INTRODUCTION

The study of the computational efficiency and inherent limitations of algorithms for computer solution of problems drawn from classical continuous mathematics has been with us as long as general purpose computers themselves. Similar studies of algorithms of other kinds, however, have been sporadic and isolated until fairly recently. With the growing realization of the critical role played by combinatory algorithms in any instance in which the computer is employed as a logical decision maker, however, has come intensive and widespread interest in understanding such algorithms. Further, these studies have progressed to the point where patterns in analysis have begun to emerge and a summary seems in order. We shall focus our attention here on analytic approaches and methodology, rather than on the specifics of the results obtained. For a summary of the latter, the reader is referred to a companion paper elsewhere in this volume.16

To begin with, one might consider what constitutes a combinatory algorithm and what does not, but rather than restrict the nature of the analyses we shall consider, let us say merely that we shall not discuss problems in numerical analysis, in the theory of automata, or in arithmetic and leave it at that. We shall take it as our primary concern to try to summarize the kinds of information people have attempted to learn about various algorithms, and then indicate some of the tools and techniques they have employed. An effort will be made to point out common threads where possible, and to indicate potential directions for future development where this seems appropriate. As illustrative vehicles for the discussion we shall focus on some simple, but basic, examples. To the best of our knowledge, the analyses and proofs (as distinct from the results) are new except where noted.

Broadly speaking, analyses of combinatory algorithms can be classified in two "dimensions," viz.,

| Analysis of a Single Case (usually worst or best) | Analysis of a Class of Algorithms |
| Analysis of Probability Distribution of Cases | |

The kinds of information sought through analysis are typically: (a) Does the algorithm(s) do what is desired? (b) How does it (or they) perform if all goes favorably or unfavorably? (c) How good is performance "on the average," and (much less frequently) (d) How average is average—i.e., what is the variance?

Of course, the study of algorithm performance presupposes a choice of a figure of merit for that performance. This choice is frequently quite difficult to make in a manner which reflects adequately the amount of real computation involved. In the case of algorithms for sorting, for example, a common figure of merit is the number of comparisons made. This measure is a valid indicator of actual program performance, however, only if comparisons constitute the bulk of the computation. An algorithm which performs address calculations based on record key values may perform few or even no compares; another, which does base its course of action on the outcomes of compares, may nevertheless access memory often to move items around without comparing them, and thus be greatly inferior in reality to one which makes a few more comparisons. It is primarily because of a lack of reasonably close correspondence with reality in the underlying models and
measures of performance that almost all of the work on "computational complexity" in the literature of automata theory finds little or no use for our purposes.

In the course of our examination of the techniques of analysis of algorithms, we will observe repeatedly that there are two qualitative "depths" to which such an analysis is normally pursued. The first is a general level where all that is specified is the operations performed by the algorithm and the general form of the data structures it accesses. The second is the level of at least a source language implementation in which considerable attention must be paid to the implementation of these operations and data structures. An operation such as "find a circuit" of a linear graph, for example, is far more complex than "compare this item with that item," even in cases where the latter requires reference to a lexicography. Again, the data structures or accessing patterns required by the general specification of an algorithm may not be directly implementable or may incur excessive overhead in implementation.

With these caveats in mind, let us begin an examination of some of the main themes of the extant literature in the analysis of algorithms.

CERTIFICATION: DOES THE ALGORITHM DO WHAT IS DESIRED?

Many combinatory algorithms, generally stated, do not require detailed formal certification of correctness and/or termination because it is obvious that these properties are satisfied. Most sorting algorithms, for example, fall into this category, as do most search algorithms. As noted earlier, there are two levels at which one can question these properties—that of the abstract or "pure" statement of the algorithm, and that of a particular implementation. Obviously, even verification at the level of implementation in a source language program will not guarantee that the compiled program will perform as desired, but it is a much better guarantee than verification at the abstract level. In practice, however, both levels are frequently desirable, since the techniques required for one are often qualitatively different from those for the other, and it is most desirable to know that the algorithm is free of flaws before carrying it to the level of a detailed implementation. As an example of some of these ideas let us consider the following problem:

Consider a linear graph consisting of a finite collection of vertices, some of which are joined pairwise by a collection of edges. With each such edge is associated a positive length, and each pair of vertices may be joined by at most one edge. The only other restriction is that there must be some sequence of edges, or path, joining any pair of vertices (i.e., no group of vertices is "isolated"). A (spanning) tree, \( T \), is a collection of edges having the property that within \( T \) there is exactly one path joining any pair of vertices. Our problem is to find a tree of minimum total length, \( T_{\text{min}} \).

