INFORMATION AND CONTROL 13, 186–206 (1968)

Time and Tape Complexity of Pushdown Automaton Languages

A. V. AHO, J. E. HOPCROFT,* AND J. D. ULLMAN

Bell Telephone Laboratories, Incorporated
Murray Hill, New Jersey 07971

An algorithm is presented which will determine whether any string w in Σ^*, of length n, is contained in a language $L \subseteq \Sigma^*$ defined by a two-way nondeterministic pushdown automaton. This algorithm requires time n^3 when implemented on a random access computer. It requires n^4 time and n^2 tape when implemented on a multitape Turing machine.

If the pushdown automaton is deterministic, the algorithm requires n^2 time on a random access computer and $n^2 \log n$ time on a multitape Turing machine.

I. INTRODUCTION

A pushdown store is a list in which information can be accessed only on a last-in first-out principle of operation. The use of pushdown stores is an important technique in the construction of compilers and other language processing devices.

Of particular interest from both practical and theoretical considerations is how the time and memory required to process a language is functionally related to the length of the input sentence under consideration. In this paper we consider languages defined by pushdown store systems and we investigate how much time and memory is needed in order to determine whether an arbitrary input sentence belongs or does not belong to some language in this class.

To obtain analytical results, a pushdown automaton (PDA for short) will be used as the model of a pushdown store system. There are four types of pushdown automaton depending on whether the automaton is deterministic or nondeterministic and whether it moves one or two ways on the input. We will use the following abbreviations for pushdown au-

* Currently at Cornell University, Ithaca, New York.

tomata:

1 = one-way on the input
2 = two-way on the input
D = deterministic
N = nondeterministic.

For example, a 2N PDA will refer to a two-way nondeterministic pushdown automaton.

Informally, a language $L \subseteq \Sigma^{*}$[1] is said to be recognized in time $f(n)$ if, given any input sentence w in Σ^* of length n, we can determine in at most $f(n)$ elementary operations whether w is in L. L is said to be recognized in space $g(n)$ if given any w of length n, we can determine whether w is in L using at most $g(n)$ memory cells.

In this paper an algorithm is presented for the recognition of a language L defined by a 2N PDA. Implemented on an appropriate random access computer, this algorithm recognizes L in time n^3 and space n^2. Using this algorithm, a multitape Turing machine can recognize L in time n^4 and space n^2.

If L is specified by a 2D PDA, then L can be recognized in time n^2 on a random access computer and in time $n^2 \log n$ on a multitape Turing machine.

In the parlance of computational complexity we thus show that the 2N PDA languages are contained within the classes of languages of time complexity n^4 and tape complexity n^2. The 2D PDA languages are contained within the class of languages of time complexity $n^2 \log n$.

These results complement the results on one-way pushdown automata. It has been shown that the 1N PDA languages are contained within the class of languages of time complexity n^3 (Younger, 1967) and that the 1N PDA and 1D PDA languages are contained within the class of languages of tape complexity $(\log n)^2$ [Lewis, Stearns, and Hartmanis (1965)].

II. BASIC DEFINITIONS AND PRELIMINARY RESULTS

Informally, a pushdown automaton is a mathematical model of a device consisting of an input tape, a finite state control and a pushdown stack (Fig. 1). The input tape holds a string of input symbols delimited by left and right endmarkers. The finite state control is always in one of a

[1] Σ^* represents all finite length strings over Σ including ϵ, the string of zero length.

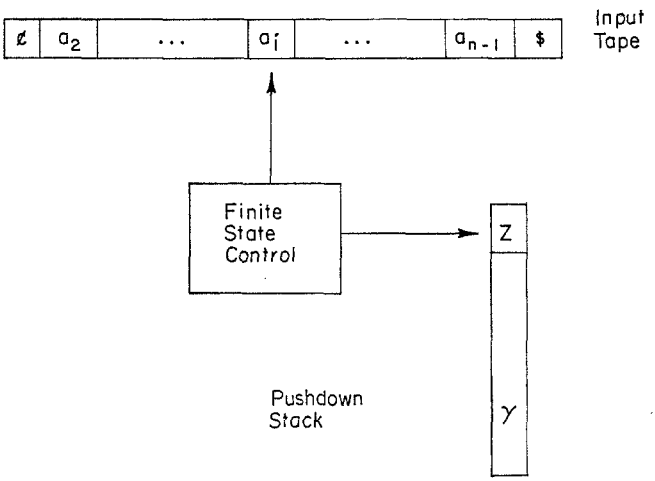

FIG. 1. Pushdown Automaton

finite number of states. The pushdown stack operates on a last-in first-out principle of operation and can hold any finite length string of pushdown symbols.

Formally, a *two-way nondeterministic pushdown automaton* (2N PDA), P, is denoted by a 7-tuple $(Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ where:

(1) Q is a finite nonempty set of *states*.

(2) Σ is a finite nonempty set of *input symbols*. Σ includes the two special symbols, ¢ and \$, which will be used exclusively as the *left* and *right endmarkers*, respectively.

(3) Γ is a finite nonempty set of *pushdown symbols*.

(4) δ is a mapping from $Q \times \Sigma \times \Gamma$ into the finite subsets of $Q \times \Gamma^* \times \{-1, 0, 1\}$.

(5) q_0, in Q, is the *initial state*.

(6) Z_0, in Γ, is the *initial pushdown symbol* on the pushdown stack.

(7) F, a subset of Q, is the set of *final states*.

If a triple (q, α, d) is in $\delta(p, a, Z)$, and if the finite state control of P is in state p, with the input symbol a under the input head and the pushdown symbol Z on the top of the pushdown stack, then P may enter state q, move the input head in the direction d (where $d = -1, 0$, or 1 indicates the head is to move one square to the left, remain stationary, or move one square to the right, respectively) and replace the symbol Z

on the top of the stack by the string of pushdown symbols α. If $\alpha = \epsilon$ (the empty string), then Z is erased from the top of the stack.

If for no (q, a, Z) in the domain of δ does $\delta(q, a, Z)$ contain an element of the form $(p, \alpha, -1)$, then P is said to be *one-way*.

If $\delta(q, a, Z)$ never contains more than one element, then P is *deterministic*.

Thus, there are four types of pushdown automaton, depending on whether the automaton is deterministic or nondeterministic and whether it is one-way or two-way.

An instantaneous description of any pushdown automaton $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ will be denoted by a quadruple, $(q, \not{\epsilon}w\$, i, \gamma)$, called a *configuration* of P, where:

(1) q is a state in Q.

(2) $w = a_2 \cdots a_{n-1}$, where each a_i is in $\Sigma - \{\not{\epsilon}, \$\}$. W calle w the *input string*.

