1982
Turing
Award
Lecture

An Overview of Computational Complexity

STEPHEN A. COOK
University of Toronto

The 1982 Turing Award was presented to Stephen Arthur Cook, Professor of Computer Science at the University of Toronto, at the ACM Annual Conference in Dallas on October 25, 1982. The award is the Association's foremost recognition of technical contributions to the computing community.

The citation of Cook's achievements noted that "Dr. Cook has advanced our understanding of the complexity of computation in a significant and profound way. His seminal paper, The Complexity of Theorem Proving Procedures, presented at the 1971 ACM SIGACT Symposium on the Theory of Computing, laid the foundations for the theory of NP*-completeness. The ensuing exploration of the boundaries and nature of the* NP*-complete class of problems has been one of the most active and important research activities in computer science for the last decade.*

Cook is well known for his influential results in fundamental areas of computer science. He has made significant contributions to complexity theory, to time-space tradeoffs in computation, and to logics for programming languages. His work is characterized by elegance and insights and has illuminated the very nature of computation."

During 1970–1979, Cook did extensive work under grants from the National Research Council. He was also an E. W. R. Staecie Memorial

Author's present address: Department of Computer Science, University of Toronto, Toronto, Canada M5S 1A7.

Fellowship recipient for 1977–1978. The author of numerous landmark papers, he is currently involved in proving that no "good" algorithm exists for NP*-complete problems.*

The ACM Turing Award memorializes A. M. Turing, the English mathematician who made major contributions to the computing sciences.

An historical overview of computational complexity is presented. Emphasis is on the fundamental issues of defining the intrinsic computational complexity of a problem and proving upper and lower bounds on the complexity of problems. Probabilistic and parallel computation are discussed.

This is the second Turing Award lecture on Computational Complexity. The first was given by Michael Rabin in 1976.[1] In reading Rabin's excellent article [62] now, one of the things that strikes me is how much activity there has been in the field since. In this brief overview I want to mention what to me are the most important and interesting results since the subject began in about 1960. In such a large field the choice of topics is inevitably somewhat personal; however, I hope to include papers which, by any standards, are fundamental.

1 Early Papers

The prehistory of the subject goes back, appropriately, to Alan Turing. In his 1937 paper, *On computable numbers with an application to the Entscheidungsproblem* [85], Turing introduced his famous Turing machine, which provided the most convincing formalization (up to that time) of the notion of an effectively (or algorithmically) computable function. Once this notion was pinned down precisely, impossibility proofs for computers were possible. In the same paper Turing proved that no algorithm (i.e., Turing machine) could, upon being given an arbitrary formula of the predicate calculus, decide, in a finite number of steps, whether that formula was satisfiable.

After the theory explaining which problems can and cannot be solved by computer was well developed, it was natural to ask about the relative computational difficulty of computable functions. This is the subject matter of computational complexity. Rabin [59, 60] was one of the first persons (1960) to address this general question explicitly: what does it mean to say that f is more difficult to compute than g? Rabin suggested an axiomatic framework that provided the basis for the abstract complexity theory developed by Blum [6] and others.

A second early (1965) influential paper was *On the computational complexity of algorithms* by J. Hartmanis and R. E. Stearns [37].[2] This paper was widely read and gave the field its title. The important

[1]Michael Rabin and Dana Scott shared the Turing Award in 1976.

[2]See Hartmanis [36] for some interesting reminiscences.

notion of complexity measure defined by the computation time on multitape Turing machines was introduced, and hierarchy theorems were proved. The paper also posed an intriguing question that is still open today. Is any irrational algebraic number (such as $\sqrt{2}$) computable in real time, that is, is there a Turing machine that prints out the decimal expansion of the number at the rate of one digit per 100 steps forever?

A third founding paper (1965) was *The intrinsic computational difficulty of functions* by Alan Cobham [15]. Cobham emphasized the word "intrinsic," that is, he was interested in a machine-independent theory. He asked whether multiplication is harder than addition, and believed that the question could not be answered until the theory was properly developed. Cobham also defined and characterized the important class of functions he called $\mathscr{L}$: those functions on the natural numbers computable in time bounded by a polynomial in the decimal length of the input.

Three other papers that influenced the above authors as well as other complexity workers (including myself) are Yamada [91], Bennett [4], and Ritchie [66]. It is interesting to note that Rabin, Stearns, Bennett, and Ritchie were all students at Princeton at roughly the same time.

2
Early Issues and Concepts

Several of the early authors were concerned with the question: What is the right complexity measure? Most mentioned computation time or space as obvious choices, but were not convinced that these were the only or the right ones. For example, Cobham [15] suggested ". . . some measure related to the physical notion of work [may] lead to the most satisfactory analysis." Rabin [60] introduced axioms which a complexity measure should satisfy. With the perspective of 20 years experience, I now think it is clear that time and space—especially time—are certainly among the most important complexity measures. It seems that the first figure of merit given to evaluate the efficiency of an algorithm is its running time. However, more recently it is becoming clear that parallel time and hardware size are important complexity measures too (see Section 6).

Another important complexity measure that goes back in some form at least to Shannon [74] (1949) is Boolean circuit (or combinational) complexity. Here it is convenient to assume that the function f in question takes finite bit strings into finite bit strings, and the complexity $C(n)$ of f is the size of the smallest Boolean circuit that computes f for all inputs of length n. This very natural measure is closely related to computation time (see [57a], [57b], [68b]), and has a well-developed theory in its own right (see Savage [68a]).

Another question raised by Cobham [15] is what constitutes a "step" in a computation. This amounts to asking what is the right computer model for measuring the computation time of an algorithm. Multitape Turing machines are commonly used in the literature, but they have artificial restrictions from the point of view of efficient implementation of algorithms. For example, there is no compelling reason why the storage media should be linear tapes. Why not planar arrays of trees? Why not allow a random access memory?

