CONF-911257--1

SAND--91-2039C

DE92 000817

Improved Selection in Totally Monotone Arrays

Yishay Mansour [*] James K. Park [†] Baruch Schieber [‡]
Sandeep Sen [§]

Abstract

This paper's main result is an $O((\sqrt{m}\lg m)(n\lg n)+m\lg n)$-time algorithm for computing the kth smallest entry in each row of an $m\times n$ totally monotone array. (A two-dimensional array $A=\{a[i,j]\}$ is *totally monotone* if for all $i_1<i_2$ and $j_1<j_2$, $a[i_1,j_1]<a[i_1,j_2]$ implies $a[i_2,j_1]<a[i_2,j_2]$.) For large values of k (in particular, for $k=\lceil n/2\rceil$), this algorithm is significantly faster than the $O(k(m+n))$-time algorithm for the same problem due to Kravets and Park (1991). An immediate consequence of this result is an $O(n^{3/2}\lg^2 n)$-time algorithm for computing the kth nearest neighbor of each vertex of a convex n-gon. In addition to the main result, we also give an $O(n\lg m)$-time algorithm for computing an approximate median in each row of an $m\times n$ totally monotone array; this approximate median is an entry whose rank in its row lies between $\lfloor n/4\rfloor$ and $\lceil 3n/4\rceil-1$.

1 Introduction

An $m\times n$ array $A=\{a[i,j]\}$ is *totally monotone* if for all i_1, i_2, j_1, and j_2 satisfying $1\le i_1<i_2\le m$ and $1\le j_1<j_2\le n$, $a[i_1,j_1]<a[i_1,j_2]$ implies

[*] Aiken Computation Laboratory, Harvard University, 33 Oxford Street, Cambridge, MA 02138, U. S. A. This work was done while the author was visiting IBM's T. J. Watson Research Center during the summer of 1989.

[†] Sandia National Laboratories, Division 1423, P. O. Box 5800, Albuquerque, NM 87185-5800, U. S. A. This work was done while the author was a member of MIT's Laboratory for Computer Science and was supported in part by the Defense Advanced Research Projects Agency under Contract N00014-87-K-0825 and the Office of Naval Research under Contract N00014-86-K-0593.

[‡] IBM Research Division, T. J. Watson Research Center, P. O. Box 218, Yorktown Heights, NY 10598, U. S. A.

[§] AT&T Bell Laboratories, 600 Mountain Avenue, Murray Hill, NJ 07974, U. S. A. This work was done while the author was visiting IBM's T. J. Watson Research Center during the summer of 1989.

$a[i_2, j_1] < a[i_2, j_2]$. Equivalently, A is totally monotone if for all j_1 and j_2 satisfying $1 \leq j_1 < j_2 \leq n$, there exists a unique I in the range $0 \leq I \leq m$ such that $a[i, j_1] \geq a[i, j_2]$ for all i satisfying $1 \leq i \leq I$ and $a[i, j_1] < a[i, j_2]$ for all i satisfying $I < i \leq m$. Figure 1 depicts an array with this property.

Totally monotone arrays were introduced by Aggarwal, Klawe, Moran, Shor, and Wilber [AKM+87]. They showed that several problems in computational geometry and VLSI river routing could be reduced to the problem of computing a maximum entry in each row of a totally monotone array. Furthermore, they gave an optimal sequential algorithm for computing the leftmost maximum in each row of an $m \times n$ totally monotone array A in $O(n)$ time when $m \leq n$ and in $O(n(1 + \lg(m/n)))$ time when $m > n$, provided any entry of A can be looked up in constant time. We will refer to this algorithm as the SMAWK algorithm, following the convention of [Wil88].

Since the publication of Aggarwal, Klawe, Moran, Shor, and Wilber's seminal paper, many additional applications of totally monotone arrays have been discovered [Wil88, Epp90, GP90, Kla89, LS91, KK90, AP89b, AP89a, AP91, AALM90, AKL+89, AKPS90, Ata90, AK91, LP91]. Almost all of these applications involve computing maximal or minimal entries in totally monotone arrays. The exception is the work of Kravets and Park [KP91]. They considered several selection and sorting problems involving the entries of totally monotone array. Among their results was an $O(k(m + n))$-time algorithm for what they called the *row-selection* problem: given an $m \times n$ totally monotone array A and an integer k in the range $1 \leq k \leq n$, find the kth smallest entry in each row of A. (For small k, this is significantly faster than the naive $O(mn)$-time algorithm obtained by ignoring the structure of A and applying the linear-time selection algorithm of Blum, Floyd, Pratt, Rivest, and Tarjan [BFP+73] to each row of A.) Kravets and Park also gave several applications of their selection and sorting algorithms. Of relevance to this paper is their linear-time reduction of the following problem from computational geometry to the totally-monotone-array row-selection problem: given a convex polygon P with vertices $p_1, p_2, \ldots, p_n$ in clockwise order, for each vertex p_i, find the vertex p_j whose Euclidean distance from p_i is kth smallest among the vertices of P. This reduction, combined with Kravets and Park's row-selection algorithm, gave an $O(kn)$-time algorithm for the aforementioned geometric selection problem.

