Two-Tape Simulation of Multitape Turing Machines

F. C. HENNIE

Massachusetts Institute of Technology, Cambridge, Massachusetts*

AND

R. E. STEARNS

General Electric Company,† Schenectady, New York

Abstract. It has long been known that increasing the number of tapes used by a Turing machine does not provide the ability to compute any new functions. On the other hand, the use of extra tapes does make it possible to speed up the computation of certain functions. It is known that a square factor is sometimes required for a one-tape machine to behave as a two-tape machine and that a square factor is always sufficient.

The purpose of this paper is to show that, if a given function requires computation time T for a k-tape realization, then it requires at most computation time $T \log T$ for a two-tape realization. The proof of this fact is constructive; given any k-tape machine, it is shown how to design an equivalent two-tape machine that operates within the stated time bounds. In addition to being interesting in its own right, the trade-off relation between number of tapes and speed of computation can be used in a diagonalization argument to show that if $T(n)$ and $U(n)$ are two time functions such that

$$\inf \frac{T(n) \log T(n)}{U(n)} = 0$$

then there exists a function that can be computed within the time bound $U(n)$ but not within the time bound $T(n)$.

1. *Introduction*

The study of computability by computer-like devices was initiated by Turing [1], who postulated that the functions that can be mechanically evaluated are precisely those functions that can be computed by a finite-state device with a single unbounded read-write tape. Such a device is commonly called a Turing machine. When slight variations in input-output procedure are used, Turing machines may be applied to other problems such as generating sequences or recognizing sets. Thus functions, sequences and sets can be classified as "computable" or "noncomputable" depending on whether or not they can be defined by an appropriate Turing machine.

With the advent of the modern high-speed computer, interest in computability has spread to questions concerning the "difficulty" or "complexity" of a calculation. One fruitful measure of the complexity of a computation is the number of time units needed to carry out that computation. For any function $T(n)$ of integers into integers, we say that a given function, sequence or set is $T(n)$-computable if and only if there is some (appropriately modified) Turing machine which, depending on the context, either computes the nth term of a sequence in $T(n)$ operations or processes an input sequence of length n within $T(n)$ operations. Three modifications of

* Department of Electrical Engineering

† Research and Development Center

Journal of the Association for Computing Machinery, Vol. 13, No. 4 (October, 1966), pp. 533–546

the Turing machine for which this complexity criterion has been studied are discussed later. The common feature of these models is that they all contain a finite-state device which controls one or more unbounded tapes. The use of more than one tape does not give a machine any extra power, but it does enable some calculations to be completed in fewer operations, and is a step toward making the Turing machine model more like present computers.

A central problem in the theory of computational complexity is to determine how much faster a problem can be done on a k_1-tape machine than on a k_2-tape machine. Not only is this problem interesting in its own right, but certain proofs based on Cantor diagonal techniques depend on the speed with which one is able to simulate a sequence of machines with arbitrary numbers of tapes by a single machine that must have a fixed number of tapes. Such applications are discussed in later sections where the specific models are discussed.

Hartmanis and Stearns [2] have shown that any k-tape machine can be simulated by a one-tape machine whose computation time is no greater than the square of the computation time of the k-tape machine. Hennie [3] has shown that there are cases in which reducing the number of tapes from two to one actually requires a squaring of the computation time. Thus more efficient simulation techniques can only be obtained by using more than one tape.

The object of this paper is to describe a scheme whereby n operations of a many-tape Turing machine can be simulated by $n \log n$ operations of a two-tape Turing machine. This improvement over the one-tape case allows a corresponding improvement in results proved by diagonal techniques. These applications are discussed in later sections.

2. *Preliminary Considerations*

The purpose of this section is to describe our objectives more explicitly. However, the definitions and proofs must of necessity be somewhat informal because a completely rigorous treatment would require a prohibitive amount of space and would obscure the simple principles upon which the construction is based.

The *basic multitape Turing machine* model we are trying to simulate may be described as a computing device that has a finite automaton as a control unit. Attached to this control unit is a fixed number of tapes which are linear, unbounded at both ends and ruled into an infinite sequence of squares. The control unit has one reading head assigned to each tape, and each head rests on a single square of the assigned tape. There are a finite number of distinct symbols which can appear on the tape squares. Each combination of symbols under the reading heads, together with the state of the control unit, determines a unique machine operation. A *machine operation* consists of overprinting a symbol on each tape square under the heads, shifting each tape independently either one square left, one square right or no squares and changing the state of the control unit. The machine is then ready to perform its next operation as determined by the tapes and control state. If a given operation does not call for any tape shift, change of tape symbol or change of state, then the machine is said to have *stopped.*

In order to use a machine for calculations, one must add some means of supplying input data to the machine and obtaining output data from it. In later sections several ways of doing this are considered. However, for the purposes of simulation,

the particular input output convention used is irrelevant; only the basic machine operations are important. Thus the first concern is with the simulation of the basic model; the problem of supplying input data and obtaining output data is ignored. As usual, by "simulation" we mean simply a process of imitating, one after the other, the operations performed by a given machine.

