MASSACHUSETTS INSTITUTE OF TECHNOLOGY
A. I. LABORATORY

Artificial Intelligence
Memo No. 237 - PRELIMINARY RELEASE September 1971

AN INQUIRY INTO ALGORITHMIC COMPLEXITY

Patrick E. O'Neil*

This Research was begun at the I.B.M. Cambridge Scientific Center in 1968, continued while the author was an Office of Naval Research Post Doctoral Fellow at the M.I.T. Mathematics Department, grant N00014-67-A-0204-0034 during 1969-70; and is presently being supported by the Artificial Intelligence Laboratory, a Massachusetts Institute of Technology research program supported in part by the Advanced Research Projects Agency of the Department of Defense and monitored by the Office of Naval Research under Contract Number N00014-70-A-0362-0002.

*Department of Electrical Engineering, Massachusetts Institute of Technology.

Forward

This is the first section in a proposed monograph on algorithmic complexity theory. Future sections shall include: Information Theory as a Proof Technique; Algorithms Using Linear Form Inequalities; Some Probabilistic Analyses of Algorithms, etc. Comments, suggestions, and corrections are welcomed. Please let me know what you think.

This is not a limited distribution document, although I may wish to publish it later. Anyone who develops an idea based on this work to a more advanced state is welcome to publish first. I would be very eager to see any such result as soon as possible.

SECTION I: A FORMULATION OF BASIC CONCEPTS

§1 Definition of the Problem

Many readers will recall the advent of the "Fast Fourier Series" paper of Cooley and Tukey [12]. This paper described an algorithm to evaluate the complex Fourier series

$$x(j) = \sum_{k=0}^{n-1} A(k) \cdot W^{jk} \quad , \; j = 0,1,\ldots,n-1,$$

where $W = e^{2\pi i/n}$. The algorithm required only cn log n complex additions and multiplications to perform this evaluation, where previous methods had used cn^2 such operations. The immediate result of this discovery was a dramatic improvement in performance of calculations which were basic in a wide spectrum of applications; programs which had used hours of computer time every day could now be rewritten using the Cooley-Tukey algorithm to take only minutes!

This improvement of an old and long accepted algorithm dramatizes the need for a systematic study of the following problem:

THE PROBLEM OF ALGORITHMIC COMPLEXITY: Given an "important" task which may be performed by some set of algorithms implementable on a "standard computer", what is the "minimum time" needed to perform this task using any such algorithm.

The concept of the "importance" of a task shall be left intuitive, but the reader may substitute the phrase "occurring frequently in applications", as for example the ordering of a sequence of numbers or the multiplication of two matrices. We shall spend a great deal of time in this section clarifying what we mean by "standard computer" and "time taken by an algorithm".

Our eventual aim of course is to find the best algorithm to perform each

task; but to prove that an algorithm is best for a task we must solve the Problem of Algorithmic Complexity. As has been pointed out by a number of researchers, notably by Jack Schwartz, there are other measures of complexity for a task: the length of the shortest program for an algorithm that performs the task, the minimal size of storage used by any algorithm performing the task, etc. With no intent to ignore these other measures of complexity (indeed we cannot, see §2, Example 1, this section), we shall focus our attention on the time performance of an algorithm.

In the last few years a great deal of research has been done on problems of algorithmic complexity. At the same time, many sophisticated computer scientists have expressed perplexity (or lodged objections) concerning both the aims and the basic assumptions of this field of inquiry. This is probably due to the fact that algorithmic complexity is such a new field that little has been written on it and misconceptions tend to arise in such circumstances.

In this section we shall attempt to indicate the scope of the field and scrutinize the assumptions of commonly employed techniques of proof. We undertake this to answer some of the objections we have mentioned, but more importantly to clarify the basic assumptions upon which any useful theory must be founded.

