JOURNAL OF COMPUTER AND SYSTEM SCIENCES 7, 354–375 (1973)

Time Bounded Random Access Machines

STEPHEN A. COOK AND ROBERT A. RECKHOW

Department of Computer Science, University of Toronto, Toronto, M5S 1A7, Ontario

Received August 3, 1972

The RAM, an abstract model for a random access computer, is introduced. A unique feature of the model is that the execution time of an instruction is defined in terms of $l(n)$, a function of the size of the numbers manipulated by the instruction. This model has a fixed program, but it is shown that the computing speeds of this model and a stored-program model can differ by no more than a constant factor. It is proved that a $T(n)$ time-bounded Turing machine can be simulated by an $O(T(n) \cdot l(T(n)))$ time-bounded RAM, and that a $T(n)$ time-bounded RAM can be simulated by a Turing machine whose execution time is bounded by $(T(n))^2$ if $l(n)$ is constant, or $(T(n))^2$ if $l(n)$ is logarithmic.

The main result states that if $T_2(n)$ is a function such that there is a RAM that computes $T_2(n)$ in time $O(T_2(n))$, and if $T_1(n)$ is any function such that

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

then there is a set S that can be recognized by some RAM in time $O(T_2(n))$, but no RAM recognizes S in time $O(T_1(n))$. This is a sharper diagonal result than has been obtained for Turing machines.

The proofs of most of the above results are constructive and are aided by the introduction of an ALGOL-like programming language for RAM's.

In this paper we introduce a formal model for random access computers and argue that the model is a good one to use in the theory of computational complexity. In the past, general results concerning the time and storage needed to solve computational problems have often been quoted in terms of multitape Turing machines, even though such machines are not much like any existing computers that would actually carry out the computations. Even when an algorithm intended for a "real" computer is analyzed, it is often not made explicit what arithmetic operations must be available to the machine and what idealizations concerning unlimited storage, word length, etc., must be assumed. We believe it is often desirable to make these points explicit when quoting positive results about specific algorithms, and essential to make them explicit when quoting general complexity results such as lower time bounds or hierarchy theorems.

The model we develop involves a number of hard choices concerning which arithmetic operations to make primitive, and what assumptions to make concerning unlimited storage and word length. Our choices are partly justified by experience with formalizing algorithms, and partly by the naturalness of the theorems presented in this paper. A further discussion of these points is presented in [2]. Our model includes addition as primitive, but not multiplication, since the latter can be rapidly simulated by addition, and multiplication is not needed for the algorithms presented here. The random access storage is unlimited, and the word length is unlimited, although we propose making a time charge roughly equal to the logarithm of the magnitude of each number placed in storage. This is like charging for the number of words required to store a high precision number on a fixed wordsize machine.

One of our theorems compares our fixed program computer model with a stored program model. A second result compares the run times for executing algorithms using our random access model and multitape Turing machines, and shows that the run times do not differ greatly. The main result, Theorem 3, shows the existence of a time complexity hierarchy which is finer than is known to exist for any standard abstract computer model. The proofs of the theorems make use of an ALGOL-like programming language introduced for our random access machines.

1. Random Access Machines

A *random access machine* (RAM) consists of a finite program operating on an infinite sequence of registers. Each register can hold an arbitrary integer (positive, negative, or zero). The contents of the registers is denoted by the sequence $X_0, X_1, X_2, \ldots$. Associated with the machine is the function $l(n)$ which, roughly speaking, denotes the time required to store the number n. The most natural values for $l(n)$ are (1) take $l(n)$ identically one, and (2) take $l(n)$ approximately $\log |n|$. For the results stated here, if the value of $l(n)$ is not given explicitly, we need only assume $l(n)$ is positive, nondecreasing on the positive integers, symmetric ($l(n) = l(-n)$), and subadditive ($l(n + m) \leqslant l(n) + l(m)$).

The possible instructions, together with their execution time in terms of $l(n)$, are given in Table I. Here i, j, k, are any nonnegative integers. The effect of most of the instructions should be evident. For example, $X_i \leftarrow C$ causes X_i to assume the value C, while $X_i \leftarrow X_j + X_k$ causes X_i to assume the value $X_j + X_k$. The instruction TRA m if $X_j > 0$ causes control to be transferred to line m of the program if $X_j > 0$. Normally control is transferred from one line to the next. READ X_i causes X_i to take on the value of the next input number, and PRINT X_i causes X_i to be printed on the output tape.

The indirect instruction $X_i \leftarrow X_{X_j}$ causes register number X_j to be copied into register number i, provided $X_j \geqslant 0$. The instruction $X_{X_i} \leftarrow X_j$ has an analogous

TABLE I

RAM Instructions and Execution Times

Instruction	Execution time
$X_i \leftarrow C$, C any integer	1
$X_i \leftarrow X_j + X_k$	$l(X_j) + l(X_k)$
$X_i \leftarrow X_j - X_k$	$l(X_j) + l(X_k)$
$X_i \leftarrow X_{X_j}$	$l(X_{X_j}) + l(X_j)$
$X_{X_i} \leftarrow X_j$	$l(X_i) + l(X_j)$
TRA m if $X_j > 0$	$l(X_j)$
READ X_i	l(input)
PRINT X_i	$l(X_i)$

meaning. The indirect instructions are necessary in order for a fixed program to access an unbounded number of registers as the inputs vary.

A RAM program is started at the first instruction, with all registers initially zero, and it halts when a transfer is made to a line with no instructions, or when a negative indirect address is encountered.

We propose the following value for the function $l(n)$:

$$l(n) = \begin{cases} \lceil \log_2 \mid n \mid \rceil & \text{if} \quad \mid n \mid \geqslant 2 \\ 1 & \text{if} \quad \mid n \mid < 2 \end{cases}$$

(Here $\lceil x \rceil$ is the least integer $\geqslant x$.) We will call this the *logarithmic function.*

Briefly, the reason for taking $l(n)$ logarithmic instead of constant is that each register is allowed to hold arbitrarily large integers. Thus any fixed word length machine simulating our RAM would require about log n registers to store the number n held in a single RAM register (where the base of the logarithm is the largest number that can be stored in a single register). A more extensive argument is given in [2], along with motivation for our choice of instructions.

