Reprinted from:
PERSPECTIVES ON COMPUTER SCIENCE: From the 10th Anniversary Symposium at the Computer Science Department, Carnegie-Mellon University
© 1977
ACADEMIC PRESS, INC.
NEW YORK SAN FRANCISCO LONDON

Time and Space†

Albert R. Meyer

Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, Massachusetts

Michael Ian Shamos

Department of Computer Science
Carnegie-Mellon University
Pittsburgh, Pennsylvania

Time and space are fundamental parameters for measuring the efficiency of algorithms, and the notion of trading off one for the other is a familiar one in the programmer's informal repertoire. Nonetheless, providing satisfactory mathematical definitions of computational time and space and establishing formal relationships between them remains a central problem in computing theory. In this chapter we examine the interplay between time and space determined by a variety of machine models and explore the connection between time and space complexity classes. We consider a number of possible inclusion relationships among these classes and discuss their consequences, along with recent results indicating that mechanical procedures may be available for reducing the space used by programs. This rosy picture is darkened somewhat by a counterexample due to Cobham, which states that minimum time and space cannot always be achieved by a single program.

1. INTRODUCTION

If I had had more time, I could have written you a shorter letter.

Blaise Pascal‡

Every programmer has observed that he can often reduce the storage required by a program at the expense of its running time. This can sometimes be done by compressing the data in clever ways; the added cost

†This research was supported in part under NSF grant GJ 43634X, contract number DCR 7412997 A01.

‡This quotation was kindly supplied by Theodore J. Ronca, Jr.

is the time taken to perform the encoding and decoding. Other times, it may be necessary to redesign the entire algorithm or use a different data structure for representing the problem in order to decrease storage. With extraordinary luck, the new representation may permit a reduction in both storage and execution time—a recent example is Hopcroft and Tarjan's linear-time planarity algorithm [Hopcroft and Tarjan 74], which masterfully exploits the list representation of graphs. In this chapter we will survey some of the theoretical results that bear on the question of whether the ability to exchange time for space is a general phenomenon in computation or merely a fortuitous property of some unrepresentative programs. We will indicate how to define computational space and time for several models of automata and try to present a convincing case that the notion of a time–space tradeoff transcends any specific machine or programming language.

To clarify the concepts of time and space, we look first at the problem of recognizing palindromes. A palindrome is a character string that is identical to its reverse, such as the owl's complaint "TOOHOTTOHOOT". Given a string, how much time and space are needed to determine whether or not it is a palindrome?

To gain an intuitive grasp of the question, let us use a familiar theoretical model and imagine that the input string is provided on a two-way, read-only input tape. That is, we can scan the tape one square at a time and move it in either direction one square at a time, but not change anything on it. One method is to begin at the left end of the string, "remember" the character there, and move to the right end to see if the character matches. Now, at the right end, we can pick up the second symbol from the right, travel back down the tape, and compare it with the second symbol from the left. This procedure is repeated until either every character is checked or a mismatch is found.

How much time does this method require to determine whether a given string is a palindrome? If the string to be checked is a palindrome N characters long, we will need $N/2$ trips across it, having average length approximately $N/2$. A reasonable definition of computation time is the number of primitive operations performed, in this case the number of moves made during the trips, or about $N^2/4$.

The amount of space used is not so apparent. Indeed, it may seem at first glance that no space at all is required other than the tape itself. However, during the scan we must remember in some way which square of the tape to stop at in order to check the current character. This requires being able to store numbers up to size N, which means that we have auxiliary space somewhere for about $\log N$ symbols, or we will get lost while trying to examine the string. What about the space occupied by the

program that is controlling this procedure? We will ignore such space for the purposes of this discussion because the size of the program is a constant, independent of the length of the input string. The justification is that no matter how long the program is there exist inputs so large that the program will be small by comparison.

A faster way to accomplish palindrome recognition is to copy the input tape into auxiliary memory and then compare the copy, character by character, with the input tape read backwards. This method requires a number of steps only proportional to N, but now the number of symbols that must be stored in memory rises to N as well.

Is there a single method that recognizes palindromes simultaneously in time proportional to N and space less than proportional to N? We shall return to this question in the last section.

2. TIME AND SPACE IN VARIOUS MACHINE MODELS

Let us make these intuitive concepts of time and space more precise, choosing initially the Turing machine model because it is simple and well-known. Let Turing machine M have a finite-state control, a two-way read-only input tape, and k semi-infinite work tapes, as in Fig. 1. This is the definition given by [Hopcroft and Ullman 69]. Assume that an input string x is given, symbol by symbol, on sequence of consecutive nonblank squares on the read-only tape. Let $T_M(x)$ be the number of moves made by M before halting, when presented with input x. We define the time complexity $T_M(N)$ of M as

$$T_M(N) = \max\{T_M(x) \mid \text{length}(x) = N\},$$

that is, the largest number of steps taken by M on any input of length N. Similarly, we define the space complexity $S_M(N)$ as the maximum number

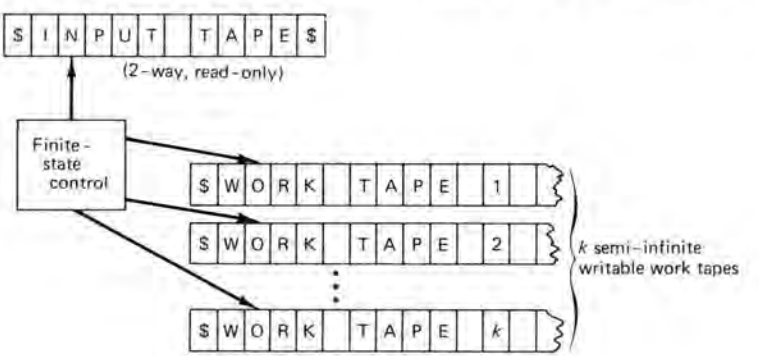

Fig. 1. A multitape Turing machine.

of *work tape squares* scanned by M on any input of length N. From now on we will dispense with the subscript M if no ambiguity results.

