Characterizations of Pushdown Machines in Terms of Time-Bounded Computers

STEPHEN A. COOK*

University of California,† *Berkeley, California*

ABSTRACT. A class of machines called *auxiliary pushdown machines* is introduced. Several types of pushdown automata, including stack automata, are characterized in terms of these machines. The computing power of each class of machines in question is characterized in terms of time-bounded Turing machines, and corollaries are derived which answer some open questions in the field.

KEY WORDS AND PHRASES: Turing machines, multitape Turing machines, time-bounded computers, abstract computer models, pushdown automata, multihead pushdown automata, stack automata, writing pushdown acceptors, auxiliary pushdown machines, computational complexity

CR CATEGORIES: 5.22, 5.23

1. *Introduction*

In this paper we have two main purposes: first, to provide a unified treatment of several types of two-way pushdown automata and stack automata, and, second, to give an interesting characterization of the computing power of each of these machines in terms of deterministic time-bounded Turing machines.

In Section 2 we introduce the notion of *time-bounded computer*. This is simply a general computer model used to distract attention from picky arguments about Turing machine tapes and heads. In Section 3 we introduce the notion of *auxiliary pushdown machine*, which is like a tape-bounded Turing machine but has a pushdown store attached whose storage does not count in the tape bound. The main theorem, in Section 4, characterizes the computing power of auxiliary pushdown machines for tape bounds $L(n) \geq \log_2 n$ in terms of time-bounded computers, and states that for any tape bound $L(n) \geq \log_2 n$ the deterministic and nondeterministic versions have the same computing power. Here n is the length of the input string.

Theorem 2 in Section 4 states that the computing power of two-way multihead pushdown machines, writing pushdown acceptors, and two-way stack automata can be characterized in terms of auxiliary pushdown machines, and hence in terms of time-bounded computers. In particular, a deterministic two-way stack automaton is equivalent to a deterministic n^{cn} time-bounded Turing machine, and a nondeterministic two-way stack automaton is equivalent to a deterministic 2^{cn^2} time-bounded Turing machine. A number of corollaries are derived from Theorems 1 and 2, some of which answer open questions in the literature.

* Present address: Department of Computer Science, University of Toronto, Toronto, Ontario, Canada.

† Department of Mathematics. The research reported here was supported in part by Office of Naval Research Contract Nonr 3656 (23) and by the National Science Foundation Contract GJ474. Most of the results in this paper were announced in [4].

2. *Time-Bounded Computers*

Turing machines of various kinds are the most common abstract computer model used in the theory of computational complexity. Yet they are often criticized as models of real computers, since they do not behave much like random-access machines. In fact, some of the results in the literature showing Turing machines require a great deal of time to recognize simple sets of strings, surely depend on exploiting the inefficient storage arrangement of a single head on a single tape, and do not reflect any fundamental property of computation. Many other results, however, hold as well for any reasonable computer model as they do for Turing machines.

The results in this paper are of the latter kind. In order to emphasize this point we shall quote our results in terms of "time-bounded computers" instead of Turing machines. Informally, a time-bounded computer is any device equipped to accept a set A of strings within a certain time bound $T(n)$, provided some Turing machine accepts A within time $(T(n))^k$ for some k. Our formal definition will not characterize the possible such devices, but just the notion of acceptance within a time bound.

Definition. Let $T(n)$ be a function from positive integers to positive integers, and let A be a set of strings over a finite alphabet Γ. Then we say A *is accepted by a time-bounded computer within time* $T(n)$ provided some simple deterministic Turing machine M (see Section 3) accepts A within time $(T(n))^k$, for some constant k.

Examples of time-bounded computers are simple Turing machines, multitape Turing machines, iterative arrays of finite-state machines [3], Shepherdson-Sturgis machines [14], and abstract random access machines in the sense of Earley [5, p. 102]. The proofs in each case are straightforward; see, for example, [9] for the case of multitape Turing machines.

The motivation for this notion of time-bounded computer rests on a feeling that (1) any reasonable computer model should be at least as strong as a simple Turing machine [i.e. to demonstrate conclusively that a task can be performed within time $T(n)$ it is more than sufficient to show that a simple Turing machine can do so], (2) as stated above a large class of computer models is included in the notion of time-bounded computer—in particular, a plausible formal notion of random-access machine—and (3) at the present state of the art it is difficult to find a convincing more restricted definition of computer.

In Section 5 we refer to the class $\mathcal{L}_*$ of sets of strings, first defined by Cobham [2] in terms of numerical functions instead of sets of strings.

Definition. A set A of strings is in the class $\mathcal{L}_*$ if and only if A is accepted by a time-bounded computer within time $P(n)$, for some polynomial $P(n)$.

The class $\mathcal{L}_*$ is the smallest class which can be characterized (in a reasonable way) in terms of time-bounded computers. As pointed out by Cobham [2], the class is unchanged if the word "computer" in the definition is replaced by any of "random-access machine," "Turing machine," "Shepherdson-Sturgis machine," or "iterative array of finite-state machines." It will turn out (see Section 5) that $\mathcal{L}_*$ has an interesting characterization in terms of multihead pushdown machines.

3. *Other Machine Models*

By a *simple Turing machine* we mean the familiar device consisting of a finite-state control attached to a single read/write head moving along a single two-way infinite tape. In one step, the machine can assume a new state, print one of a finite set of symbols on the tape square currently scanned, and shift its head either left or right one square, or leave it stationary. The action of the machine in a step depends on the current state and tape symbol being scanned. By a *multitape Turing machine* we mean a finite-state control attached to both a two-way read-only *input head* moving along an *input tape*, and finitely many read/write *work* heads each moving along a distinct two-way infinite *work tape*. In one step, the machine can assume a new state, print one of a finite set of symbols (excluding the blank symbol) on each of its work tapes (but not its input tape), and shift some of its heads left or right one square in any combination. The action taken depends on the current state of the machine and the symbols scanned by each of its heads.

Before any computation a finite input string is placed on the input tape delimited by blanks at each end. The finite-state control is designeo so that the input head will never leave the segment consisting of this string and the two blanks during any computation.