In this paper, we propose an $O((\sqrt{m} \lg m)(n \lg n) + m \lg n)$-time algorithm for the row-selection problem considered by Kravets and Park. For large values of k (in particular, for $k = \lceil n/2 \rceil$), this algorithm represents a significant improvement over the row-selection algorithm of Kravets and

DISCLAIMER

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

$$
\begin{array}{c cc|c|c|c|cc}
 & & & j_1 & & j_2 & & \\
\cline{4-4}\cline{6-6}
 & 49 & 38 & 23 & 28 & 13 & 17 & 10 \\
 & 45 & 35 & 23 & 29 & 16 & 22 & 17 \\
 & 40 & 33 & 24 & 34 & 22 & 28 & 24 \\
I & 21 & 15 & 7 & 17 & 6 & 13 & 11 \\
 & 32 & 28 & 23 & 37 & 32 & 44 & 45 \\
 & 10 & 7 & 6 & 21 & 19 & 33 & 36 \\
 & 31 & 30 & 34 & 53 & 51 & 66 & 73 \\
 & 6 & 9 & 13 & 32 & 32 & 52 & 62 \\
 & 8 & 15 & 21 & 43 & 45 & 66 & 77 \\
\cline{4-4}\cline{6-6}
\end{array}
$$

Figure 1: Consider the 9×7 array $A = \{a[i,j]\}$ shown above, which is drawn so that the entry $a[1,1] = 49$ appears in the upper left corner. This array is totally monotone, since for any two columns $j_1 < j_2$, there exists an I in the range $0 \leq I \leq 9$ such that $a[i,j_1] \geq a[i,j_2]$ for $i \leq I$ and $a[i,j_1] < a[i,j_2]$ for $i > I$. For example, if $j_1 = 3$ and $j_2 = 5$, then $I = 4$ in the above array.

Park. An immediate consequence of our result is an $O(n^{3/2} \lg^2 n)$-time algorithm for computing the kth nearest neighbor of each vertex of a convex n-gon.

We also give an $O(n \lg m)$-time algorithm for approximating the median entry in each row of an $m \times n$ totally monotone array. The approximate median we find for each row is an entry whose rank in the row lies between $\lfloor n/4 \rfloor$ and $\lceil 3n/4 \rceil - 1$. We remark that finding these approximate medians takes significantly less time than finding exact medians with our $O((\sqrt{m} \lg m)(n \lg n) + m \lg n)$-time row-selection algorithm.

We conclude the introduction by mentioning a related class of arrays called *Monge* arrays. An $m \times n$ array $A = \{a[i,j]\}$ is *Monge* if for all i_1, i_2, j_1, and j_2 satisfying $1 \leq i_1 < i_2 \leq m$ and $1 \leq j_1 < j_2 \leq n$,

$$a[i_1,j_1] + a[i_2,j_2] \;\leq\; a[i_1,j_2] + a[i_2,j_1] \;.$$

The name of the French mathematician Gaspard Monge (1746–1818) is associated with such arrays because of work done by Hoffman [Hof63] on easily-solved special cases of the transportation problem. Hoffman showed that if the cost array associated with a transportation problem is an $m \times n$ Monge array, then a simple greedy algorithm solves the transportation problem in $O(m+n)$ time. He applied Monge's name to such arrays because, as Hoffman remarked, "the essential idea behind [the crucial observation exploited in Hoffman's paper] was first noticed by Monge in 1781!" It is easy to see that a Monge array A can be converted into a totally monotone array (but not vice-versa) by either negating all its entries or reversing the ordering of its columns. Thus, the algorithms given in [AKM+87], [KP91], and this paper are easily adapted to Monge arrays.

The remainder of this paper is organized as follows. In Section 2, we present the two basic properties of totally monotone arrays that we require for our row-selection algorithm. Section 3 gives a simple rank-computing algorithm that our row-selection algorithm uses as a subroutine. Section 4 then describes the row-selection algorithm itself, and Section 5 analyzes its running time. In Section 6, we present a faster algorithm for computing approximate row medians. Finally, we make some concluding remarks and present several open problems in Section 7.

2 Two Properties of Totally Monotone Arrays

In this section, we present the two simple properties of totally monotone arrays that we use in obtaining our row-selection algorithm. These are the only two properties that we require.

Property 1 Let $A = \{a[i,j]\}$ denote an $m \times n$ array, and let B denote any subarray of A corresponding to a subset of A's rows and columns. If A is totally monotone, then B is also totally monotone. ■

This property follows immediately from the definition of a totally monotone array.

The second property relates what we will call the *left* and *right ranks* of entries from the same column of a totally monotone array. Imagine an $m \times n$ array $A = \{a[i,j]\}$ drawn in the plane as in Figure 2, so that $a[1,1]$ is the upper- and leftmost entry and $a[m,n]$ is the lower- and rightmost entry. Roughly speaking, the left rank of $a[i,j]$ in row i of A, denoted $L(i,j)$, is the number of entries to the left of $a[i,j]$ that are smaller than $a[i,j]$, and the right rank of $a[i,j]$ in row i of A, denoted $R(i,j)$, is the number of entries to the right of $a[i,j]$ that are not bigger than $a[i,j]$. More precisely,

$$L(i,j) = |\{j' : 1 \leq j' < j \text{ and } a[i,j'] < a[i,j]\}|$$

and

$$R(i,j) = |\{j' : j \leq j' \leq m \text{ and } a[i,j'] \leq a[i,j]\}| .$$

$$\begin{array}{cccccccc}
 & & & j & & & & \\
 & 49 & 38 & 23 & 28 & 13 & 17 & 10 \\
 & 45 & 35 & 23 & 29 & 16 & 22 & 17 \\
 & 40 & 33 & 24 & 34 & 22 & 28 & 24 \\
i_1 & 21 & 15 & \boxed{7} & 17 & 6 & 13 & 11 \\
 & 32 & 28 & 23 & 37 & 32 & 44 & 45 \\
 & 10 & 7 & 6 & 21 & 19 & 33 & 36 \\
i_2 & 31 & 30 & \boxed{34} & 53 & 51 & 66 & 73 \\
 & 6 & 9 & 13 & 32 & 32 & 52 & 62 \\
 & 8 & 15 & 21 & 43 & 45 & 66 & 77
\end{array}$$

Figure 2: Given any two rows $i_1 < i_2$ and any column j from a totally monotone array $A = \{a[i,j]\}$, the left rank of $a[i_1,j]$ in row i_1 is at most the left rank of $a[i_2,j]$ in row i_2 (i.e., $L(i_1,j) \leq L(i_2,j)$), and the right rank of $a[i_1,j]$ in row i_1 is at least the right rank of $a[i_2,j]$ in row i_2 (i.e., $R(i_1,j) \geq R(i_2,j)$). For example, in the totally monotone array depicted above, $L(4,5) = 0 \leq L(7,5) = 2$ and $R(4,5) = 2 \geq R(7,5) = 1$.

