The complexity of loop programs*

by ALBERT R. MEYER
IBM Watson Research Center
Yorktown Heights, New York

and

DENNIS M. RITCHIE
Harvard University
Cambridge, Massachusetts

INTRODUCTION

Anyone familiar with the theory of computability will be aware that practical conclusions from the theory must be drawn with caution. If a problem can theoretically be solved by computation, this does not mean that it is practical to do so. Conversely, if a problem is formally undecidable, this does not mean that the subcases of primary interest are impervious to solution by algorithmic methods.

For example, consider the problem of improving assembly code. Compilers for languages like FORTRAN and MAD typically check the code of an assembled program for obvious inefficiences—say, two "clear and add" instructions in a row—and then produce edited programs which are shorter and faster than the original. From the theory of computability one can conclude quite properly that no code improving algorithm can work all the time. There is always a program which can be improved in ways that no particular improvement algorithm can detect, so no such algorithm can be perfect. But the non-existence of a perfect algorithm is not much of an obstacle in the practical problem of finding an algorithm to improve large classes of common programs.

The question of detecting improvable programs will appear again later in this paper, but our main concern will be with a related question: can one look at a program and determine an upper bound on its running time? Again, a fundamental theorem in the theory of computability implies that this cannot be done.* The theorem does *not* imply that one cannot bound the running time of broad categories of interesting programs, including programs capable of computing all the arithmetic functions one is likely to encounter outside the theory of computability itself.

In the next section we describe such a class of programs, called "Loop programs." Each Loop program consists only of assignment statements and iteration (loop) statements, the latter resembling the DO statement of FORTRAN, and special cases of the FOR and THROUGH statements of ALGOL and MAD. The bound on the running time of a Loop program is determined essentially by the length of the program and the depth of nesting of its loops.

Although Loop programs cannot compute all the computable functions, they can compute all the *primitive* recursive functions. The functions computable by Loop programs are, in fact, precisely the primitive recursive functions. Several of our results can be regarded as an attempt to make precise the notion that the complexity of a primitive recursive function is apparent from its definition or program. This property is one of the reasons that the primitive recursive functions are used throughout the theory of computability, for, as we remarked in the opening paragraph, knowing that a function is computable is

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

*Roughly speaking, the undecidability of the halting problem for Turing machines implies that if a programming language is powerful enough to describe arbitrarily complex computations, then the language must inevitably be powerful enough to describe infinite computations. Furthermore, descriptions of finite and infinite computations are generally indistinguishable, so there is certainly no way to choose for each program a function which bounds its running time.

not very useful unless one can tell how difficult the function is to compute. A bound on the running time of a Loop program provides a rough estimate of the degree of difficulty of the computation defined by the program.

Loop programs are so powerful that our bounds on running time cannot be of practical value—for functions computable by Loop programs are almost wholly beyond the computational capacity of any real device. Nevertheless they provide a good illustration of the theoretical issues involved in estimating the running time of programs, and we believe that readers with a practical orientation may find some of the results provocative.

Loop programs

A Loop program is a finite sequence of instructions for changing 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 fixed number of registers.

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 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.

A sequence of instructions is a Loop program providing that type (4) and type (5) instructions are matched like left and right parentheses. 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 **P** is a Loop program, and the integer in X is x, then "LOOP X, **P**, END" means that **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 **P** is being repeated do not alter the number of times **P** is to be repeated. The final clause is needed to ensure that executions of Loop programs always terminate. For example, the program

```
LOOP X          (2.1)
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.

Since LOOP's and END's appear in paris like left and right parentheses, the block of instructions affected by a LOOP instruction is itself a Loop program and is uniquely determined by the matching END instruction. L_n is defined as the class of programs with LOOP-END pairs nested to a depth of at most n; depth zero means the program has no LOOP's.

For example, (2.2) is an L_2 program in which type (4) and (5) instructions are paired as indicated by the indentations.

