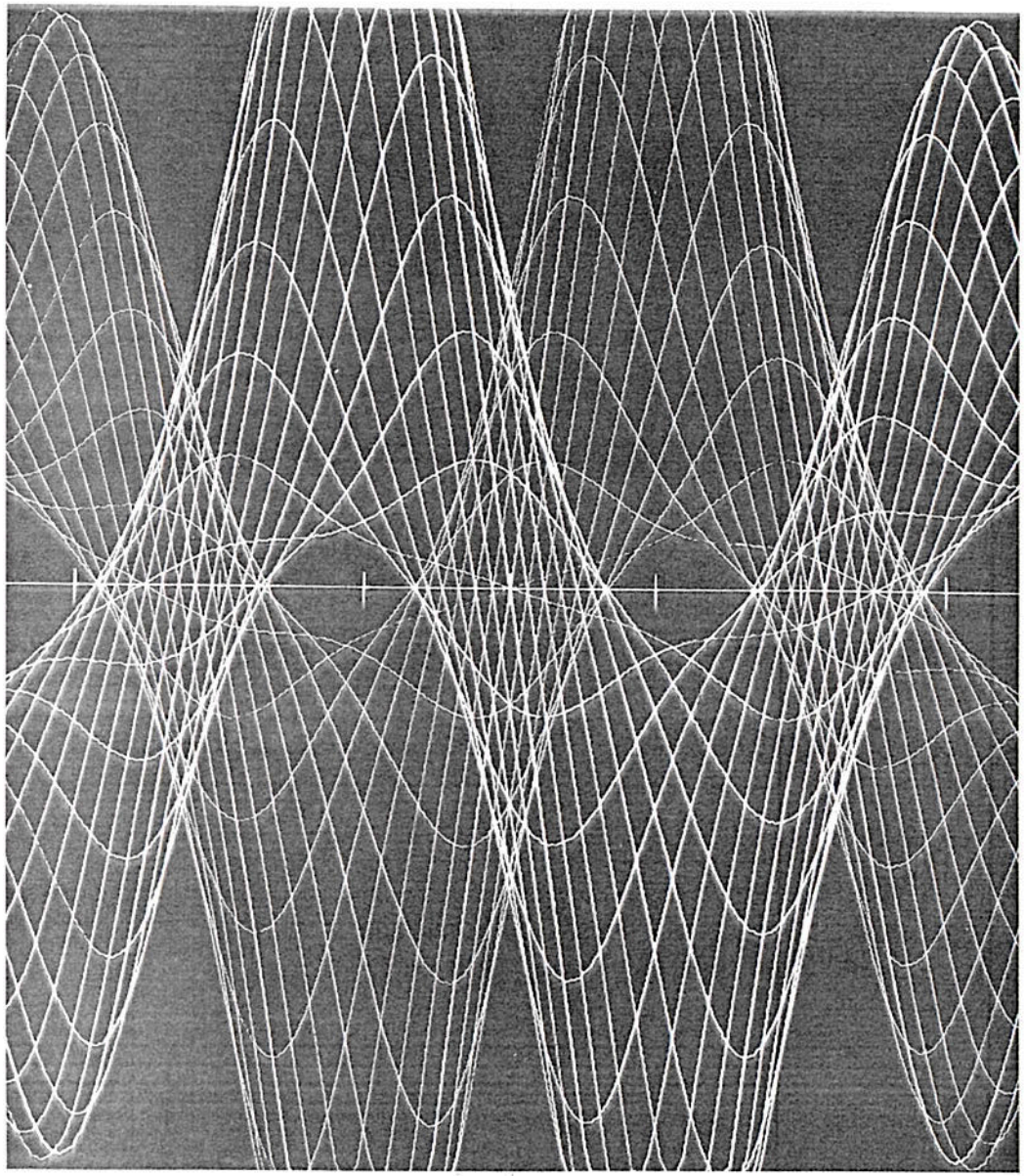
Available via Cyber Digest



RC 1817

COMPUTATIONAL COMPLEXITY AND PROGRAM STRUCTURE

A. R. Meyer/ D. M. Ritchie

May 15, 1967

IBM RESEARCH

COMPUTATIONAL COMPLEXITY AND PROGRAM STRUCTURE*

Albert R. Meyer†
IBM Watson Research Center
Yorktown Heights, New York

Dennis M. Ritchie
Aiken Computation Laboratory
Harvard University
Cambridge, Massachusetts

ABSTRACT: Loop programs have the property that an upper bound on the running time of a program is determined by its structure. Each program consists only of assignment and iteration (loop) statements, but all of the arithmetic functions commonly encountered in digital computation can be computed by Loop programs. A simple procedure for bounding the running time is shown to be best possible; some programs actually achieve the bound, and it is effectively undecidable whether a program runs faster than the bound. The complexity of functions can be measured by the loop structure of programs which compute them. The functions computable by Loop programs are precisely the primitive recursive functions.

Research Paper
RC-1817
May 15, 1967

---

* This research was supported in part by NSF GP-2880 under Professor P. C. Fischer, and by the Division of Engineering and Applied Physics, Harvard University.

† Currently at the Aiken Computation Laboratory, Harvard University, Cambridge, Massachusetts.

1. INTRODUCTION

Predicting how long a digital computer program will require to process given inputs is sometimes impossibly difficult. This difficulty can be partially explained as a reflection of the theorem that there is no effective method for bounding the computation time of a Turing machine from inspection of its program, or for bounding the computation time of a program in any language capable of describing all recursive functions.

In this paper we describe a class of programs, called Loop programs, which do have the property that the computation time of any program can be bounded by a particularly simple function of its inputs, and that moreover the bounding function is itself determined in a simple way from the structure of the program. The study of classes of programs and functions with this property is a constant theme throughout automata theory and the related area which Cobham [C] has christened "meta-numerical analysis," notably including Rabin and Scott's paper on finite automata [RS] and R. W. Ritchie's* work on predictably computable functions [R].

Necessarily the class, $\mathcal{K}$, of functions computable by Loop programs is a proper subset of the recursive functions. We ultimately show that $\mathcal{K}$ is precisely the class of primitive recursive functions.

Loop programs are designed to exploit the power of a construction familiar in higher-level languages; the LOOP instruction of Loop programs is analogous to the DO statement of Fortran and to special cases of the FOR and THROUGH statements of Algol and MAD. We show that the running time of a Loop program for a given input is determined essentially by the depth of nesting of its Loop instructions.

