# Measures of Presortedness and Optimal Sorting Algorithms

HEIKKI MANNILA

*Abstract* — **The concept of presortedness and its use in sorting are studied. Natural ways to measure presortedness are given and some general properties necessary for a measure are proposed. A concept of a sorting algorithm optimal with respect to a measure of presortedness is defined, and examples of such algorithms are given. A new insertion sort algorithm is shown to be optimal with respect to three natural measures. The problem of finding an optimal algorithm for an arbitrary measure is studied, and partial results are proven.**

*Index Terms* — **Local insertion sort, measures, optimality, presortedness, sorting.**

## I. Introduction

THE question of identifying in some sense "easy" cases of a computational problem and utilizing this easiness has intrinsic interest. In sorting, easiness can be identified with existing order. Indeed, when discussing sorting, it is customary to note that the input can be almost in order or at least have some existing order (see, e.g., [16, p. 339], [22, p. 126], [6, p. 223], and [14, p. 165]).

In this paper we study the use of presortedness in sorting. We do this by trying to answer three questions.

1) How can the existing order (*presortedness*) of a sequence be measured?

2) What does it mean that an algorithm utilizes the presortedness of input (measured in some way)?

3) Do there exist algorithms utilizing presortedness (in the sense of the answers to the previous questions)?

Our questions can be seen as special cases of a more general problem: How can the structure of the input be used in sorting? For structure different from presortedness, this problem has been analyzed in, e.g., [9], [10], and [13].

We start in Section II by discussing the first question. We present four natural ways to measure presortedness, review briefly their properties, and then give general conditions which any measure of presortedness should satisfy.

In Section III we tackle the second question. We give a definition of an $m$-optimal algorithm where $m$ is a measure of presortedness. The definition is similar to the optimality criteria used in [21] and [12] in the case of a particular measure of presortedness. We give two justifications for the definition: one information-theoretic, and the other based on the behavior of $m$-optimal algorithms under various probability distributions.

Section IV moves to the third question and gives examples of $m$-optimal algorithms for various natural choices of the measure $m$. In particular, we are able to exhibit an algorithm which is optimal with respect to three natural measures, is intuitively simple, and has a reasonably straightforward implementation.

Section V studies the existence of optimal algorithms for arbitrary measures. While we cannot exhibit concrete algorithms, we can still show that for any measure $m$ there exists an almost $m$-optimal sequence of comparison trees, i.e., we are able to give nonuniform algorithms. If the measure is computable in linear time, then we get truly optimal trees. The result is based on a search technique of Fredman [11].

In Section VI we discuss the results and outline some possible extensions.

If $A$ is a set, then $|A|$ denotes its cardinality; for a sequence $X$ the notation $|X|$ means the length of $X$. The notation log means logarithm in base 2.

## II. Measures of Presortedness

### A. *Natural Measures*

Let $X = \langle x_1, \cdots, x_n \rangle$ be the input to be sorted. For simplicity, we assume that the $x_i$'s are distinct integers. We consider ascending order to be the correct one.

*Inversions:* Let

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

be the number of pairs in the wrong order (*inversions*). Then $0 \le \operatorname{inv}(X) \le n(n-1)/2$ for all $X$, the smallest value occurring in the case of an already sorted list, and the largest value in the case of a list in descending order. Inv($X$) indicates how many exchanges of adjacent elements are needed to sort $X$ (in, e.g., bubblesort).

This quantity has been extensively studied (see [16, sec. 5.1.1]). It has been used as a measure of presortedness in [21] and [12]. The drawback with this measure is that inputs of the type

$$(n+1, n+2, n+3, \cdots, 2n, 1, 2, 3, \cdots, n)$$

have a quadratic number of inversions, even though such sequences are intuitively almost in order and are also easy to sort using merging. The next measure handles this case very well.

Manuscript received August 14, 1984; revised October 24, 1984. A preliminary version of this paper was presented at the Eleventh International Colloquium for Automata, Languages and Programming, Antwerp, Belgium, July 1984.

The author is with the Department of Computer Science, University of Helsinki, Helsinki, Finland.

*Runs:* Define

$$\text{runs}(X) = |\{i \mid 1 \leq i < n \text{ and } a_{i+1} < a_i\}| + 1 .$$

Then runs($X$) is the number of ascending substrings of $X$, i.e., the number of runs at the beginning of the so-called natural mergesort ([16, p. 161]). For an already sorted list we have runs($X$) = 1, and for a sequence in reverse order runs($X$) = $n$.

Also, this quantity belongs to the family of well-studied properties of permutations ([16, sec. 5.1.3]). It has even been used to measure the randomness of a pseudorandom sequence ([17, sec. 3.3.2]).

This measure is quite natural: a small number of ascending runs clearly indicates a high degree of presortedness. The drawback with this measure is opposite to the one of inversions: local disorder, as in $\langle 2, 1, 4, 3, \cdots, n, n-1\rangle$, produces a lot of runs, even though the sequence is intuitively quite ordered and is certainly easy to sort using, e.g., bubblesort.

The measure could be modified to allow ascending or descending runs (so-called long runs [16, ex. 5.1.3-23]) since from the point of view of merge sorting it seems to make no difference if the runs are either ascending or descending. In Section II-B we will see that this modification does not, however, fully qualify as a measure of presortedness, as it ignores the reversal operations needed.