3. *The Simulation Method*

The Approach Used. The purpose of this section is to describe a fast method of simulating the behavior of a given k-tape machine M_k with an appropriate two-tape machine M_2. The basic unit of the simulation process, called a *step*, consists in imitating the actions that M_k performs during a single move. At the beginning of each step in its simulation, M_2 must have on its tapes an up-to-date record of the patterns currently appearing on M_k's tapes, and of the locations of these patterns with respect to M_k's reading heads. During the course of a typical step, M_2 must not only update its record of M_k's tape pattern, but in general it must also rearrange its record of these patterns so as to provide easy access to the symbols that M_k will scan next.

One of M_2's tapes will be called the *storage tape*. We imagine that this tape is divided lengthwise into k *tracks*, one for each of M_k's tapes. Each track is in turn divided into an upper and lower *level*, as shown in Figure 1. Each of the resulting small squares may contain any one of M_k's tape symbols, together with some special "marking symbols." The head that scans the storage tape is assumed to read all the squares in one column in one operation. This specialized way of viewing the storage tape is in keeping with the traditional Turing machine model, for the vector of symbols that appears in a given column of this tape may be regarded as a single symbol in an expanded tape alphabet.

One column of the storage tape is designated the *home column*. The columns to the left and right of the home column are grouped into a number of *storage areas*, as indicated in Figure 1. The ith storage area on each side of the home column consists of exactly 2^{i-1} columns. Thus the number of columns in the ith left (or right) storage area is exactly one greater than the total number of columns in the first $i-1$ left (or right) storage areas. At the beginning of a simulation the boundaries of the storage areas are not marked in any way; special symbols marking these boundaries are supplied by M_2 as needed during the course of the simulation.

Although this organization of the storage tape may appear strange at first, it provides a framework within which the behavior of M_k can be simulated efficiently. The lower level of each track serves as the primary means of storing the symbols that appear on M_k's tapes. In particular, at the beginning of a simulation, all of M_k's tape symbols are recorded in the lower levels of M_2's storage tracks, with the initially scanned symbols appearing in the home column. After each step in its simulation, M_2 realigns its recorded patterns so that the next symbols to be scanned by M_k are placed in the home column. This realignment is accomplished by copying the symbols in question onto M_2's second tape, and then copying them back into the desired locations on the storage tape.

If M_2 were to shift all of the symbols in each of its patterns at each step in the simulation, the time needed to simulate n steps of M_k's computation might be as large as n^2. In order for the simulation time to be decreased, M_2 will be designed so

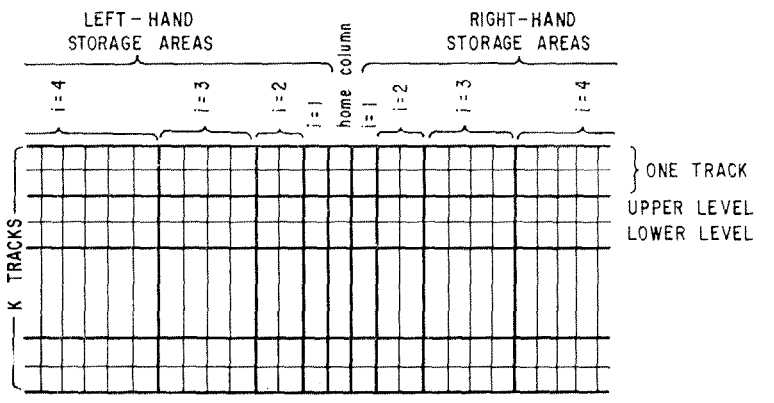

FIG. 1. Storage tape

that during a typical step in the simulation it leaves the bulk of each recorded pattern in place and shifts only a small segment near the home column. Such limited shifts would require that certain shifted segments of the tape be placed on top of certain unshifted segments. These excess symbols are accommodated in the upper level of the track in question until such time as they can be reincorporated into the lower level by a more extensive realignment. For each segment of excess symbols, there must also be a corresponding gap on the other side of the home square. These gaps remain until refilled by a more extensive realignment.

The number of symbols to be shifted at each step is determined by the sizes of the storage areas and by the number and locations of the excess symbols currently being stored in the upper level. The sizes of the storage areas have been chosen so that the ith upper storage area will just hold the overflow resulting from a realignment of the symbols in the first $i-1$ storage areas. In this way the number of steps needed to perform each realignment is kept to a minimum, and the simulation can be accomplished quite efficiently. In particular, it will be shown that n operations of M_k's computation require at most $n \log n$ operations to simulate on M_2.