The theorems in the paper are of primarily theoretical interest; the functions with which we are concerned are almost wholly beyond the computational capacity of any real device. We believe that readers with an orientation toward practical programming may nevertheless find some of the results provacative.

2. Loop programs. A Loop program is a finite sequence of instructions for manipulating non-negative integers stored in registers. There is no limit to the size of an integer which may be stored in a register, nor any limit to the number of registers to which a program may refer, although any given program will refer to only a finite number of registers. Throughout this paper upper case English letters will be used as register names, and a sequence of register names $X_1, X_2, \ldots, X_m$ will be abbreviated as $\overline{X}_m$. We let $\mathbf{N}$ be the non-negative integers and abbreviate a sequence of non-negative integers $x_1, \ldots, x_m$ as $\overline{x}_m$. Boldface letters stand for programs, and if $\underset{\sim}{P}$ is a program Reg ($\underset{\sim}{P}$) will be the set of register names appearing in $\underset{\sim}{P}$.

Instructions are of five types: (1) X = Y, (2) X = X + 1, (3) X = 0, (4) LOOP X, (5) END, where "X" and "Y" may be replaced by any names for registers.

---

*The second author is unrelated to R. W. Ritchie

2.1 Definition. The class L of Loop programs is $\bigcup_{n=0}^{\infty} L_n$ where the classes L_n are given by

(i) L_0 is the class of finite sequences of type (1), (2), and (3) instructions,

(ii) $L_{n+1} \supseteq L_n$,

(iii) If $\underset{\sim}{Q}$, $\underset{\sim}{R}$, $\in L_{n+1}$ and $\underset{\sim}{P}$ is $\underset{\sim}{Q}$ concatenated with $\underset{\sim}{R}$ then $\underset{\sim}{P} \in L_{n+1}$,

(iv) If $\underset{\sim}{Q} \in L_n$, and $\underset{\sim}{P}$ is a type (4) instruction concatenated with $\underset{\sim}{Q}$ concatenated with a type (5) instruction, then $\underset{\sim}{P} \in L_{n+1}$,

(v) The only members of L_{n+1} are those implied by clauses (ii) - (iv).

The first three types of instructions have the same interpretation as in several common languages for programming digital computers. "X = Y" means that the integer contained in Y is to be copied into X; previous contents of X disappear, but the contents of Y remain unchanged. "X = X + 1" means that the integer in X is to be incremented by one. "X = 0" means that the contents of X are to be set to zero. These are the only instructions which affect the registers.

The instructions in a Loop program are normally executed sequentially in the order in which they occur in the program. Type (4) and (5) instructions affect the normal order by indicating that a block of instructions is to be repeated. Specifically if $\underset{\sim}{P}$ is a Loop program, and the integer in X is x, then "LOOP X, $\underset{\sim}{P}$, END" means that $\underset{\sim}{P}$ is to be performed x times in succession before the next instruction, if any, after the END is executed; changes in the contents of X while $\underset{\sim}{P}$ is being repeated do not alter the number of times $\underset{\sim}{P}$ is to be repeated. The final clause is needed to ensure that executions of Loop programs always terminate. For example, the program

(2.2)

```
LOOP X
X = X + 1
END
```

is a program for doubling the contents of X, rather than an infinite loop. Note that when X initially contains zero the second instruction is not executed.

By (2.1), type (4) and type (5) instructions occur in matched pairs like left and right parentheses, so that the block of instructions affected by a type (4) instruction is itself a Loop program and is unambiguously determined by the matching type (5) instruction. A program is in L_n if and only if the LOOP-END pairs are nested to a depth of at most n. For example, in the L_2 program

(2.3)

```
LOOP Y
  A = 0
  LOOP X
    X = A
    A = A + 1
  END
END
```

type (4) and (5) instructions are paired as indicated by the indentations. If X and Y initially contain x and y, execution of (2.3) would leave $x \dot{-} y$ in X, where $x \dot{-} y$ equals

$x - y$ if $x \geq y$, and is zero otherwise.

Given the initial contents of Reg (P), the running time of a Loop program P will be measured by the number of individual instruction executions required to execute P. It should be clear how to count the number of times instructions of types (1), (2), and (3) are executed, but some further interpretation is needed for instructions of types (4) and (5). Associate with each LOOP-END pair in P a special register distinct from Reg (P). Execution of "LOOP X" places the integer in X into the special register, and the next instruction to be executed is the matching END instruction. Execution of "END" tests the associated special register for zero. If the special register contains zero, the next instruction to be executed is the one immediately following. If the special register does not contain zero, its contents are decremented by one, and the next instruction to be executed is the one immediately following the matching type (4) instruction. For example, if initially X contains $x \in \mathbb{N}$, execution of program (2.2) requires one execution of "LOOP X", x executions of "X = X + 1", and $x + 1$ executions of "END", so that the running time is $2x + 2$.

(2.4) Definition. Let P be a Loop program and $X_1, \ldots, X_m$ be the members of Reg (P) in order of appearance in P. The value $T_P(\bar{x}_m)$ of the function $T_P: \mathbb{N}^m \to \mathbb{N}$ equals the running time of P when X_i initially contains x_i, $1 \leq i \leq m$, providing this number is finite.

A formal definition of execution of a Loop program and of running time can be given, though we shall not do so. In the next section we show that T_P is always defined.

3. Bounds on running time. We now describe exactly what is meant by the claim that the running time of a Loop program can be bounded by inspection of the form of the program.

(3.1) Definition. If $g: \mathbb{N} \to \mathbb{N}$, the function $h: \mathbb{N}^2 \to \mathbb{N}$ is called the iterate of g providing

$$h(z, 0) = z$$
$$h(z, y + 1) = g(h(z, y)).$$

The iterate $h(z, y)$ is also written as $g^{(y)}(z)$.

Thus, $g^{(y)}(z) = g(g(\ldots g(z)\ldots))$, the composition being taken y times.

(3.2) Definition. For $n \in \mathbb{N}$, the functions $f_n: \mathbb{N} \to \mathbb{N}$ are defined by the equations

$$f_0(0) = 1,$$
$$f_0(1) = 2,$$
$$f_0(x) = x + 2 \text{ for } x > 1,$$
$$f_{n+1}(x) = f_n^{(x)}(1).$$

We will say that $g: \mathbb{N}^m \to \mathbb{N}$ is bounded by $f: \mathbb{N} \to \mathbb{N}$ whenever $g(\bar{x}_m) < f(\max\{\bar{x}_m\})$ for all $\bar{x}_m \in \mathbb{N}$; max $\{\bar{x}_m\}$ is the largest member of $\{\bar{x}_m\}$.

