source shortest path problem. Given a directed graph G with non-negative edge weights, and a distinguished vertex v, find the minimum-weight path from v to each other vertex of G. A more interesting variation allows negative-weight edges but no 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 "look at" each of them. If some edge were ignored by any algorithm, we could change its weight so that a shortest path was missed and force the algorithm to give a wrong answer, so there are inputs which require $O(n^2)$ time for any algorithm to solve the modified single-source shortest path problem.

Similarly, multiplication of a pair of nxn matrices requires that n^2 outputs be produced, and is therefore at least $O(n^2)$. Notice that this says nothing about the number of multiplications, for example, required to solve the problem, but only that some operation must be performed at least $O(n^2)$ times; therefore, the dominant operation must be performed at least that many times.

Generally speaking, trivial lower bounds are easy to come by and, therefore, of less interest than sharper bounds which can sometimes be proved by more sophisticated methods. However, trivial bounds are often the only lower bounds available, and because they are usually easy to prove, should probably be tried before less obvious techniques or tricks are applied.

B. INFORMATION-THEORETIC BOUNDS

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" is used in the technical sense). 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).

The most celebrated example of a lower bound from information theory is for the problem of sorting a list of elements from a linearly ordered set (see Knuth [1973], who uses a "decision tree" model which is based on the same argument). For a list of n items, there are n! possible permutations. If an algorithm would sort any of them (or equivalently, identify the original permutation), it must perform at least $\lceil \log n! \rceil$ comparisons. Using Stirling's approximation to n! before taking the logarithm gives a lower bound of O(n log n) for the sorting problem.

Application of this technique to the problem of merging two ordered lists from a linearly ordered set gives a lower bound for that problem as well.

C. ORACLES

Knuth [1973] points out that a better bound can be obtained for the merging problem by another technique which he calls the construction of an "oracle". An oracle is a fiendish enemy of an algorithm which at every opportunity tries to make the algorithm do as much work as possible. In the case of merging the two lists $A_1 < A_2 < \ldots < A_n$ and $B_1 < B_2 < \ldots < B_n$ by any comparison-based algorithm, 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.

Now if comparisons are resolved by this oracle, merging must end with the configuration:

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

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:

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

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.

It may be argued that this bound is not very interesting, since it is less than the trivial bound of 2n. However, as a bound on the number of comparisons it is not so trivial, and it serves to illustrate the technique. Also, comparisons may take much longer than bookkeeping operations in some applications, and the trivial bound simply says that <u>something</u> must be done 2n times. Asymptotically, this distinction is of no consequence, but for practical values of n it may be quite important. Hyafil [1976] has used an oracle to prove a lower bound for the selection problem (finding the k^{th} largest of n elements), where the trivial bound is simply n. In this case, both upper and lower bounds are known to be O(n), and the oracle provides a way of refining the lower bound to permit comparison with precise upper bounds.

D. 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 could be used to solve another problem P_2 for which a lower bound is known. This means that P_1 can be solved no faster than P_2 and provides a lower bound on P_1, provided that an instance of P_1 can be mapped into an instance of P_2 at least as fast as P_2 can be solved. The power of this approach is substantial.

Shamos [1975] uses problem reduction to show that an algorithm for finding the convex hull of n points in the plane could be used to sort on one of the coordinates and therefore must take time at least O(n log n). Other celebrated applications include the reduction of context-free language recognition to matrix multiplication (see Valiant [1975]); the mutual reductions between boolean matrix multiplication and transitive closure (see Fischer and Meyer [1971]); and the relationship between integer multiplication and the discrete Fourier transform (see Schonhage and Strassen [1971]).

Many other examples of this technique are found in transformations between so-called NP-complete problems (see section V). Note that it is not always too clear how to identify the problem P_2, which is of course a requirement for using this approach.

E. OTHER TRICKS

Among the newer approaches for proving lower bounds is the use of graph models of algorithms. Kung and Hyafil [1975] show trade-offs between the depth and breadth of trees describing the parallel evaluation of arithmetic expressions to show that the possible speed-up using k processors is bounded by (2k+1)/3. This result is counter to the intuition that having k processors available would allow a speed-up of k. In fact, for certain computations (such as adding up a list of k numbers) the speed-up is even less, in this case 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, even though the prospect of inexpensive parallel hardware compels us to study such algorithms.

Valiant [1975] uses a similar approach to find various non-linear lower bounds by concentrating on graph-theoretic properties. Lawler [1975] expresses confidence that such arguments will continue to prove useful in demonstrating lower bounds, and recent results show this optimism to be well-founded.