(3) i is an integer between 1 and n. a_i, the ith input symbol, is the symbol currently being scanned by the input head, with the assumption $a_1 = \not{\epsilon}$ and $a_n = \$$.

(4) γ, in Γ^*, represents the contents of the pushdown stack. If $\gamma = \alpha Z$, with Z in Γ, then the symbol Z is understood to be at the top of the pushdown stack.

A move by P will be denoted by the relation $\vdash_P$ (or $\vdash$ whenever P is clear) over the set of configurations. We say $(p, \not{\epsilon}w\$, i, \alpha Z) \vdash (q, \not{\epsilon}w\$, i + d, \alpha\gamma)$ if $\delta(p, a_i, Z)$ contains (q, γ, d). However, we must have $1 \leq i + d \leq n$ if $| \not{\epsilon}w\$ | = n$, where $| x |$ is the length of x.

A sequence of moves by P will be described by the reflexive and transitive closure of the relation $\vdash$, denoted by the symbol $\vdash^*$. We define $\vdash^*$ as follows:

(1) $(q, \not{\epsilon}w\$, i, \gamma) \vdash^0 (q, \not{\epsilon}w\$, i, \gamma)$.

(2) If $(q_1, \not{\epsilon}w\$, i_1, \gamma_1) \vdash^{k-1} (q_2, \not{\epsilon}w\$, i_2, \gamma_2)$ and $(q_2, \not{\epsilon}w\$, i_2, \gamma_2) \vdash (q_3, \not{\epsilon}w\$, i_3, \gamma_3)$, then $(q_1, \not{\epsilon}w\$, i_1, \gamma_1) \vdash^k (q_3, \not{\epsilon}w\$, i_3, \gamma_3)$, for all $k \geq 1$.

(3) $(q_1, \not{\epsilon}w\$, i_1, \gamma_1) \vdash^* (q_2, \not{\epsilon}w\$, i_2, \gamma_2)$ if and only if $(q_1, \not{\epsilon}w\$, i_1, \gamma_1) \vdash^i (q_2, \not{\epsilon}w\$, i_2, \gamma_2)$ for some $i \geq 0$.

A sequence of moves by which P goes from configuration $(p, \not{\epsilon}w\$, i, \alpha)$ to configuration $(q, \not{\epsilon}w\$, j, \beta)$ will be called a *scan*. It is often convenient to give such a scan the name $(p, \not{\epsilon}w\$, i, \alpha) \vdash^* (q, \not{\epsilon}w\$, j, \beta)$, even though this name may not uniquely specify a particular sequence of moves.

Each of the four types of pushdown automaton can accept an input

string w by one of two modes of acceptance. A pushdown automaton P starts off in the initial configuration $(q_0, \not\!w\$, 1, Z_0)$, where q_0 is the specified initial state and Z_0 is the specified initial pushdown symbol. The input head is over the left endmarker. The input string w is accepted by P if P can then make any sequence of moves such that it ends up in a configuration with the input head over the right endmarker and with either the pushdown stack empty or the finite state control in a final state.

More precisely, let $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ be a pushdown automaton of some type. An input string $w = a_2 \cdots a_{n-1}$ is *accepted by empty stack* by P if $(q_0, \not\!w\$, 1, Z_0) \vdash^* (q, \not\!w\$, n, \epsilon)$, for some q in Q. An input string w is *accepted by final state* by P if $(q_0, \not\!w\$, 1, Z_0) \vdash^* (p, \not\!w\$, n, \gamma)$ for some p in F, γ in Γ^*.

$N(P)$ and $T(P)$ are the sets of input strings accepted by P by empty stack and final state, respectively. $N(P)$ and $T(P)$ are the languages defined by P.

Acceptance by final state or empty stack does not change the class of languages that can be defined by a class of pushdown automata of the same type. That is to say, the class of languages defined by 2N PDA (2D PDA, 1N PDA, or 1D PDA) accepting by final state is equivalent to the class of languages defined by 2N PDA (2D PDA, 1N PDA, or 1D PDA, respectively) accepting by empty stack.

Moreover, the endmarkers, $\not\!$ and \$, do not add any power to 1N PDA. However, 1D PDA without a right endmarker accepting by empty stack are less powerful than 1D PDA accepting by final state (Ginsburg and Greibach, 1966).

Removing endmarkers from 2D PDA reduces their recognitive power (Gray, Harrison and Ibarra, 1967). The importance of endmarkers to 2N PDA is not yet fully understood.

For the remainder of this paper we will assume a pushdown automaton to have left and right endmarkers.

For notational convenience, we will assume that on a single move any pushdown automaton of a given type either writes exactly one symbol on the pushdown stack or erases the top symbol from the pushdown stack. The following lemma shows that this represents no loss of generality.

LEMMA 1. *Given a pushdown automaton* $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ *of a given type, we can effectively construct an equivalent pushdown automaton* $P' = (Q', \Sigma, \Gamma', \delta', q_0', Z_0', F')$ *of the same type such that each element in* $\delta'(q, a, Z)$ *is either of the form* (p, ϵ, d) *or* (p, ZY, d) *with* Y *in* Γ, *for all* q, a, Z *in the domain of* δ'.

The proof is straightforward and will be left for the reader.

Again for notational convenience, we will further assume that a pushdown automaton will accept an input string by both emptying its pushdown stack and going into a designated final state. A pushdown automaton satisfying Lemma 1 and accepting an input string in this fashion is said to be in *normal form.*

LEMMA 5. *Given any pushdown automaton P of a certain type an equivalent normal form pushdown automaton P' of the same type can be constructed from P.*

The proof is left for the reader.

Languages defined by pushdown automata are closely related to several other well-known classes of languages. The 1D PDA languages are the deterministic context-free languages studied by Ginsburg and Greibach (1966) and these languages are equivalent to the $LR(k)$ languages described by Knuth (1965).

The class of 1N PDA languages is exactly the class of context-free languages (Chomsky 1962). The context-free languages properly include the deterministic languages, and this class of languages has received considerable attention.

A 2D PDA can define languages which are not contest-free. For example, $\{a^n b^n c^n \mid n \geq 1\}$ and $\{ww \mid w \text{ is in } \{a, b\}^*\}$ are two examples of 2D PDA languages which are not context-free languages. The 2D PDA languages are a subset of the deterministic context-sensitive languages (Stearns, Hartmanis, Lewis, 1965; Gray, Harrison, Ibarra, 1967), but it is not known whether a 2D PDA can define an arbitrary context-free language. However, we suspect $\{xx^R yy^R \mid x, y \text{ are in } \{a, b\}^*\}$[2] to be a context-free language which cannot be defined by a 2D PDA.