The first hint of a formal connection between time and space is that $T(N)$ and $S(N)$ are not independent.

Let M be a Turing machine with k work tapes. For convenience, assume that M halts on every input and that $S(N) \geqslant \log N$. (The latter assumption holds in most cases, since, speaking informally, M needs this much space to detect which part of its input is being read.)

Theorem 1: There is a constant $\epsilon > 0$ such that for all N,

$$\epsilon \log T_M(N) \leqslant S_M(N) \leqslant kT_M(N).$$

Sketch of proof: Since M has only k work tapes, each with a single read–write head, it can visit at most k new work tape squares at each step, so obviously $S(N) \leqslant kT(N)$. To prove the other inequality, consider the number C of distinct configurations in which M can find itself. If M has k work tapes, m internal states, and a tape alphabet of a symbols, then a configuration is determined uniquely by the state, the position of the read head on the input tape, the positions of the work tape heads, and the contents of the storage tapes. Thus $C \leqslant m(N+2)(S(N))^k a^{S(N)k}$. Now, if M ever enters the same configuration twice it will not halt, so $T(N) \leqslant C$, and the result follows by taking logarithms.

Either of the bounds of Theorem 1 can essentially be achieved. For example, there is a machine M that runs for precisely 2^N steps while using precisely N tape squares for any input of length N, so that $S_M(N) = \log_2 T_M(N)$. There also exists a machine L, which visits new work tape squares with all but one of its work tape heads at every time step, so that $S_L(N) \geqslant (k-1)T_L(N)$ for such a machine.

While either of the bounds may be tight for specific machines, we are interested in solving problems (computing recursive functions), for which there are many Turing machines that will work, each with possibly different complexities S and T. One of the machines may use very little time, and another may use little space, but Theorem 1 says nothing about this possibility, since it applies to a single specific machine.

Let us now see how invariant the quantities time and space remain as we modify the machine model.

2.1. Turing Machine Variants

A natural way of generalizing the Turing machine is to drop the restriction that the work tapes be one dimensional. Such variant Turing machines (VTMs) might be supplied with a finite-number of finite dimen-

sional work "tapes" each of which could be scanned by a finite number of read–write heads. For example, a VTM with a single two-dimensional tape scanned by three heads is illustrated in Fig. 2. In a single step, each of the heads may independently change the symbol in the square it is scanning and move up, down, left, or right. It may be helpful to think of a two-dimensional VTM as having pieces of paper on which to compute as a human might.

In order to be able to compare VTMs and ordinary multitape TMs, we supply the VTMs with a one-dimensional input tape as well as their work tapes. Time and space for VTMs are defined exactly as before, namely, as the number of steps performed and the number of work tape squares scanned.

Theorem 2: [Hartmanis and Stearns 65]. For any VTM V with time complexity $T_V(N)$, there is a Turing machine M, which computes the same function as V, using time that is at most proportional to $(T_V(N))^2$.

In particular, this means that whatever can be done by VTMs in time bounded by a polynomial in N can also be done by ordinary Turing machines in polynomial time. It is known, incidentally, that $n+1$-dimensional VTMs are a bit faster than n-dimensional VTMs; adding a reasonable technical condition that simulations be "on-line", it has even been shown that the quadratic slowdown of Theorem 2, when ordinary Turing machines simulate VTMs, cannot be improved [Hennie 66].

The result for space is even more attractive:

Theorem 3: For any VTM V there is a Turing machine M, with possibly more states and a larger tape alphabet, which computes the same function as V, using no more space than V.

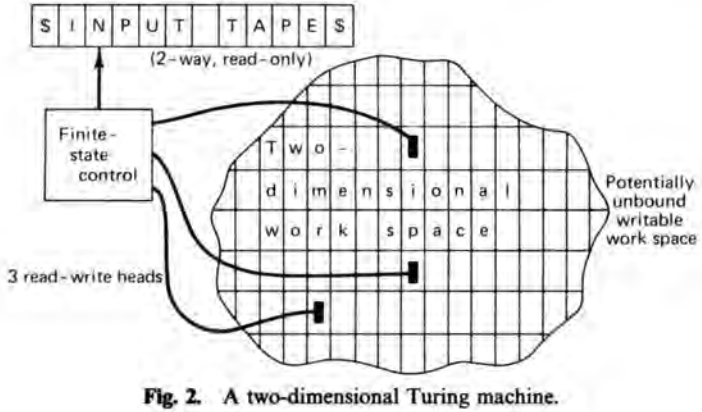

Fig. 2. A two-dimensional Turing machine.

Thus, as we increase the dimensionality of the working storage, space remains invariant and time is preserved to within a polynomial.

2.2. Counter Machines

We turn now to a model that does not outwardly resemble a Turing machine, but is equivalent in that it can compute any function that a TM can compute. A *counter machine* is composed of the following:

(1) A finite-state control.
(2) A two-way read-only input tape.
(3) A finite collection of counters, each of which can contain an arbitrary integer.
(4) Three instructions to control the counters:
  (a) Increment a counter by one.
  (b) Decrement a counter by one.
  (c) Test to determine whether a counter is zero.

All the counters can be tested or modified in different ways at each time step.

The time used by a counter machine is the number of steps that pass before the machine halts, a direct extension of the Turing machine definition of time. One straightforward definition of the space used in processing an input is the largest absolute value attained by any of the counters. Let CMspace($f(N)$) denote the family of formal languages that can be recognized by a counter machine using at most space $f(N)$ on inputs of length N. Define TMspace($f(N)$), CMtime($f(N)$), and TMtime($f(N)$) similarly. Then

Theorem 4: [Fischer *et al.* 68]. For $f(N) \geqslant N$,

$$\text{CMspace}(f(N)) = \text{TMspace}(\log f(N)).$$

Thus this somewhat arbitrary space measure for counter machines turns out to be the same as Turing machine space except for a logarithmic distortion of scale. In fact, Theorem 4 makes it clear that a better definition of space for counter machines should have been the *size of the radix representation* of the largest value in a counter, in which case CM space and TM space would turn out to be the same.

