Selection and Sorting in Totally Monotone Arrays*

Dina Kravets and James K. Park

Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA 02139, USA

Abstract. A two-dimensional array $A = \{a[i, j]\}$ is called 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 $a[i_2, j_1] < a[i_2, j_2]$. Totally monotone arrays were introduced in 1987 by Aggarwal, Klawe, Moran, Shor, and Wilber, who showed that several problems in computational geometry and VLSI river routing could be reduced to the problem of finding a maximum entry in each row of a totally monotone array. In this paper we consider several selection and sorting problems involving totally monotone arrays and give a number of applications of solutions for these problems. In particular, we obtain the following results for an $m \times n$ totally monotone array $A$:

1. The $k$ largest (or $k$ smallest) entries in each row of $A$ can be computed in $O(k(m + n))$ time. This result allows us to determine the $k$ farthest (or $k$ nearest) neighbors of each vertex of a convex $n$-gon in $O(kn)$ time.

2. Provided the transpose of $A$ is also totally monotone, the $k$ largest (or $k$ smallest) entries overall in $A$ can be computed in $O(m + n + k \lg(mn/k))$ time. This result allows us to find the $k$ farthest (or $k$ nearest) pairs of vertices from a convex $n$-gon in $O(n + k \lg(n^2/k))$ time.

3. The rows of $A$ can be sorted in $O(mn)$ time when $m \geq n$ and in $O(mn(1 + \lg(n/m)))$ time when $m < n$. This result allows us to solve the

following problem in $O(n^2 \log l)$ time: given $l$ convex polygons with a total of $n$ vertices, for all vertices $v$, sort the other vertices by distance from $v$.
4. Sorting all the entirets of $A$ requires $\Omega(mn \log m)$ time.

1. Introduction

1.1. Motivation and Previous Work on Totally Monotone Arrays

Let $A = \{a[i, j]\}$ denote an $m \times n$ array, and for all $i$, let $c(i)$ denote the column of $A$ containing the lefmost maximum entry in row $i$ of $A$. In other words, $c(i)$ is the unique column of $A$ such that, for all $j < c(i)$, $a[i, j] < a[i, c(i)]$ and, for all $j \geq c(i)$, $a[i, j] \leq a[i, c(i)]$. $A$ is called monotone if the $c(i)$ are nondecreasing in $i$, i.e., $c(1) \leq c(2) \leq \cdots \leq c(m)$. $A$ is called totally monotone if every $2 \times 2$ subarray of $A$, corresponding to any two rows and any two columns of $A$, is monotone. Equivalently, $A$ is totally monotone if, for all $i_1 < i_2$ and $j_1 < j_2$, $a[i_1, j_1] < a[i_2, j_2]$ implies $a[i_2, j_1] < a[i_2, j_2]$, as is suggested in Figure 1(a). Note that total monotonicity implies monotonicity.

Although the notion of a totally monotone array may seem rather odd at first glance, such arrays arise naturally in a wide variety of contexts. For example, consider a convex polygon $P$ in the plane with vertices $p_1, \ldots, p_n$ in clockwise order. As Aggarwal et al. observed in [AKM$^+$], the distances separating pairs of vertices of $P$ form a totally monotone array. Specifically, let $A = \{a[i, j]\}$ denote the $n \times (2n - 1)$ array given by the equation

$$a[i, j] = \begin{cases} 
(j - i) & \text{if } 1 \leq j < i, \\
\frac{d(p_i, p_j)}{d(p_i, p_j - n)} & \text{if } i \leq j \leq n, \\
\frac{d(p_i, p_j - n)}{d(p_i, p_j - n)} & \text{if } n < j < i + n, \\
i + n - j - 1 & \text{if } i + n \leq j < 2n,
\end{cases}$$

where $d(p_i, p_j)$ denotes the Euclidean distance between $p_i$ and $p_j$. (We call this array the distance array for $P$.) The nonnegative entries in row $i$ of $A$ are precisely

![Fig. 1. (a) In a totally monotone array, for no $i_1 < i_2$ and $j_1 < j_2$ is $a[i_1, j_1] < a[i_1, j_2]$ and $a[i_2, j_1] > a[i_2, j_2]$. (b) For any quadrilateral with vertices $p_{i_1}, p_{i_2}, p_{j_1},$ and $p_{j_2}$ in clockwise order, $d(p_{i_1}, p_{j_1}) + d(p_{i_2}, p_{j_2}) > d(p_{i_1}, p_{j_2}) + d(p_{i_2}, p_{j_1})$.](image-url)
the \( n \) distances separating \( p_i \) from \( p_1, \ldots, p_n \); moreover, both \( A \) and its transpose are totally monotone, as the following lemma shows.

**Lemma 1.** \( A \) and its transpose \( A^T \) are both totally monotone.

**Proof.** Let \( i_1 \) and \( i_2 \) denote any two rows of \( A \), where \( i_1 < i_2 \), and let \( j_1 \) and \( j_2 \) denote any two columns of \( A \), where \( j_1 < j_2 \). We must show that the four entries \( a[i_1, j_1] \), \( a[i_1, j_2] \), \( a[i_2, j_1] \), and \( a[i_2, j_2] \) satisfy neither of the following two conjunctions:

\[
\begin{align*}
& a[i_1, j_1] < a[i_1, j_2] \quad \text{and} \quad a[i_2, j_1] \geq a[i_2, j_2], \\
& a[i_1, j_1] < a[i_2, j_1] \quad \text{and} \quad a[i_1, j_2] \geq a[i_2, j_2].
\end{align*}
\]

(1) \hspace{10cm} (2)

We consider three possibilities.

If \( i_1 < i_2 \leq j_1 < j_2 < i_1 + n \), then all four entries \( a[i_1, j_1] \), \( a[i_1, j_2] \), \( a[i_2, j_1] \), and \( a[i_2, j_2] \) correspond to distances between pairs of vertices. Moreover, if we let \( J_1 = ((i_1 - 1) \mod n) + 1 \) and \( J_2 = ((j_2 - 1) \mod n) + 1 \), then \( p_{i_1}, p_{i_2}, p_{j_1}, \) and \( p_{j_2} \) are the vertices of a quadrilateral in clockwise order, as suggested in Figure 1(b). Furthermore, \( a[i_1, j_1] \) and \( a[i_2, j_2] \) are the lengths of the quadrilateral's two diagonals, and \( a[i_1, j_2] \) and \( a[i_2, j_1] \) are the lengths of two opposite sides. Now the quadrangle inequality tells us that the sum of the lengths of the diagonals of any quadrilateral is strictly greater than the sum of the lengths of two opposite sides. Thus, \( a[i_1, j_1] + a[i_2, j_2] > a[i_1, j_2] + a[i_2, j_1] \), which implies neither (1) nor (2) holds.

If \( j_1 < i_2 \), then \( a[i_2, j_1] = j_1 - i_2 \). By the definition of \( A \), this observation implies \( a[i_1, j_1] > a[i_2, j_1] \), i.e., (2) does not hold. Furthermore, \( a[i_2, j_1] \geq a[i_2, j_2] \) implies \( [i_2, j_1] \) is negative, which means \( i_2 + n \leq j_2 \). This observation implies \( i_1 + n \leq j_2 \) and \( a[i_2, j_2] > a[i_1, j_2] \), which in turn implies \( a[i_1, j_1] > a[i_1, j_2] \), i.e., (1) does not hold.