A very effective algorithm (in fact, the best published to date) for this problem is due independently to Prim\textsuperscript{15} and Dijkstra\textsuperscript{2}. We begin by selecting an arbitrary vertex, \( v_o \), and assigning to \( T_{\text{min}} \) the shortest of its incident edges, say one linking \( v_o \) to \( v_1 \), denoted \( (v_o, v_1) \). Next, from all of the edges linking either \( v_o \) or \( v_1 \) to other vertices we select the shortest, which links \( v_o \) or \( v_1 \) to, say, \( v_2 \). We now repeat the process, looking this time for the shortest edge joining \( v_o, v_1 \), or \( v_2 \) to some other vertex, and continue in this manner until all vertices have been included. Notice that at any time we need only keep track of a shortest edge from some vertex already in \( T_{\text{min}} \) to each of those vertices not yet included. As each new vertex is joined to \( T_{\text{min}} \), the lengths of its edges to vertices not yet in \( T_{\text{min}} \) are compared with the current minima, and the latter replaced where appropriate.

That this procedure will terminate is clear, even from the informal statement just given, because a new vertex is added to the tree at each step. It is only slightly less clear that the resulting collection of edges will be a tree, but a moment's reflection will convince the reader that a path between any pair of vertices must be found and will be unique. There is no way to add to \( T_{\text{min}} \) an edge joining vertices already joined by a path, since the only edges under consideration are those joining such vertices to vertices not yet reachable by edges in \( T_{\text{min}} \).

The remaining major question, that of the minimality of \( T_{\text{min}} \), is less obvious. Suppose \( T_{\text{min}} \) were not minimal, and suppose there were some minimal tree \( T' \) which differed from \( T_{\text{min}} \) in a minimal number of edges. Let \( E \) be the set of edges of \( T_{\text{min}} \) not in \( T' \) and let \( e \) be a shortest edge of \( E \). Now, in \( T_{\text{min}} \) \( e \) uniquely joins the set of vertices already in \( T_{\text{min}} \) at the time \( e \) was added, \( V_e \), to the set of the remaining vertices of the graph, \( W_e \). Since \( T' \) is a tree, it also has at least one edge

![Figure 1](image-url)
joining \( V_e \) to \( W_e \); furthermore, adding \( e \) to \( T' \) will create two paths (in \( T' \cup \{e\} \)) between some vertices of \( V_e \) and some of \( W_e \), one path via \( e \) and one via some edge, \( e' \), of \( T' \) (cf. Figure 1). Now length \( (e') \geq \text{length} \ (e) \), since \( e \) was chosen by our algorithm. On the other hand, if we substitute \( e \) for \( e' \) in \( T' \) the result will still be a tree, \( T'_e \). (The vertices in \( V_e \) and \( W_e \), formerly connected in \( T' \) via \( e' \) will now be connected via \( e \).) But \( T'_e \) can have no less total length than \( T' \), since \( T' \) was minimal; hence, length \( (e) \geq \text{length} \ (e') \). Combining these observations, \( \text{length} \ (e) = \text{length} \ (e') \) and length \( (T'_e) = \text{length} \ (T') \). But \( T'_e \) differs from \( T_{\text{min}} \) in one fewer edges than \( T' \) and this contradicts our choice of \( T' \); thus, \( T' \) does not exist and \( T_{\text{min}} \) is minimal, as required.

We note in passing that the device employed here is a powerful tool in the analysis of graph algorithms. One verifies the optimality of the solution produced by an algorithm by postulating the existence of an optimal solution differing from the generated solution in a minimal way, and then constructively refutes the minimality of this difference.

