A TIME-SPACE TRADEOFF FOR SORTING ON NON-OBLIVIOUS MACHINES*

Allan Borodin[1]
Michael J. Fischer[2]
David G. Kirkpatrick[3]
Nancy A. Lynch[4]
Martin Tompa[2]

ABSTRACT

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. Uniform and non-uniform sorting algorithms are presented which show that this lower bound is nearly tight.

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 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, 5, 7, 10, 11, 13, 14, 15], 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[15], 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[6], where they are called "sorting networks"), 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[9] 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 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[16] in fact demonstrated a similar trade-off 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[9, 15, 16]. 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 confer the three references and observe that each of these results applies as well to the problems 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

[1]Dept. of Computer Science, Univ. of Toronto, Toronto, Ontario, Canada M5S 1A7
[2]Dept. of Computer Science, FR-35, Univ. of Washington, Seattle, Wa. 98195 U.S.A.
[3]Dept. of Computer Science, Univ. of British Columbia, Vancouver, B.C., Canada V6T 1W5
[4]School of Information and Computer Science, Georgia Inst. of Technology, Atlanta, Ga.30332 USA

*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 MCS 77-02474 and MCS77-15628.

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 registers 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 use $\log_2 Q(n)$ space. This space might be viewed as an elaborate "program counter", for without at least $\log_2 Q(n)$ bits the program cannot distinguish which configuration it is in, and therefore which comparisons to make next.

Control space as defined is precisely Cobham's notion of "capacity"[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 (an illustration is provided by theorem 3), but such time considerations are ignored in this model. Since the main 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 serve to 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 $\Omega(n^2)$. Section 4 presents algorithms (and less effective upper bounds) for sorting which demonstrate that the established tradeoff is nearly tight. The last section discusses some directions for further research.

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[6]). 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[16], 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 lists (x_1,x_2) and (y_1,y_2) satisfying $x_1 \le x_2$ and $y_1 \le y_2$, using queries of the form "Is $x_i \le$ 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 $x_1 \le x_2$ with $y_1 \le y_2$ using the same set of queries as the tree program of figure 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.

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 $\log_c(2^S) = \Omega(S)$ 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 $T\le 2^S$ (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 out-degree 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 $S+\log_2 T\le 2S$, 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 $\log_2(vT) = S + \log_2 T \le 2S$, 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[8] 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 $(1,2,\ldots,n)$.

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

3. A Time-Capacity Tradeoff for Sorting

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

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

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

(In a more conventional version of sorting, $(r_1,r_2,\ldots,r_n)$ might be required to be the identity permutation of $(1,2,\ldots,n)$. 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 $ST=\Omega(n^2)$. The proof proceeds through a sequence of 4 lemmas. 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 $\{<,>\}$-tree programs and $\{<,>\}$-branching programs, respectively, which compute functions related to sorting.

A Hasse diagram over $\{i_1,i_2,\ldots,i_n\}$ is an acyclic directed graph with n vertices which are labelled distinctly with the elements $i_1,i_2,\ldots,i_n$. A permutation $\pi=(\pi(i_1),\pi(i_2),\ldots,\pi(i_n))$ of $(i_1,i_2,\ldots,i_n)$ 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 $\{i_1,i_2,\ldots,i_n\}$ define

(1) $P(H)=\{\pi|\pi$ is a permutation of $(i_1,i_2,\ldots,i_n)$ consistent with $H\}$.
(2) $C(H,i)=\{j|j$ is comparable to i 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

$$n\cdot|P(H-i)|\le|C(H,i)|\cdot|P(H)|.$$

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

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

$D'=\{j|\pi(i)>\pi(j)$ but there is a directed path from j to i 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 $\pi(j_1)<\pi(j_2)<\ldots<\pi(j_m)$. Consider the following scheme which "rotates" the values of π at $j_0(=i),j_1,j_2,\ldots,j_m$:

$$\pi'(j)=\begin{cases}\pi(j_m) & \text{, if } j=j_0\\ \pi(j_{h-1}) & \text{, if } j=j_h \text{ for } 1\le h\le m\ .\\ \pi(j) & \text{, otherwise}\end{cases}$$

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).