We are interested in the time required for a RAM to recognize a set A of strings on a finite alphabet $\Sigma = \{\sigma_1, \ldots, \sigma_p\}$. A string $w = \sigma_{i_1}\sigma_{i_2} \cdots \sigma_{i_n}$ will be presented to the machine as the sequence of integers $i_1, i_2, \ldots, i_n, 0$, where 0 indicates the end of the string. The machine must execute a sequence of $n + 1$ Read instructions to acquire the string. The machine accepts w by printing a 1 and halting (in a computation with input w) and rejects w by printing a 2 and halting. The computation time is the sum of the execution times of all steps in the computation as given by the second column of Table I. We say the machine M *recognizes a set* A *of strings within time* $T(n)$ iff for every string w on Σ, M halts on input w within time $T(\mid w \mid)$ and accepts w

if $w \in A$ and rejects w if $w \notin A$. We are interested only in the rate of growth of run times as a function of the size of the input, which accounts for the fact that the execution times in Table I may differ from their "proper" value by a constant factor.

2. The Programming Language

Since the proofs of the results reported here require construction of lengthy RAM programs, it is convenient to introduce a programming language called RAM–ALGOL, which is a subset of ALGOL 60 as described in the revised ALGOL Report [3]. The main difference between RAM–ALGOL and ALGOL 60 are:

1. Real numbers have been eliminated.
2. The only arithmetic operators are $+$ and $-$.
3. Procedures and switches are not allowed to be recursive.
4. Arrays are one-dimensional and infinite.

These differences are described in more detail in [1].

In order for RAM–ALGOL to be useful for proving theorems about time bounds, it is necessary to implement RAM–ALGOL on a RAM in such a way that the execution time of a RAM "object program" can be determined (to within a constant multiple) by inspection of the RAM–ALGOL "source program." This type of implemention can be done in a relatively straightforward way, and is described in detail in [1].

The most interesting feature of the language to implement is arrays. The declaration

integer array A;

declares A to be an infinite array of integers:

$$A[0], A[1], A[2], \ldots .$$

A RAM–ALGOL program can contain many arrays, but since there is no recursion, a fixed program P references at most a fixed number of arrays, say k of them. We can then interleave the storage for these k arrays and the scalars and temporaries of P as follows:

array A_1 is stored in RAM registers
$X_1, X_{(k+1)+1}, X_{2(k+1)+1}, \ldots$
array A_2 is stored in RAM registers
$X_2, X_{(k+1)+2}, X_{2(k+1)+2}, \ldots$
$\vdots$
array A_k is stored in RAM registers
$X_k, X_{(k+1)+k}, X_{2(k+1)+k}, \ldots$
scalars of P are stored in RAM registers
$X_0, X_{k+1}, X_{2(k+1)}, \ldots$

so that in general, the value of $A_i[j]$ is found in RAM register $X_{j(k+1)+i}$. The accessing of elements of arrays can now be implemented as follows. Suppose, for example, that we want to implement the RAM–ALGOL statement $Y := A[Z]$ on a RAM. Assume that the integers Y and Z are stored in the RAM registers X_Y and X_Z, respectively, that A is the ith of k arrays in program P, and that X_{temp} is a temporary register. Then we have the following implementation:

Program	*Timing*
$X_{\text{temp}} \leftarrow i$	1
$k+1$ times: $X_{\text{temp}} \leftarrow X_{\text{temp}} + X_Z$	$l(i) + l(Z)$
$\vdots$	
$X_{\text{temp}} \leftarrow X_{\text{temp}} + X_Z$	$l(kZ + i) + l(Z)$
$X_Y \leftarrow X_{X_{\text{temp}}}$	$l((k+1)Z + i) + l(A[Z])$

Since k and i are fixed, and $l(n)$ is subadditive, the total execution time of this statement is $O(l(Z) + l(A[Z]))$. Storing into an array is implemented in an analogous way, so that the execution time of $A[Z] := Y$ is $O(l(Z) + l(Y))$.

Since we are not concerned with constant factors in the timing analysis, the other features of ALGOL can generally be implemented in the most obvious way. Since procedures are not allowed to be recursive, we can actually replace each procedure statement by its procedure body (as described in the revised ALGOL Report [3]), and translate the resulting procedureless program into RAM instructions.

Appendix A to this paper contains an example of a RAM–ALGOL program and its timing analysis used in the proof of Theorem 1. Complete RAM–ALGOL programs are given for most of the proofs in [1].

3. Stored Program Machines

The RAM model described in Sec. 1 has a fixed program. Since most existing computers are stored program devices, it is worthwhile asking whether the stored program feature adds to the computation speed. The Random Access Stored-Program Machine (RASP) described here is similar to the RASP's described by Hartmanis [4]. The specific machine RASP1 described by Hartmanis is essentially the same as our RASP with indirect addressing and with $l(n)$ identically one. Our machine has an *accumulator* (AC), which holds an arbitrary integer, an *instruction counter* (IC), which holds a nonnegative integer, and an infinite sequence of *memory registers* $X_0, X_1, X_2, \ldots$, each of which can hold an arbitrary integer. The instructions for our RASP, along with their timings, are given in Table II.

An instruction is stored in two consecutive memory registers. The first register

TABLE II

RASP Instructions and Execution Times

Operation	Mnemonic	Operation code	Description of the operation	Execution time
load constant	LOD, j	1	AC $\leftarrow j$; IC $\leftarrow$ IC + 2	l(IC) + $l(j)$
add	ADD, j	2	AC $\leftarrow$ AC + X_j; IC $\leftarrow$ IC + 2	l(IC) + $l(j)$ + l(AC) + $l(X_j)$
subtract	SUB, j	3	AC $\leftarrow$ AC $-$ X_j; IC $\leftarrow$ IC + 2	l(IC) + $l(j)$ + l(AC) + $l(X_j)$
store	STO, j	4	$X_j \leftarrow$ AC; IC $\leftarrow$ IC + 2	l(IC) + $l(j)$ + l(AC)
branch on positive accumulator	BPA, j	5	if AC $>$ 0 then IC $\leftarrow j$; otherwise IC $\leftarrow$ IC + 2	l(IC) + $l(j)$ + l(AC)
read	RD, j	6	$X_j \leftarrow$ next input; IC $\leftarrow$ IC + 2	l(IC) + $l(j)$ + l(input)
print	PRI, j	7	output X_j; IC $\leftarrow$ IC + 2	l(IC) + $l(j)$ + $l(X_j)$
halt	HLT	$-\infty$ to 0 and 8 to ∞	stop	l(IC) + $l(X_{IC})$

Note:
The machine halts if

a. The operation code (X_{IC}) is not between 1 and 7, or
b. The parameter j (i.e., X_{IC+1}) is negative for operations 2 through 7.

contains an operation code (shown in the third column of Table II), and the second contains the parameter of the instruction, j, which is either an address or an integer constant. Note that since indirect addressing is not provided for by the instruction set, RASP programs *must* modify themselves in order to access an unbounded number of registers. Normally a RASP program consists of a finite number of instructions (the first of which is stored in registers X_0 and X_1) and data words, stored in certain specified memory registers. All other memory registers and the AC and IC are initially set to zero. Execution of an instruction consists of retrieving an operation code from register X_{IC} and (provided the operation code does not specify a halt operation)

the parameter j from register $X_{\text{IC}+1}$. The appropriate operation is then performed as indicated in the fourth column of Table II, and the cycle is repeated until an illegal operation code or parameter is encountered, at which time the machine halts.