Description of the Simulation. At the beginning of each step in the simulation the current tape patterns of M_k are to be arranged within their respective tracks on the storage tape of M_2 so that the symbols currently scanned by M_k are all within the home column. Thus M_2 needs only to examine the home column to make all the symbol changes required by the current step in the simulation, and to determine the directions in which M_k will move its various tapes. To prepare for the following step in the simulation, however, M_2 must rearrange its recorded tape patterns so that the next symbols to be scanned by M_k are all in the home column. This rearrangement lies at the heart of the simulation procedure, and will be discussed in considerable detail. Because the process is basically the same for each of the k-tape patterns, the discussion that follows is restricted to one of M_k's tapes and to the corresponding track on M_2's storage tape.

At any given time, only some of the squares within a typical track will be used to store symbols from the corresponding tape pattern of M_k. Squares that do store symbols from M_k's tape pattern will be called *full*; squares that do not will be called *empty*. At the beginning of each step in a simulation, each of M_k's tape patterns is

to be recorded within the corresponding track on the storage tape of M_2 in accordance with the following rules.

1. Within any given level of any given storage area, either every square is full or every square is empty. Note that a distinction must be made between empty squares and squares that are being used to store blanks from M_k's tape. If the tapes of M_k are initially blank, then M_2's storage tape is also initially blank, but these blanks must represent the situation in which the lower level of each track stores blanks from M_k's tapes, while the upper level of each track is empty.

2. Within any given track, if the upper level of the ith left (right) storage area is full, then the lower level of the same area must also be full, and both levels of the ith right (left) area must be empty. Conversely, if the lower level of the ith left (right) area is empty, then the upper level of the same area must also be empty, and both levels of the ith right (left) area must be full. The upper level of the home column is always empty; the lower level, hereafter referred to as the *home square*, is always full. At the beginning of a simulation, the upper level of every area will be empty, and the lower level of every area will be full.

3. The symbols stored within any given track must be arranged in such a way that the corresponding pattern on M_k's tape can be obtained as follows. Starting with the leftmost nonblank symbol in the given track, read through the various storage areas from left to right, ignoring empty areas. To the left of the home square, all the symbols stored in the lower level of a given area are to be read before those in the upper level. To the right of the home square this order is reversed.

As an example, Figure 2a shows a typical configuration of symbols that might appear within one track of the storage tape. Here the letters represent symbols from M_k's tape alphabet, while blanks represent empty squares. The reader may verify that this configuration satisfies rules 1 and 2. According to rule 3, it represents the pattern . . . t h e q u i c k g r a y f o x j u m p . . . on M_k's tape, with the symbol "y" under M_k's reading head.

Suppose that at the beginning of some step in its simulation, one of the tracks on M_2's storage tape contains the configuration of symbols shown in Figure 2a, and that during its next move M_k overprints its scanned symbol with an "m" and shifts the tape in question to the right. In order to imitate this action, M_2 must change the symbol in the home square to an "m" and then rearrange the entire configuration of symbols so that the symbol "a" appears in the home square and rules 1 through 3 are satisfied. This can be done quickly by simply shifting the symbol "a" into the home square and putting the displaced symbol "m" in the upper level of the first right-hand storage area, as shown in Figure 2b.

Now suppose that the next step in M_k's computation requires overprinting the scanned symbol with an "i" and shifting the tape one more square to the right. In this case, the rearrangement of the symbols on M_2's tape is not quite so simple. The symbol that must be moved into the home square currently appears in the upper level of the third left-hand storage area. It is not possible simply to shift this symbol into the home square and leave the rest of the third storage area intact, for each level of a storage area must be either completely full or completely empty (rule 1). The quickest way to restore proper order is to remove all of the symbols from the upper level of the third storage area and relocate them in the lower levels of the first and second left-hand storage areas. Because of the way in which the

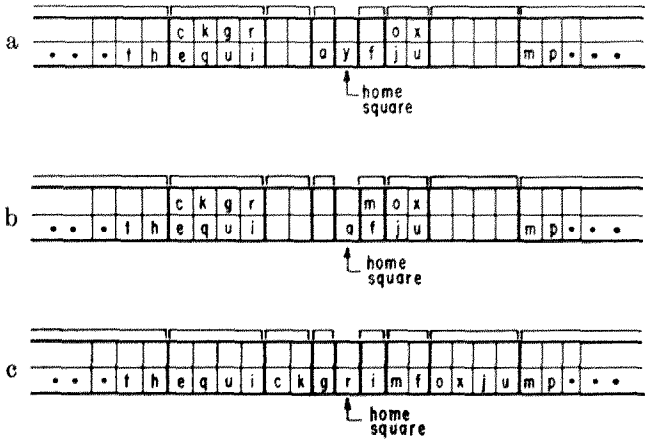

FIG. 2a–c. Two successive cleanups

lengths of the storage areas have been chosen, three of these symbols will just fill up the first and second areas, leaving the fourth symbol "r" for the home square.

