The 1976 ACM Turing Award was presented jointly to Michael A. Rabin and Dana S. Scott at the ACM Annual Conference in Houston, October 20. In introducing the recipients, Bernard A. Galler, Chairman of the Turing Award Committee, read the following citation:

"The Turing Award this year is presented to Michael Rabin and Dana Scott for individual and joint contributions which not only have marked the course of theoretical computer science, but have set a standard of clarity and elegance for the entire field.

Rabin and Scott's 1959 paper, 'Finite Automata and Their Decision Problems' has become a classic paper in formal language theory that still forms one of the best introductions to the area. The paper is simultaneously a survey and a research article; it is technically simple and mathematically impeccable. It has even been recommended to undergraduates!

In subsequent years, Rabin and Scott have made individual contributions which maintain the standards of their early paper. Rabin's applications of automata theory to logic and Scott's development of continuous semantics for programming languages are two examples of work providing depth and dimension: the first applies computer science to mathematics, and the second applies mathematics to computer science.

Rabin and Scott have shown us how well mathematicians can help a scientist understand his own subject. Their work provides one of the best models of creative applied mathematics."

That was the formal citation, but there is a less formal side to this presentation. I want you to understand that the recipients of this award are real people, doing excellent work, but very much like those of us who are here today. Professor Michael Rabin was born in Germany and emigrated as a small child with his parents to Israel in 1935. He got a MSc. degree in Mathematics from the Hebrew University and later his Ph.D. in Mathematics from Princeton University. After obtaining his Ph.D. he was an H.B. Fine Instructor in Mathematics at Princeton University and Member of the Institute for Advanced Studies at Princeton. Since 1958 he has been a faculty member at the Hebrew University in Jerusalem. From 1972 to 1975 he was also Rector of the Hebrew University. The Rector is elected by the Senate of the University and is Academic Head of the institution.

Professor Dana S. Scott received his Ph.D. degree at Princeton University in 1958. He has since taught at the University of Chicago, the University of California at Berkeley, Stanford University, University of Amsterdam, Princeton University and Oxford University. His present title is Professor of Mathematical Logic at Oxford University in England.

Professor Rabin will speak on "Computational Complexity," and Professor Scott will speak on "Logic and Programming Languages."

Rabin's paper begins below; Scott's paper begins on page 634.

Michael O. Rabin

Dana S. Scott

Complexity of Computations

Michael O. Rabin
Hebrew University of Jerusalem

The framework for research in the theory of complexity of computations is described, emphasizing the interrelation between seemingly diverse problems and methods. Illustrative examples of practical and theoretical significance are given. Directions for new research are discussed.

Key Words and Phrases: complexity of computations, algebraic complexity, intractable problems, probabilistic algorithms

CR Categories: 5.25

Copyright © 1977, Association for Computing Machinery, Inc. General permission to republish, but not for profit, all or part of this material is granted provided that ACM's copyright notice is given and that reference is made to the publication, to its date of issue, and to the fact that reprinting privileges were granted by permission of the Association for Computing Machinery.

Author's address: Department of Mathematics, Hebrew University of Jerusalem, Jerusalem, Israel.

1. Introduction

The theory of complexity of computations addresses itself to the quantitative aspects of the solutions of computational problems. Usually there are several possible algorithms for solving a problem such as evaluation of an algebraic expression, sorting a file, or parsing a string of symbols. With each of the algorithms there are associated certain significant cost functions such as the number of computational steps as a function of the problem size, memory space requirements for the computation, program size, and in hardware implemented algorithms, circuit size and depth.

The following questions can be raised with respect to a given computational problem P. What are good algorithms for solution of the problem P? Can one establish and prove a lower bound for one of the cost functions associated with the algorithm? Is the problem perhaps intractable in the sense that no algorithm will solve it in practically feasible time? These questions can be raised for worst-case behavior as well as for the average behavior of the algorithms for P. Another distinction is the one between sequential and parallel algorithms for P. During the last year an extension of algorithms to include randomization within the computation was proposed. Some of the above considerations can be generalized to these probabilistic algorithms.

These questions concerning complexity were the subject of intensive study during the last two decades both within the framework of a general theory and for specific problems of mathematical and practical importance. Of the many achievements let us mention the Fast Fourier Transform, recently significantly improved, with its manifold applications including those to communications:

- Showing that some of the mechanical theorem proving problems arising in proving the correctness of programs, are intractable;
- Determining the precise circuit complexity needed for addition of n-bit numbers;
- Surprisingly fast algorithms for combinatorial and graph problems and their relation to parsing;
- Considerazle reductions in computing time for certain important problems, resulting from probabilistic algorithms.

There is no doubt that work on all the abovementioned problems will continue. In addition we see for the future the branching out of complexity theory into important new areas. One is the problem of secure communication, where a new, strengthened theory of complexity is required to serve as a firm foundation. The other is the investigation of the cost functions pertaining to data structures. The enormous size of the contemplated databases calls for a deeper understanding of the inherent complexity of processes such as the construction and search of lists. Complexity theory provides the point of view and the tools necessary for such a development.

The present article, which is an expanded version of the author's 1976 Turing lecture, is intended to give the reader a bird's-eye view of this vital field. We shall focus our attention on highlights and on questions of methodology, rather than attempt a comprehensive survey.