In fact, quite a few computer models have been proposed since 1960. Since real computers have random access memories, it seems natural to allow these in the model. But just how to do this becomes a tricky question. If the machine can store integers in one step some bound must be placed on their size. (If the number 2 is squared 100 times the result has 2^{100} bits, which could not be stored in all the world's existing storage media.) I proposed charged RAM's in [19], in which a cost (number of steps) of about log $|x|$ is charged every time a number x is stored or retrieved. This works but is not completely convincing. A more popular random access model is the one used by Aho, Hopcroft, and Ullman in [3], in which each operation involving an integer has unit cost, but integers are not allowed to become unreasonably large (for example, their magnitude might be bounded by some fixed polynomial in the size of the input). Probably the most mathematically satisfying model is Schönhage's storage modification machine [69], which can be viewed either as a Turing machine that builds its own storage structure or as a unit cost RAM that can only copy, add or subtract one, or store or retrieve in one step. Schönhage's machine is a slight generalization of the Kolmogorov–Uspenski machine proposed much earlier [46] (1958), and seems to me to represent the most general machine that could possibly be construed as doing a bounded amount of work in one step. The trouble is that it probably is a little too powerful. (See Section 3 under "large number multiplication.")

Returning to Cobham's question "what is a step," I think what has become clear in the last 20 years is that there is no single clear answer. Fortunately, the competing computer models are not wildly different in computation time. In general, each can simulate any other by at most squaring the computation time (some of the first arguments to this effect are in [37]). Among the leading random access models, there is only a factor of log computation time in question.

This leads to the final important concept developed by 1965—the identification of the class of problems solvable in time bounded by a polynomial in the length of the input. The distinction between polynomial time and exponential time algorithms was made as early as 1953 by von Neumann [90]. However, the class was not defined formally and studied until Cobham [15] introduced the class $\mathcal{L}$ of functions in 1964 (see Section 1). Cobham pointed out that the class was well defined, independent of which computer model was chosen, and gave it a characterization in the spirit of recursive function theory. The idea that polynomial time computability roughly corresponds to tractability was first expressed in print by Edmonds [27], who called polynomial time algorithms "good algorithms." The now standard notation P for the class of polynomial time recognizable sets of strings was introduced later by Karp [42].

The identification of P with the tractable (or feasible) problems has been generally accepted in the field since the early 1970's. It is not immediately obvious why this should be true, since an algorithm whose running time is the polynomial n^{1000} is surely not feasible, and conversely, one whose running time is the exponential $2^{0.0001n}$ is feasible in practice. It seems to be an *empirical* fact, however, that naturally arising problems do not have optimal algorithms with such running times.[3] The most notable practical algorithm that has an exponential worst case running time is the simplex algorithm for linear programming. Smale [75, 76] attempts to explain this by showing that, in some sense, the average running time is fast, but it is also important to note that Khachian [43] showed that linear programming is in P using another algorithm. Thus, our general thesis, that P equals the feasible problems, is not violated.

3 Upper Bounds on Time

A good part of computer science research consists of designing and analyzing enormous numbers of efficient algorithms. The important algorithms (from the point of view of computational complexity) must be special in some way; they generally supply a surprisingly fast way of solving a simple or important problem. Below I list some of the more interesting ones invented since 1960. (As an aside, it is interesting to speculate on what are the all time most important algorithms. Surely the arithmetic operations $+$, $-$, $*$, and $\div$ on decimal numbers are basic. After that, I suggest fast sorting and searching, Gaussian elimination, the Euclidean algorithm, and the simplex algorithm as candidates.)

The parameter n refers to the size of the input, and the time bounds are the worst case time bounds and apply to a multitape Turing machine (or any reasonable random access machine) except where noted.

(1) **The fast Fourier transform** [23], requiring $O(n \log n)$ arithmetic operations, is one of the most used algorithms in scientific computing.

(2) **Large number multiplication.** The elementary school method requires $O(n^2)$ bit operations to multiply two n digit numbers. In 1962 Karatsuba and Ofman [41] published a method requiring only $O(n^{1.59})$ steps. Shortly after that Toom [84] showed how to construct Boolean circuits of size $O(n^{1+\epsilon})$ for arbitrarily small $\epsilon > 0$ in order to carry out

[3]See [31], pp. 6–9 for a discussion of this.

the multiplication. I was a graduate student at Harvard at the time, and inspired by Cobham's question "Is multiplication harder than addition?" I was naively trying to prove that multiplication requires $\Omega(n^2)$ steps on a multitape Turing machine. Toom's paper caused me considerable surprise. With the help of Stal Aanderaa [22], I was reduced to showing that multiplication requires $\Omega(n \log n/(\log\log n)^2)$ steps using an "on-line" Turing machine.[4] I also pointed out in my thesis that Toom's method can be adapted to multitape Turing machines in order to multiply in $O(n^{1+\epsilon})$ steps, something that I am sure came as no surprise to Toom.

The currently fastest asymptotic running time on a multitape Turing machine for number multiplication is $O(n \log n \log\log n)$, and was devised by Schönhage and Strassen [70] (1971) using the fast Fourier transform. However, Schönhage [69] recently showed by a complicated argument that his storage modification machines (see Section 2) can multiply in time $O(n)$ (linear time!). We are forced to conclude that either multiplication is easier than we thought or that Schönhage's machines cheat.

(3) **Matrix multiplication.** The obvious method requires $n^2(2n-1)$ arithmetic operations to multiply two $n \times n$ matrices, and attempts were made to prove the method optimal in the 1950's and 1960's. There was surprise when Strassen [81] (1969) published his method requiring only $4.7n^{2.81}$ operations. Considerable work has been devoted to reducing the exponent of 2.81, and currently the best time known is $O(n^{2.496})$ operations, due to Coppersmith and Winograd [24]. There is still plenty of room for progress, since the best known lower bound is $2n^2-1$ (see [13]).

(4) **Maximum matchings in general undirected graphs.** This was perhaps the first problem explicitly shown to be in P whose membership in P requires a difficult algorithm. Edmonds' influential paper [27] gave the result and discussed the notion of a polynomial time algorithm (see Section 2). He also pointed out that the simple notion of augmenting path, which suffices for the bipartite case, does not work for general undirected graphs.

(5) **Recognition of prime numbers.** The major question here is whether this problem is in P. In other words, is there an algorithm that always tells us whether an arbitrary n-digit input integer is prime, and halts in a number of steps bounded by a fixed polynomial in n? Gary Miller [53] (1976) showed that there is such an algorithm, but its validity depends on the extended Riemann hypothesis. Solovay and Strassen [77] devised a fast Monte Carlo algorithm (see Section 5) for prime recognition, but if the input number is composite there is a small chance the algorithm will mistakenly say it is prime. The best provable deterministic algorithm known is due to Adleman, Pomerance, and Rumley [2] and runs in time $n^{O(\log\log n)}$, which is slightly worse than

[4]This lower bound has been slightly improved. See [56] and [64].

polynomial. A variation of this due to H. Cohen and H. W. Lenstra Jr. [17] can routinely handle numbers up to 100 decimal digits in approximately 45 seconds.