(3.3) Bounding Theorem. Let P be a program in L_n. Then there is a $p > 0$, which can be found effectively from P, such that $f_n^{(p)}$ bounds the running time of P.

$T_{\underset{\sim}{P}}(\bar{x}_m)$ is defined whenever execution of $\underset{\sim}{P}$ with $\underline{\text{Reg}}\ (\underset{\sim}{P})$ initially containing $\bar{x}_m$ can be completed in a finite number of instruction executions. Moreover, $T_{\underset{\sim}{P}}$ can be computed effectively by executing $\underset{\sim}{P}$ and counting individual instruction executions, provided that every execution of $\underset{\sim}{P}$ eventually can be completed. It follows that when we have proved that $T_{\underset{\sim}{P}}$ is bounded by $f_n^{(p)}$, we will have proved (3.3), and shown incidentally that $T_{\underset{\sim}{P}}$ is effectively computable.

This raises an interesting point about bounding the running time of Loop programs by inspection. It is easy to prove that $T_{\underset{\sim}{P}}$ is totally defined, and hence effectively computable, without introducing the functions f_n. Accordingly, the assertion that Loop programs can be bounded $\underline{\text{a priori}}$ becomes trivial--one can always bound $\underset{\sim}{P}$ by $T_{\underset{\sim}{P}}$.

Of course bounding the running time of $\underset{\sim}{P}$ by $T_{\underset{\sim}{P}}$ is not very informative, for it amounts to "predicting" that $\underset{\sim}{P}$ will run as long as it runs. One would at least expect bounding functions which are in some sense sufficiently comprehensible that they provide more information than the previous tautology. An inevitable difficulty is that bounding functions must grow at such extraordinary rates that their sizes can hardly be called comprehensible. Nevertheless, a function $f_n^{(p)}$ is a more satisfactory bound than $T_{\underset{\sim}{P}}$ on intuitive as well as theoretical grounds, as indicated by the next two lemmas (see also (5.3)).

(3.4) Lemma. For all $x, p \in \mathbb{N}$

(i) $f_1(x) = 2x + (1 \dot{-} x)$

(ii) $f_1^{(p+1)}(x) = 2^p \cdot f_1(x) \geq 2^{p+1} \cdot x$

(iii) $f_2(x) = 2^x$

The proof of (3.4) is left to the reader.

The function $f_3(x)$ is also easy to describe;

$$f_3(x) = 2^{2^{\cdot^{\cdot^{\cdot^{2}}}}} \Big\} \text{ height } x \ .$$

(3.5) Lemma. For all $n, p, x \in \mathbb{N}$,

(i) $f_n(x) \geq x + 1$,

(ii) $f_n^{(p)}(x)$ is nondecreasing in n and increasing in p, x,

(iii) $2 \cdot f_n^{(p)}(x) \leq f_n^{(p+1)}(x)$ for $n \geq 1$,

(iv) $[f_n^{(p)}(x)]^2 \leq f_n^{(p+2)}(x)$ for $n \geq 2$.

The lemma can be proved by induction; we omit the details.

Proof of the bounding theorem: The proof is by induction on n and definition (2.1).

Let $\underset{\sim}{P}$ be a program in L_n with k registers, and let $m = \underline{\max}\ \{\bar{x}_k\}$, where $\bar{x}_k$ are the integers initially in $\underline{\text{Reg}}\ (\underset{\sim}{P})$.

If $n = 0$, then $\underset{\sim}{P}$ has no loops and so $T_{\underset{\sim}{P}}$ is identically equal to the length of $\underset{\sim}{P}$. Let $p > 0$ equal the length of $\underset{\sim}{P}$; then

$$T_{\underset{\sim}{P}}(\bar{x}_k) = p < f_0^{(p)}(0) \leq f_0^{(p)}(m) \ .$$

If $n > 0$, assume that (3.3) is true for $\underset{\sim}{P} \in L_{n-1}$. If $\underset{\sim}{P} \in L_n$ by (2.1.ii) so that $\underset{\sim}{P} \in L_{n-1}$, then $T_{\underset{\sim}{P}}$ is bounded by $f_{n-1}^{(p)}$ for some $p \in \mathbb{N}$. By (3.5.ii) $T_{\underset{\sim}{P}}$ is also bounded by $f_n^{(p)}$.

Now, suppose that $\underset{\sim}{P} \in L_n$ by (2.1.iv), so that $\underset{\sim}{P}$ equals "LOOP X" concatenated with $\underset{\sim}{Q} \in L_{n-1}$ concatenated with "END". We separate the cases $n = 1$, $n > 1$.

If $n = 1$, then $T_{\underset{\sim}{Q}}$ is identically equal to $q > 0$. The running time of $\underset{\sim}{P}$ is at most $1 + q \cdot x + (x+1)$ when X initially contains x, by the same argument used for program (2.2). Since $x \leq m$,

$$T_{\underset{\sim}{P}}(\bar{x}_k) \leq (q+1) \cdot m + 2 \leq 2^q \cdot m + 2 \leq f_1^{(q)}(m) + 2 \leq f_1^{(q+2)}(m).$$

If $n > 1$, assume that $T_{\underset{\sim}{Q}}$ is bounded by $f_{n-1}^{(q)}$ for $q \in \mathbb{N}$. After one execution of $\underset{\sim}{Q}$ the largest integer in any register is $f_{n-1}^{(q)}(m) + m \leq f_{n-1}^{(q+1)}(m)$ because each instruction execution can increase the largest integer in the registers by at most one. If $\underset{\sim}{Q}$ is now repeated because of the loop, it requires at most $f_{n-1}^{(q)}(f_{n-1}^{(q+1)}(m))$ instruction executions and leaves at most

$$f_{n-1}^{(q)}(f_{n-1}^{(q+1)}(m)) + f_{n-1}^{(q+1)}(m) < f_{n-1}^{(2\cdot(q+1))}(m)$$

in any register. An obvious induction implies that the i^{th} repetition of $\underset{\sim}{Q}$ requires at most $f_{n-1}^{(i\cdot(q+1))}(m)$ instruction executions. Therefore,

$$T_{\underset{\sim}{P}}(\bar{x}_k) \leq 1 + \sum_{i=1}^{m} f_{n-1}^{(i(q+1))}(m) + (m+1) \leq 2 + m + m \cdot f_{n-1}^{(m\cdot(q+1))}(m).$$