Thus, every i-consistent permutation π arises from some $\pi'\in P(H)$ by choosing some $j_1\in C(H,i)$ and "reverse-rotating" those $j_h\in C(H,i)$ which have values $\pi'(j_h)$ between $\pi'(i)$ and $\pi'(j_1)$. 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 this reverse-rotation process is well-defined. □

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

Then $\frac{n!}{(n-k)!}|P(H-j_1-j_2-\ldots-j_k)|\le(\prod_{h=1}^{k}|C(H,j_h)|)\cdot|P(H)|$.

Proof: by induction on k.

Basis (k=1): This is just lemma 1.

Induction (k>1):

$$\begin{aligned}&\frac{n!}{(n-k)!}|P(H-j_1-j_2-\ldots-j_k)|\\ &= n\cdot\frac{(n-1)!}{[(n-1)-(k-1)]!}\cdot|P((H-j_1)-j_2-\ldots-j_k)|\\ &\le n(\prod_{h=2}^{k}|C(H-j_1,j_h)|)\cdot|P(H-j_1)| && \text{induction hypothesis}\\ &\le(\prod_{h=2}^{k}|C(H,j_h)|)\cdot(n\cdot|P(H-j_1)|)\\ &\le(\prod_{h=1}^{k}|C(H,j_h)|)\cdot|P(H)| && \text{lemma 1}\end{aligned}$$

□

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

Lemma 2: Let H be a Hasse diagram, k of whose vertices have labels $j_1, j_2, \ldots, j_k$. Then $|P(H, r_1:j_1, r_2:j_2, \ldots, r_k:j_k)| \le |P(H-j_1-j_2-\ldots-j_k)|$.

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 $P(H-j_1-j_2-\ldots-j_k)$, and such removal from distinct permutations cannot yield identical permutations. □

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 $1 \le j \le k$.

(Notice that a tree program which computes the k-ranking problem may employ a different permutation $(r_{\phi,1}, r_{\phi,2}, \ldots, r_{\phi,k})$ 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 $(1,2,\ldots,n)$ for which τ correctly computes the k-ranking problem. Then $|P(\tau)| \le (t+1)^k (n-k)!$, 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 $r_{\phi,1}, j_{\phi,1}, r_{\phi,2}, j_{\phi,2}, \ldots, r_{\phi,k}, j_{\phi,k}$. There is a natural Hasse diagram H_ϕ 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) $|C(H_\phi, j_{\phi,h})| \le t+1$, since there are at most t comparisons made on the branch from the root of τ to ϕ, and

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

Then

$$|P(\tau)| = \sum_{\phi \text{reachable}} |P(H_\phi, r_{\phi,1}:j_{\phi,1}, \ldots, r_{\phi,k}:j_{\phi,k})|$$

$$\le \sum_{\phi \text{reachable}} |P(H_\phi - j_{\phi,1} - j_{\phi,2} - \ldots - j_{\phi,k})| \quad \text{lemma 2}$$

$$\le \sum_{\phi \text{reachable}} \left[\frac{(n-k)!}{n!} \left(\prod_{h=1}^{k} |C(H_\phi, j_{\phi,h})| \right) \cdot |P(H_\phi)| \right] \quad \text{corollary of lemma 1}$$

$$\le \sum_{\phi \text{reachable}} \left[\frac{(n-k)!}{n!} (t+1)^k \cdot |P(H_\phi)| \right] \quad \text{fact (1)}$$

$$= \frac{(t+1)^k (n-k)!}{n!} \cdot \sum_{\phi \text{reachable}} |P(H_\phi)|$$

$$= (t+1)^k (n-k)! \quad \text{fact (2)} \; □$$

Corollary: Let τ be any $\{<,>\}$-tree program which computes the k-ranking problem, and let P be an arbitrary set of permutations of $(1,2,\ldots,n)$. 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. □

