JOURNAL OF COMPUTER AND SYSTEM SCIENCES 22, 351–364 (1981)

Traub, Ed.),
uter Science,
th Princeton
;uages. Tech.
put. 684–85.
University.
Proceedings,
Proceedings,
ceedings, 1st

A Time–Space Tradeoff for Sorting on Non-Oblivious Machines*

ALLAN BORODIN

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

MICHAEL J. FISCHER

Department of Computer Science, FR35, University of Washington, Seattle, Washington 98195

DAVID G. KIRKPATRICK

Department of Computer Science, University of British Columbia, Vancouver, British Columbia V6T 1W5, Canada

NANCY A. LYNCH

School of Information and Computer Science, Georgia Institute of Technology, Atlanta, Georgia 30332

AND

MARTIN TOMPA

Department of Computer Science, FR35, University of Washington, Seattle, Washington 98195

Received April 18, 1980; Revised January 5, 1981

A model of computation is introduced which permits the analysis of both the time and space requirements of non-oblivious programs. Using this model, it is demonstrated that any algorithm for sorting n inputs which is based on comparisons of individual inputs requires time–space product proportional to n^2.

1. MOTIVATION AND CONTRAPOSITION TO PREVIOUS RESEARCH

The traditional approach to studying the complexity of a problem has been to examine the amount of some single resource (usually time or space) required to perform the computation. In an effort to better understand the complexity of certain problems, recent attention has been focused on examining the tradeoff between the

* This material is based upon work supported by the Natural Sciences and Engineering Research Council of Canada under Grants A7631 and A3583, and by the National Science Foundation under Grants MCS77-02474 and MCS77-15628.

0022-0000/81/030351-14$02.00/0

571/22/3-7

required time and space. This paper adopts the latter strategy in order to pursue the complexity of sorting.

The vast majority of time–space tradeoffs recently demonstrated have been for "straight-line" (or "oblivious") programs [1, 6, 8, 11, 12, 14, 15, 16], that is, programs in which the sequence of operations is independent of the actual values of the inputs. In this model, "time" refers to the number of operations performed, and "space" to the number of auxiliary (i.e., non-input and non-output) registers used to store intermediate results. (To distinguish this usage of space from others which follow, this will be referred to as "data space.") The problem of sorting has been considered in this context by Tompa [16], who demonstrated that any oblivious algorithm which sorts n inputs requires time–space product $\Omega(n^2)$.

Although oblivious sorting algorithms have been studied extensively (see Knuth [7]), most sorting algorithms are non-oblivious; that is, they continually test and branch based on comparisons of input values. In order to truly understand the complexity of sorting, then, a model which admits non-oblivious algorithms should be adopted. Toward this end, Munro and Paterson [10] considered non-oblivious sorting algorithms which use auxiliary registers to store selected inputs and can access other inputs only through successive passes over all the inputs. Although they count only data space (i.e., number of auxiliary registers used), the authors make it clear [10, Sect. 2] that "control space" (used, for instance, to remember which inputs to fetch into registers on a given pass) is also an issue in upper bounds. To sort n inputs within their model, they demonstrate that the product of the number of registers and the number of passes is $\theta(n)$. Since each pass requires n moves of the input head, their result might be interpreted as a lower bound of $\Omega(n^2)$ on the product of time and data space. Adopting Cobham's model [3], Tompa [17] in fact demonstrated a similar tradeoff for sorting on any general string-processing model, exploiting only the restriction of "tape input" (i.e., the input head can move at most one symbol left or right in one step).

Thus there are at least three time–space tradeoffs for sorting already known [10, 16, 17]. Each of these, however, imposes some artificial restriction on the algorithms considered (either obliviousness or tape input), and so they say more about the inadequacy of these models for sorting-type problems than they do about the inherent complexity of the problems themselves. To see this, one must simply consult the three references and observe that each of these results applies as well to the problem of merging two sorted lists of n elements, for which the standard (non-oblivious, random access input) algorithm requires only $O(n)$ time, no data space, and $O(\log n)$ control space (for pointers).

In order to be compelling, then, the model of computation used to study sorting-type problems should be not only non-oblivious, but should also permit random access input. In fact, the sole restriction which is placed on the sorting algorithms considered in this paper is that they be "conservative": inputs are viewed as indivisible elements drawn from some total order, and the only operations allowed are simple comparisons. Time will be taken as the number of comparisons, but an appropriate notion of space is not immediately apparent. This section concludes by

giving an informal motivation for the measure of space adopted, to be formalized in the next section.

An algorithm in this model might have random access read-only input registers $x_1,\ldots,x_n$, auxiliary data registers $y_1,\ldots,y_k$, random access write-only output registers $z_1,\ldots,z_m$, and control registers to record information about the state of the computation. Since the only operations allowed are comparisons, each data register can store only input values; since there is random access to the input registers themselves, the data register can be eliminated completely by using control space to remember the k indices of inputs which would have been stored in the k data registers. Of course, such control space would have to be capable of assuming n^k possible values, for which it requires $k \log_2 n$ bits; this corresponds to the intuition that each of the k indices requires $\log_2 n$ bits.