Recently three important problems have been shown to be in the class *P*. The first is linear programming, shown by Khachian [43] in 1979 (see [55] for an exposition). The second is determining whether two graphs of degree at most *d* are isomorphic, shown by Luks [50] in 1980. (The algorithm is polynomial in the number of vertices for fixed *d*, but exponential in *d*.) The third is factoring polynomials with rational coefficients. This was shown for polynomials in one variable by Lenstra, Lenstra, and Lovasz [48] in 1982. It can be generalized to polynomials in any fixed number of variables as shown by Kaltofen's result [39], [40].

4 Lower Bounds

The real challenge in complexity theory, and the problem that sets the theory apart from the analysis of algorithms, is proving lower bounds on the complexity of specific problems. There is something very satisfying in proving that a yes-no problem cannot be solved in n, or n^2, or 2^n steps, no matter what algorithm is used. There have been some important successes in proving lower bounds, but the open questions are even more important and somewhat frustrating.

All important lower bounds on computation time or space are based on "diagonal arguments." Diagonal arguments were used by Turing and his contemporaries to prove certain problems are not algorithmically solvable. They were also used prior to 1960 to define hierarchies of computable 0-1 functions.[5] In 1960, Rabin [60] proved that for any reasonable complexity measure, such as computation time or space (memory), sufficiently increasing the allowed time or space etc. always allows more 0-1 functions to be computed. About the same time, Ritchie in his thesis [65] defined a specific hierarchy of functions (which he showed is nontrivial for 0-1 functions) in terms of the amount of space allowed. A little later Rabin's result was amplified in detail for time on multitape Turing machines by Hartmanis and Stearns [37], and for space by Stearns, Hartmanis, and Lewis [78].

4.1 *Natural Decidable Problems Proved Infeasible*

The hierarchy results mentioned above gave lower bounds on the time and space needed to compute specific functions, but all such functions seemed to be "contrived." For example, it is easy to see that the function $f(x,y)$ which gives the first digit of the output of machine x on input y after $(|x| + |y|)^2$ steps cannot be computed in time $(|x| + |y|)^2$. It was not until 1972, when Albert Meyer and Larry

[5]See, for example, Grzegorczyk [35].

Stockmeyer [52] proved that the equivalence problem for regular expressions with squaring requires exponential space and, therefore, exponential time, that a nontrivial lower bound for general models of computation on a "natural" problem was found (natural in the sense of being interesting, and not about computing machines). Shortly after that Meyer [51] found a very strong lower bound on the time required to determine the truth of formulas in a certain formal decidable theory called WSIS (weak monadic second-order theory of successor). He proved that any computer whose running time was bounded by a fixed number of exponentials (2^n, 2^{2^n}, $2^{2^{2^n}}$, etc.) could not correctly decide WSIS. Meyer's Ph.D. student, Stockmeyer, went on to calculate [79] that any Boolean circuit (think computer) that correctly decides the truth of an arbitrary WSIS formula of length 616 symbols must have more than 10^{123} gates. The number 10^{123} was chosen to be the number of protons that could fit in the known universe. This is a very convincing infeasibility proof!

Since Meyer and Stockmeyer there have been a large number of lower bounds on the complexity of decidable formal theories (see [29] and [80] for summaries). One of the most interesting is a doubly exponential time lower bound on the time required to decide Presburger arithmetic (the theory of the natural numbers under addition) by Fischer and Rabin [30]. This is not far from the best known time upper bound for this theory, which is triply exponential [54]. The best space upper bound is doubly exponential [29].

Despite the above successes, the record for proving lower bounds on problems of smaller complexity is appalling. In fact, there is no nonlinear time lower bound known on a general-purpose computation model for any natural problem in *NP* (see Section 4.4), in particular, for any of the 300 problems listed in [31]. Of course, one can prove by diagonal arguments the existence of problems in *NP* requiring time n^k for any fixed k. In the case of space lower bounds, however, we do not even know how to prove the existence of *NP* problems not solvable in space $O(\log n)$ on an off-line Turing machine (see Section 4.3). This is despite the fact that the best known space upper bounds in many natural cases are essentially linear in n.

4.2 *Structured Lower Bounds*

Although we have had little success in proving interesting lower bounds for concrete problems on general computer models, we do have interesting results for "structured" models. The term "structured" was introduced by Borodin [9] to refer to computers restricted to certain operations appropriate to the problem at hand. A simple example of this is the problem of sorting n numbers. One can prove (see [44]) without much difficulty that this requires at least $n \log n$ comparisons, provided that the only operation the computer is allowed to do with the inputs is to compare them in pairs. This lower bound says nothing about Turing machines or Boolean circuits, but it has been extended to unit cost random access machines, provided division is disallowed.

A second and very elegant structured lower bound, due to Strassen [82] (1973), states that polynomial interpolation, that is, finding the coefficients of the polynomial of degree $n-1$ that passes through n given points, requires $\Omega(n\log n)$ multiplications, provided only arithmetic operations are allowed. Part of the interest here is that Strassen's original proof depends on Bezout's theorem, a deep result in algebraic geometry. Very recently, Baur and Strassen [83] have extended the lower bound to show that even the middle coefficient of the interpolating polynomial through n points requires $\Omega(n\log n)$ multiplications to compute.

Part of the appeal of all of these structured results is that the lower bounds are close to the best known upper bounds,[6] and the best known algorithms can be implemented on the structured models to which the lower bounds apply. (Note that radix sort, which is sometimes said to be linear time, really requires at least $n\log n$ steps, if one assumes the input numbers have enough digits so that they all can be distinct.)

4.3
Time–Space Product Lower Bounds

Another way around the impasse of proving time and space lower bounds is to prove time lower bounds under the assumption of small space. Cobham [16] proved the first such result in 1966, when he showed that the time–space product for recognizing n-digit perfect squares on an "off-line" Turing machine must be $\Omega(n^2)$. (The same is true of n-symbol palindromes.) Here the input is written on a two-way read-only input tape, and the space used is by definition the number of squares scanned by the work tapes available to the Turing machine. Thus, if, for example, the space is restricted to $O(\log^3 n)$ (which is more than sufficient), then the time must be $\Omega(n^2/\log^3 n)$ steps.

