Heapsort—Adapted for Presorted Files

Christos Levcopoulos and Ola Petersson
Department of Computer and Information Science*
Linköping University
S–581 83 Linköping, Sweden

Abstract

We provide a new sorting algorithm which is optimal with respect to several known, and new, measures of presortedness. A new such measure, called $Osc(X)$, measures the oscillation within the input data. The measure has an interesting application in the sweep-line technique in computational geometry. Our algorithm is based on a new approach which yields space efficiency and it uses simple data structures. For example, after a linear time preprocessing step, the only data structures used are a static tree and a heap.

Key words: sorting algorithm, measures, presortedness, heap, geometric interpretation, optimality.

1 Introduction

A *measure of presortedness* is an integer function on a permutation π of a totally ordered set that reflects how much π differs from the total order. Examples of presortedness measures include the number of runs or inversions. The term presortedness was coined by Mehlhorn [Meh79], who used the number of inversions in the file as a measure. Mehlhorn [Meh84a, Section III.5.3] also gave an algorithm, A-Sort, which is adaptive with respect to this measure. Intuitively, an algorithm is *adaptive* with respect to a measure of presortedness if it sorts all permutations, but performs particularly well on permutations that have a high degree of presortedness. Mannila [Man85] formalized the concept of presortedness and gave algorithms that are adaptive with respect to several measures. Recently, Estivill-Castro and Wood [EW87] and Skiena [Ski88] have considered other measures. Cook and Kim [CK80], and Mannila [Man84] have studied the problem of designing adaptive sorting algorithms empirically.

This paper describes a new measure of presortedness, called $Osc(X)$, motivated by a geometric interpretation of the input permutation. Intuitively, $Osc(X)$ measures how much the input sequence X oscillates. $Osc(X)$ is proved to be a better measure than the number of inversions and runs; even when they are combined. We also give an algorithm,

*Present address: Department of Computer Science, Lund University, Lund, Sweden.

named maxtree-sort, which is optimal with respect to $Osc(X)$. Maxtree-sort operates on a heap; being a variant of heapsort. However, instead of keeping all the elements in the heap, we store only some of them. If $Osc(X)$ is high, the heap will contain many elements most of the time, and maxtree-sort will run in time $\Theta(n \log n)$.[1] On the other hand, if the oscillation is low, there will only be a few elements in the heap when the operations are performed, and the sorting will be completed considerably faster. Maxtree-sort can thus be seen as a variant of heapsort that is adaptive to the oscillation in the input. The algorithm seems to be practical, since the constants in the running time are small, and it is space efficient. There is a close connection between our sorting algorithm and a paradigm known as *plane sweep* in computational geometry.

The remainder of the paper is organized as follows. In Section 2 we define $Osc(X)$. Section 3 provides a lower bound for comparison-based sorting algorithms, with respect to $Osc(X)$. Section 4 contains a description of maxtree-sort, as well as a discussion about its efficiency. In Section 5, $Osc(X)$ is compared to other measures of presortedness; in Section 6, we discuss the results.

2 The Oscillation Measure

Some preliminary definitions are given before defining the new measure of presortedness, $Osc(X)$. Let $X = \langle x_1, \ldots, x_n \rangle$ be a *sequence* of n elements x_i from some totally ordered set; that is, for all $i, j \in \{1, \ldots, n\}$, $x_i \leq x_j$ or $x_j \leq x_i$. For simplicity it can be assumed that the elements are distinct nonnegative integers. For two sequences, $X = \langle x_1, \ldots, x_n \rangle$ and $Y = \langle y_1, \ldots, y_m \rangle$, their *catenation* XY is the sequence $\langle x_1, \ldots, x_n, y_1, \ldots, y_m \rangle$. Further, let $|X|$ denote the *length* of X and $|S|$ the *cardinality* of a set S. If $Z = \langle x_{f(1)}, x_{f(2)}, \ldots, x_{f(m)} \rangle$ and $f\{1, \ldots, m\} \rightarrow \{1, \ldots, n\}$ is injective and monotonically increasing, then Y is called a *subsequence* of X. In particular, Z is a *consecutive subsequence* of X if there exists an i, $1 \leq i \leq n - m + 1$, such that $Z = \langle x_i, \ldots, x_{i+m-1} \rangle$.