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 $0 \le i \le \lfloor (n-1)/(k-1) \rfloor$. 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,

$$|P| \ge n! - V_{(n-2)+(i-1)t}(t+1)^k (n-k)!.$$

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

$$\sum_{j=1}^{m}\left[|P_j|-(t+1)^k(n-k)!\right] = |P| - m(t+1)^k(n-k)!$$
$$\geq\left[n! - V_{(n-2)+(i-1)t}(t+1)^k(n-k)!\right] - m(t+1)^k(n-k)!$$
$$\geq n! - V_{(n-2)+it}(t+1)^k(n-k)!$$

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. Then t can be chosen as large as desired, subject to the constraint that at least 1 permutation remains; that is, $n!-V_T(t+1)^S(n-S)!>0$. Choosing $t+1=\lfloor(n-S)/2\rfloor$ satisfies this, as

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

Then by lemma 4,

$$T\geq(n-1)+it$$
$$=(n-1)+\lfloor(n-1)/(S-1)\rfloor\,(\lfloor(n-S)/2\rfloor-1)$$
$$\geq(n-1)+\lceil(n-S)/S\rceil\,\lceil(n-S-3)/2\rceil$$

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

Corollary: Any {<,>}-branching program which sorts n inputs requires $S\overline{T}=\Omega(n^2)$, where $\overline{T}$ 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 1 but $n!(2^S-1)/2^S$ of the n! permutations require at least half the stated time. □

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. □

4. Upper Bounds on the Time to Sort in Limited Space

This section turns to upper bounds for sorting, in order to demonstrate that the tradeoff $ST=\Omega(n^2)$ is nearly optimal.

Theorem 2 (Munro and Paterson[9]): There is an algorithm which, given k and $x_1,x_2,\ldots,x_n$, where $1\leq k\leq n$, outputs $x_1,x_2,\ldots,x_n$ in sorted order in simultaneous space $S=O(k\log n)$ and time $T=O(n^2(1+\log k)/k)$.

Proof: The algorithm proceeds by making successive passes over the inputs, each time outputting the next k smallest inputs. To this end it maintains approximately 3k registers which are used to store the indices of inputs. Register R_0 contains the index of the largest input which has been output, and $R_1,R_2,\ldots,R_k$ contain indices of candidates for the next k smallest, sorted in accordance with $x_{R_0}<x_{R_1}<\ldots<x_{R_k}$. During each pass the inputs are considered in blocks of size k, and for each such block,

1. the elements are sorted into registers $B_1,B_2,\ldots,B_k$ using any $O(k\log k)$ sorting algorithm,
2. those B_i which satisfy $x_{B_i}<x_{R_0}$ are disregarded, as they have already been output, and
3. the remaining elements are merged with $R_1,R_2,\ldots,R_k$ (possibly using an extra k registers), disregarding all but the k smallest which are returned to $R_1,R_2,\ldots,R_k$.

At the end of each pass, the contents of $R_1,R_2,\ldots,R_k$ are output, and R_0 is reassigned the value in R_k. The space used is $S=O(k\log n)$, and the time $T=O((n/k)(n/k)(k\log k+k))=O(n^2(1+\log k)/k)$. □

This algorithm reveals the upper bound $ST=O(n^2\log n(1+\log k))=O(n^2\log n(1+\log S-\log\log n))$ $=O(n^2\log^2 n)$, which is not far from the lower bound given in section 3. At the extreme k=1, S=O(logn) and the upper bound is in fact $ST=O(n^2\log n)$. The next result demonstrates that this tighter upper bound on the product of time and space can also be achieved at the extreme $S=\Theta(n)$. This is done by exhibiting branching programs which sort in simultaneous time O(nlogn) and capacity O(n). Because of the non-effective nature of branching programs, this result does not seem to yield an implementable algorithm running in time O(nlogn) and space O(n). However, it does show that the branching model cannot be used to disprove the existence of such algorithms.

Theorem 3 (Tompa[16]): There is a $\{<,=,>\}$-branching program which sorts n inputs in simultaneous time $O(n\log n)$ and capacity $O(n)$.