Having satisfied ourselves as to the basic soundness of the algorithm, the next step is to embody it in a program and verify the correctness of the program. We will not pursue this step in detail here for lack of time and space but will rely upon the reader’s experience to assist in understanding the qualitative difference between such an undertaking and what we have just discussed. To begin with, even in a higher language one is now faced with a host of syntax related problems having to do with whether variables are of the appropriate type and with the legitimacy of program statements, for example. There are also semantic questions such as whether loop indices are tested against range boundary values before or after incrementing and whether their values are preserved intact where required, as well as implementation-dependent considerations such as whether input and output are performed correctly, and so on. Next come considerations of program logic and control flow including again the question of termination, this time for the program. Finally, and perhaps most difficult, one has the question of whether the program actually embodies the algorithm at all. This process is clearly vastly more detailed than our proof above, even if the latter were to be thoroughly formalized. The most commonly accepted current approach to such a certification entails first supplementing the program code with assertions about statements in that code and verifying by hand that these assertions reflect accurately the desired properties of the statement [cf. e.g., References 5, 10, 18]. Next the assertions, couched in a form more suitable for treatment via classical mathematical proof techniques, are themselves verified. There is substantial effort under way in a number of areas to address these problems. It is abundantly clear that the first process—that of establishing the correspondence between statements and assertions—could benefit from mechanical assistance. On the other hand, a hopeful sign is that the prospects for providing such assistance appear reasonably good as a consequence of increased understanding of programs resulting from, for example, compiler optimization studies.1 Automatic verification of the assertions is considerably more difficult, though a good deal of work is being done [cf. References 5, 10, 18]. There are a number of recent instances, however, in which humans have been quite successful in accomplishing this without undue tedium, and thus automatic assistance may be all that is required for a viable system.

EXTREME PERFORMANCE: WHAT HAPPENS AT BEST (OR AT WORST)?

Granted that one has assured himself of the validity of an algorithm, it is next natural to ask how well the algorithm performs its assigned task. Here again, we find the opportunity (or need) for two levels of analysis: one at the general algorithm level and the other at the detailed program level.

In the case of our example, the best and worst cases would arise when the graph being processed consisted of a single path and a “complete graph” (i.e., when each vertex has an edge link to every other vertex), respectively. It is clear that a reasonable figure of merit for the performance of this algorithm must include some measure of the work performed in finding the shortest edge at each stage; it may or may not be appropriate in addition to examine the amount of storage space required by the procedure.

Suppose that we have \( n \) vertices. Then in the former case, only \( (n-1) \) edges need be examined since only a single choice is available at each stage, while in the latter one must choose the first edge from among \( (n-1) \), the second from among \( 2 \times (n-2) \), etc., and the last from among \( (n-1) \) again. It should be fairly obvious that one can find the smallest of \( k \) objects in \( (k-1) \) comparison steps, and that one can do no better than this. (This is an instance in which one can make a very strong statement about existence and optimal performance of a whole family of algorithms.) Thus, a first analysis might lead to the conclusion that \( \sum_{i=1}^{n-1} i(n-i) = [n(n-1)(n+1)/6] \) comparison steps are required. Recall, however, that as each new vertex joins the growing \( T_{\text{min}} \), its edges to vertices not yet included need be compared only with the shortest of the corresponding edges from vertices already in-
requires that one move to the next level of analysis—still account for the work required to keep track of vertices and edges already in a program—at least in a sense.

An accurate account of this kind, however, really means that we need to keep track of the number of comparisons and the number of operations performed. At a cost of some additional storage to keep track of the current minima, we have avoided a cumbersome search at each stage and improved performance from $O(n^3)$ to $O(n^2)$. This serves to emphasize that even at this general level of analysis, one must exercise great care to be sure that he has captured the essence of the algorithm, for $O(n^2)$ performance is significantly different from $O(n^3)$.

Of course, the analysis is not yet complete. We must still account for the work required to keep track of which vertices and edges are already in $T_{min}$ and which are not, and to do the bookkeeping for the main loop. An accurate account of this kind, however, really means that one move to the next level of analysis—that of the program itself.

Not surprisingly, the level of detail in such an analysis is again considerable. Still, the number of parameters is not overwhelming, for the flow of control in a program—at least in a good one—is normally highly systematic. In addition, the certification process has by this time assured one that his program is both consistent and free of infinite loops. Thus, each loop will be entered and exited an equal number of times, and this number can be used as a parameter characterizing the number of executions of a whole set of statements within and without the loop; also, several of these parameters are often available from the analysis of the algorithm at the “general” level. Again, we shall omit this phase of the analysis in order to move on. It is worthwhile noting, however, that the kind of analysis would be essentially similar to that which we have just seen—i.e., finite discrete mathematics involving integer quantities—although quite likely more complex than our simple example [cf. Reference 13].