Definition 2.1 Let $X = \langle x_1, \ldots, x_n \rangle$ be a sequence. For each $i \in \{1, \ldots, n\}$, let $Cross(x_i) = \{j \mid 1 \leq j < n \text{ and } \min\{x_j, x_{j+1}\} < x_i < \max\{x_j, x_{j+1}\}\}$.

We define the following measure of presortedness.

Definition 2.2 For a sequence X of length n, let $Osc(X) = \sum_{i=1}^{n} |Cross(x_i)|$.

Figure 1 illustrates the geometric motivation behind the definition.

Some basic properties of $Osc(X)$ are:

1. $Osc(X) \geq 0$, with equality if and only if X is in ascending or descending order.
2. If $X = \langle x_1, \ldots, x_n \rangle$ and $Y = \langle y_1, \ldots, y_n \rangle$ and $x_i < x_j$ if and only if $y_i < y_j$, for all $i, j \in \{1, \ldots, n\}$, then $Osc(X) = Osc(Y)$.
3. If Y is a subsequence of X then $Osc(Y) \leq Osc(X)$.

[1] $\log n$ is defined as $\max\{1, \log_2 n\}$.

4. For all $x \in N$, $Osc(\langle x \rangle X) \leq |X| + Osc(X)$.
5. $Osc(X) \leq n^2/2$.
6. $Osc(\langle x_n, x_{n-1}, \ldots, x_1 \rangle) = Osc(X)$.

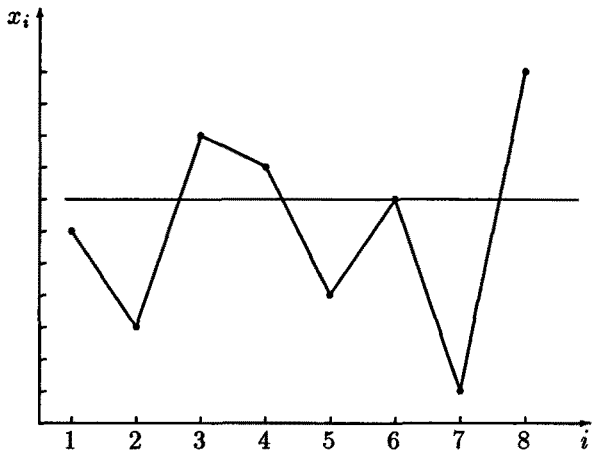

Figure 1. *$Osc(X)$ is the number of proper intersections between horizontal lines through the x_i's and line segments defined by consecutive x_i's. Here, for example, $Cross(x_6) = \{2, 4, 7\}$ and $Osc(X) = 17$.*

3 A Lower Bound

The concept of an optimal algorithm with respect to a measure of presortedness was given in a general form by Mannila [Man85].

Definition 3.1 Let M be a measure of presortedness, and A a sorting algorithm which uses $T_A(X)$ comparisons to sort a sequence X. We say that A is *M-optimal*, or *optimal with respect to M* if, for some $c > 0$, we have, for all $X = \langle x_1, \ldots, x_n \rangle$:

$$T_A(X) \leq c \cdot \max\{|X|, \log(|below(X, M)|)\},$$

where $below(X, M) = \{\pi \mid \pi \text{ is a permutation of } \{1, \ldots, n\} \text{ and } M(\pi(X)) \leq M(X)\}$.

Giving a lower bound for an algorithm with respect to $Osc(X)$ thus means to bound the cardinality of $below(X, Osc)$ from below.

Let X be a sequence of length n. We prove that if $Osc(X) = p$ is not $O(n)$ then $\log(|below(X, Osc)|)$ is $\Omega(n \log(Osc(X)/n))$. (If $p = O(n)$ the lower bound is trivially $\Omega(n)$.) For simplicity, assume that n divides p. We construct the set

$$A_i = \{1 + i \cdot (\frac{p}{n} - 2), 2 + i \cdot (\frac{p}{n} - 2), \ldots, (\frac{p}{n} - 2) + i \cdot (\frac{p}{n} - 2)\},$$

for $0 \leq i \leq \lceil n^2/(p-2n) \rceil - 1$. Let B_i be any permutation of the elements of A_i. We build a sequence Y by the catenation of the B_i:

$$Y = B_0 B_1 \cdots B_{\lceil n^2/(p-2n) \rceil - 1}.$$

It is easily verified that $Osc(Y) \leq p$, and there are $(p/n-2)!^{\lceil n^2/(p-2n) \rceil}$ such permutations Y. Thus, the size of $below(X, Osc)$ is at least $(p/n-2)!^{\lceil n^2/(p-2n) \rceil}$. If we take the logarithm and apply Definition 3.1 we have proved the following theorem.