Finally, if \( i_1 + n < j_2 \), then \( a[i_1, j_2] = i_1 + n - j_2 - 1 \). By the definition of \( A \), this observation implies \( a[i_2, j_2] > a[i_1, j_2] \), i.e., (2) does not hold. Furthermore, \( a[i_1, j_2] > a[i_1, j_1] \) implies \( a[i_1, j_1] \) is negative, which means \( j_1 < i_1 \). This observation implies \( j_1 < i_2 \) and \( a[i_1, j_1] > a[i_2, j_1] \), which in turn implies \( a[i_2, j_2] > a[i_2, j_1] \), i.e., (1) does not hold.

Totally monotone arrays were introduced in 1987 by Aggarwal et al. [AKM+]. They showed that several problems in computational geometry and VLSI router routing could be reduced to the problem of computing a maximum entry in each row of a totally monotone array. (These entries are referred to as row maxima.) They also gave a sequential algorithm for computing the leftmost maximum in each row of an \( m \times n \) totally monotone array \( A \) in \( O(m + n) \) time.\(^1\)

\[^1\] Aggarwal et al. actually gave slightly tighter bounds for their row-maxima algorithm: they showed that the row maxima of an \( m \times n \) totally monotone array can be computed in \( O(n) \) time when \( m \leq n \) and in \( O(n(1 + \log(m/n))) \) time when \( m > n \). They also proved that these bounds are optimal up to constant factors. However, for the purposes of this paper, the weaker \( O(m + n) \) bound is sufficient.
We refer to this algorithm as the SMAWK algorithm, following the convention of Wilber [W].

Note that the totally monotone arrays mentioned in the previous paragraph are not represented explicitly;\(^2\) rather, we merely assume that any particular entry of \(A\) can be computed in constant time. Also note that the SMAWK algorithm is easily adapted to computing a minimum entry in each row of a totally monotone array \(A\). Conceptually, we need only negate the entries of \(A\) and reverse the ordering of its columns to transform the row-maxima problem for \(A\) to the row-minima problem for \(A\) or vice versa.

Returning to our example of the convex polygon in the plane, the SMAWK algorithm gives us an \(O(n)\)-time algorithm for computing a farthest neighbor for each vertex of a convex \(n\)-gon \(P\). This result follows because a maximum entry in the \(i\)th row of the distance array for \(P\) corresponds to a farthest neighbor of the \(i\)th vertex of \(P\) and because any entry in the distance array for \(P\) can be computed in constant time.

Since the publication of Aggarwal et al.'s seminal paper, a number of additional applications of totally monotone arrays have been discovered. In [AK] Aggarwal and Kravets showed that all farthest neighbors for each vertex of a convex \(n\)-gon can be computed in \(O(n)\) time using the SMAWK algorithm. (In fact, their algorithm is easily modified to compute all maximum entries in each row of an \(m \times n\) totally monotone array \(A\) in \(O(n + m + s)\) time, where \(s\) is the output size, i.e., the total number of row maxima.) In [W] Wilber used the SMAWK algorithm to speed up dynamic programming in the context of the concave least-weight-subsequence problem. Eppstein [Ep] then applied a variant of Wilber's technique to additional dynamic-programming problems involving string editing and the prediction of RNA secondary structure, and Galil and Park [GP], Klawe [Kl], and Larmore and Schieber [LS] independently extended Wilber and Eppstein's work to an even more general dynamic-programming setting. Aggarwal and Park [AP1], [AP2] generalized the notion of a totally monotone array to higher dimensions. They also developed new sequential and parallel algorithms for computing maxima in totally monotone arrays, and applied these algorithms, along with the SMAWK algorithm, to additional problems involving computational geometry, dynamic programming, VLSI river routing, and string editing. Finally, Apostolico et al. [AALM] and Atallah [A] used totally monotone arrays to obtain efficient parallel algorithms for string editing and related problems.

1.2. Our Results

As suggested in the last subsection, previous work relating to totally monotone arrays has been limited to maximization (or minimization) problems. In this paper we consider two more comparison problems—selection and sorting—in the

\(^2\) The row-maxima problem for an \(m \times n\) array \(A\) would be rather trivial if \(A\) were represented explicitly, as simply inputting \(A\) would require \(\Omega(mn)\) time, and \(O(mn)\) is a trivial upper bound on the time necessary for computing \(A\)'s row maxima.
context of totally monotone arrays. Given a set $S$ of $N$ values $a_1, \ldots, a_N$ and an integer $k$ in the range $1 \leq k \leq N$, the selection problem is that of finding $k$ largest values, i.e., a subset $S' \subseteq S$ such that $|S'| = k$ and, for all $a_i \in S'$ and $a_j \in S - S'$, $a_i \geq a_j$. Given a set $S$ of $N$ values $a_1, \ldots, a_N$, the sorting problem is that of finding a permutation $\sigma : \{1, \ldots, N\} \to \{1, \ldots, N\}$ such that $a_{\sigma(1)} \geq a_{\sigma(2)} \geq \cdots \geq a_{\sigma(N)}$.

For arbitrary values $a_1, \ldots, a_N$, the selection and sorting problems are well understood: the general selection problem has time complexity $\Theta(N)$ time [BFP+] and the general sorting problem has time complexity $\Theta(N \lg N)$ time (see [Kn], for example). Using the special structure of totally monotone arrays, we obtain significantly better results for certain selection and sorting problems involving totally monotone arrays than are possible with the classical selection and sorting algorithms. We then apply these results to a number of problems involving convex polygons in the plane.

The remainder of this paper is organized as follows.

In Section 2 we consider the problem of computing $k$ largest entries in each row of a totally monotone array $A$. We call this problem the row-selection problem for $A$. For an $m \times n$ array $A$, we show that the row-selection problem can be solved in $O(k(m + n))$ time. For small values of $k$, this result represents a significant improvement over the naive $O(mn)$-time algorithm obtained by applying the linear-time selection algorithm of [BFP+] $m$ times.\(^3\) Note that computing $k$ smallest entries in each row of $A$ is no harder than computing $k$ largest entries in each row, just as computing a minimum entry in each row is no harder than computing a maximum entry in each row; we need only negate the entries of $A$ and reverse the ordering of its columns to convert back and forth between the two problems.

As an application of our row-selection algorithm, we show how $k$ farthest or $k$ nearest neighbors for each vertex of a planar convex $n$-gon can be computed in $O(kn)$ time. Previous results along these lines include an $O(n^{9/5} \lg n)$-time algorithm suggested by Chazelle [C2], an $O(kn^{3/2} \lg n)$-time algorithm based on Edelsbrunner’s $k$th-order Voronoi diagram algorithm [Ed], and an $O(k^2n + n \lg n)$-time algorithm based on the $k$th-order Voronoi diagram algorithm of Aggarwal et al. [AGSS]. All three of these algorithms can compute either $k$ farthest or $k$ nearest neighbors for every point from a set of $n$ arbitrary points in the plane; thus, they are more general than our algorithm, which works only for the vertices of a convex polygon. However, for the special case of $n$ points that are the vertices of a convex polygon in clockwise order, our algorithm’s running time is asymptotically the best when $k = o(n^{4/5} \lg n)$.