*Longest Ascending Subsequence:* Define

$$\begin{aligned}\text{las}(X) = \max\{t \mid\ & \exists\, i(1), \cdots, i(t) \text{ such that} \\ & 1 \leq i(1) < \cdots < i(t) \leq n \text{ and} \\ & x_{i(1)} < \cdots < x_{i(t)}\} .\end{aligned}$$

Thus, las($X$) is the length of the longest ascending subsequence of $X$; therefore, $1 \leq \text{las}(X) \leq n$. The direction of growth of this quantity is opposite to the previous ones. Therefore, we rather use the quantity rem($X$) = $n$ − las($X$), which indicates how many elements have to be removed from $X$ to leave a sorted list. So rem($X$) = 0, if $X$ is sorted, and rem($X$) attains its maximum value $n - 1$ for a list in reverse order. This quantity can be defined operationally as the least number of data movements needed to sort $X$.

As a measure of presortedness rem is a little less intuitive than the previous two. A long ascending run guarantees a quite low value of rem; so does little local disorder. Thus, rem seems to be to a certain degree immune to the deficiencies of the previous measures.

The number rem($X$) has been used as a measure of presortedness in an empirical study investigating the applicability of usual sorting algorithms for nearly sorted lists [3]. Algorithms for calculating rem($X$) or las($X$) have been given in [5] and [4].

*Number of Exchanges:* Define

$$\begin{aligned}\text{exc}(X) = \ & \text{the smallest number of exchanges of} \\ & \text{arbitrary elements needed to bring } X \\ & \text{into ascending order.}\end{aligned}$$

Obviously, exc($X$) = 0 for a sorted list $X$, and $\text{exc}(X) \leq \text{inv}(X)$ for all $X$. To find the maximum value of exc, we use the following result:

$$\begin{aligned}\text{exc}(X) = n - \ & \text{the number of cycles in the permutation} \\ & \text{of } \{1, \cdots, n\} \text{ corresponding to } X .\end{aligned}$$

(See [16, ex. 5.2.2-2].) As every permutation has at least one cycle, we have $\text{exc}(X) \leq n - 1$ for all $X$. An $X$ realizing this upper bound is $\langle n, 1, 2, \cdots, n-1\rangle$. If $X$ is in reverse order, then exc($X$) = $\lfloor n/2 \rfloor$.

This measure indicates how many exchange operations we have to perform in a sorting algorithm based on solely this method of ordering the input. Actually finding sorting methods realizing this bound and using comparisons efficiently seems to be quite difficult.

The operational definition of exc is simple and corresponds to some intuitive idea of presortedness. On the other hand, the largest value of exc is obtained, e.g., for a permutation which is almost in order according to the three previous measures.

*Remarks:* Comparing these measures is by no means easy. Inv and runs are quite well known and their behavior is easy to grasp: inv measures global presortedness, runs local presortedness. Rem seems to combine these properties somewhat, while exc seems to differ drastically from the others, in spite of its natural definition. The differences show that presortedness can be measured in many ways.

## B. General Properties of Measures

In this section we list some general conditions which a measure of presortedness should satisfy.

We first fix the functionality of measures. A measure $m$ is a function $N^{<N} \rightarrow N$ where $N^{<N}$ denotes the set of all finite sequences of (distinct) integers; thus, we consider only integer-valued measures.

The first two conditions state that a sorted list has zero disorder, and that the value of a measure $m$ depends only on the order of its argument:

1) $m(X) = 0$, if $X$ is in ascending order;

2) if $X = \langle x_1, \cdots, x_n\rangle$, $Y = \langle y_1, \cdots, y_n\rangle$ and $x_i < x_j$ if and only if $y_i < y_j$ for all $i$ and $j$, then $m(X) = m(Y)$.

The second condition concentrates our attention on comparison-based algorithms and leaves the methods using other properties of keys [7], [8] outside the scope of this paper.

Conditions 1) and 2) do not restrict the allowed measures much. The following stronger conditions are, in our opinion, necessary for a measure:

3) if $X$ is a subsequence of $Y$, then $m(X) \leq m(Y)$;

4) if $X < Y$, i.e., every element of $X$ is smaller than every element of $Y$, then $m(XY) \leq m(X) + m(Y)$;

5) for all $a$ we have $m(\langle a\rangle X) \leq |X| + m(X)$.

Next we outline why these conditions are necessary for a measure of presortedness.

Two approaches to the concept of presortedness or disorder are:

a) disorder is quantified by the number of operations of a given type which is needed to order the input (concrete approach);

b) disorder is quantified by how much information of the form $x_i < x_j$ is needed to identify the sequence, using a given way of collecting the information (information-theoretic approach).

Both these approaches lead to measures of disorder satisfying properties 3), 4), and 5). For approach a), we have to assume that the set of allowed operations is closed under subsequences. If we can sort a supersequence of $X$ by a certain number of operations, then the restrictions of these operations to $X$ will sort $X$; if $X < Y$, then $XY$ can be sorted by first sorting $X$ and then sorting $Y$, in a total of $m(X) + m(Y)$ operations; and sorting $\langle a \rangle X$ can be done by sorting $X$ in $m(X)$ operations and inserting $a$ into its place, for which there are $|X|$ possibilities different from the original position of $a$. For approach b), if we have enough information to identify a supersequence of $X$, then the same information identifies $X$; if $X < Y$, then $XY$ is identified by identifying $X$ and $Y$; and $\langle a \rangle X$ is identified by identifying $X$ and then locating the place of $a$.