There is a drawback in this kind of extreme analysis, however, which may not yet be apparent. Observe that there is a major qualitative difference between the best and worst case performance. It is probably unlikely that one would encounter either in practice, and thus, while the time performance for most graphs lies somewhere between these extremes, the range is so large as to furnish little information. This is the motivation for undertaking the kind of analyses we shall consider next, those which attempt to quantify “average” performance. Before moving on to consider analysis of average performance, however, let us digress briefly to consider a problem in which extreme performance is of paramount importance.

Consider a two-input, two-output device which compares the values of, and then propagates, its inputs in such a way that the larger input reappears on the first output line and the smaller on the second. The objective in the construction of “sorting networks” is to devise arrays of such devices which accept a string of input items as input and then shuffle these input items about internally over fixed connections to produce the items into sorted order. For such networks one figure of merit is obviously the number of comparisons (i.e., devices) required. The major difference between this approach and most standard approaches to sorting, however, is that here the set of comparisons is immutable regardless of the input sequence (since the connections are fixed). In return for this inflexibility, one gains a potential for parallel operation and a saving in time. Thus, an equally important figure of merit for most applications is the total time required to sort, and this is determined by the maximum number of comparisons made against any item in moving from input to output. In this instance, as with many involving algorithms whose course of action is essentially independent of the values or positions of data items, extreme performance is of primary interest.

EXPECTED PERFORMANCE: HOW GOOD IS THE ALGORITHM “ON THE AVERAGE”?

A more difficult kind of analysis of algorithm performance, in general, is that required to determine expected or “average” performance. Still, it is precisely this kind of analysis which usually yields the most insight into actual observed algorithm performance. Outside the realm of information theory where random coding arguments have found some utility, it is highly unusual to find an analysis of expected performance of families of algorithms; far more often is a single algorithm analyzed in this way, and this is the kind of analysis we shall pursue next. For reasons to which we shall return later, we shall forsake our minimal spanning tree algorithm and concentrate on another algorithm to illustrate how such an analysis is carried out.

The algorithm we shall examine is one which sorts (i.e., permutes) an input sequence of records, each of which consists of both a “key” chosen from some linearly ordered set and other associated information, into, say, ascending lexicographic key order. For convenience in analysis and description, we shall assume that all of the key values are distinct. That this makes no essential difference will be easily verifiable in retrospect. For brevity, we shall refer to “record values”
Quick sort constructs a binary tree whose vertices correspond to the, say \( n \) records in the input sequence. The construction of the tree is controlled by the following rule, applied recursively to each subtree: if a record with (key) value \( x_0 \) is assigned to vertex \( i \) during construction, then any subsequent record, with value \( x \), assigned to the (sub)tree rooted at \( i \), will be assigned to the left subtree of \( i \) if \( x < x_0 \) and to the right subtree if \( x > x_0 \). Thus, the first input record becomes the root of the Quick sort tree, the second input record the root of one of the two principal subtrees, and so on.

Evaluation of expected values presupposes the existence of a probability distribution over input permutations; we shall adopt the hypothesis that all such permutations are equally likely. Although there are definitely situations in which such is not the case, there are many others to which it is a reasonable approximation. In addition, the results of such an analysis usually are observed to be good indicators of the measured performance of running programs. Parenthetically, although this may seem a strong assumption, it is actually weaker than assuming, for example, that the key values are independent, identically distributed, random variables.

To proceed with the analysis, we need once again to choose a figure of merit, but before we can choose one we need more information about how the algorithm is to be implemented. Suppose the input sequence is in locations numbered 1 through \( n \): Transfer the first record to another location \( i \), and begin a search of 2, 3, \ldots, looking for the first record, say \( x_2 \) in location \( j \), larger than \( x_1(x_2 > x_1) \). Following examination, each such record is returned to a location numbered one less than its original one. Now begin a search of locations \( n, n-1, \ldots \), looking for the first record, say \( x_2 \) in location \( k \), smaller than \( x_1 \), and returning each record to its original location. Interchange the contents of locations \( j-1 \) and \( k \), i.e., \( x_2 \) and \( x_1 \). Now all records in locations 1 through \( j-1 \) are smaller than \( x_1 \) and all those in locations \( k \) through \( n \) are larger; location \( j \) is redundant. We now repeat the above process on locations \( j+1 \) through \( k-1 \) and continue this until the pointers \( j \) and \( k \) converge to adjacent locations, say, \( r \) and \( r+1 \). Then \( x_2 \) is inserted into \( r \), its correct position in the output sequence, and the two sequences in locations 1 through \( r-1 \) and \( r+1 \) through \( n \) are each treated by the above process.