In order to satisfy rule 2, the upper levels of the first two right-hand storage areas must now be emptied out and the lower level of the third right-hand storage area must be filled in. Fortunately, the number of symbols that appears in the upper level of the first two areas, together with the symbol "i" displaced from the home square, just matches the number of available squares in the lower level of the third right-hand storage area. Thus the symbols can be reshuffled as shown in Figure 2c to create a new pattern that satisfies rules 1 and 2 and accurately reflects M_k's new tape pattern.

With the examples of Figure 2 in mind, we shall now state general rules for rearranging symbols within one of M_2's storage tracks. If it is assumed that the appropriate overprinting has already been done, and that M_2 is to imitate a shift of M_k's tape toward the right, the procedure is as follows.

4. Search to the left from the home square until the first nonempty square is found. Let i be the number of the left-hand storage area that contains this square.

5. If the upper track of the ith left-hand storage area is full, collect all the symbols in that level and relocate them (in the proper order) in the lower levels of the first $i-1$ left-hand storage areas, with the rightmost symbol going in the home square. If the upper level of the ith left-hand storage area is empty, collect the symbols in the lower level and relocate them in the same way.

6. Collect all the symbols in both levels of the first $i-1$ right-hand storage areas, together with the one symbol displaced from the home square, and relocate them (in the proper order) in the lower levels of the first $i-1$ right-hand storage areas, with the excess symbols going in the ith storage area. The lower level of this area is to be used if it is available; otherwise the upper level is to be used.

If M_k should shift its tape to the left instead of to the right, it is only necessary to interchange the words "left" and "right" in these three rules. In either case the process of rearranging the symbols will be referred to as a *cleanup of order i.* The transition from Figure 2b to Figure 2c represented a third-order cleanup. In this case, the upper level of the third left-hand storage area was full, so the first part of

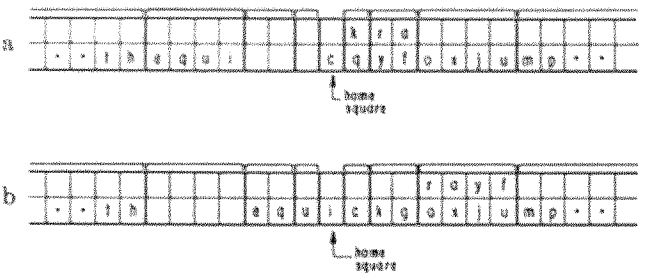

FIG. 3a–b. A third-order cleanup

rule 5 was applied. The transition from Figure 2a to Figure 2b represented a first-order cleanup in which the second part of rule 5 was applied. To illustrate a less trivial application of rule 5 to the case when the upper level is empty, Figure 3 has been provided; it shows a third-order cleanup of this type. We take it as self-evident that the procedure works; i.e., that each cleanup results in a configuration that satisfies rules 1 through 3 and that is suitable for a subsequent application of rules 4 through 6. The reader may easily verify this himself by experimentation with a few examples.

It must be remembered that we have been considering only one of M_2's tapes and the corresponding track on M_2's storage tape. At each step in its simulation M_2 must in fact clean up each track on its storage tape, each cleanup being governed by the rules given above. Of course, the cleanups performed on the various tracks at any given step in the simulation need not be of the same order. In general they will not be, since the tapes of M_2 may experience quite different sequences of left and right shifts.

Implementation of the Cleanup Procedure. Having explained the principles of our cleanup scheme, we shall now indicate how this scheme can be accomplished by a suitable two-tape Turing machine, M_2. Again suppose that the current operation of M_2 calls for the shifting of some particular tape to the right. After overprinting the symbol originally in the home square, the machine M_2 moves its storage-tape head toward the left until it comes to the first square containing a symbol from the tape alphabet of M_2. For each square that the storage-tape head moves over, M_2 marks off one square on its second or *copy tape*. Thus when the storage-tape head reaches the desired symbol, the number of squares marked off on the copy tape will just equal 2^{i-1}, where i is the number of the storage area containing the symbol.

M_2 now proceeds to transfer the symbols from the appropriate level of the ith storage area onto the marked portion of the copy tape. When the marked portion of the copy tape is filled, the left end of the ith area will have been reached. If this end of the storage area has not previously been marked, a special marking symbol is now placed there. The machine then retraces its steps until its storage head has returned to the left end of the $(i-1)$-st storage area, and proceeds to transfer the copied symbols into the lower levels of the first $i-1$ areas. When the home square is reached there will be one symbol left over; this symbol is interchanged with the symbol currently in the home square.