This leads directly to the measure of (control) space which will be adopted: suppose that the program enters $Q(n)$ distinct configurations over all sequences of n inputs, where a configuration reflects all aspects of the program's status except the contents of the input and output registers. Then the program will be said to have $\log_2 Q(n)$ "capacity." Capacity is certainly a lower bound on the program's space requirement, for without at least $\log_2 Q(n)$ bits of storage the program cannot distinguish which configuration it is in, and therefore which comparisons to make next.

The notion of capacity is taken from Cobham [3]; the difference is only in the context. It is important to note the non-effective nature of this concept: a space-efficient encoding of the configuration might entail time-consuming encoding and decoding procedures in practice, but such time considerations are ignored in this model. Since the emphasis of this paper is on *lower bounds*, the model's non-uniformity (a different program may be used for each n) and non-effectiveness strengthen the results.

The next section formalizes the model of computation which has been motivated here. The model is sufficiently realistic to differentiate between the complexities of merging and sorting, as it admits the $O(n)$ time, $O(\log n)$ space merging algorithm, while in Section 3 it is used to demonstrate the main technical result of the paper: any algorithm for sorting n inputs which is based on comparisons of individual inputs requires time–space product $\Omega(n^2)$.

An algorithm which demonstrates that this lower bound is nearly tight was discovered by Munro and Paterson [10]. Frederickson [4] tightened their upper bound further by exhibiting an algorithm which uses time–space product $O(n^2 \log n)$, for any amount of space which is $\Omega(\log n)$ and $O(n)$.

2. Branching Programs

The model which has been used extensively to study the time requirements of sorting-type problems is the model of "tree programs" (see Knuth [7]). Pippenger [personal communication] suggested a generalization of tree programs which could

be used to study the space requirements of these problems as well. These generalized tree programs were studied initially in [17], where they were called "branching programs." This section defines and compares tree programs and branching programs, and presents some simple properties of the latter.

A *tree program* is a directed tree with bounded outdegree d whose internal vertices are labelled by queries of the inputs, whose edges are labelled by the (at most d) possible responses to those queries, and whose leaves are labelled by vectors of outputs. Given an input vector, the query at the root is tested first, and control follows that edge emanating from the root which is labelled by the correct answer for the given input values. This leads to the next query to be tested, and processing continues similarly. Each input vector thus determines a path from the root to some leaf which is labelled with the correct output for the given input. Figure 1 shows an example of a tree program which merges two sorted list (x_1, x_2) and (y_1, y_2) satisfying $x_1 \leqslant x_2$ and $y_1 \leqslant y_2$, using queries of the form "Is $x_i \leqslant$ or $> y_j$?"

The *time required by a tree program* is the length of the longest path followed by any input. The *time required to compute a function f* is the minimum, over all tree programs P which compute f, of the time required by P.

Tree programs give no indication of the space requirements of problems and this motivates the generalization to branching programs. A *branching program* is a directed multigraph, with bounded outdegree and a distinguished vertex of indegree 0 called the *source*, whose vertices of non-zero outdegree are labelled by queries, and whose edges are labelled by the possible responses to those queries together with (possibly empty) vectors of outputs. Beginning at the source, processing proceeds exactly as in tree programs, with the addition that whenever control traverses an edge it outputs the associated output vector. Figure 2 shows an example of a branching program which merges $x_1 \leqslant x_2$ with $y_1 \leqslant y_2$ using the same set of queries as the tree program of Fig. 1. In fact, this branching program is derived from the tree program by moving outputs as high as possible in the tree and then coalescing all identical subtrees.

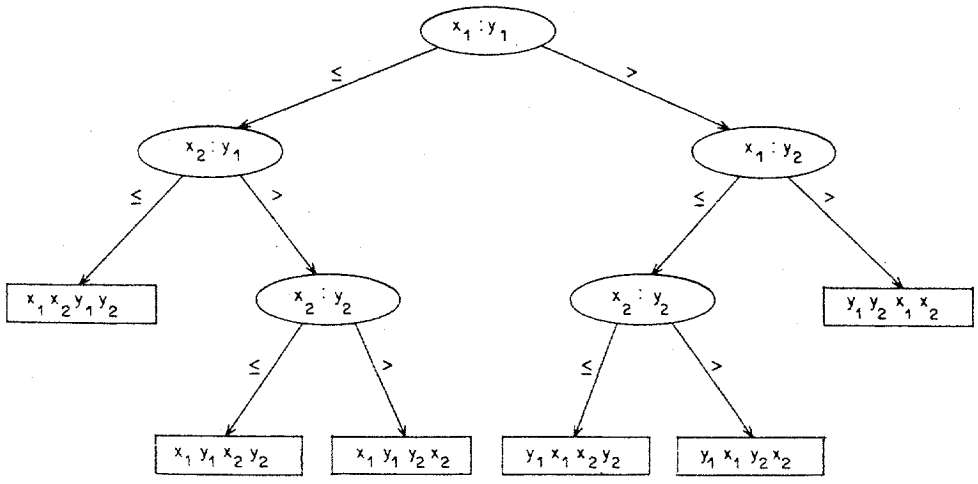

FIG. 1. Tree program for merging $x_1 \leqslant x_2$ with $y_1 \leqslant y_2$.