An *auxiliary pushdown machine* (auxiliary PDM) is a multitape Turing machine which has an extra tape called the *pushdown tape*, which operates in a special fashion. The pushdown alphabet has a distinguished symbol s_0 which appears initially on the pushdown tape. The machine is designed so that the pushdown head never shifts left of s_0 or changes s_0. Further, the pushdown head can never shift left when scanning any tape symbol unless it first erases (i.e. overwrites a blank on) that symbol, and it can never shift right from a square unless it first prints a nonblank symbol on that square. Initially the pushdown head is scanning the symbol s_0 and all other squares on the pushdown tape are blank. Thus throughout the computation all squares on the pushdown tape are blank except for the segment bounded on the left by s_0 and on the right by the pushdown head, which is nonblank.

The last type of machine we describe here is the two-way stack automaton, introduced in [6]. For our purposes, a two-way stack automaton, or briefly *stack automaton*, consists of a finite-state control attached to an *input tape* and a *stack tape*. The input tape is just like that for multitape-Turing machines: it is of the two-way read-only type, with input delimited by blanks. The stack tape is like the pushdown tape for an auxiliary PDM, except the stack head is allowed to read the information on the stack between the symbol s_0 and the rightmost nonblank symbol. At the beginning of any computation, the stack tape is blank except for one square containing the symbol s_0, and the stack head scans this square. The input tape is initially blank, except for the input string w, and the input head scans the leftmost symbol of w. In one step, the machine will, depending on its internal state and the symbols scanned by the input head and stack head, assume a new state, shift its input head one or zero squares left or right (but not outside the blanks delimiting the input), and shift the stack head one or zero squares left or right, but not left of the symbol s_0. In addition, the stack head may print a symbol before shifting, provided either it is scanning the *leftmost* blank symbol to the right of s_0 on the stack tape or if on the preceding step the stack head printed a blank and shifted left, and in addition provided the scanned symbol is not s_0 and provided the stack head does not print a blank and shift right. These provisions are implemented by the design of the finite-state control, and by use of an enlarged symbol alphabet for the stack tape.

We shall sometimes allow auxiliary PDM's and stack automata to be nondeterministic, although Turing machines are here always assumed to be deterministic, unless the contrary is explicitly stated.

Each of the above types of machines has certain distinguished states called *accepting* states, and a single distinguished state q_0 called the initial state. Let M be one of the above machines (deterministic or not), and let Γ be a (finite) subset of the set of symbols which M can read on its input tape (here we will refer to the single tape of a simple Turing machine as the input tape), and suppose Γ does not contain the blank symbol. Then we say M *accepts* a string w on Γ provided that some computation of M terminates in an accepting state, where initially M is in the state q_0, and all tapes are blank except the input tape contains the string w with the input head scanning the leftmost symbol of w (and the pushdown or stack tape contains the symbol s_0). For the case of auxiliary PDM's we shall require in addition that the pushdown head scan the symbol s_0 (i.e. the pushdown list is empty) at the end of the accepting computation.

Suppose $T(n)$ and $L(n)$ are functions on the positive integers, and suppose A is a set of strings on Γ. Then we say a machine M *accepts* the set A *within time* $T(n)$ provided, for each $w \in A$, M accepts w in some computation consisting of $T(|w|)$ or fewer steps, where $|w|$ is the length of w, and provided M accepts no strings w not in A. We say a multitape Turing machine or auxiliary PDM *accepts* A *within storage* $L(n)$ provided, for each $w \in A$, M accepts w in some computation in which no work tape (excluding the input tape and pushdown tape) scans more than $L(|w|)$ distinct squares, and provided M accepts no strings w not in A.

4. The Main Theorem

THEOREM 1 (*Main theorem*). *The following three conditions are equivalent for any set A of strings on an alphabet* Γ *and for any function* $L(n) \geq \log_2 n$ *on the positive integers.*

(*a*) *A is accepted by some deterministic auxiliary PDM within storage* $L(n)$.

(*b*) *A is accepted by some nondeterministic auxiliary PDM within storage* $L(n)$.

(*c*) *A is accepted by some time-bounded computer within time* $T(n) = 2^{cL(n)}$ *for some constant c.*

PROOF. (a) $\Rightarrow$ (b). Obvious.

(b) $\Rightarrow$ (c). The argument is a generalization of one appearing in [1]. Suppose M_1 is a nondeterministic auxiliary PDM which accepts the set A within storage $L(n)$. We will construct a deterministic multitape Turing machine M_2 which accepts A within time $T(n) = 2^{cL(n)}$ for some constant c.

Fix a string w on the input alphabet Γ, let $n = |w|$ (the length of w), and suppose M_1 has k work tapes in addition to the input tape and pushdown tape. Then a *configuration* of M_1 with input w is a string $P = pqu_1{\downarrow}v_1{*}u_2{\downarrow}v_2{*} \cdots {*}u_k{\downarrow}v_k{*}s$, where p is the dyadic notation for an integer between 0 and $n + 1$, q is a state of M_1, $u_1, \cdots, u_k$ and $v_1, \cdots, v_k$ are strings on the work tape alphabets, s is a symbol on the pushdown alphabet, and $\downarrow$ and $*$ are new symbols. We say M_1 with input w is in configuration P provided M_1 has the string w delimited by blanks written on its input tape, the input head is scanning symbol number p from the left (counting the blank immediately to the left of w as symbol number 0), M_1 is in state q, the k work tapes have the strings $u_1v_1, u_2v_2, \cdots, u_kv_k$ written on them with work head i scanning the first symbol of v_i ($i = 1, 2, \cdots, k$), and the symbol s is currently scanned by the pushdown head (i.e. on top of the pushdown list). Thus a configuration completely specifies an instantaneous description of M_1, except the input string w and the contents of the pushdown tape (other than the currently scanned symbol) are left unspecified.

We say the pair (P, Q) of configurations of M_1 with input w is *realizable* provided there is some partial computation of M_1 with input w such that at the beginning of the partial computation M_1 is in configuration P with the pushdown head scanning some square c, and at the end of the partial computation M_1 is in configuration Q with the pushdown head scanning the same square c (although c need not have the same symbol written on it) and throughout the partial computation the pushdown head never moves to the left of c. If we let P_0 correspond to the initial configuration, so that $P_0 = 1q_0{\downarrow}\beta * \cdots *{\downarrow}\beta s_0$ (where β represents the blank symbol), then M_1 accepts the input w if and only if there is some configuration Q_a, whose state symbol is an accepting state, such that the pair (P_0, Q_a) is realizable. We shall call such a pair (P_0, Q_a) an *accepting pair.*

We shall design the Turing machine M_2 to build a list of realizable pairs (P, Q) on one of its work tapes. If ever M_2 finds an accepting pair (P_0, Q_a), it will halt and accept the input string w. Otherwise M_2 computes forever.