As with the RAM, RASP execution times are weighted by a cost function $l(n)$. The execution times of the various RASP instructions are listed in the last column of Table II. Contrary to the RAM instruction timings, the processing of addresses and constants is explicitly charged for on a RASP. This is reasonable, since by modifying itself, a RASP program can generate instructions with arbitrarily large addresses and constants and then execute them. We also charge for the value of the instruction counter at the time of execution of each instruction, since the IC may also grow arbitrarily large during execution.

Note that the inclusion of statements for indirect addressing could increase the computation speed by no more than a constant factor. For example, if we had included LDI, j (load indirect) meaning $\text{AC} \leftarrow X_{X_j}$, its cost would have been $l(\text{IC}) + l(j) + l(X_j) + l(X_{X_j})$. But this can be simulated by the sequence:

```
    LOD, 0
    ADD, j
    STO, a + 1
    LOD, 0
a   ADD, 0.
```

This sequence has execution time $O(l(\text{IC}) + l(j) + l(X_j) + l(X_{X_j}))$.

THEOREM 1. *For each function $T(n) \geqslant n$, a set A is recognizable by a* RAM *in time $O(T(n))$ if and only if A is recognizable by a* RASP *in time $O(T(n))$.*

The proof that a RAM can simulate a RASP program P in a proportional amount of time consists of exhibiting the RAM that does the simulation. This RAM program is given in Appendix A, written in RAM–ALGOL. It uses an array, MEMORY, to hold the contents of P's memory. It first initializes the first p elements of the MEMORY array, where p is the length of P. This initialization takes a fixed amount of time. It then simulates the execution of P by retrieving the instructions from the MEMORY array and executing them interpretively. Appendix A shows that the time required to simulate the execution of an instruction of P is proportional to the execution time for that instruction in P. We can then conclude that if P halts within time $O(T(n))$, the RAM halts within time $O(T(n))$.

Simulation of a RAM by a RASP is accomplished by replacing each RAM instruction by an equivalent sequence of RASP instructions. Since part of the RASP storage is needed for the program, all of the registers used (explicitly or by indirect reference) by the RAM must be offset by a fixed constant in the simulation. We illustrate by

giving in Table III the sequence of RASP instructions which would replace the RAM instruction $X_{X_i} \leftarrow X_j$. Here p is a bound on the total length of the RASP program.

Since it takes at most six RASP instructions to simulate a single RAM instruction, the bound p can be taken to be $12 \cdot m$, where m is the number of RAM instructions. In fact, all components of the execution times given in Table III, such as $l(\text{IC})$, $l(a+1)$, $l(i+p)$, are bounded by a constant depending only on the RAM program.

TABLE III

Simulation of $X_{X_i} \leftarrow X_j$

RASP code	Execution time
LOD, p	$l(\text{IC}) + l(p)$
ADD, $(i+p)$	$l(\text{IC}) + l(i+p) + l(p) + l(X_i)$
STO, $(a+1)$	$l(\text{IC}) + l(a+1) + l(X_i + p)$
LOD, 0	$l(\text{IC}) + l(0)$
ADD, $(j+p)$	$l(\text{IC}) + l(j+p) + l(0) + l(X_j)$
a STO, 0	$l(\text{IC}) + l(X_i + p) + l(X_j)$

Thus the time required to simulate $X_{X_i} \leftarrow X_j$ is $O(l(X_i) + l(X_j))$, which is proportional to the RAM execution time of the instruction. A similar statement can be made for each instruction simulated. Thus the simulation time is bounded by a constant times the original execution time.

4. Turing Machines and RAM's

Since the multitape Turing machine is the standard computer model used in complexity literature, it is interesting to compare its computing power with that of RAM's. The result shows they are not as different as one might expect.

THEOREM 2. (a) *If a set A is recognized by a* RAM *P within time $T(n) > n$, and*

(i) *if P has $l(n)$ logarithmic, then A is recognized by some multitape Turing machine within time $(T(n))^2$, and*

(ii) *if P has $l(n)$ identically constant, then A is recognized by some multitape Turing machine within time $(T(n))^3$.*

(b) *Conversely, if some Turing machine recognizes A within time $T(n) \geqslant n$, then some* RAM *(with arbitrary cost function $l(n)$) recognizes A within time $O(T(n) \cdot l(T(n)))$.*

The proof of (b) is straightforward. The simulating RAM simply stores the contents of the Turing machine's k tapes in $2k$ arrays, one tape symbol per array element, with one array for the left half of each tape, and one array for the right half. The finite control of the Turing machine is handled by the simulating program in a fixed amount of time per Turing machine step, except for accessing of the tape arrays. Since a head of the Turing machine can move at most $T(n)$ squares away from the starting position in $T(n)$ steps, the time it takes the RAM to access this farthest tape square is $O(l(T(n)))$.

The proof of (a) is a bit more subtle. The problem is that the Turing machine needs an efficient way to simulate the random access storage and the instructions of a RAM. Our simulating machine M will use one of its work tapes to hold the contents of P's memory in the following format:

$$\$*a_1 \# c_1 *a_2 \# c_2 \cdots *a_j \# c_j \not\#$$

Here the a_i $(i = 1, 2, \ldots, j)$ are the addresses of the registers of P that have been stored into thus far in the computation. The a_i are represented in binary and are arranged in increasing order. The c_i $(i = 1, 2, \ldots, j)$ are the contents of the corresponding registers, also stored in binary.

LEMMA. *The nonblank portion of the above work tape contains $O(T(n))$ squares in case $l(n)$ is logarithmic, and $O((T(n))^2)$ squares in case $l(n)$ is constant.*

For the proof, suppose first that $l(n)$ is logarithmic. After each RAM instruction is executed, at most one register is altered, and the number stored in that register has a number of bits bounded by a constant times the RAM cost of executing that instruction. The number of bits in the address of that register is also bounded by a constant times the cost of the instruction. Since the total RAM cost of the computation is bounded by $T(n)$, the length of the work tape will never exceed a constant times $T(n)$.

If $l(n)$ is constant, the argument is similar, except the bound on the length of each register and address is $O(T(n))$. This is because (except for loading constants) the maximum of the numbers in the registers can at most double after each RAM instruction is executed. Thus in $T(n)$ steps the maximum is $O(2^{T(n)})$, and the binary length of this is $O(T(n))$. Since at most $T(n)$ registers can be accessed in $T(n)$ steps, the total length of the work tape is $O((T(n))^2)$, as stated in the lemma.