§2 The "Standard" Computer

Since we are attempting to present the groundwork for a theory which will find wide applications, we would like our model for a "standard" computer to reflect the structure of the general purpose computer. Thus we stipulate that information shall be stored as sequences of zeroes and ones (and thereby exclude analog computers from consideration). We also require that the information stored should be directly accessible; any bit of information stored in memory may be retrieved in a length of time which is constant, irrespective of its "position" with respect to our previous access.

By imposing this requirement, of course, we eliminate the entire spectrum of tape automata as subjects of consideration. There are drawbacks to this step. In particular we exclude Turing machines, which are capable of performing any task for which an effective procedure has been conceived, and are attractive to the theorist because of their simplicity of concept and firm basis in rigor. We cannot hope to achieve comparable standards of rigor in dealing with general purpose computers for years to come. Furthermore, progress has already been made in studying the complexity of algorithms, using Turing machines to good effect. We shall consider this latter point further when we discuss the field of computational complexity. For the present, we limit ourselves to a few remarks in defense of the assumption that "standard" computers should have direct access memory.

The paramount consideration here is that general purpose computers do have direct access memory; our desire to reflect this fact in our model springs from our purpose of developing a theory with wide applications. Turing machines were developed historically to investigate the question of what problems were effectively decidable; the time which was spent in working out

an algorithm was never a consideration. Now it is possible, because of the elemental nature of the Universal Turing machine, that one might be able to calculate a measure of implicit complexity of a task which bore a one to one correspondence to the minimal time needed to perform the task on any conceivable machine. Naturally this calculation would only be possible for tasks known to be computable, or we would have solved the halting problem. Even if we postulate the computability of such an implicit complexity (which is far removed from present day capabilities) the following problem still faces us: what is the one to one correspondence which will enable us to deduce the minimal time required to perform the task on a general purpose computer? One cannot get out of a theory what has not been put into it! A general purpose computer has direct access memory, and time considerations are highly sensitive to this fact. Hence, a model such as the standard computer with direct access memory must be developed and studied.

Returning to our description of the standard computer, we stipulate that memory is segmented into contiguous strings of bits, each of standard length, which shall be called "words". We make no assumption concerning the length of a word except to indicate that it is quite small in relation to the number of bits in memory (an acceptable extreme case is that of words consisting of single bits). The words are said to have "locations" in memory and are the basic elements on which the machine acts; indeed it is the words which we may assume are directly addressable, rather than the bits themselves.

The computer acts on the words by performing "elementary operations". These are the usual machine language operations and include: zeroing a word; moving a word from one location to another; performing logical operations on two words such as "and", "or", "nor", etc.; performing arithmetic operations on two words such as "add", "subtract", "multiply", "divide", etc.; comparison

of two words with a branch in the control of the "program" of the computer depending on the outcome of the comparison. This is not a complete list, but it may be filled in by any reader with machine language experience. One word of warning, however: single instructions which act on a "field" of words such as moving an arbitrarily large contiguous block of memory from one location to another can not be thought of as elementary operations. The reason for this is that in the present early stage of development of algorithmic complexity, the time taken by an algorithm is often estimated by counting the number of elementary operations the algorithm performs. The elementary operations are assumed to take about the same length of time, up to a reasonable constant factor of multiplication. (This crude assumption is not always used: see S. Winograd, [1].) An instruction which acts on a field of words is really a machine implemented macro-instruction which may take an arbitrarily greater time to apply than the other operations mentioned above. This automatically excludes it from our list of elementary operations.

Now that we have imposed a structure on the memory of our standard computer by segmenting it into words, we note that this structure is not essential to the theory we are developing! Recall that words which contained only one bit were said to be an acceptable special case; in this circumstance the only elementary operations are those which are normally implicit in register arithmetic: zeroing a bit, and, nor, or, not, move, and test if bit is one with branch. Using these we may segment core into finite length words and create "macro-operations" which will emulate the elementary operations used for words, above. While it is true that it will now take much longer to multiply two words than to add them, the time differential is constant, a function of the word length which is very small in comparison to the total memory size.