In Section 3 we consider the problem of computing $k$ largest entries overall in an array $A$ such that both $A$ and its transpose are totally monotone. We call this problem the array-selection problem for $A$. For an $m \times n$ array $A$, we show that the array-selection problem can be solved in $O(m + n + k \lg(mn/k))$ time. For small

\(^3\) Subsequent to the submission of an earlier version of this paper to the 1st Annual ACM–SIAM Symposium on Discrete Algorithms, Mansour et al. [MSS] obtained an $O(m^2n)$-time algorithm for the row-selection problem, where $\alpha = \log_2(63/8) \approx 0.9924$. However, for small values of $k$, our algorithm remains the best known.
values of $k$, this result again represents a significant improvement over the naive $O(mn)$-time algorithm obtained by applying the linear-time selection algorithm of [BFP$^+$] directly. (Moreover, for all $k$ in the range $1 \leq k \leq mn$, our algorithm is at least as fast as the linear-time selection algorithm.) Again note that computing $k$ smallest entries overall in $A$ is no harder than computing $k$ largest entries overall.

As an application of our array-selection algorithm, we show how $k$ farthest or $k$ nearest pairs of vertices from a planar convex $n$-gon can be computed in $O(n + k \log(n^2/k))$ time. As far as previous results are concerned, all three of the algorithms mentioned above in the context of computing $k$ farthest or $k$ nearest neighbors—the $O(n^{9/5} \log n)$-time algorithm suggested by Chazelle [C2], the $O(kn^{3/2} \log n)$-time algorithm based on Edelsbrunner’s $k$th-order Voronoi diagram algorithm [Ed], and the $O(k^2n + n \log n)$-time algorithm based on the $k$th-order Voronoi diagram algorithm of Aggarwal et al. [AGSS]—can be used to compute $k$ farthest or $k$ nearest pairs of points from a set of $n$ arbitrary points in the plane. In particular, Chazelle’s approach yields an $O(n^{9/5} \log n)$-time algorithm for computing $k$ farthest or $k$ nearest pairs. Again all three approaches are more general than ours, in that they work for arbitrary point sets. However, for the special case of $n$ points that are the vertices of a convex polygon in clockwise order, our algorithm’s running time is asymptotically the best when $k = o(n^{4/5})$.

In Section 4 we consider the problem of sorting the rows of a totally monotone array $A$. We call this problem the row-sorting problem for $A$. For an $m \times n$ array $A$, we show that the row-sorting problem can be solved in $O(mn)$ time when $m \geq n$ and in $O(mn(1 + \log(n/m)))$ time when $m < n$. This result represents an improvement over the naive $O(mn \log n)$-time algorithm obtained by applying a general sorting algorithm to each row of $A$.

As an application of our row-sorting algorithm, we show that, given a convex $n$-gon $P$ in the plane, for all vertices $v$ of $P$, we can sort the other vertices by distance from $v$ in $O(n^2)$ time. We then generalize this algorithm to $l$ polygons with a total of $n$ vertices, showing that, for each vertex $v$, we can sort the other vertices by distance from $v$ in $O(n^2 \log l)$ time. The $l = 1$ result allows us to identify all triples of vertices from a convex polygon that form isosceles triangles in $O(n^2)$ time. This result provides a partial answer to an open question posed by Guibas [G].

In Section 5 we consider the problem of sorting all the entries of a totally monotone array $A$. We call this the array-sorting problem for $A$. For an $m \times n$ array $A$, we show that the array-sorting problem requires $\Omega(mn \log m)$ comparisons and thus $\Omega(mn \log m)$ time. (If both $A$ and its transpose are totally monotone, then we show an only slightly weaker $\Omega(mn \log t)$-time lower bound, where $t = \min\{m, n\}$.) Thus, for $m = \Theta(n)$, the total monotonicity of $A$ does not make sorting the entries of $A$ any easier than sorting $mn$ arbitrary values. Note that this lower bound implies that there is no straightforward way of using totally monotone arrays to sort in $o(n^2 \log n)$ time the $\binom{n}{2}$ Euclidean distances separating $n$ points in the plane, even if the points are the vertices of a convex polygon in clockwise order. This problem remains open. (If the $L_1$ metric is used in place of the $L_2$
metric, then only $O(n^2)$ pairs of distances need be compared in solving this problem [F], [L]. However, no $o(n^2 \log n)$-time algorithm is known for the problem, either.

Finally, in Section 6, we present some open problems.

In the following discussion we assume that all the entries in our totally monotone arrays are distinct, so that we can refer to the $k$ smallest entries in a row or an array without ambiguity. We make this assumption merely to simplify our presentation; all the algorithms and analyses presented in this paper are easily modified to handle equalities.

2. Row Selection

2.1. A Row-Selection Algorithm

In this subsection we describe an algorithm that, given an $m \times n$ totally monotone array $A = \{a[i, j]\}$ and an integer $k$ in the range $1 \leq k \leq n$, computes the $k$ largest entries in each row of $A$ in $O(k(m + n))$ time. The algorithm combines two previous results with a simple property of totally monotone arrays to achieve the specified time bounds. The first of these previous results is the SMAWK algorithm, described in the introduction. The second is the selection algorithm of Frederickson and Johnson [FJ], that computes, as a special case, the $k$ largest elements overall in $O(k)$ sorted lists in $O(k)$ time. The property of totally monotone arrays linking these two algorithms is given in the following lemma.

Lemma 2. Let $B = \{b[i, j]\}$ denote an $m \times n$ totally monotone array, where $m \geq n$. If each column of $B$ contains at least one row maximum, then each row of $B$ is bitonic, i.e., for $1 \leq i \leq m$,

$$b[i, 1] < \cdots < b[i, c(i) - 1] < b[i, c(i)]$$

and

$$b[i, c(i)] > b[i, c(i) + 1] > \cdots > b[i, n],$$

where $c(i)$ denotes the column containing the maximum entry in row $i$.

Proof. Suppose each column of $B$ contains at least one row maximum, but $B$ is not bitonic. Since $B$ is not bitonic, there exist indices $i$, $j_1$, and $j_2$ such that $1 \leq i \leq m$, $1 \leq j_1 < j_2 \leq n$, and either

1. $j_1 < j_2 < c(i)$ and $b[i, j_1] > b[i, j_2]$, or
2. $c(i) < j_1 < j_2$ and $b[i, j_1] < b[i, j_2]$.

We consider only the first possibility; the proof for the second possibility is analogous. Since each column of $B$ contains at least one row maximum, there exists an $i'$ such that $c(i') = j_2$. We must have $i' < i$, since total monotonicity implies monotonicity. Now consider the $2 \times 2$ subarray of $B$ corresponding to rows $i'$ and $i$ and columns $j_1$ and $j_2$. (This subarray is depicted in Figure 2.) Since $b[i', c(i')]$
Fig. 2. If the maximum entry in row $i'$ lies in column $j_2$, then, by the total monotonicity of $B$, we cannot have $b[i, j_1] > b[i, j_2]$.

is the maximum entry in row $i'$, we have $b[i', j_1] < b[i', c(i')]$. By assumption (1), $b[i, j_1] > b[i, c(i')]$. These two inequalities together contradict our assumption that $B$ is totally monotone.