2. Typical Problems

We start with listing some representive computational problems which are of theoretical and often also of practical importance, and which were the subject of intensive study and analysis. In subsequent sections we shall describe the methods brought to bear on these problems, and some of the important results obtained.

2.1 Computable Functions from Integers to Integers

Let us consider functions of one or more variables from the set $N = \{0, 1, 2, \ldots\}$ of integers into N. We recognize intuitively that functions such as $f(x) = x!$, $g(x, y) = x^2 + y^x$ are computable.

A.M. Turing, after whom these lectures are so aptly named, set for himself the task of defining in precise terms which functions $f: N \to N$, $g: N \times N \to N$, etc. are effectively computable. His model of the idealized computer and the class of recursive functions calculable by this computer are too well known to require exposition.

What concerns us here is the question of measurement of the amount of computational work required for finding a value $f(n)$ of a computable function $f: N \to N$. Also, is it possible to exhibit functions which are difficult to compute by every program? We shall return to these questions in 4.1.

2.2 Algebraic Expressions and Equations

Let $E(x_1, \ldots, x_n)$ be an algebraic expression constructed from the variables $x_1, \ldots, x_n$ by the arithmetical operations $+, -, *, /$. For example, $E = (x_1 + x_2) * (x_3 + x_4)/x_1 * x_5$. We are called upon to evaluate $E(x_1, \ldots, x_n)$ for a numerical substitution $x_1 = c_1, \ldots, x_n = c_n$. More generally, the task may be to evaluate k expressions $E_1(x_1, \ldots, x_n), \ldots, E_k(x_1, \ldots, x_n)$, for the simultaneous substitution $x_1 = c_1, \ldots, x_n = c_n$.

Important special cases are the following. Evaluation of a polynomial

$$f(x) = a_n x^n + a_{n-1}x^{n-1} + \ldots + a_0. \tag{1}$$

Matrix multiplication AB, where A and B are $n \times n$ matrices. Here we have to find the values of the n^2 expressions $a_{i1}b_{1j} + \ldots + a_{in}b_{nj}$, $1 \le i, j \le n$, for given numerical values of the a_{ij}, b_{ij}.

Our example for the solution of equations is the system

$$a_{il}x_1 + \ldots + a_{in}x_n = b_i, \quad 1 \le i \le n, \tag{2}$$

for n linear equations in n unknowns $x_1, \ldots, x_n$. We have to solve (evaluate) the unknowns, given the coefficients a_{ij}, b_i, $1 \le i, j \le n$.

We shall not discuss here the interesting question of *approximate solutions* for algebraic and transcendental equations, which is also amenable to the tools of complexity theory.

2.3 Computer Arithmetic

Addition. Given two n-digit numbers $a = \alpha_{n-1}\alpha_{n-2} \ldots \alpha_0$, $b = \beta_{n-1}\beta_{n-2} \ldots \beta_0$ (e.g. for $n = 4$, $a = 1011$, $b = 1100$), to find the $n + 1$ digits of the sum $a + b = \gamma_n\gamma_{n-1} \ldots \gamma_0$.

Multiplication. For the above a, b, find the $2n$ digits of the product $a * b = \delta_{2n}\delta_{2n-1} \ldots \delta_0$.

The implementation of these arithmetical tasks may be in *hardware*. In this case the base is 2, and α_i, $\beta_i = 0, 1$. Given a fixed n we wish to construct a circuit with $2n$ inputs and, for addition, $n + 1$ outputs. When the $2n$ bits of a, b enter as inputs, the $n + 1$ outputs will be $\gamma_n, \gamma_{n-1}, \ldots, \gamma_0$. Similarly for multiplication.

Alternatively we may think about implementation of arithmetic by an algorithm, i.e. in *software*. The need for this may arise in a number of ways. For example, our arithmetical unit may perform just addition, multiplication must then be implemented by a subroutine.

Implementation of arithmetic by a program also

comes up in the context of *multiprecision arithmetic*. Our computer has word size k and we wish to add and multiply numbers of length nk (n-word numbers). We take as base the number 2^k, so that $0 \leq \alpha_i, \beta_i < 2^k$, and use algorithms for finding $a + b$, $a * b$.

2.4 Parsing Expressions in Context-Free Languages

The scope of complexity theory is by no means limited to algebraic or arithmetical calculations. Let us consider *context-free grammars* of which the following is an example. The *alphabet* of G consists of the symbols t, x, y, z, (,), $+$, $*$. Of these symbols t is a *nonterminal* and all the other symbols are *terminals*. The *productions* (or rewrite rules) of G are

1. $t \to (t + t)$, 2. $t \to t * t$, 3. $t \to x$,
4. $t \to y$, 5. $t \to z$.

Starting from t, we can successively rewrite words by use of the productions. For example,

$$\bar{t} \to^1 (\bar{t} + t) \to^3 (x + \bar{t}) \to^2 (x + \bar{t} * t) \to^4 \cdot(x + y * \bar{t}) \to^5 (x + y * z). \quad (3)$$

The number above each arrow indicates the production used, and $\bar{t}$ stands for the nonterminal to be rewritten. A sequence such as (3) is called a *derivation*, and we say that $(x + y * z)$ is derivable from t. The set of all words u derivable from t and containing only terminals, is called the *language generated* by G and is denoted by $L(G)$. The above G is just an example, and the generalization to arbitrary context-free grammars is obvious.