Another new approach is the use of theorems from complex analysis by Shamos and Yuval [1976] to show that finding the mean distance between n points in the plane requires $O(n^2)$ square-root operations. Their proof is based on the ambiguity of the square root function. The primary significance of this result is that it had previously been almost impossible to obtain lower bounds except for the four common arithmetic operations and comparisons, whereas the new approach applies to any multiple-valued function (such as square root, inverse trigonometrics, logarithms, etc.).

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, non-trivial results and general techniques are quite sparse.

III. Upper Bounds

Method is good in all things. Order governs the world.

-Jonathan Swift

In contrast to the lack of methods for proving lower bounds, there are two widely used ways of proving upper bounds by analyzing the worst-case behavior of an algorithm. One technique (the use of recurrence relations) sometimes dominates the other, but it is usually so easy to simply count instructions that this approach is considered separately. A third alternative, the application of brute force, has been known to work in at least one instance, but has little else to recommend it and is dealt with only briefly.

A. IDENTIFYING A WORST CASE AND COUNTING STEPS

Since we seek an upper bound on a problem, and our approach is to demonstrate that a particular algorithm to solve the problem never takes more than $U(n)$ time, the first task is to identify a "worst case"; i.e., an input of size n which requires the algorithm to do as much work as any other input of the same size.

In some cases, this is not difficult, because the algorithm may do the same amount of work for every input of size n. This phenomenon is easily recognized by inspection of a description of the algorithm in which the flow of control is clearly stated. If this flow does not depend on the data, then every case is a "worst" one. For example, the obvious algorithm for finding the largest 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 nxn matrices in the classical way takes $O(n^3)$ 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 a worst case.

However, sometimes there are data-dependent decisions which affect the flow of control and make the job of finding a worst case slightly more difficult. Search algorithms have this feature; for example, the binary search algorithm looks through a sorted array consisting of n elements for the position of a particular "key" item which is known to be in the array:

```
procedure binsearch(A,key);
   begin
      compare key to middle element of A
         doing   < : return(binsearch(first half of A,key));
                 > : return(binsearch(last half of A,key));
                 = : return(pointer to middle of A)
   end;
```

It is not too difficult to see that a worst case input is one for which each comparison results in another recursive call to the procedure, and we only find the key when the array has been narrowed down to just one element. This takes $\lfloor \log n \rfloor + 1$ comparisons (which is essentially optimal, since a lower bound of $\lfloor \log n \rfloor$ is obtained from information-theoretic arguments).

A worst case which is only slightly harder to manufacture is one for quicksort, an ingenious sorting algorithm which was proposed by Hoare [1962]. The algorithm is very simple to describe:

```
procedure quicksort(S);
   begin
      if $|S| \le 1$ then return(S);
      choose some element x from S;
      partition S into those 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;
```

Note that if all elements of S are distinct and the algorithm is unlucky enough to pick 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. Using recurrences, as in the next subsection, it is found that quicksort requires $O(n^2)$ time in this case. This is not optimal, since sorting algorithms which never require more than O(n log n) steps are known, and the lower bound is O(n log n). An 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.

For more complex algorithms, particularly those for graph problems and discrete optimization problems, finding a worst case can be more difficult. An interesting case in point is the modified single-source shortest path problem. In an article on global flow analysis, Edmonds and Karp [1972] mention in passing that a modified version of Dijkstra's [1959] algorithm runs in $O(n^3)$ time for any directed graph satisfying the conditions of the modified problem (see section II). Their one-sentence justification is convincing to most, but D.B. Johnson [1973] shows an entire family of directed graphs which require $O(n2^n)$ time! All of which demonstrates that even the most respected people in the field can be misled by faulty identification of the worst case.

B. 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 $U(n)$ be denoted by $T(n)$; "T" is for "time to solve the problem". Then it is usually 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 is 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:

$$T(n) \leq T(n/2) + 1 \qquad \text{for } n > 1$$

$$T(1) = 0$$

Again, the initial condition is zero because by hypothesis the key item is in the array A, and if A consists of just one element it must be the key. The recurrence is discovered by recognizing that in the worst case, one comparison is used to determine which remaining half to search recursively, so the total number of comparisons $T(n)$ is the sum of this comparison and the number $T(n/2)$ required to find the key in an array essentially half as large. Because n may be odd, the relation is not exact, hence the use of "≤" rather than "=". The remaining part can never be larger than $n/2$ whether n is odd or even.

Solving for $T(n)$ and noting that the number of comparisons must be an integer gives $T(n) \le \lfloor \log n \rfloor$, which is optimal. The particular implementation of binary search discussed in the previous subsection makes a redundant comparison for $n=1$, which causes the slight discrepancy between this result and the one obtained for that implementation.