A 2D PDA is capable of defining the language

$$\{x_1 c x_2 \cdots c x_n c c y_1 c y_2 \cdots c y_m \mid x_i \text{ is in } \{a, b\}^*, \quad x_i \neq x_j \text{ for } i \neq j,$$

and for all j, $y_j = x_i$ for some $1 \leq i \leq n\}$. This language reflects syntactic features found in algorithmic programming languages with declarations. Thus, it seems a 2D PDA can define much of the syntax of most high-level programming languages.

The 2N PDA languages are a proper superset of the context-free languages. However, at present it is not known whether the 2N PDA languages are a subset of the context-sensitive languages.

[2] z^R is the reversal of z.

III. ALGORITHM FOR THE RECOGNITION OF A 2N PDA LANGUAGE

An algorithm is said to recognize a language $L \subseteq \Sigma^*$ if for any w in Σ^*, the algorithm will, with a finite amount of computation, render the decision whether or not w is in L.

We will now proceed to present an algorithm which will recognize an arbitrary 2N PDA language $L \subseteq (\Sigma - \{\not\epsilon, \$\})^*$. Let us assume that the input string which is to be tested for membership in L is $w = a_2 \cdots a_{n-1}$, where each a_i is in $\Sigma - \{\not\epsilon, \$\}$, $2 \leq i \leq n-1$. The algorithm will render the decision whether or not w is in L after $\sim n^3$ elementary computations. (The notation $g(n) = \sim f(n)$ will mean that there exists some constant c such that $g(n) \leq cf(n)$ for all n.) The notion of an elementary computation will be defined shortly.

Without loss of generality, we can assume that the language L is $N(P)$ for some normal form 2N PDA, $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, \{q_f\})$ where $\#(Q) = s, \#(\Gamma) = t$, and q_f is the designated final state.[3]

Let $a_2 \cdots a_{n-1}$ be the test string, and let $a_1 = \not\epsilon$, $a_n = \$$. The heart of the recognition algorithm is an $n \times n$ matrix R, called the *recognition matrix*. Each entry $r(i, j)$, $1 \leq i, j \leq n$, in R has a value which is a subset of $Q \times \Gamma \times Q$. An entry $r(i, j)$ will contain an element (p, Z, q) if and only if $(p, a_i a_2 \cdots a_n, i, Z) \vdash_p^* (q, a_1 \cdots a_n, j, \epsilon)$. Consequently, if and only if $r(1, n)$ contains the element (q_0, Z_0, q_f) does $(q_0, a_1 \cdots a_n, 1, Z_0) \vdash^* (q_f, a_1 \cdots a_n, n, \epsilon)$. Of course, $(q_0, a_1 \cdots a_n, 1, Z_0) \vdash^* (q_f, a_1 \cdots a_n, n, \epsilon)$ if and only if $a_2 \cdots a_{n-1}$ is in $N(P)$. Thus, in order to determine whether or not w is in L, the matrix R will be computed and the entry $r(1, n)$ examined.

The mapping δ, which specifies the behavior of P, is partitioned into two set-valued maps δ_1 and δ_2. δ_1 is a mapping from $J_n \times J_n$ into the subsets of $Q \times \Gamma \times Q$, where J_n is the set of integers $\{1, 2, \cdots, n\}$. δ_2 maps $J_n \times J_n$ into the subsets of $Q \times \Gamma\Gamma \times Q$. δ_1 represents those rules of δ which cause the top symbol to be erased from the stack and δ_2 represents that part of δ which causes one symbol to be written on the top of the stack. More precisely,

(1) $\delta_1(i, i + d)$ contains (p, Z, q) if and only if $\delta(p, a_i, Z)$ contains (q, ϵ, d).

(2) $\delta_2(i, i + d)$ contains (p, ZY, q) if and only if $\delta(p, a_i, Z)$ contains (q, ZY, d).

For convenience, we will express our recognition algorithm in terms of

[3] $\#(A)$ is the number of elements in the set A.

an operation Θ, which will be called a *convolution*. Θ has three arguments, each of which is a set of triples. The first argument will be a subset of $Q \times \Gamma\Gamma \times Q$, and the second and third arguments, as well as the range of Θ, are subsets of $Q \times \Gamma \times Q$.

We define $\Theta(A_1, A_2, A_3)$ to be the set $\{(p, Z, q) \mid (\exists q_1, q_2, Z')((p, ZZ', q_1)$ is in A_1, (q_1, Z', q_2) is in A_2, and (q_2, Z, q) is in $A_3)\}$.

For example, if

(1) $\delta_2(i, i + d)$ contains (q, ZY, q_1),

(2) $r(i + d, j)$ contains (q_1, Y, q_2) and

(3) $r(j, k)$ contains (q_1, Z, q),

then $\Theta(\delta_2(i, i + d), r(i + d, j), r(j, k))$ contains (p, Z, q). In this case P can make the following scan:

$$\begin{aligned}(p, a_1 \cdots a_n, i, Z) &\vdash (q_1, a_1 \cdots a_n, i + d, ZY)\\ &\vdash^* (q_2, a_1 \cdots a_n, j, Z)\\ &\vdash^* (q, a_1 \cdots a_n, k, \epsilon).\end{aligned}$$

We will define one convolution operation to be one elementary step of the algorithm. The total amount of computation will be in terms of the number of convolutions required before the algorithm will terminate. Notice that in order to evaluate $\Theta(A_1, A_2, A_3)$ for any set of arguments only a fixed amount of calculation is required. That is to say, for a given 2N PDA P, the value of $\Theta(A_1, A_2, A_3)$ for any set of arguments is dependent only upon the specification of P and not upon the length of the test string $w = a_2 \cdots a_{n-1}$. We could have chosen some other finite computation as the elementary operation for the computation complexity of the algorithm. However, since we are not concerned with implementation constants here, the convolution operation will serve as a convenient measure of complexity.

To assist in the computation of the R matrix an auxiliary list A is used. A operates as a pushdown list. Each entry of A except the bottommost is an element of the form $(i, j, (p, Z, q))$ with $1 \leq i, j \leq n$, p and q in Q, and Z in Γ. If an entry of A contains $(i, j, (p, Z, q))$, then entry $r(i, j)$ of R contains (p, Z, q).

To evaluate R we will determine what new triples of the form (p, Z, q) can be added to R as a consequence of the current contents of R. Each new triple of the form (p, Z, q) added to some entry $r(i, j)$ of R is also added to the list A as an element of the form $(i, j, (p, Z, q))$.