Context-free grammars and languages commonly appear in programming languages and, of course, also in the analysis of natural languages. Two computational problems immediately come up. Given a grammar G and a word W (i.e. string of symbols) on the alphabet of G, is $W \in L(G)$? This is the *membership* problem.

The *parsing problem* is the following. Given a word $W \in L(G)$, find a derivation sequence by productions of G, similar to (3), of W from the initial symbol of G. Alternatively, we want a *parse tree*, of W. Finding a parse tree of an algebraic expression, for example, is an essential step in the compilation process.

2.5 Storing of Files

A file of records $R_1, R_2, \ldots, R_n$ is stored in either secondary or main memory. The index i of the record R_i indicates its location in memory. Each record R has a key (e.g. the social-security number in a income-tax file) $k(R)$. The computational task is to rearrange the file in memory into a sequence $R_{i_1}, \ldots, R_{i_n}$ so that the keys are in ascending order

$$k(R_{i_1}) < k(R_{i_2}) < k(R_{i_n}).$$

We emphasize both the distinction between the key and the record, which may be considerably larger than the key, and the requirement of actually rearranging the records. These features make the problem more realistic and somewhat harder than the mere sorting of numbers.

2.6 Theorem Proving by Machine

Ever since the advent of computers, trying to endow them with some genuine powers of reasoning was an understandable ambition resulting in considerable efforts being expended in this direction. In particular, attempts were made to enable the computer to carry out logical and mathematical reasoning, and this by proving theorems of pure logic or by deriving theorems of mathematical theories. We consider the important example of the theory of addition of natural numbers.

Consider the system $\mathcal{N} = \langle N, + \rangle$ consisting of the natural numbers $N = \{0, 1, \ldots\}$ and the operation $+$ of addition. The formal language L employed for discussing properties of $\mathcal{N}$ is a so-called first-order predicate language. It has variables $x, y, z, \ldots$ ranging over natural numbers, the operation symbol $+$, equality $=$, the usual propositional connectives, and the quantifiers $\forall$ ("for all") and $\exists$ ("there exists").

A sentence such as $\exists x \forall y[x + y = y]$ is a formal transcription of "there exists a number x so that for all numbers y, $x + y = y$." This sentence is in fact true in $\mathcal{N}$.

The set of all sentences of L true in $\mathcal{N}$ will be called the *theory* of $\mathcal{N}$ ($Th(\mathcal{N})$) and will be denoted by $PA = Th(\mathcal{N})$. For example,

$$\forall x \forall y \exists z[x + z = y \vee y + z = x] \in PA.$$

We shall also use the name "Presburger's arithmetic," honoring Presburger, who has proved important results about $Th(\mathcal{N})$.

The *decision problem* for PA is to find an algorithm, if indeed such an algorithm exists, for determining for every given sentence F of the language L whether $F \in PA$ or not.

Presburger [12] has constructed such an algorithm for PA. Since his work, several researchers have attempted to devise efficient algorithms for this problem and to implement them by programs. These efforts were often within the framework of projects in the area of automated programming and program verification. This is because the properties of programs that one tries to establish are sometimes reducible to statements about the addition of natural numbers.

3. Central Issues and Methodology of Computational Complexity

In the previous section we listed some typical computational tasks. Later we shall present results which were obtained with respect to these problems. We shall now describe in general terms the main questions that are raised, and the central concepts that play a role in complexity theory.

3.1 Basic Concepts

A class of similar computational tasks will be called a *problem*. The individual cases of a problem P are

called *instances* of P. Thus P is the set of all its instances. The delineation of a problem is, of course, just a matter of agreement and notational convenience. We may, for example, talk about the problem of matrix multiplication. The instances of this problem are, for any integer n, the pairs A, B of $n \times n$ matrices which are to be multiplied.

With each instance $I \in P$ of a problem P we associate a size, usually an integer, $|I|$. The size function $|I|$ is not unique and its choice is dictated by the theoretical and practical considerations germane to the discussion of the problem at hand.

Returning to the example of matrix multiplication, a reasonable measure on a pair $I = (A, B)$ of $n \times n$ matrices to be multiplied, is $|I| = n$. If we study memory space requirement for an algorithm for matrix multiplication, then the measure $|I| = 2n^2$ may be appropriate. By way of contrast, it does not seem that the size function $|I| = n^n$ would naturally arise in any context.

Let P be a problem and AL an algorithm solving it. The algorithm AL executes a certain computational sequence S_I when solving the instance $I \in P$. With S_I we associate certain measurements. Some of the significant measurements are the following: (1) The length of S_I, which is indicative of computation time. (2) Depth of S_I, i.e. the number of layers of concurrent steps into which S_I can be decomposed. Depth corresponds to the time S_I would require under parallel computation. (3) The memory space required for the computation S_I. (4) Instead of total number of steps in S_I, we may count the number of steps of a certain kind such as arithmetical operations in algebraic computations, number of comparisons in sorting, or number of fetches from memory.

For hardware implementations of algorithms, we usually define the size $|I|$ so that all instances I of the same size n are to be solved on one circuit C_n. The complexity of a circuit C is variously defined as number of gates; depth, which is again related to computing time; or other measurements, such as number of modules, having to do with the technology used for implementing the circuit.