We have made the inequalities in the definition of $R(i,j)$ nonstrict so that for $1 \leq i \leq m$ and $1 \leq j \leq n$, the sum $r(i,j) = L(i,j) + R(i,j)$ corresponds to what is commonly called the *rank* of $a[i,j]$ in row i of A. (By convention, we assign a higher rank to the leftmost of two identical entries, so that for every row i, the ranks $r(i,1), r(i,2), \ldots, r(i,n)$ are distinct.) Note that, in terms of ranks, the row-selection problem for A is that of determining for each row i the unique column j such that $r(i,j) = k$.

Property 2 Let $A = \{a[i,j]\}$ denote an $m \times n$ array. If A is totally monotone, then for any j in the range $1 \leq j \leq n$, the left ranks in column j of A are nondecreasing in i and the right ranks in column j are nonincreasing in i. In other words,

$$L(1,j) \leq L(2,j) \leq \cdots \leq L(m,j)$$

and

$$R(1,j) \geq R(2,j) \geq \cdots \geq R(m,j) .$$

Proof Suppose $L(i,j) > L(i+1,j)$ for some i in the range $1 \leq i < m$ and some j in the range $1 \leq j \leq n$. This assumption implies that there exists a j' such that $1 \leq j' < j$, $a[i,j'] < a[i,j]$, and $a[i+1,j'] \geq a[i+1,j]$, which contradicts our assumption that A is totally monotone. Similarly, if we assume $R(i,j) < R(i+1,j)$ for some i in the range $1 \leq i < m$ and some j in the range $1 \leq j \leq n$, then there exists a j' such that $j < j' \leq n$, $a[i,j'] > a[i,j]$, and $a[i+1,j'] \leq a[i+1,j]$, which again leads to a contradiction. ■

3 Computing Left and Right Ranks

We need one more building block for our row-selection algorithm: an algorithm for computing left and right ranks. We begin by generalizing the notion of left and right ranks introduced in the previous section to arbitrary ordered sets of real numbers. We then describe an $O(n \lg n)$-time algorithm for computing the left and right ranks associated with an ordered set of size n.

Given a set $S = \{a_1, a_2, \ldots, a_n\}$, for $1 \leq j \leq n$, we define the left and right ranks of a_j in S, denoted $L(a_j, S)$ and $R(a_j, S)$, respectively, as follows:

$$L(a_j, S) = |\{j' : 1 \leq j' < j \text{ and } a_{j'} < a_j\}|$$

and

$$R(a_j, S) = |\{j' : j \leq j' \leq n \text{ and } a_{j'} \leq a_j\}| .$$

In terms of this new notation, the left and right ranks of entry $a[i,j]$ in row i of array A (denoted $L(i,j)$ and $R(i,j)$, respectively, in the last section) are $L(a[i,j], S_i)$ and $R(a[i,j], S_i)$, respectively, where S_i is the ordered set corresponding to row i of A.

Clearly, sorting the elements of an ordered set $S = \{a_1, a_2, \ldots, a_n\}$ is no harder than computing its elements' left and right ranks, since for any j in the range $1 \leq j \leq n$, the rank of a_j in S (denoted $r(a_j, S)$ for consistency) is $L(a_j, S) + R(a_j, S)$. Moreover, as Lemma 1 shows, computing left and right ranks is no harder than computing ranks (i.e., sorting)[1].

Lemma 1 Given an ordered set $S = \{a_1, a_2, \ldots, a_n\}$, of real numbers, we can compute $L(a_j, S)$ and $R(a_j, S)$ for $1 \leq j \leq n$ in $O(n \lg n)$ time.

Proof To compute the $L(a_j, S)$ and $R(a_j, S)$ in $O(n \lg n)$ time, we use a divide-and-conquer approach reminiscent of mergesort. We begin by partitioning S into two subsets S' and S'' so that S' contains the first $\lfloor n/2 \rfloor$

[1] This computational equivalence is not surprising, as there exists a one-to-one correspondence between sequences of left ranks $L(a_1, S), L(a_2, S), \ldots, L(a_n, S)$ and sequences of ranks $r(a_1, S), r(a_2, S), \ldots, r(a_n, S)$ for n-element ordered sets S.

values $a_1, \ldots, a_{\lfloor n/2 \rfloor}$ and S'' contains the remaining values $a_{\lfloor n/2 \rfloor+1}, \ldots, a_n$. We then recursively compute the $L(a_j, S')$ and $R(a_j, S')$ for $1 \le j \le \lfloor n/2 \rfloor$ and the $L(a_j, S'')$ and $R(a_j, S'')$ for $\lfloor n/2 \rfloor < j \le n$. Then, for $1 \le j \le \lfloor n/2 \rfloor$, $L(a_j, S) = L(a_j, S')$, and for $\lfloor n/2 \rfloor < j \le n$, $R(a_j, S) = R(a_j, S'')$. Furthermore, for $1 \le j \le \lfloor n/2 \rfloor$, $R(a_j, S) = R(a_j, S') + r_{\le}(a_j, S'')$, and for $\lfloor n/2 \rfloor < j \le n$, $L(a_j, S) = L(a_j, S'') + r_{<}(a_j, S')$, where for $b \notin T = \{b_1, \ldots, b_N\}$, we define