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 $P(n) = n-1$, so:

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

$$T(1) = T(0) = 0$$

for which the exact solution is $T(n) = n(n-1)/2$. Of course, it is apparently not too intelligent to choose x arbitrarily if a worst case of $O(n^2)$ must be avoided. Rather, x should partition S into approximately equal parts (Aho, Hopcroft, and Ullman [1974] call this the "principle of balancing"). This can be accomplished by choosing x as the median element of S, whereupon the recurrence becomes:

$$T(n) \le 2T(n/2) + P(n) + C(n) \qquad \text{for } n > 1$$

$$T(1) = T(0) = 0$$

As in the case of binary search, "≤" replaces "=" because n may be odd, and $T(n)$ still provides an upper bound.

Now, $P(n) = n-1$ as before, but $C(n)$ is no longer zero but the number of comparisons necessary to find the median of n elements. Blum, et. al. [1973] present an algorithm which finds the median in at most $5.43n$ comparisons, and Hyafil [1976] reports that Paterson, Pippenger, and Schonhage have an algorithm which uses at most $3n$ comparisons. Taking $C(n) = 3n$:

$$T(n) \le 2T(n/2) + 4n - 1 \qquad \text{for } n > 1$$

$$T(1) = T(0) = 0$$

for which the solution is $T(n) \le 4n \log n - n + 1$, so that $T(n)$ is $O(n \log n)$ in the worst case.

As a final example, consider Strassen's [1969] algorithm for matrix multiplication. In order to multiply two nxn matrices (for n a power of two; if it is not, embed the original matrices in ones with n equal to the next higher power of two), the algorithm performs 7 multiplications of 2x2 matrices (recursively) and 18 additions of (n/2)x(n/2) matrices. Assuming that the matrix additions take $(n/2)^2$ scalar additions, the recurrence is:

$$T(n) = 7T(n/2) + 18(n/2)^2 \qquad \text{for } n > 1$$

$$T(1) = 1$$

Here, all scalar multiplications and additions are counted as elementary operations, and the initial condition is obvious because multiplication of two 1x1 matrices consists of a single scalar multiplication. The solution is $T(n) = 7n^{\log 7} - 6n^2$, so T(n) is $O(n^{\log 7})$ compared to $O(n^3)$ for the classical method. Because of the factor of 7, though, the classical algorithm (which takes $2n^3 - n^2$ operations) is still faster for n less than about 600. 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=38 (see Spiess [1974]).

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 $O(n \log^2 n)$, which means that our guess is $cn \cdot \log^2 n$, resulting in:

$$\begin{aligned} T(n) &= 2T(n/2) + n \log n \\ &= cn \log^2(n/2) + n \log n \\ &= cn \log^2 n + O(n \log n) \end{aligned}$$

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

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 $an \cdot \log^2 n$ 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 $bn \cdot \log n + cn$, 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 [1968], Liu [1968], and Kleinrock [1975] 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, Dahlquist and Bjorck [1974] 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 [1973].

C. BRUTE FORCE

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, has only recently been attempted. Despite lacking finesse and thereby appealing to some, it has really only proven to be useful in one rather minor instance which is primarily a curiosity.

The question of how to sort using a minimum number of comparisons is considered in detail by Knuth [1973], who points out that the merge-insertion algorithm of Ford and Johnson [1959] 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 [1965] by using brute force computing power to exhaustively demonstrate that no algorithm could sort 12 items using fewer than 30 comparisons, and so 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, by running an algorithm on every possible input, to prove upper bounds. Such an approach is not to be recommended so long as there remain useful ways to consume computer time.

IV. 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 behavior on the average; i.e., finding the complexity of a computation averaged over some distribution of inputs. Generally, the same 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 applied.

A. 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", a better 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 to a considerable extent. If this were the only objection to doing average-case analysis, all that would be required would be more sophisticated tools, and this section could dwell on those. However, more serious questions have been raised which tend to cast considerable doubt on the entire adventure; this is the reason for not going into a 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 there is typically no way to identify the probability distribution over all problem occurrences. While it may be reasonable in some situations to assume that every possible problem is equally likely (such as assuming 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), this assumption really only makes sense if the problem space is finite. It apparently 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 in some situations.

In one attempt to answer this objection, Yuval [1975] has suggested that algorithms might "randomize" their inputs in order to make the assumption appear valid. He has pointed out that with suitable random steps 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 Rabin [1976]). For example, quicksort could choose the partitioning element randomly (this idea was offered by Hoare [1962] in his original paper on quicksort). Even though this might seem like a case of the tail wagging the dog, there is some justification for such an approach in this case. In order to make the analysis of the algorithm tractable, Sedgewick [1975] assumes that the files to be sorted are random, but presents evidence that the algorithm works better if they are!