The weakness in Cobham's result is that although the off-line Turing machine is a reasonable one for measuring computation time and space separately, it is too restrictive when time and space are considered together. For example, the palindromes can obviously be recognized in $2n$ steps and constant space if two heads are allowed to scan the input tape simultaneously. Borodin and I [10] partially rectified the weakness when we proved that sorting n integers in the range one to n^2 requires a time–space product of $\Omega(n^2/\log n)$. The proof applies to any "general sequential machine," which includes off-line Turing machines with many input heads, or even random access to the input tape. It is unfortunately crucial to our proof that sorting requires many output bits, and it remains an interesting open question whether a similar lower bound can be made to apply to a set recognition problem,

[6]See Borodin and Munro [12] for upper bounds for interpolation.

such as recognizing whether all n input numbers are distinct. (Our lower bound on sorting has recently been slightly improved in [64].)

4.4
NP-*Completeness*

The theory of *NP*-completeness is surely the most significant development in computational complexity. I will not dwell on it here because it is now well known and is the subject of textbooks. In particular, the book by Garey and Johnson [31] is an excellent place to read about it.

The class *NP* consists of all sets recognizable in polynomial time by a nondeterministic Turing machine. As far as I know, the first time a mathematically equivalent class was defined was by James Bennett in his 1962 Ph.D. thesis [4]. Bennett used the name "extended positive rudimentary relations" for his class, and his definition used logical quantifiers instead of computing machines. I read this part of his thesis and realized his class could be characterized as the now familiar definition of *NP*. I used the term $\mathscr{L}^+$ (after Cobham's class $\mathscr{L}$) in my 1971 paper [18], and Karp gave the now accepted name *NP* to the class in his 1972 paper [42]. Meanwhile, quite independent of the formal development, Edmonds, back in 1965 [28], talked informally about problems with a "good characterization," a notion essentially equivalent to *NP*.

In 1971 [18], I introduced the notion of *NP*-complete and proved 3-satisfiably and the subgraph problem were *NP*-complete. A year later, Karp [42] proved 21 problems were *NP*-complete, thus forcefully demonstrating the importance of the subject. Independently of this and slightly later, Leonid Levin [49], in the Soviet Union (now at Boston University), defined a similar (and stronger) notion and proved six problems were complete in his sense. The informal notion of "search problem" was standard in the Soviet literature, and Levin called his problems "universal search problems."

The class *NP* includes an enormous number of practical problems that occur in business and industry (see [31]). A proof that an *NP* problem is *NP*-complete is a proof that the problem is not in *P* (does not have a deterministic polynomial time algorithm) unless every *NP* problem is in *P*. Since the latter condition would revolutionize computer science, the practical effect of *NP*-completeness is a lower bound. This is why I have included this subject in the section on lower bounds.

4.5
#P-*Completeness*

The notion of *NP*-completeness applies to sets, and a proof that a set is *NP*-complete is usually interpreted as a proof that it is intractable. There are, however, a large number of apparently intractable *functions* for which no *NP*-completeness proof seems to be relevant. Leslie Valiant [86,87] defined the notion of *#P*-completeness to help remedy this situation. Proving that a function is *#P*-complete shows that it is apparently intractable to compute in the same way that proving a set is *NP*-complete shows that it is apparently intractable to recognize; namely, if a *#P*-complete function is computable in polynomial time, then $P = NP$.

Valiant gave many examples of *#P*-complete functions, but probably the most interesting one is the permanent of an integer matrix. The permanent has a definition formally similar to the determinant, but whereas the determinant is easy to compute by Gaussian elimination, the many attempts over the past hundred odd years to find a feasible way to compute the permanent have all failed. Valiant gave the first convincing reason for this failure when he proved the permanent *#P*-complete.

5
Probabilistic Algorithms

The use of random numbers to simulate or approximate random processes is very natural and is well established in computing practice. However, the idea that random inputs might be very useful in solving deterministic combinatorial problems has been much slower in penetrating the computer science community. Here I will restrict attention to probabilistic (coin tossing) polynomial time algorithms that "solve" (in a reasonable sense) a problem for which no deterministic polynomial time algorithm is known.

The first such algorithm seems to be the one by Berlekamp [5] in 1970, for factoring a polynomial f over the field $GF(p)$ of p elements. Berlekamp's algorithm runs in time polynomial in the degree of f and $\log p$, and with probability at least one-half it finds a correct prime factorization of f; otherwise it ends in failure. Since the algorithm can be repeated any number of times and the failure events are all independent, the algorithm in practice always factors in a feasible amount of time.

A more drastic example is the algorithm for prime recognition due to Solovay and Strassen [77] (submitted in 1974). This algorithm runs in time polynomial in the length of the input m, and outputs either "prime" or "composite." If m is in fact prime, then the output is certainly "prime," but if m is composite, then with probability at most one-half the answer may also be "prime." The algorithm may be repeated any number of times on an input m with independent results. Thus if the answer is ever "composite," the user knows m is composite; if the answer is consistently "prime" after, say, 100 runs, then the user has good evidence that m is prime, since any fixed composite m would give such results with tiny probability (less than 2^{-100}).

Rabin [61] developed a different probabilistic algorithm with properties similar to the one above, and found it to be very fast on computer trials. The number $2^{400} - 593$ was identified as (probably) prime within a few minutes.

One interesting application of probabilistic prime testers was proposed by Rivest, Shamir, and Adleman [67a] in their landmark paper on public key cryptosystems in 1978. Their system requires the generation of large (100 digit) random primes. They proposed testing random 100 digit numbers using the Solovay–Strassen method until one was found that was probably prime in the sense outlined above. Actually with the new high-powered deterministic prime tester of Cohen and Lenstra [17] mentioned in Section 3, once a random 100 digit "probably prime" number was found it could be tested for certain in about 45 seconds, if it is important to know for certain.