Initially each entry of R is empty and the list A contains the element $(0, 0, 0)$.

We then add to R and to A all elements which result from δ_1. That is, if $\delta_1(i, i + d)$ contains (p, Z, q), then $(p, ¢w\$, i, Z) \vdash (q, ¢w\$, i + d, \epsilon)$. Thus, (p, Z, q) is added to $r(i, i + d)$ and the element $(i, i + d, (p, Z, q))$ is placed on the list A.

We then examine the topmost entry of A, say $(i, j, (p, Z, q))$, and determine what new entries can be added to R as a consequence of (p, Z, q) being in $r(i, j)$.

If, for some d in $\{-1, 0, 1\}$, $\delta_2(i - d, i)$ contains $(p', Z'Z, p)$ and if at this time, for some k in J_n, $r(j, k)$ contains (q, Z', q') then

$$\begin{aligned}(p', ¢w\$, i - d, Z') &\vdash (p, ¢w\$, i, Z'Z)\\ &\vdash^* (q, ¢w\$, j, Z')\\ &\vdash^* (q', ¢w\$, k, \epsilon).\end{aligned}$$

Thus, $r(i - d, k)$ should contain (p', Z', q'). If $r(i - d, k)$ does not contain (p', Z', q'), then this element is added to $r(i - d, k)$ and $(i - d, k, (p', Z', q'))$ is added to the top of A.

There is one other way in which the element $(i, j, (p, Z, q))$ in A can give rise to additional elements in R. If, for some h in J_n and d in $\{-1, 0, 1\}$, $\delta_2(h, h + d)$ contains (p', ZZ', q') and if $r(h + d, i)$ already contains (q', Z', p), then

$$\begin{aligned}(p', ¢w\$, h, Z) &\vdash (q', ¢w\$, h + d, ZZ')\\ &\vdash^* (p, ¢w\$, i, Z)\\ &\vdash^* (q, ¢w\$, j, \epsilon)\end{aligned}$$

Thus, $r(h, j)$ should contain (p', Z, q). If $r(h, j)$ does not contain (p', Z, q), then this element is added to $r(h, j)$ and $(h, j, (p', Z, q))$ is added to the list A.

In general, the list A is used in the following fashion. The top element of A is read and erased. If this element is $(i, j, (p, Z, q))$, then $\Theta(\delta_2(i - d, i), \{(p, Z, q)\}, r(j, k))$ and $\Theta(\delta_2(h, h + d), r(h + d, i), \{(p, Z, q)\})$ are evaluated, for all d in $\{-1, 0, 1\}$ and k and h in J_n such that $1 \leq i - d \leq n$ and $1 \leq h + d \leq n$. Thus, one determines what new triples (p', Z', q') can be added to the entries $r(i - d, k)$ and $r(h, j)$ of R, respectively, as a consequence of (p, Z, q) being in $r(i, j)$.

Each new triple of the form (p', Z', q') added to $r(i', j')$ of R as a result of this computation results in a new entry of the form $(i', j',$

$(p', Z', q'))$ being added to the top of list A. In this manner, each triple added to an entry of R is also placed on the list A exactly once at some time during the computation.

The details of the algorithm are as follows:

Algorithm 1. Let $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, \{q_f\})$ be a normal form 2N PDA where $\#(Q) = s$ and $\#(\Gamma) = t$. Let $w = a_2 \cdots a_{n-1}$ be the input string and let $a_1 = \not{\epsilon}$, $a_n = \$$.

R is an $n \times n$ matrix with each element $r(i, j)$, $1 \leq i, j \leq n$, initially set to ϕ, the empty set. A is a pushdown list which initially contains the entry $(0, 0, 0)$. $A(1)$, $A(2)$, and $A(3)$ denote the first, second and third components, respectively, of the element currently at the top of A.

1. For all d in $\{-1, 0, 1\}$ and all i in J_n such that $1 \leq i + d \leq n$, set $r(i, i + d) = \delta_1(i, i + d)$. For each triple of the form (p, Z, q) added to $r(i, i + d)$, add to the top of A the element $(i, i + d, (p, Z, q))$.
2. If $A(1) = 0$, go to step 3. Otherwise, let $i = A(1)$, $j = A(2)$ and $B = A(3)$. Erase the element at the top of A.
   (a) For all d in $\{-1, 0, 1\}$ such that $1 \leq i - d \leq n$ and for all k in J_n, adjoin to $r(i - d, k)$ the new triples in $\Theta(\delta_2(i - d, i), \{B\}, r(j, k))$. For each new triple (p', Z', q') so adjoined to $r(i - d, k)$ also add to the top of A the element $(i - d, k, (p', Z', q'))$.
   (b) For all h in J_n and for all d in $\{-1, 0, 1\}$ such that $1 \leq h + d \leq n$, adjoin to $r(h, j)$ the new triples in $\Theta(\delta_2(h, h + d), r(h + d, i), \{B\})$. For each new triple (p', Z', q') adjoined to $r(h, j)$ add to the top of A the element $(h, j, (p', Z', q'))$.
   (c) Repeat step 2.
3. Accept $w = a_2 \cdots a_{n-1}$ if $r(1, n)$ contains (q_0, Z_0, q_f). Otherwise reject w. Halt.

This algorithm must terminate, since there are at most s^2tn^2 triples in R, and each triple in R gives rise to exactly one element on the list A. Each element appearing on the top of list A causes the evaluation of $6n$ convolutions. Therefore, the total number of convolutions to be calculated during the course of Algorithm 1 is at most $6s^2tn^3$.

We now need to verify that the algorithm recognizes the language L.

LEMMA 3. *If* $(p, \not{\epsilon}w\$, i, Z) \vdash^m (q, \not{\epsilon}w\$, j, \epsilon), m \geq 1$, *then* $r(i, j)$ *contains* (p, Z, q) *after Algorithm* 1 *has terminated.*

Proof. The proof will be by induction on m. Lemma 3 is clearly true when $m = 1$.

Suppose that the statement of Lemma 3 is true for all $m < m'$, where

$m' > 1$. Consider an arbitrary scan consisting of exactly m' moves:

$$\begin{aligned}(p, \not{c}w\$, i, Z) &\vdash (p', \not{c}w\$, i + d, ZY)\\ &\vdash^{m_1} (q', \not{c}w\$, k, Z)\\ &\vdash^{m_2} (q, \not{c}w\$, j, \epsilon)\end{aligned}$$

where $m_1 + m_2 + 1 = m'$, p' and q' are in Q.