In this theory we are concerned almost entirely with the asymptotic behavior, up to a multiplication constant, of the time required by an algorithm: e.g. the time required to multiply two n x n matrices using the standard method in a general purpose computer is cn^3. The constant may only be determined when one is confronted by a specific computer, and is best derived by a simple test: the time taken to multiply two n x n matrices where n is known quickly gives the value of c. The asymptotic behavior, up to a constant of the time taken by an algorithm is adequately estimated by counting the number of elementary operations performed. The estimate is not sensitive to the difference between the usual machine and the one bit word machine which emulates it. We shall continue to speak of memory as consisting of words and we will use standard elementary operations, but we must be clear that this is a convenience and not a basic assumption of the model.

The following assumption, on the other hand, is entirely basic. In the operation of the standard computer we stipulate that the elementary operations must be applied serially, that is, in strict consecutive order. The necessity for this assumption for a model which purports to mirror the behavior of general purpose computers in common use is obvious. Drawbacks exist however. There are special purpose computers in existence (such as array processors) whose performance is not restricted by this assumption.* From a standpoint of applications, the theory of algorithmic complexity cannot afford to ignore such special purpose computers. From a more theoretical standpoint, the investigations of S. Winograd [2,3] which give

---

*However sequential operation is canonical in the following sense: The number of connections (from memory to accumulator) grows linearly with the size of memory if operation is sequential; for parallel processors the number of connections must grow at a much greater rate.

lower bounds on the times to perform register arithmetic would fit in the framework of standard computers if parallel computation were permitted. We must, however, focus our attention on a computer with serial operation. It is true that we are thereby ignoring a byway of algorithmic complexity which permits parallel operation, but the two alternative assumptions are so basic and give rise to such different models that we feel any attempt to investigate both in the same work would cause unnecessary confusion and complication.

We sum up the discussion above in a definition:

Definition: A "standard computer" is a sequential machine with finite direct access binary memory.

The assumption of finite memory was made implicitly when we spoke of finite word length which was "small in comparison to the total size of memory". In actual practice, we shall assume that memory is large enough to hold the intermediate and final results of the algorithms we will be considering; putting this another way, we shall have to guard against considering algorithms which require a ridiculously large amount of memory. We illustrate this by an example.

Example 1: We are given the task of multiplying two n x n (0,1)-matrices. We may assume that the task will be performed a tremendous number of times, so that any amount of "precalculation" which will simplify the task for a proposed algorithm is justified. We proceed as follows: multiply all possible pairs of n x n (0,1)-matrices in the standard way and store the results in memory. Now given any two matrices to multiply, we need merely "look up" the product, calculating the address where the result was stored by any of a number of schemes based on the entries of the two matrices to be multiplied. This calculation only takes cn^2 operations, which may be shown to be best possible from a standpoint of information theory (more on this later). However we require $n^2 2^{2n^2}$ words of memory to store all these matrices, which is clearly impractical for any large value of n.

There can be no hard and fast rule by which one excludes algorithms from our theory because of memory considerations. A. Meyer and M. Fischer [4]

use a variant on the above scheme to improve the performance of Strassen's method in the multiplication of (0,1)-matrices; however they do not assume that precalculation is "free", so their variant has intrinsic value.

We note that a restriction was placed on word length: that it should be small in comparison to the size of memory. We should at the same time keep some common-sense absolute bound on the word length. We are trying here to abstract a model of general purpose computers which is independent of the length of a word in a particular case; the argument given above, that we may emulate a standard computer with words of constant length c using one bit words, breaks down if c is allowed to grow without bound. The reason for this is that the parallel processing of register arithmetic must be emulated by sequential commands, and different elementary operations such as addition and multiplication are intrinsically different in their complexity of computation in a serial processor (see S.A. Cook, [5]). This fact also holds true for the parallel computation performed in registers [3,4], and the usefulness of the concept of elementary operations would be destroyed if word size were allowed to grow without bound.