The class of sets with polynomial time probabilistic recognition algorithms in the sense of Solovay and Strassen is known as R (or sometimes RP) in the literature. Thus a set is in R if and only if it has a probabilistic recognition algorithm that always halts in polynomial time and never makes a mistake for inputs not in R, and for each input in R it outputs the right answer for each run with probability at least one-half. Hence the set of composite numbers is in R, and in general $P \subseteq R \subseteq NP$. There are other interesting examples of sets in R not known to be in P. For example, Schwartz [71] shows that the set of nonsingular matrices whose entries are polynomials in many variables is in R. The algorithm evaluates the polynomials at random small integer values and computes the determinant of the result. (The determinant apparently cannot feasibly be computed directly because the polynomials computed would have exponentially many terms in general.)

It is an intriguing, open question whether $R = P$. It is tempting to conjecture yes on the philosophical grounds that random coin tosses should not be of much use when the answer being sought is a well-defined yes or no. A related question is whether a probabilistic algorithm (showing a problem is in R) is for all practical purposes as good as a deterministic algorithm. After all, the probabilistic algorithms can be run using the pseudorandom number generations available on most computers, and an error probability of 2^{-100} is negligible. The catch is that pseudorandom number generators do not produce truly random numbers, and nobody knows how well they will work for a given probalistic algorithm. In fact, experience shows they seem to work well. But if they *always* work well, then it follows that $R = P$, because pseudorandom numbers are generated deterministically so true randomness would not help after all. Another possibility is to use a physical process such as thermal noise to generate random numbers. But it is an open question in the philosophy of science how truly random nature can be.

Let me close this section by mentioning an interesting theorem of Adlemen [1] on the class R. It is easy to see [57b] that if a set is in P, then for each n there is a Boolean circuit of size bounded by a fixed polynomial in n which determines whether an arbitrary string of length

n is in the set. What Adleman proved is that the same is true for the class R. Thus, for example, for each n there is a small "computer circuit" that correctly and rapidly tests whether n digit numbers are prime. The catch is that the circuits are not uniform in n, and in fact for the case of 100 digits it may not be feasible to figure out how to build the circuit.[7]

6 Synchronous Parallel Computation

With the advent of VLSI technology in which one or more processors can be placed on a quarter-inch chip, it is natural to think of a future composed of many thousands of such processors working together in parallel to solve a single problem. Although no very large general-purpose machine of this kind has been built yet, there are such projects under way (see Schwartz [72]). This motivates the recent development of a very pleasing branch of computation complexity: the theory of large-scale synchronous parallel computation, in which the number of processors is a resource bounded by a parameter $H(n)$ (H is for *hardware*) in the same way that space is bounded by a parameter $S(n)$ in sequential complexity theory. Typically $H(n)$ is a fixed polynomial in n.

Quite a number of parallel computation models have been proposed (see [21] for a review), just as there are many competing sequential models (see Section 2). There are two main contenders, however. The first is the class of shared memory models in which a large number of processors communicate via a random access memory that they hold in common. Many parallel algorithms have been published for such models, since real parallel machines may well be like this when they are built. However, for the mathematical theory these models are not very satisfactory because too much of their detailed specification is arbitrary: How are read and write conflicts in the common memory resolved? What basic operations are allowed for each processor? Should one charge log $H(n)$ time units to access common memory?

Hence I prefer the cleaner model discussed by Borodin [8] (1977), in which a parallel computer is a uniform family $\langle B_n \rangle$ of acyclic Boolean circuits, such that B_n has n inputs (and hence takes care of those input strings of length n). Then $H(n)$ (the amount of hardware) is simply the number of gates in B_n, and $T(n)$ (the parallel computation time) is the depth of the circuit B_n (i.e., length of the longest path from an input to an output). This model has the practical justification that presumably all real machines (including shared memory machines) are built from Boolean circuits. Furthermore, the minimum Boolean size and depth needed to compute a function is a natural mathematical problem and was considered well before the theory of parallel computation was around.

[7]For more theory on probabilistic computation, see Gill [32].

Fortunately for the theory, the minimum values of hardware $H(n)$ and parallel time $T(n)$ are not widely different for the various competing parallel computer models. In particular, there is an interesting general fact true for all the models, first proved for a particular model by Pratt and Stockmeyer [58] in 1974 and called the "parallel computation thesis" in [33]; namely, a problem can be solved in time polynomial in $T(n)$ by a parallel machine (with unlimited hardware) if and only if it can be solved in space polynomial in $T(n)$ by a sequential machine (with unlimited time).

A basic question in parallel computation is: Which problems can be solved substantially faster using many processors rather than one processor? Nicholas Pippenger [57a] formalized this question by defining the class (now called *NC*, for "Nick's class") of problems solvable ultra fast [time $T(n) = (\log n)^{0(1)}$] on a parallel computer with a feasible [$H(n) = n^{0(1)}$] amount of hardware. Fortunately, the class *NC* remains the same, independent of the particular parallel computer model chosen, and it is easy to see that *NC* is a subset of the class *FP* of functions computable sequentially in polynomial time. Our informal question can then be formalized as follows: Which problems in *FP* are also in *FC*?

It is conceivable (though unlikely) that $NC = FP$, since to prove $NC \neq FP$ would require a breakthrough in complexity theory (see the end of Section 4.1). Since we do not know how to prove a function f in *FP* is not in *NC*, the next best thing is to prove that f is log space-complete for *FP*. This is the analog of proving a problem is *NP*-complete, and has the practical effect of discouraging efforts for finding super fast parallel algorithms for f. This is because if f is log space-complete for *FP* and f is in *NC*, then $FP = NC$, which would be a big surprise.

Quite a bit of progress has been made in classifying problems in *FP* as to whether they are in *NC* or log space-complete for *FP* (of course, they may be neither). The first example of a problem complete for *P* was presented in 1973 by me in [20], although I did not state the result as a completeness result. Shortly after that Jones and Laaser [38] defined this notion of completeness and gave about five examples, including the emptiness problem for context-free grammers. Probably the simplest problem proved complete for *FP* is the so-called circuit value problem [47]: given a Boolean circuit together with values for its inputs, find the value of the output. The example most interesting to me, due to Goldschlager, Shaw, and Staples [34], is finding the (parity of) maximum flow through a given network with (large) positive integer capacities on its edges. The interest comes from the subtlety of the completeness proof. Finally, I should mention that linear programming is complete for *FP*. In this case the difficult part is showing that the problem is in *P* (see [43]), after which the completeness proof [26] is straightforward.

