Algorithmica (1993) 9: 629–648

Algorithmica

Sublinear Merging and Natural Mergesort

Svante Carlsson,[1] Christos Levcopoulos,[2] and Ola Petersson[2]

Abstract. The complexity of merging two sorted sequences into one is linear in the worst case as well as in the average case. There are, however, instances for which a sublinear number of comparisons is sufficient. We consider the problem of measuring and exploiting such instance easiness. The merging algorithm presented, Adaptmerge, is shown to adapt optimally to different kinds of measures of instance easiness. In the sorting problem the concept of instance easiness has received a lot of attention, and it is interpreted by a measure of presortedness. We apply Adaptmerge in the already adaptive sorting algorithm Natural Mergesort. The resulting algorithm, Adaptive Mergesort, optimally adapts to several, known and new, measures of presortedness. We also prove some interesting results concerning the relation between measures of presortedness proposed in the literature.

Key Words. Merging, Sorting, Instance easiness, Measures.

1. Introduction. It is well known that $\Omega(n \log n)$ time is necessary to sort n elements in a comparison-based model of computation [1]. Despite this fact, some instances seem easier than others and can be sorted faster than indicated by the lower bound, for example, an already sorted sequence or the concatenation of two sorted sequences. This observation was first made by Burge [2], who identified this *instance easiness* with the amount of existing order (*presortedness*) in the sequence, and also proposed measures of existing order.

After being considered by, among others, Cook and Kim [3], Knuth [4], and Mehlhorn [1], the concept of presortedness was eventually formalized by Mannila [5], who studied several measures of presortedness including the number of runs and the number of inversions. Mannila also studied the problem of how a sorting algorithm can take advantage of, and thereby adapt to, the existing order. A sorting algorithm is said to be *adaptive* with respect to a measure of presortedness if it sorts all sequences, but performs particularly well on those having a high degree of presortedness according to the measure.

For merging, the concepts of instance easiness and adaptive algorithms have hardly received any attention in the literature. What is known is that in the worst case $\Omega(m \log((n+m)/m))$ time is necessary for merging two sorted sequences X and Y of length n and m, respectively, $m \leq n$, into one, in a comparison-based model of computation [1]. Also for merging some instances are much easier than others. For example, it may be enough to merge a small portion in the end of X with a small portion in the beginning of Y, or the sorted output may be

[1] Department of Computer Science, Luleå University, S-95187 Luleå, Sweden.
[2] Department of Computer Science, Lund University, Box 118, S-22100 Lund, Sweden.

Received November 6, 1990; revised June 26, 1991. Communicated by Takao Nishizeki.

obtained by simply splitting X into a small number of parts and inserting parts of Y in between.

As in the sorting problem, in order to be able to take advantage of some kind of instance easiness, or *easiness* for short, when merging we must be more precise with what it means. We examine two different approaches: *Max* measures the maximum distance that an element is from its correct position; *Block* intuitively tells into how many consecutive subsequences the input sequences have to be partitioned for the merging to take place.

We also provide an algorithm, *Adaptmerge*, which adapts to the measures considered. (The concept of adaptivity of a merging algorithm is analogous to that of a sorting algorithm.) Adaptmerge is even *optimal* with respect to *Max* and *Block*. Here, optimality with respect to a measure means maximum adaptation (in an asymptotic sense).

Max has previously been used as a measure of presortedness, while *Block* is a new measure that generalizes two well-known measures, namely *Rem*, the minimum number of elements that have to be removed from a sequence in order to leave a sorted sequence, and *Exc*, the minimum number of arbitrary exchanges needed to bring a sequence into sorted order.

Mannila [5] proved that Natural Mergesort is optimal with respect to the measure *Runs*, that is, the number of consecutive upsequences of maximum length, even if a straightforward merging algorithm is used. We show how to apply Adaptmerge in Natural Mergesort to extend its adaptivity. The resulting algorithm, *Adaptive Mergesort*, is optimal with respect to several measures, namely *Max*, *Block*, *Rem*, *Exc*, and *Runs*.