Clearly, not all algorithms can be modified in this manner. The key element in a binary search can not 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. The best hope is that if there is some reason to believe that certain keys are more likely, some kind of balanced tree should be used in place of the sorted array as the data structure for holding the list elements, and we are therefore analyzing the wrong algorithm.

There is some evidence that despite the shaky basis for the assumption of random inputs, the results of analysis may not be extremely sensitive to this assumption, so long as the distribution over inputs is not too far from random. Also, no other distribution is likely to yield to analysis at all, owing to the difficulty of dealing with recurrences that result from such alternative assumptions. Any other distribution would undoubtedly have as little as or even less basis than the uniform, in any event. The justification seems to be: "Better some kind of average than none at all".

The objections do not stop there, however. Knowing the behavior of an algorithm on the average provides some information, but it would bolster our confidence in the result if we also knew the variance. Few attempts at average-case analysis take this next step and find higher moments of the solution time. One notable exception is Sedgewick's [1975] analysis of quicksort (see also Knuth [1973]), in which he shows that while the average number of comparisons is about $2n \log n$, the standard deviation is approximately $.68n$, so that our confidence that the algorithm will work efficiently grows with n. This is clearly a nice property for an algorithm to possess and contributes to the explanation of why quicksort works so well in practice. Unfortunately, such attempts at this more thorough analysis are rare.

One further step, which to the author's knowledge has not been explored, is to characterize the distribution of solution times by making use of the underlying mechanisms which govern the operation of the algorithm. For example, if an algorithm can be considered to consist of many separate tasks, no one of which dominates the running time (in a sense which must be made explicit; see Feller [1968], for example), then under certain conditions the central limit theorem could be applied to show that the distribution of solution times is approximately normal. This is true for any distributions of times to complete the subtasks which satisfy rather weak conditions, so that the randomness assumption is not as critical. If we are willing to make that assumption, finding the mean and variance by the usual methods would then complete a fairly good description of the algorithm's behavior.

B. SOME EXAMPLES OF AVERAGE-CASE ANALYSIS

There is a large and growing number of algorithms which have been subjected to analysis of average-case complexity. Only a few will be discussed here, but they are among the most important. Many others can be found in the two "standard" references.

What is surely the most comprehensive analysis of any algorithm is presented by Sedgewick [1975] in his Ph.D. thesis entitled "Quicksort", which tells everything you always wanted to know about quicksort (and much, much more that you didn't). His analysis of the average number of comparisons, exchanges, partitioning stages, etc. is carried out through the use of recurrences, just as for the simple worst-case analysis of the number of comparisons described in section III. Along with Appendix B of the thesis, a good reference for the kinds of techniques used by Sedgewick is Knuth [1968, 1973]. The recurrences are solved by standard methods described in both places, and look more complicated than they are, due in part to the large number of symbols needed to represent the important quantities using their notation.

All recurrences are not easy to solve, however. In an analysis of radix exchange sorting, Knuth [1973] uses properties of the gamma function and complex variable theory to derive asymptotic results from what looks like a fairly simple recurrence for which the usual techniques fail.

One algorithm which has been analyzed in at least three different ways is the alpha-beta search algorithm for game trees. Good descriptions of the algorithm can be found in Fuller, Gaschnig, and Gillogly [1973] and in Knuth and Moore [1975]. The former authors 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 proceed to derive expressions for the probability of expanding individual nodes, and the expected number of bottom positions evaluated. 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 $O(N^{.72D})$.

Knuth and Moore [1975] make the same assumptions about the tree and the random ordering of leaf-node values, and show an upper and lower bound on the average behavior of the algorithm of $O((N/\log N)^D)$. They suggest that the simulation results by Fuller, et. al., result in a fit to $N^{.72D}$ because N is so small.

Newborn [1976] uses a model in which the branch (rather than the node) values are randomly ranked and obtains even different results, but only for the cases D=2,3,4. This is yet another example of the dependence of results on the assumptions of the model being used in the analysis: the three different methods for D=2 give complexities of $O(N^{1.44})$, $O((N/\log N)^2)$, and $O(N \log N)$, respectively. Of course, the first is an estimate and the second a bound using the same model, whereas the third is different because the model is different.

Other examples of average-case analysis include Guibas and Szemeredi [1976] on double hashing, O'Neil and O'Neil [1973] on boolean matrix multiplication, and Knuth [1971] and Floyd and Rivest [1975] on selection.

V. Approximation 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 previously mentioned. 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, most optimization and graph problems are "hard" (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 which find approximate solutions to hard problems has been developed. Along with these algorithms have come new measures of "goodness" and associated techniques for designing and analyzing approximation algorithms. Two classes of approximation (guaranteeing a near-optimal solution always, and producing an optimal or near-optimal solution "almost everywhere") are discussed.