We now describe our algorithm for computing the $k$ largest entries in each row of $A$. The algorithm has two parts. First, we decompose $A$ into a series of $m$-row subarrays $B_1, \ldots, B_k$. The first subarray $B_1$ consists of those columns of $A$ that contain row maxima of $A$. If we let $A_1$ denote the $m$-row subarray of $A$ consisting of those columns of $A$ not in $B_1$, then $B_2$ consists of those columns of $A_1$ that contain row maxima of $A_1$. In general, if we let $A_{i-1}$ denote the $m$-row subarray of $A$ consisting of those columns of $A$ not in any of $B_1, \ldots, B_{i-1}$, then $B_i$ consists of those columns of $A_{i-1}$ that contain row maxima of $A_{i-1}$. Using the SMAWK algorithm, we can compute $B_1, \ldots, B_k$ (or, more precisely, the columns forming these arrays) in $O(k(m + n))$ total time.

For $1 \leq l \leq k$, it is clear from the definition of $B_l$ that each column of $B_l$ must contain at least one row maximum; thus, by Lemma 2, the rows of $B_l$ are bitonic. Furthermore, if an entry is among the $k$ largest entries in some row of $A$, then the entry must be contained in one of $B_1, \ldots, B_k$. Thus, to compute the $k$ largest entries in row $i$ of $A$, we merely need to compute the $k$ largest elements in the $2k$ sorted lists associated with row $i$. (Each $B_l$ contributes two sorted lists, the first consisting of those entries in the $i$th row of $B_l$ to the right of the $i$th row's maximum and the second consisting of those entries to the maximum's left.) This task can be accomplished in $O(k)$ time using the selection algorithm given by Frederickson and Johnson in [FJ]. Since $A$ contains $m$ rows, the total time for this second part of the algorithm is $O(km)$, which gives the entire row-selection algorithm a running time of $O(k(m + n))$.

Note that our algorithm does not output the $k$ largest entries in a row of $A$ in sorted order, as the algorithm of [FJ] does not provide its output in sorted order. Also note that the size of our algorithm's output, $km$, is not necessarily a lower bound on the time required for the row-selection problem; there may be a more concise representation for the output, given the highly structured nature of totally monotone arrays. Finally, note that our row-selection algorithm can also be used
to find the \( k \) smallest entries in each row of a totally monotone array; as suggested in the introduction, we merely negate each entry of the array and reverse the ordering of its columns before applying the algorithm.

2.2. Applications of Row Selection

Using the row-selection algorithm of the previous subsection, we can solve two selection problems involving convex polygons in the plane. Given a set \( S = \{p_1, \ldots, p_n\} \) of \( n \) points in the plane and an integer \( k \) in the range \( 1 \leq k \leq n \), the \( k \)-farthest-neighbors problem for \( S \) is that of computing the \( k \) farthest neighbors for each point \( p_i \). More precisely, for all \( i \) in the range \( 1 \leq i \leq n \), we must find a subset \( S_i \subseteq S \) such that \( |S_i| = k \) and, for all \( q \in S_i \) and \( q' \in S - S_i, d(p_i, q) \geq d(p_i, q') \).

The \( k \)-nearest-neighbors problem for \( S \) is defined analogously. If the points \( p_1, \ldots, p_n \) are the vertices of a convex \( n \)-gon in clockwise order, then using our algorithm for computing the \( k \) largest entries in each row of a totally monotone array, we can obtain efficient algorithms for both the \( k \)-farthest-neighbors problem and the \( k \)-nearest-neighbors problem.

To reduce the \( k \)-farthest-neighbors problem for \( p_1, \ldots, p_n \) to a row-selection problem, we use the \( n \times (2n - 1) \) totally monotone distance array \( A \) defined in Section 1. As the \( n \) largest entries in row \( i \) of \( A \) are the \( n \) distances \( d(p_i, p_1), d(p_i, p_2), \ldots, d(p_i, p_n) \), we can use our row-selection algorithm to solve the \( k \)-farthest-neighbors problem for \( p_1, \ldots, p_n \) in \( O(nk) \) time.

To solve the \( k \)-nearest-neighbors problem for \( p_1, \ldots, p_n \), we would like to use the distance array \( A \) again; however, to compute the \( k \) nearest neighbors of \( p_i \), we need the \( n - 1 + k \) smallest entries in row \( i \), since then \( n - 1 \) smallest entries in this row are negative integers that do not correspond to distances. For \( 1 \leq k \leq \lfloor n/2 \rfloor \), our upper bound on the time to compute the \( n - 1 + k \) smallest entries in \( A \) is \( O(n^2) \). Thus, to obtain an \( O(kn) \)-time algorithm for the \( k \)-nearest-neighbors problem, we need a slightly more complicated reduction. (Note that we cannot circumvent this difficulty by replacing the negative integers in \( A \) with large positive integers, as these new entries would destroy the total monotonicity of \( A \).

In [LP] Lee and Preparata consider the nearest-neighbor problem (the \( k = 1 \) special case of the \( k \)-nearest-neighbors problem) for the vertices of a convex \( n \)-gon. In obtaining an \( O(n) \)-time solution for this problem, they introduce an interesting property of certain convex polygons which they call the semicircle property. A convex polygon \( P \) with vertices \( p_1, \ldots, p_n \) in clockwise order is said to possess the semicircle property if \( p_2, \ldots, p_{n-1} \) lie inside the circle with diameter \( p_1p_n \).

Lemma 3 [LP]. Let \( P \) denote a convex polygon with vertices \( p_1, \ldots, p_n \) in clockwise order. If \( P \) satisfies the semicircle property, then, for all \( i \) in the range \( 1 \leq i \leq n \), the sequence of distances \( d(p_i, p_1), d(p_i, p_2), \ldots, d(p_i, p_n) \) in bitonic, i.e.,

\[
d(p_i, p_1) > d(p_i, p_2) > \cdots > d(p_i, p_{i-1})
\]

and

\[
d(p_i, p_{i+1}) < \cdots < d(p_i, p_{n-1}) < d(p_i, p_n).
\]
Fig. 3. \( Q_1, Q_2, Q_3, \) and \( Q_4 \) have the semicircle property.

Lee and Preparata also showed how to decompose an arbitrary convex \( n \)-gon into four convex polygons possessing the semicircle property. We use a slightly simpler decomposition, due to Yang and Lee [YL]:

**Lemma 4 [YL].** Let \( p_L \) and \( p_R \) denote vertices of \( P \) whose \( x \)-coordinates are minimum and maximum, respectively, and let \( p_B \) and \( p_T \) denote vertices of \( P \) whose \( y \)-coordinates are minimum and maximum, respectively. Let \( Q_1 \) denote the polygon formed by vertices \( p_T \) through \( p_R \) (i.e., \( p_T, p_R \), and those vertices between \( p_T \) and \( p_R \) in the clockwise ordering of \( P \)'s vertices). Similarly, let \( Q_2, Q_3, \) and \( Q_4 \) denote the polygons formed by vertices \( p_R \) through \( p_B, p_B \) through \( p_L \), and \( p_L \) through \( p_T \), respectively, as shown in Figure 3. \( Q_1, Q_2, Q_3, \) and \( Q_4 \) possess the semicircle property.