At the level of detail we are considering, the only obvious figure of merit is the number of comparisons made in the searches. There are no other obvious memory accesses or operations, except those required to maintain pointers to the boundaries of partitioned subsequences, such as \( (r-1) \) and \( (r+1) \) above. In fact, it is possible to hold this requirement to quite reasonable limits\(^{5,7,8} \) and we shall therefore concentrate on the expected number of comparisons, which corresponds to the expected path length of the Quick sort tree.

Let \( E(n) \) be the expected compares required to sort a sequence of \( n \) records. Then, if \( p(r) = \) probability that the root of the Quick sort tree ranks \( r \)th in the input sequence,

\[
E(n) = \sum_{r=1}^{n} p(r) [E(r-1 \mid r) + E(n-r+1 \mid r) + (n-1)]
\]

where \( E(y \mid r) \) denotes the expected compares to sort a sequence of \( y \) records conditioned on the fact that \( r \) is the root. But under our assumptions, any record is equally likely to be the root; thus \( p(r) = 1/n \) for all \( r \); furthermore \( E(y \mid r) \) is similarly independent of \( r \). Thus:

\[
E(n) = (n-1) + \frac{2}{n} \sum_{r=1}^{n-1} E(r-1)
\]

Similarly, following Windley:\(^{29} \)

\[
E(n) = (n-2) + \frac{2}{n-1} \sum_{r=1}^{n-2} E(r-1)
\]

\[
nE(n) - (n-1)E(n-1) = 2(n-1) + 2E(n-1) \tag{1}
\]

\[
\frac{E(n)}{n+1} - \frac{E(1)}{2} = \left[ \frac{4}{n+1} - \frac{2}{n} + \frac{4}{n-1} - \frac{2}{n-2} + \cdots + \frac{4}{3} - \frac{2}{2} \right]
\]

or

\[
E(n) = 2(n+1) \sum_{i=1}^{n} \frac{1}{i+1} - 2n
\]

\[
\approx 1.39(n+1) \log_2 n - 0(n)
\]

This kind of analysis is roughly the same as that which we encountered previously, and the technique of arranging for some linear combination of the variables on the left which will be equal to a tractable sum is one frequently employed. Often, however, recurrence equations will not succumb to such an analysis. Consequently,
another technique frequently called upon is the use of generating functions. A generating function for a sequence is an infinite formal power series expansion in which the coefficient of \( x^n \) is the \( n \)th term in the sequence. Thus, if \( G(x) \) is the generating function for \( E(n) \):

\[
G(x) = E(1)x + E(2)x^2 + \cdots + E(n)x^n + \cdots
\]

Furthermore the generating function for \( nE(n) \) is

\[
x(d/dx)G(x) = E(1)x + 2E(2)x^2 + \cdots
\]

Now using this fact the recurrence equation (1) can be rewritten in terms of generating functions as

\[
x(d/dx)G(x) = G(2(n-1)) + \frac{d}{dx}(x^2G(x))
\]

where

\[
G(2(n-1)) = \frac{2x^2}{(1-x)^2}
\]

\[
= 2x^2 + 4x^3 + 6x^4 + \cdots
\]

Notice that this equation is in fact an equation involving infinite power series and thus constitutes an infinite set of coefficient equations, one for each power of \( x \) since the powers of \( x \) are linearly independent. Setting \( y = G(x) \), we have

\[
x^2y' = [2x^2/(1-x)^2] + 2xy + x^2y'
\]

\[
y' - [2/(1-x)]y = 2x/(1-x)^2
\]

This differential equation has an integrating factor

\[
\exp \left[ - \int \left( \frac{2}{1-x} \right) dx \right] = (1-x)^2.
\]

Thus

\[
(1-x)^2y' - 2(1-x)y = 2x/(1-x)
\]

Integrating both sides:

\[
y(1-x)^2 = 2(-x - \ln(1-x)) + c
\]

or

\[
G(x) = y = \frac{2}{(1-x)^2} \ln \left( \frac{1}{1-x} \right) + \frac{c}{(1-x)^2} - \frac{2x}{(1-x)^2}
\]