Theorem 3.1 Any comparison-based sorting algorithm for a sequence X, of length n, needs at least $\Omega(n + n \log(Osc(X)/n))$ comparisons.

4 An Optimal Algorithm

This section describes a sorting algorithm, maxtree-sort, which is adaptive to the oscillation in the input sequence. The algorithm runs in time $\Theta(n \log n)$ in the worst case, but is considerably faster if $Osc(X)$ is low.

Maxtree-sort is a variant of heapsort (see e.g. [Meh84a, Section II.1.2]). Recall that heapsort starts by building a heap consisting of n elements and then repeatedly performs the operation *DeleteMax*, n times. The number of elements in the heap during most *DeleteMax* operations is $\Theta(n)$. Since the time taken by each *DeleteMax* is logarithmic in the size of the heap, the total running time of heapsort is $\Theta(n \log n)$. Maxtree-sort operates on a heap, too. Instead of storing *all* elements in the heap, however, only the ones that can possibly be the maximun of the remaining elements, the *max-candidates*, are stored. Initially, linear preprocessing time is spent to construct a data structure that supports efficient retrieval of new max-candidates, to be inserted into the heap before each *DeleteMax* is performed. This data structure, which is described in Section 4.1, is the maxtree. What follows is a high-level description of *maxtree-sort*.

```
Construct the maxtree corresponding to X;
for i := 1 to n do
  begin
    Retrieve new max-candidates from the maxtree;
    Insert new max-candidates into the heap;
    Perform DeleteMax on the heap;
  end;
```

4.1 The Maxtree Data Structure

The *maxtree* corresponding to a sequence $X = \langle x_1, \ldots, x_n \rangle$ is the binary tree with root $x_i = \max\{x_1, \ldots, x_n\}$. Its left child is the maxtree corresponding to $\langle x_1, \ldots, x_{i-1} \rangle$ and its right child is the maxtree corresponding to $\langle x_{i+1}, \ldots, x_n \rangle$. The maxtree corresponding to the sequence of length zero is the empty binary tree (see Figure 2).

The construction of the maxtree in linear time is not obvious. First we build a level-linked binary search tree, where the leaves are the indices $1, \ldots, n$. An internal node with

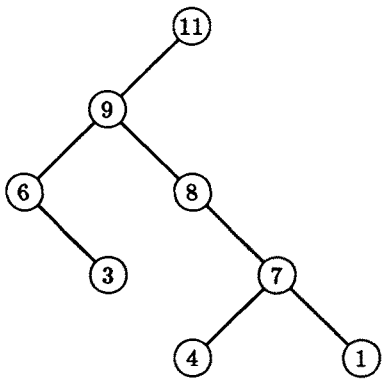

Figure 2. *The maxtree corresponding to* $X = \langle 6,3,9,8,4,7,1,11 \rangle$.

descendants $i, \ldots, i+k$ contains the index of the maximum of the elements $x_i, \ldots, x_{i+k}$ (see Figure 3). As will be discussed later (see Section 4.3) none of the pointers need to be stored explicitely. We also create two pointers, or fingers, pointing to the leftmost and rightmost leaves. Clearly, this tree can be built in linear time, since there are not more than $2n$ nodes, and the index stored in each node can be found by a single comparison of the indices stored in the children.

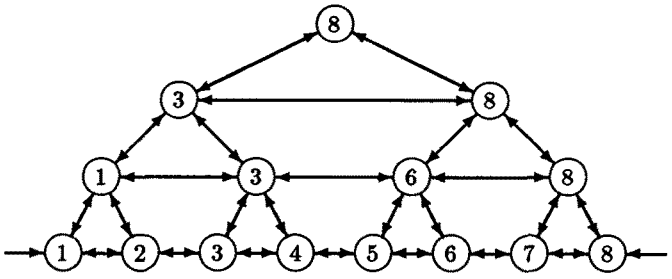

Figure 3. *The level-linked binary search tree corresponding to* $X = \langle 6,3,9,8,4,7,1,11 \rangle$.