The measures presented in Section II-A are easily seen to satisfy properties 1)–5), once they have been scaled to have value 0 for a sorted list. Examining the definitions of inv, runs, rem, and exc we see that they in fact satisfy the conditions for approach a) needed in the above reasoning. Counting long runs violates property 4), as is seen by considering the sequences $X = \langle 2, 1 \rangle$ and $Y = \langle 4, 3 \rangle$. Both have one long run, i.e., zero disorder; we also have $X < Y$, but $XY$ does not have zero disorder. Counting long runs does not satisfy the assumptions of approach a) either. A descending run like $\langle 4, 3 \rangle$ needs a reversal operation to be in order, but the measure does not take this into account.

## III. The Concept of an Optimal Algorithm

We turn now to our second question: What does it mean that an algorithm utilizes presortedness?

Let $m$ be a measure of presortedness and $z$ an integer. Define

$$\text{below}'(z, X, m) = \{Y \mid Y \text{ is a permutation of } \{1, \cdots, |X|\} \text{ and } m(Y) \leq z\}$$

where $m(X) \leq z$, and

$$\text{below}(X, m) = \{Y \mid Y \text{ is a permutation of } \{1, \cdots, |X|\} \text{ and } m(Y) \leq m(X)\}.$$

These sets are nonempty for all $X$ and $m$, as the permutation of $\{1, \cdots, |X|\}$ order-isomorphic to $X$ belongs to them.

Consider algorithms having as input not only the sequence $X$ but also an upper bound $z$ on the quantity $m(X)$. Let $z$ be given, and analyze the comparison tree (see [16, pp. 182–183]) for input length $X$ of such an algorithm, restricted to the cases where $m(X) \leq z$. This subtree must have at least $|\text{below}'(z, X, m)|$ leaves, as the number of possible inputs is at least this; so the height of the tree is at least $\log|\text{below}'(z, X, m)|$. Therefore, the algorithm takes at least $\log|\text{below}'(z, X, m)|$ time. Thus, we could require that an algorithm utilizing presortedness in the sense of $m$ would work in time $O(\log|\text{below}'(z, X, m)|)$. But $\text{below}'(z, X, m)$ can be a very small set, even a singleton, and this would lead to a sublinear time requirement. However, it seems reasonable to give at least linear time to any sorting algorithm, although some highly constrained sorting problems can indeed be solved in a sublinear number of comparisons [9].

So we have the following definition. An algorithm $S$, which uses $T_S(X, z)$ steps on inputs $X$ and $z$, is *weakly m-optimal,* if for some $c > 0$ and for all $X$ and $z$ we have

$$T_S(X, z) \leq c \cdot \max\{|X|, \log|\text{below}'(z, X, m)|\}.$$

What if an upper bound $z$ on $m(X)$ is not available? We strengthen the definition by assuming that it is always given as the best possible one, i.e., $z = m(X)$ always, with the following definition as a result.

Let $m$ be a measure of presortedness, and $S$ a sorting algorithm, which uses $T_S(X)$ steps on input $X$. We say that $S$ is *m-optimal* (or optimal with respect to $m$), if for some $c > 0$ we have for all $X$

$$T_S(X) \leq c \cdot \max\{|X|, \log|\text{below}(X, m)|\}.$$

This is the definition of optimality from [21] and [12] generalized to arbitrary measures.

As trivial examples of optimal algorithms consider the measures $m_0$ and $m_{01}$ defined by

$$m_0(X) = 0, \quad \text{for all } X;$$
$$m_{01}(X) = 0, \quad \text{if } X \text{ is sorted};$$
$$m_{01}(X) = 1, \quad \text{otherwise}.$$

These functions satisfy the conditions 1)–5) imposed on measures. Any sorting algorithm having $O(n \log n)$ worst case (e.g., heapsort) is $m_0$-optimal, and an algorithm which first checks whether the input is in order and uses heapsort only if it is not is $m_{01}$-optimal.

The definition of optimality can be related to probability distributions on the input. Let $p(X)$ be the probability that an input of length $|X|$ is $X$; then $\Sigma\{p(X) \mid |X| = n\} = 1$ for all $n$. Consider the comparison tree of an algorithm for input length $n$. Labeling the left branch of each comparison node with 0 and the right branch with 1, the tree gives a binary code for the set of permutations. The codeword for permutation $q$ is obtained by traversing the path from the root to the leaf corresponding to $q$. From the noiseless coding theorem of Shannon (see [1, p. 36]) we know that the average code length for the set of permutations is at least

$$\sum_{|X|=n} p(X) \cdot \log(1/p(X)).$$

Thus, no sorting method can have a better average case number of comparisons than this.

By our definition the average time used by an $m$-optimal algorithm for distribution $p$ is at most

$$c \sum p(X) \cdot \max\{|X|, \log|\text{below}(X, m)|\}.$$

So if

$$\log(1/p(X)) \geq \max\{|X|, \log|\mathrm{below}(X,m)|\}$$

for all $X$, then an $m$-optimal algorithm comes within a constant factor of $c$ from the theoretical minimum. This condition simplifies to

$$p(X) \leq \min\{2^{-|X|}, |\mathrm{below}(X,m)|^{-1}\}.$$

This restricts the allowed distributions in two ways: no input can have a very large weight, and more importantly, inputs $X$ with large below$(X, m)$ sets must have small weights.