Since the machine M_1 is supposed to operate within storage $L(n)$, M_2 need only consider configurations $P = pqu_1{\downarrow}v_1 * \cdots *u_k{\downarrow}v_k*s$ whose work tape strings u_iv_i do not exceed $L(n)$ in length, where $n = |w|$. However, the function L may be difficult or impossible to compute, so rather than calculate the value $L(n)$, M_2 uses a parameter l, stored on one of its work tapes, to guess at $L(n)$. Initially $l = 1$. In general, after M_2 has found all realizable pairs (P, Q) whose work tape storage does not exceed l, and if no accepting pair (P_0, Q_a) has been found, then l is increased by one. The process continues indefinitely.

The list of realizable pairs (P, Q) formed by M_2 is initially empty. In general, after all realizable pairs have been found for a particular value of l, l is increased by one and the list is increased as follows. First, all pairs (P, P) of configurations whose work storage is equal to l are added to the list. Next, for every two pairs (P_1, Q_1), (P_2, Q_2) appearing on the list, each pair (P_3, Q_3) is added to the list such that (1) the pairs (P_1, Q_1), (P_2, Q_2) "yield" the pair (P_3, Q_3) in the sense defined below, (2) configurations P_3, Q_3 each have work storage at most l, and (3) the pair (P_3, Q_3) does not already appear on the list. This last step is repeated until either an accepting pair (P_0, Q_a) is found, in which case M_2 accepts w, or no new realizable pairs can be found, in which case l is increased by one and the process starts all over again.

Definition. The two pairs (P_1, Q_1) and (P_2, Q_2) of configurations are said to *yield* the pair (P_3, Q_3) provided $P_1 = P_3$, and either (i) $Q_1 = P_2$ and M_1 can go from configuration Q_2 to Q_3 in one step without shifting its pushdown head, or (ii) M_1 can go from Q_1 to P_2 in one step by printing some symbol s on its pushdown tape and shifting its pushdown head right, M_1 can go from Q_2 to Q_3 in one step by shifting its pushdown head left, and when M_1 is in configuration Q_3 its pushdown head scans this symbol s.

Clearly if (P_1, Q_1) and (P_2, Q_2) are realizable pairs which yield (P_3, Q_3), then (P_3, Q_3) is realizable. Conversely, the following lemma shows that the procedure outlined above for M_2 will eventually produce all realizable pairs.

LEMMA 1. *Every realizable pair (P, Q) of configurations of work tape storage l or* *less can be obtained from pairs of the form (P, P) by successively applying the yield relation, where all configurations appearing have work tape storage l or less.*

PROOF. Suppose the pair (P, Q) is realizable and the work storage of the configurations P, Q does not exceed l. Then there is some partial computation of M_1 with input w whose initial configuration is P, whose final configuration is Q, and such that the pushdown head satisfies the restrictions in the definition of "realizable." Let $P = P_1, P_2, \cdots, P_t = Q$ be the configurations of the successive steps in the computation. Then the work storage for none of the P_i can exceed l, since the storage cannot shrink during a computation. We shall prove by induction on t that (P, Q) can be obtained as stated in the lemma. If $t = 1$, then (P, Q) is one of the initial pairs (P, P). Suppose $t > 1$. There are two cases:

(i) Suppose the pushdown head remains stationary in the step P_{t-1}, P_t. Then the pair (P_1, P_{t-1}) is realizable, and by the induction hypothesis it can be obtained as stated in the lemma. But (P_1, P_1) and (P_1, P_{t-1}) yield (P_1, P_t). Therefore $(P_1, P_t) = (P, Q)$ can be obtained as stated in the lemma.

(ii) Suppose the pushdown head shifts left in the step P_{t-1}, P_t (it cannot shift right). Let P_i be the first configuration in the computation such that the pushdown head shifts right in the step P_i, P_{i+1}. Then both the pairs (P_1, P_i) and (P_{i+1}, P_{t-1}) are realizable, and by the induction hypothesis they can be obtained as stated in the lemma. But the pairs (P_1, P_i) and (P_{i+1}, P_{t-1}) yield (P_1, P_t). Therefore $(P_1, P_t) = (P, Q)$ can be obtained as stated in the lemma.

It remains to estimate the time required by M_2 to carry out the simulation. Clearly there is some constant c_1 (independent of w and $n = |w|$) such that for each value of $l \geq \log_2 n$ [recall $L(n) \geq \log_2 n$], M_2 requires at most $c_1 l + c_1$ squares to represent any pair (P, Q) of configurations on its work tape, provided the work storage indicated in P, Q does not exceed l. Thus there is some constant γ such that the total number of possible configuration pairs of work storage not exceeding l is bounded by $\gamma^{c_1 l + c_1}$. Hence the number of realizable pairs of configurations appearing in the list on M_2's work tape never exceeds $\gamma^{c_1 l + c_1}$. Since the time required to add one pair to the list will not exceed a constant times, say, the fourth power of this bound, and since if M_1 accepts w [within storage $L(n)$] then M_2 will accept w for some $l \leq L(n)$, it is clear that there is some constant c such that if M_1 accepts w, then M_2 will accept w within $2^{cL(n)}$ steps.

(c) $\Rightarrow$ (a). Suppose some time-bounded computer accepts the set A within time $T(n) = 2^{cL(n)}$. Then, by definition of acceptance by a time-bounded computer, there is a simple (deterministic) Turing machine S and a constant d such that S accepts A within time $2^{dL(n)}$. We first modify S to form a simple Turing machine M_1 which accepts A within time $2^{c_1 L(n)}$ for some constant c_1, such that the head of M_1 operates in a regular predictable pattern as follows. Suppose initially M_1 has the input w of length n on its tape, delimited by blanks, with the head scanning the leftmost symbol of w. Throughout its computation, M_1 prints only nonblank symbols. First the head shifts right for n successive steps until reaching the first blank square, immediately to the right of w. Then the head prints a nonblank symbol and shifts left for $n + 1$ successive steps until reaching the first blank square, immediately to the left of w. The head prints a nonblank symbol, shifts right for $n + 2$ steps, then left for $n + 3$ steps, and so on. Meanwhile M_1 is simulating S by carrying out one or more print operations of S's computation for each sweep of M_1's head (except possibly the first). On the first sweep right, M_1 simulates S as long as S continues to shift right at the rate of one shift per step. Then M_1 marks that square, and continues the simulation when its head sweeps back left. The details of the simulation are left to the reader. The important thing is that M_1 requires at most one sweep, and hence at most $T(n) = 2^{dL(n)}$ steps, for each step of S, and hence if S accepts w, then M_1 accepts w within $[\mathrm{T}(n)]^2 = 2^{2dL(n)}$ steps.