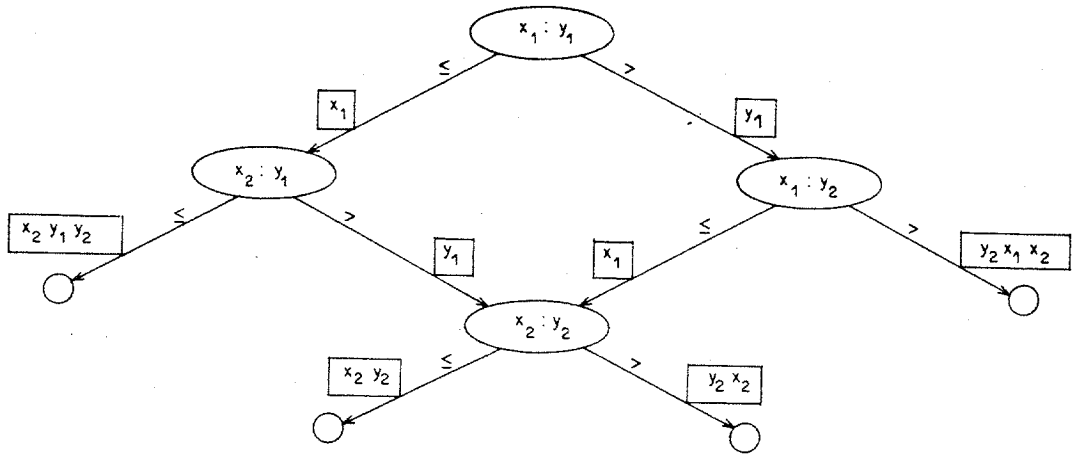

FIG. 2. Branching program for merging $x_1 \leqslant x_2$ with $y_1 \leqslant y_2$.

The *time required by a branching program* is the length of the longest path followed by any input from the source to a vertex of outdegree 0. (If some input causes control to go around a cycle, the time is undefined.) The *capacity required by a branching program* is $\log_2 |V|$, where

$$V = \{v \mid \text{some input causes control to reach vertex } v\}.$$

The *time (capacity) required to compute a function f* is the minimum, over all branching programs P which compute f, of the time (capacity) required by P. As discussed in Section 1, the capacity S required by branching programs is a lower bound to within a constant factor on the space requirement of any "reasonable" machine which solves the problem using the same set of queries, as any such machine must assume at least 2^S different configurations, for which it requires $\Omega(S)$ space.

It is reassuring to observe that branching programs share some of the same relationships of time and space as the classical models:

PROPOSITION 1. *If S and T are respectively the capacity and time required by some branching program, then $S = O(T)$ and $T \leqslant 2^S$ (provided T is not undefined).*

Proof. The first relationship follows from the observation that the number of vertices reachable from a given vertex in a graph with outdegree d and depth T is at most d^{T+1}. The second relationship follows from the observation that T is the number of vertices on a non-self-intersecting path followed by some input, whereas 2^S is the number of vertices accessed by any input. ∎

Notice that it is possible for a branching program to have cycles and yet have a finite time requirement, as long as no input can cause control to go completely around any cycle. Pippenger's original suggestion was that branching programs be acyclic; that suggestion is justified by the following normal form:

PROPOSITION 2 (Pippenger). *Let P be a branching program which uses time T*

and capacity S. Then there is a branching program P' which, using the same set of queries, computes the same function as P in time T and capacity at most $S + \log_2 T \leqslant 2S$, and has the property that its vertices can be partitioned into $T+1$ sets $V_1, V_2, \ldots, V_{T+1}$ such that any edge emanating from a vertex in V_i terminates at a vertex in V_{i+1}.

Proof. Suppose P is a branching program which uses time T and capacity S, and let $v = 2^S$ be the number of reachable vertices of P. P may have cycles, but no input can force P to take more than T steps before halting. Define an acyclic version P' of P as follows: P' has $T+1$ copies of the vertex set of P, and the edges of P' are provided by the following rule: If some input causes control in P to pass from vertex a to vertex b at time i, then there is an edge in P' from copy i of vertex a to copy $i+1$ of vertex b. P' uses time T and capacity at most $\log_2(vT) = S + \log_2 T \leqslant 2S$, by Proposition 1. ∎

This proposition aids in conceptualizing branching programs, as it shows that, for asymptotic considerations, it suffices to study those that arise from tree programs by coalescing identical subtrees.