But by (3.5) and the definition of f_n,

$$f_{n-1}^{(m\cdot(q+1))}(m) \leq f_{n-1}^{(m\cdot(q+1))}(f_n(m)) = f_n(m\cdot(q+2)),$$

and further calculation using (3.5), especially (3.5.iv) since $n \geq 2$, implies

$$T_{\underset{\sim}{P}}(\bar{x}_k) \leq f_n^{(q+5)}(m).$$

The one remaining case is that $\underset{\sim}{P} \in L_n$ by (2.1.iii), so that $\underset{\sim}{P}$ is $\underset{\sim}{Q}$ concatenated with $\underset{\sim}{R}$, where $\underset{\sim}{Q}, \underset{\sim}{R} \in L_n$. The proof for this case is similar to the previous one and is left to the reader.

This completes the proof that every $\underset{\sim}{P} \in L_n$ is bounded by $f_n^{(p)}$ for some $p \in \mathbb{N}$. Moreover, by hypothesis the integers q, r mentioned in the proof can be found effectively, so that p can also be found effectively. Choosing p equal to six times the length of $\underset{\sim}{P}$ is sufficient.

4. Functions computable by Loop programs. If a set of registers is designated for storing input and output, a Loop program describes the computation of a function.

(4.1) Definition. Let $\bar{X}_m$ be distinct register names, $m > 0$, and P be a register name which need not be distinct from $\bar{X}_m$. If $\underset{\sim}{P}$ is a Loop program, the $m+2$-tuple $\langle \underset{\sim}{P}, \bar{X}_m, P \rangle$ will be called a program with input and output, $\bar{X}_m$ being the input registers and P the output register. The function $f: \mathbb{N}^m \to \mathbb{N}$ is computed by $\langle \underset{\sim}{P}, \bar{X}_m, P \rangle$ providing that $f(\bar{x}_m)$ equals the contents of P after execution of $\underset{\sim}{P}$ when X_i initially contains x_i, $1 \leq i \leq m$, and all other members of

Reg ($\underset{\sim}{P}$) initially contain zero.

For example, if $\underset{\sim}{P}$ is the program (2.3), then $\langle \underset{\sim}{P}, X, Y, X \rangle$ computes $x \dot{-} y$, and $\langle \underset{\sim}{P}, X, Y, Y \rangle$ computes the projection $p_{22}(x, y) = y$.

(4.2) Definition. $\mathcal{L}_n$ is the set of functions computable by programs in L_n with input and output. $\mathcal{L} = \bigcup_{n=0}^{\infty} \mathcal{L}_n$.

Obviously, $\mathcal{L}_{n+1} \supseteq \mathcal{L}_n$ for all $n \in \mathbb{N}$. As we remarked in the introduction, Loop programs cannot compute all the functions which are formally computable, but they are nevertheless extremely powerful.

The full power of Loop programs will be revealed as we prove that $\mathcal{L}$ includes a class of functions familiar in recursive function theory: the primitive recursive functions.

(4.3) Definition. For $1 \leq i \leq m$ let $p_{im} : \mathbb{N}^m \to \mathbb{N}$ be the projection on the i^{th} coordinate: $p_{im}(\bar{x}_m) = x_i$. The class $\mathcal{P}$ of primitive recursive functions is the smallest class satisfying

(i) $\mathcal{P}$ contains the projection functions, the successor function $(p_{11}+1)$, and the function of one variable identically equal to zero.

(ii) If $h: \mathbb{N}^m \to \mathbb{N}$ and $g: \mathbb{N}^n \to \mathbb{N}$ are in $\mathcal{P}$, then $f: \mathbb{N}^{m+n-1} \to \mathbb{N}$ is in $\mathcal{P}$ where

$$f(\bar{x}_{m+n+1}) = h(\bar{x}_{m-1}, g(x_m, \ldots, x_{m+n-1})),$$

(iii) If $g: \mathbb{N}^m \to \mathbb{N}$ is in $\mathcal{P}$ and $\pi: \{1, \ldots, m\} \to \{1, \ldots, k\}$, then $f: \mathbb{N}^k \to \mathbb{N}$ is in $\mathcal{P}$ where

$$f(\bar{x}_k) = g(p_{\pi(1), k}(\bar{x}_k), \ldots, p_{\pi(m), k}(\bar{x}_k)),$$

(iv) If $g: \mathbb{N}^m \to \mathbb{N}$ and $h: \mathbb{N}^{m+2} \to \mathbb{N}$ are in $\mathcal{P}$, then $f: \mathbb{N}^{m+1} \to \mathbb{N}$ is in $\mathcal{P}$ where

$$f(\bar{x}_m, 0) = g(\bar{x}_m)$$

$$f(\bar{x}_m, y+1) = h(\bar{x}_m, y, f(\bar{x}_m, y)).$$

The fourth clause of (4.3) is the standard scheme for defining f from g and h by primitive recursion.

The second and third clauses of (4.3) together imply that $\mathcal{P}$ is closed under composition of functions, permutation of variables, identification of variables (e.g., obtaining $h(x) = f(x, x)$ from $f(x, y)$), and by (4.3.i) substitution of constants, all of which we will heceforth call the operations of substitution.

It is well-known that $\mathcal{P}$ includes all the functions normally encountered in digital computation and number theory. Addition, multiplication, and exponentiation are all members of $\mathcal{P}$, as are functions yielding the decimal expansion of $\sin(x)$, the x^{th} prime number, etc. In fact, it requires a careful analysis of the definition of $\mathcal{P}$ to find a function which is not primitive recursive.

(4.4) Theorem. Every primitive recursive function is computed by some Loop program with input and output.

Proof: The proof is by induction on definition (4.3).

It is obvious from the definition of type (1), (2), and (3) instructions that the functions which are in $\mathcal{P}$ by (4.3.i) are also in $\mathcal{L}$.