Using this decomposition of \( P \) (which is easily computed in linear time), we can compute the \( k \) nearest neighbors of each vertex of \( P \). We restrict our attention to the vertices of \( Q_1 \), showing that their \( k \) nearest neighbors in \( P \) can be computed in \( O(kn) \) time; the computation of the \( k \) nearest neighbors of the vertices of \( Q_2, Q_3, \) and \( Q_4 \) is analogous. For each \( v \) in \( Q_1 \), the \( k \) nearest neighbors of \( v \) in \( Q_1 \) can be computed in \( O(k) \) time, since, by the semicircle property, these \( k \) nearest neighbors must be within \( k \) of \( v \) in the original ordering of \( P \)'s vertices. We can also compute for each \( v \) in \( Q_1 \) its \( k \) nearest neighbors in \( Q_2 \). Consider the \( |Q_1| \times (|Q_2| - 1) \) array \( A = \{a[i,j]\} \) where \( a[i,j] \) is the distance from the \( i \)-th vertex of \( Q_1 \) to the \((j-1)\)-st vertex of \( Q_2 \). (We ignore the first vertex of \( Q_2 \) since it is also the last vertex of \( Q_1 \).) It is readily verified that \( A \) is totally monotone; moreover, the \( k \) smallest entries in row \( i \) of \( A \) correspond to the \( k \) nearest neighbors in \( Q_2 \) of the \( i \)-th vertex of \( Q_1 \). Thus, using our row-selection algorithm, we can find the \( k \) nearest neighbors in \( Q_2 \) of all the vertices in \( Q_1 \) in \( O(kn) \) total time. In a similar manner, we can compute for each \( v \) in \( Q_1 \) its \( k \) nearest neighbors in \( Q_3 \) and its \( k \) nearest neighbors in \( Q_4 \). We now have \( 4k \) neighbors for each \( v \) in \( Q_1 \); using the linear-time selection algorithm of [BFP⁺], we can select the \( k \) nearest of these
neighbors in \( O(k) \) additional time. This last computation gives the \( k \) nearest neighbors in \( P \) of each \( v \) in \( Q_1 \); moreover, the total time spent in computing these neighbors is \( O(kn) \).

3. **Array Selection**

3.1. **An Array-Selection Algorithm**

In this subsection we describe an algorithm that, given an \( m \times n \) array \( A = \{a[i,j]\} \) such that both \( A \) and its transpose \( A^T \) are totally monotone and an integer \( k \) in the range \( 1 \leq k \leq mn \), computes the \( k \) largest entries overall in \( A \) in \( O(m + n + k \lg(mn/k)) \) time. We first present an algorithm for those values of \( k \) that are greater than or equal to both \( m \) and \( n \) and then show how to modify this algorithm to handle smaller values of \( k \).

To compute the \( k \) largest entries of \( A \) when \( \max\{m, n\} \leq k \leq mn \), we begin by checking the relative magnitudes of \( k \) and \( mn \). If \( k \geq mn/2 \) (the "easy" case), we use the linear-time selection algorithm of [BFP⁺] to compute the \( k \) largest entries of \( A \) in \( O(k) \) time. If, on the other hand, \( k < mn/2 \), we consider two subcases.

If \( m \geq n \) we use the row-selection algorithm of Section 2 to compute the \([2k/m]\) largest entries in each row of \( A \) in \( O([2k/m](n + m)) = O(k) \) time. Let \( b_i \) denote the \([2k/m]\)th largest entry in row \( i \) of \( A \). Using the linear-time selection algorithm, we can compute the \([m/2]\)th largest of \( b_1, \ldots, b_m \) in \( O(m) \) time. Let \( b^* \) denote this \([m/2]\)th largest \( b_i \), and let \( B \) denote the \([m/2] \times n \) subarray of \( A \) consisting of those rows \( i \) such that \( b_i \geq b^* \). Furthermore, let \( L \) denote the list of \([2k/m][m/2] = O(k) \) entries formed from the \([2k/m]\) largest entries of each row of \( A \) not in \( B \). Now if row \( i \) of \( A \) is not in \( B \), i.e., \( b_i < b^* \), then the \( n - [2k/m] \) smallest entries in row \( i \) are all smaller than \( b^* \), which means they are all smaller than the \([2k/m]\) largest entries in each row of \( B \). Since \( B \) has \([m/2]\) rows, this observation implies the \( n - [2k/m] \) smallest entries in row \( i \) are all smaller than at least \([2k/m][m/2] \geq k \) other entries, i.e., these entries need not be considered as candidates for the \( k \) largest entries overall of \( A \). Thus, if we recursively compute the \( k \) largest entries in \( B \) and then use the linear-time selection algorithm to compute in \( O(k) \) time the \( k \) largest of these entries and the \( O(k) \) entries of \( L \), we obtain the \( k \) largest entries in \( A \).

If \( m < n \) we apply the procedure described in the last paragraph to \( A^T \) rather than \( A \). This computation requires \( O(k) \) time plus the time needed to compute recursively the \( k \) largest entries in an \( m \times [n/2] \) subarray of \( A \).

Letting \( T(k, m, n) \) denote our algorithm's running time in computing the \( k \) largest entries in an \( m \times n \) array \( A \), where \( \max\{m, n\} \leq k \leq mn \) and both \( A \) and \( A^T \) are totally monotone, we have

\[
T(k, m, n) = \begin{cases} 
O(k) & \text{if } k \geq mn/2, \\
T(k, [m/2], n) + O(k) & \text{if } k < mn/2 \text{ and } m \geq n, \\
T(k, m, [n/2]) + O(k) & \text{if } k < mn/2 \text{ and } m < n.
\end{cases}
\]

The solution to this recurrence is

\[
T(k, m, n) = O(k \lg(mn/k)).
\]
Now suppose \( k < m \). We can eliminate all but \( k \) of \( A \)'s rows from consideration as follows. In \( O(n + m) \) time we can compute the row maxima of \( A \). Then, using the linear-time selection algorithm, we can select the \( k \) largest of these maxima in an additional \( O(m) \) time. Now consider the \( m - k \) rows of \( A \) corresponding to the \( m - k \) smallest row maxima. The entries in these rows are all smaller than the \( k \) largest row maxima, which means they are not among the \( k \) largest entries of \( A \). Thus, we can eliminate these \( m - k \) rows from consideration. Similarly, if \( k < n \), we can eliminate all but \( k \) of \( A \)'s columns in \( O(n + m) \) time.

Once the number of rows in \( A \) has been reduced to \( k \) or less and the number of columns in \( A \) has been reduced to \( k \) or less, we can apply our \( O(k \lg(mn/k)) \)-time selection algorithm for arrays with \( m \leq k \) rows and \( n \leq k \) columns. This observation gives an algorithm for computing the \( k \) largest entries in \( A \) that works for all values of \( k \) in the range \( 1 \leq k \leq mn \) and runs in \( O(m + n + k \lg(st/k)) \) time, where \( s = \min\{m, k\} \) and \( t = \min\{n, k\} \).

We can simplify the above expression for our algorithm's running time by observing that \( m + n + k \lg(st/k) = \Theta(m + n + k \lg(mn/k)) \) for all \( m, n, \) and \( k \) such that \( 1 \leq k \leq mn \). To see why this claim holds, first note that \( m + n + k \lg(st/k) = O(m + n + k \lg(mn/k)) \), since \( s \leq m \) and \( t \leq n \). Now suppose \( m + n + k \lg(st/k) = o(m + n + k \lg(mn/k)) \). This assumption implies

\[
\lg(st/k) = o(\lg(mn/k))
\]

and

\[
m + n = o(k \lg(mn/k)).
\]

Clearly, (3) implies \( k \) is smaller than at least one of \( m \) and \( n \). Thus, if we assume without loss of generality that \( m \leq n \), only two possibilities need be considered: \( m < k \leq n \) and \( k \leq m \leq n \).