M_2 next transfers the contents of the first $i-1$ right-hand storage areas onto the copy tape. Of course, in order for M_2 to transfer these symbols in their proper

order, it must have some way of identifying the ends of the various storage areas. As will be seen, the boundaries of the first $i-1$ storage areas on each side of the home square must already have been marked as the result of previous lower order cleanups. Once the contents of the first $i-1$ right-hand storage areas have been placed on the copy tape, M_2 proceeds to transfer them back into the lower levels of the first $i-1$ right-hand storage areas, and into the appropriate level of the ith area. The machine can identify the ith area as being the first one that has an empty upper level. When all the symbols have been transferred, the right-hand end of the ith storage area will have been reached, and M_2 marks that point if it has not already been marked. Note that at the end of the ith-order cleanup M_2 is guaranteed to have the ends of the ith storage area marked. Since an ith-order cleanup must always be preceded by cleanups of all smaller orders (see Lemma 2), the corresponding area end markers required to locate the lesser storage areas must already be on the tape, and so the transfers can indeed be carried out as advertised.

M_2 now returns to the home square and erases the marks from its copy tape. The cleanup has been completed, and the machine can go on to clean up the next tape track, or to begin the next step in the simulation. Note that in the process of performing a single ith-order cleanup, M_2 need not visit any tape squares outside the ith storage areas. Furthermore, the entire process consists of making a small, fixed number of passes over each of the storage areas involved, and over an equivalent segment of the copy tape. Therefore, the number of moves needed to perform an ith-order cleanup is proportional to the number of squares contained within the first i storage areas on each side of the home square.

The Time Required for Simulation. We shall now turn to the problem of determining the amount of time that M_2 needs in order to simulate the first n moves in a computation performed by M_k. In particular, it will be shown that this time is proportional to $n \log n$. This will be done through a series of easy lemmas, *each of which refers to the cleanup operations performed on a single track.*

LEMMA 1. *The number of operations of M_2 needed to perform an i-th-order cleanup is less than $\beta 2^i$, where β is a constant independent of i.*

PROOF. It was noted earlier that the number of operations needed to perform an ith-order cleanup is at most proportional to the number of squares contained within the first i storage areas on both sides of the home square. But this number of squares is just

$$1 + 2\sum_{j=1}^{i} 2^j = 2^{i+2} - 1 < 4 \cdot 2^i,$$

and the total time is thus proportional to 2^i.

LEMMA 2. *Prior to the first cleanup of order greater than i, and between any two cleanups of order greater than i, there must be at least one cleanup of order i.*

PROOF. Any cleanup of order greater than i must empty out the upper levels of each of the two ith storage areas. These areas will also be empty at the beginning of a simulation. Before a subsequent cleanup of order greater than i can occur, the upper level of one of the two ith areas must be full, for otherwise a cleanup of order i or less would be performed according to rules 4 and 2. But the upper level of an ith storage area can only be filled by a cleanup of order i.

LEMMA 3. *Prior to the first cleanup of order greater than i, and between any two cleanups of order greater than i, there must be at least $2^{i-1}-1$ cleanups of order less than or equal to i.*

PROOF. The proof is by induction on i. If $i = 1$, the statement of the lemma is trivially true. Assume, therefore, that the statement is true for all i less than some integer m. Then consider two cleanups c_1 and c_2 of order greater than m. By Lemma 2, there must be at least one mth-order cleanup c_3 between c_1 and c_2. But by assumption there are at least $2^{m-2} - 1$ cleanups between c_1 and c_3, and at least $2^{m-2} - 1$ cleanups between c_3 and c_2. Thus the total number of cleanups between c_1 and c_2 (exclusive) is at least $2^{m-1} - 1$. Since a similar argument applies if c_2 is the first cleanup of order greater than m, the lemma follows by induction.

LEMMA 4. *Let n_i denote the number of cleanups of order i performed in the process of simulating the first n steps of the computation of M_k. Then $n_i \leq n/2^{i-2}$.*

PROOF. Prior to the first cleanup of order i, and between any two successive cleanups of order i, there are by Lemma 3 at least $2^{i-2} - 1$ cleanups of lesser order. Therefore, at most one cleanup out of any 2^{i-2} consecutive cleanups can be of order i, and out of n consecutive cleanups at most $n/2^{i-2}$ can be of order i.

LEMMA 5. *$n_i = 0$ for $i > 2 + \log_2 n$.*

PROOF. If $i > 2 + \log_2 n$, then by Lemma 4

$$n_i \leq \frac{n}{2^{i-2}} < \frac{n}{2 \log_2 n} = 1.$$

Because n_i must be an integer, $n_i = 0$.