We now prove that $\mathcal{L}$ satisfies (4.3.ii)-(4.3.iv) by combining various programs, and this will imply that $\mathcal{L} \supseteq \mathcal{P}$. In order to prevent different programs from interacting improperly because of common register names, extend the function Reg to programs with input and output by letting $\underline{\text{Reg}}\ (\langle \underset{\sim}{P}, \overline{X}_m, P\rangle) = \underline{\text{Reg}}\ (\underset{\sim}{P}) \cup \{\overline{X}_m, P\}$. When the registers of two programs with input and output are disjoint we shall say that the programs are compatible. There is obviously no loss in generality in assuming below that the programs to be combined are compatible.

Suppose f is defined from g, h by (4.3.ii). Let $\langle \underset{\sim}{H}, \overline{X}_{m-1}, Y, H\rangle$ and $\langle \underset{\sim}{G}, X_m, \ldots, X_{m+n-1}, G\rangle$ compute h and g respectively. A program for f is simply $\langle \underset{\sim}{F}, \overline{X}_{m+n-1}, H\rangle$ where $\underset{\sim}{F}$ is

(4.5)
$$\begin{array}{l} \underset{\sim}{G} \\ Y = G \\ \underset{\sim}{H} \end{array}$$

Now suppose f is fefined from g and π: $\{1, \ldots, m\} \to \{1, \ldots, k\}$ by (4.3.iii). Let $\langle \underset{\sim}{G}, \overline{Y}_m, G\rangle$ compute g, and let $\overline{X}_k$ be register names not in $\underline{\text{Reg}}$ $(\langle \underset{\sim}{G}, \overline{Y}_m, G\rangle)$.

Then $\langle \underset{\sim}{F}, \overline{X}_k, G\rangle$ computes f where $\underset{\sim}{F}$ is

(4.6)
$$\begin{array}{l} Y_1 = X_{\pi(1)} \\ \quad \vdots \\ Y_m = X_{\pi(m)} \\ \underset{\sim}{G} \end{array}$$

Finally, suppose that f is defined from g, h by (4.3.iv) and that $\langle \underset{\sim}{G}, \overline{Y}_m, G\rangle$ and $\langle \underset{\sim}{H}, \overline{Z}_m, V, W, H\rangle$ compute g and h respectively. Then $\langle \underset{\sim}{F}, \overline{X}_m, U, F\rangle$ computes f, where $\underset{\sim}{F}$ is

(4.7)
$$\begin{array}{lll}
\underline{\text{Reg}}\ (\underset{\sim}{G}) = 0 & & \text{clear registers of } \underset{\sim}{G} \\
\overline{Y}_m = \overline{X}_m & & \text{set up arguments of } \underset{\sim}{G} \\
\quad \underset{\sim}{G} & & \text{do } \underset{\sim}{G} \\
F = G & & F = f(\overline{x}_m, 0) = g(\overline{x}_m) \\
C = 0 & & \text{clear repetition counter} \\
\text{LOOP } U & & \\
\quad \underline{\text{Reg}}\ (\underset{\sim}{H}) = 0 & & \text{clear registers of } \underset{\sim}{H} \\
\quad \left.\begin{array}{l} \overline{Z}_m = \overline{X}_m \\ V = C \\ W = F \end{array}\right\} & & \text{set up arguments of } \underset{\sim}{H} \\
\quad \underset{\sim}{H} & & \text{do } \underset{\sim}{H} \\
\quad F = H & & F = H(\overline{x}_m, C, F) = f(\overline{x}_m, C+1) \\
\quad C = C+1 & & \text{count repetitions} \\
\text{END} & &
\end{array}$$

The reader can verify that $\underset{\sim}{F}$ behaves as outlined. We have used $\overline{X}_m$, U, F, and C as register names not appearing in the programs with input and output for g and h. The expressions "$\underline{\text{Reg}}$ $(\underset{\sim}{G}) = 0$", "$\overline{Y}_m = \overline{X}_m$", etc., are obvious abbreviations for finite blocks of instructions. This completes the proof of (4.4).

(4.8) Corollary. For $n \geq 0$, $\mathcal{L}_n$ is closed under substitution.

Proof: If $\underset{\sim}{G}, \underset{\sim}{H} \in L_n$ then the programs (4.5) and (4.6) are also in L_n by definition (2.1).

(4.9) Corollary. If g and h are in $\mathcal{L}_{n+1}$ and $\mathcal{L}_n$ respectively, and if f is defined from them by primitive recursion (see (4.3.iv)) then $f \in \mathcal{L}_{n+1}$.

Proof: The depth of nesting of loops in (4.7) is the greater of one plus the depth of $\underset{\sim}{H}$, and the depth of $\underset{\sim}{G}$; if $\underset{\sim}{G} \in L_{n+1}$ and $\underset{\sim}{H} \in L_n$, then the depth in (4.7) is at most n+1.

5. A hierarchy of functions. The family $\{\mathcal{L}_n\}_0^\infty$ forms an infinite hierarchy of sets of functions, namely, $\mathcal{L}_0 \subsetneq \mathcal{L}_1 \subsetneq \mathcal{L}_2 \subsetneq \cdots$. One of the implications of this fact is that depth of loops in a Loop program can be used to classify the complexity of the functions computable by the program.

(5.1) Lemma. If $f \in \mathcal{L}_n$, then f is bounded by $f_n^{(p)}$ for some $p \in \mathbb{N}$.

Proof: The function f is bounded by the running time of any program with input and output which computes f, plus the largest integer initially in the registers of the program. If $n = 0$, this means f is bounded by $x + q$ where $q \in \mathbb{N}$ is the length of a program for f. If $n > 0$, f is bounded by $f_n^{(q)}(x) + x$ for some $q \in \mathbb{N}$. In any case, f is bounded by $f_n^{(p)}$, where $p = q + 1$.

(5.2) Definition. The function $f: \mathbb{N} \to \mathbb{N}$ $\underline{\text{majorizes}}$ the function $g: \mathbb{N} \to \mathbb{N}$ providing $f(x) > g(x)$ for all sufficiently large $x \in \mathbb{N}$.

(5.3) Lemma. For all $n, p \in \mathbb{N}$, f_{n+1} majorizes $f_n^{(p)}$.

Proof: By definition (3.2) if $x \geq 2$, then $f_1(x) = 2x$ and $f_0^{(p)}(x) = x + 2p$, so f_1 majorizes $f_0^{(p)}$ for any fixed p.