Each triple in each element of R must at some time appear at the top of the list A. From the inductive hypothesis $r(i + d, k)$ contains (p', Y, q') and $r(k, j)$ contains (q', Z, q) upon termination of Algorithm 1. Two cases can occur. Either the element $(i + d, k, (p', Y, q'))$ appears at the top of A before $(k, j, (q', Z, q))$ or conversely.

Suppose $(i + d, k, (p', Y, q'))$ is the first to appear at the top of A. If at this point $r(k, j)$ already contains (q', Z, q), then the triple (p, Z, q) will be adjoined to $r(i, j)$ as a result of step 2(a) of Algorithm 1. If however, $r(k, j)$ does not contain (q', Z, q) at this time, then when $(k, j, (q', Z, q))$ appears at the top of A, $r(i + d, k)$ will contain (p', Y, q') and the triple (p, Z, q) will be added to $r(i, j)$ in step 2(b) of the algorithm. The argument for the case where $(k, j, (q', Z, q))$ appears at the top of A before $(i + d, k, (p', Y, q'))$ is handled similarly.

Thus, Lemma 3 holds for all $m \geq 1$.

LEMMA 4. *If (p, Z, q) is the mth triple to be added to R by Algoirthm 1, and if (p, Z, q) is adjoined to element $r(i, j)$, then $(p, \not{c}w\$, i, Z) \vdash^* (q, \not{c}w\$, j, \epsilon)$.*

Proof. The proof will again be by induction on m. Lemma 4 is obviously true when $m = 1$.

Assuming the statement of Lemma 4 is true for all $m < m'$ where $m' > 1$, consider the mth triple to be added to R. Suppose this triple is (p, Z, q) and it is added to $r(i, j)$.

If (p, Z, q) is added to $r(i, j)$ in the course of step 1 of the algorithm, then the statement of the lemma is trivially true.

If (p, Z, q) is added to $r(i, j)$ as a consequence of step 2(a) of the algorithm, then for some d in $\{-1, 0, 1\}$, k in J_n, Y in Γ, p' and q' in Q, $\delta_2(i, i + d)$ contains (p, ZY, p'), $r(i + d, k)$ contains (p', Y, q') and $r(k, j)$ contains (q', Z, q). From the inductive hypothesis,

$$\begin{aligned}(p, \not{c}w\$, i, Z) &\vdash (p', \not{c}w\$, i + d, ZY)\\ &\vdash^* (q', \not{c}w\$, k, Z)\\ &\vdash^* (q, \not{c}w\$, j, \epsilon).\end{aligned}$$

The argument is similar if (p, Z, q) is added to $r(i, j)$ in step 2(b) of Algorithm 1.

Thus, Lemma 4 applies to all triples added to R.

From Lemmas 3 and 4 we have the following result immediately.

THEOREM 1. *After Algorithm* 1 *has terminated,* $r(i, j)$ *contains* (p, Z, q) *if and only if* $(p, \not{w}\$, i, Z) \vdash^* (q, \not{w}\$, j, \epsilon)$.

Since $r(1, n)$ contains (q_0, Z_0, q_f) if and only if w is in L, Algorithm 1 does indeed recognize L.

If the address portion of a memory word of a random access computer can contain a number as large as n, then Algorithm 1 can be implemented on this computer in time $\sim n^3$ using at most $\sim n^2$ memory words. In Section V, we will show that this algorithm can be implemented on a multitape Turing machine of time complexity no greater than n^4 and tape complexity no greater than n^2.

It should be made clear that although the total number of convolutions required to compute the matrix R is $\sim n^3$, this does not imply that the total number of moves by which the 2N PDA P, accepts (or rejects) the input string $w = a_2 \cdots a_{n-1}$ is bounded by cn^3 for any constant c.

In fact, given a 2N PDA (or 2D PDA) P, it is possible to effectively design from P a 2N PDA (or 2D PDA) P', which in addition to stimulating the behavior of P scanning any input string $w = a_2 \cdots a_{n-1}$, also counts from 1 to 2^n between each move made by P. The number of moves made by P and the number of moves made by P' during corresponding scans of an input string w differ by an exponential factor of the length of w. However, using Algorithm 1, the number of convolutions required to determine whether w is accepted by P differs by at most a constant factor from the number of convolutions required to determine whether w is accepted by P'.

IV. RECOGNITION OF A 2D PDA LANGUAGE

If a 2N PDA $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ has the property that for each w in $(\Sigma - \{\not{}, \$\})^*$, p in Q, Z in Γ and $1 \leq i \leq |\not{w}\$|$, there exists at most one j and q such that $(p, \not{w}\$, i, Z) \vdash^* (q, \not{w}\$, j, \epsilon)$, then P is said to be *uniquely erasing.*

LEMMA 4. *If* $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, \{q_f\})$ *is a normal form* 2D PDA, *then* P *is uniquely erasing.*

Lemma 4 follows directly from the definition of a 2D PDA.

Applying Algorithm 1 to a uniquely erasing normal form 2N PDA P with input $w = a_2 \cdots a_{n-1}$, we observe that for each i, the sum over j of

the number of triples in $r(i, j)$ is less than or equal to st. Thus, upon termination of Algorithm 1, R contains no more than stn triples. This, in turn, implies that the total number of elements appearing on the list A is no greater than stn and, hence, the total number of convolutions evaluated in the course of Algorithm 1 is no greater than $6stn^2$.

Thus, Algorithm 1 for a language defined by a uniquely erasing 2N PDA can be implemented in time $\sim n^2$ on an appropriate random access computer. In Section VI we show that in this case the algorithm can be implemented on a multitape Turing machine of time complexity no greater than $n^2 \log n$.

V. TAPE AND TIME COMPLEXITY OF 2N PDA LANGUAGES

In questions concerning the computational complexity of languages, an off-line multitape Turing machine is customarily used as the yardstick in determining the relative computational difficulty of languages.

We will now show that Algorithm 1 for a 2N PDA can be implemented on a multitape Turing machine of time complexity n^4.

The Turing machine model that we will use is the off-line model proposed by Hartmanis and Stearns (1965). A Turing machine M consists of a finite state control coupled by tape heads to one input tape with a read only head and to a finite number of working tapes each with a read-write head. All tapes are divided into an infinity of squares, each square containing either a blank or a tape symbol. Each working tape can be arbitrarily long in both directions.

Each move of the Turing machine is determined by a specified mapping. In one atomic move the Turing machine executes the following sequence of actions.

(1) Each read head reads the symbol on the square currently under the head.
(2) The finite state control changes state.
(3) Each write head overprints a nonblank symbol from the working tape alphabet on the square under the head.
(4) Each tape head either moves left one square, right one square or remains stationary.