Among the problems known to be in *NC* are the four arithmetic operations ($+$, $-$, $*$, $\div$) on binary numbers, sorting, graph connectivity, matrix operations (multiplication, inverse, determinant, rank), polynomial greatest common divisors, context-free languages, and finding a minimum spanning forest in a graph (see [11], [21], [63], [67b]). The size of a maximum matching for a given graph is known [11] to be in "random" *NC* (*NC* in which coin tosses are allowed), although it is an interesting open question of whether finding an actual maximum matching is even in random *NC*. Results in [89] and [67b] provide general methods for showing problems are in *NC*.

The most interesting problem in *FP* not known either to be complete for *FP* or in (random) *NC* is finding the greatest common divisor of two integers. There are many other interesting problems that have yet to be classified, including finding a maximum matching or a maximal clique in a graph (see [88]).

7 The Future

Let me say again that the field of computational complexity is large and this overview is brief. There are large parts of the subject that I have left out altogether or barely touched on. My apologies to the researchers in those parts.

One relatively new and exciting part, called "computational information theory," by Yao [92], builds on Shannon's classical information theory by considering information that can be accessed through a feasible computation. This subject was sparked largely by the papers by Diffie and Hellman [25] and Rivest, Shamir, and Adleman [67a] on public key cryptosystems, although its computational roots go back to Kolmogorov [45] and Chaitin [14a], [14b], who first gave meaning to the notion of a single finite sequence being "random," by using the theory of computation. An interesting idea in this theory, considered by Shamir [73] and Blum and Micali [7], concerns generating pseudorandom sequences in which future bits are provably hard to predict in terms of past bits. Yao [92] proves that the existence of such sequences would have positive implications about the deterministic complexity of the probabilistic class R (see Section 5). In fact, computational information theory promises to shed light on the role of randomness in computation.

In addition to computational information theory we can expect interesting new results on probabilistic algorithms, parallel computation, and (with any luck) lower bounds. Concerning lower bounds, the one breakthrough for which I see some hope in the near future is showing that not every problem in P is solvable in space $O(\log n)$, and perhaps also $P \neq NC$. In any case, the field of computational complexity remains very vigorous, and I look forward to seeing what the future will bring.

Acknowledgments

I am grateful to my complexity colleagues at Toronto for many helpful comments and suggestions, especially Allan Borodin, Joachim von zur Gathen, Silvio Micali, and Charles Rackoff.

References

1. Adleman, L. Two theorems on random polynomial time. *Proc. 19th IEEE Symp. on Foundations of Computer Science. IEEE Computer Society, Los Angeles* (1978), 75-83.
2. Adleman, L., Pomerance, C., and Rumley, R. S. On distinguishing prime numbers from composite numbers. *Annals of Math 117* (January 1983), 173-206.
3. Aho, A. V., Hopcroft, J. E., and Ullman, J. D. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading, Mass., 1974.
4. Bennett, J. H. *On Spectra.* Doctoral dissertation, Department of Mathematics, Princeton University, 1962.
5. Berlekamp, E. R. Factoring polynomials over large finite fields. *Math. Comp. 24* (1970), 713-735.
6. Blum, M. A machine independent theory of the complexity of recursive functions. *JACM 14,* 2 (April 1967), 322-336.
7. Blum, M., and Micali, S. How to generate cryptographically strong sequences of pseudo random bits. *Proc. 23rd IEEE Symp. on Foundations of Computer Science. IEEE Computer Society, Los Angeles* (1982), 112-117.
8. Borodin, A. On relating time and space to size and depth. *SIAM J. Comp. 6* (1977), 733-744.
9. Borodin, A. Structured vs. general models in computational complexity. In *Logic and Algorithmic,* Monographie no. 30 de L'Enseignement Mathematique Université de Genève, 1982.
10. Borodin, A., and Cook, S. A time-space tradeoff for sorting on a general sequential model of computation. *SIAM J. Comput. 11* (1982), 287-297.
11. Borodin, A., von zur Gathen, J., and Hopcroft, J. Fast parallel matrix and GCD computations. *23rd IEEE Symp. on Foundations of Computer Science. IEEE Computer Society, Los Angeles* (1982), 65-71.
12. Borodin, A., and Munro, I. *The Computational Complexity of Algebraic and Numeric Problems.* Elsevier, New York, 1975.
13. Brockett, R. W., and Dobkin, D. On the optimal evaluation of a set of bilinear forms. *Linear Algebra and Its Applications 19* (1978), 207-235.

14a. Chaitin, G. J. On the length of programs for computing finite binary sequences. *JACM 13,* 4 (October 1966), 547-569; *JACM 16,* 1 (January 1969), 145-159.

14b. Chaitin, G. J. A theory of program size formally identical to informational theory. *JACM 22,* 3 (July 1975), 329-340.

15. Cobham, A. The intrinsic computational difficulty of functions. *Proc. 1964 International Congress for Logic, Methodology, and Philosophy of Sciences.* Y. Bar-Hellel, Ed., North Holland, Amsterdam, 1965, 24-30.
16. Cobham, A. The recognition problem for the set of perfect squares. *IEEE Conference Record Seventh SWAT* (1966), 78-87.