Proof: Given inputs $x_1, x_2, \ldots, x_n$, the program follows those steps:

1. Find the median m in time and capacity $O(n)$, using the result of [2]. (Recall from proposition 1 of section 2 that any branching program using linear time uses at most linear capacity.)
2. For each i from 1 to n, set

$$\text{SPLIT}(i) = \begin{cases} -1 , & \text{if } x_i<m \\ 0 , & \text{if } x_i=m \\ +1 , & \text{if } x_i>m \end{cases}$$

(Of course, the branching program does not explicitly "compute" such tables: this information is implicit in the vertex where control resides. The SPLIT table is simply a convenient method of specifying a particular vertex).

3. Recursively sort and output the subset satisfying SPLIT(i)=-1.
4. Output m for each i satisfying SPLIT(i)=0.
5. Recursively sort and output the subset satisfying SPLIT(i)=+1, "reusing" the capacity of step 3.

This program uses time $T(n)\le 2T(n/2)+O(n)=O(n\log n)$ and capacity $S(n)\le S(n/2)+O(n)=O(n)$. □

Theorem 3 does not seem to yield an implementable algorithm which uses $O(n\log n)$ time and $O(n)$ space, since it is not clear how to decide in constant time which comparison should be made next, even though the branching program has all the information it needs in the recursive SPLIT tables to "know" which comparison to make.

Programs with the same upper bounds as theorems 2 and 3 are given in [16] for the problems of intersecting two sets, determining if two sets are equal, and computing the frequency of occurrence of each input. In that reference, it is in fact shown that, for any $1\le k\le n$, the intersection of two sets can be computed in simultaneous time $O(n^2(1+\log k)/k)$ and space $O(k\log k+\log n)$. Thus, unlike the upper bound for sorting given in theorem 2, set intersection can be computed in $O(\log n)$ space and $O((n\log\log n)^2/\log n)=o(n^2)$ time, by choosing $k=\log n/\log\log n$.

5. 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. Similar results are not known for general string-processing models, except for the results of Cobham[3] which rely strongly on the restriction of tape input with a single read head.

It is natural to ask whether the techniques presented can be extended directly to string-processing models with random access input (the "decision graphs" of Masek[8]), and whether they can be extended to establish more dramatic results for branching programs. Although there are natural problems which appear to be candidates for such dramatic results, observations by Cook and Tompa (see [16]) show why proving $\omega(n^2\log n)$ time-space lower bounds or $\omega(\log n)$ space lower bounds for $\{=,\neq\}$- or $\{<,=,>\}$-branching programs would yield $\omega(n\log n)$ time-space lower bounds or $\omega(\log n)$ space lower bounds, respectively, for general string-processing models with random access input. Is it possible that the combinatorial focus of branching programs will lend itself to the insights required to establish such general lower bounds? To appreciate this challenge notice that the only techniques to date for establishing $\omega(n\log n)$ time-space lower bounds for Turing machines with two read heads on a single input tape have been diagonalization arguments.

There are also a number of 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. The observations of Cook and Tompa do not seem to apply to such 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 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[12], 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[4] and computing the median[9]. 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 [4], more than a fixed number of registers)?

References

1. Abelson, H. A Note on Time-Space Tradeoffs for Computing Continuous Functions. *Information Processing Letters*, *8* (April 1979), 215-217.

2. Blum, M., Floyd, R.W., Pratt, V., Rivest, R.L., and Tarjan, R.E. Time Bounds for Selection. *Journal of Computer and System Sciences*, *7* (August 1973), 448-461.

3. Cobham, A. The Recognition Problem for the Set of Perfect Squares. Research Paper RC-1704, IBM Watson Research Center, Yorktown Heights, NY, April 1966.

4. Galil, Z. and Seiferas, J. Saving Space in Fast String-Matching. 18th Annual Symposium on Foundations of Computer Science, Providence, RI, Oct.-Nov. 1977, 179-188.

5. Grigoryev, D. Yu. An Application of Separability and Independence Notions for Proving Lower Bounds of Circuit Complexity. *Notes of Scientific Seminars*, *60* (1976), Steklov Mathematical Institute, Leningrad Department, 38-48 (in Russian).