The theorem follows readily from the lemma. Each RAM instruction can be simulated by the Turing machine by a small fixed number of passes down the work tape described above, with the help of other work tapes. Each pass requires time $O(T(n))$ or $O((T(n))^2)$, depending on $l(n)$. Since the RAM executes at most $T(n)$ instructions before halting, the total simulation time is bounded by $O((T(n))^2)$ or

$O((T(n))^3)$. Since Turing machines have linear speedup and since $(T(n))^2 > n$, there is a Turing machine which simulates P in time $((T(n))^2$ (if $l(n)$ is logarithmic) or $(T(n))^3$ (if $l(n)$ is constant). (See [1] for more details.)

5. The RAM Complexity Classes

Definition. A function $T(n)$ on the positive integers is *time constructable* (with respect to a cost function $l(n)$) iff there is some RAM program P (with cost function $l(n)$) such that for all n, P reads n, calculates and stores $T(n)$ in some register, and halts within time $O(T(n))$.

We note that "time constructable" is similar to "real-time countable" for Turing machines, and is the analog for time of "constructable" (as defined in [5]) for storage.

Theorem 3. *If $l(n) = 1$ or $l(n)$ is logarithmic, and if $T_2(n) \geqslant nl(n)$ is time constructable with respect to $l(n)$, then there is a set $A \subseteq \{1, 2\}^*$ such that some* RAM *program recognizes A within time $O(T_2(n))$, but for any function $T_1(n)$ satisfying*

$$\liminf_{n\to\infty} \frac{T_1(n)}{T_2(n)} = 0 \tag{1}$$

no RAM *program recognizes A within time $O(T_1(n))$.*

We note that Hennie and Stearns [6] prove a similar result for multitape Turing machines, but their result is weaker in that the numerator in Eq. (1) is $T_1(n) \log T_1(n)$ instead of $T_1(n)$. In fact, we know of no argument in the literature proving a result as strong as Theorem 3 for a class of abstract computing machines.

The proof of Theorem 3 is straightforward for $l(n) = 1$, but is rather subtle for $l(n)$ logarithmic. The trouble in the latter case stems from the necessity of estimating the logarithmic function $l(n)$ sufficiently rapidly after each step of computation so that the diagonalizing machine recognizing A can shut itself off within time $O(T_2(n))$.

RAM instructions can be encoded on the alphabet $\{1, 2\}$ according to Table IV. Note that numbers are represented in unary notation. A RAM program P can be encoded by concatenating the encodings of its instructions to form the string w_P. For $w \in \{1, 2\}^*$, we use P_w to denote the longest program whose encoding is an initial segment of w. Note the identity $P_{w_{P'}} = P'$. Also, for any program R and all sufficiently large integers l, there is a string w of length l such that $P_w = R$.

We can now describe the set A in the statement of Theorem 3 with the help of a function $k(\ )$ mapping RAM programs P to real numbers $k(P)$. (The specific choice of $k(\ )$ will be given later.)

If P_w with input w halts within time $k(P_w) \cdot T_2(n)$, then $w \in A$ iff P_w does not

TABLE IV

RAM Program Codes

RAM instruction	Encoding on {1, 2}
1. $X_i \leftarrow c$	$121^i21^{\hat{c}}2$
2. $X_i \leftarrow X_j + X_k$	$1^221^i21^j21^k2$
3. $X_i \leftarrow X_j - X_k$	$1^321^i21^j21^k2$
4. $X_i \leftarrow X_{X_j}$	$1^421^i21^j2$
5. $X_{X_i} \leftarrow X_j$	$1^521^i21^j2$
6. TRA m IF $X_j > 0$	$1^621^m21^j2$
7. READ X_i	1^721^i2
8. PRINT X_i	1^821^i2

Notes:

1. 1^n denotes the string $\overbrace{11 \cdots 11}^{\text{repeated } n \text{ times}}$
2. $\hat{c} = \begin{cases} 2 \cdot c & \text{if } c \geqslant 0 \\ -2 \cdot c - 1 & \text{if } c < 0 \end{cases}$

accept w. When P_w with input w does not halt within time $k(P_w) \cdot T_2(n)$, we do not care whether $w \in A$ or not.

For $l(n) = 1$, $k(P_w) = 1$, and $A = \{w \mid P_w$ with input w halts within time $T_2(|\, w \,|)$ without accepting $w\}$, but for $l(n)$ logarithmic, $k(P_w)$ may be very small for large P_w.

The proof that no RAM recognizes A in time $O(T_1(n))$ is now easy, for suppose P' recognizes A in time $c \cdot T_1(n)$. Then by Eq. (1) there is a long string w formed from $w_{P'}$ by adding suitably many 2's to the right such that $P_w = P'$ and $cT_1(|\, w \,|) < k(P') \cdot T_2(|\, w \,|)$. Thus P_w with input w halts within time $k(P_w) \cdot T_2(|\, w \,|)$, so by our description of the set A, $w \in A$ iff P_w does not accept w. But this contradicts our assumption that P' recognizes A, since $P' = P_w$. Thus no such P' can exist.

The proof that some RAM program M recognizes A within time $O(T_2(n))$ is constructive. Complete RAM–ALGOL programs for M are given in [1]. We will describe the programs here in sufficient detail to convince the knowledgeable reader that they can be written and will execute within the stated times. After describing M and showing that it operates within time $O(T_2(n))$, we will show how to choose the coefficients $k(P_w)$ so that M recognizes the set A described above. M consists of two parts: a preprocessor and a simulator.

The preprocessor first reads the input string w and stores it in the array W. The value of $n = |\, w \,|$ is also accumulated. The program P_w encoded by w is then obtained from W and stored in the four arrays opcode, operand1, operand2, and operand3. The operation code of the pth instruction of P_w is stored in opcode[p], and its operands are stored in operand1[p], operand2[p], and operand3[p]. For example, if the 13th

instruction of P_w is $X_4 \leftarrow X_{17} - X_{321}$, then opcode[13] = 3, operand1[13] = 4, operand2[13] = 17, and operand3[13] = 321. The execution time of the preprocessor is $O(n \cdot l(n)) = O(T_2(n))$, since $O(n)$ steps are taken, and each takes time $O(l(n))$.

The simulator first sets a timing counter to $T_2(n)$. This step takes time $O(T_2(n))$. The simulator then runs the program P_w, as stored in the arrays, interpretively with the string w as input. It uses another array to represent the registers used by P_w. If P_w runs for too long, the simulation is halted anyway, and w is rejected. If P_w halts on input w within time $k(P_w) \cdot T_2(n)$, then M accepts w iff P_w does not accept w.