LEMMA 6. *The total number of operations that M_2 needs to perform all the cleanups on a single track in the process of simulating the first n operations of M_k is at most $4\beta n(2 + \log_2 n)$.*

PROOF. The time in question is equal to $T = \sum_{i=1}^{\infty} t_i n_i$, where t_i is the time required for an ith-order cleanup. But according to Lemma 5, the summation need only be extended to $i = \log_2 n + 2$. Then applying Lemmas 1 and 4, one has

$$T = \sum_{i=1}^{\log n+2} t_i n_i \leq \sum_{i=1}^{\log n+2} \beta 2^i n_i \leq \sum_{i=1}^{\log n+2} \beta \frac{2^i n}{2^{i-2}} = 4\beta n(\log n + 2),$$

which completes the proof.

Finally, considering the cleanups performed on all tracks, we have:

THEOREM 1. *The number of operations of M_2 needed to simulate n operations of M_k is at most $\alpha n \log_2 n$ for $n > 1$, where α is a constant independent of n.*

PROOF. Multiplying the maximum number of operations needed for each track by the total number of tracks gives

$$4\beta kn(\log_2 n + 2) \leq 12\beta kn \log_2 n = \alpha n \log_2 n.$$

It can also be shown that there is a constant α' such that, in the worst case, the number of operations of M_2 needed to simulate n operations of M_k is at least $\alpha' n \log_2 n$. Thus the bound of Theorem 1 is the best that can be obtained for the simulation method being considered here. The interested reader may verify that this worst-case situation occurs when some tape of M_k is shifted in the same direction at every step in the computation.

4. *Application to Sequence Generators*

A *sequence generator* is a basic multitape Turing machine that has output values assigned to certain of the states in its control unit. It begins its computation in a [illegible] state with all its tapes blank. As a result of its subsequent

operations, it will from time to time visit states to which output values have been assigned. The (infinite) sequence of these output values is called the sequence that the machine *generates*. A particular infinite sequence ω is called *$T(n)$-computable* if and only if there is some sequence generator M that produces the nth member of ω within $T(n)$ or fewer operations. This machine M is then said to *operate within time $T(n)$*.

THEOREM 2. *If a sequence ω can be generated by a k-tape machine within time $T(n)$, and $T(n) \geq n$, then it can be generated by a two-tape machine within time $T(n) \log_2 T(n)$.*

PROOF. From Theorem 1 we know that the given k-tape machine can be simulated by a two-tape machine M_2 within time $\alpha T(n) \log_2 T(n)$, for an appropriately chosen constant α. Machine M_2 can then be modified to form a new machine M_2' that in each operation performs α or more consecutive operations from the computation of M_2. Such a speed-up construction is detailed by Hartmanis and Stearns [2], and leads to a computation time

$$T'(n) \leq \max [n, T(n) \log T(n)].$$

But if $T(n) \geq n$ and $n > 1$, one has

$$T'(n) \leq T(n) \log T(n),$$

which completes the proof.

A function $U(n)$ is called *real-time countable* [4] if and only if it is monotone increasing and there exists some sequence generator whose output on the jth operation is 1 if $j = U(i)$ for some integer i, and whose output is 0 otherwise. In other words, $U(n)$ is real-time countable if its characteristic sequence is n-computable. It is now possible to state our first tape-independent result.

THEOREM 3. *If $U(n)$ is a real-time countable function and if $T(n)$ is a computable function, then*

$$\inf_{n \to \infty} \frac{T(n) \log T(n)}{U(n)} = 0$$

implies that there is a binary sequence ω that is $U(n)$-computable but not $T(n)$-computable.

PROOF. The proof is basically the same as the proof of Theorem 9 in Hartmanis and Stearns [2], except that the "Square Law" used in that theorem is now replaced by our Theorem 2. The basic idea is to generate the desired sequence ω with a sequence generator that operates within time $U(n)$ and computes enough of each $T(n)$-computable binary sequence ω' to ensure that at least one symbol of ω differs from the corresponding symbol of ω'. Only the key ideas of the construction are mentioned here.

Let M_i denote the ith two-tape, binary-output sequence generator, according to some (computable) method of ordering all such Turing machines. It is then desired to design a "diagonal" sequence generator D whose operation really comprises three simultaneous computations. One of these computations consists of simulating, one after the other, appropriate initial portions of the computations performed by $M_1, M_2, \cdots$, and so on. The second computation consists of the generation, in real time, of the characteristic sequence of $U(n)$. The third consists of the generation, not necessarily in real time, of the characteristic sequence of $T(n) \log T(n)$.