Curiously, CM time also relates directly to CM space. Abusing notation in a hopefully perspicuous way, let CMtime(poly($f(N)$)) denote the family of languages recognizable by a counter machine using a number of steps bounded by any polynomial in $f(N)$ on inputs of length N.

Theorem 5: [Fischer *et al.* 68]. For $f(N) \geqslant N$,

$$\text{CMtime}(\text{poly}(f(N))) = \text{CMspace}(f(N)).$$

If Theorem 5 muddies which is time and which is space for counter machines, it serves with Theorem 4 to make the point that these quantities still reflect the underlying quantity of Turing machine space.

2.3. Space and Time in Formal Language Theory

The Chomsky hierarchy of formal languages is defined by structural considerations alone. Regular, context-free, context-sensitive, and type-0 grammars are distinguished by the form of their production rules. These grammars and their relation to automata are one of the standard topics for courses in the theory of computation. Hopcroft and Ullman [69] provide an introductory textbook treatment.

Time and space enter in an elegant and unexpected way. Kuroda [64] and Landweber [63] showed that the context-sensitive languages are precisely those that can be recognized by a nondeterministic Turing machine operating in linear space, a so-called *linear bounded automaton* (LBA).

The concept of a nondeterministic computation enters here in an essential way. A Turing machine or similar automaton is *nondeterministic* when the state of the machine and symbols read by its heads determine, not necessarily a *unique* next step of computation, but possibly more than one permissible next step. Thus, a nondeterministic machine has many permissible complete computations which it may perform in response to a single input word. It is said to *accept* an input word if *at least one* of its possible computations leads to acceptance of the input; the time (or space) required to accept an input word is taken to be the minimum number of steps (or tape squares) among all accepting computations.

Note that there is nothing probabilistic in these notions of nondeterministic computation. Nondeterministic automata simply specify a family of possible computational behaviors any one of which may lead to successful acceptance. (The adjective "multipath" has been suggested as more appropriate than "nondeterministic" to describe these automata, but, unfortunately, it has not been accepted by the research community.) The possible computations can be thought of as possible proofs in a formal proof system. Following each line or step of a proof, several next steps may be possible, and a theorem is proved just when there is some possible sequence of steps of proof which lead to it. The definition of time required by a nondeterministic automaton to accept an input is thus analogous to the number of steps in the shortest proof of a theorem.

The way in which a nondeterministic machine "performs" a computation is quite different from that of ordinary deterministic computers, and there is no direct or efficient means known by which nondeterministic computations can be carried out by ordinary computers. For this reason nondeterministic computation may seem an artificial concept, but it has proved to be a fruitful one. Indeed some of the most difficult and important questions in the theory of computation involve the relation between deterministic and nondeterministic time and space. The two most celebrated problems of this kind are the following:

1. *The LBA problem*—whether deterministic and nondeterministic LBAs accept the same family of languages.
2. *The $P = NP$ problem*—whether P, the family of languages recognizable by Turing machines in time bounded by a polynomial, is equal to NP, the family of languages recognizable by *nondeterministic* Turing machines in time bounded by a polynomial.

For a discussion of the profound consequences of a solution (affirmative or negative) of the $P = NP$ problem see Cook [71a] and Karp [72], and for the LBA problem see Hartmanis and Hunt [74] . For example, if $P = NP$, then there exist far more efficient algorithms than any now known for such classical operations-research optimization problems as the knapsack or traveling salesman problems and a host of other apparently intractable computations.

The languages determined by regular grammars and recursive grammars can also be characterized by bounds on time or space although the bounds degerate—the regular languages are precisely the family TMspace(1) and the recursive languages are precisely those recognizable without any bounds on time or space. The context-free languages cannot be characterized precisely in terms of time or space. (For example, it is known that there are context-free languages that require space proportional to $\log N$, but there are languages recognizable in space $\log N$ that are not context-free [Lewis *et al.* 65, Alt and Mehlhorn 76].) There *is* an elegant characterization of context-free languages in terms of pushdown automata, however, and we shall indicate in Section 2.9 how a simple extension of the pushdown automaton model ties together the notions of time and space.

2.4. Stack Automata

Explaining the relation between the syntactic structure of grammars and the complexity of recognizing the languages they generate can be counted among the fundamental insights of formal language theory. There is

another such relation between a peculiarly structured computer model called a stack automaton and Turing machine space.

Stack automata were initially proposed as a variant of pushdown automata that had additional abilities to cope with certain constructs in computer languages like ALGOL. Basically they are pushdown automata that can "peek" at the pushdown store without modifying it. Specifically, a stack automaton is composed of the following:

(1) A finite state control.
(2) A two-way read-only input tape.
(3) A pushdown stack with a two-way head. The head is free to move up and down the stack reading symbols, but it may write a symbol only when it is at the top of the stack. Symbols are never removed from the stack.

Actually this describes only one species, called a two-way deterministic nonerasing stack automaton (Fig. 3), among a bestiary of stack automata that have been collected. Let 2DNESA denote the class of languages accepted by two-way deterministic nonerasing stack automata. Notice that there is no *a priori* bound imposed on how much the stack may grow during a computation. In fact, the stack may grow to be more than exponentially longer than the input, even in halting computations. However, the structural limitation on this large storage space imposed by the stack discipline diminishes its value to that of considerably less Turing machine space.