If $l(n) = 1$, it is clear that M can simulate $T_2(n)$ steps of P_w on input w, taking no more than some fixed constant amount of time to simulate each step. Hence, in this case, the simulation takes time $O(T_2(n))$, and the total running time for M is $O(n) + O(T_2(n)) = O(T_2(n))$.

For $l(n)$ logarithmic, two problems arise. First, the straightforward way of keeping track of the simulation time is too slow. The cost of updating the timer in the ordinary way is $O(l(T_2(n)))$ at each step, so that the total execution time of the simulation would be $O(T_2(n) \cdot l(T_2(n)))$. We circumvent this problem by storing the bits of the binary representation of the time remaining for the simulation in an array called *time*. This requires that the number $T_2(n)$ initially be converted to binary. The conversion can easily be done in time $O((\log T_2(n))^2)$ by using the standard method of generating and storing powers of two up to $T_2(n)$ and, starting with the highest power, successively subtracting off those powers which leave a nonnegative difference.

During simulation, after each step of the input program has been simulated, an estimate for the time required to simulate that step is subtracted from the time array. If this difference is negative, simulation is terminated and the input w is rejected. This subtraction is carried out by a RAM–ALGOL procedure called updatetime. As explained below, the time estimate for each step is always a power of two, so its binary notation consists of a single one bit followed by zeroes. The total time required to perform all the subtractions can be estimated as follows.

Let $b_0, b_1, \ldots, b_k$ be the bits of the time array, so that initially

$$\sum_{i=0}^{k} b_i \times 2^i = T_2(n).$$

Each time a number is subtracted from the time array, one or more bits (because of "borrows") is changed in the array. But each time the bit b_i is changed, the value represented by the array is decremented by at least 2^i. Since the sum of all the decrements cannot exceed 2^{k+1}, it follows that during the entire simulation b_i is changed at most 2^{k-i+1} times. Since the cost for changing bit b_i is $O(\log i)$, $i \geqslant 2$, it follows that the total cost of all subtractions is

$$O\left(\sum_{i=2}^{k} \log i \cdot 2^{k-i+1}\right).$$

Since the infinite series $\Sigma \log i/2^i$ is convergent (say, by the ratio test), the above time is bounded by $O(2^k) = O(T_2(n))$.

The second problem for the logarithmic case is that we know of no way to compute the logarithmic $l(n)$ in time $O(l(n))$. The solution we adopt is to use an estimate of $l(n)$ which is relatively easy to compute. First we define the *norm* of a number as follows:

$$\begin{aligned} \text{norm}(n) &= 0, && \text{if } |n| < 2 \\ &= \lceil \log_2 \log_2 |n| \rceil, && \text{if } |n| \geqslant 2. \end{aligned}$$

The norm has the following two properties:

$$l(n) \leqslant 2^{\text{norm}(n)} < 2 \cdot l(n) \tag{2}$$

$$\text{norm}(m \pm n) \leqslant \text{norm}(\max(|m|, |n|)) + 1. \tag{3}$$

The first property means that $2^{\text{norm}(n)}$ is a good estimate for $l(n)$, since it is never off by more than a factor of 2. The second property means that if we know the norms of m and n, we can easily obtain an upper bound for the norms of $m + n$ and $m - n$.

In order to compute norms, we need a function to compute powers of two. In order to save computing time, the powers that have already been computed are stored in an array. Thus each power of two is computed (by doubling the next lower power of two) only once. We can estimate the total time Δ spent during the simulation computing powers of two as follows. Suppose 2^e is the highest power computed. Then the cost of generating $2^0, 2^1, \ldots, 2^e$ by successive doublings will be proportional to the cost of storing these numbers, and since $l(n)$ is logarithmic, we have

$$\Delta = O(1 + 2 + \cdots + e) = O(e^2).$$

But the Norm procedure (discussed below) never needs a power of two which exceeds twice the largest number N ever appearing in a register of the RAM being simulated. Appendix B shows that $N < 2^{4\sqrt{T_2(n)}}$, roughly speaking because the largest number can be at most doubled at each step. Thus $2^e < 2^{4\sqrt{T_2(n)}+1}$, and $e < 4\sqrt{T_2(n)} + 1$, and $\Delta = O(T_2(n))$.

A RAM–ALGOL procedure to compute norms is shown in Fig. 1. We assume it has access to a second procedure two-to-the(n) which returns 2^n. The procedure two-to-the(n) checks the array of powers to see whether 2^n has already been computed. If so, the value 2^n is copied from the array, and if not, the array is filled in by successive doublings as described above.

The procedure Norm(k, Y) in Fig. 1 returns norm(Y), provided either $k = -1$ or

$$|Y| \leqslant 2^{2^{k+1}}. \tag{4}$$

The procedure will never be called unless one of these two conditions holds. Note that if (4) is satisfied, $k + 1$ serves as an upper bound for norm(Y), and the procedure uses this fact to save time. In either case, the method of computing norm(Y) depends

Procedure Norm

RAM-ALGOL statements	*Timing*
integer procedure Norm(k, Y);	
value k,	$O(l(k))$
Y;	$O(l(Y))$
integer k,	
Y;	
begin	
comment This procedure computes the norm of Y as follows:	
(1) If $k \geqslant 0$, we assume $\mid Y \mid \leqslant 2^{2^{k+1}}$, and compute norm($Y$) in time $O(2^k)$ as described in the text.	
(2) If $k = -1$, we compute norm(Y) in time $O(l(Y) \cdot \text{norm}(Y)) = O((l(Y))^2)$;	
if $Y < 0$ *then* $Y := -Y$;	$O(l(Y))$
if $Y > 2$ *then*	$O(l(Y))$
begin	
if $k \geqslant 0$ *then*	$O(l(k))$
begin	
$k := k + 1$;	$O(l(k))$
comment two-to-the(n) is a procedure which returns 2^n;	
for $k := k - 1$ *while*	$O(l(k))$
two-to-the(two-to-the(k)) $\geqslant Y$ *do*;	$O(k + 2^k + l(Y))$
Norm $:= k + 1$	$O(l(k))$
end	
else	
begin	
Norm $:= -1$;	$O(1)$
for Norm $:=$ Norm $+ 1$ *while*	$O(l(\text{Norm}))$
two-to-the(two-to-the(Norm)) $< Y$ *do*;	$O(l(Y))$
end	
end	
else	
Norm $:= 0$	$O(1)$
end Norm;	

FIGURE 1

on the fact that norm(Y) is the least integer m such that $| Y | \leqslant 2^{2^m}$. If (4) is satisfied, the total time spent in the *for* loop is

$$O(l(2^{2^k}) + l(2^{2^{k-1}}) + \cdots + l(2^{2^0})) = O(2^k + 2^{k-1} + \cdots + 2^0) = O(2^k).$$

This does not count the time required by the function two-to-the(n) to generate new powers of two, but we argued above that during the entire simulation this time is $O(T_2(n))$.