Next, we compute two arrays, GB (GreaterBefore) and GA (GreaterAfter). These tell, for each x_i, which are the positions of the closest elements before and after x_i that are greater than x_i. For simplicity, let $x_0 = x_{n+1} = \infty$. More formally, we compute

$$\begin{aligned} GB[i] &= \max\{j \mid 0 \leq j < i \text{ and } x_i < x_j\}, \text{ and} \\ GA[i] &= \min\{j \mid i < j \leq n+1 \text{ and } x_i < x_j\}, \end{aligned}$$

for $i \in \{1, \ldots, n\}$. GB (GA) can be computed in linear time by scanning the sequence backward (forward) once, while maintaining a stack that holds the elements that have not yet found their closest greater predecessor (successor).

Given the arrays, GB and GA, and the binary search tree, the maxtree can be constructed as follows. Search up along the left and right paths of the binary search tree concurrently until we find a subtree guaranteed to contain the index of the maximal element. The element corresponding to the index found is the root of the maxtree. The roots of the left and right subtrees of the maxtree are then found recursively. By consulting GA (GB) for every node we encounter during the upward search on the left (right) path, it takes only constant time to decide whether we have to continue upward or not. If the element searched for is located k positions away from either end, the time taken to find it is $O(\log(\min\{k, n-k\}))$. Since all time spent so far has been linear, the total time taken for constructing the maxtree is thus linear plus the amount given by the recurrence

$$T(n) = \begin{cases} O(1) & \text{if } n = 1 \\ \max\limits_{1 \le k < n} \{T(k) + T(n-k) + O(\log(\min\{k, n-k\}))\} & \text{if } n > 1 \end{cases}$$

which is linear [Meh84a, p.185]. Hence, we have obtained the following

Lemma 4.1 The maxtree corresponding to a sequence $X = \langle x_1, \ldots, x_n \rangle$ can be constructed in linear time.

4.2 Maxtree-sort

As already mentioned, the maxtree is used in maxtree-sort to support efficient retrieval of the new max-candidates. Initially, the only max-candidate is the root of the maxtree, which is inserted into the empty heap. In each of the following iterations, the new max-candidates are the children in the maxtree of the element last deleted from the heap. The correctness of maxtree-sort is fairly straightforward, and is therefore omitted. A proof of its time complexity, however, needs a lemma:

Lemma 4.2 The number of elements in the heap, when x_i is deleted, is not greater than $\lfloor |Cross(x_i)|/2 \rfloor + 2$.

Proof From a geometrical point of view, a horizontal line through x_i divides X into a number of consecutive subsequences. In each subsequence either all elements are smaller than x_i, or all elements are greater than x_i. Denote by a *valley* a consecutive subsequence in which all elements are smaller than x_i. It should be clear that the number of valleys is bounded by $\lfloor |Cross(x_i)|/2 \rfloor + 2$. When x_i is deleted, all other elements in the heap are smaller than x_i, and must therefore come from the valleys. The lemma follows if we can prove that each valley contributes with at most one element to the heap at the moment x_i is deleted.

Suppose x_{j_1} and x_{j_2}, $j_1 < j_2$, are two elements in the heap which originate from the same valley. Let x_{j_0} be the lowest common ancestor of x_{j_1} and x_{j_2} in the maxtree. Since an element is inserted into the heap only when its father (in the maxtree) is deleted from the heap, x_{j_0} must have been deleted earlier, and hence, $x_{j_0} > x_i$. The same argument implies that $j_0 \neq j_1$ and $j_0 \neq j_2$. The construction of the maxtree guarantees, however, that $j_1 < j_0 < j_2$. Therefore x_{j_1} and x_{j_2} cannot belong to the same valley, which contradicts the assumption and concludes the proof. ∎

Lemma 4.3 Maxtree-sort runs in time $O(n + \sum_{x_i} \log(|Cross(x_i)|))$.