One final elementary observation concerning branching programs demonstrates that they are indeed generalizations of tree programs:

PROPOSITION 3. *If sufficient capacity is available, the time requirements of branching programs and tree programs to compute a given function using the same set of queries are identical, in the following strong sense: there is a tree program which for each input x uses t_x steps iff there is a branching program which for each input x uses t_x steps.*

Proof. In one direction the statement is trivial, since any tree program is also a branching program. In the other direction, suppose there is a branching program which uses t_x steps on input x. By Proposition 2 there is an acyclic version which also uses t_x steps. From this version a tree program can be constructed by beginning at the source and splitting all vertices of indegree greater than 1, and finally pushing all outputs down to the leaves. ∎

Masek [9] investigated space requirements for a type of branching program, but considered queries of the form "What is the ith bit of the input?" The remainder of this paper will not deal with such general programs, but will consider instead queries based on simple comparisons of inputs. In the concluding section we return to questions concerning the more general model.

A *$\{<, >\}$-branching program* (or a *$\{<, >\}$-tree program*) is one which employs only queries of the form "$x_i : x_j$," where x_i and x_j are inputs, and receives one of the two replies "$x_i < x_j$" or "$x_i > x_j$." ($\{<, =, >\}$-programs and $\{=, \neq\}$-programs are defined analogously.) It is important to observe that since the action of a $\{<, >\}$-

program is determined solely by the ordering of the inputs, it is sufficient to examine its behavior on the $n!$ permutations of $(1, 2,..., n)$.

The next section examines the time and capacity requirements of $\{<, >\}$-branching programs for sorting.

3. A Time-Capacity Tradeoff for Sorting

Given distinct inputs $x_1, x_2,..., x_n$ drawn from some total order, the *sorting problem* is to output a sequence $r_1, i_1, r_2, i_2,..., r_n, i_n$, where

(1) $(r_1, r_2,..., r_n)$ is any permutation of $(1, 2,..., n)$, and

(2) x_{i_j} is the r_jth smallest input, for all $1 \leqslant j \leqslant n$.

(In a more conventional version of sorting, $(r_1, r_2,..., r_n)$ might be required to be the identity permutation of $(1, 2,..., n)$. For the purpose of establishing more general lower bounds, this restriction will not be assumed, nor even that the same permutation is employed for all inputs.)

This section is devoted to proving that any $\{<, >\}$-branching program which sorts n inputs in time T and capacity S requires $ST = \Omega(n^2)$. It may be helpful at this point to describe informally the mechanism by which branching programs may trade off capacity for time. A tree program (that is, a branching program which maximizes capacity) may be able to save time by the fact that at any point in the computation it "remembers" the results of all previous queries. On the other hand if the number of configurations reachable after j steps is restricted well below 2^j, the resulting "confusion" may well require extra time to be resolved. Theorem 1 shows that this is in fact the case for sorting: if, at strategic points in the computation, the number of configurations is restricted to 2^S, for any $S = o(n)$, then the ranking of the next S inputs is achievable only at the expense of $\Omega(n)$ comparisons. Since all n inputs must be ranked, the total time is $\Omega(n^2/S)$.

The proof of Theorem 1 proceeds through a sequence of four lemmas. Since Theorem 1 measures the "confusion" which arises when many permutations reach the same configuration, the lemmas deal with properties of arbitrary sets of permutations. Lemmas 1 and 2 establish some interesting combinatorial relationships among certain sets of permutations consistent with a given Hasse diagram. Lemmas 3 and 4 give lower bounds on the times required by programs which determine the ranks of many inputs.

A *Hasse diagram over* $\{i_1, i_2,..., i_n\}$ is an acyclic directed graph with n vertices which are labelled distinctly with the elements $i_1, i_2,..., i_n$. A permutation $\pi = (\pi(i_1), \pi(i_2),..., \pi(i_n))$ of $(i_1, i_2,..., i_n)$ is said to be *consistent* with a Hasse diagram H if $\pi(j) > \pi(k)$ whenever there is a directed path of positive length from j to k in H. Two elements j and k are said to be *comparable* in H if there is a directed path either from j to k or from k to j in H. If H is a Hasse diagram over $\{i_1, i_2,..., i_n\}$ define

(1) $P(H) = \{\pi \mid \pi$ is a permutation of $(i_1, i_2,..., i_n)$ consistent with $H\}$.

(2) $C(H, i) = \{j \mid j \text{ is comparable to } i \text{ in } H\}$.

(3) $H - i$ is the Hasse diagram on $n-1$ vertices which results from removing the vertex labelled i and adding an edge (j, k) for each pair of edges (j, i) and (i, k) in H.