The remainder of the paper is organized as follows. In Section 2 we state some preliminary definitions and basic results for adaptive sorting. We also formally define our measures of presortedness and establish time lower bounds on sorting algorithms that adapt to them. Section 3 is similar to Section 2, but for merging instead of sorting. In Section 4 the algorithm Adaptmerge is presented and analyzed. In Section 5 we apply Adaptmerge in Natural Mergesort and analyze its behavior with respect to the measures. In Section 6 we examine how *Block* is related to other measures of presortedness. Finally, in Section 7 we give some concluding remarks.

2. Adaptive Sorting

2.1. Measures of Presortedness. Let $X = \langle x_1, \ldots, x_n \rangle$ be a *sequence* of n elements x_i from some totally ordered set. For simplicity, we assume that the elements are distinct. For two sequences, $X = \langle x_1, \ldots, x_n \rangle$ and $Y = \langle y_1, \ldots, y_m \rangle$, their *concatenation* XY is the sequence $\langle x_1, \ldots, x_n, y_1, \ldots, y_m \rangle$. Furthermore, let $|X|$ denote the *length* of X and $\|S\|$ the *cardinality* of a set S. S_n denotes the set of all permutations of $\{1, \ldots, n\}$. A sequence obtained by deleting zero or more elements from X is called a *subsequence* of X. In particular, a sequence Y of length m is a *consecutive subsequence* of X if there exists an i, $1 \leq i \leq n - m + 1$, such that $Y = \langle x_i, \ldots, x_{i+m-1} \rangle$. For an element x,

we define its *rank* in a sequence Y of length m as the number of elements in Y that are smaller than x. More formally,

$$rank(x, Y) = \|\{y_j | 1 \leq j \leq m \text{ and } x > y_j\}\|.$$

Maximum Distance. The measure *Max* tells the maximum distance that an element is from its correct position. More formally,

DEFINITION 1. For a sequence X of length n, $Max(X) = \max_{1 \leq i \leq n} |i - \pi(i)|$, where $\pi(i) = rank(x_i, X) + 1$, for $1 \leq i \leq n$.

That is, $\pi(i)$ the correct position for the element x_i, and *Max* tells the maximum distance that an element has to be moved.

Max is a natural measure of presortedness that, unfortunately, is very sensitive to global disorder. For instance, the sequence $\langle n, 1, 2, \ldots, n-1 \rangle$ maximizes the measure.

Number of Blocks. The measure *Block* tells the number of consecutive subsequences in the input that remain in the sorted output.

DEFINITION 2. For a sequence X of length n,

$$Block(X) = \|\{i | 1 \leq i < n \text{ and } \pi(i) + 1 \neq \pi(i+1)\}\| + 1,$$

where $\pi(i) = rank(x_i, X) + 1$, for $1 \leq i \leq n$.

Note that the elements contributing to *Block* are those that receive a new successor in the sorted sequence. These elements divide the input sequence into $Block(X)$ consecutive subsequences, which will henceforth be called *blocks*.

2.2. Optimality. The concept of optimality of a sorting algorithm with respect to a measure of presortedness was defined by Mannila [5]. We use the following equivalent definitions [6].

DEFINITION 3. Let M be a measure of presortedness, and let T_n be the set of binary comparison trees for the set S_n. Then, for any $k \geq 0$ *and* $n \geq 1$,

$$C_M(n, k) = \min_{T \in T_n} \max_{\pi \in below_M(n,k)} \{\text{number of comparisons spent by } T \text{ to sort } \pi\},$$

where $below_M(n, k) = \{\pi | \pi \in S_n \text{ and } M(\pi) \leq k\}$.

DEFINITION 4. Let M be a measure of presortedness, and let A be a comparison-based sorting algorithm that uses $T_A(X)$ steps on input X. We say that A is *M-optimal*, or *optimal with respect to M*, if $T_A(X) = O(C_M(|X|, M(X)))$.

When proving optimality of an adaptive sorting algorithm the following theorem is helpful.

THEOREM 1. *Let M be a measure of presortedness. Then*

$$C_M(n, k) = \Theta(n + \log\|below_M(n, k)\|).$$

PROOF. As any binary comparison tree for $below_M(n, k)$ has at least $\|below_M(n, k)\|$ leaves, its height is at least $\log\|below_M(n, k)\|$. Further, any comparison tree for S_n must examine each element at least once. Therefore,

$$C_M(n, k) = \Omega(n + \log\|below_M(n, k)\|).$$

The upper bound follows from a result by Fredman [7], who proved that if $S \subseteq S_n$, there exists a comparison tree of height at most $2n + \log\|S\|$ that differentiates between the elements of S. □

A lower bound for sorting algorithms with respect to a measure of presortedness is thus obtained by bounding the size of the *below*-set from below.