If $n > 0$, we proceed by induction on p. By (3.5.i), $f_{n+1}(x) > x = f_n^{(0)}(x)$. Now assume that f_{n+1} majorizes $f_n^{(p)}$;

then

$f_n^{(p+1)}(x) < f_n^{(p+1)}(2\cdot(x-2)) = f_n^{(p+1)}(f_1(x-2))$, for $x \geq 5$ by (3.5), and

$f_n^{(p+1)}(f_1(x-2)) \leq f_n^{(p+1)}(f_n(x-2)) = f_n^{(2)}(f_n^{(p)}(x-2)) < f_n^{(2)}(f_{n+1}(x-2))$,

for large x because f_{n+1} majorizes $f_n^{(p)}$, and

$f_n^{(2)}(f_{n+1}(x-2)) = f_{n+1}(x)$, by definition (3.2).

(5.4) Theorem. For all $n \in \mathbf{N}$, $f_{n+1} \in \mathcal{L}_{n+1} - \mathcal{L}_n$, and

$$\mathcal{L}_{n+1} \supsetneq \mathcal{L}_n.$$

Proof: We need only show that $f_{n+1} \in \mathcal{L}_{n+1}$ and $f_{n+1} \notin \mathcal{L}_n$.

By (5.1) and (5.3), f_{n+1} is a function of one variable which majorizes every function of one variable in $\mathcal{L}_n$, and so f_{n+1} certainly is not in $\mathcal{L}_n$.

It is easy to write an L_1 program which can compute f_1, so $f_1 \in \mathcal{L}_1$. Assuming $f_n \in \mathcal{L}_n$, then the iterate of f_n, namely $f_n^{(y)}(z)$ regarded as a function of y and z, is in $\mathcal{L}_{n+1}$. This follows from corollary (4.9) because iteration, (3.1), is a special case of primitive recursion, (4.3.iv). Therefore, $f_{n+1}(x) = f_n^{(x)}(1)$ is obtained by substitution from a function in $\mathcal{L}_{n+1}$, and so $f_{n+1} \in \mathcal{L}_{n+1}$. By induction, the proof is complete.

Theorem (5.4) is a rigorous verification that depth of loops is a measure of the power of Loop programs. Obviously functions computable with a small number of loops can be computed with additional unnecessary loops, but there are functions, for example f_n, which can be computed by a program with loops nested to depth n but not less.

We also conclude that the bounds of theorem (3.3) are reasonable in the following sense: for every n, $p \in \mathbf{N}$, there is an L_n program whose running time is bounded by no function smaller than $f_n^{(p)}$. For example, we know that $f_n^{(p)}$, for fixed p, is obtained by substitution from f_n. Therefore, $f_n^{(p)} \in \mathcal{L}_n$, and obviously any L_n program which computes $f_n^{(p)}$ requires nearly $f_n^{(p)}$ instruction executions just to leave the answer in its output register.

6. Characterization of $\mathcal{L}_n$ by running time. The bounding theorem of section 3 implies that every function in $\mathcal{L}_n$ can be computed within time $f_n^{(p)}$ for some $p \in \mathbf{N}$. This kind of bound on running time is commonly taken as a rough measure of complexity. Thus, context-free languages are ℓ^3 recognizable, and context-sensitive languages are at worst 2^{2^ℓ} recognizable, where ℓ is the length of an input word.* $\mathcal{L}_n$ can be characterized as precisely the f_n computable functions.

---

*These bounds on running time apply to Turing machines and not Loop programs, but for computations bounded by f_2 or more, the distinction is unnecessary (see [C], [MR]).

(6.1) Theorem. For $n \geq 2$, a function is in $\mathscr{L}_n$ if and only if it can be computed by a Loop program with input and output whose running time is bounded by $f_n^{(p)}$ for some $p \in \mathbb{N}$.

Half of theorem (6.1) follows immediately from the bounding theorem. The other half is proved by showing that if the running time of a Loop program is bounded by $f_n^{(p)}$, then regardless of the actual depth of loops, the program can be rewritten as a program in L_n (providing $n \geq 2$). Specifically, (6.1) follows from

(6.2) Lemma. If $\underset{\sim}{P}$ is a Loop program and $T_{\underset{\sim}{P}}$ is bounded by $f_n^{(p)}$ for some $p \in \mathbb{N}$, $n \geq 2$, then there is an $\underset{\sim}{M} \in L_n$ such that $\langle \underset{\sim}{P}, \bar{X}_{k+2} \rangle$ and $\langle \underset{\sim}{M}, \bar{X}_{k+2} \rangle$ compute the same function for any $\bar{X}_{k+2} \in \underline{\mathrm{Reg}}\,(\underset{\sim}{P})$, $k \in \mathbb{N}$.

Proof: $\underset{\sim}{M}$ will be a "mimicking" program for $\underset{\sim}{P}$. If $\underset{\sim}{P}$ is the sequence of instructions $I_1, I_2, \ldots, I_s$, and $\underline{\mathrm{Reg}}\,(\underset{\sim}{P}) = \{\bar{X}_m\}$, then $\underset{\sim}{M}$ will contain subprograms $\underset{\sim}{I}_1, \underset{\sim}{I}_2, \ldots, \underset{\sim}{I}_s$, and $\underline{\mathrm{Reg}}\,(\underset{\sim}{M})$ will include $\bar{X}_m$, $\bar{A}_s$, $\bar{B}_s$ and $\bar{C}_s$.

Register A_i, $1 \leq i \leq s$, will always contain zero or one. Each $\underset{\sim}{I}_i$, $1 \leq i \leq s$, will be an L_1 program which tests A_i for one. If the test is successful, $\underset{\sim}{I}_i$ has the same effect on $\bar{X}_m$ as execution of I_i in $\underset{\sim}{P}$, and $\underset{\sim}{I}_i$ will set A_i and B_i to zero and B_j to one, where I_j is the next instruction which would be executed in $\underset{\sim}{P}$. If the test is unsuccessful or there is no next instruction, $\underset{\sim}{I}_i$ will have no effect. The registers $\bar{C}_s$ are used to count repetitions of loops.

If we let $\underset{\sim}{M}_2$ be the L_2 program

(6.3)
```
B̄_s = 0
B_1 = B_1 + 1      } flag the first instruction of P
LOOP T
    Ā_s = B̄_s
        I_1
        I_2
        .
        .
        I_s
END
```

then $\langle \underset{\sim}{M}_2, \bar{X}_m, T, X_i \rangle$ will obviously compute the integer left in X_i after t instruction executions in the execution of $\underset{\sim}{P}$, where t is the integer initially in T.