It is easy to see that if $k = -1$, the time for Norm(k, Y) is $O((l(Y))^2)$. We can summarize the timing as follows:

$$\text{Time for Norm}(k, Y) \text{ is } \begin{cases} O((l(Y))^2), & \text{if } k = -1 \\ O(2^k), & \text{otherwise.} \end{cases} \tag{5}$$

We can now describe the method of simulation for the logarithmic case. Throughout the simulation, the current contents $X_0, X_1, X_2, \ldots$ of the registers of the simulated program are stored in an array $X[0], X[1], X[2], \ldots$. In addition, the current norms of the register values are stored in a second array norm[0], norm[1], norm[2],.... Thus norm[i] = norm($X[i]$) during the simulation. Before simulation, the preprocessor, described earlier, decodes the program P_w to be simulated into four arrays. During simulation, P_w is simulated instruction by instruction, and the X, norm, and time arrays are updated appropriately. In Fig. 2 we present as an example the section of RAM–ALGOL code for the simulation of one type of instruction; namely $X_i \leftarrow X_j + X_k$. Here the variable code has been assigned the code number of the instruction type (see Table IV), and the variables op1, op2, op3 have been assigned the three operands for the instruction; in this case, i, j, and k. Notice how the norm array is updated, using max(norm(X_j), norm(X_k)) for the value of k when calling the procedure Norm(k, Y). This is justified by the inequality (3). The procedure updatetime(n) causes the value 2^n to be subtracted from the binary time array. (Recall that this array stores the amount of simulation time remaining.) In the present case, the number $\tau = 2^{\text{norm}(X_j)} + 2^{\text{norm}(X_k)} + 2^{Nu}$ is subtracted from the time array. Here Nu has been assigned norm(u) at the start of simulation, where u is the length of w_{P_w} (i.e., the length of the part of the input w which codes a program). The number τ is a suitable estimate for the time needed to simulate the instruction $X_i \leftarrow X_j + X_k$.

A similar section of code can be written for the remaining seven instruction types.

There remain only two difficult points which must be examined for the logarithmic case. First, that the simulator does in fact run in time $O(T_2(n))$ and, second, that $k(P_w)$ can be chosen in such a way that if the input program halts within time $k(P_w) \cdot T_2(| w |)$, then simulation will be completed before the time array goes negative (which automatically termines the simulator). If the simulation is completed, of course, the simulator will accept the input w iff P_w does not accept w.

Section of Code Used to Simulate $X_i \leftarrow X_j + X_k$

RAM–ALGOL Instructions	*Timing*
else if code = 2 *then*	$O(1)$
begin	
comment $X_i \leftarrow X_j + X_k$;	
X[op1] := X[op2] + X[op3];	$O(l(i) + l(j) + l(k) + l(X_j) + l(X_k))$
$n2$:= norm[op2];	$O(l(j) + l(\text{loglog } X_j))$
$n3$:= norm[op3];	$O(l(k) + l(\text{loglog } X_k))$
temp := *if* $n2 > n3$ *then* $n2$ *else* $n3$;	$O(l(\text{loglog } X_j) + l(\text{loglog } X_k))$
norm[op1] := Norm(temp, X[op1]);	$O(l(i) + l(X_j) + l(X_k))$
updatetime($n2$);	*
updatetime($n3$);	*
updatetime(Nu)	*
end	

* Time required to update the counter is accounted for elsewhere.

FIGURE 2

Concerning the first point, we reason as follows. We have already argued that all calls to the procedure updatetime and the generation of powers of two require only time $O(T_2(n))$. Hence we need only make sure that sufficiently large values are subtracted from the time array after each instruction is simulated. Precisely, it is sufficient to find an absolute constant K such that the simulation time for each instruction is at most K times the number subtracted from the time array after that instruction is simulated. Table V shows the actual amount subtracted from the time array for each type of instruction.

Note that by Eq. (2), $2^{\text{norm}(n)} \geqslant l(n)$. Also $Nu = \text{norm}(|\, w_{P_w}\,|)$ and $|\, w_{P_w}\,|$ is an upper bound on the indices i, j, k appearing in the instructions and the constants C appearing in $X_i \leftarrow C$. From this it is not hard to verify that the simulation time for each instruction (see Fig. 2, for example) exceeds the number in the right hand column of Table V by at most a small constant factor K. The only worry is that the time required to compute norm(X) for a new value of X stored in a register would be too great. To see this is not so, note that only four instruction types require computing a new value of norm(X); namely $X_i \leftarrow C$, $X_i \leftarrow X_j \pm X_k$, and READ X_i. For the first, according to our estimate (5), the time to compute norm(C) is $O((l(C))^2)$, and $(l(C))^2$ is less than the entry $2^{2 \cdot Nu}$ in Table V. For $X_i \leftarrow X_j \pm X_k$, the simulating routine calculates norm(X_i) by calling Norm(m, X_i), where

$$m = \max(\text{norm}(X_j), \text{norm}(X_k)).$$

Again, our estimate (5) shows the time spent in the Norm procedure is $O(2^m)$, and

in this case 2^m is less than the right hand entry in Table V. Finally, for READ X_i, $\text{norm}(X_i) = 0$, since the only possible inputs are 0, 1, and 2.

To complete the proof of Theorem 3, it remains to find a constant $k(P_w)$ such that if P_w halts within time $k(P_w) \cdot T_2(|\, w \,|)$, the simulation will be completed before the time array goes negative. But the time array initially has the value $T_2(|\, w \,|)$, and the values subtracted are the values in the right hand column of Table V, after

TABLE V

Instruction	Amount subtracted from the time array
$X_i \leftarrow C$	$2^{2 \cdot Nu}$
$X_i \leftarrow X_j + X_k$	$2^{\text{norm}(X_j)} + 2^{\text{norm}(X_k)} + 2^{Nu}$
$X_i \leftarrow X_j - X_k$	$2^{\text{norm}(X_j)} + 2^{\text{norm}(X_k)} + 2^{Nu}$
$X_i \leftarrow X_{X_j}$	$2^{\text{norm}(X_j)} + 2^{\text{norm}(X_{X_j})} + 2^{Nu}$
$X_{X_i} \leftarrow X_j$	$2^{\text{norm}(X_j)} + 2^{\text{norm}(X_i)} + 2^{Nu}$
TRA m IF $X_j > 0$	$2^{\text{norm}(X_j)} + 2^{Nu}$
READ X_i	2^{Nu}
PRINT X_i	$2^{\text{norm}(X_i)} + 2^{Nu}$

Note: $Nu = \text{norm}(|\, w_{P_w} \,|)$ and $|\, w_{P_w} \,|$ is an upper bound on all constants and addresses appearing in P_w.

each instruction is simulated. According to Eq. (2) and the execution times shown in Table I, these values exceed the execution times for the instructions by at most a factor of 2, plus $2^{2 \cdot Nu}$. An upper bound for the amount subtracted for the ith instruction is thus $(2 + 2^{2Nu})\, t_i$, where t_i is the execution time of the ith instruction. Hence we have

$$(2 + 2^{2Nu})E \geqslant T_2(|\, w \,|), \tag{6}$$

where E is the execution time of that portion of the computation of P_w which is simulated before the time array goes negative.