17. Cohen, H., and Lenstra, H. W., Jr. Primarily testing and Jacobi sums. Report 82-18, University of Amsterdam, Dept. of Math., 1982.
18. Cook, S. A. The complexity of theorem proving procedures. *Proc. 3rd ACM Symp. on Theory of Computing.* Shaker Heights, Ohio (May 3–5, 1971), 151–158.
19. Cook, S. A. Linear time simulation of deterministic two-way pushdown automata. *Proc. IFIP Congress 71 (Theoretical Foundations).* North Holland, Amsterdam, 1972, 75–80.
20. Cook, S. A. An observation on time-storage tradeoff. *JCSS 9* (1974), 308–316. Originally in *Proc. 5th ACM Symp. on Theory of Computing,* Austin, TX (April 30–May 2 1973), 29–33.
21. Cook, S. A. Towards a complexity theory of synchronous parallel computation. L'Enseignement Mathematique XXVII (1981), 99–124.
22. Cook, S. A., and Aanderaa, S. O. On the minimum computation time of functions. *Trans. AMS 142* (1969), 291–314.
23. Cooley, J. M., and Tukey, J. W. An algorithm for the machine calculation of complex Fourier series. *Math. Comput. 19* (1965), 297–301.
24. Coppersmith, D., and Winograd, S. On the asymptomatic complexity of matrix multiplication. *SIAM J. Comp. 11* (1982), 472–492.
25. Diffie, W., and Hellman, M. E. New direction in cryptography. *IEEE Trans. on Inform. Theory IT-22,* 6 (1976), 644–654.
26. Dobkin, D., Lipton, R. J., and Reiss, S. Linear programming is log-space hard for P. *Inf. Processing Letters 8* (1979), 96–97.
27. Edmonds, J. Paths, trees, flowers. *Canad. J. Math. 17* (1965), 449–67.
28. Edmonds, J. Minimum partition of a matroid into independent subsets. *J. Res. Nat. Bur. Standards Sect. B,* 69 (1965), 67–72.
29. Ferrante, J., and Rackoff, C. W. The Computational Complexity of Logical Theories. *Lecture Notes in Mathematics.* #718, Springer Verlag, New York, 1979.
30. Fischer, M. J., and Rabin, M. O. Super-exponential complexity of Presburger arithmetic. In *Complexity of Computation. SIAM-AMS Proc. 7,* R. Karp, Ed., 1974, 27–42.
31. Garey, M. R., and Johnson, D. S. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman, San Francisco, 1979.
32. Gill, J. Computational complexity of probabilistic Turing machines. *SIAM J. Comput. 6* (1977), 675–695.
33. Goldschlager, L. M. *Synchronous Parallel Computation.* Doctoral dissertation, Dept. of Computer Science, Univ. of Toronto, 1977. See also *JACM 29,* 4 (October 1982), 1073–1086.
34. Goldschlager, L. M., Shaw, R. A., and Staples, J. The maximum flow problem is log space complete for P. *Theoretical Computer Science 21* (1982), 105–111.
35. Grzegorczyk, A. Some classes of recursive functions. *Rozprawy Matemtyczne,* 1953.
36. Hartmanis, J. Observations about the development of theoretical computer science. *Annals Hist. Comput. 3,* 1 (Jan. 1981), 42–51.
37. Hartmanis, J., and Stearns, R. E. On the computational complexity of algorithms. *Trans. AMS 117* (1965), 285–306.
38. Jones, N. D., and Laaser, W. T. Complete problems for deterministic polynomial time. *Theoretical Computer Science 3* (1977), 105–427.

39. Kaltofen, E. A polynomial reduction from multivariate to bivariate integer polynomial factorization. *Proc. 14th ACM Symp. in Theory Comp.*, San Francisco, CA (May 5-7 1982), 261-266.
40. Kaltofen, E. A polynomial-time reduction from bivariate to univariate integral polynomial factorization. *Proc. 23rd IEEE Symp. on Foundations of Computer Science. IEEE Computer Society, Los Angeles* (1982), 57-64.
41. Karatsuba, A., and Ofman, Yu. Multiplication of multidigit numbers on automata. *Doklady Akad. Nauk 145,* 2 (1962), 293-294. Translated in *Soviet Phys. Doklady 77* (1963), 595-596.
42. Karp, R. M. Reducibility among combinatorial problems. In: *Complexity of Computer Computations.* R. E. Miller and J. W. Thatcher, Eds., Plenum Press, New York, 1972, 85-104.
43. Khachian, L. G. A polynomial time algorithm for linear programming. *Doklady Akad. Nauk SSSR. 244,* 5 (1979), 1093-96. Translated in *Soviet Math. Doklady 20,* 191-194.
44. Knuth, D. E. *The Art of Computer Programming, vol. 3. Sorting and Searching.* Addison-Wesley, Reading, MA, 1973.
45. Kolmogorov, A. N. Three approaches to the concept of the amount of information. *Probl. Pered. Inf. (Probl. of Inf. Transm.) 1* (1965).
46. Kolmogorov, A. N., and Uspenski, V. A. On the definition of an algorithm, *Uspehi Mat. Nauk. 13* (1958), 3-28: AMS Transl. 2nd ser. 29 (1963), 217-245.
47. Ladner, R. E. The circuit value problem is log space complete for P. *SIGACT News 7,* 1 (1975), 18-20.
48. Lenstra, A. K., Lenstra, H. W., and Lovasz, L. Factoring polynomials with rational coefficients. Report 82-05, University of Amsterdam, Dept. of Math., 1982.
49. Levin, L. A. Universal search problems. *Problemy Peredaci Informacii 9* (1973), 115-116. Translated in *Problems of Information Transmission 9,* 265-266.
50. Luks, E. M. Isomorphism of graphs of bounded valence can be tested in polynomial time. *Proc. 21st IEEE Symp. on Foundations of Computer Science. IEEE Computer Society, Los Angeles* (1980), 42-49.
51. Meyer, A. R. Weak monadic second-order theory of successor is not elementary-recursive. *Lecture Notes in Mathematics 453.* Springer Verlag, New York, 1975, 132-154.
52. Meyer, A. R., and Stockmeyer, L. J. The equivalence problem for regular expressions with squaring requires exponential space. *Proc. 13th IEEE Symp. on Switching and Automata Theory* (1972), 125-129.
53. Miller, G. L. Riemann's hypothesis and tests for primality. *J. Comput. System Sci. 13* (1976), 300-317.
54. Oppen, D. C. A $2^{2^{2pn}}$ upper bound on the complexity of Presburger arithmetic. *J. Comput. Syst. Sci. 16* (1978), 323-332.
55. Papadimitriou, C. H., and Steiglitz, K. *Combinatorial Optimization: Algorithms and Complexity.* Prentice-Hall, Englewood Cliffs, NJ, 1982.
56. Paterson, M. S., Fischer, M. J., and Meyer, A. R. An improved overlap argument for on-line multiplication. *SIAM-AMS Proc. 7,* Amer. Math. Soc., Providence, 1974, 97-111.
57a. Pippenger, N. On simultaneous resource bounds (preliminary version). *Proc. 20th IEEE Symp. on Foundations of Computer Science.* IEEE Computer Society, Los Angeles (1979), 307-311.