Since $T_{\underset{\sim}{P}}$ is bounded by $f_n^{(p)}$, we need only precede $\underset{\sim}{M}_2$ by a program which places an integer larger than $f_n^{(p)}(m)$ in T, where m is an integer larger than the initial contents of $\bar{X}_m$. It is easy to find an L_n program $\underset{\sim}{M}_n$ which does this, and which also leaves the contents of $\bar{X}_m$ unchanged. Therefore, $\underset{\sim}{M}$ will be the concatenation of $\underset{\sim}{M}_n$ and $\underset{\sim}{M}_2$, and since $n \geq 2$, $\underset{\sim}{M} \in L_n$.

To complete the construction we need only exhibit the L_1 programs $\underset{\sim}{I}_i$.

If I_i is a type (1), (2), or (3) instruction, then $\underset{\sim}{I}_i$ will be

(6.4)

```
LOOP  A_i          test A_i for one
  A_i = 0
  B_i = 0
B_i+1 = B_i+1+1*   flag the next instruction of P
  I_i              execute I_i
END
```

If I_i is the type (4) instruction "LOOP X_j", and the matching type (5) instruction is I_k, then $\underset{\sim}{I}_i$ will be

(6.5)

```
LOOP  A_i          test A_i for one
  A_i = 0
  B_i = 0
  B_k = B_k+1      flag the next instruction of P
  C_i = X_j        load the repetition counter
END
```

Finally, if I_i is a type (5) instruction, and the matching type (4) instruction is I_k, then $\underset{\sim}{I}_i$ will be

(6.6)

```
R = 0  }
S = 0  }  set two flags for the repetition
S = S+1}     counter test
U = C_k          copy the repetition counter
V = 0            clear the auxiliary storage
LOOP U           test the repetition counter for
                    non-zero
  U = V  }
  V = V+1}  U = C_k - 1
  R = 0  }
  R = R+1}  reverse flags if the repetition
  S = 0  }     counter is non-zero
END
LOOP  A_i        test A_i for one
  A_i = 0
  B_i = 0
B_k+1 = R  }
B_i+1 = S* }     flag the next instruction of P
  C_k = U        decrement the repetition counter
END
```

The reader can verify that the programs $\underset{\sim}{I}_i$ behave as outlined, and hence that $\underset{\sim}{M}$ satisfies (6.2).

The proof is complete.

---

* If i = s, omit this instruction.

---

* If i = s, omit this instruction.

7. The complexity problem. The running time of an L_n program is bounded by $f_n^{(p)}$ for some p, and there are L_n programs that actually run that long. On the other hand, we have just considered L_n programs whose running times can be bounded by $f_m^{(p)}$ for $m < n$. This naturally suggests the question: can one tell if a program runs more rapidly than its loop structure indicates? The answer is no.

(7.1) Definition. The complexity problem for Loop programs is: given $\underset{\sim}{P} \in L_n$, determine whether the running time of $\underset{\sim}{P}$ is bounded by $f_{n-1}^{(p)}$ for any $p \in \mathbb{N}$.

(7.2) Theorem. There is no effective procedure for solving the complexity problem for Loop programs.

In order to prove (7.2), we shall have to appeal to two familiar results from the theory of computability. First the halting problem for Turing machines is effectively undecidable; second, there is a primitive recursive function $t: \mathbb{N}^2 \to \mathbb{N}$ such that $t(e,x) = 0$ if and only if the Turing machine with Gödel number e halts on input e in fewer than x steps, and when $t(e,x) \neq 0$, then $t(e,x) = x$, (cf. [D], [K]).

Proof: We show that if the complexity problem had an effective solution, then so would the halting problem for Turing machines.

Since $\mathcal{L} \supseteq \mathcal{P}$, the function t is in $\mathcal{L}_{n_0}$ for some n_0. Given the L_{n_0} program for t, and given an integer e, it should be obvious that a program $\underset{\sim}{T}_e \in L_{n_0}$ can be constructed with the following property: when a designated register of $\underset{\sim}{T}_e$, say register X, initially contains x, then execution of $\underset{\sim}{T}_e$ leaves $t(e,x)$ in X.

Now let $\underset{\sim}{F}_n$ be an L_n program whose running time is not bounded by $f_{n-1}^{(p)}$ for any p. There is no loss of generality if we assume $\underset{\sim}{F}_n$ is of the form "LOOP X, $\underset{\sim}{Q}$, END" for some $\underset{\sim}{Q} \in L_{n-1}$. Suppose that $n > n_0$, and let $\underset{\sim}{P}_e$ be the program $\underset{\sim}{T}_e$ followed by $\underset{\sim}{F}_n$. Clearly, $\underset{\sim}{P}_e \in L_n$.

If the Turing machine with Gödel number e halts on input e, then only finitely many numbers placed in X will lead to the execution of the subprogram $\underset{\sim}{F}_n$ in $\underset{\sim}{P}_e$, in which case the running time of $\underset{\sim}{P}_e$ is bounded by $f_{n_0}^{(p)}$ for some p. Conversely, if the Turing machine does not halt, the subprogram $\underset{\sim}{F}_n$ is always executed, and the running time of $\underset{\sim}{P}_e$ is not bounded by $f_{n-1}^{(p)}$ for any p. Therefore, if we could determine from $\underset{\sim}{P}_e \in L_n$ whether or not the running time of $\underset{\sim}{P}_e$ was bounded by $f_{n-1}^{(p)}$ for any p, we could also solve the halting problem for Turing machines. Hence, for $n > n_0$, the complexity problem is effectively unsolvable.

The value of n_0 in the preceding proof happens to be two. The easiest way to prove this is by constructing an L_2 program which "mimics" Turing Machines. The construction is similar to the one in the previous section, and makes the existence of a primitive recursive function t an unnecessary assumption in the proof of (7.2).

8. We sketch a proof of the fact that the functions computable by Loop programs with input and output are precisely the primitive recursive functions.

(8.1) Theorem. $\mathcal{L} = \mathcal{P}$.