As D simulates any particular machine M_i, it keeps track of the numbers of basic

operations that M_i needs to generate its various output symbols. These numbers are compared with the computed values of $T(n) \log T(n)$ in order to check whether or not M_i has been operating within the time bound $T(n) \log T(n)$. The fact that this determination cannot in general keep up with the simulation of M_i is unimportant. The important point is that each M_i that does *not* operate within the time $T(n) \log T(n)$ can eventually be identified and eliminated from further consideration.

Since the machine D must operate within the time $U(n)$, it will be designed to produce an output symbol coincident with each of the 1's in the characteristic sequence of $U(n)$. Two situations may arise at the time at which the jth such 1 is generated. First, D may not yet have had time to determine the jth output of the machine that it is currently simulating. In this case, D will produce some arbitrary output symbol, say 0. On the other hand, D may already have determined the jth output symbol of the machine that it is currently simulating. In this case, D will produce as its own output the complement of the symbol produced by the simulated machine.

The simulation of each machine M_i is continued until one of the following situations arises:

(a) D determines that M_i does not operate within the time $T(n) \log T(n)$.

(b) D is able to determine the jth output of machine M_i at or before time $U(j)$, where $U(j)$ is measured from the very beginning of D's computation.

Note that if M_i does operate within the bound $T(n) \log T(n)$, then situation (b) must eventually arise. For suppose that the simulation of machine M_i is begun after D has executed a total of B_i basic operations, and that at most C_i operations are required to simulate one of M_i's operations. Then because

$$\inf \frac{T(n) \log T(n)}{U(n)} = 0,$$

there must be a finite value of j for which

$$B_i + C_i T(j) \log T(j) < U(j).$$

Once either of the situations (a) or (b) arises, D stops its simulation of M_i and begins to simulate the computation of M_{i+1}.

Now let ω' be any binary sequence that can be generated within time $T(n)$. By virtue of Theorem 2, ω' can be generated by some two-tape machine M within time $T(n) \log T(n)$. But in the course of its computation, D eventually simulates machine M and produces as its jth output symbol the complement of the jth symbol produced by M. Therefore, the sequence ω produced by D differs in at least one symbol from the sequence ω', and the proof is completed.

5. *Application to On-Line Turing Machines*

Like a sequence generator, an *on-line Turing machine* is a basic multitape Turing machine with output values assigned to certain of the states of its control unit. In addition, however, an on-line machine is provided with a special terminal at which input symbols can be supplied. Such a machine begins its computation in a designated starting state, with all of its tapes blank. Upon being supplied with an input symbol, it goes through a number of basic operations leading to a state to which an output symbol has been assigned. Once such a state is reached, the ma-

chine is ready for a new input symbol, at which point the process is repeated. Any finite sequence of input symbols that leads the machine to a state with output 1 is said to be *accepted* by the machine. A set R of finite sequences is said to be $T(n)$-*recognizable* (by an on-line machine) if and only if there is an on-line Turing machine M that processes any input sequence of length n within $T(n)$ operations and such that R is precisely the set of finite sequences accepted by M.

THEOREM 4. *If a set R can be recognized by a k-tape on-line Turing machine within time $T(n)$, then it can be recognized by a two-tape on-line machine within time $T(n)$ log $T(n)$.*

PROOF. The proof is substantially the same as that of Theorem 2.

THEOREM 5. *If $U(n)$ is a real-time countable function and if $T(n)$ is a computable function, then*

$$\inf_{n\to\infty} \frac{T(n)\ log\ T(n)}{U(n)} = 0$$

implies that there is a set R which is $U(n)$-recognizable but not $T(n)$-recognizable.

PROOF. Let ω be the infinite sequence of Theorem 3. Then define R to be the set of all input sequences of length n for all n such that the nth bit of ω is one. R is obviously $U(n)$ recognizable. It is easy to see that a $T(n)$-recognizer for R could easily be converted into a sequence generator for ω, contrary to Theorem 3.

6. *Application to Off-Line Turing Machines*

An *off-line Turing machine* is a basic multitape Turing machine that has binary outputs associated with some of its states. In addition, one of the machine's tapes is designated as the *input tape*. The machine begins its computation in a specified starting state, and with a finite pattern of symbols written on its input tape. This pattern is surrounded by blank squares, and is positioned so that its leftmost symbol is under the reading head. For any given input pattern, the ensuing machine operations are guaranteed to lead to a (stopping) state to which an output is assigned. The input pattern is said to be *accepted* if and only if this output value is 1. A set R of finite patterns is said to be $T(n)$-*recognizable* (by an off-line machine) if and only if there exists an off-line Turing machine that processes any input pattern of length n within $T(n)$ operations and that accepts precisely the members of R.