The following lemma relates $P(H-i)$ to $P(H)$:

LEMMA 1. *Let H be a Hasse diagram on n vertices, one of which has label i. Then $n \cdot |P(H-i)| \leqslant |C(H, i)| \cdot |P(H)|$.*

Proof. A permutation π is said to be *i-consistent* with H if, for all $j \neq i$ and $k \neq i$, $\pi(j) > \pi(k)$ whenever there is a directed path of positive length from j to k in H. Since there are n places to insert i into each permutation whose inverse is in $P(H-i)$ (ignoring the constraints of H on i) the left-hand side $n \cdot |P(H-i)|$ is the number of permutations which are i-consistent with H. It remains to show that the right hand side is an upper bound on this number.

Suppose π is i-consistent with H. Consider the set of elements each of which, together with i, "disrespects" H; that is, the set $D \cup D'$, where

$$D = \{j \mid \pi(i) < \pi(j) \text{ but there is a directed path from } i \text{ to } j \text{ in } H\},$$

$$D' = \{j \mid \pi(i) > \pi(j) \text{ but there is a directed path from } j \text{ to } i \text{ in } H\}.$$

Notice that $D \cup D' \subseteq C(H, i)$. Notice also that either D or D' must be empty, since $j \in D$ and $k \in D'$ implies $\pi(j) > \pi(k)$ but there is a directed path from k to j in H, which is impossible. Suppose $D' = \phi$, the case when $D = \phi$ being dual. Let the elements of D be $j_1, j_2, \ldots, j_m$, where $\pi(j_1) < \pi(j_2) < \cdots < \pi(j_m)$. Consider the following scheme which "rotates" the values of π at $j_0(=i), j_1, j_2, \ldots, j_m$:

$$\pi'(j) = \pi(j_m), \qquad \text{if} \quad j = j_0,$$

$$\pi'(j) = \pi(j_{h-1}), \qquad \text{if} \quad j = j_h \text{ for } 1 \leqslant h \leqslant m,$$

$$\pi'(j) = \pi(j), \qquad \text{otherwise.}$$

Then $\pi' \in P(H)$, since

(a) By the definition of D, i is now in its "correct" place.

(b) The only other pairs (j, k) which have changed relative positions (i.e., $\pi(j) > \pi(k)$ but $\pi'(j) < \pi'(k)$) satisfy $j \in D$ and $k \notin C(H, i)$. In this case there can be no directed path from j to k in H, since if there were the directed path from i to j in H would place k in $C(H, i)$.

The fact that $|C(H, i)| \cdot |P(H)|$ is an upper bound on the number of i-consistent permutations follows from the observation that the pair (j_1, π') uniquely determines at most one i-consistent permutation π. ∎

COROLLARY. *Let H be a Hasse diagram on n vertices, k of which have labels $j_1, j_2, \ldots, j_k$. Then*

$$\frac{n!}{(n-k)!}\,|P(H-j_1-j_2-\cdots-j_k)| \leqslant \left(\prod_{h=1}^{k} |C(H, j_h)|\right) \cdot |P(H)|.$$

Proof. By induction on k.

Basis ($k = 1$): This is just Lemma 1.

Induction ($k > 1$):

$$\begin{aligned}
&\frac{n!}{(n-k)!}\,|P(H-j_1-j_2-\cdots-j_k)| \\
&\quad = n \cdot \frac{(n-1)!}{[(n-1)-(k-1)]!} \cdot |P((H-j_1)-j_2-\cdots-j_k)| \\
&\quad \leqslant n \left(\prod_{h=2}^{k} |C(H-j_1, j_h)|\right) \cdot |P(H-j_1)| \qquad \text{induction hypothesis} \\
&\quad \leqslant \left(\prod_{h=2}^{k} |C(H, j_h)|\right) \cdot (n \cdot |P(H-j_1)|) \\
&\quad \leqslant \left(\prod_{h=1}^{k} |C(H, j_h)|\right) \cdot |P(H)| \qquad \text{Lemma 1.} \quad \blacksquare
\end{aligned}$$

Let $P(H, r_1: j_1, r_2: j_2, \ldots, r_k: j_k) = \{\pi \mid \pi \in P(H) \text{ and } \pi(j_h) = r_h \text{ for all } 1 \leqslant h \leqslant k\}$.