Theorem 6: [Hopcroft and Ullman 67].

$$2\text{DNESA} = \text{TMspace}(N \log N).$$

That 2DNESA should contain languages of only bounded computational complexity might have been anticipated by students of automata theory, but that 2DNESA should have an *exact* characterization in terms

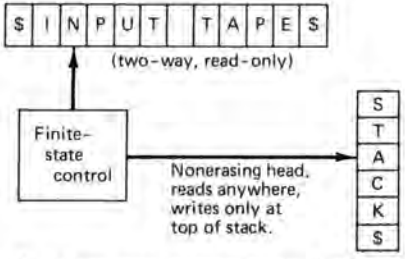

Fig. 3. A two-way, nonerasing stack automaton.

of Turing machine space complexity, and that the space on the Turing machine should be so much smaller than that on the stack, is remarkable. The proof of Theorem 6 is one of the little gems of automaton theory; it has the unusual aspect that the equivalence is nontrivial in both directions. The theorem itself reveals an instance in which the concept of space appears unexpectedly in a fundamental role.

2.5. Vector Random-Access Machines

We saw earlier that, roughly speaking, time on counter machines corresponds to logarithmic space on Turing machines (Theorems 4 and 5). There is another model of computation, however, in which time bears an even closer relationship to TM space—the vector random-access machine (VRAM):

(1) A finite-state control.
(2) A two-way read-only input tape.
(3) A finite number of registers, each holding a bit vector of potentially unbounded length.
(4) An instruction set comprising the operations of assignment, binary addition and multiplication, bitwise OR and NOT, with indirect addressing (that is, the contents of a register may be used as the address of an operand).
(5) A test-for-zero operation.

This model differs from more primitive ones in that multiplication is regarded as an elementary operation and data can be accessed directly instead of through the laborious mechanism of tape storage. VRAMs were intended as a model that better reflects "real" computers in many circumstances. The indirect addressing feature is familiar in actual machine languages, although it turns out to play an unimportant role in the following theorem, that is, the theorem is true even if indirect addressing is disallowed.

Theorem 7: [Pratt and Stockmeyer 76, Hartmanis and Simon 74]. For $f(N) \geqslant N$,

$$\text{VRAMtime}(\text{poly}(f(N))) = \text{TMspace}(\text{poly}(f(N))).$$

The set of languages recognizable by VRAMs operating in polynomial time is thus the same as the set of languages recognizable by Turing machines in polynomial space. This is another result in which the proofs of containment in both directions are nontrivial. The method employed is to show that each machine can simulate the other, but these simulations are

difficult. The trouble stems from the fact that on a VRAM, multiplication takes one unit of time, no matter how long the bit vectors are. So in polynomial time one can create bit vectors that are exponentially long, and the TM performing the simulation cannot simply maintain a copy of the VRAM memory, or it would not operate in polynomial space. Again we have an instance in which time and space may appear in each other's guise.

2.6. Recursive Functions

Another way to specify computable functions, which at first sight seems quite different from Turing machines or grammars, is by means of recursive definitions. For example, if $A(x, y) = x + y$, then we can define another function $M(u, v)$ on the nonnegative integers by the equations

$$M(0, v) = 0,$$

$$M(u + 1, v) = A(v, M(u, v)).$$

It is not too hard to see that, despite the apparent circularity of recursively defining M in terms of itself, the function M is uniquely determined by these equations and in fact $M(u, v) = u \times v$.

These equations for defining M from A conform to a scheme of recursive definition known as *primitive recursion*. Computable functions can be classified by the form of recursive schemes sufficient to define them, just as formal languages can be classified by forms of grammars or automata sufficient to generate them.

One such classification was proposed by Grzegorczyk [53]. Grzegorczyk's class $\mathcal{E}^2$ is defined by starting with the functions of addition and multiplication, and then constructing new functions by composing, substituting constants and new variables, and applying primitive recursion to functions already obtained. The application of primitive recursion is constrained so that only functions bounded above by functions already obtained may be constructed. A completely different description of $\mathcal{E}^2$ is provided by the following result.

Theorem 8: [Ritchie 63]. $\mathcal{E}^2$ equals the class of functions on the nonnegative integers that are computable by Turing machines using space proportional to the length in radix notation (e.g., arabic numerals) of integers presented as inputs.

Results similar to Theorem 8 can be proved about Grzegorczyk's classes $\mathcal{E}^3$, $\mathcal{E}^4$, . . . , and other classes which have been studied such as the primitive recursive functions or the double recursive functions [Cobham

64, Meyer and Ritchie 72]. Such computational characterizations of recursive definitions help to clarify their expressive power and have contributed to the solution of some technical problems relating different classifications [Meyer and Ritchie 67]. Thus we see another example of an independent line of research about recursive functions converging on underlying concepts of time and space.

2.7. Boolean Networks and Table Look-Up Time

Boolean networks (also called logical or combinational networks) are one of the standard models used by digital hardware designers. Such a network with n input lines and one output line provides a recipe for computing a boolean function from the n zeros or ones that are presented at the inputs to a single zero or one at the output.

The number of "gates" at which atomic operations combining zeros and ones are performed in the network provides an obvious measure of the cost or size of a network (Fig. 4). The *combinational complexity* of a boolean *function* is defined to be the minimum size of any network that computes the function.

This measure of complexity of boolean functions has an intuitive appeal beyond its familiarity in hardware design. Digital computation as currently understood means the manipulation of discrete symbols that ultimately can be coded as strings of zeros and ones. The basic operations by which

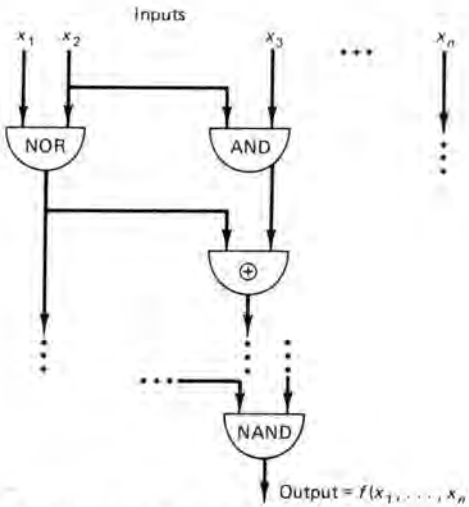

Fig. 4. An n-input boolean network with two-input gates.

such symbols are combined or compared must also ultimately reduce to the atomic operations performed on pairs of zeros and ones at gates. In this sense one would expect the combinational complexity of a boolean function to reflect the irreducible minimum effort necessary to compute the function.