6. Knuth, D.E. *The Art of Computer Programming*, *3*. Addison-Wesley, Reading, MA, 1973.

7. Lengauer, T. and Tarjan, R.E. Upper and Lower Bounds on Time-Space Tradeoffs. Proceedings of the Eleventh Annual ACM Symposium on Theory of Computing, Atlanta, GA, April-May 1979, 262-277.

8. Masek, W.J. A Fast Algorithm for the String Editing Problem and Decision Graph Complexity. M. Sc. Thesis, M.I.T., May 1976.

9. Munro, J.I. and Paterson, M.S. Selection and Sorting with Limited Storage. 19th Annual Symposium on Foundations of Computer Science, Ann Arbor, MI, Oct. 1978, 253-258.

10. Paul, W.J. and Tarjan, R.E. Time-Space Trade-Offs in a Pebble Game. Automata, Languages and Programming. *Lecture Notes in Computer Science*, *52*. Springer-Verlag, Berlin, 1977.

11. Pippenger, N. A Time-Space Trade-Off. *Journal of the Association for Computing Machinery*, *25* (July 1978), 509-515.

12. Reingold, E.M. On the Optimality of Some Set Algorithms. *Journal of the Association for Computing Machinery*, *19* (Oct. 1972), 649-659.

13. Reischuk, R. Improved Bounds on the Problem of Time-Space Trade-Off in the Pebble Game. 19th Annual Symposium on Foundations of Computer Science, Ann Arbor, MI, Oct. 1978, 84-91.

14. Savage, J.E. and Swamy, S. Space-Time Tradeoffs on the FFT Algorithm. *IEEE Transactions on Information Theory*, *24* (Sept. 1978), 563-568.

15. Tompa, M. Time-Space Tradeoffs for Computing Functions, Using Connectivity Properties of their Circuits. Proceedings of the Tenth Annual ACM Symposium on Theory of Computing, San Diego, CA, May 1978, 196-204.

16. Tompa, M. Time-Space Tradeoffs for Straight-Line and Branching Programs. Ph.D. Thesis, Technical Report 122/78, University of Toronto, July 1978.