Proof: Let $\langle \underset{\sim}{P}, \bar{X}_m, Y \rangle$ be a Loop program with input and output. Given $\langle \underset{\sim}{P}, \bar{X}_m, Y \rangle$, it can be shown by methods closely paralleling those used in section 6 that there is a primitive recursive function $M_{\underset{\sim}{P}} : \mathbb{N}^{m+1} \to \mathbb{N}$ with the following property: if y exceeds the running time of $\underset{\sim}{P}$ with input $\bar{x}_m$, then $M_{\underset{\sim}{P}}(\bar{x}_m, y)$ equals the number in Y (the output register of $\underset{\sim}{P}$) after $\bar{x}_m$ has been placed in $\bar{X}_m$ and $\underset{\sim}{P}$ executed. Now if $\underset{\sim}{P} \in L_n$, there is a $p > 0$ such that $f_n^{(p)}$ bounds $T_{\underset{\sim}{P}}$; that is, $f_n^{(p)}(\underline{\max}\{\bar{x}_m\})$ exceeds the number of steps required to execute $\underset{\sim}{P}$ with input $\bar{x}_m$.

Suppose f is the function computed by $\langle \underset{\sim}{P}, X_m, Y \rangle$; then obviously

$$f(\bar{x}_m) = M_{\underset{\sim}{P}}(\bar{x}_m, f_n^{(p)}(\underline{\max}\{\bar{x}_m\})).$$

Since f_n is defined by iteration, a special case of primitive recursion, f_n is primitive recursive. The function $\underline{\max}$ of m variables is also primitive recursive. The right side of the equation above is defined by substitution from primitive recursive functions, and therefore is a primitive recursive function. This shows that $\mathcal{P} \supseteq \mathcal{L}$. Since by (4.4) $\mathcal{L} \supseteq \mathcal{P}$, the theorem is proved.

The fact that $\mathcal{L} = \mathcal{P}$ suggests that the hierarchy $\mathcal{L}_0 \subsetneq \mathcal{L}_1 \subsetneq \mathcal{L}_2 \subsetneq \ldots$, may have an analogue more directly related to the definition of primitive recursive functions. This is in fact the case. Let $\mathcal{P}_n$ be the class of functions definable using primitive recursions nested to a depth no greater than n; that is, $\mathcal{P}_n$ is the class of functions obtained by closing under substitution the functions obtained by one primitive recursion from $\mathcal{P}_{n-1}$.* We prove in [MR] that $\mathcal{L}_n = \mathcal{P}_n$, for $n \geq 4$.

A hierarchy $\mathcal{E}^0 \subsetneq \mathcal{E}^1 \subsetneq \ldots$, of primitive recursive functions has been studied by Grzegorczyk [G]. The Grzegorczyk classes are defined by closure properties rather than programs, e.g., $\mathcal{E}^3$ is the smallest class containing $x \dot{-} y$, and closed under substitution and the functional operations which transform $f(x, y)$ into $g(x, z) = \sum_{i=0}^{z} f(x, i)$ or into $h(x, z) = \pi_{i=0}^{z} f(x, i)$. The class $\mathcal{E}^3$ is known as Kalmar's elementary functions.

Our class $\mathcal{L}_2$ equals the elementary functions; in fact for $n \geq 2$, $\mathcal{E}^{n+1} = \mathcal{L}_n$. The Axt and Grzegorczyk hierarchies will be considered in a forthcoming paper by the authors [MR].

---

*The definition is due to Axt [A].

9. Summary. Can the running time of a program, regarded as a function of the program inputs, be bounded by inspecting the structure of the program? In general the answer is no, for a fundamental theorem of the theory of computability asserts that if a programming language is powerful enough to describe arbitrarily complex computations, it must inevitable be powerful enough to describe infinite computations. Furthermore, descriptions of finite and infinite computations are in general indistinguishable, so there is certainly no way to choose for each program a function which bounds its running time.

In Section 2 of this paper we defined a class of programs, called Loop programs, which do have the property that the running time of any program in the class can be bounded by a particularly simple function of its inputs. Furthermore, the bounding function can be determined in a simple way from the structure of the program. Necessarily, the set $\mathscr{L}$ of functions computable by Loop programs is a proper subset of the set of computable functions, but in fact all of the functions ordinarily encountered in digital computation are in $\mathscr{L}$.

Loop programs are classified by the depth to which "LOOP" instructions are nested. These instructions resemble the DO statement of Fortran, and the FOR and THROUGH statements of ALGOL and MAD. We proved in Section 3 that every Loop program with loops nested to a depth of at most n has running time bounded by a particular function, f_n, possibly composed with itself.

We offered in subsequent sections of the paper a collection of theorems designed to support two contentions: first that Loop programs comprise a distinctly non-trivial and interesting programming language, and second that the bounding procedure applied to any given Loop program can provide significant information about that program. In particular we proved that

(1) There are Loop programs with loops nested to depth n whose running time is no less than the f_n,

(2) There are functions which can be computed with loops nested to depth n but not $n-1$, for every $n > 0$,

(3) If the running time of a Loop program is actually bounded by f_n, then however deeply its loops may be nested, the program could be rewritten with loops nested no deeper than n, for every $n > 1$,

(4) There is no effective procedure to determine if a program with loops nested to depth n actually has running time bounded by f_{n-1}, for any $n > 3$,

(5) The functions in $\mathscr{L}$ are precisely the primitive recursive functions.

Loop programs illustrate several of the theoretical issues involved in estimating the running time of programs.

References

[A] Axt, P. "Iteration of Primitive Recursion," Abstract 597-182, Notices Amer. Math. Soc., Jan. 1963.

[C] Cobham, A. "The Intrinsic Computational Difficulty of Functions," Proc. of the 1964 Cong. for Logic, Meth. and Phil. of Science, North-Holland, Amsterdam, 1964.

[D] Davis, M. "Computability and Unsolvability, McGraw Hill, New York, 1958.

[G] Grzegorczyk, A. "Some classes of recursive functions," Rozprawy Matematyczne, Warsaw, 1953.

[K] Kleene, S. C. Introduction to Metamathematics, Van Nostrand, New York, 1952.

[MR] Meyer, A. R., and Ritchie, D. M. "Hierarchies of Primitive Recursive Functions," in preparation.

[RS] Rabin, M. O., and Scott, D. "Finite Automata and Their Decision Problems," IBM Journal of Res. & Dev., vol. 3, April 1959, pp. 114-125.

[R] Ritchie, R. W. "Classes of Predictably Computable Functions," Trans. Amer. Math. Soc., vol. 106, Jan. 1963, pp. 139-173.