THEOREM 6. *If a set R can be recognized by a k-tape off-line Turing machine within time $T(n)$, then it can be recognized by a two-tape off-line machine within time $T(n)$ log $T(n)$.*

PROOF. Again the proof is substantially the same as that of Theorem 2.

THEOREM 7. *If $U(n)$ is a real-time countable function, then there is a set of finite sequences R that is $U(n)$-recognizable by an off-line machine and is not $T(n)$-recognizable for any function $T(n)$ such that*

$$\inf_{n\to\infty} \frac{T(n)\ log\ T(n)}{U(n)} = 0.$$

PROOF. We shall describe the construction of an off-line machine D that has a binary input alphabet $\{0, 1\}$ and recognizes the desired set R within time $U(n)$. This machine interprets any input pattern with which it is presented as representing the binary expansion of an integer, i, in the obvious manner. The major portion

of D's computation consists in simulating the computation that the ith two-tape off-line Turing machine would perform when presented with the given input pattern. For this purpose, two of D's tapes are set aside for use in recording the tape patterns of the simulated machines M_i.

Upon being presented with an input pattern, D first measures off the length of that pattern onto an auxiliary tape. It then copies the pattern, in suitably encoded form, onto the tape being used to record the input pattern of the simulated machine. It next examines the input pattern in detail, and prepares itself to simulate the machine M_i which that pattern describes. Finally, it proceeds to simulate the computation that M_i would perform when presented with the given input pattern.

As it is doing all of this, D simultaneously generates the characteristic sequence of $U(n)$, and matches the number of 1's in that sequence against the length marked off on the auxiliary tape. As soon as the number of 1's in the generated sequence equals the length of the given input pattern, D halts, thereby ensuring that it operates within the time $U(n)$. If at this time D has already simulated the entire computation of M_i, and determined whether M_i accepts the given input pattern, it now produces an output complementary to that of M_i. If D has not yet completed the simulation of M_i's computation (as would be the case if M_i does not operate within the time $U(n)$), D now produces some arbitrary output, say, 0.

Because D is guaranteed to halt, regardless of its input pattern, it must recognize some set of input patterns—call it R. By construction, R is $U(n)$-recognizable. It must now be shown that R is not $T(n)$-recognizable. Suppose that, to the contrary, R is $T(n)$-recognizable by some multitape Turing machine. According to Theorem 6, it must then be $T(n) \log T(n)$-recognizable by some two-tape machine M_k. If the integer k is represented in binary form within an input pattern of length n, and presented to the machine M_k, the resulting computation will require at most $T(n) \log T(n)$ operations.

Now consider the number of steps needed to simulate that same computation with machine D. Marking off the length of the input pattern requires at most n operations, and recoding the input pattern requires at most αn operations, where α is some constant. Let B_k denote the number of operations needed to prepare for the simulation proper, and let C_k denote the maximum number of operations needed to simulate one of M_k's operations. Then the total number of operations that D would need to get through all of the computation performed by M_k is

$$n + \alpha n + B_k + C_k T(n) \log T(n).$$

Now D will actually complete the simulation only if this quantity is less than or equal to $U(n)$. But because of the assumed relationship between $T(n)$ and $U(n)$, it is always possible to choose a value of n sufficiently large that

$$n + \alpha n + B_k + C_k T(n) \log T(n) < U(n).$$

An input pattern whose length equals or exceeds this critical value of n is easily obtained by adding a sufficient number of 0's at the left end of the binary representation of k. The resulting pattern is one for which D and M_k behave differently; hence D and M_k cannot recognize the same set. This contradiction establishes the fact that R is not recognizable within the time $T(n)$, and completes the proof.

The import of Theorems 3, 5 and 7 can be summarized briefly as follows. As long as the function $T(n)$ is computable, and $U(n)$ is real-time countable, and $U(n)$

grows faster than some constant times $T(n) \log T(n)$, then there is some computing operation that can be carried out within the bound $U(n)$ but not within the bound $T(n)$.

RECEIVED AUGUST, 1965; REVISED FEBRUARY, 1966

REFERENCES

1. TURING, A. M. On computable numbers, with an application to the Entscheidungsproblem. *Proc. London Math. Soc.* {2}, *42* (1936–37), 230–265; Correction, *ibid.*, *43* (1937), 544–546.
2. HARTMANIS, J., AND STEARNS, R. E. On the computational complexity of algorithms. *Trans. Amer. Math. Soc. 117* (May 1965), 285–306.
3. HENNIE, F. C. One-tape, off-line Turing machine computations. *Inform. and Contr. 8*, 6 (Dec. 1965), 553–578.
4. YAMADA, H. Real-time computation and recursive functions not real-time computable. *IRE Trans. EC-11* (1962), 753–760.