$$r_{\le}(b, T) = |\{j : 1 \le j \le N \text{ and } b_j \le b\}|$$

and

$$r_{<}(b, T) = |\{j : 1 \le j \le N \text{ and } b_j < b\}| .$$

Since we know the sorted order of $a_1, \ldots, a_{\lfloor n/2 \rfloor}$ and the sorted order of $a_{\lfloor n/2 \rfloor+1}, \ldots, a_n$, we can merge these two lists in $O(n)$ time, thereby obtaining $r_{\le}(a_j, S'')$ for $1 \le j \le \lfloor n/2 \rfloor$ and $r_{<}(a_j, S')$ for $\lfloor n/2 \rfloor < j \le n$. This gives the following (familiar) recurrence for the time $T(n)$ to compute $L(a_1, S), L(a_2, S), \ldots, L(a_n, S)$ and $R(a_1, S), R(a_2, S), \ldots, R(a_n, S)$:

$$T(n) = \begin{cases} T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + O(n) & \text{if } n \ge 2, \\ O(1) & \text{if } n = 1. \end{cases}$$

As the solution for this recurrence is $T(n) = O(n \lg n)$, this completes our proof. ■

4 The Row-Selection Algorithm

With the preliminaries of Sections 2 and 3 behind us, we can now describe our $O((\sqrt{m} \lg m)(n \lg n) + m \lg n)$-time algorithm for computing the kth smallest entry in each row of an $m \times n$ totally monotone array $A = \{a[i,j]\}$. If $m \le 4$, we use the linear-time selection algorithm of Blum *et al.* [BFP+73] to obtain the kth smallest entry in each row in $O(n)$ time. Otherwise, we use the following divide-and-conquer approach.

We begin by partitioning A into x subarrays $A_1, \cdots, A_x$, where x is a parameter of our algorithm in the range $1 < x \le m$. (We will later set $x = \lceil \sqrt{m} \rceil$ to minimize our algorithm's running time, but for now, it is simpler to think of x as a parameter.) In particular, for $1 \le t \le x$, A_t consists of rows $(t-1)\lceil m/x \rceil + 1$ through $t\lceil m/x \rceil$ of A. (The last subarray A_x may actually contain fewer than $\lceil m/x \rceil$ rows, but for simplicity, we will ignore this detail.) In order to simplify the notation, we set

$$\lambda_t(x) = (t-1)\lceil m/x \rceil + 1 ;$$

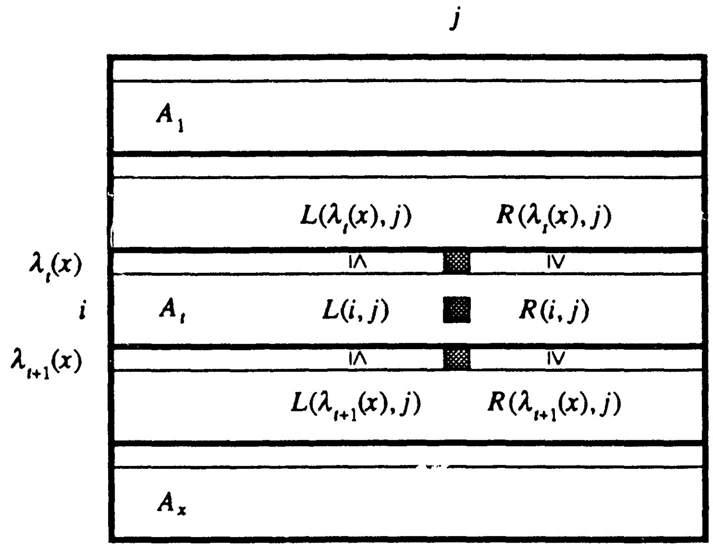

Figure 3: Suppose $\lambda_t(x) \leq i < \lambda_{t+1}(x)$ (i.e., row i of A is in A_t) and $1 \leq j \leq n$. Since A is totally monotone, we must have $L(\lambda_t(x),j) \leq L(i,j) \leq L(\lambda_{t+1}(x),j)$ and $R(\lambda_t(x),j) \geq R(i,j) \geq R(\lambda_{t+1}(x),j)$. Thus, the rank $r(i,j)$ of $a[i,j]$ in row i of A must lie between $L(\lambda_t(x),j) + R(\lambda_{t+1}(x),j)$ and $L(\lambda_{t+1}(x),j) + R(\lambda_t(x),j)$.

thus, A_t includes rows $\lambda_t(x)$ through $\lambda_{t+1}(x) - 1$ of A.

We then compute left and right ranks in the first row of each A_t. Specifically, for $1 \leq t \leq x$ and $1 \leq j \leq n$, we compute $L(\lambda_t(x),j)$ and $R(\lambda_t(x),j)$. By Lemma 1, this computation can be performed in $O(n \lg n)$ time per row (i.e., $O(xn \lg n)$ total time).