Proof In each iteration we perform at most two operations on the maxtree and three on the heap. Given the element last deleted from the heap, the operations on the maxtree can be done in constant time, say c_1. The complexity of the heap operations are logarithmic in the number of elements in the heap, which is not greater than $\lfloor |Cross(x_i)|/2 \rfloor + 2$, by Lemma 4.2. Since the construction of the maxtree was done in linear time, by Lemma 4.1, the time complexity of maxtree-sort is

$$\begin{aligned} T(n) &\leq O(n) + \sum_{x_i} (c_1 + c_2 \cdot \log(\lfloor \frac{|Cross(x_i)|}{2} \rfloor + 2)) \\ &= O(n + \sum_{x_i} \log(|Cross(x_i)|)). \end{aligned}$$

∎

Theorem 4.4 Maxtree-sort is optimal with respect to $Osc(X)$.

Proof By Lemma 4.3,

$$\begin{aligned} T(n) &= O(n + \sum_{x_i} \log(|Cross(x_i)|)) \\ &= O(n + \log(\prod_{i=1}^{n} |Cross(x_i)|)) \\ &= O(n + n \cdot \log(\prod_{i=1}^{n} |Cross(x_i)|)^{1/n}) \\ &\leq O(n + n \cdot \log(\frac{1}{n} \cdot \sum_{i=1}^{n} |Cross(x_i)|)) \\ &= O(n + n \log(\frac{Osc(X)}{n})) \end{aligned}$$

where the inequality follows from the inequality for geometric and arithmetic averages. The optimality follows from Theorem 3.1. ∎

4.3 Implementation Considerations

In this subsection we analyse maxtree-sort's space reqirements, and make some remarks regarding its implementation.

The level-linked binary search tree needs no explicit pointers since it is static. It can be stored in an array in level-order, like a partially ordered tree implemented by a heap. Furthermore, the lowest level can be omitted since we know that it contains the numbers $1 \ldots n$ in sorted order. Hence, n pointers suffice for storing the search tree. The arrays GA and GB need n pointers each, and thus the total amount required so far is $3n$ pointers in addition to the input array.

A node in the maxtree contains the corresponding array element and two children pointers. The heap is just an array of pointers to the nodes in the maxtree. (This

guarantees that given the element last deleted from the heap, its children in the maxtree can be found in constant time, as assumed in the proof of Lemma 4.3.) The amount of space needed during the sorting phase, i.e., after the preprocessing, is thus $3n$ pointers in addition to the input elements. We conclude that maxtree-sort can be implemented by using $3n$ pointers, and that the constants in running time are low since no complicated data structures are used.[2]

It should be noted that we have assumed all elements to be distinct. If multiple copies of the same element are allowed we have to be a bit careful when constructing the maxtree to achieve the same time complexity.

5 A Comparison With Other Measures

We start this section by defining three known measures of presortedness. Then, a comparison is made which proves that $Osc(X)$ is superior.

The two measures that have been studied most are probably the number of *inversions* (see e.g. [GMPR77], [Meh79] and [Man85]), defined by

$$Inv(X) = |\{(i,j) \mid 1 \leq i < j \leq n \text{ and } x_i > x_j\}|,$$

and the number of *runs* (see e.g. [Man85]), defined by

$$Runs(X) = |\{i \mid 1 \leq i < n \text{ and } x_{i+1} < x_i\}| .$$

Yet another approach is to express presortedness in terms of the smallest distance, p, such that each pair of elements, that are at least distance p from each other, are in correct order. We define [EW87]

$$Par(X) = p \text{ if and only if } p = \min\{q \mid X \text{ is } q - \text{sorted}\}$$

where X is $q-$sorted if and only if, for all $i, j \in \{1, \ldots, n\}$, $i - j > q$ implies $x_j \leq x_i$.

A major drawback with all the above measures, when compared to $Osc(X)$, is that a sequence X and the reverse of X yield different measures, which seems a bit strange; especially if we consider a nonincreasing sequence, which intuitively has a lot of order in it. This fact will be exploited in the proof of the next theorem. We prove that, in some sense, $Osc(X)$ is not included in any of the measures defined above.

Theorem 5.1 For each of the measures, $Inv(X)$, $Runs(X)$ and $Par(X)$, there is no $c > 0$ such that $M(X) \leq c \cdot Osc(X)$, for all X.