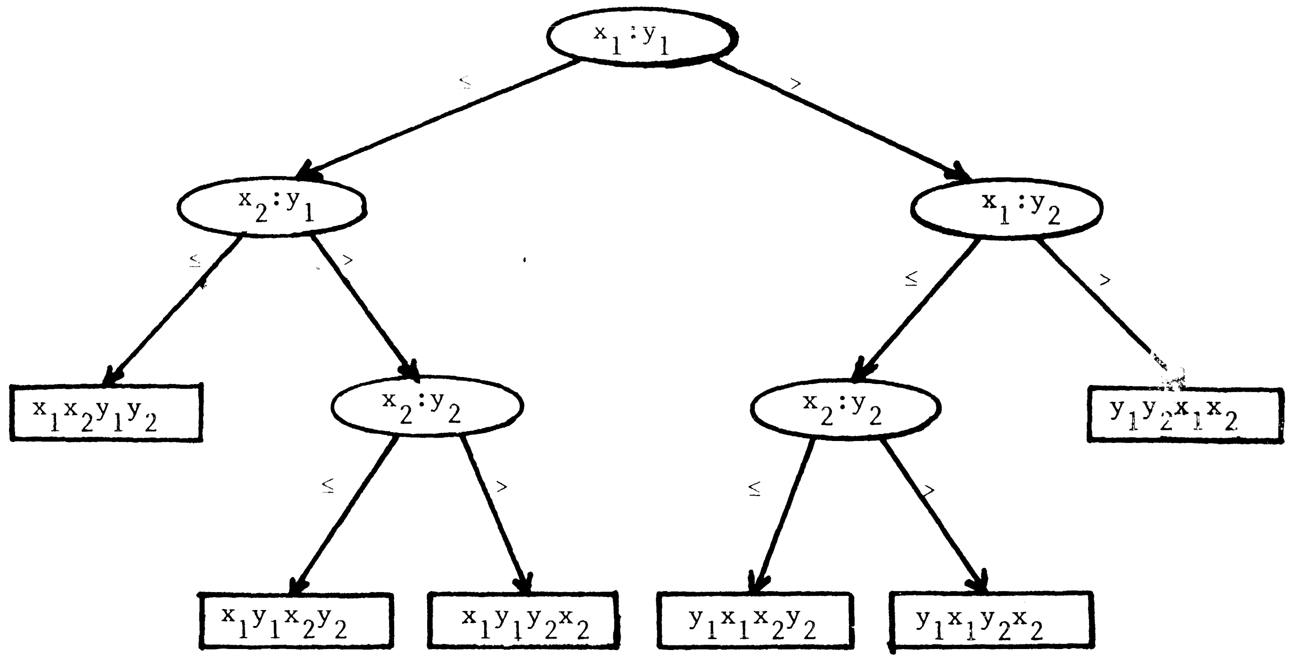

Figure 1: Tree program for merging $x_1 \leq x_2$ with $y_1 \leq y_2$.

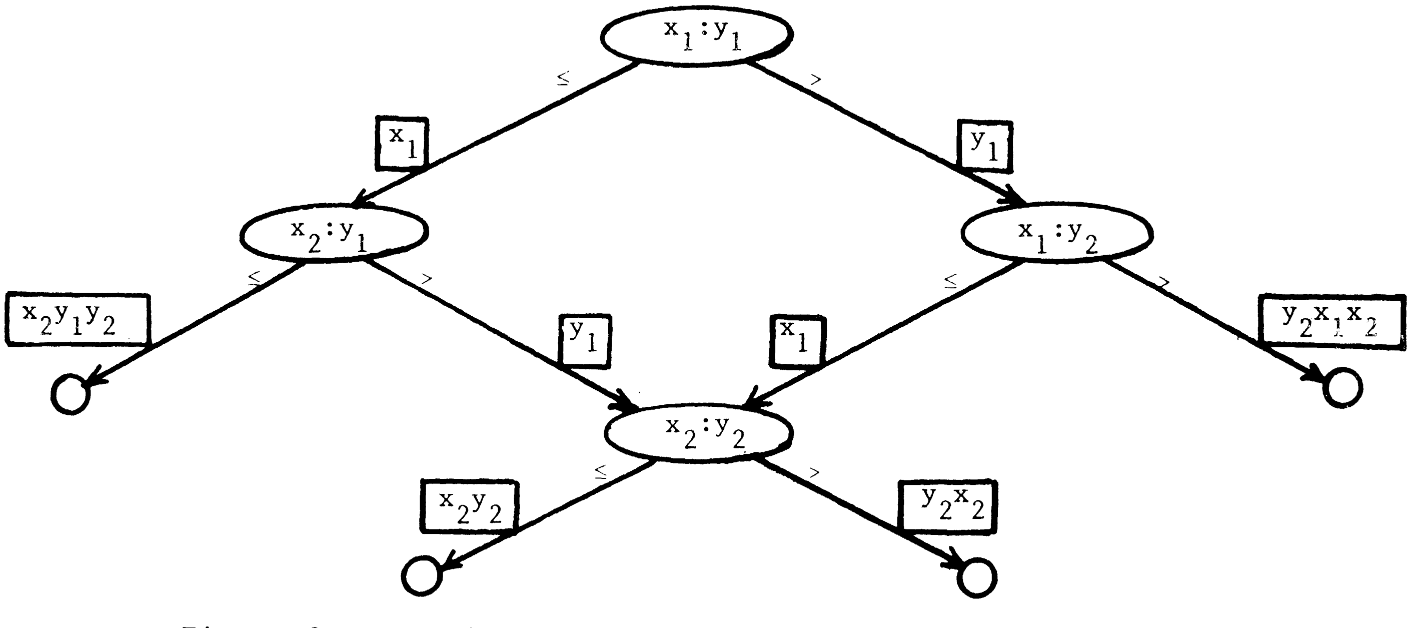

Figure 2: Branching program for merging $x_1 \leq x_2$ with $y_1 \leq y_2$.