We shall now construct a deterministic auxiliary PDM M_2 which indirectly simulates M_1. Let us fix an input string w of length n, and consider triples $\langle t, q, s\rangle$, where t is a nonnegative integer, q is a state of M_1, and s is a symbol in the tape alphabet of M_1. Such a triple is said to be *realizable* provided that, at step number t of the computation of M_1 with input w, M_1 is in state q scanning the symbol s. The simulating machine M_2, with input w, will operate by making a list of coded forms of realizable triples $\langle t, q, s\rangle$ on its pushdown tape, always searching for one such that q is an accepting state. Note that M_1 accepts w if and only if there is some realizable triple $\langle t, q, s\rangle$ such that q is an accepting state.

Given a realizable triple $\langle t, q, s\rangle$, the immediate task of M_2 is to calculate the realizable triple $\langle t + 1, q', s'\rangle$, which describes M_1 at step $t + 1$ of its computation. The state q' is easily determined by applying M_1's state transition function to the pair (q, s). However, in order to determine s', usually M_2 must have access to a realizable triple $\langle t_1, q_1, s_1\rangle$ describing M_1 the last time it scanned the square it scans at step $t + 1$. We make the relationship among these three triples precise as follows:

Definition. The triples $\langle t, q, s\rangle$ and $\langle t_1, q_1, s_1\rangle$ are said to *yield* the triple $\langle t + 1, q', s'\rangle$ provided that when M_1 is in state q scanning the symbol s, it next assumes the state q', and either (i) M_1 scans a square for the first time at time $t + 1$, in which case s' is the symbol originally occupied by that square (either blank or a symbol of w), or (ii) t_1 is the greatest integer less than $t + 1$ such that M_1 scans the same square at steps t_1 and $t + 1$, in which case s' is the symbol printed by M_1 when in state q_1 scanning the symbol s_1.

It should be clear that if this yield relation holds, and if both the triples $\langle t, q, s\rangle$ and $\langle t_1, q_1, s_1\rangle$ are realizable, then $\langle t + 1, q', s'\rangle$ is realizable. Furthermore, the question of whether or not the three triples satisfy the yield relation can be easily answered by M_1, because of the predictable pattern described by M_1's head. This point will be discussed later.

We can describe M_2's operation informally as follows. We assume M_2 has enough work tape space available to store, say, at least four triples. Thus M_2 has no trouble in calculating realizable triples $\langle t, q, s\rangle$ for $t \leq n$, i.e. triples describing the first sweep of M_1's head. Now let us assume as an informal induction hypothesis that for some fixed t, and every $\tau \leq t$, there will be a point in M_2's computation at which the realizable triple $\langle \tau, q_\tau, s_\tau\rangle$, which describes M_1 at time τ, will appear alone on M_2's pushdown tape. Suppose M_2 has just written the realizable triple $\langle t, q, s\rangle$ on its pushdown tape, which is otherwise empty. Let $\langle t_1, q_1, s_1\rangle$ be a realizable triple such that $\langle t, q, s\rangle$ and $\langle t_1, q_1, s_1\rangle$ yield $\langle t + 1, q', s'\rangle$. M_2 will proceed by repeating its entire computation to date, except now $\langle t, q, s\rangle$ will be treated as the bottom (left end) of the pushdown tape, and $\langle t, q, s\rangle$ will remain undisturbed. By our induction hypothesis, the triple $\langle t_1, q_1, s_1\rangle$ will appear immediately to the right of $\langle t, q, s\rangle$ on the pushdown tape at some point in the computation. M_2 will constantly check (using work tape storage) to see whether the rightmost two triples on its pushdown tape can be combined to yield a third triple, and if this is possible, M

will replace the two triples by the third triple. Thus, at some point in the computation, the realizable triple $\langle t+1, q', s'\rangle$ will appear alone on the pushdown tape, and we have carried the induction one step farther. Of course we must supply a program for M_2 before the argument can be made precise. This is done below.

We note that in the course of computing the triple $\langle t, q, s\rangle$ and placing it at the left end of its pushdown tape, M_2 will develop a list of triples on its pushdown tape which, at a maximum, contains as many triples plus one as the head of M_1 makes sweeps up to step t.

We now give the precise algorithm which M_2 follows. We assume that throughout its computation M_2 has a list of triples on its pushdown tape. (The value of t in $\langle t, q, s\rangle$ is stored in binary notation.) Initially this list is empty. The *top* of the list means the rightmost entry on the list. Of course throughout the computation M_2 has access to the input string w.

ALGORITHM FOR M_2

P1 (ADD FIRST TRIPLE). Add the triple $\langle 0, q_0, \bar{s}\rangle$ to the top of the pushdown list, where q_0 is the initial state of M_1 and $\bar{s}$ is the leftmost symbol of w.

P2 (ACCEPT?). If the top triple is $\langle t, q, s\rangle$, where q is an accepting state of M_1, then ACCEPT the input w. Otherwise, go to P3.

P3 (APPLY YIELD RELATION). If the list has at least two triples, and x and y are the top two triples, and x and y yield a triple z, then remove x and y from the list and add z, and go to P2. Otherwise go to P1.

To prove the algorithm works (i.e. causes M_2 to accept the input w if and only if M_1 accepts w), one need only formalize the informal argument given above. Notice that initially P1 is executed twice before P3 succeeds, so that two copies of $\langle 0, q_0, \bar{s}\rangle$ appear on the pushdown list.

It remains to estimate the auxiliary storage (i.e. work tape storage) required by M_2. We shall confine our remarks to step P3, since it is clearly the most difficult.