M is deterministic in that each move is uniquely specified by the symbols currently under the read heads and the current state of the finite control.

At the start M is in an initial configuration in which:

(1) the finite state control is in a specified initial state,

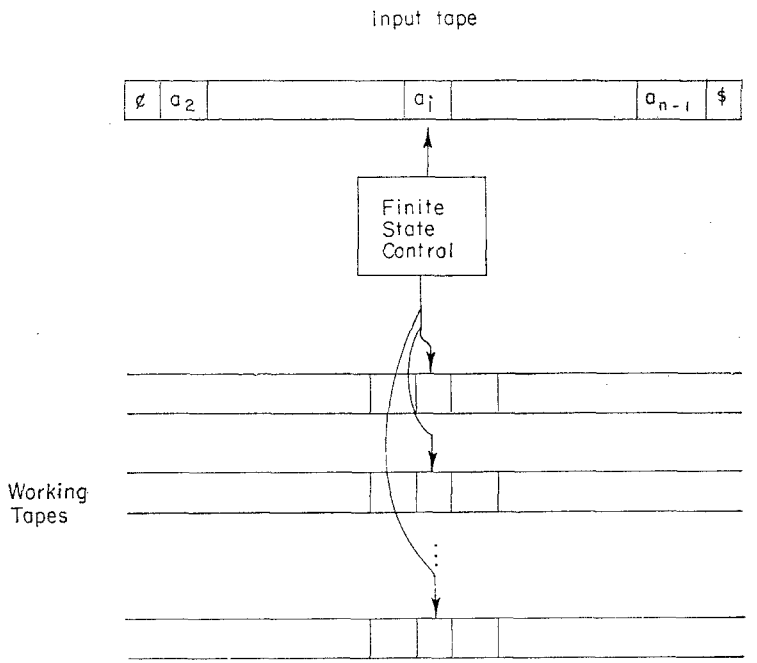

FIG. 2. Off-Line Multitape Turing Machine

(2) the input tape contains $\not{c}a_2 \cdots a_{n-1}\$$, each a_i in $\Sigma - \{\not{c}, \$\}$, and the input head is over the square containing $\not{c}$,

(3) each working tape is filled with blanks.

Starting from its initial configuration, M will make a sequence of moves. If M halts with its finite state control in a specified final state, then the input string $w = a_2 \cdots a_{n-1}$ is said to be accepted by M. If M does not halt or halts in a state which is not a final state, then the input string w is rejected by M. The language defined by the Turing machine M is denoted $T(M)$ and is defined to be the set of input strings accepted by M.

If a language $W \subseteq (\Sigma - \{\not{c}, \$\})^*$ can be defined by an off-line Turing machine M with one working tape such that $T(M) = W$ and each input string $w = a_2 \cdots a_{n-1}$ in $(\Sigma - \{\not{c}, \$\})^*$ is accepted or rejected with M using at most $L(n)$ working tape squares, then W is said to be of *tape complexity* $L(n)$.

It is well known that the number of working tapes does not affect tape complexity. That is, if a language W can be defined by an off-line

Turing machine with k working tapes such that each input string $w = a_2 \cdots a_{n-1}$ is accepted or rejected by M using no more than $L(n)$ working tape squares on any one working tape, then W can be defined by an off-line Turing machine M' with one working tape, using at most $L(n)$ working tape squares when accepting or rejecting any input string. See, for example, Hopcroft and Ullman (in press).

If a language $W \subseteq (\Sigma - \{\not\epsilon, \$\})^*$ can be defined by an off-line Turing machine with the property that each input string $w = a_2 \cdots a_{n-1}$ in $(\Sigma - \{\not\epsilon, \$\})^*$ is accepted or rejected by having M halt after at most $T(n)$ moves, then W is said to be of *time complexity* $T(n)$.

It has been shown that the number of working tape squares or the number of moves required for a computation by a Turing machine can be reduced by any constant factor by enlarging the working tape symbol alphabet and appropriately recoding the computation (Hartmanis and Stearns, 1965; Stearns, Hartmanis and Lewis, 1965). Thus the class of languages of tape complexity $cL(n)$ (or time complexity $cT(n)$) is equivalent to the class of languages of tape complexity $L(n)$ (or time complexity $T(n)$) where c is any positive constant. This remark, however, does not apply to languages of time complexity $T(n) = kn$ where k is a constant.

We will now show that all 2N PDA languages are of time complexity class n^4. Given a normal form 2N PDA $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, \{q_f\})$ with $\#(Q) = s$ and $\#(\Gamma) = t$, we will outline how a Turing machine M can be constructed to implement Algorithm 1, such that with any input of the form $\not\epsilon w\$ = \not\epsilon a_2 \cdots a_{n-1}\$$, w in $(\Sigma - \{\not\epsilon, \$\})^*$, on the input tape, M will halt after making at most $\sim n^4$ moves and accept w if and only if w is in $N(P)$.

M will have three working tapes. The first two working tapes will each contain a copy of the recognition matrix R. These two tapes are divided into s^2t tracks and each track in each nonblank square contains a symbol denoting an empty track or of the form $[i, (p, Z, q)]$, where i is called the prefix of the element and can have values 0, 1 or 2, and where p and q are in Q, and Z is in Γ. The recognition matrix is represented in the fashion shown in Fig. 3 with the element $r(i, j)$ stored on the $(i - 1)n + j$th square from the left end of the array. s^2t tracks are used since each $r(i, j)$ in R can contain up to s^2t triples. Those triples in R which appear in the pushdown list of Algorithm 1 are written as elements with prefix 0. For example, if A contains the element $(i, j, (p, Z, q))$, then the square representing element $r(i, j)$ of R has the symbol $[0, (p, Z, q)]$ written on

Track	First Component 1				First Component 2				First Component n		
	$r(1,1)$	$r(1,2)$	$\cdots$	$r(1,n)$	$r(2,1)$	$\cdots$	$r(2,n)$	$\cdots$	$r(n,1)$	$\cdots$	$r(n,n)$
1	A_{11}	A_{12}	$\cdots$	A_{1n}	A_{21}	$\cdots$	A_{2n}	$\cdots$	A_{n1}	$\cdots$	A_{nn}
2	B_{11}	B_{12}	$\cdots$	B_{1n}	B_{21}	$\cdots$	B_{2n}	$\cdots$	B_{n1}	$\cdots$	B_{nn}
$\vdots$											
s^2t	C_{11}	C_{12}	$\cdots$	C_{1n}	C_{21}	$\cdots$	C_{2n}	$\cdots$	C_{n1}	$\cdots$	C_{nn}

FIG. 3. Representation of the Recognition Matrix on Working Tapes One and Two.

some track. The element at the top of A will be written with a prefix of 1 and those elements not appearing on A have prefix 2.