Having settled on a measure μ on computations S, a complexity of computation function F_{AL} can be defined in a number of ways, the principal two being *worst-case* complexity and *average-behavior* complexity. The first notion is defined by

$$F_{AL}(n) = \max \{\mu(S_I) | I \in P, |I| = n\}. \quad (4)$$

In order to define average behavior we must assume a probability distribution p on each set $P_n = \{I | I \in P, |I| = n\}$. Thus for $I \in P$, $|I| = n$, $p(I)$ is the probability of I arising among all other instances of size n. The *average behavior* of AL is then defined by

$$M_{AL}(n) = \sum_{I \in P_n} p(I)\ \mu(S_I). \quad (5)$$

We shall discuss in 4.7 the applicability of the assumption of a probability distribution.

The *analysis* of algorithms deals with the following question. Given a size-function $|I|$ and a measure $\mu(S_I)$ on computations, to exactly determine for a given algorithm AL solving a problem P either the worst-case complexity $F_{AL}(n)$ or, under suitable assumptions, the average behavior $M_{AL}(n)$. In the present article we shall not enter upon questions of analysis, but rather assume that the complexity function is known or at least sufficiently well determined for our purposes.

3.2 The Questions

We have now at our disposal the concepts needed for posing the central question of complexity theory: Given a computational problem P, how well, or at what cost, can it be solved? We do not mention any specific algorithm for solving P. We rather aim at surveying *all* possible algorithms for solving P and try to make a statement concerning the inherent computational complexity of P.

It should be borne in mind that a preliminary step in the study of complexity of a problem P is the choice of the measure $\mu(s)$ to be used. In other words, we must decide on mathematical or practical grounds, *which* complexity we want to investigate. Our study proceeds once this choice was made.

In broad lines, with more detailed examples and illustrations to come later, here are the main issues that will concern us. With the exception of the last item, they seem to fall into pairs.

(1) Find efficient algorithms for the problem P.
(2) Establish lower bounds for the inherent complexity of P.
(3) Search for exact solutions of P.
(4) Algorithms for approximate (near) solutions.
(5) Study of worst-case inherent complexity.
(6) Study of the average complexity of P.
(7) Sequential algorithms for P.
(8) Parallel-processing algorithms for P.
(9) Software algorithms.
(10) Hardware-implemented algorithms.
(11) Solution by probabilistic algorithms.

Under (1) we mean the search for good practical algorithms for a given problem. The challenge stems from the fact that the immediately obvious algorithms are often replaceable by much superior ones. Improvements by a factor of 100 are not unheard of. But even a saving of half the cost may sometimes mean the difference between feasibility and nonfeasibility.

While any one algorithm AL for P yields an *upper bound* $F_{AL}(n)$ to the complexity of P, we are also interested in *lower bounds*. The typical result states that every AL solving P satisfies $g(n) \leq F_{AL}(n)$, at least for $n_0 < n$ where $n_0 = n_0(AL)$. In certain happy circumstances upper bounds meet lower bounds. The complexity for such a problem is then completely known. In any case, besides the mathematical interest in lower bounds, once a lower bound is found it guides us in the search

for good algorithms by indicating which efficiencies should not be attempted.

The idea of near-solutions (4) for a problem is significant because sometimes a practically satisfactory near-solution is much easier to calculate than the exact solution.

The main questions (1) and (2) can be studied in combination with one or more of the alternatives (3)–(11). Thus, for example, we can investigate an upper bound for the average time required for sorting by k processors working in parallel. Or we may study the number of logical gates needed for sorting n input-bits.

It would seem that with the manifold possibilities of choosing the complexity measure and the variety of questions that can be raised, the theory of complexity of computations would become a collection of scattered results and unrelated methods. A theme that we try to stress in the examples we present is the large measure of coherence within this field and the commonality of ideas and methods that prevail throughout.

We shall see that efficient algorithms for the parallel evaluation of polynomials, are translatable into circuits for fast addition of n-bit numbers. The Fast Fourier Transform idea yields good algorithms for multiprecision number multiplication. On a higher plane, the relation between software and hardware algorithms mirrors the relation between sequential and parallel computation. Present-day programs are designed to run on a single processor and thus are sequential, whereas a piece of hardware contains many identical subunits which can be viewed as primitive processors operating in parallel. The method of preprocessing appears time and again in our examples, thus being another example of commonality.

4. Results

4.1 Complexity of General Recursive Functions

In [13, 14] the present author initiated the study of classification of computable functions from integers to integers by the complexity of their computation. The framework is axiomatic so that the notions and results apply to every reasonable class of algorithms and every measure on computations.

Let K be a class of algorithms, possibly based on some model of mathematical machines, so that for every computable function $f: N \to N$ there exists an $AL \in K$ computing it. We do not specify the measure $\mu(S)$ on computations S but rather assume that μ satisfies certain natural axioms. These axioms are satisfied by all the concrete examples of measures listed in 3.1. The size of an integer n is taken to be $|n| = n$. The computation of f is a problem where for each instance n we have to find $f(n)$. Along the lines of 3.1 (4), we have for each algorithm AL for f the complexity of computation function $F_{AL}(n)$ measuring the work involved in computing $f(n)$ by AL.