An alternative to our definition of $m$-optimality would be to consider sets

$$\mathrm{equal}(X,m) = \{Y \mid Y \text{ is a permutation of } \{1,\cdots,|X|\} \text{ and } m(Y) = m(X)\}$$

instead of the below-sets. This would use the classifying power of the measure $m$ fully. The modified definition would, however, require that every sequence with a small equal-set be sorted fast, no matter how disordered the sequence is. The below-sets take the presortedness and disorder aspect into account: i.e., the sequence with the maximal number of inversions is not required to be processed fast, although its equal-set consists of just one element.

## IV. Examples of Optimal Algorithms

We start this section by showing that natural mergesort is optimal with respect to the number of runs. Next we review an insertion sort algorithm given by Mehlhorn [21], which is optimal with respect to the number of inversions. We then show that by a suitable choice of data structure a modified insertion sort can be made to be simultaneously optimal with respect to inv, runs, and rem. The algorithm can be implemented using, e.g., level-linked 2–3 or 2–4 trees [2], [15].

Generally, proofs of $m$-optimality of an algorithm $S$ consist of two parts: calculating the running time of $S$ for sequences $Y$ with $m(Y) \leq z$, and estimating the size of the corresponding below-set.

Natural mergesort (see [16, p. 161]) is a sorting method which starts from the runs present in the input and as a first phase merges them pairwise. Pairwise merging of the resulting longer runs is continued until there is only one run.

*Theorem 1:* Natural mergesort is optimal with respect to the number of runs.

*Proof:* Let $X$ be the list of length $n$ to be sorted, and let $t = \mathrm{runs}(X)$ be the number of of ascending substrings of $X$. Then the time required by natural mergesort is proportional to $n(1 + \log t)$, as the number of merging phases is bounded by $1 + \log t$ and each phase takes linear time.

To show the optimality of the algorithm we have to estimate the sizes of the sets below$(X$, runs$)$. This is done in the following lemma and corollary.

*Lemma 2:* For all integers $n$ and $t$ with $t \leq (n + 1)/2$ there are at least $5^{-t}t^n$ permutations of $\{1, \cdots, n\}$ with exactly $t$ runs.

*Proof:* Let $R(n, t)$ denote the number of such permutations. These numbers are known as Eulerian numbers. We will use the following facts: for all $n, t \geq 1, t \leq n$.

i) $R(n, 1) = 1$;

ii) $R(n,t) = R(n, n - t + 1)$;

iii) $R(n,t) = tR(n - 1,t) + (n - t + 1)R(n - t, t - 1)$.

Property i) is trivial. Properties ii) and iii) are from [16, eqs. 5.1.3.-(7) and 5.1.3.-(2)].

The proof of the lemma proceeds by induction on $n + t$. If $n + t \leq 4$, the claim is trivial.

Assume that $n + t > 4$ and that the claim holds for all $n'$, $t'$ with $t' \leq (n' + 1)/2$ and $n' + t' < n + t$. If $t \leq n/2$, we can apply iii) and the induction assumption since $t \leq ((n - 1) + 1)/2$. Thus, we get

$$\begin{aligned} R(n,t) &\geq tR(n - 1,t) \\ &\geq t5^{-t}t^{n-1} = 5^{-t}t^n \end{aligned}$$

proving the claim.

The remaining case, $n/2 < t \leq (n + 1)/2$, causes all the difficulty since we cannot simply use the induction assumption. In this case $n = 2t - 1$ and $t = (n + 1)/2$. Using ii) we obtain

$$\begin{aligned} R(n - 1,t) &= R(n - 1, n - 1 - t + 1) \\ &= R(n - 1, n - t) = R(n - 1, t - 1). \end{aligned}$$

Thus, iii) gives us

$$\begin{aligned} R(n,t) &= tR(n - 1,t) + (n - t + 1)R(n - 1,t - 1) \\ &= tR(n - 1,t - 1) \\ &\quad + (n - t + 1)R(n - 1,t - 1) \\ &= (n + 1)R(n - 1,t - 1) = 2tR(n - 1,t - 1). \end{aligned}$$

The induction assumption can be applied to $R(n - 1, t - 1)$ since $t - 1 = \lfloor((n - 1) + 1)/2\rfloor$. We get

$$\begin{aligned} R(n,t) &\geq 2t5^{-(t-1)}(t - 1)^{(n-1)} \\ &= 10t5^{-t}(t - 1)^{(n-1)}. \end{aligned}$$

Our claim is therefore

$$10t5^{-t}(t - 1)^{(n-1)} \geq 5^{-t}t^n$$

which has the equivalent forms

$$10t/(t - 1) \geq (t/(t - 1))^n$$

that is,

$$10(1 + 1/(t - 1)) \geq (((1 + 1/(t - 1))^{t-1})^2(1 + 1/(t - 1))$$

(as $n = 2t - 1$), and finally,

$$10 \geq ((1 + 1/(t - 1))^{t-1})^2.$$

But $(1 + 1/x)^x$ is an ascending function of $x$ with limit $e < 3$. Thus, the right-hand side is bounded by $9 < 10$, and the lemma has been proved. □

*Corollary 3:* For some $c, d > 0$ and for all $X$ we have

$$\log|\mathrm{below}(X, \mathrm{runs})| \geq cn \log t - dt$$

if $|X| = n$ and $t$ is the number of runs in $X$.