Proof Let $X = \langle n, n-1, \ldots, 1 \rangle$. Then $Inv(X) = n(n-1)/2$, $Runs(X) = n - 1$ and $Par(X) = n - 1$, while $Osc(X) = 0$. ∎

The next theorem shows that the measures defined above are included in $Osc(X)$. In other words, sequences considered to be presorted according to these measures are also presorted according to $Osc(X)$.

[2] Observe that we have not described how the maxtree can be *built* without wasting space, if we are given the search tree and the two arrays. Doing this requires compacting the arrays and cutting off more levels in the search tree. It results in slightly higher constants in running time during the linear preprocessing phase.

Theorem 5.2 For all sequences X, of length n,

1. $Osc(X) \leq 4 \cdot Inv(X)$,
2. $Osc(X) \leq 2n \cdot Runs(X) + n$,
3. $Osc(X) \leq n \cdot Par(X)$.

Proof

1. We prove that if $j \in Cross(x_i)$, this causes at least one inverted pair, and the same pair is not counted more than four times. It should be clear that if $j \in Cross(x_i)$ then either (i,j), $(i,j+1)$, (j,i), or $(j+1,i)$ is an inversion. (Even though $(j,j+1)$ or $(j+1,j)$ is an inversion we do not count it because it might appear $O(n)$ times.) Assume, without loss of generality, that (i,j) is the inversion. Since we do not count inversions caused by j's only, the only other possibilities for this inversion to be counted is if $j-1 \in Cross(x_i)$, $i \in Cross(x_j)$, or $i-1 \in Cross(x_j)$. Hence, $Inv(X) \geq Osc(X)/4$.

2. Clearly, for all i, $Cross(x_i)$ cannot contain more than two elements from each run (maximal ascending consecutive subsequence). Furthermore, the run to which x_i belongs cannot contribute with more than one element to $Cross(x_i)$. Hence,

$$Osc(X) = \sum_{i=1}^{n} |Cross(x_i)| \leq \sum_{i=1}^{n} (2 \cdot Runs(X) + 1) = 2n \cdot Runs(X) + n.$$

3. Choose i', such that $|Cross(x_{i'})| \geq |Cross(x_i)|$, for all $i \in \{1, \ldots, n\}$. We distinguish three cases:

   Case 1. If $|Cross(x_{i'})| = 0$, then X is sorted in ascending or descending order, and $Par(X) = 0$ or $Par(X) = n-1$.

   Case 2. If $|Cross(x_{i'})| = 1$, then there is at least one pair of consecutive elements that are out of order and hence $Par(X) \geq 1$.

   Case 3. Otherwise, if $|Cross(x_{i'})| > 1$, we let $j_{min} = \min\{j \mid j \in Cross(x_{i'})\}$ and $j_{max} = \max\{j \mid j \in Cross(x_{i'})\}$. By the definition of $Cross(x_{i'})$,

$$\min\{x_{j_{max}}, x_{j_{max}+1}\} < x_{i'} < \max\{x_{j_{min}}, x_{j_{min}+1}\}.$$

   The definition of $Par(X)$ gives, $Par(X) > j_{max} - (j_{min}+1)$. Obviously, $|Cross(x_{i'})| \leq j_{max} - (j_{min}+1)$.

   In all cases we have that $Par(X) \geq |Cross(x_{i'})|$, and thus

$$Osc(X) \leq n \cdot |Cross(x_{i'})| \leq n \cdot Par(X).$$

■

Theorem 5.2 becomes interesting if we consider the lower bounds for the measures $Inv(X)$, $Runs(X)$ and $Par(X)$, which are $\Omega(n + n\log(Inv(X)/n))$ [GMPR77], $\Omega(n + n\log(Runs(X)))$ [Man85] and $\Omega(n + n\log(Par(X)))$ [EW87], respectively. By combining Theorem 3.1 and Theorem 5.2 we can conclude

Corollary 5.3 Any *Osc*-optimal sorting algorithm is also optimal with respect to $Inv(X)$, $Runs(X)$ and $Par(X)$.

In particular, by Theorem 4.4,

Corollary 5.4 Maxtree-sort is optimal with respect to the measures $Osc(X)$, $Inv(X)$, $Runs(X)$, and $Par(X)$.

Observe that by Theorem 5.1, there are sequences which are sorted fast by an *Osc*-optimal algorithm, while an algorithm which is optimal with respect to $Inv(X)$, $Runs(X)$, and $Par(X)$ might need $\Theta(n \log n)$ time.