THEOREM [13, 14]. *For every computable function $g: N \to N$ there exists a computable function $f: N \to \{0, 1\}$ so that for every algorithm $AL \in K$ computing f there exists a number n_0 and*

$$g(n) < F_{AL}(n), \quad \text{for } n_0 < n \tag{6}$$

We require that f be a 0-1 valued function because otherwise we could construct a complex function by simply allowing $f(n)$ to grow very rapidly so that writing down the result would be hard.

The limitation $n_0 < n$ in (6) is necessary. For every f and k we can construct an algorithm incorporating a table of the values $f(n)$, $n \leq k$, making the calculation trivial for $n \leq k$.

The main point of the above theorem is that (6), with a suitable $n_0 = n_0(AL)$ holds for *every* algorithm for f. Thus the inherent complexity of computing f is larger than g.

Starting from [14], E. Blum [1] introduced different but essentially equivalent axioms for the complexity function. Blum obtained many interesting results, including the speed-up theorem. This theorem shows the existence of computable functions for which there is no best algorithm. Rather, for every algorithm for such a function there exists another algorithm computing it much faster.

Research in this area of abstract complexity theory made great strides during the last decade. It served as a basis for the theory of complexity of computations by first bringing up the very question of the cost of a computation, and by emphasizing the need to consider and compare all possible algorithms solving a given problem.

On the other hand, abstract complexity theory does not come to grips with specific computational tasks and their measurement by practically significant yardsticks. This is done in the following examples.

4.2 Algebraic Calculations

Let us start with the example of evaluation of polynomials. We take as our measure the number of arithmetical operations and use the notation (nA, kM) to denote a cost of n additions/subtractions and k multiplications/divisions. By rewriting the polynomial (1) as

$$f(x) = (\ldots((a_n x + a_{n-1})x + a_{n-2}))x + \ldots + a_0,$$

we see that the general n-degree polynomial can be evaluated by (nA, nM). In the spirit of questions 1 and 2 in 3.2, we ask whether a clever algorithm might use fewer operations. Rather delicate mathematical arguments show that the above number is optimal so that this question is completely settled.

T. Motzkin introduced in [9] the important idea of *preprocessing* for a computation. In many important applications we are called upon to evaluate the same polynomial $f(x)$ for many argument values $x = c_1, x = c_2, \ldots$. He suggested the following strategy of preprocessing the coefficients of the polynomial (1). Calculate

once and for all numbers $\alpha(a_0, \ldots, a_n) \ldots \alpha_n(_0, \ldots, a_n)$ from the given coefficients $a_0, \ldots, a_n$. When evaluating $f(c)$ use $\alpha_0, \ldots, \alpha_n$. This approach makes computational sense when the cost of preprocessing is small as compared to the total savings in computing $f(c_1)$, $f(c_2)$, ..., i.e. when the expected number of arguments for which $f(x)$ is to be evaluated is large. Motskin obtained the following.

THEOREM. *Using preprocessing, a polynomial of degree n can be evaluated by* $(nA, ([n/2] + 2)M)$.

Again one can prove that this result is essentially the best possible. What about evaluation in parallel? If we use k processors and have to evaluate an expression requiring at least m operations, then the best that we can hope for is computation time $(m/k) - 1 + \log_2 k$.

Namely, assume that all processors are continuously busy, then $m - k$ operations are performed in time $(m/k) - 1$. The remaining k operations must combine by binary operations k inputs into one output, and this requires time $\log_2 k$ at least. In view of the above, the following result due to Munro and Paterson [10] is nearly best possible.

THEOREM. *The polynomial* (1) *can be evaluated by k processors working in parallel in time* $(2n/k) + \log_2 k + O(1)$.

With the advances in hardware it is not unreasonable to expect that we may be able to employ large numbers of processors on the same task. Brent [3], among others, studied the implications of unlimited parallelism and proved the following.

THEOREM. *Let* $E(x_1, \ldots, x_n)$ *be an arithmetical expression, where each variable* x_i *appears only once. The expression E can be evaluated under unlimited parallelism in time* $4 \log_2 n$.

Another important topic is the Fast Fourier Transform (FFT). The operation of convolution which has many applications such as to signal processing, is an example of a computation greatly facilitated by the FFT. Let $a_1, \ldots, a_n$ be a sequence of n numbers, $b_1, b_2, \ldots$, be a stream of incoming numbers. Define for $i = 1, 2, \ldots$,

$$c_i = a_1 b_i + a_2 b_{i+1} + \ldots + a_n b_{i+n-1}. \quad (7)$$

We have to calculate the values $c_1, c_2, \ldots$. From (7) it seems that the cost per value of c_i is $2n$ operations. If we compute the c_i's in blocks of size n, i.e. $c_1, \ldots, c_n$, and $c_{n+1}, \ldots, c_{2n}$, etc. using FFT, then the cost per block is about $4n \log_2$ so that the cost of a single c_i is $4 \log_2 n$.

Using a clever combination of algebraic and number-theoretic ideas, S. Winograd [20] recently improved computation times of convolution for small values of n and of the discrete Fourier transform for small to medium values of n. For $n \sim 1000$, for example, his method is about twice as fast as the convential FFT algorithm.

The obvious methods for $n \times n$ matrix multiplication and for the solution of the system (2) of n linear equations in n unknowns require about n^3 operations. Strassen [17] found the following surprising result.