The third working tape is a scratch tape used as a counter for numbers between 1 and n.

The specification of the mapping δ of P is encoded directly into the finite state control of M. (A universal 2N PDA language recognizer could also be built by having the Turing machine read the specification of the 2N PDA from an input tape. However, such a universal machine will not be discussed in this paper.) The algorithm itself is also encoded into the finite state control of M.

M is constructed to be capable of executing the following set of composite moves. We leave it to the reader to supply the individual atomic moves whereby M will be capable of executing these and only these sequences of moves.

1. M initially lays out on the first two working tapes n^2 squares with each track of each square containing the symbol $[2, \epsilon]$.
2. Then M computes $r(i, i + d) = \delta_1(i, i + d)$ for all i in J_n and d in $\{-1, 0, 1\}$ such that $1 \leq i + d \leq n$. If $\delta_1(i, i + d)$ contains (p, Z, q), then M enters in an empty track of the square representing $r(i, i + d)$ on working tapes one and two the element $[0, (p, Z, q)]$.
3. M locates the leftmost square on the first working tape containing an element of the form $[0, (p, Z, q)]$. If no such element exists, M consults the square containing $r(1, n)$ to determine whether some track of this square contains the element [2,

(q_0, Z_0, q_f)]. If so, M accepts w; otherwise, M rejects w. Then, M halts.

However, if an element of the form [0, (p, Z, q)] is found in some square containing entry $r(i, j)$ of R, then M changes the prefix of this element to 1.

4. Then, for $d = -1, 0, 1$, such that $1 \leq i - d \leq n$, and for $k = 1, 2, \cdots, n$, M computes $r(i - d, k) = \Theta(\delta_2(i - d, i), \{(p, Z, q)\}, r(j, k))$. To perform this computation, M stores the value of the triple (p, Z, q) in its finite control and positions its first and second working tape heads over the squares containing $r(i - d, 1)$ and $r(j, 1)$, respectively. M then moves its first and second working tape heads synchronously n squares to the right. For $k = 1, 2, \cdots, n$, M reads the values of $r(j, k)$ from the second working tape and stores in an empty track of the first working tape each new triple (p', Z', q') in $r(i - d, k)$ as an element of the form [0, (p', Z', q')]. Once M has computed $r(i - d, n)$, M copies working tape one on to working tape two and then repeats this computation for the next value of d.
5. Once M has completed the computation in step 4, M locates the element [1, (p, Z, q)] in $r(i, j)$ on the first working tape and changes the prefix of the element from 1 to 2.
6. Then, for $d = -1, 0$, and 1 and for $h = 1, 2, \cdots, n$, such that $1 \leq h + d \leq n$, M computes $r(h, j) = \Theta(\delta_2(h, h + d), r(h + d, i), \{(p, Z, q)\})$. Again, M reads the values of $r(h + d, i)$ from the second working tape and enters into square $r(h, j)$ of the first working tape those elements containing new triples not in $r(h, j)$. Each new element thus entered is written with prefix 0. Once M has computed $r(n, j)$, M copies the contents of the first working tape onto the second.
7. M then repeats the computation starting from step 3.

No composite move above requires more than $\sim n^2$ atomic moves by M for its execution. Since computations 3, 4, 5 and 6 need to be repeated at most s^2tn^2 times, the total number of moves made by M in executing Algorithm 1 is $\sim n^4$.

Moreover, the maximum number of nonblank squares used on any working tape is n^2. Thus, we have the following results.

THEOREM 2. *The class of* 2N PDA *languages is contained within the class of languages of time complexity* n^4.

THEOREM 3. *The class of* 2N PDA *languages is contained within the class of languages of tape complexity* n^2.

VI. TIME COMPLEXITY OF 2D PDA LANGUAGES

In this section we will show how a Turing machine M_d can be effectively constructed to implement Algorithm 1 for a 2D PDA $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, \{q_f\})$,[4] with $\#(Q) = s$ and $\#(\Gamma) = t$, such that given an input string $w = a_2 \cdots a_{n-1}$, M_d will halt and decide whether w is in L after making at most $\sim n^2 \log n$ moves.

M_d uses two working tapes. The first working tape is used to store the recognition matrix, and the second is used as a scratch tape.

The first working tape is divided into st tracks and each track is further subdivided into an upper and lower subtrack. Each upper subtrack in each square contains either a blank, an element of the form $[2, \epsilon]$ or an element of the form $[k, (p, Z, q)]$ with $k = 0$, 1 or 2, p and q in Q, and Z in Γ. Each square of the lower subtrack contains either a 0 or a 1. The recognition matrix R is laid out on this first working tape in n blocks each of $\log n$ squares.[5] We will denote these blocks by $R(i)$ for $1 \leq i \leq n$.

If $r(i, j)$ contains (p, Z, q), then one track of $R(i)$ contains this information with element $[k, (p, Z, q)]$, $k = 0$, 1 or 2, written on the first square of the upper subtrack and with the binary representation of j on $\log n$ squares of the lower subtrack. The remaining $\log n - 1$ squares on the upper subtrack are filled with blanks.

Track		$R(1)$				$R(2)$		$\cdots$	$R(n)$
1	Upper	X	B	$\cdots$	B				
	Lower	ϕ	ϕ		ϕ				
2	Upper	Y	B	$\cdots$	B				
	Lower	ϕ	ϕ		ϕ				
st	Upper	Z	B	$\cdots$	B				
	Lower	ϕ	ϕ		ϕ				
		$\log n$ squares				$\log n$ squares			$\log n$ squares

FIG. 4. Representation of the Recognition Matrix on the First Working Tape of M_d. Here $\phi = 0$ or 1.

[4] Actually P can be any uniquely erasing 2N PDA.

[5] By "$\log n$" we mean the smallest integer greater than or equal to $\log n$.

Since the total number of triples in entries $r(i, 1)$ to $r(i, n)$ cannot exceed st for any i between 1 and n, there is sufficient space in $R(i)$ to store all triples in $r(i, 1)$ to $r(i, n)$.

Apart from the representation of the R matrix, M_d works in much the same way as M did for the 2N PDA. The encoding of the 2D PDA P is contained in the finite state control. M_d is constructed to execute the following sequences of composite moves. Again, we will leave it to the reader to supply the necessary atomic moves for M_d.