```
LOOP Y          (2.2)
  A=0
  LOOP X
    X=A
    A=A+1
  END
END
```

If X and Y initially contain x and y, execution of (2.2) would leave $x \dot{-} y$ in X, where $x \dot{-} y$ equals $x - y$ if $x \geq y$, and is zero otherwise.

We say that a Loop program computes a function as soon as some of the registers are designated for input and output. If f is a function (from non-negative integers into non-negative integers) of m-variables, then a Loop program **P** with input registers $X_1, \ldots,$ X_m and output register F *computes* f providing that, when registers $X_1, \ldots, X_m$ initially contain integers $x_1, \ldots, x_m$ and all other registers initially contain zero, then the integer left in F after **P** has been executed is $f(x_1, \ldots, x_m)$. Thus, program (2.1) with X serving as both input and output register computes the function 2x. For each $n \geq 0$, $\mathscr{E}_n$ is defined as the set of functions computable by a program in L_n.

Primitive recursive functions

Loop programs are extremely powerful despite their simple definition. Addition, multiplication, exponentiation, the x^{th} digit in the decimal expansion of sin (½), the x^{th} prime number, etc., are all functions computable by Loop programs. In fact, a fairly careful analysis of the definition of Loop programs is required in order to discover a function which they cannot compute.

These properties are well-known for the class of computable functions known as the *primitive recursive functions*. They apply to Loop programs as well by Theorem 1.

Theorem 1. Every primitive recursive function is computed by some Loop program.

Thorough treatments of the primitive recursive functions can be found in many elementary texts on logic and computability.[1,2] For the reader's convenience we provide a definition of th primitive recursive functions. It is fairly easy to translate a definition by primitive recursions into a Loop program, and thereby prove Theorem 1.

Definition 1. The primitive recursive functions are the smallest class of functions $\mathscr{P}$ satisfying

(1) the functions $s(x) = x + 1$, $p_{12}(x,y) =$ are in $\mathscr{P}$

(2) $\mathscr{P}$ is closed under the operations of *substitution:* substituting constants, permuting variables (obtaining the function h from f where $h(x,y) = f(y,x)$), identifying variables (obtaining $h(x) = f(x,x)$ from $f(y,z)$), and composing functions.

(3) $\mathscr{P}$ is closed under the scheme of primitive recursion: if $g,h \in \mathscr{P}$, then $f \in \mathscr{P}$ where f is defined as $f(x_1,\ldots,x_m,0) = g(x_1,\ldots,x_m)$
$f(x_1,\ldots,x_m,y+1) = h(x_1,\ldots,x_m,y,f(x_1,\ldots,x_m,y))$.

Bounding the running time

Given a Loop program and the integers initially in its registers, the running time of a program is defined as the number of individual instruction executions required to execute the entire program. Thus each Loop program, **P**, has an associated running time, T_p, which is a function of as many variables as there are registers in **P**.

Hopefully Section 2 makes it obvious how to count the number of individual executions of type (1), (2), and (3) instructions in any particular computation. The number of executions of LOOP and END instructions can be counted in several ways, but the simplest course is to ignore them. For example, the running time function of an L_0 program (a program with no loops) is a constant function equal to the length of the program; the running time function of program (2.1) is the identity function $f(x) = x$.

This definition of running time has the advantage that it leads to a trivial proof of

Theorem 2. If **P** is a Loop program in L_n, then the running time function T_p is in $\mathscr{E}_n$.

The proof consists of the observation that if **P'** is the program obtained by inserting the instruction "T = T + 1" after each type (1), (2) and (3) instruction on **P**, then **P'**, computes T_p when all registers of **P'** except for T are designated as input registers, and T is designated as output register. We assume that "T" itself does not already occur in **P**. Clearly if **P** is in L_n, then so is **P'**, and therefore $T_p \in \mathscr{E}_n$.

Now this argument depends heavily on our conventions for measuring running time, and these conventions may seem arbitrary. Actually, Theorem 2 remains true under a wide variety of definitions of running time, as does the following

Corollary. If a Loop program computes a function f, then f is totally defined and is effectively computable.

This amounts to saying that computations defined by Loop programs are always finite no matter what the initial contents of the registers may be, i.e., Loop programs always halt.

As a result of this theorem and corollary, the claim that the running time of Loop programs can be bounded *a priori* by inspection of the program becomes trivial. After all, T_p certainly bounds the running time since by definition it *is* the running time, T_p is totally defined and effectively computable, and moreover given **P** one can effectively describe how to compute T_p. Therefore, one can bound the running time of **P** by T_p.

Of course, bounding **P** by T_p gives absolutely no new information. It amounts to "predicting" that the program **P** will run as long as it runs, which is not much of an answer to the question, "How long does my program run?"

A proper answer should be in terms of familiar bounding functions whose properties are simple and understandable. For example, the running time of a typical program computing $f(x) =$ "the x^{th} digit in the decimal expansion of $\sqrt{2''}$ is bounded by x^2; for any context-free language there is a recognition algorithm whose running time is bounded by a constant times the cube of the length of an input word; for context-sensitive languages the bound is an exponential of an exponential of the length of an input word, etc.