*Proof:* If $t \le (n+1)/2$, then by Lemma 2 there are at least $5^{-t}t^n$ permutations $Y$ with runs$(Y)$ = runs$(X)$. All these belong to the set below$(X, \text{runs})$, and thus the claim holds.

If $(n+1)/2 < t \le n$, then below$(X, \text{runs})$ contains at least half of the permutations of $\{1, \cdots, n\}$, by fact ii) used in the proof of Lemma 2. Therefore, the claim holds. □

Now Theorem 1 follows easily from the definition of $m$-optimality. □

We now turn to algorithms optimal with respect to other measures. Consider the following skeleton for insertion sorting of $X = \langle x_1, \cdots, x_n\rangle$:

```
for $i := 1$ to $n$ do
    insert $x_i$ to its proper place
    among the elements of the sorted list
    formed by elements $x_1, \cdots, x_{i-1}$.
```

Depending on the implementation of the sorted list used to represent $x_1, \cdots, x_{i-1}$ this skeleton gives different algorithms.

If we use a data structure for sorted lists allowing insertions at distance $h$ from the end of the list in time $O(1 + \log(h+1))$, we arrive at algorithms described in [21] (and briefly in [12] where also the use of other measures is suggested). Defining

$$h_j = |\{i \mid 1 \le i < j \text{ and } x_i > x_j\}|$$

we have that $h_j$ is the distance in inserting $x_j$, and that $\sum h_j = \text{inv}(X)$ is the number of inversions in $X$. From [21] we have the following reasoning. The execution time of the algorithm is

$$\begin{aligned}\sum_j c(1 + \log(h_j+1)) &= cn + c\log\prod_j (h_j+1)\\ &= cn + cn\log\Big(\prod_j (h_j+1)^{1/n}\Big)\\ &\le cn + cn\log\Big(\sum_j (h_j+1)/n\Big)\\ &= cn + cn\log(1 + \text{inv}(X)/n)\end{aligned}$$

as the arithmetical average $((\Sigma_j (h_j+1))/n)$ is never smaller than the geometrical one $((\Pi_j (h_j+1))^{1/n})$. The result [12]

$$\log|\text{below}(X, \text{inv})| = \theta(n\log(1 + \text{inv}(X)/n))$$

shows therefore that insertion sort implemented in this way is optimal with respect to the number of inversions.

This insertion sort algorithm is not optimal with respect to the number of runs: among the inputs with two runs the sequence $X = \langle n+1, n+2, \cdots, 2n, 1, 2, \cdots, n\rangle$ makes the algorithm use $O(n\log n)$ time. There are, however, only $O(2^{2n})$ inputs of length $2n$ with two runs, and therefore an optimal algorithm should sort them in linear time.

The smoothsort algorithm of Dijkstra [6] gives an example of an algorithm which, despite its claimed "smoothness," is not optimal with respect to the number of inversions, as was recently shown by Hertel [14].

*Local Insertion Sort:* Suppose now that we have a data structure for representing sorted lists which is capable of making insertions in time $O(1 + \log(d+1))$ where $d$ is the distance from the previous element inserted. We call an insertion sort algorithm implemented using such a data structure *local insertion sort*.

A data structure applicable here is the finger tree of [12] or [18]. Simpler alternatives, however, are the level-linked 2–3 and 2–4 trees of Brown and Tarjan [2] and Huddleston and Mehlhorn [15]. These structures are simple enough to make local insertion sort quite practical; their disadvantage is the large amount of space needed. Actually, these structures achieve the $O(1 + \log(d+1))$ bound only in the amortized sense, i.e., averaged over a sequence of searches. This is, however, quite sufficient for our purposes.

In the sequel we will show that local insertion sort is optimal with respect to three natural measures of presortedness: number of inversions, number of runs, and number of elements which have to be removed to leave a sorted list (rem).

The execution time of local insertion sort is $\Sigma_j c(1 + \log(d_j+1))$ where $d_j$ is the distance from the previous element

$$d_j = |\{i \mid 1 \le i < j \text{ and } (x_{j-1} < x_i < x_j \text{ or } x_j < x_i < x_{j-1})\}|.$$

*Theorem 4:* Local insertion sort is optimal with respect to the number of inversions.

*Proof:* The intuitive idea for the result is simple. The efficiency of local insertion sort depends on the distance between successive insertions. In an input containing few inversions most insertions occur near the end of the list, as the previous analysis showed. In such a sequence the distance between successive insertions must also be small, as it is bounded by the larger of the distances from the end of the list of the element to be inserted and the previous element.

If $h_j$ is the quantity describing the distance of element $x_j$ from the end of the list (used in the analysis of the previous algorithm), then as explained above, we have

$$\begin{aligned}d_j + 1 &\le \max\{h_j, h_{j-1}\} + 1 \le h_j + h_{j-1} + 1\\ &\le (h_j+1)(h_{j-1}+1).\end{aligned}$$

Thus,

$$\log(d_j+1) \le \log(h_j+1) + \log(h_{j-1}+1)$$

and therefore

$$\begin{aligned}\sum c\Big(1 + \log(d_j+1)\Big) &\le cn + c\sum\log(d_j+1)\\ &\le cn + 2c\sum\log(h_j+1).\end{aligned}$$

As in the previous analysis, we obtain that this quantity is

$$O(n + n\log(1 + \text{inv}(X)/n)),$$

showing the optimality of the algorithm. □

*Theorem 5:* Local insertion sort is optimal with respect to the number of runs.