A particular boolean function always has a fixed finite number of zero–one valued arguments and so only represents a finite computational problem. But it is a simple matter to extend the measure of combinational complexity to any infinite problem of interest—recognizing the infinite set of prime numbers, for example. Define the combinational complexity of the set of primes to be a function of N equal to the combinational complexity of the boolean function of N arguments, which has value one if and only if the values of the arguments comprise the N-bit representation in binary notation of a prime number.

Notice that at first sight this formulation of the complexity of recognizing languages is very different from the Turing machine approach. To recognize some formal language L we require a single Turing machine which correctly handles the possibly infinite whole of L. Moreover, the Turing machine time or space complexity of a language L may grow as rapidly as any recursive function of the input length N. On the other hand, the combinational complexity of L only reflects the complexity of larger and larger finite segments of L, since entirely different networks may be used for different values of N. The combinational complexity of any L can never be much greater than $2^N/N$ because any boolean function of N arguments may be computed by a circuit of this size. (Remember that simply expanding a boolean function into disjunctive normal form would already yield an upper bound on combinational complexity of $N2^N$.) Furthermore, Turing machine complexity only makes sense for computable or at best recursively enumerable languages L, whereas combinational complexity has a perfectly definite meaning for any language L whatsoever.

The connection between these complexities can be made by providing Turing machines with oracles. An oracle Turing machine has, in addition to the usual paraphernalia of input and work tapes, an *oracle tape* on which an infinite sequence of zeros and ones may be presented. The oracle tape has a single read-only two-way head, which may move between adjacent squares on the oracle tape. The same pattern on the oracle tape is preserved for each input given on the input tape. In this way the oracle Turing machine can be thought of as having a fixed infinite table of answers or subresults available on its oracle tape. Of course, if the head on the oracle tape is far away from a desired entry in the table, the lookup may take a long time.

Let Combinational($T(N)$) denote the family of languages whose combinational complexity is at most proportional to $T(N)$. Let Oracle TMtime($T(N)$) denote the family of languages that can be recognized within time $T(N)$ by some oracle Turing machine provided with some appropriate oracle tape.

Theorem 9: [Pippenger and Fischer 77, Schnorr 75]. For $T(N) \geqslant N$,

$$\text{OracleTMtime}(T(N)) \subset \text{Combinational}(T(N) \log T(N)),$$

and

$$\text{Combinational}(T(N)) \subset \text{OracleTMtime}(\text{poly}(T(N))).$$

Thus the time measure for oracle Turing machines, which models the time required to perform computations by table look-up, matches well with another intuitively appealing concept of complexity based on boolean networks.

If we regard the size of a network as being analogous to storage space, then Theorem 9 provides still another example in which a space measure for one model corresponds to a time measure on another. Curiously, a reverse correspondence also holds in this case. The time required by a network is usually defined to be the maximum depth of the network, that is, the length of the longest path from any input wire to the output wire. Using this definition, Borodin [75] has observed that the network–time complexity of any language corresponds (to within a quadratic polynomial) to the oracle Turing machine space required to recognize the language.

As an aside it seems worth mentioning that the first containment given in Theorem 9 provides an interesting technique for hardware design. In some cases it is easier to see how to program a Turing machine to perform certain computations efficiently than it is to design a small circuit. The proof of Theorem 9 provides a simple means of translating an efficient Turing machine into a comparably economical circuit.

2.8. Tapes and Heads

Thus far, time has proved to be invariant from machine to machine to within a polynomial of low degree. But for accurate guidance in concrete cases, we need to have a much more exact idea of the effect of machine structure on speed of computation. Unfortunately such results are few and difficult to obtain; we shall mention two.

Turing machines as we have defined them with several one dimensional

tapes but only one head per tape can obviously be simulated without time loss by Turing machines with only a single tape but with several independent heads on the tape. (Simply divide the single tape into "tracks" and let each head attend to only one track.) The converse, that multitape machines can simulate multihead machines without time loss, is also true but seems to require an intricate simulation requiring nine times as many tapes as heads to be simulated [Fischer *et al.* 72]. It is not known whether the number of tapes can be kept down to the number of heads. Neither is it known if the result can be extended to two-dimensional tapes.†

Recently Aanderaa [74] settled the question posed by Hartmanis and Stearns [65] of whether $k+1$ one-dimensional tapes are faster than k one-dimensional tapes. By means of a sophisticated analysis, Aanderaa was able to show that there are languages recognizable in time exactly N, so called "real-time" recognizable languages, on $k+1$ tape Turing machines that cannot be recognized in time $N+$ constant on machines with only k tapes. It remains open whether three tapes are more than a *constant multiple* faster than two tapes. It is also not known whether Aanderaa's results extend to two-dimensional tapes. In the one-dimensional case, we at least know that many tapes cannot be too musch faster than two tapes: Hennie and Stearns [66] have shown that TMtime($T(N)$) $\subset$ Two-tape TMtime($T(N) \log T(N)$).

2.9. Auxiliary Pushdown Machines

Rounding out the menagerie of machine variants is the auxiliary pushdown automaton (APDA) of Cook [71b], which is made up of the following:

(1) A Turing machine, possibly nondeterministic, with a two-way read-only input tape and a finite number of work tapes.

(2) A pushdown stack subject to the same restrictions as those on a conventional PDA.

Since an APDA (Fig. 5) has an embedded Turing machine, it is clear that the pushdown store is unnecessary in that it does not expand the class of languages recognizable by an APDA. In fact, the pushdown store is less powerful than a single additional work tape, but its inclusion will be justified by Cook's measure of APDA space. He counts only the number of *work tape* squares scanned during the computation—space on the stack, potentially unbounded, is free!