The functions we use to bound the running time of Loop programs are given in

Definition 2. For a function g of one variable, let
$$g^{(y)}(z) = g(g(\ldots g(z)\ldots)),$$
the composition being taken y times. By convention, $g^{(0)}(z) = z$. For $n \geq 0$, the functions f_n are defined by:
$$f_0(x) = \begin{cases} x+1 & \text{if } x = 0,1 \\ x+2 & \text{if } x \geq 2, \end{cases}$$
$$f_{n+1}(x) = f_n^{(x)}(1).$$

We will say that a function f of one variable *bounds* a function g of several variables if $g(x_1,\ldots,x_m) \leq f(max\{x_1,\ldots,x_m\})$ for all integers $x_1,\ldots,x_m$; $max\{x_1,\ldots,x_m\}$ is the largest member of $\{x_1,\ldots,x_m\}$.

Theorem 3. Bounding Theorem. If **P** is in L_n, then T_p is bounded by $f_n^{(k)}$ where k is the length (number of instructions) of **P**.

Theorem 3 at least provides a particular class of bounding functions which are reasonably simple and

well-behaved. The first few functions f_n are familiar, viz., $f_1(x) = 2x$ for $x > 0$, and $f_2(x) = 2^x$. The function f_3 is also easy to describe:

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

Furthermore, $f_n^{(k)}(x)$ is strictly increasing in n, k, and x whenever $x \geq 2$, and in fact f_{n+1} grows more rapidly than $f_n^{(k)}$ for any fixed value of k. The latter property implies that f_{n+1} *majorizes* $f_n^{(k)}$ for any fixed values of n and k, i.e., $f_{n+1}(x) > f_n^{(k)}(x)$ for all values of x larger than some bound which depends on n and k.

The definition of f_{n+1} from f_n is by a special case of primitive recursion. Translating this definition into a Loop program, it is easy to show that for $n \geq 1$, f_n is in $\mathscr{L}_n$. The majorizing property and the fact that the running time of a program for any function g(x) is at least $g(x) - x$ (it requires this many steps just to leave the answer in the output register), may be combined to prove

Lemma. For all $n \geq 0, f_{n+1} \in \mathscr{L}_{n+1} - \mathscr{L}_n$.

Theorem 4. For all $n,k \geq 0$, there are functions in $\mathscr{L}_{n+1}$, which cannot be computed by any Loop program whose running time is bounded by $f_n^{(k)}$.

One can still question whether a bound on running time of f_7, for example, is any better than no bound at all. In practice it probably is not, since the underlying practical question is always whether a program runs in a reasonable amount of time on a reasonable finite domain of inputs. If a program's running time could not be bounded by a function smaller than f_7, the program might just as well contain an infinite loop. In both cases, computation for inputs larger than two would not terminate during the lifetime of the Milky Way galaxy. Of course similar objections can be raised against a bound of x^7, or even $10^7 \cdot x$.

Complexity classes

The sequence $\mathscr{L}_0, \mathscr{L}_1, \mathscr{L}_2, \ldots$ is, by Theorem 4, a strictly increasing sequence of sets which provide an infinite number of categories for classifying functions. Although the sets $\mathscr{L}_n$ were defined syntactically, by depth of loops, the Bounding Theorem implies that classification by position in the sequence relates to classification by running time or computational complexity. If we define complexity classes $C_0, C_1, C_2, \ldots$ by letting C_n be precisely the functions computable by programs with running time bounded by $f_n^{(k)}$ for any fixed $k \geq 0$, then the results of Section 4 imply that $C_n \supset \mathscr{L}_n$, and $f_{n+1} \in \mathscr{L}_{n+1} - C_n$.

The fact that $C_n = \mathscr{L}_n$ is still something of a surprise.

Theorem 5. For $n \geq 2$, a function is in $\mathscr{L}_n$ if and only if it can be computed by a Loop program whose running time is bounded by $f_n^{(k)}$ for some k.

The "only if" part of the theorem is simply the Bounding Theorem. The reverse implication can be proved by showing that if the running time of a program **P** is bounded by $f_n^{(k)}$, then regardless of the actual depth of loops, **P** can be rewritten as a program with loops nested to depth n (providing $n \geq 2$).

The rewriting of **P** proceeds roughly as follows: if **P** is a sequence of instructions $I_1, I_2, \ldots, I_k$, then a sequence of L_1 program $I_1, I_2, \ldots, I_k$ is constructed. Each program I_i has a flag which it tests. If its flag is off, I_i has no effect. If its flag is on, I_i has the same effect on the registers of **P** as execution of the instruction I_i, and I_i also sets a flag for the next instruction which would be executed in a computation of **P**. Let **M** be the program "LOOP T, I_1, I_2, ..., I_k, END" where T is a new register name. Given the integers $x_1, \ldots, x_m$ initially in the registers of **P**, if an integer larger than $T_p(x_1, \ldots, x_m)$ is initially in register T, then **M** will simulate the computation of **P**.

Since T_p is bounded by $f_n^{(k)}$, one need only construct an L_n program $\mathbf{M}_n$ which leaves $f_n^{(k)}$ in register T. Then $\mathbf{M}_n$ followed by the L_2 program **M** together form a program in L_n which computes the same functions as **P**. This composite program is the "rewritten" version of **P**.

Theorem 5 would be trivial if every program not in L_n took more time than f_n, for in that case rewriting would never be possible. So far we have shown that some programs in L_n do run as long as f_n, but an obvious flaw in our Bounding Theorem remains: there are programs in L_n with running times which can be bounded by functions smaller than f_n. For example, a program of the form