Now

\[
\frac{1}{(1-x)^2} \ln \left( \frac{1}{1-x} \right) = x + (1 + \frac{1}{2})x^2 + (1 + \frac{1}{2} + \frac{1}{3})x^3 + \cdots
\]

Thus

\[
\frac{d}{dx} \left[ \frac{1}{(1-x)^2} \ln \left( \frac{1}{1-x} \right) \right] = 1 + 2(1 + \frac{1}{2})x + 3(1 + \frac{1}{2} + \frac{1}{3})x^2 + \cdots
\]

\[
= \frac{1}{(1-x)^2} \ln \left( \frac{1}{1-x} \right) + \frac{1}{(1-x)^2}
\]

\[
\therefore y = 2 \frac{d}{dx} \left[ \frac{1}{(1-x)^2} \ln \left( \frac{1}{1-x} \right) \right] - \frac{2x}{(1-x)^2} + \frac{c - 2}{(1-x)^2}
\]

Looking at \( E(1) \), the coefficient of \( x \) in \( G(x) \):

\[
E(1) = 0 = 4(1 + \frac{1}{2}) - 2 + 2(c - 2)
\]

\[
\therefore c = 0
\]

and

\[
E(n) = 2(n+1) \sum_{i=1}^{n+1} \frac{1}{i} = 2(n+1) - 2(n+1) - 2n
\]

\[
= 2(n+1) \sum_{i=1}^{n+1} \frac{1}{i+1} - 2n
\]

as before.

Although in this instance the generating function solution is somewhat more elaborate than the direct solution, we took the trouble to illustrate both because the existence of a direct solution is rather a fortuitous circumstance. (The example was chosen, in fact, because both kinds of solution were possible.) The availability of all of the functional power of infinite series vastly increases one’s flexibility when dealing with problems of analysis, as may be apparent from the operations above. Of course, there are a large number of other approaches to the solution of recurrence and difference equations [cf. References 9, 12, 14, 17].

A final remark is in order regarding the question of expected performance of graph algorithms. The unfortunate fact is that there seems at present to be no way effectively to characterize a “random” graph. The kinds of graphs submitted to graph algorithms normally have the imprint of human intelligence in design or at least selection. This imprint is not captured effectively by choosing a random incidence matrix, for example, nor by any of the other obvious choices of a definition of randomness to which one is tempted. The discovery of a natural and analytically tractable characterization of a random graph remains a major open question in the analysis of algorithms.

VARIANCE: HOW OFTEN DOES “AVERAGE” PERFORMANCE OCCUR?

Expected performance is most probable performance, but for some algorithms actual performance can be a
highly variable function of input data as we have noted. Among all of the kinds of analysis to which algorithms are subjected, analysis of the variance of performance is unquestionably the most difficult—and the most often omitted.

There are, however, sometimes legitimate ways in which to justify omission of computation of the variance. For example, in the case of Quicksort, the extremes of performance occur when the tree is a single path (i.e., completely unbalanced) and when it is completely balanced. In the former case, this number of comparisons is \( n(n-1)/2 \), while in the latter it is \( (n+1) \log_2(n+1) - 2n \). Thus, expected performance is qualitatively the same as best performance, i.e., \( O(n \log n) \).

Therefore, even though performance in the worst case can deteriorate somewhat severely, this must be a consequence of a long “tail” in the distribution and hence quite unlikely under the hypothesis of equally likely input permutations.

As a final example, consider the problem of inserting a new record into a sorted sequence of records. Rather than adopt the standard “binary search” approach, however, let us proceed as follows: Given an \( n \)th record to be inserted into a sorted sequence of \( (n-1) \) records, we shall pick a point, \( i \), at random, make a comparison with the \( i \)th record, and thereby determine whether the new record belongs among the first \((i-1)\) or the last \((n-i+1)\). (We shall again assume that all key values are distinct.) We will then repeat the “random probe” process on the appropriate subsequence and continue recursively in this way until the proper position for the new record has been found. Again, let us assume that the new record is equally likely to belong anywhere in the sequence—including at the ends. Let \( p(k, n) = \text{prob. that } k \text{ compares are required to insert an item into a set of } n \) records. Then

\[
p(k, n) = \sum_{i=1}^{n} \left( \frac{1}{n} \right) \left( \frac{i}{n+1} \right) p(k-1, i-1) + \left( \frac{n-i+1}{n+1} \right) \left( \frac{1}{n} \right) p(k-1, n-i)
\]