From (6) we have $E \geqslant 1/(2 + 2^{2Nu}) \cdot T_2(|\, w \,|)$, so we can choose $k(P_w) = 1/(2 + 2^{2Nu})$.

APPENDIX A: RAM–ALGOL PROGRAM TO SIMULATE RASP PROGRAM P

RAM–ALGOL Statements	timing
begin	
comment Declarations for program R;	
integer array MEMORY;	
integer dummy,	
AC,	
IC,	
j;	
Boolean simulating;	
comment The first thing we do is to initialize the first p elements of MEMORY, so as to mimic the initial configuration of P;	
MEMORY[0] := $\overline{\text{initial value}_0}$;	$O(1)$
MEMORY[1] := $\overline{\text{initial value}_1}$;	$O(1)$
⋮	
MEMORY[$p-1$] := $\overline{\text{initial value}_{p-1}}$;	$O(1)$
AC := IC := 0;	$O(1)$
comment The main part of the simulator is the following loop, which is executed once for each instruction simulated. If t_i is the execution time of a single instruction of P, then the pass through this loop simulating the execution of that instruction takes time $O(t_i)$. Thus if $I(n)$ is the number of instructions executed by P on inputs of length n, then $T(n) = \sum_{i=1}^{I(n)} t_i$, and the execution time of R is $\sum_{i=1}^{I(n)} O(t_i) = O\left(\sum_{i=1}^{I(n)} t_i\right) = O(T(n))$;	
simulating := *true*;	$O(1)$
for dummy := 0	$O(1)$
while simulating *do*	$O(1)$

```
begin
  integer opcode;
  opcode := MEMORY[IC];                                   O(l(IC) + l(opcode))
  if opcode ≤ 0 ∨ opcode ≥ 8 then                         O(l(opcode))
    simulating := false                                   O(1)

  else begin
    j := MEMORY[IC + 1];                                  O(l(IC) + l(j))
    IC := IC + 2;                                         O(l(IC))
    if opcode = 1 then begin                              O(1)
      comment The execution time of
        LOD, j is l(IC) + l(j);
      AC := j                                             O(l(j))
    end else if opcode = 2 then begin                     O(1)
      comment The execution time of
        ADD, j is l(IC) + l(j) + l(AC) + l(X_j);
      if j < 0 then simulating := false                   O(l(j))
      else AC := AC + MEMORY[j]                           O(l(j) + l(AC) + l(X_j))
    end else if opcode = 3 then begin                     O(1)
      comment The execution time of
        SUB, j is l(IC) + l(j) + l(AC) + l(X_j);
      if j < 0 then simulating := false                   O(l(j))
      else AC := AC − MEMORY[j]                           O(l(j) + l(AC) + l(X_j))
    end else if opcode = 4 then begin                     O(1)
      comment The execution time of
        STO, j is l(IC) + l(j) + l(AC);
      if j < 0 then simulating := false                   O(l(j))
      else MEMORY[j] := AC                                O(l(j) + l(AC))
    end else if opcode = 5 then begin                     O(1)
      comment The execution time of
        BPA, j is l(IC) + l(j) + l(AC);
      if j < 0 then simulating := false                   O(l(j))
      else if AC > 0 then IC := j                         O(l(j) + l(AC))
    end else if opcode = 6 then begin                     O(1)
      comment The execution time of
        RD, j is l(IC) + l(j) + l(input);
      if j < 0 then simulating := false                   O(l(j))
      else read(MEMORY[j])                                O(l(j) + l(input))
    end else begin
      comment The execution time of
        PRI, j is l(IC) + l(j) + l(X_j);
```

```
        if j < 0 then simulating := false                    O(l(j))
        else print(MEMORY[j])                                O(l(j) + l(X_j))
      end
    end
  end of the simulation loop;
end of program R;
```

APPENDIX B

We wish to obtain an upper bound on the largest number computable by a RAM program (for a RAM with logarithmic cost function) as a function of execution time. We must add some restrictions, however, since for any integer C there is a RAM program with execution time 1 that computes C, namely, $X_0 \leftarrow C$. But the length of our encoding of this program on $\{1, 2\}$ grows linearly with C. Also the short program READ X_0 can "compute" the number 2^T in time T by reading in a large input. To circumvent these two problems we define $M(T, n)$ to be the largest number appearing in any register after a computation of duration T or less by any RAM program (with logarithmic cost function) whose encoding has length no greater than n, provided the inputs are 0, 1, or 2. This restriction on the inputs corresponds to the restrictions in Theorem 3.

THEOREM. *For* $n > 0$, $M(T, n) < n \cdot 2^{2 \cdot \sqrt{T}}$, *and* $M(T, 0) = 0$.

Proof. The proof will proceed by induction on the number of instruction executions during the computation. In order to facilitate breaking the proof down into cases, we define $M^i(T, n)$ to be the largest number computable in time T by a program of length no greater than n, where the last instruction executed is an instruction of type i. We note that $M(T, n) = \max_{1 \leqslant i \leqslant 8} M^i(T, n)$, so that to prove $M(T, n) < n \cdot 2^{2 \cdot \sqrt{T}}$, it will suffice to show that for $i = 1,..., 8$, $M^i(T, n) < n \cdot 2^{2 \cdot \sqrt{T}}$. As the basis of our induction, it is trivially true that $M(T, 0) = M(0, n) = 0$, since nothing can be computed with no program or with no time. (Recall our convention that a RAM computation begins with all registers set to zero.) Thus $n > 0 \Rightarrow M(0, n) = 0 < n = n \cdot 2^0 = n \cdot 2^{2 \cdot \sqrt{0}}$. As induction hypothesis, assume that for all $t < T$, $M(t, n) < n \cdot 2^{2 \cdot \sqrt{t}}$. We will now show that $M^i(T, n) < n \cdot 2^{2 \cdot \sqrt{T}}$ for $i = 1,..., 8$, and thus that $M(T, n) < n \cdot 2^{2 \cdot \sqrt{T}}$, completing the induction.