Now, by Property 2, for $1 \leq t \leq x$, $\lambda_t(x) \leq i < \lambda_{t+1}(x)$, and $1 \leq j \leq n$, we have

$$L(\lambda_t(x),j) \;\leq\; L(i,j) \;\leq\; L(\lambda_{t+1}(x),j)$$

and

$$R(\lambda_t(x),j) \;\geq\; R(i,j) \;\geq\; R(\lambda_{t+1}(x),j)\,.$$

(To handle the last row, we define $L(\lambda_{x+1}(x),j) = j-1$ and $R(\lambda_{x+1}(x),j) = 1$ for $1 \leq j \leq n$.) These bounds are illustrated in Figure 3. Since $r(i,j) = L(i,j) + R(i,j)$, this gives us lower and upper bounds on the rank of every entry $a[i,j]$ in the jth column of A_t; specifically, for $\lambda_t(x) \leq i < \lambda_{t+1}(x)$, we must have

$$\alpha(t,j) \;\leq\; r(i,j) \;\leq\; \beta(t,j)$$

where

$$\alpha(t,j) = L(\lambda_t(x),j) + R(\lambda_{t+1}(x),j)$$

and

$$\beta(t,j) = L(\lambda_{t+1}(x),j) + R(\lambda_t(x),j) .$$

The above bounds allow us to reduce the size of our row-selection problem. In particular, for all t in the range $1 \leq t \leq x$ and all j in the range $1 \leq j \leq n$, we can compute $\alpha(t,j)$ and $\beta(t,j)$ and then delete column j frc n A_t if $\alpha(t,j) > k$ or $\beta(t,j) < k$, as in both these cases, no entry in column j of A_t can have rank k in its row. Deleting columns in this manner takes $O(xn)$ time, since $A_1, \ldots, A_x$ contain xn total columns.

Now let A'_t denote the subarray of A_t consisting of those columns not deleted, i.e., those columns j such that

$$\alpha(t,j) \leq k \leq \beta(t,j) .$$

The kth smallest entry in each row of A_t is the k_tth smallest entry in each row of A'_t, where

$$k_t = k - |\{j : 1 \leq j \leq n \text{ and } \beta(t,j) < k\}| .$$

Thus, since $A'_1, A'_2, \ldots, A'_x$ are all totally monotone (by Property 1), we need only recursively solve x smaller row-selection problems to obtain the kth smallest entry in each row of A. This completes the description of our row-selection algorithm.

5 Running-Time Analysis

In this section, we analyze the running time of our row-selection algorithm. In particular, we prove the following theorem.

Theorem 2 The row-selection algorithm described in Section 4 computes the kth smallest entry in each row of an $m \times n$ totally monotone array in $O((\sqrt{m}\lg m)(n \lg n) + m \lg n)$ time.

Proof The correctness of the algorithm follows from arguments sketched in Section 4. As for the algorithm's time complexity, the key issue in bounding the algorithm's running time is bounding the number of columns eliminated from $A_1, A_2, \ldots, A_x$. Intuitively, the more columns eliminated, the smaller the problems that remain and the better the running time of our algorithm.

Before proving any bounds, we first introduce some notation. Let n_t denote the number of columns in A'_t, i.e.,

$$n_t = |\{j : \alpha(t,j) \leq k \leq \beta(t,j)\}| .$$

In terms of this new notation, our algorithm spends $O(xn \lg n)$ time reducing one row-selection problem of size $m \times n$ to x row-selection problems such that the tth problem has size $\lceil m/x \rceil \times n_t$. (Since the bound we want to prove on our algorithm's running time does not depend on k, we can ignore the change from looking for the kth smallest entry in each row of A to looking for the k_tth smallest entry in each row of A_t.) In what follows, we derive a bound on $\sum_{1 \leq t \leq x} n_t$, which we then use to bound the running time of our algorithm.

To bound $\sum_{1 \leq t \leq x} n_t$, we first select x representative rows $I_1, I_2, \ldots, I_x$, one from each of the A_t. (I_t may take any value from $\lambda_t(x)$ to $\lambda_{t+1}(x) - 1$.) Then, for $1 \leq t \leq x$ and $1 \leq j \leq n$, consider the difference $r(I_t, j) - \alpha(t,j)$. This difference is always nonnegative, since $\alpha(t,j)$ is a lower bound on $r(I_t, j)$. Moreover, since $r(I_t, j)$ is at most n,

$$\sum_{\substack{1 \leq t \leq x \\ 1 \leq j \leq n}} (r(I_t, j) - \alpha(t,j)) \leq xn^2 .$$

However, we can prove a tighter $O(n^2)$ bound on this sum as follows.

From the definition of $\alpha(t,j)$, we have

$$\begin{aligned}
\sum_{\substack{1 \leq t \leq x \\ 1 \leq j \leq n}} \alpha(t,j) &= \left(\sum_{\substack{1 \leq t \leq x \\ 1 \leq j \leq n}} L(\lambda_t(x), j) \right) + \left(\sum_{\substack{1 \leq t \leq x \\ 1 \leq j \leq n}} R(\lambda_{t+1}(x), j) \right) \\
&\geq \left(\sum_{\substack{1 \leq t \leq x-1 \\ 1 \leq j \leq n}} (L(\lambda_{t+1}(x), j) + R(\lambda_{t+1}(x), j)) \right) + n \\
&= \left(\sum_{\substack{1 \leq t \leq x-1 \\ 1 \leq j \leq n}} r(\lambda_{t+1}(x), j) \right) + n .
\end{aligned}$$

(The n term in the above expression comes from the last subarray I_x and our convention that $R(\lambda_{x+1}(x), j) = 1$.) Furthermore, for all rows i,

$$\sum_{1 \le j \le n} r(i,j) \;=\; 1 + 2 + \cdots + n \;=\; \frac{n(n+1)}{2} \;.$$

Thus,

$$\begin{aligned}\sum_{\substack{1 \le t \le x \\ 1 \le j \le n}} (r(I_t, j) - \alpha(t,j)) \;&\le\; x\frac{n(n+1)}{2} - (x-1)\frac{n(n+1)}{2} - n \\ &=\; \frac{n(n-1)}{2} \;. \qquad (1)\end{aligned}$$

In a similar fashion, we can show that

$$\sum_{\substack{1 \le t \le x \\ 1 \le j \le n}} (\beta(t,j) - r(I_t, j)) \;\le\; \frac{n(n-1)}{2} \;. \qquad (2)$$

Now let N_1 denote the total number of entries $a[I_t, j]$ in rows $I_1, I_2, \ldots, I_x$ such that $r(I_t, j) < k \le \beta(t,j)$, and let N_2 denote the total number of entries $a[I_t, j]$ in rows $I_1, I_2, \ldots, I_x$ such that $\alpha(t,j) \le k < r(I_t, j)$. Since column j of A_t is a column of A'_t if and only if

$$\alpha(t,j) \le k \le \beta(t,j) \;,$$

we must have

$$\sum_{1 \le t \le x} n_t \;=\; N_1 + N_2 + x \;.$$

(The x term in the above expression comes from the unique column in each A_t such that $r(I_t, j) = k$.)