*Proof:* In our study of natural mergesort we showed that its time requirement, $O(n(1 + \log t))$ for an input sequence of length $n$ with $t$ runs, is sufficient for optimality. So it is enough to show that local insertion sort achieves the same time bound.

Let $d_j$ be defined as before, and let $R(a)$, $a = 1, \cdots, t$, be the subrange of $[1 \cdots n]$ containing the indexes of the $a$th run. Then the running time of local insertion sort can be expressed as

$$\sum_j c(1 + \log(d_j + 1)) = cn + c\left(\sum_j \log(d_j + 1)\right)$$
$$= cn + c\left(\sum_{a=1}^{t}\left(\sum_{j \in R(a)} \log(d_j + 1)\right)\right).$$

For all $a$ we have $\Sigma\{d_j \mid j \in R(a)\} \le 2n$; furthermore, $\Sigma\{R(a) \mid a = 1, \cdots, t\} = n$. Thus, we have to show that the double sum is $O(n \log t)$ under these restrictions. We use two lemmas, whose proofs are fairly straightforward and are therefore omitted.

*Lemma 6:* Let $R$ be a nonempty set of size $r$ and $d_j$ a positive real number for each $j \in R$. Assume $\Sigma d_j = m$. Then the maximum of

$$\sum_{j \in R} \log(d_j + 1)$$

is obtained by letting $d_j = m/r$ for all $j \in R$. □

Lemma 6 implies that

$$\sum_{j \in R(a)} \log(d_j + 1) \le |R(a)| \cdot \log(2n/|R(a)| + 1)$$

for all $a$. We still have the problem of bounding the sum

$$\sum_{a=1}^{t} |R(a)| \cdot \log(2n/|R(a)| + 1).$$

*Lemma 7:* Assume $0 < r_a \le n$ for all $a = 1, \cdots, t$ and $\Sigma r_a = n$. Then

$$\sum_{a=1}^{t} \log(2n/r_a + 1)$$

is bounded by $2n(1 + \log t)$. □

Lemma 7 gives the bound $O(n(1 + \log t))$ for the running time of local insertion sort. As in Theorem 1, this is sufficient for optimality with respect to the measure runs, and thus Theorem 5 is proved. □

*Theorem 8:* Local insertion sort is optimal with respect to the measure rem.

*Proof:* Let $|X| = n$ and $\text{rem}(X) = s$. The running time of local insertion sort can be approximated as follows.

The time required for inserting element $x_j$ is $O(1 + \log(d_j + 1))$ where $d_j$ is as before. Let $A$ be the set of indexes of an ascending subsequence of length $n - s$, and let $B$ be the set of remaining indexes of $[1 \cdots n]$. Let

$$C = \{j \in A \mid j = 1 \text{ or } j - 1 \in A\} \subseteq A$$

be the indexes in $A$ such that the previous element is in $A$. Then $\Sigma\{d_j \mid j \in C\} \le s$ since only elements in $B$ can contribute to these $d_j$'s. Let $D = (A - C) \cup B$; then $C \cup D = [1 \cdots n]$. Since $|B| = s$, we have $|A - C| \le s$ and $|D| \le 2s$. For $j \in D$ we have $d_j \le j$.

Now

$$\sum c(1 + \log(d_j + 1))$$
$$= cn + c\left(\sum_{j \in C} \log(d_j + 1)\right) + c\left(\sum_{j \in D} \log(d_j + 1)\right).$$

We apply Lemma 6 to the second term. Note first that the sum is maximal if all $d_j$'s are positive. Thus, the second term is bounded by

$$c(n - s) \log(s/(n - s) + 1) \le cn \log 2 = cn.$$

For the third term we have the upper bounds

$$c\left(\sum_{j \in D} \log(j + 1)\right) \le c\left(\sum_{j \in D} \log 2j\right)$$
$$\le cn + c\sum_{j \in D} \log j \le cn + \sum_{j=0}^{2s-1} \log(n - j).$$

As

$$\prod_{j=0}^{2s-1} (n - j) \le \left(\prod_{j=0}^{s-1} (n - j)\right)^2 (n - s)$$

we have

$$\sum_{j=0}^{2s-1} \log(n - j) \le 2\sum_{j=0}^{s-1} \log(n - j) + \log(n - s).$$

Thus, the running time is bounded by

$$3cn + 2c\sum_{j=0}^{s-1} \log(n - j) + \log(n - s)$$
$$\le 4cn + 2c\sum_{j=0}^{s-1} \log(n - j).$$

Next we estimate the number of permutations of $\{1, \cdots, n\}$ with $\text{rem}(X) \le s$. There are at least

$$\binom{n}{n - s} s! = \frac{n!}{(n - s)!}$$

such permutations since we can first choose the $n - s$ elements forming the ascending subsequence (of length $n - s$) and place them in ascending order at the beginning of the permutation. The remaining $s$ elements can then be permuted at will. The permutations clearly have $\text{rem}(X) \le s$. Thus,

$$\log|\text{below}(X, \text{rem})| \ge \log(n!/(n - s)!)$$
$$\ge \log \sum_{j=0}^{j-1} (n - j) = \sum_{j=0}^{j-1} \log(n - j),$$

and the theorem is proven. □

We do not know whether local insertion sort is exc-optimal.