By symmetry

\[
p(k, n) = \frac{2}{n(n+1)} \sum_{i=1}^{n} i \ p(k-1, i-1)
\]

Setting

\[
G_n(x) = \sum_{k=1}^{\infty} p(k, n) x^k
\]

we have

\[
G_n(x) = \sum_{i=1}^{n} \left[ \frac{2}{n(n+1)} \sum_{i=1}^{n} i \ p(k-1, i-1) \right] x^i
\]

It now appears that we are in trouble, for not only do we have two indices in the recurrence, but, in addition, the index upon which we based the definition of the generating function will not help with the factors \( i \) or \( 1/n(n+1) \). In order to get such assistance, we define

\[
F = \sum_{n=0}^{\infty} G_n(x) y^n
\]

\[
A = \frac{1}{1-\frac{1}{n} \frac{1}{1-y}} = \sum_{n=1}^{\infty} \frac{nG_{n-1} y^{n-1}}{n+1}
\]

\[
A = \sum_{n=1}^{\infty} \frac{nG_{n-1} y^{n-1}}{n+1}
\]

\[
\int \frac{A y}{1-y} \frac{dy}{1-y} = \sum_{n=1}^{\infty} \frac{\sum_{i=1}^{n} iG_{i-1} y^n}{n+1} = \sum_{n=1}^{\infty} \frac{\sum_{i=1}^{n} iG_{i-1} y^n}{n+1} = \sum_{n=1}^{\infty} \frac{\sum_{i=1}^{n} iG_{i-1} y^n}{n+1} = \sum_{n=1}^{\infty} \frac{\sum_{i=1}^{n} p(k, i-1, x^i y^{i+1})}{n+1}
\]

Notice how we have approached, step by step, the form we wish on the right hand side. Each step utilizes a standard technique for generating functions, and the result is very close to what we need; all that is required is a factor of two and an adjustment of the \( x \) and \( y \) indices:

\[
\frac{2x}{y} \sum_{n=1}^{\infty} \left[ \sum_{k=1}^{n} \left[ \sum_{i=1}^{n} \frac{2ip(k-1, i-1)}{n(n+1)} \right] x^i \right] y^n = F
\]
Differentiating:
\[ A = \frac{1 - y}{2x} \frac{\partial}{\partial y} A \]
\[ 1 \frac{\partial A}{A \partial y} = \frac{2x}{(1 - y)} \]
\[ A = \frac{c}{(1 - y)^{2x}} \]
\[ = c \sum_{n=0}^{\infty} \binom{-2x}{n} (-y)^n \]

where \( \binom{-2x}{n} \) is a “generalized” binomial coefficient defined by
\[ \binom{-2x}{n} = \frac{(-1)^n (2x+1)(2x+2) \ldots (2x+n-1)}{n!} \]

Therefore, using the fact that \( G_1(x) = x \) to find \( c = 1 \),

\[ A = \frac{\partial}{\partial y} yF = 1 + xy + \frac{(2x)(2x+1)}{2} y^2 + \ldots \]
\[ yF = \sum_{n=0}^{\infty} \frac{(-2x)}{n} \binom{-2x}{n} (-y)^{n+1} \frac{n+1}{n+1} \]
\[ F = \sum_{n=0}^{\infty} \frac{1}{n+1} \binom{2x+n-1}{n} y^n \]

and
\[ G_n(x) = \frac{1}{n+1} \binom{2x+n-1}{n} \]

Why go to all this trouble? Observe that
\[ G_n'(x) = \frac{d}{dx} G_n(x) = \sum_{k=0}^{\infty} kp(k,n)x^{k-1}. \]

This means that \( G_n'(1) = \text{mean } (k) \). Similarly
\[ G_n''(x) = \sum_{k=0}^{\infty} k(k-1)p(k,n)x^{k-2} \]

and
\[ G_n''(1) = \text{variance } (k) = \text{mean } (k) + \text{mean }^2 (k) \]

or
\[ \text{variance } (k) = G_n''(1) + G'(1) - [G'(1)]^2. \]

This provides the motivation; when the generating function is a generating function for a probability distribution, the moment information can be obtained easily from it as shown. A particularly fortuitous occurrence is the occasional emergence of a tractable recurrence involving the derivative of the generating function itself which enables one to arrive at an expected value and a variance without ever obtaining a closed form for the generating function (cf. Reference 12, Vol. I, p. 99), but this did not happen here.