THEOREM. *Two* $n \times n$ *matrices can be multiplied using at most* $4.7n^{2.81}$ *operations. A system of n linear equations in n unknowns can be solved by* $4.8n^{2.81}$ *operations.*

It is not likely that the exponent $\log_2 7 \sim 2.81$ is really the best possible, but at the time of writing of this article all attempts to improve this result have failed.

4.3 How Fast Can We Add or Multiply?

This obviously important question underwent thorough analysis. A simple fan-in argument shows that if gates with r inputs are used, then a circuit for the addition of n-bit numbers requires at least time $\log_r n$. This lower bound is in fact achievable.

It is worthwhile noticing that, in the spirit of the remarks in 3.2 concerning the analogy between parallel algorithms and hardware algorithms, one of the best results on circuits for addition (Brent [2]) employs Boolean identities which are immediately translatable into an efficient parallel evaluation algorithm for polynomials.

The above results pertain to the binary representation of the numbers to be added. Could it be that under a suitably clever coding of the numbers $0 \leq a < 2^n$, addition mod 2^n is performable in time less than $\log_r n$? Winograd [19] answered this question. Under very general assumptions on the coding, the lower bound remains $\log_r n$.

Turning to multiprecision arithmetic, the interesting questions arise in connection with multiplication. The obvious method for multiplying numbers of length n involves n^2 bit-operations. Early attempts at improvements employed simple algebraic identities and resulted with a reduction to $O(n^{1.58})$ operations.

Schönhage and Strassen [16] utilized the connection between multiplication of natural numbers and polynomial multiplication and employed the FFT to obtain the following theorem.

THEOREM. *Two n-bit numbers can be multiplied by* $O(n \log n \log \log n)$ *bit-operations.*

Attempts at lower bounds for complexity of integer multiplication must refer to a specific computation model. Under very reasonable assumptions Paterson Fischer and Meyer [11] have considerably narrowed the gap between the upper and lower bounds by showing the following.

THEOREM. *At least* $O(n \log n / \log \log n)$ *operations are necessary for multiplying n-bit numbers.*

4.4 Speed of Parsing

Parsing of expressions in context-free grammars would seem at first sight to require a costly backtracking computation. A dynamical computation which simultaneously seeks the predecessors of all substrings of the string to be parsed, leads to an algorithms requiring $O(n^3)$ steps for parsing a word of length n. The coeffi-

cient of n^3 depends on the grammar. This was for a long while the best result, even though for special classes of context-free grammars better upper bounds were obtained.

Fischer and Meyer observed that Strassen's algorithm for matrix multiplication can be adapted to yield an $O(n^{2.81}c(n))$ bit-operations algorithm for the multiplication of two $n \times n$ boolean matrices. Here $c(n) = \log n \log\log n \log\log\log n$ and is thus $O(n^\alpha)$ for every $0 < \alpha$.

Valiant [18] found that parsing is at most as complex as boolean matrix multiplication. Hence, since actually $\log_2 7 < 2.81$, the following theorem holds:

THEOREM. *Expressions of length n in the context-free language L(G) can be parsed in time $d(G)n^{2.81}$.*

We again see how results from algebraic complexity bear fruit in the domain of complexity of combinatorial computations.

4.5 Data Processing

Of the applications of complexity theory to data processing we discuss the best known example, that of sorting. We follow the formulation given in 2.5.

It is well known that the sorting of n numbers in random access memory requires about $n \log n$ comparisons. This is both the worst-case behavior of some algorithms and the average behavior of other algorithms under the assumption that all permutations are equally likely.

The rearrangement of records $R_1, R_2, \ldots, R_n$, poses additional problems because the file usually resides in some sequential or nearly sequential memory such as magnetic tape or disk. Size limitations enable us to transfer into the fast memory for rearrangement only a small number of records at a time. Still it is possible to develop algorithms for the actual reordering of the files in time $cn \log n$ where c depends on characteristics of the system under discussion.

An instructive result in this area is due to Floyd [6]. In his model the file is distributed on a number of pages $P_1, \ldots, P_m$ and each page contains k records so that P_i contains the records $R_{i1}, \ldots, R_{ik}$. For our purposes we may assume without loss of generality that $m = k$. The task is to redistribute the records so that R_{ij} will go to page P_j for all $1 \leqq i, j \leqq k$. The fast memory is large enough to allow reading in two pages P_e, P_j redistribute their records and read the pages out. Using a recursion analogous to the one employed in the FFT, Floyd proved the following.

THEOREM. *The redistribution of records in the above manner can be achieved by $k \log_2 k$ transfers into fast memory. This result is the best possible.*

The lower bound is established by considering a suitable entropy function. It applies under the assumption that within fast memory the records are just shuffled. It is not known whether allowing computations with the records, viewed as strings of bits, may produce an algorithm with fewer fetches of pages.

4.6 Intractable Problems

The domain of theorem proving by machine serves as a source of computational problems which require such an inordinate number of computational steps so as to be intractable. In attempts to run programs for the decision problem of Presburger's arithmetic (PA) on the computer, the computation terminated only on the simplest instances tried. A theoretical basis for this pragmatic fact is provided by the following result due to Fischer and Rabin [5].