LEMMA 2. *Let H be a Hasse diagram, k of whose vertices have labels $j_1, j_2, \ldots, j_k$. Then*

$$|P(H, r_1: j_1, r_2: j_2, \ldots, r_k: j_k)| \leqslant |P(H-j_1-j_2-\cdots-j_k)|.$$

Proof. This is most easily seen by examining the inverses of permutations in $P(H, r_1: j_1, \ldots, r_k: j_k)$; that is, permutations with the value j_h in the r_hth position. The removal of $j_1, j_2, \ldots, j_k$ from such permutations yields the inverse of a permutation in $P(H-j_1-j_2-\cdots-j_k)$, and such removal from distinct permutations cannot yield identical permutations. $\blacksquare$

Lemmas 1 and 2 lay the necessary foundation to prove an interesting result concerning $\{<, >\}$-tree programs which compute a function related to sorting. Given n distinct inputs $x_1, x_2, \ldots, x_n$ drawn from some total order, the *k-ranking problem* is to output a sequence $r_1, i_1, r_2, i_2, \ldots, r_k, i_k$, where

(1) $(r_1, r_2, \ldots, r_k)$ is any permutation of any k elements from $\{1, 2, \ldots, n\}$, and

(2) x_{i_j} is the r_jth smallest input, for all $1 \leqslant j \leqslant k$.

(Notice that a tree program which computes the k-ranking problem may employ a different permutation $(r_{\phi,1}, r_{\phi,2},..., r_{\phi,k})$ at each leaf ϕ.)

LEMMA 3. *Let τ be a $\{<, >\}$-tree program which outputs $2k$ integers between 1 and n at each leaf, and let $P(\tau)$ be the set of permutations of $(1, 2,..., n)$ for which τ correctly computes the k-ranking problem. Then $|P(\tau)| \leqslant (t+1)^k (n-k)!$, where t is the length of the longest branch of τ.*

Proof. Let ϕ be any leaf which is reached by some input, and suppose the sequence output at ϕ is $r_{\phi,1}, j_{\phi,1}, r_{\phi,2}, j_{\phi,2},..., r_{\phi,k}, j_{\phi,k}$. There is a natural Hasse diagram H_ϕ on n vertices associated with ϕ; namely, edge (i, j) is in H_ϕ iff on the branch from the root of τ to ϕ the response ">" to query "$x_i : x_j$" was received (or equivalently the response "<" to query "$x_j : x_i$"). Notice the following two facts about H_ϕ:

(1) $|C(H_\phi, j_{\phi,h})| \leqslant t+1$, since there are at most t comparisons made on the branch from the root of τ to ϕ, and

(2) $\{P(H_\phi) \mid \phi$ is a reachable leaf of $\tau\}$ partitions the set of $n!$ permutations of $(1, 2,..., n)$.

Then

$$
\begin{aligned}
|P(\tau)| &= \sum_{\phi \text{ reachable}} |P(H_\phi, r_{\phi,1}: j_{\phi,1},..., r_{\phi,k}: j_{\phi,k})| \\
&\leqslant \sum_{\phi \text{ reachable}} |P(H_\phi - j_{\phi,1} - j_{\phi,2} - \cdots - j_{\phi,k})| && \text{Lemma 2} \\
&\leqslant \sum_{\phi \text{ reachable}} \frac{(n-k)!}{n!} \left(\prod_{h=1}^{k} |C(H_\phi, j_{\phi,h})| \right) \cdot |P(H_\phi)| && \text{corollary of Lemma 1} \\
&\leqslant \sum_{\phi \text{ reachable}} \frac{(n-k)!}{n!} (t+1)^k \cdot |P(H_\phi)| && \text{fact (1)} \\
&= \frac{(t+1)^k (n-k)!}{n!} \cdot \sum_{\phi \text{ reachable}} |P(H_\phi)| \\
&= (t+1)^k (n-k)! && \text{fact (2).} \quad \blacksquare
\end{aligned}
$$

COROLLARY. *Let τ be any $\{<, >\}$-tree program which computes the k-ranking problem, and let P be an arbitrary set of permutations of $(1, 2,..., n)$. Then for any t, at least $|P| - (t+1)^k (n-k)!$ permutations in P each follow branches of length at least $t+1$ in τ.*

Proof. Suppose to the contrary that more than $(t+1)^k (n-k)!$ permutations follow branches which terminate after at most t comparisons. The result of cutting off all branches of τ after t comparisons is a tree program which violates Lemma 3. $\blacksquare$

We are finally in a position to prove a lower bound on the time required by

$\{<, >\}$-branching programs to sort. For the remainder of this section it will be assumed that branching programs are in the normal form described in Proposition 2 of Section 2.

LEMMA 4. *Let τ be any $\{<, >\}$-branching program which sorts n inputs, and let V_j be the total number of vertices of distance at most $j-1$ from the source of τ. Let t and k be arbitrary positive integers, and let i be any integer satisfying $0 \leqslant i \leqslant \lfloor (n-1)/(k-1) \rfloor$. Then after $(n-2)+it$ comparisons, at least $n! - V_{(n-2)+it}(t+1)^k (n-k)!$ permutations of $\{1, 2, \ldots, n\}$ have each output only $i(k-1)$ inputs together with their ranks.*

Proof. By induction on i.