†(Added in proof.) A solution to this problem has recently been announced by Seiferas and Leong at Penn State.

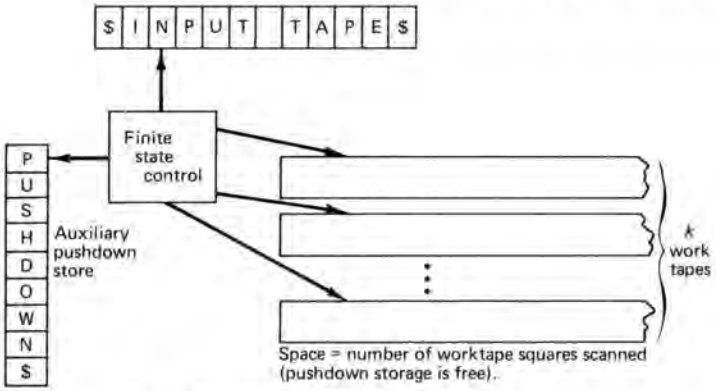

Fig. 5. An auxiliary pushdown automaton.

Theorem 10: [Cook 71b]. If $T(N) \geqslant N$, then

$$\text{APDAspace}(\log T(N)) = \text{TMtime}(\text{poly}(T(N))).$$

Cook's theorem thus asserts that any language recognizable in time T on a Turing machine can be recognized in space $\log T$ on an APDA, and conversely space S on an APDA can be simulated in time exponential in S on a Turing machine. These results apply, it turns out, equally well to nondeterministic APDAs.

Again the proof involves clever simulations of APDAs by Turing machines and vice versa, and again the simulations cannot be carried out by "step-by-step" simulations since, for example, an APDA operating within space $\log N$ may actually run for 2^N steps, whereas Theorem 10 asserts that such an APDA can be simulated by a Turing machine running in time poly(N). Giuliano [72] and Ibarra [71] extend Cook's methods to define auxiliary stack automata and obtain similar results; a combination stack-PDA is the basis for further generalizations by van Leeuwen [76].

While the addition of free pushdown storage may seem contrived, it motivates an important unanswered question in automaton theory. Theorem 1 says that, for Turing machines, space is bracketed between T and $\log T$. For an APDA, space is equal to $\log T$. The open question is whether or not the containment holds when the pushdown store is removed and only an ordinary Turing machine remains. This is tantamount to asking whether any Turing machine that uses time $T(N)$ can be "reprogrammed", or transformed, into another Turing machine that uses only space $\log(T(N))$ but possibly more time. (Theorem 1 implies that as much as $\text{poly}(T(N))$ time might be used after such reprogramming.) In the next section we discuss some of the implications of such a time–space tradeoff.

3. INCLUSION RELATIONS AMONG COMPLEXITY CLASSES

Although we do not know whether TMtime(T) is contained in TMspace(log T), or vice versa, or even whether the classes are comparable, there is nothing to prevent our examining the several alternatives.

POSSIBLE RESULT 1: TMtime(T) $\subset$ TMspace(log T).

If PR1 is true, then by Theorem 1 it is actually the case that TMspace(log T) and TMtime(poly(T)) are the same. Hence the two fundamental complexity measures of time and space would be measures on different scales of the same underlying quantity. Further, if PR1 is true, an immediate consequence is a positive solution of the LBA problem mentioned in Section 2.3.

On another front, PR1 might provide some help in certain mechanical theorem-proving tasks. For example, a new mechanical procedure significantly improving Tarski's decision method for the theory of the real field has recently been developed [Collins 75]. This procedure requires time and space that both grow doubly exponentially (like 2^{2^N}). PR1 would imply that space for this procedure could at least be reduced to ordinary exponential growth, and since space, not time, is often the limiting factor in practical mechanical theorem proving, such a reduction might make a few more short theorems accessible to the method.

We cannot pass by this example of mechanical theorem proving without also mentioning one of the triumphant results of complexity theory: within the past four years ways have been found to *prove* that most of the classical theorem-proving problems of mathematical logic, even if they are solvable in principle by Turing machines, are of exponential time complexity or worse. (See Meyer [75] for a summary of these results.) This includes the above problem of proving theorems about the real field, so that the general task of proving such theorems mechanically is inherently intractable [Fischer and Rabin 74].

To speculate on a speculation, let us remark that if PR1 is true, it might be possible to refine the correspondence between Turing machine measures and boolean network measures mentioned in Section 2.7, to show that network depth is the logarithm of network size. This would imply the existence of fast boolean circuits of depth proportional to log N for finding shortest paths in graphs, parsing context-free languages, inverting matrices, and dividing binary numbers [Csanky 76, Valiant 75]. For each of these problems the best currently known networks require depth proportional to $(\log N)^2$.

Since PR1 is a very powerful conjecture, let us consider instead some weaker possibilities:

POSSIBLE RESULT 2: $\text{TMtime}(\text{poly}(N)) \subset \text{TMspace}(N)$.

Here we assume not a logarithmic reduction but only that polynomial time algorithms can be run in *linear* space (on a possibly different Turing Machine). If PR2 is true, then, in a very general and far-reaching sense, any computer program using time N^k (which might simultaneously be using space N^k as well) can be rewritten to use only space linear in N. The cost of this improvement is that the resulting program may use exponential time. Such an effective transformation would be a programming technique of vast importance, leading potentially to optimizing compilers of great power. We confidently expect that it would be an idea fully as useful as such fundamental computer science concepts as recursion and iteration.

POSSIBLE RESULT 3: $\text{TMspace}(N) - \text{TMtime}(\text{poly}(N))$ is nonempty.