6 Concluding Remarks

In this paper we introduced a new measure of presortedness. The measure $Osc(X)$ measures how much the input sequence oscillates when interpreted as a polygonal chain in the plane. In addition, a sorting algorithm maxtree-sort was presented.

Maxtree-sort is a variant of heapsort which is optimal with respect to $Osc(X)$. As opposed to heapsort, maxtree-sort adapts to the oscillation within the input by keeping just the necessary elements in the heap, instead of all of them. Maxtree-sort is space efficient and uses simple data structures.

The oscillation measure was proved superior to other known measures of presortedness. In particular, any *Osc*-optimal algorithm (for example maxtree-sort) is also optimal with respect to the measures $Inv(X)$, $Runs(X)$, and $Par(X)$.

Maxtree-sort has an interesting application in the sweep-line technique in computational geometry [Meh84b], [PS85], [Ede87], [ORo87]. The sweep-line technique is, for example, often used when decomposing polygons into other polygons which satisfy certain conditions, such as being triangular or monotone. Such algorithms contain a preprocessing step in which the vertices of the polygon are sorted according to one of its coordinates. Then, an orthogonal line is swept over the polygon, starting at the first vertex (according to the sorted order). During the sweep, information relevant for solving the problem is updated at each new vertex encountered. The time complexity of the update depends on the amount of data in the data structure keeping the information. This amount, in turn, is usually proportional to the number of polygonal edges currently crossed by the sweepline. If the data structure supports logarithmic time updating, the time consumed by the sweep will be the same as that of maxtree-sort when applied to sort the coordinates, by Lemma 4.3. Hence, maxtree-sort is a good candidate in the preprocessing step for the sweeping, and it can be viewed as adaptive to the oscillation.

It should be noted that the oscillation is highly depending on the orientation of the polygon in the plane; a slight rotation might increase the oscillation from $\Theta(n)$ to $\Theta(n^2)$. An interesting question is, therefore, whether there is a fast algorithm for determining an orientation for which the oscillation approximates the minimum oscillation.

References

[CK80] C.R Cook and D.J. Kim. Best sorting algorithms for nearly sorted lists. *Communications of the ACM*, 23(11):620–624, 1980.

[Ede87] H. Edelsbrunner. *Algorithms in Combinatorial Geometry.* Springer-Verlag, Berlin/Heidelberg, F.R.Germany, 1987.

[EW87] V. Estivill-Castro and D. Wood. *A new measure of presortedness.* Research Report CS–87–58, University of Waterloo, Department of Computer Science, Waterloo, Canada, 1987.

[GMPR77] L.J. Guibas, E.M. McCreight, M.F. Plass, and J.R. Roberts. A new representation of linear lists. In *Proc. 9th Annual ACM Symposium on Theory of Computing*, pages 49–60, 1977.

[Man84] H Mannila. *Implementation of a sorting algorithm suitable for presorted files.* Technical Report, Department of Computer Science, University of Helsinki, Finland, 1984.

[Man85] H. Mannila. Measures of presortedness and optimal sorting algorithms. *IEEE Transactions on Computers*, C-34(4):318–325, 1985.

[Meh79] K. Mehlhorn. Sorting presorted files. In *Proc. 4th GI Conference on Theoretical Computer Science*, pages 199–212, Springer-Verlag, 1979.

[Meh84a] K. Mehlhorn. *Data Structures and Algorithms, Vol 1: Sorting and Searching.* Springer-Verlag, Berlin/Heidelberg, F.R.Germany, 1984.

[Meh84b] K. Mehlhorn. *Data Structures and Algorithms, Vol. 3: Multidimensional Searching and Computational Geometry.* Springer-Verlag, Berlin/Heidelberg, F.R.Germany, 1984.

[ORo87] J. O'Rourke. *Art Gallery Theorems and Algorithms.* Oxford University Press, New York/Oxford, 1987.

[PS85] F.P. Preparata and M.I. Shamos. *Computational Geometry: An Introduction.* Springer-Verlag, New York, N.Y., 1985.

[Ski88] S.S. Skiena. Encroaching lists as a measure of presortedness. *BIT*, 28:775–784, 1988.