THEOREM. *There exists a constant $0 < c$ so that for every decision algorithm AL for PA there is a number n_0 such that every $n_0 < n$ there is a sentence H of the language L (the language for addition of numbers) satisfying (1) $1(H) = n$, (2) AL takes more than $2^{2^{cn}}$ steps to determine whether $H \in PA$,* i.e. *whether H is true in $\langle N, +\rangle$.*

The constant c depends on the notation used for stating properties of $\langle N, +\rangle$. In any case, it is not very small. The rapid growth of the inherent lower bound $2^{2^{cn}}$ shows that even when trying to solve the decision problem for this very simple and basic mathematical theory, we run into practically impossible computations. Meyer [8] produced examples of theories with even more devastatingly complex decision problems.

The simplest level of logical deduction is the propositional calculus. From propositional variables $p_1, p_2, \ldots$, we can construct formulas such as $[p_1 \wedge \sim p_1] \vee [p_2 \wedge \sim p_1]$ by the propositional connectives. The *satisfiability problem* is to decide for a propositional formula $G(p_1, \ldots, p_n)$ whether there exists a truth-value assignment to the variables $p_1, \ldots, p_n$ so that G becomes true. The assignment $p_1 = F$ (false), $p_2 = T$ (true), for example, satisfies the above formula.

The straightforward algorithm for the satisfiability problem will require about 2^n steps for a formula with n variables. It is not known whether there exist nonexponential algorithms for the satisfiability problem.

The great importance of this question was brought to the forefront by Cook [4]. One can define a natural process of so called polynomial reduction of one computational problem P to another problem Q. If P is polynomially reducible to Q and Q is solvable in polynomial time then so is P. Two problems which are mutually reducible are called polynomially equivalent. Cook has shown that the satisfiability problem is equivalent to the so called problem of cliques in graphs. Karp [7] brings a large number of problems equivalent to satisfiability. Among them the problems of 0-1 integer programming, the existence of Hamiltonian circuits in a graph, and the integer-valued traveling-salesman problem, to mention just a few examples.

In view of these equivalences, if any one of these important problems is solvable in polynomial time then so are all the others. The question whether satisfiability is of polynomial complexity is called the $P = NP$ problem and is justly the most celebrated problem in the theory of complexity of computations.

4.7 Probabilistic Algorithms

As mentioned in 3.1, the study of the average behavior, or expected time, of an algorithm is predicated on the assumption of a probability distribution on the space of instances of the problem. This assumption involves certain methodological difficulties. We may postulate a certain distribution such as all instances being equally likely, and in a practical situation the source of instances of the problem to be solved may be biased in an entirely different way. The distribution may be shifting with time and will often not be known to us. In the extreme case, most instances which actually come up are precisely those for which the algorithm behaves worst.

Could we employ probability in computations in a different manner, one over which we have total control? A *probabilistic algorithm* AL for a problem P uses a source of random numbers. When solving an instance $I \in P$, a short sequence $r = (b_1, \ldots, b_k)$ of random numbers is generated, and these are used in AL to solve P in *exact terms*. With the exception of the random choice of r, the algorithm proceeds completely deterministically.

We say that such an AL solves P in expected time $f(n)$ if for every $I \in P$, $|I| = n$, AL solves I in expected time less or equal to $f(n)$. By expected time we mean the average of all solution times of I by AL for all possible choice sequences r (which we assume to be equally likely).

Let us notice the difference between this notion and the well known Monte-Carlo method. In the latter method we construct for a problem a stochastic process which emulates it and then measure the stochastic process to obtain an approximate solution for the problem. Thus the Monte-Carlo method is, in essence, an analog method of solution. Our probabilistic algorithms, by contrast, use the random numbers $b_1, \ldots, b_k$ to determine branchings in otherwise deterministic algorithms and produce exact rather than approximate solutions.

It may seem unlikely that such a consultation with a "throw of the dice" could speed up a computation. The present author systematically studied [15] probabilistic algorithms. It turns out that in certain cases this approach effects dramatic improvements.

The nearest pair in a set of points $x_1, \ldots, x_n \in R^k$ (k-dimensional space) is the pair x_i, x_j $i \neq j$, for which the distance $d(x_i, x_j)$ is minimal. A probabilistic algorithm finds the nearest pair in expected time $O(n)$, more rapidly than any conventional algorithm.

The problem of determining whether a natural number n is prime becomes intractable for large n. The present methods break down around $n \sim 10^{60}$ when applied to numbers which are not of a special form. A probabilistic algorithm devised by the author works in time $O(\log n)^3)$. On a medium-sized computer, $2^{400}-593$ was recognized as prime within a few minutes. The method works just as well on any other number of comparable size.

The full potential of these ideas is not yet known and merits further study.

5. New Directions

Of the possible avenues for further research, let us mention just two.

5.1 Large Data Structures

Commercial needs prescribe the creation of ever larger databases. At the same time present-day and, even more so imminent future technologies, make it possible to create gigantic storage facilities with varying degrees of freedom of access.

Much of the current research on databases is directed at the interface languages between the user and the system. But the enormous sizes of the lists and other structures contemplated would tend to make the required operations on these structures very costly unless a deeper understanding of the algorithms to perform such operations is gained.

We can start with the problem of finding a theoretical, but at the same time practically significant, model for lists. This model should be versatile enough to enable specialization to the various types of list structures now in use.

What are operations on lists? We can enumerate a few. Search through a list, garbage collection, access to various points in a list, insertions, deletions, mergers of lists. Could one systematize the study of these and other significant operations? What are reasonable cost functions that can be associated with these operations?