That is, there may exist some problem that can be solved in linear space but not in polynomial time. PR3 would imply that many problems for which no fast algorithms are known are, in fact, computationally infeasible because they cannot be done in polynomial time. Among these are (1) minimizing the number of states in a nondeterministic finite automaton and deciding the equivalence of regular expressions [Meyer and Stockmeyer 72], (2) deciding first-order predicate calculus in which equality is the only predicate [Stockmeyer 76], and (3) determining which player has a winning strategy in some simple games on graphs such as generalized versions of HEX and the Shannon switching game [Even and Tarjan 76].

All of the above possibilities are implied by PR1. Let us see what would happen if the inclusion in PR2 were reversed.

POSSIBLE RESULT 4: $\text{TMspace}(N) \subset \text{TMtime}(\text{poly}(N))$.

This is an electrifying possibility, since it would mean that $P = NP$, that deterministic and nondeterministic Turing machines operating in polynomial time accept the same set of languages. PR4 would also imply that all the apparently infeasible problems mentioned after PR3 could in fact be solved in polynomial time.

If any of the possibilities PR1–PR4 are true, then interesting conclusions follow. Pessimistically, however, there is a fifth choice. It may be that there is a problem in $\text{TMtime}(\text{poly}(N))$ that cannot be solved in linear space. Some work of Cook [74], Cook and Sethi [74], and Jones and Laaser [76] suggests that this "uninteresting" possibility may be the correct one, and

our intuition (albeit a faulty barometer) about difficult problems tends to support this view.

It is disappointing that we know so little about time and space as to be unable to distinguish between the blatantly contradictory hypotheses PR3 and PR4. It is positively irksome, though, that we know definitely that the classes of polynomial time and linear space are not the same [Book 72]. We can prove this by showing that there exist transformations that preserve polynomial-time recognizability but not linear-space recognizability, but no example is known of a problem that belongs to one class and not the other. Yet such a problem must exist![†]

3.1. Space Is More Valuable Than Time

We come now to the recent result of Hopcroft *et al.* [75], which is the strongest theorem known regarding time and space. Informally, it says that having space T is strictly more valuable than having time T:

Thorem 11: $\mathrm{TMtime}(T \log T) \subset \mathrm{TMspace}(T)$.

This theorem is the first solid example we know that guarantees the existence of a mechanical procedure for reducing space. It asserts, for example, that programs that run in time $N \log N$, even if they use space $N \log N$, can be reprogrammed to use only linear space. The price we pay is that the time required for the new algorithm may be exponential. A weakness of the result is that it appears to apply only to ordinary multitape Turing machines with one-dimensional tapes, and not to VTMs, but the theorem is a very good beginning. It was proved by means of a particularly clever simulation on one-dimensional tapes and will undoubtedly be a focal point of future work on space and time.

For completeness we mention an earlier result of this kind, which applies to the highly restricted model of classical Turing machines with only a single one-dimensional tape: for these machines time T^2 can be simulated in space T [Paterson 72].

3.2. Space–Time Tradeoff

The central question at this point is whether there are any *inherent* time–space tradeoffs. Theorem 11 shows how to reduce space in certain cases, but it does not claim that the time *must* increase. It may be that

[†](Added in proof.) Some further surprising connections between time and space have recently been observed by Kozen [76] and Chandra and Stockmeyer [76].

minimal time and space are achievable by the same program. At present, there is only one known counterexample to this enticing possibility, due to Cobham [66]:

Theorem 12: If a Turing machine that performs palindrome checking uses time $T(N)$ and space $S(N)$, then $T(N) \times S(N)$ is at least proportional to N^2, and this bound is achievable in each of these cases:

(a) $T(N) = 2N$,
(b) $T(N) = N^2/\log N$,
(c) $T(N) = N^{(1+r)}$, where r is a rational between zero and one.

This quadratic lower bound for the product of time and space actually applies more generally to all manner of machine models besides Turing machines. The proof rests on analyzing the number of different internal configurations which a palindrome-checking automaton must assume as it crosses boundaries between tape squares on its input tape. The proof does not apply, however, if the input head can jump between non-adjacent input tape squares in a single step. The ideas of the proof do not seem to extend to yield larger than quadratic lower bounds.

Nonetheless, Cobham's theorem is the only instance in which we can prove the existence of a tradeoff that most programmers (and theorists) believe occurs in some form or other. Thus the palindrome problem, which we first explored in order to develop an intuitive feeling for computational time and space, provides the first piece of evidence that we must give up one in order to reduce the other.

REFERENCES

Aanderaa, S. O., On k-tape versus $(k-1)$-tape real time computation. *SIAM-AMS Prog.* **7**, 75–96 (1974).

Alt, H. and Mehlhorn, K., Lower bounds for the space complexity of context-free recognition. In *Automata Languages and Programming, Third International Colloquium*, (S. Michaelson and R. Milner, eds.) 338–354, Edinburgh Univ. Press, 1976.

Book, R. V., On languages accepted in polynomial time. *SIAM J. Computing* **1**, 281–287 (1972).

Borodin, A., Some remarks on time-space and size-depth. Computer Science Dept. Rep., Univ. of Toronto, Toronto, Ontario, 1975.

Chandra, A., and Stockmeyer, L., Alternation. *Proc. 17th IEEE Symp. Foundations of Computer Science 1976*, 98–108.

Cobham, A., The intrinsic computational difficulty of functions. *Proc. Intern. Cong. Logic, Methodology, Philos. Sci., 1964*, 24–30.

Cobham, A., The recognition problem for the set of perfect squares. *Rec. 7th IEEE Symp. Switching and Automata Theory, 1966*, 78–87.

Collins, G. E., Quantifier eliminations for real closed fields by cylindrical algebraic decomposition. In (*Automata Theory and Formal Languages 2nd G.I. Conference*), *Lecture Notes in Computer Science*, Vol. 33, 134–183. Springer Verlag, New York, 1975.

Cook, S. A., The complexity of theorem proving procedures. *Proc. 3rd ACM Symp. Theory of Computing. Shaker Heights, Ohio, May 1971*a, 151–158.