57b. Pippenger, N. J., and Fischer, M. J. Relations among complexity measures. *JACM 26,* 2 (April 1979), 361-381.
58. Pratt, V. R., and Stockmeyer, L. J. A characterization of the power of vector machines. *J. Comput. System Sci. 12* (1976), 198-221. Originally in *Proc. 6th ACM Symp. on Theory of Computing,* Seattle, WA (April 30-May 2, 1974), 122-134.
59. Rabin, M. O. Speed of computation and classification of recursive sets. *Third Convention Sci. Soc.,* Israel, 1959, 1-2.
60. 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.
61. Rabin, M. O. Probabilistic algorithms. In *Algorithms and Complexity, New Directions and Recent Trends,* J. F. Traub, Ed., Academic Press, New York, 1976, 29-39.
62. Rabin, M. O. Complexity of computations. *Comm. ACM 20,* 9 (September 1977), 625-633.
63. Reif, J. H. Symmetric complementation. *Proc. 14th ACM Symp. on Theory of Computing,* San Francisco, CA (May 5-7, 1982), 201-214.
64. Reisch, S., and Schnitger, G. Three applications of Kolmorgorov complexity. *Proc. 23rd IEEE Symp. on Foundations of Computer Science.* IEEE Computer Society, Los Angeles (1982), 45-52.
65. Ritchie, R. W. *Classes of Recursive Functions of Predictable Complexity.* Doctoral Dissertation, Princeton University, 1960.
66. Ritchie, R. W. Classes of predictably computable functions. *Trans. AMS 106* (1963), 139-173.
67a. Rivest, R. L., Shamir, A., and Adleman, L. A method for obtaining digital signatures and public-key cryptosystems. *Comm. ACM 21,* 2 (February 1978), 120-126.
67b. Ruzzo, W. L. On uniform circuit complexity. *J. Comput. System Sci. 22* (1981), 365-383.
68a. Savage, J. E. *The Complexity of Computing.* Wiley, New York, 1976.
68b. Schnorr, C. P. The network complexity and the Turing machine complexity of finite functions. *Acta Informatica 7* (1976), 95-107.
69. Schönhage, A. Storage modification machines. *SIAM J. Comp. 9* (1980), 490-508.
70. Schönhage, A., and Strassen. V. Schnelle Multiplication grosser Zahlen. *Computing 7* (1971), 281-292.
71. Schwartz, J. T. Probabilistic algorithms for verification of polynomial identities. *JACM 27,* 4 (October 1980), 701-717.
72. Schwartz, J. T. Ultracomputers. *ACM Trans. on Prog. Languages and Systems 2,* 4 (October 1980), 484-521.
73. Shamir, A. On the generation of cryptographically strong pseudo random sequences. 8th Int. Colloquium on Automata, Languages, and Programming (July 1981). *Lecture Notes in Computer Science* 115. Springer Verlag, New York, 544-550.
74. Shannon, C. E. The synthesis of two terminal switching circuits. *BSTJ 28* (1949), 59-98.
75. Smale, S. On the average speed of the simplex method of linear programming. Preprint, 1982.
76. Smale, S. The problem of the average speed of the simplex method. Preprint, 1982.

77. Solovay, R., and Strassen, V. A fast monte-carlo test for primality. *SIAM J. Comput. 6* (1977), 84–85.
78. Stearns, R. E., Hartmanis, J., and Lewis, P. M. II Hierarchies of memory limited computations. *6th IEEE Symp. on Switching Circuit Theory and Local Design* (1965), 179–190.
79. Stockmeyer, L. J. The complexity of decision problems in automata theory and logic. Doctoral Thesis, Dept. of Electrical Eng., MIT, Cambridge, MA., 1974; Report TR-133, MIT, Laboratory for Computing Science.
80. Stockmeyer, L. J. Classifying the computational complexity of problems. Research Report RC 7606 (1979), Math. Sciences Dept., IBM T.J. Watson Research Center, Yorktown Heights, N.Y.
81. Strassen, V. Gaussian elimination is not optimal. *Num. Math. 13* (1969), 354–356.
82. Strassen, V. Die Berechnungskomplexität von elementarsymmetrischen Funktionen und von Interpolationskoeffizienten. *Numer. Math. 20* (1973), 238–251.
83. Baur, W., and Strassen, V. The complexity of partial derivatives. Preprint, 1982.
84. Toom, A. L. The complexity of a scheme of functional elements realizing the multiplication of integers. *Doklady Akad. Nauk. SSSR 150,* 3 (1963), 496–498. Translated in *Soviet Math. Doklady 3* (1963), 714–716.
85. Turing, A. M. On computable numbers with an application to the Entscheidnungsproblem. *Proc. London Math. Soc. ser. 2,* 42 (1963–7), 230–265. A correction. *ibid. 43* (1937), 544–546.
86. Valiant, L. G. The complexity of enumeration and reliability problems. *SIAM J. Comput. 8* (1979), 189–202.
87. Valiant, L. G. The complexity of computing the permanent. *Theoretical Computer Science 8* (1979), 189–202.
88. Valiant, L. G. Parallel computation. *Proc. 7th IBM Japan Symp. Academic 6 Scientific Programs*, IBM Japan, Tokyo (1982).
89. Valiant, L. G., Skyum, S., Berkowitz, S., and Rackoff, C. Fast parallel computation on polynomials using few processors. Preprint (Preliminary version in *Springer Lecture Notes in Computer Science 118* (1981), 132–139.
90. von Neumann, J. A certain zero-sum two-person game equivalent to the optimal assignment problem. *Contributions to the Theory of Games II.* H. W. Kahn and A. W. Tucker, Eds. Princeton Univ. Press, Princeton, NJ, 1953.
91. Yamada, H. Real time computation and recursive functions not real-time computable. *IRE Transactions on Electronic Computers EC-11* (1962), 753–760.
92. Yao, A. C. Theory and applications of trapdoor functions (extended abstract). *Proc. 23rd IEEE Symp. on Foundations of Computer Science. IEEE Computer Society, Los Angeles* (1982), 80–91.

Categories and Subject Descriptors:

F.1.2 **[Computation by Abstract Devices]**: Modes of Computation—*parallelism; probabilistic computation*; F.2.1 **[Analysis of Algorithms and**

Problem Complexity]: Numerical Algorithms and Problems—*computations on polynomials*; G.3 [**Mathematics of Computing**]: Probability and Statistics: *probabilistic algorithms*

General Terms:
Algorithms, Theory

Additional Key Words and Phrases:
Fast Fourier transform, Monte Carlo algorithm, NP-completeness