The argument we have made in favor of restricting the size of words, that the parallel nature of computation in word arithmetic is poorly simulated by serial one bit word operations, may be viewed from a different direction. One of our most basic assumptions is the serial operation of a standard computer; permitting arbitrarily large bit strings to be operated on in parallel weakens this assumption. I am grateful to R. Berlekamp for the following illustrative example.

Example 2: We are given the task of counting the number of 1^{s}'s in a string of bits which we may assume fits in a word, w. Because of the seriality of computation, it seems intuitively

obvius that we must view each bit position in the string, incrementing a counter when a bit turns out to be 1. This problem was given to machine language programmers on a 32-bit word machine, and almost every solution submitted was of this form. However, consider the following solution.

We first define a number of constant "masks":

a_1 = 10101010...1010
b_1 = 01010101...0101
a_2 = 11001100...1100
b_2 = 001100110011...0011

where a_k is a concatenation of the string

$$\underbrace{111\ldots1}_{2^{k-1}}\underbrace{000\ldots0}_{2^{k-1}},$$

repeated the number of times required until a 32-bit word has been constructed. The words b_k are defined as NOT(a_k), i.e. all bits are reversed. Clearly $k \leq 5$.

The following process is used:

```
        x ← w   the word whose 1 bits we wish to count.
        i ← 0
Proceed i ← i + 1
        y ← x.and.a_i
        z ← x.and.b_i
        j ← 2**(i-1)
        y ← RIGHT SHIFT j(y)
        x ← y + z
        if i.LT.5, go to proceed.
```

It is left as a simple exercise that when the loop is exhausted, x is the number of 1-bits in the word w. Suppose that the word length were $N = 2^k$. Then the number of 1-bits in an N bit word would be calculated as in the process above in c $\log_2 N$ operations instead of the c'N one would expect from a serial search.

This example may seem startling, but is merely a demonstration of how the assumption of serial computation may be circumvented if arbitrarily large words are permitted. For simplicity, we shall assume that the word length is great enough so that, for the numbers met with in our tasks, we need take no precautions to guard against overflow.

§3 Minimal Time for an Algorithm

We speak rather glibly of the time taken by an algorithm to perform a task, but there is an unfortunate lack of precision in this concept. It springs from the fact that the prescription of a task is vague: typically we are supplied with input in a certain format (e.g.: a list to be ordered) and the "task"is to process the input to arrive at a well-defined output (e.g., the ordered list).

Given an algorithm to perform some task, <u>together with a specific input</u>, the time taken by the algorithm may be arrived at empirically. However, given a <u>different input</u> we may find that the time taken to perform the same algorithm is different. Let us make it clear that we are not making a trivial distinction e.g.: it takes longer using most algorithms to order a list of 1000 words than it does for a list of 10 words. We may define the task more precisely by letting T_n be the task of ordering a list of n words. Indeed we shall usually have this dependence on a parameter n. However, even for tasks that are defined this explicitly, algorithms may vary in their execution time according to what input is given; this is a function of the implicit structure of the input. We illustrate this idea by the following example.

<u>Example 1</u>: We are given a list of n integers X(1),X(2),...,X(n). The task is to order this list, so that X(1) becomes the smallest element in the list, X(2) the second smallest, ..., X(n) the largest. We do this using the algorithm INTERCHANGE SORT.

```
          PASS ← 0
NEWPASS   PASS ← PASS + 1
          I ← 0
          COUNT ← 0          ...initialize for pass
PROCEED   I ← I + 1
          IF X(I+1).GT.X(I)go to CHECK, else
          DUMMY ← X(I)
          X(I) ← X(I+1)      ...perform interchange
          X(I+1) ← DUMMY
          COUNT = COUNT + 1  ...count interchange
CHECK     IF I.LT.n-PASS,go to PROCEED, else
          IF COUNT.EQ.0, HALT, else
          IF PASS.LT.n,go to NEWPASS, else
          HALT.
```

This algorithm is in very common use for ordering short lists and we shall not explain its operation. We do note that a count is kept on the number of interchanges in each pass, and the algorithm halts when a pass is completed with no interchange made. This feature is used only occasionally, and is in fact inefficient for most applications.