A. PROBLEM CLASSES AND REDUCIBILITY

For the moment it is necessary to return to the formal setting of Turing machines (TM) and languages to define a couple of important concepts. It is somewhat artificial, but convenient, to state a problem in terms of a language recognition task by formulating it so that it has a yes-no solution (for example, "does this traveling salesman problem have a solution with cost less than k?"), and then asking a TM to accept the input if the answer is yes and to reject it if it is not. A TM to solve the problem then accepts only input strings from some language L which consists of precisely those problem instances with "yes" answers. It is appropriate to refer to the original problem and the language L more or less interchangeably in the context of the classes P and NP (see Aho, Hopcroft, and Ullman [1974] for more details).

Formally, the class P (for "polynomial") is the set of languages L for which each string $x \in L$ is either accepted or rejected by a deterministic TM in a number of steps which is bounded by a fixed polynomial in the length of x (i.e., in "polynomial time"). Similarly, the class NP (for "nondeterministic polynomial") is the set of languages L for which each $x \in L$ is either accepted or rejected by a nondeterministic TM in polynomial time.

Clearly, $P \subseteq NP$ 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 [1972, 1975]). 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 which is an exponential function of the problem size.

Fortunately, it is sufficient to consider solving these problems in polynomial time on a normal random-access computer rather than on a TM, and the problems in P remain the same (see Aho, Hopcroft, and Ullman [1974]). Roughly speaking, problems which 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 ever 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 $x \in L_1$ iff $f(x) \in L_2$. The key to the argument that $P \neq NP$ is a remarkable theorem by Cook [1971] which says that every problem in NP can be polynomially reduced to boolean satisfiability (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) which converts it to an instance of satisfiability, and then solving the resulting satisfiability problem. A general-purpose problem like satisfiability is called "NP-complete", "P-complete", or simply "complete".

Karp [1972] shows reducibilities among other 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 represents the basis for believing (even if not being able to prove) that $P \neq NP$, since none of the hundreds of algorithms for the scores of problems which are NP-complete runs in polynomial time. If any of these problems could be solved fast, all of them could be; the fact that so far none of them can be is a convincing argument (but not a proof) that they never will be.

The question of whether P=NP is not the only open question to be answered, however. Some problems are still unclassified. For example, deciding the equivalence of regular expressions and context-sensitive language recognition are at least as hard as any problems in NP, but are not known to be in NP themselves; they (along with the NP-complete problems) are called "NP-hard" since their inclusion in P would imply that P=NP. On the other hand, graph isomorphism (which is clearly in NP) has not been shown to be in P, but also has not been shown to be NP-complete, so there may be problems which require non-polynomial deterministic time but whose solutions will not help solve the other hard problems in NP. While the latter possibility would hardly have been thought unlikely only a few years ago, the discovery that practically all hard problems in NP are in fact NP-complete makes the existence of such a problem of "intermediate" difficulty seem less plausible.

B. DESIGN AND ANALYSIS OF "GUARANTEED" APPROXIMATION ALGORITHMS

Since all known algorithms for solving NP-complete problems exactly require exponential time, it is impractical to solve very large instances of these problems. Happily, many real-world applications which require solutions to optimization or graph problems do not require exact solutions, but only approximately correct solutions. As a result, many researchers have developed new approximation algorithms (or are now analyzing old approximation schemes) for these applications, which attempt to guarantee near-optimal solutions to all instances of a problem.

A generally accepted measure of the efficacy of this type of approximation algorithm (see D.S. Johnson [1973]) is the maximum relative error introduced by the algorithm; normally, this error appears along with the problem size in the complexity. Most often, the quantities to be compared to determine the relative error are obvious: the objective function value for an integer program, the number of colors required for graph coloring, and so forth. Occasionally (for example, with the satisfiability problem) some artificial problem must be created so that the original one looks like some kind of optimization. The ratio of the absolute error to the exact solution value is given the symbol ϵ.

In designing approximation algorithms, one is, in this case, 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 follow the same patterns as are 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 is within a factor of $1\pm\epsilon$ of being optimal. Recognizing this takes much of the mystery out of what can appear to be very complicated algorithms.

Because many algorithms are designed in this fashion, their analysis is often "built-in", and is straightforward. Sahni [1976] gives a very nice summary of the general techniques in this class, dividing the methods into three categories: digit truncation (the most complicated; see, for example, Ibarra and Kim [1975]), interval partitioning, and separation. These are especially useful for the knapsack problem, packing problems, and certain scheduling problems. They are particularly attractive from the point of view of analysis, having "templates" for complexity analysis as a part of the design. The worst-case complexities they produce are usually like $O(n^c/\epsilon)$ for some constant c which depends on the problem.