We have implemented local insertion sort in Pascal using level-linked 2–3 trees. The algorithm works as predicted. It is faster than a carefully tuned version of quicksort on, e.g., inputs consisting of two runs. On random inputs, local insertion sort uses approximately 1.5 times as many comparisons as quicksort does. For CPU time, this ratio is about 2.5. Further details of the implementation and experiments can be found in [19].

Note that if we have measures $m$ and $m'$, and an $m$-optimal algorithm $A$ and an $m'$-optimal algorithm $A'$, we can construct an algorithm which is both $m$- and $m'$-optimal. This algorithm simply simulates $A$ and $A'$ in parallel until one of them halts. The point of the results presented above is that local insertion sort is a natural method, which is optimal with respect to three measures.

## V. The Existence of Optimal Algorithms

The previous section showed that for certain measures $m$ there exist $m$-optimal algorithms. In this section we address the problem of finding optimal algorithms for arbitrary measures.

As the conditions imposed on measures are light, we cannot expect to find concrete algorithms. We rather direct our attention to the existence of comparison trees. For them we can show a weaker result: for any measure $m$ there exist weakly $m$-optimal comparison trees.

For restricted measures we get truly optimal trees. If $m$ is a measure for which $m(X)$ is computable in a linear number of comparisons (or, more generally, in time $c \max\{|X|, \log|\text{below}(X, m)|\}$), then there exist $m$-optimal comparison trees.

The existence proof is based on a result of Fredman [11] showing that for the sorting type problem the information-theoretic bound on the number of comparisons is tight.

*Lemma 9 [11, Theorem 3.1]:* Let $S$ be a set of permutations $X = \langle x_1, \cdots, x_n \rangle$ of a set of $n$ integers. Then there exists a comparison tree $T$ whose nodes contain comparisons of the type "$x_i < x_j$?", differentiating between the elements of $S$, and the height of $T$ is at most $\log(|S|) + 2n$. □

If we could find for any $S$ a comparison "$x_i < x_j$?" approximately halving $S$, then the claim would follow even without the term $2n$. Such a choice of comparison is not always possible, as is seen by considering the set $Q$ of permutations with at most one inversion. Then $|Q| = n$, but each comparison differentiates only one element from the set.

The nice idea in Fredman's proof is to use insertion sort and longer sequences of comparisons. Although we cannot be sure to halve the set in one comparison, we can ensure that if we have to make $r$ comparisons to insert one element into its correct place among the previous elements, then the number of permutations in $S$ still possible after this element is sufficiently small.

As noted in [20] for the case of inv, Lemma 9 implies that for fixed $b$ there exist comparison trees operating in the $m$-optimal time bound $\log|\text{below}'(b, X, m)|$. The following theorem considers the case when $b$ is given as input.

*Theorem 10:* For any measure of presortedness $m$ there exist weakly $m$-optimal comparison trees.

*Proof:* Let input length $n$ be given. We suppose that in addition to the input $X$ of length $n$ the comparison tree also receives as an input an upper bound $b$ for the $m$-disorder of $X$, and that we are able to compare this number against integers.

Denote by $a(i)$ the $i$th smallest possible value of $m$ on inputs of length $n$. As a first stage in the comparison tree we search for $b$ among the $a(i)$'s using one-sided (or unbounded) binary search: we compare $b$ to $a(2^j - 1)$ for $j = 1, \cdots,$ until $b < a(2^j - 1)$, and then search for $b$ in the interval $a(2^{j-1}) \cdots a(2^j - 1)$ using ordinary (bounded) binary search. This search takes time $O(\log i)$ where $i$ = smallest $i$ such that $b < a(i)$.

At the node of the tree where $i$ has been determined we append the tree given by Fredman's lemma for the set of permutations $X$ with $m(X) \leq b$. The height of this subtree is at most $2 \cdot \max\{n, \log|S|\}$ where $S = \text{below}'(b, X, m)$. So the theorem is proven if we can show that the branching according to $b$ can be made in this time. But this is easy: as

$$\{Y \mid |Y| = n \text{ and } m(Y) = a(j)\} \neq 0$$

for all $j$ [by definition of $a(j)$], we have $|S| \geq i$, and hence the time used for searching $b$, $O(\log i)$, is bounded by $O(\log|S|)$. □

*Theorem 11:* Assume that $m$ is a measure of presortedness such that for some $c > 0$ and for all $X$ the value of $m(X)$ is computable in

$$c \max\{|X|, \log|\text{below}(X, m)|\}$$

comparisons. Then there exist $m$-optimal comparison trees.

*Proof:* The tree is constructed as follows. Starting from the root, construct a tree calculating $m(X)$ in the stated time. At the node where $m(X)$ is known, append the tree given by Lemma 9 for the set below$(X, m)$. The optimality of the resulting tree follows directly from the assumptions. □

The proofs in this section depend on Lemma 9 and are quite nonconstructive. However, the proof of Lemma 9 is based on structural information about the sets below$(X, m)$. If $m$ is a reasonably behaving measure, such information is available, and it can perhaps be used to turn the comparison trees into concrete algorithms.