Basis $(i=0)$: Every branch of any $\{<, >\}$-tree program which computes the 1-ranking problem must have length at least $n-1$, since the Hasse diagram associated with each leaf must be connected. The basis then follows from Proposition 3 of Section 2.

Induction $(i>0)$: Let P be the set of permutations each of which has output at most $(i-1)(k-1)$ ranked inputs after $(n-2)+(i-1)t$ comparisons. By the induction hypothesis, $|P| \geqslant n! - V_{(n-2)+(i-1)t}(t+1)^k (n-k)!$. Consider the set of vertices $v_1, v_2, \ldots, v_m$ at the $(n-1)+(i-1)t$th comparison along each path, and let P_j be the set of permutations in P which arrive at v_j. From the corollary to Lemma 3 and Proposition 3 of Section 2, at least $|P_j| - (t+1)^k (n-k)!$ permutations in P_j require $t+1$ comparisons starting at v_j to output the next k indices and ranks. Then the total number of permutations in P which require $t+1$ additional comparisons is at least

$$\begin{aligned}
&\sum_{j=1}^{m} (|P_j| - (t+1)^k (n-k)!) \\
&\quad = |P| - m(t+1)^k (n-k)! \\
&\quad \geqslant (n! - V_{(n-2)+(i-1)t}(t+1)^k (n-k)!) - m(t+1)^k (n-k)! \\
&\quad \geqslant n! - V_{(n-2)+it}(t+1)^k (n-k)!
\end{aligned}$$

Therefore, after only t additional comparisons ($n-2+it$ in total), at least this many permutations will have output only $(i-1)(k-1)+(k-1) = i(k-1)$ indices and ranks. ∎

THEOREM 1. *Any $\{<, >\}$-branching program which sorts n inputs in time T and capacity S requires $ST = \Omega(n^2)$.*

Proof. Let $k = S$ and $i = \lfloor (n-1)/(S-1) \rfloor$ in Lemma 4. (These choices maximize ST asymptotically.) Then t can be chosen as large as desired, subject to the constraint that at least one permutation remains; that is, $n! - V_T(t+1)^S (n-S)! > 0$. Choosing $t+1 = \lfloor (n-S)/2 \rfloor$ satisfies this, as

$$n! - V_T(t+1)^S (n-S)! \geqslant n! - 2^S \lfloor (n-S)/2 \rfloor^S (n-S)!$$
$$\geqslant n! - (n-S)^S (n-S)!$$
$$> n! - n(n-1)\cdots(n-S+1)(n-S)! = 0.$$

Then by lemma 4,

$$T \geqslant (n-1) + it = (n-1) + \lfloor (n-1)/(S-1) \rfloor (\lfloor (n-S)/2 \rfloor - 1)$$
$$\geqslant (n-1) + ((n-S)/S)((n-S-3)/2)$$

or $2ST \geqslant (n-S)(n-S-3) + 2S(n-1) = n^2 - 3n + S^2 + S$. ∎

COROLLARY. *Any $\{<, =, >\}$-branching program which sorts n (not necessarily distinct) inputs requires $ST = \Omega(n^2)$.*

Proof. Removal of the "=" branches yields a $\{<, >\}$-branching program which sorts n distinct inputs. ∎

THEOREM 2. *Any $\{<, >\}$-branching program which sorts n inputs requires $S\bar{T} = \Omega(n^2)$, where $\bar{T}$ is the average time required when all $n!$ permutations are considered equally likely.*

Proof. Choosing $t + 1 = \lfloor (n-S)/4 \rfloor$ in the proof of Theorem 1 shows that not one but $n!(2^S - 1)/2^S$ of the $n!$ permutations require at least half the stated time. ∎

4. QUESTIONS FOR FURTHER RESEARCH

This paper has presented a time–space tradeoff for sorting using a very general model which is non-oblivious, has random access input, and places no restrictions on the manner in which space is used. Its only restriction is its "conservative" nature: inputs are assumed to be indivisible entities which can only be compared. Borodin and Cook [2] have extended the techniques introduced here to prove a similar tradeoff for sorting on general (non-conservative) string processing models with random access input.

It is natural to ask whether the techniques presented can be used to establish more dramatic tradeoffs for branching programs. Although there are natural problems which appear to be candidates for such dramatic tradeoffs, results of Cook and Tompa (see [17]) show that such a tradeoff would have broader ramifications: if no $\{=, \neq\}$- or $\{<, >\}$-branching program solves a given problem in capacity S and time T, then no general string-processing machine solves the problem in space $S - \log n$ and time T/n. The work of Borodin and Cook indicates that the combinatorial focus of branching programs may well lend itself to the establishment of such general lower bounds.

There are also questions relating specifically to branching programs that do not

appear to yield such implications for general string-processing models. For example, the set of allowable queries might be extended from simple comparisons to comparisons of linear functions of the inputs. (Yao [18] has extended the time-space tradeoff for sorting proved in this paper to such a model.) The results of Cook and Tompa do not seem to apply to these generalized branching programs. However, the resulting model would be a compelling one for demonstrating time–space tradeoffs for problems such as determining shortest paths and network flows.