Cook, S. A., Characterizations of pushdown machines in terms of time-bounded computers. *J. ACM* **18** 1, 4–18 (1971b).

Cook, S. A., An observation on time-storage trade off. *J. Computer and System Sciences* **9** 3, 308–316 (1974).

Cook, S. A., and Sethi, R., Storage requirements for deterministic polynomial time recognizable languages. *Proc. 6th ACM Symp. Theory of Computing, Seattle, Washington, April 1974*, 33–39.

Csanky, L., Fast parallel matrix inversion algorithms. *SIAM J. Computing* **5** 4, 618–623 (1976).

Even, S., and Tarjan, R. E., A combinatorial problem which is complete in polynomial space. *J. ACM* **23** 4, 710–719 (1976).

Fischer, M. J., and Rabin, M. O., Super-exponential complexity of Presburger arithmetic. *SIAM-AMS Proc.* **7**, 27–41 (1974).

Fischer, P. C., Meyer, A. R., and Rosenberg, A. L., Counter machines and counter languages, *Mathematical Systems Theory* **2**, 265–283 (1968).

Fischer, P. C., Meyer, A. R., and Rosenberg, A. L., Real-time simulation of multihead tape units, *J. ACM* **19** 4, 590–607 (1972).

Giuliano, J. A., Writing stack acceptors. *J. Computer and System Sciences* **6**, 168–204 (1972).

Grzegorczyk, A., Some classes of recursive functions. *Rozprawy Mat.* **4**, 1–45 (1953).

Hartmanis, J., and Hunt, H. B., III, The LBA problem and its importance in the theory of computing. *SIAM-AMS Proc.* **7**, 1–26 (1974).

Hartminis, J., and Simon, J., On the power of multiplication in random access machines. *15th IEEE Computer Soc. Symp. Switching and Automata Theory*, 13–23 (1974).

Hartmanis, J., and Stearns, R. E., On the computation complexity of algorithms. *Trans. Amer. Math. Soc.* **117**, 285–306 (1965).

Hennie, F. C., On-line Turing machine computations. *IEEE Trans. Computers* **EC-15**, 35–44 (1966).

Hennie, F. C., and Stearns, R. E., Two-tape simulation of multitape Turing machines. *J. ACM* **13** 4, 533–546 (1966).

Hopcroft, J., and Tarjan, R., Efficient planarity testing. *J. ACM* **21** 4, 549–568 (1974).

Hopcroft, J. E., and Ullman, J. D., Nonerasing stack automata. *J. Computer and System Sciences* **1** 2, 166–186 (1967).

Hopcroft, J. E., and Ullman, J. D., *Formal Languages and Their Relation to Automata*. Addison–Wesley, Reading, Massachusetts, 1969.

Hopcroft, J., Paul, W., and Valiant, L., On time versus space and related problems. *Proc. 16th IEEE Symp. Foundations of Computer Science, 1975*, 57–64.

Ibarra, O. H., Characterizations of some tape and time complexity classes of Turing machines in terms of multihead and auxiliary stack automata. *J. Computer and Systems Sciences* **5**, 88–117 (1971).

Jones, N. D., and Laaser, W. T., Complete problems for deterministic polynomial time. *Theoretical Computer Science* **3** 1, 105–117 (1976).

Karp, R. M., Reducibility among combinatorial problems. In *Complexity of Computer Computations* (R. E. Miller and J. W. Thatcher, eds.), 85–104. Plenum, New York, 1972.

Kozen, D., On parallelism in Turing machines. *Proc. 17th IEEE Symp. Foundations of Computer Science 1976*, 89–97.

Kuroda, S. Y., Classes of languages and linear-bounded automata. *Information and Control* **7**, 207–223 (1964).

Landweber, P., Three theorems on phrase structure grammars of type 1. *Information and Control* **6** 2, 131–136 (1963).

Lewis, P. M., II, Stearns, R. E., and Hartmanis, J., Memory bounds for recognition of context-free and context-sensitive languages. *Rec. 6th IEEE Symp. Switching Circuit Theory and Logical Design, 1965*, 191–202.

Meyer, A. R., The inherent computational complexity of theories of ordered sets. *Proc. Intern. Cong. Mathematicians* **2**, 477–482 (1975).

Meyer, A. R., and Ritchie, D. M., The complexity of loop programs. *Proc. 22nd Nat. Conf., ACM, 1967*, 465–469.

Meyer, A. R., and Ritchie, D. M., A classification of the recursive functions. *Z. Math. Logik Grundlagen Math.* **16**, 71–82 (1972).

Meyer, A. R., and Stockmeyer, L., The equivalence problem for regular expressions with squaring requires exponential space. *13th IEEE Computer Society Symp. Switching and Automata Theory, 1972*, 125–129.

Paterson, M. S., Tape bounds for time-bounded Turing machines. *J. Computer and System Sciences* **6** 2, 116–124 (1972).

Pippenger, N., and Fischer, M. J., Relationships among complexity measures. Manuscript, IBM Watson Research Center, Yorktown Heights, New York, 1977 (to appear).

Pratt, V., and Stockmeyer, L., A characterization of the power of vector machines. *J. Computer and System Sciences* **12** 2, 198–221 (1976).

Ritchie, R. W., Classes of predictably computable functions. *Trans. Amer. Math. Soc.* **106**, 139–173 (1963).

Schnorr, C. P., The network complexity and the Turing machine complexity of finite functions. Manuscript, Fachbereich Mathematik, Univ. of Frankfurt, 1975.

Stockmeyer, L., The polynomial-time hierarchy. *Theoretical Computer Science* **3** 1, 1–23 (1976).

Valiant, L. G., General context-free recognition in less than cubic time. *J. Computer and System Sciences* **10** 2, 308–315 (1975).

van Leeuwen, J., Variants of a new machine model. *Proc. 17th IEEE Symp. Foundations of Computer Science 1976*, 228–235.