We now ask the anticipated question: how long does this algorithm take to perform its task? Not surprisingly, the answer depends on the form of the input. Let us assume the array X(1),X(2),...,X(n) is some permutation of the integers 1,2,...,n. Now if the array X consists of the integers 1,2,...,n in reversed order, it should be clear that the maximum number, n, of passes will be made and further that the interchange of X(I) with X(I+1) will never be "skipped" by the IF statement which precedes it. The number of elementary operations in pass k is given by the number of elementary operations between the statements labeled PROCEED and CHECK inclusive (i.e., seven), multiplied by the number of values i takes on in the $k^{\underline{th}}$ pass (n-k). The remaining statements, being outside the central loop, may be ignored as insignificant and we estimate the number of elementary operations performed by the algorithm with this input as

$$7[(n-1) + (n-2) + \ldots + 2 + 1] = \frac{7(n)(n-1)}{2} \quad .$$

On the other hand, if the array X consists of the integers 1,2,3,...,n in increasing order, then in the first pass no interchanges will be made, the count will remain zero, and we will halt after the first pass having performed 3(n-1) operations in the central loop; we add 5 to this number to get the total number of operations performed.

As we have stated before, the tasks we are concerned with are "parametrized" by an integer n (multiplying n x n matrices; evaluating an $n^{\underline{th}}$ degree polynomial) and we study algorithms from a standpoint of the asymptotic behavior, up to a multiplicative constant, of the time they need for execution. INTERCHANGE SORT is an example of what we call a "high variance" algorithm: its

asymptotic behavior is not defined, for with different inputs it may take cn to $c'n^2$ elementary operations to execute. There are two approaches in general use to give some meaning to the concept of "the time taken" by a high variance algorithm.

We first consider the technique of worst case analysis: what is the greatest length of time that an algorithm may take when supplied with any conceivable input of the right format. It is easily seen that the worst possible input for INTERCHANGE SORT is the list of integers in inverse order, which we have analyzed, and the algorithm is said to take cn^2 steps.

A second type of technique often employed to find the time taken by high variance algorithms is that of probabilistic analysis. Here we assume that all possible inputs to an algorithm have a known probability; for each input we evaluate the time taken by the algorithm, t_a (input), and we calculate the expected time which the algorithm will take, $E(t_a)$. Although this calculation may seem to be so incredibly difficult that it is not possible in any but the most elementary algorithms, this impression is incorrect; a truly beautiful application of probabilistic analysis of this type is given in [7], which deals with searching in dynamically changing binary tree structured lists. In the particular case of INTERCHANGE SORT given above, assuming all permutations of the integers 1,2,...,n are equally likely inputs, the expected time the algorithm will take, $E(t_a)$, is cn^2 steps. (We do not prove this, but it is not difficult.)

It is not at all common that worst case analysis and probabilistic analysis give the same asymptotic time estimate, as they do in the case of INTERCHANGE SORT. The expected time for a search in the binary tree structured list referred to in [7] is $c \log_2 n$, where n is the number of entries in the list; but a worst case analysis gives cn as the time taken by the search algorithm! Which

estimate should one believe?

Certainly both approaches have their merits and neither should be discarded in favor of the other. One of the advantages of studying the expected time to perform an algorithm of high variance type is that it gives the researcher something to aim at: can an algorithm be found to perform the same task whose worst time is of the same order of magnitude as the expected time of the known high variance algorithm? In the case of the binary tree structured list search this was done! It was achieved by Adel'son-Vel'skiy and Landis in Moscow and reported on with some improvements by C.C. Foster [8]. It is probably that the motivation was supplied by a probabilistic analysis carried out by P.F. Windley [9] who developed independently many of the same results as those in [7]. Without such an analysis, which showed the promise of binary tree structures for sorting and searching, the improved algorithm (with worst time for search c $\log_2$ n) might not have been found.