An example of the interval partitioning technique is a simple algorithm for solving the 0/1 knapsack problem:

$$\text{maximize: } c \cdot x$$

$$\text{subject to: } a \cdot x \leq b$$

$$x \in \{0,1\}^n$$

Here, c and 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 some of the n items in such a way that the total utility cx is maximized while the capacity constraint $ax \leq b$ is satisfied. Item i is to be included iff $x_i = 1$.

The problem can be solved by a straightforward tree search method. Clearly, for any partial assignment to the variables $x_1, x_2, \ldots x_i$ 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 $ax \leq b$, and a partial assignment at level i consists of the fixed variables x_1 through x_i, with the remaining ones set to 0. It is easy to see that if a partial assignment is not feasible, then no completion of it will be feasible, and 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 constructed 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 $\epsilon>0$.

One interesting feature of the NP-complete problems is that not all seem to be equally well suited to approximation, even though they are in some sense of equivalent difficulty to solve exactly. D.S. Johnson [1973] points out that for some problems (those for which Sahni's techniques are applicable) 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 $O(n^c)$ for some $c>0$. Garey and Johnson [1976a] show that approximating the chromatic number of a graph to within a factor of 2 is NP-hard; 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 that if there is a polynomial-time algorithm for approximating the size of the maximum clique of a graph to within <u>some</u> bounded ratio, then there is a polynomial-time algorithm for approximating it to within <u>any</u> bounded ratio. Despite being of comparable complexity for exact solution, the NP-complete problems are not all equally easy to approximate.

For more examples of approximation algorithms and analysis of errors and complexities, see also Johnson [1974], Sahni [1975], and Coffman and Sethi [1976], among many others. Garey and Johnson [1976b] is a fine annotated bibliography for this area. Also of interest is an algorithm by Shamos and Yuval [1976] for approximating the solution to a problem which is relatively easy (the mean distance in the plane); the problem requires and can be solved in $O(n^2)$ time exactly. Their algorithm guarantees an ϵ-approximation in $O(n)$ time.

C. PROBABILISTIC BEHAVIOR OF APPROXIMATION ALGORITHMS

As mentioned above, it is not easy to guarantee good approximate solutions to certain NP-complete problems. These negative results suggest that finding an alternative to the "guaranteed" approximation approach is warranted. One such scheme which looks promising is to, effectively, make such a sophisticated guess about the solution that the likelihood of being wrong is negligibly small. These ideas can be made explicit; define p_n as 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 $p_1+p_2+\ldots$ is finite.

Optimality or near-optimality "almost everywhere" is a strong condition. Suppose that we randomly chose one problem instance of each size n, for $n=1,2,\ldots$, and ran 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 [1976] gives some examples of 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 [1959] 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. This theorem is, in turn, based on an application of the central limit theorem.

This example of the use of results from geometry and probability is typical of the nature of design and analysis of probabilistic algorithms. Results and techniques from a large number of fields of mathematics seem to apply in this area. For example, much pertinent material is found in the theory of random graphs (see Erdos and Spencer [1974]), where one finds many theorems regarding connectedness, maximum cliques, chromatic numbers, etc. Rabin [1976] presents a very fast algorithm for testing whether a number is prime, and for which 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 of the algorithms being proposed for probabilistic analysis are incredibly simple, yet have been shown to work extremely well almost everywhere. Addition of heuristics which would seem to improve the behavior of an algorithm add to the difficulty of analysis, but (if they can be shown not to reduce the accuracy of a guessed solution) may still be used in practice without weakening the results.

The probabilistic approach suffers from the same objections as average-case analysis, primarily the question of the underlying distribution over problem instances. However, it provides some assurance that an approximation algorithm which might fail in some cases will not fail very often, and hence instills some confidence in the user that he will be "surprised" only rarely. For some applications, this may be insufficient; since the alternative may be to expend a horrendous amount of computer time on a problem, or not to solve it at all, the method still may be useful.

VI. 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, a gap which may be partially filled by the approaches using data flow graphs and similar models. Techniques for proving upper bounds will not change significantly, although algorithm design will continue to advance with more unified principles (this prediction seems safe!). Average-case analyses must expand to include estimates of the variance and/or the nature of the distribution of solution times.

One practically unexplored path for future work in analysis of algorithms consists of developing a realistic model of parallel computation and techniques for decomposing problems on a "macroscopic" level (as opposed to previous approaches, which have concentrated primarily on parallel computation at the single-instruction level; see, for example, Traub [1973]). This area is practically devoid of design and analysis techniques, and because of the proliferation of actual parallel hardware, offers a unique opportunity for one inclined to work in this field.