The key to executing step P3 is computing the function $\pi(n, t)$, whose value is the position of the head of M_1 at step number t of a computation with an input string of length n. The position is an integer assigned to the square scanned at step t. Integers are assigned to squares on the tape by assigning 0 to the square containing the leftmost symbol of the input w (in its position at the start of the computation), consecutive positive integers to squares to the right of square 0, and consecutive negative integers to squares to the left of square 0. Assuming the argument t is available on a work tape in binary notation, and n is the length of the input string, M_2 computes $\pi(n, t)$ by simulating on a work tape the head motion of M_1 for t steps, keeping track of M_1's head position during the simulation by a parameter p written in binary notation. Thus the number of work tape squares required to compute $\pi(n, t)$ is at most $d_1 \max(\log_2 n, \log_2 t)$ for some small constant d_1. If M_1 accepts w then it does so within $2^{c_1 L(n)}$ steps. Hence values of t considered by M_2 will never exceed $2^{c_1 L(n)}$, so that the storage used is at most $d_1 \max(\log_2 n, c_1 L(n))$ and, hence, at most $d_2 L(n)$ for some constant d_2, since $L(n) \geq \log_2 n$. If M_1 fails to accept w, we are not concerned with the storage required by M_2.

To execute step P3, M_2 copies the top two triples, if they exist, from the pushdown tape (where they will be destroyed) to a work tape. If the pushdown tape contains only one triple, M_2 restores that triple to the pushdown tape and passes control to step P1. Otherwise, M_2 determines whether or not the top two triples yield a triple z. This is easily done by computing $\pi(n, t + 1)$, and $\pi(n, \tau)$ for successive values of $\tau \leq t$, and by referring to the transition function for M_1 and referring to the input w. If the triple z can be found, it is added to the top of the pushdown list and control is passed to step P2. Otherwise, the top two triples are restored to the pushdown tape and control is passed to step P1.

As mentioned above, if M_1 accepts w, then the value of t in the triples $\langle t, q, s\rangle$ considered by M_2 never exceeds $2^{c_1 L(n)}$, and hence the space required to store a triple is bounded by $d_3 L(n) + d_3$ for some constant d_3. From this it is clear from the above discussion that the auxiliary storage required by M_2, in case M_1 (and hence M_2) accepts w, is at most $d_4 L(n) + d_5$, for some constants d_4, d_5. By using the standard techniques of increasing the work tape alphabet and finite-state control for M_2, the constants d_4 and d_5 can be reduced to 1 and 0. Thus in fact M_2 can be made to accept the set A within storage $L(n)$.

5. *Applications of the Main Theorem*

The theorem below gives characterizations of the computing power of several types of pushdown machines in terms of time-bounded computers. In addition to the pushdown devices described in Section 3, the theorem mentions two others: the writing pushdown acceptors of Mager [13] and the two-way multihead pushdown automata discussed in [8]. A *writing pushdown acceptor* was defined in [13] to be a nondeterministic linearly bounded automaton equipped with a pushdown store, but this is easily seen to be equivalent in computing power to a nondeterministic auxiliary PDM with storage bounded by $L(n) = n$. A multihead two-way pushdown automaton consists of a nondeterministic finite-state control attached to a pushdown store, and to several two-way read-only input heads operating on an input tape with endmarkers.

THEOREM 2. *A set A of strings is accepted by a machine of type i $(1 \leq i \leq 6)$ in the first column of Table I if and only if A is accepted by some auxiliary PDM (deterministic or not) within storage $L(n)$ given in row i of the second column, and again if and only if A is accepted by a time-bounded computer within time $T(n)$ given in row i of the third column.*

PROOF. The equivalences indicated between columns 2 and 3 of the table follow directly from Theorem 1. Of the equivalences between columns 1 and 2, those in rows 3 and 4 follow easily from Mager's definition of writing pushdown acceptor in the light of Theorem 1, which guarantees that the deterministic and nondeterministic versions of auxiliary PDM's bounded by storage $L(n) = n$ have the

TABLE I

Machine type	*$L(n)$ for auxiliary PDM*	*$T(n)$ for deterministic Turing machine*
1. Two-way multihead pushdown automaton	$\log n$	n^c, constant c
2. Deterministic two-way multihead pushdown automaton	$\log n$	n^c, constant c
3. Writing pushdown acceptor	n	2^{cn}, constant c
4. Deterministic writing pushdown acceptor	n	2^{cn}, constant c
5. Deterministic two-way stack automaton	$n \log n$	n^{cn}, constant c
6. Nondeterministic two-way stack automaton	n^2	2^{cn^2}, constant c

same computing power. The equivalence between two-way multihead pushdown automata and $(\log n)$-bounded auxiliary PDM's follows from unpublished but fairly well-known proof techniques by Alan Cobham and others showing that two-way multihead finite-state machines are equivalent to $(\log n)$ tape-bounded multitape Turing machines.

The equivalences between stack automata and auxiliary PDM's remain to be demonstrated. These will follow immediately from the next three lemmas.

LEMMA 2. *If a set A of strings can be recognized by a deterministic [nondeterministic] stack automaton, then A can be recognized by an auxiliary PDM within storage $n \log n$ [storage n^2].*

PROOF. The argument is similar to the one used by Hopcroft and Ullman [11] to simulate nonerasing stack automata by tape-bounded Turing machines. Suppose S is an (erasing) stack automaton (deterministic or not) and let w be its input. At any point in the computation of S with input w the nonblank portion of the stack tape will be a string y. Following [11], we associate with each w and y a transition matrix $M_{w,y}$ defined as follows.

Let n be the length of the input w, and let N be the set of integers between 0 and $n + 1$. Then $M_{w,y}$ is a binary relation on $(Q \times N) \cup \{A\}$. We say $M_{w,y}$ holds between (p, i) and (q, j) provided there is some partial computation of S in which initially S is in state p with the input head scanning symbol number i of the input w (counting symbol 0 as the blank to the left of w and symbol $n + 1$ as the blank to the right of w) with y the nonblank portion of the stack tape and the stack head scanning the rightmost symbol of y, and finally the input head is scanning the position j of w and the stack head is scanning the blank immediately to the right of y. Further, throughout the partial computation the string y must remain unchanged, and except for the last step, the stack head must never leave the string y.

We say $M_{w,y}$ holds between (p, i) and A if there is some partial computation of S which satisfies the same initial conditions as above, but ends in S accepting w before the stack head leaves y.