There is a school of thought which holds that time estimates derived by probabilistic analysis of algorithms serve no utilitarian purpose in application This attitude is justified in cases where the basic assumption of probabilistic analysis is suspect, e.g. all inputs to an algorithm are not equally probable. If we may assign probabilities (even though not equal) to the various possible inputs, a probabilistic analysis is still feasible, however: the expected time for the execution of the algorithm can sometimes be derived.

A much more basic objection exists to probabilistic analysis estimates, however, which is quite difficult to answer. The objection is to a large extent subjective but may be expressed as follows: how can one trust an algorithm whic has good expected time for performance? What if, while performing a sequence of important calculations, a high variance algorithm with excellent expected performance continually receives input which shows it down to a performance time close to what the worst case analysis would predict? Would it not be

better to use an algorithm which has a better worst case time estimate?

Certainly, in very special cases, such as real time control applications where time need not be optimized but only kept within an absolute bound, such caution is justified. But consider the hash coding algorithm as it is usually applied, where the hashing function is not calculated with a particular list of key-words in mind. A probabilistic analysis of this algorithm reveals that under most circumstances it rivals associative memory for speed of "look-up" of a key-word. But a worst case analysis would consider only the case where all key-words are hashed to the same location, and the algorithm would be assigned the same execution time as linear search, cn operations. The AVL tree structure referred to in [8] would match any key-word in at most c log n operations, as would logarithmic search (the latter algorithm however must be performed on an ordered list, which is not amenable to insertions and deletions). It is doubtful that many programmers would use logarithmic search in place of hash coding, in spite of the terrible performance of hash coding revealed by worst case analysis.

So far in §3, we have been concerned mainly with "high variance algorithms" which we have treated by example. We should make this concept more concrete. Let a be an algorithm parametrized by an integer n. Denote the time taken by the algorithm a in processing some acceptable input, α, as $t_a^n(\alpha)$. Assume there exists a function f(n) and two constants, c_1 and c_2, not depending on n, such that:

$$c_1 f(n) \leq t_a^n(\alpha) \leq c_2 f(n) \quad , \tag{3.1}$$

for any acceptable input α. Then the algorithm a is defined as a <u>low variance</u> algorithm; we say that the time taken by the algorithm is $c_2 f(n)$. Any algorithm which does not have this property is defined as a <u>high variance</u> algorithm.

There are many examples of low variance algorithms which come immediately to mind, but the necessity for two constants, c_1 and c_2 to bound the time taken by an algorithm may not at first be obvious. It might seem that either an algorithm has high variance or else the time it takes to perform its task is independent of its input. To see that this is not so, one need merely consider the INTERCHANGE SORT algorithm of Example 1 with all instructions containing the variable COUNT removed. Then the number of elementary operations required by this algorithm to sort the array $(n,n-1,n-2,\ldots,3,2,1)$ is about $3n^2$, but to sort the array $(1,2,3,\ldots,n-1,n)$, it is about $\frac{3}{2}n^2$.

This is one of the major reasons that in speaking of the time taken by an algorithm, we are concerned with asymptotic behavior only up to a constant of multiplication! Another reason which has already been mentioned is that we wish our analysis to have as much generality as possible. Our basic assumption that all elementary operations take the same length of time is necessary for any kind of general theory, but it is not precise; it is true only up to a multiplicative constant.

§4 Toward a More Rigorous Theory

In this section, we have tried to develop a model, called the standard computer, which abstracts the essential features of general purpose computers in most common use today. By "essential" of course, we mean essential to our purpose of estimating the time taken by algorithms in important applications. Hopefully, it is clear to the reader that this model has been very carefully constructed; at each point that a difference of opinion might arise, as in the question of whether memory could be taken to be infinite, we have tried to show by example why one course is better than another. We could go on at great length in this vein, as the most severe limitation we feel in this writing is the necessity of limiting the number of relevant examples so as not to lose the impetus of our presentation in a welter of detail.