An upper bound on N_1 follows from (2). Since the entries in rows $I_1, I_2, \ldots, I_x$ satisfying $r(I_t, j) < k \le \beta(t,j)$ are a subset of all the entries in rows $I_1, I_2, \ldots, I_x$, we must have

$$\sum_{\substack{1 \le t \le x \\ 1 \le j \le n \\ \text{s.t. } r(I_t, j) < k \le \beta(t,j)}} (\beta(t,j) - r(I_t, j)) \;\le\; \sum_{\substack{1 \le t \le x \\ 1 \le j \le n}} (\beta(t,j) - r(I_t, j)) \;.$$

For any rank ℓ in the range $1 \leq \ell \leq n$, there is exactly one entry $a[I_t, j]$ in each row I_t, such that $r(I_t, j) = \ell$. Thus, for any positive d, there are at most xd entries $a[I_t, j]$ in rows $I_1, I_2, \ldots, I_x$ such that $r(I_t, j) < k \leq \beta(t, j)$ and $\beta(t, j) - r(I_t, j) \leq d$. Picking q_1 and r_1 so that $N_1 = q_1 x + r_1$ and $0 \leq r_1 < x$, this last observation implies

$$\begin{aligned}
\sum_{\substack{1 \leq t \leq x \\ 1 \leq j \leq n \\ \text{s.t. } r(I_t, j) < k \leq \beta(t, j)}} (\beta(t, j) - r(I_t, j)) &\geq \left(\sum_{s=1}^{q_1} xs \right) + r_1(q_1 + 1) \\
&= \left(\frac{q_1 x + 2r_1}{2} \right) (q_1 + 1) \\
&\geq \frac{N_1^2}{2x} .
\end{aligned}$$

Combining the preceding two inequalities with (2), we find

$$\frac{N_1^2}{2x} \leq \frac{n(n-1)}{2} \leq \frac{n^2}{2}$$

or

$$N_1 \leq \sqrt{x} n .$$

By a similar argument, we can show

$$N_2 \leq \sqrt{x} n .$$

Thus, the total number of columns in $A'_1, A'_2, \ldots, A'_x$ is at most $2\sqrt{x}n + x$.

Given the above upper bound on the number of columns in the arrays $A'_1, A'_2, \ldots, A'_x$, we can now write down a recurrence relation describing the running time of our algorithm (as a function of x). Let $T(m, n)$ denote the time spent computing the kth smallest entry in each row of an $m \times n$ totally monotone array. Then, for $m > 4$,

$$T(m, n) = O(xn \lg n) + \max_{\substack{n_1, n_2, \ldots, n_x \\ \text{s.t. } n_1 + \cdots + n_x \leq 2\sqrt{x}n + x \\ \text{and } 1 \leq n_t \leq n \text{ for } 1 \leq t \leq x}} \sum_{t=1}^{x} T(\lceil m/x \rceil, n_t) .$$

To obtain the desired running time for our algorithm, we use $x = \left\lceil \sqrt{m} \right\rceil$. The above recurrence then yields $T(m, n) = O((\sqrt{m} \lg m)(n \lg n) + m \lg n)$. (The interested reader is referred to [MPSS91] for a detailed proof of this claim.) ■

6 Computing Approximate Row Medians

In Section 4, we described an algorithm for the general row-selection problem. This algorithm can be used to compute the median (i.e., $\lceil n/2 \rceil$th smallest) entry in each row of an $m \times n$ totally monotone array $A = \{a[i,j]\}$ in $O((\sqrt{m} \lg m)(n \lg n) + m \lg n)$ time. In this section, we will show that this time bound can be significantly improved, *provided* we are willing to settle for an approximation to each row's median entry. Specifically, we will describe an $O(n \lg m)$-time algorithm that identifies an entry $a[i,j^*]$ in each row i of A whose rank $r(i,j^*)$ in row i satisfies $\lfloor n/4 \rfloor \leq r(i,j^*) < \lceil 3n/4 \rceil$. (We call such an entry an *approximate row median.*)

Our approximation algorithm consists of two phases. To describe the first phase, we need a bit of additional notation. Let A_L denote the $m \times \lceil n/2 \rceil$ subarray of A consisting of columns 1 through $\lceil n/2 \rceil$ of A, and let A_R denote the $m \times \lfloor n/2 \rfloor$ subarray of A consisting of columns $\lceil n/2 \rceil + 1$ through n. Furthermore, for $1 \leq i \leq m$, let p_i denote the column of A containing the $(\lceil n/4 \rceil - 1)$st smallest entry in row i of A_L, and let q_i denote the column of A containing the $(\lfloor n/4 \rfloor - 1)$st smallest entry in row i of A_R.

In the first phase of our approximation algorithm, we find an I in the range $0 \leq I \leq m$ such that $a[I,p_I] \geq a[I,q_I]$ and $a[I+1,p_{I+1}] < a[I+1,q_{I+1}]$. If we define $a[0,p_0] \geq a[0,q_0]$ and $a[m+1,p_{m+1}] < a[m+1,q_{m+1}]$, then such an I always exists. Moreover, we can find such an I in $O(n(1+\lg m))$ time, using the following binary search.