Finally, a deep quantitative understanding of data structures could be a basis for recommendations as to technological directions to be followed. Does parallel processing appreciably speed up various operations on data structures? What useful properties can lists be endowed with in associative memories? These are, of course, just examples.

5.2 Secure Communications

Secure communications employ some kind of coding devices, and we can raise fundamental questions of complexity of computations in relation to these systems. Let us illustrate this by means of the system of block-encoding.

In block-encoding one employs a digital device which takes as inputs words of length n and encodes them by use of a key. If x is a word of length n and z is a key (let us assume that keys are also of length n), then let $E_z(x) = y$, $1(y) = n$, denote the result of encoding x by use of the key z. A message $w = x_1, x_2, \ldots, x_k$ of length kn is encoded as $E_z(x_1)\, E_z(x_2) \ldots E_z(x_k)$.

If an adversary is able to obtain the current key z, then he can decode the communications between the parties since we assume that he is in possession of the coding and decoding equipment. He can also interject

into the line bogus messages which will decode properly. In commercial communications this possibility is perhaps even more dangerous than the breach of security.

In considering security one should take into account the possibility that the adversary gets hold of a number of messages $w_1, w_2, \ldots$, in clear text, and in encoded form $E_z(w_1), E_z(w_2), \ldots$. Can the key z be computed from this data.

It would not do to prove that such a computation is intractable. For the results of current complexity theory give us worst-case information. Thus if, say, for the majority of key-retrieval computations a lower bound of 2^n on computational complexity will be established, then the problem will be deemed intractable. But if an algorithm will discover the key in practical time in one case in a thousand then the possibilities of fraud would be unacceptably large.

Thus we need a theory of complexity that will enable us to state and prove that a certain computation is intractable in virtually every case. For example, a block-encoding system is safe if any algorithm for key-determination will terminate in practical time only on $O(2^{-n})$ of the cases. We are very far from creation of such a theory, especially at the present stage when $P = NP$ is not yet settled.

References

1. Blum, M. A machine independent theory of the complexity of recursive functions. *J. ACM 14* (1967), 322–336.

2. Brent, R.P. On the addition of binary numbers. *IEEE Trans. Comptrs. C-19* (1970), 758–759.

3. Brent, R.P. The parallel evaluation of algebraic expressions in logarithmic time. *Complexity of Sequential and Parallel Numerical Algorithms,* J.F. Traub, Ed., Academic Press, New York, 1973, pp. 83–102.

4. Cook, S.A. The complexity of theorem proving procedures. Proc. Third Annual ACM Symp. on Theory of Comptng., 1971, pp. 151–158.

5. Fischer, M.J., and Rabin, M.O. Super-exponential complexity of Presburger arithmetic. In *Complexity of Computations* (SIAM-AMS Proc., Vol. 7), R.M. Karp Ed., 1974, pp. 27–41.

6. Floyd, R.W. Permuting information in idealized two-level storage. In *Complexity of Computer Computations,* R. Miller and J. Thatcher Eds., Plenum Press, New York, 1972, pp. 105–109.

7. Karp, R.M. Reducibility among combinatorial problems. In *Complexity of Computer Computations,* R. Miller and J. Thatcher Eds., Plenum Press, New York, 1972, pp. 85–103.

8. Meyer, A.R. The inherent computational complexity of theories of order. Proc. Int. Cong. Math., Vol. 2, Vancouver, 1974, pp. 477–482.

9. Motzkin, T.S. Evaluation of polynomials and evaluation of rational functions. *Bull. Amer. Math. Soc. 61* (1955), 163.

10. Munro, I., and Paterson, M. Optimal algorithms for parallel polynomial evaluation. *J. Comptr. Syst. Sci., 7* (1973), 189–198.

11. Paterson, M., Fischer, M.J., and Meyer, A.R. An improved overlap argument for on-line multiplication. Proj. MAC Tech. Report 40, M.I.T. (1974).

12. Presburger, M. Über die Vollständigkeit eines gewissen Systems Arithmetic ganzer Zahlen in welchem die Addition als einzige Operation hervortritt. Comptes-rendus du I Congrés de Mathematiciens de Pays Slaves, Warsaw, 1930, pp. 92–101,395.

13. Rabin, M.O. Speed of computation and classification of recursive sets. Third Convention Sci. Soc., Israel, 1959, pp. 1–2.

14. Rabin, M.O. Degree of difficulty of computing a function and a partial ordering of recursive sets. Tech. Rep. No. 1, O.N.R., Jerusalem, 1960.

15. Rabin, M.O. Probabilistic algorithms. In *Algorithms and Complexity, New Directions and Recent Trends,* J.F. Traub Ed., Academic Press, New York, 1976, pp. 21–39.

16. Schönhage, A., and Strassen, V. Schnelle Multiplication grosser Zahlen. *Computing 7* (1971), 281–292.

17. Strassen, V. Gaussian elimination is not optimal. *Num. Math. 13* (1969), 354–356.

18. Valiant, L.G. General context-free recognition in less than cubic time. Rep., Dept. Comptr. Sci., Carnegie-Mellon U., Pittsburgh, Pa., 1974.

19. Winograd, S. On the time required to perform addition. *J. ACM 12* (1965), 277–285.

20. Winograd, S. On computing the discrete Fourier transform. *Proc. Natl. Acad. Sci. USA 73* (1976), 1005–1006.