We construct an auxiliary PDM X to simulate S as follows. X will be deterministic if S is deterministic, and sometimes nondeterministic if S is nondeterministic. Suppose the stack automaton S is in state q with input head scanning symbol i of the input w, and let $y = Y_1Y_2 \cdots Y_k$ be the nonblank portion of the stack tape. If the stack head is scanning the leftmost blank on the stack tape (to the right of y), we say S is in a *regular configuration*. This regular configuration is represented in X by the pair $\langle q, i\rangle$ on a work tape and by the information Y_1M_{w,Y_1}, Y_2, M_{w,Y_1Y_2}, $\cdots$, Y_k, $M_{w,y}$ on the pushdown tape (where the transition matrices $M_{w,Y_1\cdots Y_j}$ are specified in some suitable notation).

Given a regular configuration occurring in a computation of S, X proceeds by finding the next regular configuration in the computation; or, in case S is nondeterministic, X will nondeterministically choose a possible next regular configuration for S.

Before S next assumes a regular configuration, the nonblank portion y of the stack tape may either (1) remain unchanged; or be changed in any of the following ways: (2) a new (nonblank) symbol Y_{k+1} is added to the right of y, resulting in yY_{k+1}, (3) the right symbol of y is altered to a new nonblank symbol Y_k', resulting in $Y_1Y_2 \cdots Y_{k-1}Y_k'$, or (4) the right symbol of y is erased, resulting in $Y_1Y_2 \cdots Y_{k-1}$.

In case (2), X must calculate the transition matrix $M_{w,yY_{k+1}}$ and add it to th right of its pushdown tape. X can easily calculate the new transition matrix fro the symbol Y_{k+1} and the transition matrix $M_{w,y}$, which appears on the pushdow tape. The method is described in detail in [11]. In case (3), X must calculat $M_{w,Y_1 \cdots Y_k'}$. This new transition matrix is calculated from Y_k' and $M_{w,Y_1 \cdots Y_{k-}}$ (which appears on the pushdown tape immediately to the left of Y_k, $M_{w,y}$) usin exactly the same method as for case (2). Finally, in case (4), X need only delet the pair Y_k, $M_{w,y}$ from its pushdown tape and update the pair $\langle q, i\rangle$ on one of it work tapes.

In any of the cases (1), (2), or (3), S may shift its stack head to the left of th rightmost nonblank symbol on its stack tape at some point before reaching the nex regular configuration. Then, by the restrictions placed on the stack head in th definition in Section 3, the stack head cannot further change any symbols on th stack tape before reaching a regular configuration. Hence X can find the next state input position pair $\langle q, i\rangle$ (or the possible next pairs in the nondeterministic case of S by referring to the transition matrix on the right of its pushdown tape. O course X can also tell whether S can accept the input before the next regular con figuration, and if S can accept, then X accepts.

The auxiliary (i.e. work tape) storage required by X for the simulation is a con stant multiple of the number of squares required to store the largest transitio matrix. In case S is deterministic, for each transition matrix $M_{w,y}$ and each pai (p, i) there is at most one possible pair (q, i) standing in the relation $M_{w,y}$ to (p, i) Hence the storage required to write down all pairs $\langle (p, i), (q, j)\rangle$ or $\langle (p, i), A$ satisfying $M_{w,y}$ is at most $cn \log n$ for some constant c, where n is the length of th input w. (The integers i and j are stored in binary notation.) In case S is non deterministic, the possibilities for $M_{w,y}$ are increased. However, $M_{w,y}$ is always subset of $(Q \times N) \times ((Q \times N) \cup \{A\})$, a set of at most $c_1 n^2$ elements, where th constant c_1 depends only on the cardinality of the state set Q. Since a set of $c_1 n$ elements has $2^{c_1 n^2}$ subsets, an arbitrary subset, and hence any transition matri $M_{w,y}$, can be specified by using $c_1 n^2$ tape squares, using any efficient notation. De tails can be found in [11]. Also, the method for calculating the transition matri $M_{w,yY}$ from $M_{w,y}$ and Y using at most the storage $c_2 n \log n$ (if S is deterministic or $c_2 n^2$ (if S is nondeterministic) is described in [11]. The constants c, c_1, and can be reduced to unity by standard methods.

LEMMA 3. *If a set A of strings is accepted by some deterministic auxiliary PD within storage $L(n) = n^2$, A is accepted by some nondeterministic stack automaton.*

PROOF. The argument uses the same techniques as are used in [11] to show nondeterministic nonerasing stack automaton can simulate a nondeterministi n^2 tape-bounded Turing machine.

Let X be a deterministic auxiliary PDM which accepts A within storage $L(n) =$ n^2. Let us fix an input w for X. We will abbreviate the notation for configuration given in Section 4 in the proof (b) $\Rightarrow$ (c), and write $P = \langle q, i, u, Y\rangle$ instead o $P = iqu_1{\downarrow}v_1{*}u_2{\downarrow}v_2{*} \cdots {*}u_k{\downarrow}v_k{*}Y$, where u stands for $u_1{\downarrow}v_1{*} \cdots {*}u_k{\downarrow}v_k$. Thus is in the configuration $\langle q, i, u, Y\rangle$ provided X is in state q, with input head scannin symbol number i of w, with work tapes described by u, and pushdown head scannin the symbol Y.

We now show how a nondeterministic stack automaton S simulates X. Suppos at some point in its computation X is in configuration $\langle q, i, u, Y_k\rangle$, with $y = Y_1$

$\cdots Y_k$ equal to the contents of its pushdown tape (i.e. the string of symbols between s_0 on the left and the pushdown head on the right, including both ends). For each symbol Y_j on the pushdown tape, let $\phi(j)$ be the configuration of X the last time in the computation to date that the square on which Y_j is printed was scanned. Thus, regardless of the values of $Y_1, Y_2, \cdots, Y_{j-1}$, if $j < k$, then we know that when X is in the configuration $\phi(j)$, it will print Y_j on the pushdown tape and compute with its pushdown head to the right of this Y_j until at some point it is in the configuration $\langle q, i, u, Y_k \rangle$, with the symbols $Y_{j+1} \cdots Y_k$ written to the right of Y_j on the pushdown tape. Further, $\phi(k) = \langle q, i, u, Y_k \rangle$.

The stack tape of S encodes the computation of X to date as follows. The stack tape is divided into an upper channel and a lower channel. On the upper channel is written the sequence of configurations $\phi(1), \phi(2), \cdots, \phi(k)$, with a certain amount of "garbage" between adjacent entries. The garbage consists of obsolete configurations which are labeled with $*$'s and ignored by the machine. On the lower channel below each $\phi(j)$ is a configuration $\psi(j)$ which represents a guess by S as to the configuration of X the next time X scans square j of its pushdown tape.