Case 1. Last instruction executed is $X_i \leftarrow C$. Execution time of the last instruction is 1. Either the last instruction produced the largest number, or it did not. Therefore $M^1(T, n) = \max(M(T - 1, n), C) < n \cdot 2^{2 \cdot \sqrt{T}}$, since $C < n$ and $M(T - 1, n) < n \cdot 2^{2 \cdot \sqrt{T-1}} < n \cdot 2^{2 \cdot \sqrt{T}}$.

Case 2. Last instruction executed is $X_i \leftarrow X_j + X_k$. Execution time of the last instruction is $l(X_j) + l(X_k)$. To prove that $M^2(T, n) < n \cdot 2^{2 \cdot \sqrt{T}}$, we will assume that $M^2(T, n) \geqslant n \cdot 2^{2 \cdot \sqrt{T}}$ and derive a contradiction. The largest number that could have been computed prior to the last instruction is $M(T - (l(X_j) + l(X_k)), n)$, and from the induction hypothesis we know that

$$M(T - (l(X_j) + l(X_k)), n) < n \cdot 2^{2 \cdot \sqrt{T - (l(X_j) + l(X_k))}}.$$

Since $M^2(T, n) \geqslant n \cdot 2^{2 \cdot \sqrt{T}}$, $M^2(T, n) > M(T - (l(X_j) + l(X_k)), n)$, so $M^2(T, n)$ must have been computed in the last step.

(*) Thus $X_j + X_k = M^2(T, n) \geqslant n \cdot 2^{2 \cdot \sqrt{T}}$.

But since X_j and X_k must have been computed before the last step, we know that

$$X_j + X_k < 2 \cdot n \cdot 2^{2 \cdot \sqrt{T - (l(X_j) + l(X_k))}}.$$

Combining these inequalities, we get

$$n \cdot 2^{2 \cdot \sqrt{T}} < 2 \cdot n \cdot 2^{2 \cdot \sqrt{T - (l(X_j) + l(X_k))}}$$

$$\Rightarrow 2 \cdot \sqrt{T} < 1 + 2 \cdot \sqrt{T - (l(X_j) + l(X_k))},$$

canceling n and taking logs

$$\Rightarrow 4 \cdot T < 1 + 4 \cdot \sqrt{T - (l(X_j) + l(X_k))} + 4 \cdot T - 4 \cdot (l(X_j) + l(X_k)),$$

squaring both sides

$$\Rightarrow 4 \cdot (l(X_j) + l(X_k)) < 1 + 4 \cdot \sqrt{T - (l(X_j) + l(X_k))},$$

transposing and canceling $4 \cdot T$

$$\Rightarrow 4 \cdot (l(X_j) + l(X_k)) < 1 + 4 \cdot \sqrt{T}, \quad \text{since } T > T - (l(X_j) + l(X_k))$$

$$\Rightarrow l(X_j) + l(X_k) < \tfrac{1}{4} + \sqrt{T}, \quad \text{since } 4 > 0$$

$$\Rightarrow l(X_j + X_k) < \tfrac{1}{4} + \sqrt{T}, \quad \text{by subadditivity}$$

$$\Rightarrow \log_2(X_j + X_k) < \tfrac{1}{4} + \sqrt{T}, \quad \text{since } \log_2 x \leqslant l(x)$$

$$\Rightarrow X_j + X_k < 2^{1/4} \cdot 2^{\sqrt{T}}, \quad \text{exponentiating both sides}$$

$$\Rightarrow n \cdot 2^{2 \cdot \sqrt{T}} < 2^{1/4} \cdot 2^{\sqrt{T}}, \quad \text{combining with (*)}$$

$$\Rightarrow n \cdot 2^{\sqrt{T}} < 2^{1/4}, \quad \text{canceling } 2^{\sqrt{T}}$$

$$\Rightarrow n = 0 \text{ or } n = 1 \text{ and } T = 0, \quad \text{since } 1 < 2^{1/4} < 2.$$

But we have assumed that both $n > 0$ and $T > 0$, so we have a contradiction.

Thus $M^2(T, n) < n \cdot 2^{2 \cdot \sqrt{T}}$.

Case 3. $X_i \leftarrow X_j - X_k$. The analysis for this case is identical to Case 2.

Case 4. $X_i \leftarrow X_{X_j}$. This instruction computes no new values, so

$$M^4(T, n) \leqslant M(T - 1, n) < n \cdot 2^{2 \cdot \sqrt{T}}.$$

Case 5. $X_{X_i} \leftarrow X_j$. Again, no new value is computed, so

$$M^4(T, n) < n \cdot 2^{2 \cdot \sqrt{T}}.$$

Case 6. TRA m IF $X_j > 0$. No new value, so $M^6(T, n) < n \cdot 2^{2 \cdot \sqrt{T}}$.

Case 7. READ X_i. Since the input value is 0, 1, or 2, l(input) = 1. Thus $M^7(T, n) = \max(M(T - 1, n), 2) < n \cdot 2^{2 \cdot \sqrt{T}}$.

Case 8. PRINT X_i. No new value, so $M^8(T, n) < n \cdot 2^{2 \cdot \sqrt{T}}$. This completes the induction step, and the theorem is proved. ∎

In the text, we refer to $N = M(T_2(n), n)$. Since $n \leqslant T_2(n)$, and since $\log_2(T_2(n)) < 2 \cdot \sqrt{T_2(n)}$, we get

$$\begin{aligned} N = M(T_2(n), n) &< n \cdot 2^{2 \cdot \sqrt{T_2(n)}} \\ &< 2^{\log_2 n} \cdot 2^{2 \cdot \sqrt{T_2(n)}} \\ &< 2^{2 \cdot \sqrt{T_2(n)} + 2 \cdot \sqrt{T_2(n)}} \\ &< 2^{4 \cdot \sqrt{T_2(n)}}. \end{aligned}$$

References

1. Stephen A. Cook and Robert A. Reckhow, Diagonal Theorems for Random Access Machines, Technical Report No. 42, Department of Computer Science, University of Toronto, June 1972.
2. Stephen A. Cook, Linear Time Simulation of Deterministic Two-way Pushdown Automata, Proceedings of IFIP Congress 71, Foundations of Information Processing.
3. Peter Naur (ed.), Revised Report on the Algorithmic Language ALGOL 60, C.A.C.M. 6, 1 (1963), 1–17.
4. J. Hartmanis, Computational complexity of random access stored program machines, *Mathematical Systems Theory* **5**, 3 (1971), 232–245.
5. John Hopcroft and Jeffrey Ullman, "Formal Languages and their Relation to Automata," Addison-Wesley, 1969.
6. F. C. Hennie and R. E. Stearns, Two-tape simulation of multi-tape turing machines, *J.A.C.M.* **13**, 4 (1966), 533–546.