## VI. Concluding Remarks

We have studied the use of presortedness in sorting by studying various measures of presortedness, by defining optimality with respect to a measure, by giving examples of optimal sorting algorithms, and by studying the existence of $m$-optimal algorithms for arbitrary measures $m$. In this concluding section we make some remarks and sketch possible extensions of the work.

Different measures have totally different behavior, and for practical considerations it would be important to know what kind of presortedness can be expected (if any). If this information is not available, but some presortedness is expected, then the use of a sorting algorithm optimal with respect to many measures (e.g., local insertion sort) could be useful.

The properties 1)–5) necessary for a measure are not strong enough to be very useful. In trying to strengthen the results of Section V additional properties could be useful, as the present proofs in that section do not use the nontrivial properties 3), 4), and 5) at all. A possible extension is indeed to try to find necessary and sufficient conditions for the existence of $m$-optimal algorithms.

Turning the picture the other way around, one might also ask whether for any algorithm with worst case $O(n \log n)$ and best case $O(n)$ (e.g., smoothsort) there exists a measure $m$ (different from $m_0$ and $m_{01}$) such that the method is $m$-optimal.

## Acknowledgment

I would like to thank K.-J. Räihä and E. Ukkonen for useful comments.

## References

[1] R. Ash, *Information Theory.* New York: Interscience, 1965.
[2] M. R. Brown and R. E. Tarjan, "Design and analysis of a data structure for representing sorted lists," *SIAM J. Comput.*, vol. 9, no. 3, pp. 594–614, Aug. 1980.
[3] C. R. Cook and D. J. Kim, "Best sorting algorithm for nearly sorted lists," *Commun. ACM*, vol. 23, pp. 620–624, Nov. 1980.
[4] R. B. K. Dewar, S. M. Merritt, and M. Sharir, "Some modified algorithms for Dijkstra's longest upsequence problem," *Acta Informatica*, vol. 18, pp. 1–15, 1980.
[5] E. W. Dijkstra, "Some beautiful arguments using mathematical induction," *Acta Informatica*, vol. 13, pp. 1–13, 1980.
[6] ——, "Smoothsort, An alternative to sorting in situ," *Sci. Comput. Programming*, vol. 1, pp. 223–233, 1982.
[7] W. Dobosiewicz, "Sorting by distributive partitioning," *Inform. Processing Lett.*, vol. 7, pp. 1–6, 1978.
[8] G. Ehrlich, "Sorting and searching real numbers," *J. Algorith.*, vol. 2, pp. 1–12, 1981.
[9] M. H. Ellis and J. M. Steele, "Fast sorting of Weyl sequences using comparisons," *SIAM J. Comput.*, vol. 10, pp. 88–95, Jan. 1981.
[10] M. L. Fredman, "Two applications of a probabilistic search technique: Sorting X + Y and building balanced search trees," in *Proc. 7th Annu. ACM Symp. Theory Comput.*, 1975, pp. 240–244.
[11] ——, "How good is the information theory bound in sorting?" *Theor. Comput. Sci.*, vol. 1, pp. 355–361, 1976.
[12] L. J. Guibas, E. M. McCreight, M. F. Plass, and J. R. Roberts, "A new representation of linear lists," in *Proc. 9th Annu. ACM Symp. Theory Comput.*, 1977, pp. 49–60.
[13] L. H. Harper, T. H. Payne, J. E. Savage, and E. Straus, "Sorting $X + Y$," *Commun. ACM*, vol. 18, pp. 347–349, June 1975.
[14] S. Hertel, "Smoothsort's behaviour on presorted sequences," *Inform. Processing Lett.*, vol. 16, pp. 165–170, 1983.
[15] S. Huddleston and K. Mehlhorn, "A new data structure for representing sorted lists," *Acta Informatica*, vol. 17, pp. 157–184, 1982.
[16] D. E. Knuth, *The Art of Computer Programming, Vol. III: Sorting and Searching.* Reading, MA: Addison-Wesley, 1973.
[17] ——, *The Art of Computer Programming, Vol. II: Seminumerical algorithms (2nd ed.).* Reading, MA: Addison-Wesley, 1981.
[18] S. R. Kosaraju, "Localized search in sorted lists," in *Proc. 13th Annu. ACM Conf. Theory Comput.*, 1981, pp. 62–69.
[19] H. Mannila, "Implementation of a sorting algorithm suitable for presorted files," Dep. Comput. Sci., Univ. Helsinki, Tech. Rep., 1984.
[20] K. Mehlhorn, "Searching, sorting and information theory," in *Mathematical Foundations of Computer Science 1979*, J. Becvar, Ed. Berlin: Springer-Verlag, 1979, pp. 131–145.
[21] ——, "Sorting presorted files," in *4th GI Conf. Theor. Comput. Sci.*, Berlin: Springer-Verlag, 1979, pp. 199–212.
[22] R. Sedgewick, "Quicksort," Ph.D. dissertation, Stanford Univ., Stanford, CA, 1975.

**Heikki Mannila** was born in Espoo, Finland, on January 4, 1960. He received the M.Sc. degree from the University of Helsinki, Helsinki, Finland, in 1982.

He has been working at the Department of Computer Science, University of Helsinki, since 1981. The material in this paper is based on his Ph.D. dissertation "Instance complexity for sorting and NP-complete problems," which is to appear in March 1985. His current interests include analysis of algorithms, logic programming, and database theory.