The method used in this paper to prove the time–space tradeoff for sorting seems inapplicable to problems with few outputs, as the main lemma used (lemma 4) states that roughly n comparisons must be made for every S outputs. However, because of certain similarities to sorting, it seems reasonable that the tradeoff $ST = \Omega(n^2)$ might hold for the following two decision problems: (1) given n inputs, determine if they are all distinct, and (2) given two sets each of size n, determine if they are disjoint. By applying Reingold's reduction from sorting to set disjointness [13], these tradeoffs would follow if the generalization of Theorem 1 to a less constructive variant of branching programs holds: instead of actually producing outputs at specific points along each path, the branching program need only satisfy the condition that the path followed by any input receives responses sufficient to determine the sorted order.

Finally, there are problems which have been shown to possess surprisingly good algorithms with respect to simultaneous time and space requirements. These include string-matching [5] and computing the median [10]. It would be satisfying to establish corresponding lower bounds for $\{=, \neq\}$- and $\{<, =, >\}$-branching programs, respectively. For example, is it true that any linear time (or perhaps "real time") $\{=, \neq\}$-branching program for string-matching requires $\omega(\log n)$ capacity (in the terms of [5], more than a fixed number of registers)?

REFERENCES

1. H. ABELSON, A note on time–space tradeoffs for computing continuous functions, *Inform. Process. Lett.* **8** (1979), 215–217.
2. A. BORODIN AND S. A. COOK, A time–space tradeoff for sorting on a general sequential model of computation, *in* "Proceedings, Twelfth Annual ACM Symposium on Theory of Computing," Los Angeles, CA, April 1980, pp. 294–301.
3. A. COBHAM, "The Recognition Problem for the Set of Perfect Squares," Research Paper RC-1704, IBM Watson Research Center, Yorktown Heights, N.Y., April 1966.
4. G. N. FREDERICKSON, "Upper Bounds for Time–Space Trade-Offs in Sorting and Selection," Technical Report CS-80-3, The Pennsylvania State University, University Park, PA, January 1980.
5. Z. GALIL AND J. SEIFERAS, Saving space in fast string-matching, *in* "Proceedings, 18th Annual Symposium on Foundations of Computer Science," Providence, R. I., Oct.–Nov. 1977, pp. 179–188.
6. D. YU. GRIGORYEV, An application of separability and independence notions for proving lower bounds of circuit complexity, Notes of Scientific Seminars No. 60, pp. 38–48, Steklov Mathematical Institute, Leningrad Department, 1976 (in Russian).
7. D. E. KNUTH, "The Art of Computer Programming," Vol. 3, "Sorting and Searching," Addison–Wesley, Reading, Mass., 1973.
8. T. LENGAUER AND R. E. TARJAN, Upper and lower bounds on time–space tradeoffs, *in* "Proceedings, Eleventh Annual ACM Symposium on Theory of Computing," Atlanta, GA, April–May 1979, pp. 262–277.

9. W. J. MASEK, "A Fast Algorithm for the String Editing Problem and Decision Graph Complexity," M. Sc. Thesis, M.I.T., May 1976.
10. J. I. MUNRO AND M. S. PATERSON, Selection and sorting with limited storage, *in* "Proceedings, 19th Annual Symposium on Foundations of Computer Science," Ann Arbor, MI, Oct. 1978, pp. 253–258.
11. W. J. PAUL AND R. E. TARJAN, Time–space trade-offs in a pebble game, *Acta Inform.* **10** (1978), 111–115.
12. N. PIPPENGER, A time–space trade-off, *J. Assoc. Comput. Mach.* **25** (1978), 509–515.
13. E. M. REINGOLD, On the optimality of some set algorithms, *J. Assoc. Comput. Mach.* **19** (1972), 649–659.
14. R. REISCHUK, Improved bounds on the problem of time–space trade-off in the pebble game, *in* "Proceedings, 19th Annual Symposium on Foundations of Computer Science," Ann Arbor, MI, Oct. 1978, pp. 84–91.
15. J. E. SAVAGE AND S. SWAMY, Space–time tradeoffs on the FFT algorithm, *IEEE Trans. Inform. Theory* **24** (1978), 563–568.
16. M. TOMPA, Time–space tradeoffs for computing functions, using connectivity properties of their circuits, *in* "Proceedings, Tenth Annual ACM Symposium on Theory of Computing," San Diego, CA, May 1978, pp. 196–204.
17. M. TOMPA, "Time–Space Tradeoffs for Straight-Line and Branching Programs," Ph. D. Thesis, Technical Report 122/78, University of Toronto, July 1978.
18. A. C. YAO, "On the Time–Space Tradeoff for Sorting with Linear Queries," Stanford University.