We now describe how S updates its stack tape for each of the possible steps that X can take. Notice that S can determine what action X will next take by examining (without destroying) the top configuration $\phi(k)$ in its stack. The index i in the triple $\langle q, i, u, Y_k \rangle$ is stored in unary notation (i strokes) so that S can determine which symbol of the input string X is scanning.

(1) Suppose X prints a symbol on its pushdown tape and shifts its pushdown head right (and does various operations with its work heads and input head). Then S will compute the new configuration $\phi(k + 1) = \langle q_1, i_1, u_1, \beta \rangle$ and write it on the upper channel of its stack tape to the right of $\phi(k)$. The method used by S to produce $\phi(k + 1)$ to the right of $\phi(k)$ is exactly the one described in [11] that enables a nondeterministic nonerasing stack automaton to write the next instantaneous description of an n^2 tape-bounded Turing machine next to the old one. While S is producing $\phi(k + 1)$, it simultaneously writes on the lower channel below $\phi(k + 1)$ an arbitrary configuration $\psi(k + 1)$. $\psi(k + 1)$ is chosen nondeterministically, and represents a guess as to the configuration of X the next time X scans square $k + 1$ on its pushdown tape.

(2) Suppose X prints a symbol Y_k' on its pushdown tape but does not shift its pushdown head. Then S prints a $*$ to the right of $\phi(k)$ to label that configuration as obsolete (garbage), and then proceeds to write the new configuration

$$\phi'(k) = \langle q', i', u', Y_k' \rangle$$

to the right of $\phi(k)*$, using the method of the preceding paragraph. Simultaneously S guesses at the configuration $\psi'(k)$ and writes it on the lower channel below $\phi'(k)$.

(3) Suppose X prints a blank on its pushdown tape and shifts its pushdown head left. Then S searches left on its stack tape (without erasing) until it finds the rightmost configuration $\phi(k - 1)$ which is to the left of $\phi(k)$ and which is not labeled obsolete with a $*$. On the lower channel below $\phi(k - 1)$ is the configuration

$$\psi(k - 1) = \langle q_2, i_2, u_2, Y'' \rangle$$

which represents the guess made earlier by S as to the present configuration of X. The stack automaton S now checks whether $\psi(k - 1)$ is correct, by first checking whether Y'' is the symbol printed by X on its pushdown head when in the configuration $\phi(k - 1)$. The remaining entries q_2, i_2, u_2 of $\psi(k - 1)$ are checked against the rightmost configuration $\phi(k)$ on the stack to see if they are correct. The method of checking is exactly the reverse of the method mentioned in case (1) for producing $\phi(k + 1)$ next to $\phi(k)$, and as each symbol (starting with the right) of $\psi(k - 1)$ is checked, the corresponding symbol of $\phi(k)$ will be erased. If $\psi(k - 1)$ is incorrect, the computation of S is terminated unsuccessfully (recall S is nondeterministic). If $\psi(k - 1)$ is correct, then all garbage to the right of $\psi(k - 1)$ is erased, the pair $\begin{matrix}\phi(k-1)\\ \psi(k-1)\end{matrix}$ is labeled obsolete with a $*$, $\psi(k - 1)$ is copied as $\phi'(k - 1)$ on the upper channel immediately to the right of $\phi(k - 1)$, and a new guess $\psi'(k - 1)$ is written on the lower channel below $\phi'(k - 1)$.

(4) If X accepts the input w, then S accepts the input w.

LEMMA 4. *If a set A of strings is accepted by some deterministic auxiliary PDM within storage $L(n) = n \log n$, then A is accepted by a deterministic stack automaton.*

PROOF. The argument is the same as the previous proof, with two exceptions. First, when the simulating deterministic stack automaton S produces the successor $\phi(k + 1)$ to a configuration $\phi(k)$ on its stack tape, it cannot use the same method as the nondeterministic stack automaton in the previous proof. The method used is that described in [11] for a deterministic nonerasing stack automaton to update an instantaneous description of an $n \log n$ storage bounded Turing machine. Since the auxiliary PDM being simulated has its work tapes bounded by $n \log n$, the new method works. Similarly, the new method is used by S for the other copy and ckeck operations on the configurations of X.

The second difference in the present argument is that the deterministic stack automaton S cannot guess nondeterministically at the configurations $\psi(k)$. Instead, S guesses systematically. We assume the possible configurations of the auxiliary PDM have a lexicographical ordering. Initially S sets $\psi(k)$ equal to the first configuration in the order, and writes this configuration on the lower channel below $\phi(k)$. In general, suppose S is updating its stack tape to correspond to a left shift of the pushdown head of x [see case (3) in the preceding proof]. If the latest guess $\psi(k - 1)$ of the new configuration is incorrect, S erases all information to the right of the pair $\begin{matrix}\phi(k-1)\\ \psi(k-1)\end{matrix}$, labels that pair obsolete with a $*$, and copies over the pair on the space immediately to the right of the $*$, except $\psi(k - 1)$ is replaced by the next configuration in the lexicographical order. Then the simulation of X is repeated, starting with the configuration $\phi(k - 1)$. Sooner or later the right choice for $\psi(k - 1)$ will be found, and the simulation can continue.

This completes the proof of Lemma 4 and of Theorem 2. As an immediate consequence of Theorem 2 and the definition of Cobham's class $\mathfrak{L}_*$ (see Section 2) we have the following.

COROLLARY 1. *A set A is in $\mathfrak{L}_*$ if and only if A is accepted by some two-way multihead pushdown automaton.*

As a second consequence, we can answer some open questions about the effect of determinacy.

COROLLARY 2. *In the case of both writing pushdown acceptors and two-way multihead pushdown automata, the deterministic and nondeterministic versions have the same computing power.*

For the next corollary, we need a result proved in [10].

LEMMA 5 (*Hennie-Stearns*). *If $T_1(n)$ is a real-time countable function, then there is a set A of strings which is accepted by some deterministic multitape Turing machine within time $T_1(n)$, but by no such machine within time $T_2(n)$ for any function T_2 satisfying*

$$\liminf_{n\to\infty} \frac{T_2(n) \log T_2(n)}{T_1(n)} = 0.$$

The next result answers an open question in [6].