Probably the most exciting and potentially rewarding area in the near future will be in designing and analyzing algorithms for hard problems. Of course, the interesting theoretical questions include the now-infamous P = NP problem. Presumably $P \neq NP$, so that approximate algorithms will predominate. For those problems which are not conducive to guaranteed approximate solutions, an approach such as Karp's [1976] which works probabilistically may be preferred. However, the "almost everywhere" concept is too restrictive and says too little about problems of a particular size, so a new definition of what constitutes an acceptable approximation algorithm might result in a new family of practical algorithms for some NP-complete problems. Results in this area will likely stem from application of concepts from probability and statistics.

VII. References

Aho, A.V., Hopcroft, J.E., and Ullman, J.D., The Design and Analysis of Computer Algorithms, Addison-Wesley, Reading, MA, 1974.

Beardwood, J., Halton, J.H., and Hammersley, J.M., "The shortest path through many points", Proceedings of the Cambridge Philosophical Society 55, 4 (Oct. 1959), 299-327.

Blum, M., Floyd, R.W., Pratt, V., Rivest, R.L., and Tarjan, R.E., "Time bounds for selection", Journal of Computer and System Sciences 7, 4 (Aug. 1973), 448-461.

Borodin, A., in Aho, A.V., ed., Currents in the Theory of Computing, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1973, 35-89.

Coffman, E.G., and Sethi, R., "A generalized bound on LPT sequencing", Proceedings of the International Symposium on Computer Performance Modeling, Measurement, and Evaluation, Mar. 1976, 306-310.

Cook, S.A., "The complexity of theorem proving procedures", Proceedings of the 3rd Annual ACM Symposium on Theory of Computing, 1971, 151-158.

Dahlquist, G., and Bjorck, A., Numerical Methods, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1974.

Dijkstra, E.W., "A note on two problems in connexion with graphs", Numerische Mathematik 1, (1959), 269-271.

Edmonds, J., and Karp, R.M., "Theoretical improvements in algorithmic efficiency for network flow problems", Journal of the ACM 19, 2 (Apr. 1972), 248-264.

Erdos, P., and Spencer, J., Probabilistic Methods in Combinatorics, Academic Press, New York, 1974.

Feller, W., An Introduction to Probability Theory and Its Applications, John Wiley and Sons, Inc., New York, 1968.

Fischer, M.J., and Meyer, A.R., "Boolean matrix multiplication and transitive closure", Conference Record, IEEE 12th Annual Symposium on Switching and Automata Theory, 1971, 129-131.

Floyd, R.W., and Rivest, R.L., "Expected time bounds for selection", Communications of the ACM 18, 3 (Mar. 1975), 165-172.

Ford, L.R., and Johnson, S.M., "A tournament problem", American Mathematical Monthly 66, (1959), 387-389.

Frazer, W.D., "Analysis of combinatory algorithms - a sample of current methodology", AFIPS Conference Proceedings 40, 1972, 483-491.

Fuller, S.H., Gaschnig, J.G., and Gillogly, J.J., "Analysis of the alpha-beta pruning algorithm", Carnegie-Mellon University, Dept. of Computer Science, Pittsburgh, PA, (July 1973).

Garey, M.R., and Johnson, D.S. [1976a], "The complexity of near-optimal graph coloring", Journal of the ACM 23, 1 (Jan. 1976), 43-49.

Garey, M.R., and Johnson, D.S. [1976b], "Approximation algorithms for combinatorial problems", Proceedings of the Symposium on New Directions and Recent Results in Algorithms and Complexity, Academic Press, to appear in Fall 1976.

Guibas, L.J., and Szemeredi, E., "The analysis of double hashing", extended abstract in Proceedings of the 8th Annual ACM Symposium on Theory of Computing, 1976, 187-191.

Hoare, C.A.R., "Quicksort", Computer Journal 5, (1962), 10-15.

Hopcroft, J.E., "Complexity of computer computations", Proceedings of IFIP Congress 74, vol. 3, 1974, 620-626.

Hyafil, L., "Bounds on selection", SIAM Journal on Computing 5, 1 (Mar. 1976), 109-114.

Ibarra, O., and Kim, C., "Fast approximation algorithms for the knapsack and sum of subsets problems", Journal of the ACM 22, 4 (Oct. 1975), 463-468.

Johnson, D.B., "A note on Dijkstra's shortest path algorithm", Journal of the ACM 20, 3 (July 1973), 385-388.

Johnson, D.S., "Approximation algorithms for combinatorial problems", Proceedings of the 5th Annual ACM Symposium on Theory of Computing, 1973, 38-49; also in Journal of Computer and System Sciences 9, 3 (Dec. 1974), 256-278.