If \( m < k \leq n \), then \( s = m \) and \( t = k \). Equation (3) then implies \( \lg m = o(\lg(mn/k)) \), which implies \( \lg m = o(\lg(n/k)) \). This last relation implies \( k \lg(mn/k) = \Theta(k \lg(n/k)) \). Since \( k \lg(n/k) \leq n \), we then have \( k \lg(mn/k) = O(n) \), which contradicts (4).

If \( k \leq m \leq n \), then \( s = k \) and \( t = k \). Equation (3) then implies \( \lg k = o(\lg(mn/k)) \), which implies \( \lg k = o(\lg(mn)) \). This last relation implies \( k \) is less than any polynomial in \( mn \). Thus, \( k \lg(mn/k) \) is also less than any polynomial in \( mn \). In particular, \( k \lg(mn/k) = o(\sqrt{mn}) \), which again contradicts (4).

Note that the only lower bound we have on the time required for the array-selection problem is \( \Omega(n) \). Also note that our array-selection algorithm can also be used to compute the \( k \) smallest entries overall in a totally monotone array whose transpose is also totally monotone, just as our row-selection algorithm can be used to compute the \( k \) smallest entries in each row of a totally monotone array.

3.2. Applications of Array Selection

Using the array-selection algorithm of the previous subsection, we can solve two more selection problems involving convex polygons in the plane. Given a set
$S = \{p_1, \ldots, p_n\}$ of $n$ points in the plane and an integer $k$ in the range

$$1 \leq k \leq \binom{n}{2},$$

the $k$-farthest-pairs problem for $S$ is that of computing the $k$ largest values of $d(p_i, p_j)$ over all unordered pairs $(p_i, p_j)$ of points. The $k$-nearest-pairs problem for $S$ is defined analogously. If the points $p_1, \ldots, p_n$ are the vertices of a convex $n$-gon in clockwise order, then using our algorithm for computing the $k$ largest entries overall in a totally monotone array, we can obtain efficient algorithms for both the $k$-farthest-pairs problem and the $k$-nearest-pairs problem.

To reduce the $k$-farthest-pairs problem for $p_1, \ldots, p_n$ to a row-selection problem, we use a subarray of the $n \times (2n - 1)$ distance array $A$ defined in Section 1. Specifically, we use the subarray corresponding to all $n$ rows of $A$ and its first $n$ columns. Since the $\binom{n}{2}$ largest entries overall in this subarray are the $\binom{n}{2}$ distances corresponding to all unordered pairs of vertices, and since both the subarray and its transpose are totally monotone (because $A$ and its transpose are totally monotone), we can use our array-selection algorithm to solve the $k$-farthest-pairs problem for $p_1, \ldots, p_n$ in $O(n + k \log(n^2/k))$ time.

Similarly, to solve the $k$-nearest-pairs problem for $p_1, \ldots, p_n$, we use nearly the same reduction that we used for the $k$-nearest-neighbors problem, except that here we must again ensure that, for all unordered pairs $(p_i, p_j)$ of points, only one of $d(p_i, p_j)$ and $d(p_j, p_i)$ is among the distances we consider. Applying our array-selection algorithm then allows us to solve the $k$-nearest-pairs problem for $p_1, \ldots, p_n$ in $O(n + k \log(n^2/k))$ time.

4. Row Sorting

4.1. A Row-Sorting Algorithm

In this subsection we present an algorithm for sorting the rows of an $m \times n$ totally monotone array $A = \{a[i, j]\}$ in $O(mn)$ time when $m \geq n$ and in $O(mn(1 + \log(n/m)))$ time when $m < n$. We begin by describing a more basic $O(mn + n^2)$-time algorithm for the row-sorting problem and then show how this second algorithm's running time can be reduced to $O(mn(1 + \log(n/m)))$ when $m > n$.

For $1 \leq i \leq m$ and $1 \leq r \leq n$, let $c_r(i)$ denote the column of $A$ containing the entry in row $i$ of $A$ with rank $r$ in row $i$, where the rank of entry $a[i, j]$ in row $i$ is defined to be the number of entries in row $i$ that are greater than or equal to $a[i, j]$. In other words, $c_r(i)$ is the unique column of $A$ such that

$$\{|j: 1 \leq j \leq n, a[i, j] \geq a[i, c_r(i)]\} = r$$

or

$$\{|j: 1 \leq j \leq c_r(i), a[i, j] \geq a[i, c_r(i)]\} + \{|j: c_r(i) < j \leq n, a[i, j] > a[i, c_r(i)]\} = r.$$
if we remove our assumption that all the entries of \( A \) are distinct). Furthermore, for \( 1 \leq r \leq n \), let \( c_r(0) = r \). (These values may be interpreted as describing an imaginary row \( 0 \) of \( A \) such that \( a[0, 1] > a[0, 2] > \cdots > a[0, n] \); note that the addition of such a row does not affect the total monotonicity of \( A \).

Our basic algorithm consists of \( m \) phases, where, in the \( i \)th phase, we sort row \( i \) of \( A \) by computing \( c_1(i), c_2(i), \ldots, c_n(i) \) using \( c_1(i - 1), c_2(i - 1), \ldots, c_n(i - 1) \). Specifically, we use a simple insertion sort (such as the one described in [Kn]) to sort row \( i \), inserting first \( a[i, c_1(i - 1)] \), then \( a[i, c_2(i - 1)] \), then \( a[i, c_3(i - 1)] \), and so on through \( a[i, c_n(i - 1)] \). To insert a particular entry \( a[i, j] \) in the sorted list of previously inserted entries from row \( i \), we first compare \( a[i, j] \) with the smallest previously inserted entry, then with the second smallest, then with the third smallest, and so on, until an entry larger than \( a[i, j] \) is found and \( a[i, j] \)'s place in the sorted list of previously inserted entries thereby ascertained.

Clearly, the order in which we insert the entries of row \( i \) affects the time spent sorting these entries. In particular, as noted in [Kn], sorting \( N \) values with the insertion sort described above takes \( \Theta(N + I) \) time, where \( I \) is the number of inversions separating the insertion order and the final sorted order for the values. An inversion is a pair of values \((x, y)\) such that \( x \) is inserted before \( y \) but \( x < y \).

In the worst case, a sequence of \( N \) values may contain

\[
\binom{N}{2} = \Omega(N^2)
\]

inversions; however, we argue that the total number of inversions encountered in sorting all \( m \) rows of an \( m \times n \) totally monotone array is \( O(n^2) \).

Given the order in which we insert the entries of row \( i \), an inversion encountered while sorting row \( i \) corresponds to a pair of columns \( j_1 \) and \( j_2 \), such that \( a[i - 1, j_1] > a[i - 1, j_2] \) and \( a[i, j_1] < a[i, j_2] \) (where, by convention, \( a[0, j_1] > a[0, j_2] \) if and only if \( j_1 < j_2 \)). Because \( A \) is totally monotone, for each pair of columns \( j_1 \) and \( j_2 \) there exists at most one row index \( i \) such that \( a[i - 1, j_1] > a[i - 1, j_2] \) and \( a[i, j_1] < a[i, j_2] \). (In fact, such a row index exists only if \( j_1 < j_2 \).) Thus, in sorting all the rows of \( A \), we can encounter at most \( O(n^2) \) total inversions, one for each pair of columns. This bound gives our basic algorithm a running time of \( O(mn + n^2) \).