COROLLARY 3. *The class of sets accepted by deterministic two-way stack automata is properly included in the class accepted by nondeterministic such machines. Similarly, the classes defined by the machines in lines* 1, 3, *and* 5 *of Table I form an increasing sequence with proper inclusions.*

PROOF. This follows easily from Lemma 5 and Theorem 2. For example, if we set $T_1(n) = 2^{n^2}$, and $T_2(n) = n^{n \log n}$, then $T_2(n)$ eventually dominates n^{cn} for every constant c, and hence every set A accepted by some deterministic stack automaton is accepted by some Turing machine within time $T_2(n)$. But then by Lemma 5, there is some set A which is accepted by a Turing machine within time $T_1(n)$, and hence by a nondeterministic stack automaton, but A is accepted by no Turing machine within time $T_2(n)$ and hence by no deterministic stack automaton.

The next corollary was first proved in [12], by a direct but complex argument.

COROLLARY 4 (*Knuth-Bigelow*). *The class of context-sensitive languages is properly included in the class of sets accepted by deterministic two-way stack automata.*

PROOF. All context-sensitive languages are accepted by nondeterministic Turing machines which operate with storage bounded by a linear function of the length of the input. But these machines are clearly special cases of writing pushdown acceptors (row 3 of Table I), so all context-sensitive languages are accepted by writing pushdown acceptors. Corollary 4 now follows from Corollary 3.

The next two corollaries state for nondeterministic Turing machines facts which are obvious for deterministic Turing machines.

COROLLARY 5. *If a set A of strings is accepted by some nondeterministic Turing machine within storage $L(n) \geq \log_2 n$, then A is accepted by a (deterministic) time-bounded computer within time $T(n) = 2^{cL(n)}$ for some constant c.*

This follows directly from Theorem 1, by dropping the pushdown tape from the auxiliary PDM. In particular, we have for the case $L(n) = n$:

COROLLARY 6. *Every context-sensitive language is accepted by some deterministic Turing machine within time 2^{cn}, for some constant c.*

COROLLARY 7. *The class of sets accepted by each machine type of Table I is closed under union, intersection, and taking complements.*

This is clear from the characterizations given in the third column of the table. Since each of the functions there is real-time countable (i.e. suitably easy to compute), a Turing machine accepting a set in any of the classes can be made to halt on all inputs. The corollary follows easily.

6. *Conclusion and Open Questions*

The major open question concerning these results is whether the pushdown tape really adds to the computing power of auxiliary PDM's. That is, we cannot show that Theorem 1 becomes false if "auxiliary PDM" is replaced by "Turing machine" in parts (a) and (b). This modified Theorem 1 would assert the converse of Corollary 5; that is, a set is accepted by a time-bounded computer within time $2^{cL(n)}$ if and only if it is accepted by a Turing machine within storage $L(n)$. Such a general equivalence between time and storage appears unlikely, but we have no proof.

A second group of questions which remain open concerns the two-way pushdown automaton described in [7]. This device is easily characterized as an auxiliary PDM operating with zero auxiliary storage (i.e. no work tapes). Since the auxiliary storage is less than $\log_2 n$, Theorem 1 does not apply, and such questions as whether the nondeterministic version is more powerful than the deterministic version, and whether the class of sets accepted by either version is closed under complements, remain unanswered.

A final long-standing problem in the field of computational complexity is to prove that some interesting set (or function) must take a long time to compute on a reasonable general computer model. More specifically, no one can find any set not in Cobham's class $\mathcal{L}_*$, except artificial examples through use of diagonal arguments. It is, however, easy to find plausible candidates, such as the set of binary notations for the primes. It seems to me that, because of the simple nature of multihead pushdown machines, Corollary 1 in Section 5 might provide a first step toward finding something interesting not in $\mathcal{L}_*$.

ACKNOWLEDGMENT. The author wishes to thank Walter Savitch for many fruitful conversations concerning the results.

REFERENCES

1. Aho, A. V., Hopcroft, J. E., and Ullman, J. D. Time and tape complexity of pushdown automaton languages. *Inform. Contr. 13*, 3 (Sept. 1968), 186–206.
2. Cobham, A. The intrinsic computational difficulty of functions. Proc. 1964 Internat. Congr. for Logic, Methodology, and Philosophy of Science, North-Holland, Amsterdam, pp. 24–30.
3. Cole, S. N. Real-time computation by n-dimensional iterative arrays of finite-state machines. IEEE Conf. Record of 1966 Seventh Annual Symp. on Switching and Automata Theory, Oct. 1966, pp. 53–77.
4. Cook, S. A. Variations on pushdown machines. Conf. Record of ACM Symposium on the Theory of Computing, Marina del Rey, Calif., May 1969.
5. Earley, J. An efficient context-free parsing algorithm. *Comm. ACM 13*, 2 (Feb. 1970), 94–102.
6. Ginsburg, S., Greibach, S. A., and Harrison, M. A. Stack automata and compiling. *J. ACM 14*, 1 (Jan. 1967), 172–201.
7. Gray, J., Harrison, M. A., and Ibarra, O. H. Two-way pushdown automata. *Inform. Contr. 11*, 1, 2 (July, Aug. 1967), 30–70.
8. Harrison, M. A., and Ibarra, O. H. Multitape and multihead pushdown automata. *Inform. Contr. 13*, 5 (Nov. 1968), 433–470.
9. Hartmanis, J., and Stearns, R. E. On the computational complexity of algorithms. *Trans. Amer. Math. Soc. 117* (1965), 285–306.
10. Hennie, F. C., and Stearns, R. E. Two-tape simulation of multitape Turing machines. *J. ACM 13*, 4 (Oct. 1966), 533–546.
11. Hopcroft, J. E., and Ullman, J. D. Nonerasing stack automata. *J. Comput. Syst. Sci. 1*, 2 (Aug. 1967), 166–186.
12. Knuth, D. E., and Bigelow, R. H. Programming languages for automata. *J. ACM 14*, 4 (Oct. 1967), 615–635.
13. Mager, G. Writing pushdown acceptors. *J. Comput. Syst. Sci. 3*, 3 (1969), 276–319.
14. Shepherdson, J. C., and Sturgis, H. E. Computability of recursive functions. *J. ACM 10*, 2 (April 1963), 217–255.

RECEIVED AUGUST, 1969; REVISED MAY, 1970