Johnson, D.S., "Fast algorithms for bin packing", Journal of Computer and System Sciences 8, 3 (June 1974), 272-314.

Karp, R.M., "Reducibility among combinatorial problems", in Miller, R.E., and Thatcher, J.W., eds., Complexity of Computer Computations, Plenum Press, New York, 1972, 85-103.

Karp, R.M., "On the computational complexity of combinatorial problems", Networks 5, 1 (Jan. 1975), 45-68.

Karp, R.M., "The probabilistic analysis of some combinatorial search algorithms", Proceedings of the Symposium on New Directions and Recent Results in Algorithms and Complexity, Academic Press, to appear in Fall 1976.

Kleinrock, L., Queueing Systems, Vol. I: Theory, John Wiley and Sons, Inc., New York, 1975.

Knuth, D.E., The Art of Computer Programming, Vol. I: Fundamental Algorithms, Addison-Wesley, Reading, MA, 1968.

Knuth, D.E., The Art of Computer Programming, Vol. II: Seminumerical Algorithms, Addison-Wesley, Reading, MA, 1969.

Knuth, D.E., "Mathematical analysis of algorithms", Proceedings of IFIP Congress 71, vol. 1, 1971, 135-143.

Knuth, D.E., The Art of Computer Programming, Vol. III: Sorting and Searching, Addison-Wesley, Reading, MA, 1973.

Knuth, D.E., and Moore, R.W., "An analysis of alpha-beta pruning", Artificial Intelligence 6, (1975), 293-326.

Knuth, D.E., "Big omicron and big omega and big theta", SIGACT News 8, 2 (Apr.-June 1976), 18-24.

Kung, H.T., and Hyafil, L., "Bounds on the speed-up of parallel evaluation of recurrences", Carnegie-Mellon University, Dept. of Computer Science, Pittsburgh, PA, (Sept. 1975).

Lawler, E.L., "Algorithms, graphs, and complexity", Networks 5, 1 (Jan. 1975), 89-92.

Liu, C.L., Introduction to Combinatorial Mathematics, McGraw-Hill, New York, 1968.

Newborn, M.M., "The efficiency of the alpha-beta search on trees with branch-dependent terminal node scores", abstract in Proceedings of the Symposium on New Directions and Recent Results in Algorithms and Complexity, Academic Press, to appear in Fall 1976.

O'Neil, P.E., and O'Neil, E.J., "A fast expected-time algorithm for boolean matrix multiplication and transitive closure", Information and Control 22, 2 (Mar. 1973), 132-138.

Rabin, M.O., "Probabilistic algorithms", Proceedings of the Symposium on New Directions and Recent Results in Algorithms and Complexity, Academic Press, to appear in Fall 1976.

Reingold, E.M., "Establishing lower bounds on algorithms: a survey", AFIPS Conference Proceedings 40, 1972, 471-481.

Sahni, S., "Approximate algorithms for the 0/1 knapsack problem", Journal of the ACM 22, 1 (Jan. 1975), 115-124.

Sahni, S., "General techniques for combinatorial approximation", University of Minnesota, Dept. of Computer Science, Minneapolis, MN, technical report 76-6, (June 1976).

Schonhage, A., and Strassen, V., "Schnelle Multiplikation grosser Zahlen", Computing 7, (1971), 281-292.

Sedgewick, R., "Quicksort", Stanford University Ph.D. thesis, Stanford, CA, technical report STAN-CS-75-492, (May 1975).

Shamos, M.I., "Geometric complexity", Proceedings of the 7th Annual ACM Symposium on Theory of Computing, 1975, 224-233.

Shamos, M.I., and Yuval, G., "Lower bounds from complex function theory", Proceedings of the 17th Annual IEEE Symposium on Foundations of Computer Science, (Oct. 1976).

Sloane, N.J.A., A Handbook of Integer Sequences, Academic Press, New York, 1973.

Spiess, J., "Untersuchungen zur Implementierung der Algorithmen von S. Winograd und V. Strassen zur Matrizenmultiplikation", Gesellschaft fur Wissenschaftliche Datenverarbeitung MBH Gottingen, Bericht Nr. 10, (Aug. 1974).

Strassen, V., "Gaussian elimination is not optimal", Numerische Mathematik 13, (1969), 345-356.

Traub, J.F., ed., Complexity of Sequential and Parallel Numerical Algorithms, Academic Press, New York, 1973.

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

Valiant, L.G., "On non-linear lower bounds in computational complexity", Proceedings of the 7th Annual ACM Symposium on Theory of Computing, 1975, 45-53.

Wells, M.B., "Applications of a language for computing in combinatorics", Proceedings of IFIP Congress 65, vol. 2, 1965, 497-498.

Yuval, G., personal communication, 1975.