Initially, we set $s = 0$ and $t = m+1$, so that $a[s,p_s] \geq a[s,q_s]$ and $a[t,p_t] < a[t,q_t]$. Then, so long as $t - s > 1$, we repeat the following series of steps. First, we set $r = \lceil (s+t)/2 \rceil$. We then use the linear-time selection algorithm of Blum *et al.* [BFP$^+$73] to compute $a[r,p_r]$ and $a[r,q_r]$ in $O(n)$ time. Finally, we compare $a[r,p_r]$ and $a[r,q_r]$. If $a[r,p_r] \geq a[r,q_r]$, we set $s = r$; otherwise, we set $t = r$.

The above procedure maintains the invariant that $a[s,p_s] \geq a[s,q_s]$ and $a[t,p_t] < a[t,q_t]$. Moreover, each step halves $t - s$. Thus, after $O(1+\lg m)$ steps and $O(n(1+\lg m))$ total time, $s = t-1$ is the I we seek.

We begin the second phase of our algorithm by locating approximate medians for rows 1 through I of A. Let U denote the I-row subarray of A consisting of rows 1 through I of A and those columns j such that $1 \leq j \leq \lceil n/2 \rceil$ and $a[I,j] \geq a[I,p_I]$. Using the SMAWK algorithm, we can compute the row minima of U in $O(n)$ time if $I \leq n$ and in $O(n(1+\lg(I/n)))$ time if $I > n$. Now consider the minimum entry $a[i,j^*]$ in row i of U. Since the subarray U contains $\lceil n/2 \rceil - (\lceil n/4 \rceil - 1) + 1 \geq \lfloor n/4 \rfloor + 2$ columns and since

$a[i,j^*]$ is a row minimum for U, row i of A contains at least $\lfloor n/4 \rfloor + 1$ entries that are larger than $a[i,j^*]$, which implies the rank $r(i,j^*)$ of $a[i,j^*]$ in row i of A is strictly less than $\lceil 3n/4 \rceil$. On the other hand, since $a[I,p_I] \geq a[I,q_I]$, at least $\lfloor n/4 \rfloor - 1$ entries in the right half of row I (i.e., in A_R) are at most $a[I,j^*]$. This observation implies $R(I,j^*) \geq \lfloor n/4 \rfloor$. Invoking Property 2, we then have $R(i,j^*) \geq \lfloor n/4 \rfloor$, which implies $r(i,j^*) \geq \lfloor n/4 \rfloor$. Thus, $a[i,j^*]$ is an approximate median for row i of A.

To find approximate medians for the remaining rows of A (i.e., rows $I+1$ through m), let D denote the $(m-I)$-row subarray of A consisting of rows $I+1$ through m of A and those columns j such that $\lceil n/2 \rceil < j \leq n$ and $a[I,j] \geq a[I,q_I]$. Using the SMAWK algorithm, we can compute the row minima of D in $O(n)$ time if $m - I \leq n$ and in $O(n(1 + \lg((m-I)/n)))$ time if $m - I > n$. Now consider the minimum entry $a[i,j^*]$ in row $i - I$ of D. Since the subarray D contains $\lfloor n/2 \rfloor - (\lfloor n/4 \rfloor - 1) + 1 \geq \lfloor n/4 \rfloor + 2$ columns and since $a[i,j^*]$ is a row minimum for the subarray D, row i of A contains at least $\lfloor n/4 \rfloor + 1$ entries larger than $a[i,j^*]$, which implies the rank $r(i,j^*)$ of $a[i,j^*]$ in row i of A is strictly less than $\lceil 3n/4 \rceil$. On the other hand, since $a[I+1,p_{I+1}] < a[I+1,q_{I+1}]$, at least $\lceil n/4 \rceil - 1$ entries in the left half of row $I+1$ (i.e., in A_L) are smaller than $a[I+1,j^*]$. This observation implies $L(I+1,j^*) \geq \lceil n/4 \rceil - 1$. Invoking Property 2, we then have $L(i,j^*) \geq \lceil n/4 \rceil - 1$, which implies $r(i,j^*) \geq \lceil n/4 \rceil$. Thus, $a[i,j^*]$ is an approximate median for row i of A.

Theorem 3 The algorithm described above finds an approximate median in each row of an $m \times n$ totally monotone array in $O(n \lg m)$ time.

Proof The correctness of the algorithm follows from arguments sketched in its description. As for the algorithm's time complexity, the search for row I takes $O(n \lg m)$ time, while the rest of the computation takes $O(n)$ time if $m \leq n$ and $O(n(1 + \lg(m/n)))$ time if $m > n$. ■

7 Concluding Remarks

In this paper, we presented a new algorithm for row selection in totally monotone arrays. For large values of k (in particular, for $k = \lceil n/2 \rceil$), this algorithm is significantly faster than the best previous algorithm for the problem, due to Kravets and Park [KP91]. We also gave an even faster algorithm for approximating the median entry in each row of a totally monotone array.

As a closing comment, we note that the row-selection algorithm of Section 4 locates not only the kth smallest entry in each row of the given totally monotone array but also the 1st through $(k-1)$st smallest entries. Roughly speaking, these entries are in the columns deleted because $\beta(t,j) < k$.

We also remark that a slight reduction in the running time of our row-selection algorithm can be obtained as follows. In [KP91], Kravets and Park gave an algorithm that sorts the rows of an $m \times n$ totally monotone array in $O(mn)$ time when $m \geq n$ and in $O(mn(1+\lg(n/m)))$ time when $m < n$. This algorithm is easily adapted to computing the left and right ranks of every entry in a totally monotone array. Using this algorithm rather than the simpler algorithm of Section 3 reduces the running time of our row-selection algorithm to $O((\sqrt{m}\lg m)(n(1+\lg(n/\sqrt{m}))))$ when $n/\lg n \geq \sqrt{m}$.