1. M_d initially lays out n blocks of length $\log n$ on the first working tape. The first square of the upper subtrack of each track in each block is filled with the element $[2, \epsilon]$. The remaining $\log n - 1$ squares on each upper subtrack of each block are filled with blanks. Each square of each lower subtrack is filled with 0.
2. M_d then computes $r(i, i + d) = \delta_1(i, i + d)$ and stores the appropriate values in $R(i)$ for $1 \leq i, i + d \leq n$. For example, if $\delta_1(i, i + d)$ contains the triple (p, Z, q), then the element $[0, (p, Z, q)]$ is overprinted in the leftmost square of the first upper subtrack of $R(i)$ containing $[2, \epsilon]$. In the lower subtrack of this track in $R(i)$ is written the binary representation of $i + d$.
3. M_d locates the first block $R(i)$ from the leftmost nonblank containing an element of the form $[0, (p, Z, q)]$ on the upper subtrack of some track. If no such element exists, then M_d locates block $R(1)$ and accepts the input string $w = a_2 \cdots a_{n-1}$ if and only if some track in $R(1)$ contains the element $[2, (q_0, Z_0, q_f)]$ on the upper subtrack and the binary representation of n on the lower subtrack. Then, M_d halts.
4. If, however, M_d locates an element of the form $[0, (p, Z, q)]$ on the upper subtrack of a track in some block $R(i)$, then M_d stores on the second working tape the value of j written on the lower subtrack of this track. M_d changes the prefix of the element in the first square on the upper subtrack in $R(i)$ to 1 and stores the value of (p, Z, q) in its finite state control.
5. Then M_d consults block $R(j)$ on the first working tape and computes $r(i - d, j) = \Theta(\delta_2(i - d, i), \{(p, Z, q)\}, r(j, k))$ for $d = -1, 0, +1$ and $k = 1, 2, \cdots, n$ provided that $1 \leq i - d \leq n$. Each triple in $r(i - d, k)$ is temporarily stored on the second working tape along with the value of d and k. Since the total number of triples in $r(i - d, k)$ for $d = -1, 0, 1$ and $k = 1, 2, \cdots, n$ does not exceed $3st$, no more than $\sim n$ squares are required on

the second working tape. Then M_d consults block $R(i - d)$ on the first working tape and stores in this block any new triple (p', Z', q') not currently in $r(i - d, k)$ by replacing an element of the form $[2, \epsilon]$ in the upper subtrack of some track in $R(i - d)$ with the symbol $[0, (p', Z', q')]$ and by writing the binary representation of k on the lower subtrack of the same track.

6. M_d locates the element $[1, (p, Z, q)]$ written on the upper subtrack of a track in $R(i)$. M_d changes the prefix of this element to 2, and computes $r(h, j) = \Theta(\delta_2(h, h + d), r(h + d, i), \{(p, Z, q)\})$ for $d = -1, 0, 1$ and $h = 1, 2, \cdots, n$ storing new triples in $r(h, j)$ in block $R(h)$ in much the same manner as outlined above.
7. M_d repeats the computation starting at 3.

M_d takes $\sim n \log n$ atomic moves to do step 1 of this computation. Step 2 requires $\sim n \log n$ moves, since $r(i, i + d)$ can contain at most st triples, and to encode $i + d$ into binary requires $\sim(i + d)$ atomic moves. Steps 3–6 each require no more than $\sim n \log n$ moves of M_d. Since there can be at most stn elements added to R, steps 3–6 are repeated at most stn times. Thus, M_d will halt after making no more than $\sim n^2 \log n$ moves. We have:

THEOREM 4. *All* 2D PDA *languages are contained within the class of languages of time complexity* $T(n) = n^2 \log n$.

Since the 2D PDA $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, \{q_f\})$, with $\#(Q) = s$ and $\#(\Gamma) = t$, and with input string $w = a_2 \cdots a_{n-1}$ can never have at any one time more than $\sim n$ tape symbols on its pushdown tape and yet accept w, the 2D PDA languages are of tape complexity class no greater than n (Stearns, Hartmanis and Lewis, 1965; Gray, Harrison and Ibarra, 1967).

The class of context-free languages (i.e., the 1N PDA languages) have been shown to be of time complexity class no greater than n^3 (Younger,

TABLE 1

Language	Complexity Class		Time On A Random Access Computer
	Tape	Time	
2N PDA	n^2	n^4	$\sim n^3$
2*D* PDA	n	$n^2 \log n$	$\sim n^2$
1*N* PDA	$(\log n)^2$	n^3	$\sim n^3$
1*D* PDA	$(\log n)^2$	$2n$	$\sim n$

1967; Earley, 1967) and of tape complexity class no greater than $(\log n)^2$ (Lewis, Stearns, Hartmanis, 1965).

In Table 1 we summarize the best known time and tape bounds for the four classes of PDA languages.

It is not known whether any of the bounds in this table are exact. In fact, at present there is no example of a context-free language which requires more than cn time for recognition, for a positive constant c, or which requires more than $\log n$ space for recognition.

A significant result would be to further improve any bound in Table 1 or to show that a particular bound is exact.

RECEIVED: November 16, 1967

REFERENCES

CHOMSKY, N. (1962), Context-free grammars and pushdown storage. Quarterly Progress Report No. 65, Research Laboratory of Electronics, Massachusetts Institute of Technology.

EARLEY, J. (1967), An n^2-recognizer for context free grammars. Department of Computer Science, Carnegie-Mellon University.

GINSBURG, S. AND GREIBACH, S. A. (1966), Deterministic context-free languages. *Inform. Control* **9**, 620–648.

GRAY, J. N., HARRISON, M. A. AND IBARRA, O. (1967), Two-way pushdown automata. *Inform. Control* **11**, 30–70.

HARTMANIS, J. AND STEARNS, R. E. (1965), On the computational complexity of algorithms. *Trans. Am. Math. Soc.* **117**, 285–306.

HOPCROFT, J. E. AND ULLMAN, J. D., "The Theory of Languages and Their Relation to Automata." Addison-Wesley, Reading, Massachusetts. In press.

KNUTH, D. E. (1965), On the translation of languages from left to right. *Inform. Control* **8**, 607–639.

LEWIS, P. M., II, STEARNS, R. E., AND HARTMANIS, J. (1965), Memory bounds for recognition of context-free and context sensitive languages. *1965 IEEE Conf. Record Switching Circ. Theory and Log. Des.*, pp. 191–202.

STEARNS, R. E., HARTMANIS, J., AND LEWIS, P. M., II (1965), Hierarchies of memory limited computations. *1965 IEEE Conf. Record Switching Circ. Theory and Log. Des.*, pp. 179–190.

YOUNGER, D. (1967), Recognition and parsing of context free languages in time n^3. *Inform. Control* **10**, 189–208.