Pursuing our analysis, we find that
\[ G_n'(x) = \frac{2}{(n+1)!} \sum_{i=0}^{n-1} \frac{2x(2x+1) \ldots (2x+n-1)}{2x+i} \]

or
\[ G_n'(1) = 2 \sum_{i=1}^{n} \frac{1}{(i+1)} = 2 \ln(n+1) - 2 \]

Thus, strangely enough, as \( n \) grows larger this “random probing” insertion requires only about \( 2 \ln 2 \) or less than 1.4 times as many compares on the average as does binary search. We shall leave it as a simple mathematical exercise to complete the computation and verify that
\[ \text{variance } (k) = 2 \sum_{i=1}^{n} \frac{1}{i+1} - 4 \sum_{i=1}^{n} \left( \frac{1}{i+1} \right)^2. \]

So, in this instance, average performance is fairly typical, since the variance is nearly equal to the mean.

By comparison, the “worst case” performance for this algorithm requires comparison of the new item with each of the \( (n-1) \) others. Thus, here again, there is a long tail to the distribution. This is not surprising in light of the close relationship between this procedure and Quicksort, the precise nature of which we shall leave to the reader to discover for himself.

As a final remark, we observe that the factor of \( (2\ln 2) \) which appears in both the analysis just completed and that of Quicksort is characteristic of binary tree-based random algorithms. As we have seen, the technique of basing the course of action of an algorithm upon operations made on a randomly selected argument is rather more powerful than might at first be apparent [cf. Reference 6].

**SUMMARY**

In this brief space, it has of course been impossible to do justice to even the nascent field of combinatorial algorithm analysis. We have, however, attempted to indicate at least in a qualitative way both the types of information usually sought (and why) and the general character of the accompanying analysis. By far the most authoritative single current source in the areas of performance is the Knuth series, but having the
techniques and ideas in hand, there is a broad range of more classical literature to which one may turn for assistance [e.g., References 4, 9, 11, 14, 17].

This is an area of very rapid growth. As it matures, one would expect to see a considerable broadening of analytic technique, but more important will be (a) results which increase our understanding of the principles of algorithm design analogous to, but deeper than, our earlier remark about randomized binary tree algorithms, and (b) results which broaden the scope of possible analysis—such as a good definition(s) of a “random” graph.

REFERENCES

1 F E ALLEN
   Program optimization

2 K E BATCHER
   Sorting networks and their applications
   AFIPS Proc SJCC 1968 pp 307-314

3 E W DIJKSTRA
   A note on two problems in connection with graphs
   Num Mathematik I pp 269-71 1959

4 W FELLER
   An introduction to probability theory and its applications
   Vol I New York John Wiley and Sons 1950

5 R W FLOYD
   Assigning meanings to programs

6 W D Frazer A C McKELLAR
   Sample sort: A random sampling approach to minimal storage tree sorting
   JACM 17 3 pp 496-507 July 1970

7 T N HIBBARD
   Some combinatorial properties of certain trees with applications to searching and sorting
   JACM 9 pp 19-28 January 1962

8 C A R HOARE
   Quicksort
   Computer J 5 1962 pp 10-15

9 C JORDAN
   Calculus of finite differences
   2nd Ed New York Chelsea Pub 1947

10 J C KING
   A program verifier
   PhD Dissertation Carnegie-Mellon University 1969

11 K KNOPP
   Theory and application of infinite series
   Blackie and Sons Ltd London 1928

12 D E KNUTH
   The art of computer programming
   I, II, III 1968 1969 1972 Addison Wesley Reading Massachusetts

13 __________
   Mathematical analysis of algorithms
   Proc Cong IFIP-71 To appear

14 L M MILNE-THOMPSON
   The calculus of finite differences
   London McMillan and Co 1933

15 R C PHIL
   Shortest connecting networks and some generalizations
   Bell Syst Tech Journal 36 pp 1389-1401 1957

16 E M REINGOLD
   Establishing lower bounds on algorithms, a survey
   This Proceedings

17 J RIORDAN
   An introduction to combinatorial analysis
   New York John Wiley and Sons 1958

18 H R STRONG
   Translating recursion equations into flowcharts
   J Comp and Syst Sci 5 3 pp 254-286 1971

19 P F WINDLEY
   Trees, forests, and rearranging
   Computer J 3 2 pp 84-88 1960