We conclude with the following open questions:

1. The only lower bound known on the time necessary for computing the kth smallest entry in an $m \times n$ totally monotone array is $\Omega(n)$ if $m \leq n$ and $\Omega(n(1+\lg(m/n)))$ if $m > n$. Thus, ascertaining the asymptotic time complexity of the row-selection problem for totally monotone arrays remains an open problem for general k.

2. In [KP91], Kravets and Park also considered the following *array-selection* problem: given an $m \times n$ totally monotone array $A = \{a[i,j]\}$, find the kth smallest (or kth largest) of A's mn entries. They gave an $O(m+n+k\lg(mn/k))$-time algorithm for this problem; this algorithm is based on their $O(k(m+n))$-time algorithm for the row-selection problem. It remains open as to whether the row-selection techniques of this paper can be used in a similar fashion to obtain a faster algorithm for the array-selection problem.

References

[AALM90] A. Apostolico, M. J. Atallah, L. L. Larmore, and H. S. McFaddin. Efficient parallel algorithms for string editing and related problems. *SIAM Journal on Computing*, 19(5):968–988, 1990.

[AK91] M. J. Atallah and S. R. Kosaraju. An efficient parallel algorithm for the row minima of a totally monotone matrix. In *Proceedings of the 2nd Annual ACM-SIAM Symposium on Dis-*

crete Algorithms, pages 394–403, 1991. Submitted to *Journal of Algorithms.*

[AKL$^+$89] M. J. Atallah, S. R. Kosaraju, L. L. Larmore, G. Miller, and S. Teng. Constructing trees in parallel. In *Proceedings of the 1st Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 421–431, 1989.

[AKM$^+$87] A. Aggarwal, M. M. Klawe, S. Moran, P. Shor, and R. Wilber. Geometric applications of a matrix-searching algorithm. *Algorithmica*, 2(2):195–208, 1987.

[AKPS90] A. Aggarwal, D. Kravets, J. K. Park, and S. Sen. Parallel searching in generalized Monge arrays with applications. In *Proceedings of the 2nd Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 259–268, 1990.

[AP89a] A. Aggarwal and J. K. Park. Parallel searching in multidimensional monotone arrays. Research Report RC 14826, IBM T. J. Watson Research Center, August 1989. Submitted to *Journal of Algorithms.* Portions of this paper appear in *Proceedings of the 29th Annual IEEE Symposium on Foundations of Computer Science*, pages 497–512, 1988.

[AP89b] A. Aggarwal and J. K. Park. Sequential searching in multidimensional monotone arrays. Research Report RC 15128, IBM T. J. Watson Research Center, November 1989. Submitted to *Journal of Algorithms.* Portions of this paper appear in *Proceedings of the 29th Annual IEEE Symposium on Foundations of Computer Science*, pages 497–512, 1988.

[AP91] A. Aggarwal and J. K. Park. Improved algorithms for economic lot-size problems. *Operations Research*, 1991. To appear. An earlier version of this paper appears as Research Report RC 15626, IBM T. J. Watson Research Center, March 1990.

[Ata90] M. J. Atallah. A faster parallel algorithm for a matrix searching problem. In G. Goos and J. Hartmanis, editors, *Proceedings of the 2nd Scandinavian Workshop on Algorithm Theory*, pages 192–200, New York, NY, 1990. Springer-Verlag. Submitted to *Algorithmica.*

[BFP+73] M. Blum, R. W. Floyd, V. Pratt, R. L. Rivest, and R. E. Tarjan. Time bounds for selection. *Journal of Computer and System Sciences*, 7(4):448–461, 1973.

[Epp90] D. Eppstein. Sequence comparison with mixed convex and concave costs. *Journal of Algorithms*, 11(1):85–101, 1990.

[GP90] Z. Galil and K. Park. A linear-time algorithm for concave one-dimensional dynamic programming. *Information Processing Letters*, 33(6):309–311, 1990.

[Hof63] A. J. Hoffman. On simple linear programming problems. In V. Klee, editor, *Convexity: Proceedings of the Seventh Symposium in Pure Mathematics of the AMS*, volume 7 of *Proceedings of Symposia in Pure Mathematics*, pages 317–327. American Mathematical Society, Providence, RI, 1963.

[KK90] M. M. Klawe and D. J. Kleitman. An almost linear time algorithm for generalized matrix searching. *SIAM Journal on Discrete Mathematics*, 3(1):81–97, 1990.

[Kla89] M. M. Klawe. A simple linear time algorithm for concave one-dimensional dynamic programming. Technical Report 89-16, University of British Columbia, Vancouver, 1989.

[KP91] D. Kravets and J. K. Park. Selection and sorting in totally monotone arrays. *Mathematical Systems Theory*, 1991. To appear. An earlier version of this paper appears in *Proceedings of the 1st Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 494–502, 1990.

[LP91] L. L. Larmore and T. M. Przytycka. Parallel construction of trees with optimal weighted path length. In *Proceedings of the 3rd Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 71–80, 1991.

[LS91] L. L. Larmore and B. Schieber. On-line dynamic programming with applications to the prediction of RNA secondary structure. *Journal of Algorithms*, 1991. To appear. An earlier version of this paper appears in *Proceedings of the 1st Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 503–512, 1990.

[MPSS91] Y. Mansour, J. K. Park, B. Schieber, and S. Sen. Improved selection in totally monotone arrays. Research Report RC 16911, IBM T. J. Watson Research Center, May 1991.

[Wil88] R. Wilber. The concave least-weight subsequence problem revisited. *Journal of Algorithms*, 9(3):418–425, 1988.

END

DATE FILMED

11 / 14 / 91