The standard computer we have outlined seems to lack the most important feature of a model, that of simplicity. This ignores, however, what is probably the most far-reaching conclusion of this section: that a one-bit word standard computer with register arithmetic instructions, may emulate (and be emulated by) a standard computer with thirty or forty bit words and an extensive instruction set; and this without changing the asymptotic time to perform algorithms except for the constant of multiplication. A one-bit word standard computer may be defined about as simply as a Turing machine, and its programming language should be hardly more difficult. Furthermore, in the next section we shall present one of the basic approaches to proofs in computational complexity: the use of information theory to bound below the number of yes-no questions answered by any algorithm which performs a given task. This puts a measure of complexity on the task and requires at least as many elementary operations to perform the task as there are yes-no questions answered. One of the hin-

drances to this approach has been the fact that many elementary operations are hard to put into the framework of information theory: there seem to be no yes-no questions answered in their performance. In the more basic framework of one-bit word standard computers, it becomes apparent that operations such as multiplication and addition do indeed require test and branch instructions (yes-no question). In fact, we may formulate the instruction set of the one-bit word computer so that the instructions "and", "or", "nor", etc. are emulated by two test and branch instructions. Thus the only elementary operations needed for a one-bit word standard computer are: set bit zero, set bit one, move data, and test if bit is one with branch. The question of address arithmetic requires some thought.

We do not try to develop a theory of one-bit word standard computers in this work; it is felt that such an undertaking would be premature since our model is new and may require revision. Some time for consolidation and possible correction of these concepts should be allotted (and here we look to readers for suggestions) before any attempt is made to codify them.

Although we shall deal mainly with many-bit word standard computers in the following sections, we feel that the future of algorithmic complexity lies in the study of how tasks may be performed in some basic, elemental environment such as we have outlined. Research is presently being jointly undertaken by L.H. Harper (U.C. at Riverside) and J.E. Savage (Brown) based on the work of Subbotovskoya [10] and Nečiporuk [11]. This work studies the necessary length of the Boolean representation of specific functions; it is assumed that the length is proportional to the time needed for evaluation. A possible drawback to this formulation is that branching is not permitted within the Boolean representation. This objection may be surmountable, however, and we hold out great hope for this approach.

References for Section I

1. Winograd, S., On the number of multiplications necessary to compute certain functions, I.B.M. Research Report RC2285 (Nov. 1968).

2. ----------, On the time required to perform addition, Journal of the A.C.M. 12, 2 (April 1965), 277-285.

3. ----------, On the time required to perform multiplication, Journal of the A.C.M. 14, 4 (Oct. 1967), 793-802.

4. Meyer, A. and Fischer, M., Boolean matrix multiplication and transitive closure, In draft.

5. Cook. S.A., On the minimum computation time of functions, doctoral dissertation, Harvard, 1966.

6. Dreyfus, S.E., An appraisal of some shortest path algorithms, Rand Corporation Memorandum RM-5433-PR, October 1967.

7. Hibbard, T.N., Some combinatorial properties of certain trees, with applications to searching and sorting,

8. Foster, C.C., A study of AVL trees, GFR-12158, Goodyear Aerospace Corporation, Akron, Ohio, 1965.

9. Windley, P.F., Trees, forests, and rearranging, Comp. J., Vol. 3, No. 2, (1960), 84-88.

10. Subbotovskaya, B.A.,,Realization of linear functions using V, &, -, Soviet Math. Dokl. 2 (1961), 110.

11. Nečiporuk, È.I., A Boolean Function, Soviet Math. Dokl. 7 (1966), 999-1000.

12. Cooley, J.W. and Tukey, J.W., An Algorithm for the machine calculation of complex Fourier series, Mathematics of Comp. Vol. 19, No. 90 (1965) 297-301.