To obtain an algorithm that runs in \( O(mn) \) time when \( m \geq n \) and in \( O(mn(1 + \log(n/m))) \) time when \( m < n \), we first note that, if \( m \geq n \), the basic algorithm described above already has the desired running time. On the other hand, if \( m < n \), we need to modify the basic algorithm as follows. First, we partition \( A \) into \( \lceil n/m \rceil \) subarrays of size at most \( m \times m \). Then, using our basic algorithm, we sort the rows of these subarrays in \( O(m^2) \) time per subarray or \( O(mn) \) total time. Finally, we merge the \( \lceil n/m \rceil \) sorted subrows corresponding to each row of \( A \) in \( O(n(1 + \log(n/m))) \) time per row or \( O(mn(1 + \log(n/m))) \) total time.

Note that the size of our algorithm's output, \( mn \), is not necessarily a lower bound on the time required for the row-sorting problem; there may be a more concise representation for the output, given the highly structured nature of totally monotone arrays.
4.2. Applications of Row Sorting

As an application of our row-sorting algorithm, we consider the neighbor-ranking problem: given a set $S = \{p_1, \ldots, p_n\}$ of $n$ points in the plane, for each $p_i$, sort the other vertices of $S$ by distance from $p_i$.

If $p_1, \ldots, p_n$ are the vertices of a convex polygon $P$ in clockwise order, then we can solve the neighbor-ranking problem for $P$ using the $n \times (2n - 1)$ totally monotone distance array $A = \{a[i,j]\}$ defined in Section 1. Specifically, the $i$th row of $A$ contains the distances $d(p_i, p_1), \ldots, d(p_i, p_n)$, along with $n - 1$ negative entries; thus, sorting the rows of $A$ using our row-sorting algorithm gives an $O(n^2)$-time solution to the neighbor-ranking problem for $P$.

If $p_1, \ldots, p_n$ are arbitrary points in the plane, the preceding approach clearly does not work. However, we can generalize the approach to a related neighbor-ranking problem, which allows us to obtain an algorithm for the general neighbor-ranking problem that is superior to the naive $O(n^2 \lg n)$-time algorithm obtained by applying any $O(n \lg n)$-time general sorting algorithm $n$ times.

Specifically, consider the following variant of the neighbor-ranking problem: given a convex $m$-gon $P$ with vertices $p_1, \ldots, p_m$ in clockwise order and a set $S$ of $n$ arbitrary points $q_1, \ldots, q_n$, for each $p_i$, sort the points of $S$ by distance from $p_i$. We call this problem the problem of sorting $S$ with respect to $P$. We cannot represent the distances separating the vertices of $P$ and the points of $S$ as a totally monotone array. However, the correctness of our row-sorting algorithm does not depend on total monotonicity; total monotonicity is merely used to bound the number of inversions. In fact, even though the $m \times n$ array $B = \{b[i,j]\}$ where $b[i,j] = d(p_i, q_j)$ is not totally monotone, we can still apply our row-sorting algorithm.

In the context of the array $B$, an inversion corresponds to indices $i, j_1$, and $j_2$ such that $b[i - 1, j_1] < b[i - 1, j_2]$ but $b[i, j_1] > b[i, j_2]$. To bound the number of times such an inversion can occur for any particular pair of indices $j_1$ and $j_2$, note that the perpendicular bisector of $q_{j_1}$ and $q_{j_2}$ can intersect $P$ at most twice, as shown in Figure 4. Since there is an inversion for $j_1$ and $j_2$ between rows $i - 1$

![Fig. 4. The perpendicular bisector of any pair of points $q_{j_1}$ and $q_{j_2}$ can intersect $P$ at most twice.](image)
and $i$ if and only if the perpendicular bisector of $q_{j_1}$ and $q_{j_2}$ intersects the edge of $P$ connecting $p_{i-1}$ and $p_i$, there are at most two inversions associated with each pair $(j_1, j_2)$. Since there are $\binom{n}{2}$ pairs of points in $S$, the total number of inversions is no more than

$$2\binom{n}{2} = O(n^2).$$

Thus, we can sort the rows of $B$ (i.e., sort $S$ with respect to $P$) in $O(mn + n^2)$ time using our basic row-sorting algorithm.

Now consider a set $S = \{p_1, \ldots, p_n\}$ of $n$ arbitrary points in the plane. To solve the neighbor-ranking problem for $S$, we need a partition of the points of $S$ into subsets, such that the points of the $i$th subset are the vertices of a convex polygon $P_i$ in clockwise order. (Such a partition could be obtained by computing the convex layers of $S$, which requires only $O(n \log n)$ time [C1].) Assuming the points of $S$ are partitioned into $l$ such subsets, we can solve the neighbor-ranking problem for $S$ in $O(n^2 \log l)$ time as follows. First, we choose a second partition of $p_1, \ldots, p_n$, this time into $l$ arbitrary subsets $S_1, \ldots, S_l$, each of size $n/l$. Then, for $1 \leq i \leq l$ and $1 \leq j \leq l$, we sort $S_i$ with respect to $P_j$ in $O(n_j n/l + n^2/l^2)$ time, where $n_j$ is the size of $P_j$. The total time required is

$$\sum_{i=1}^{l} \sum_{j=1}^{l} O\left(n_j \frac{n}{l} + \frac{n^2}{l^2}\right) = \sum_{i=1}^{l} O\left(\frac{n^2}{l}\right) = O(n^2).$$

For each point $p_i$ of $S$, we now have $l$ sorted lists, corresponding to the points of $S_1, \ldots, S_l$, respectively. Merging the $l$ lists for point $p_i$ takes $O(n \log l)$ time, since these lists contain a total of $n$ elements. Thus, we can merge all the lists in $O(n^2 \log l)$ total time, which gives an $O(n^2 \log l)$-time algorithm solving the neighbor-ranking problem for $S$.

As a final application of our row-sorting algorithm, suppose we are given a set $S$ of $n$ arbitrary points $p_1, \ldots, p_n$ and that we want to find all triples $(p_i, p_j, p_k)$ such that $p_i$, $p_j$, and $p_k$ form an isosceles triangle. Equivalently, we want to find for each $p_i$ all pairs $(p_j, p_k)$ such that $d(p_i, p_j) = d(p_i, p_k)$. If, for each $p_i$, we have the other points sorted by distance from $p_i$, then a simple linear scan of the sorted list for $p_i$ gives us all the pairs of points that are equidistant from $p_i$. Thus, given a partition of $p_1, \ldots, p_n$ into $l$ subsets, such that the points of the $i$th subset are the vertices of a convex polygon $P_i$ in clockwise order, we can find all of the isosceles triangles formed by points of $S$ in $O(n^2 \log l)$ time. In particular, if the points of $S$ are the vertices of a single convex $n$-gon in clockwise order, then we can find all of the isosceles triangles formed by points of $S$ in $O(n^2)$ time.

5. Array Sorting

As a final variation on our paper’s theme, we consider the problem of sorting all the entries of an $m \times n$ totally monotone array. A primary motivation for considering this problem is the distance-sorting problem from computational
geometry: given \( n \) points \( p_1, \ldots, p_n \) in the plane, sort the \( \binom{n}{2} \) distances \( d(p_i, p_j) \), corresponding to all pairs \((p_i, p_j)\) of points.