```
T=0
LOOP T
P
END
```  (5.1)

has running time equal to one for all inputs and all possible programs **P**. Hence, our procedure for bounding running time can certainly be improved somewhat. This naturally suggests the question: can one tell if a program runs more rapidly than its loop structure indicates? In general, the answer is no.

Definition 3. The *complexity problem for* L_n is: given a program **P** in L_n determine whether T_p is bounded by $f_{n-1}^{(k)}$ for any integer k.

Theorem 6. For each $n \geq 3$, the complexity problem for L_n is effectively undecidable.

One of the implications of Theorem 6 is that any procedure for bounding the running time of Loop programs can be supplemented to cover special cases of the sort illustrated by (5.1). There cannot be a perfect bounding procedure. Yet no matter how many special cases are treated, there remain an infinite num-

ber of cases for which any procedure must reduce to essentially the one given by the Bounding Theorem. It is in this, admittedly weak, sense that we claim the Bounding Theorem is best possible.

If an "improvement" of the code of a program is defined as a reduction in depth of loops or number of instructions (without an increase in running time), then the proof of Theorem 6 also reveals that there can be no perfect code improvement algorithm. Thus the code improvement problem, which we noted in the introduction was undecidable for programs in general, is still undecidable for the more restricted class of Loop programs.

Primitive recursive classes.

Loop programs were devised specifically as a programming language for primitive recursive functions, so of course we have: *Theorem 7.* The functions computable by Loop programs are precisely the primitive recursive functions.

Half of Theorem 7 already follows from Theorem 1, and the other half can be proved in much the same way as Theorem 1. Parallel to the definition of L_n, one can define* $\mathscr{P}_n$ as the set of functions definable using primitive recursions nested to depth at most n. The computation defined by a LOOP-END pair is slightly more general than that defined by a single primitive recursion, so that $L_1 \supsetneq \mathscr{P}_1$ and $L_2 \supsetneq \mathscr{P}_2$ ($\mathscr{P}_0 = L_0$ by definition), but for $n \geq 4, \mathscr{P}_n = L_n$. The simplest proof of this fact, however, seems to depend on Theorem 5 rather than the translation of Loop programs into primitive recursive definitions used in Theorem 7.

Another sequence $\xi^0, \xi^1, \xi^2, \ldots$ of sets of primitive recursive functions has been defined by Grzegorczyk[4] using closure properties rather than programs. For example, ξ^3 is the class of *elementary functions* defined as the closure of the function $x \dot{-} y$ under substitution and functional operations which transform f(x,y) into $\sum_{i=0}^{y} f(x,i)$ or into $\prod_{i=0}^{y} f(x,i)$. Our class L_2 equals the elementary functions; in fact for $n \geq 2, \xi^{n+1} = L_n$.

The proofs of most of the theorem in this paper appear in [5]. The Axt and Grzegorczyk classes are treated in detail in [6].

REFERENCES

1 M DAVIS
Computability and unsolvability
McGraw-Hill Book Company Inc New York 1958

2 S C KLEENE
Introduction to metamathematics
Van Nostrand New York 1952

3 P AXT
Iteration of primitive recursion
Abstract 597-182 Notices Amer Math Soc Jan. 1963

4 A GRZEGORCZYK
Some classes of recursive functions
Rozprawy Matematyczne, Warsaw, 1953

5 A R MEYER and D M RITCHIE
Computational Complexity and Program Structure
IBM Research Research Report, RC-1817.

6 A R MEYER and D M RITCHIE
Hierarchies of primitive recursive functions, in preparation

7 A COBHAM
The intrinsic computational difficulty of functions
Proc of the 1964 Cong for Logic Meth and Phil of Science
North-Holland, Amsterdam 1964

8 R W RITCHIE
Classes of predictably computable functions
Trans Amer Math Soc Vol 106 Jan 1963 pp 139-173.

*The definition is due to Axt[3].