If distance is measured in terms of the \( L_1 \) metric (i.e., for \( p_i = (x_i, y_i) \) and \( p_j = (x_j, y_j) \), \( d(p_i, p_j) = |x_i - x_j| + |y_i - y_j| \)), then only \( O(n^2) \) pairs of distances need be compared in solving the distance-sorting problem. This result follows because this distance-sorting problem is easily reduced to the problem of sorting \( n^2 \) values of the form \( X_i + Y_j \), where \( X = \{X_i\} \) and \( Y = \{Y_j\} \) are arbitrary \( n \)-vectors. (This problem is known as the problem of sorting \( X + Y \).) In [F] Fredman showed that, for all \( n \), there exists a comparison tree of depth \( O(n^2) \) sorting \( X + Y \) when \( X \) and \( Y \) are \( n \)-vectors, and, more recently, Lambert [L] gave an explicit construction for such comparison trees. However, Lambert’s sorting algorithm is not efficient, in the sense that the time required by his algorithm for operations other than comparisons is quite large, and it remains open whether either the problem of sorting \( X + Y \) or the \( L_1 \)-metric distance-sorting problem can be solved in \( o(n^2 \lg n) \) time.

If, on the other hand, distance is measured in terms of the \( L_2 \) (Euclidean) metric, then no algorithm is known that compares \( o(n^2 \lg n) \) pairs of distances in solving the distance-sorting problem. For the special case of this problem where \( p_1, \ldots, p_n \) are the vertices of a convex polygon in clockwise order, the distance array defined in Section 1 gives a reduction to the array-sorting problem for a totally monotone array. Unfortunately, unlike the three array problems we considered in Sections 2–4, the array-sorting problem is not significantly easier than the general problem of sorting \( mn \) arbitrary values, which takes \( O(mn \lg mn) \) time. Specifically, we can show a simple \( \Omega(mn \lg m) \)-comparison lower bound on sorting the rows of a totally monotone array, and an \( \Omega(mn \lg t) \)-comparison lower bound on sorting the rows of a totally monotone array whose transpose is also totally monotone, where \( t = \min\{m, n\} \). These lower bounds imply that the aforementioned distance array reduction does not help us in sorting the \( \binom{n}{2} \) Euclidean distances associated with the vertices of a convex \( n \)-gon.

We prove the first lower bound by bounding the number of one-to-one functions \( \sigma \) mapping pairs \((i, j)\), \( 1 \leq i \leq m \) and \( 1 \leq j \leq n \), to ranks \( r \), \( 1 \leq r \leq mn \), such that the \( m \times n \) array \( A = \{a[i, j]\} \) where \( a[i, j] = \sigma(i, j) \) is totally monotone. To do this, we consider all one-to-one functions \( \sigma \) such that

\[
\sigma(i, 1) < \sigma(i, 2) < \cdots < \sigma(i, n)
\]

for all \( i \) in the range \( 1 \leq i \leq m \). There are

\[
\frac{(mn)!}{(n!)^m} \geq (m!)^n
\]

such functions. Moreover, each of these functions gives a totally monotone array \( A \), since the array’s rows are sorted. This observation implies there are at least \( (m!)^n \) one-to-one functions corresponding to totally monotone arrays, which implies \( \lg(m!)^n = \Omega(mn \lg m) \) comparisons are necessary (in the worst case) to solve the array-sorting problem for such arrays.
For the second lower bound, we again consider one-to-one functions \( \sigma \) mapping pairs \((i, j), 1 \leq i \leq m \) and \(1 \leq j \leq n\), to ranks \(r, 1 \leq r \leq mn\). Specifically, we bound the number of \( \sigma \) such that both the \( m \times n \) array \( A = \{a[i, j]\} \) defined by \( a[i, j] = \sigma(i, j) \) and its transpose \( A^T \) are totally monotone. To do this we consider all one-to-one functions \( \sigma \) such that, for all rows \( i \) and \( i' \) and all columns \( j \) and \( j' \), \( i + j < i' + j' \) implies \( \sigma(i, j) < \sigma(i', j') \). There are

\[
(t!)^{m+n-2t+1}((t-1)!)^2((t-2)!)^2 \cdots ((1)!)^2
\]

such functions, where \( t = \min\{m, n\} \), since the relative sizes of \( \sigma(i, j) \) and \( \sigma(i', j') \) are unrestricted if \( i + j = i' + j' \), which means (loosely speaking) that we can freely permute the diagonals of \( A \). Moreover, each of these functions gives a totally monotone array \( A \) whose transpose is also totally monotone, since the array's rows and columns are sorted. This observation implies there are at least

\[
(t!)^{m+n-2t+1}((t-1)!)^2((t-2)!)^2 \cdots ((1)!)^2
\]

one-to-one functions corresponding to totally monotone arrays whose transposes are also totally monotone, which implies

\[
\lg((t!)^{m+n-2t+1}((t-1)!)^2((t-2)!)^2 \cdots ((1)!)^2) = \Omega(mn \lg t)
\]

comparisons are necessary (in the worst case) to solve the array-sorting problem for such arrays.

6. Conclusion

In this paper we explored two fundamental comparison problems—selection and sorting—in the context of totally monotone arrays. We provided simple but efficient algorithms for two selection problems and a sorting problem involving totally monotone arrays, algorithms that take advantage of an array's total monotonicity to obtain significantly better results than are possible for arbitrary arrays. We also presented several applications of these algorithms to problems in computational geometry.

We conclude with a few of the more interesting questions left unresolved by this paper:

1. In Section 3 we gave an algorithm for computing the \( k \) largest entries overall in an array \( A \) such that both \( A \) and its transpose are totally monotone. It remains open whether a comparable result can be obtained for totally monotone arrays whose transposes are not totally monotone.

2. The only array problem considered in this paper for which we obtain matching upper and lower bounds is the array-sorting problem discussed in Section 5. (The bounds for array sorting are matching when \( m = \Theta(n) \).) It remains open whether the algorithms for row selection, array selection, and row sorting given in Sections 2–4 can be improved or nontrivial lower bounds for these problems obtained. (Lower bounds might follow from the sizes of the various problems' search spaces; for example, a lower bound
of $\Omega(S)$ on the number of combinations of row permutations possible for a totally monotone array would imply an $\Omega(\log S)$ lower bound on the time necessary to sort the array’s rows in a linear decision tree model.

3. In Subsection 4.2 we applied our algorithm for sorting the rows of a totally monotone array to the neighbor-ranking problem for the vertices of a convex polygon $P$. We then extended this technique to arbitrary point sets. It remains open whether our two selection algorithms for totally monotone arrays, which we also applied to the vertices of a convex polygon, can likewise be applied to arbitrary point sets.

Acknowledgments

The authors are grateful to Alok Aggarwal, who first suggested we consider selection and sorting in totally monotone arrays, brought [C2] and [F] to our attention, and made several helpful comments on an early draft of this paper. The authors also thank Michelangelo Grigni for bringing [FJ] to our attention and the anonymous referee who reviewed this paper for Mathematical Systems Theory for suggesting several improvements to our original submission.

References


Received June 6, 1990, and in revised form March 1, 1991.