2.3. Lower Bounds

LEMMA 2. $C_{Max}(n, k) = \Omega(n \log k)$.

PROOF. We bound the cardinality of $below_{Max}(n, k)$ from below.

For simplicity, assume that $k + 1$ divides n. Divide the identity permutation in S_n into $n/(k + 1)$ consecutive subsequences, each of length $k + 1$. Let $\pi \in S_n$ be any permutation obtained by rearranging the elements within each of the subsequences. As all elements in π are in their correct consecutive subsequences, it follows that $Max(\pi) \leq k$, and thus $\pi \in below_{Max}(n, k)$. Since there are $(k + 1)!$ different ways of rearranging each consecutive subsequence, we have that

$$\|below_{Max}(n, k)\| \geq ((k + 1)!)^{n/(k+1)}.$$

Taking the logarithm and applying Theorem 1 proves the lemma. □

Instead of making another combinatorial construction to derive a lower bound on $C_{Block}(n, k)$ we introduce a general technique that can be used to obtain new lower bounds from known ones.

LEMMA 3. *Let M_1 and M_2 be measures of presortedness. If there exist a function f and a constant c such that $M_1(X) \leq f(n) \cdot (M_2(X) + c)$, for any sequence X of length n, then*

$$C_{M_1}(n, k) \geq C_{M_2}\left(n, \frac{k}{f(n)} - c\right).$$

PROOF. If we show that

$$(1) \qquad below_{M_1}(n, k) \supseteq below_{M_2}\left(n, \frac{k}{f(n)} - c\right),$$

the lemma follows from Definition 3. Let $\pi \in below_{M_2}(n, k/f(n) - c)$. Then

$$M_1(\pi) \leq f(n) \cdot (M_2(\pi) + c) \leq f(n) \cdot \left(\frac{k}{f(n)} - c + c\right) = k,$$

proving that $\pi \in below_{M_1}(n, k)$, and thus (1) holds. □

Establishing a lower bound on $C_{Block}(n, k)$ thus requires bounding *Block* from above in terms of some measure M, and knowledge of a lower bound on $C_M(n, k)$. As M we use the measure *Rem* [3], [5], defined as

$$Rem(X) = n - \max\{k \mid X \text{ has an ascending subsequence of length } k\}.$$

We first bound *Block* in terms of *Rem*.

LEMMA 4. *For any sequence X, Block(X)* $\leq 3 \cdot Rem(X) + 1$.

PROOF. Consider a sequence with $Rem(X) = k$. Let X_1 be a longest ascending subsequence in X and let X_2 be the subsequence which has to be removed from X to leave X_1. By the definition of *Rem*, $|X_2| = k$.

We derive an upper bound on the number of elements in X_1 and X_2 that can contribute to *Block*(*X*), starting with X_1. Let x_i be an element in X_1. We distinguish two cases.

(a) If x_{i+1} is in X_2, it cannot be the successor of x_i in the sorted sequence, because in that case the element following x_i in X_1 is greater than x_{i+1}, and x_{i+1} could thus be added to X_1 without violating its sortedness. This contradicts the assumption that X_1 is a *longest* ascending subsequence in X. Hence, in this case x_i receives a new successor. We note that there can be at most $|X_2|$ x_i's of this kind.

(b) If x_{i+1} is in X_1, the successor of x_i in the sorted sequence is either x_{i+1} or some element from X_2. In the former case x_i does not receive a new successor, while in the latter it does. Again, there can be at most $|X_2|$ x_i's of the latter kind.

We conclude that there are at most $2|X_2| = 2k$ elements in X_1 that receive new successors. If also all elements in X_2 receive new successors, we arrive at the claimed bound. □

At first it might appear that the lemma can be strengthened to

$$Block(X) \leq 2 \cdot Rem(X) + 1.$$

However, the stated bound is tight.

Next, we need a lower bound on $C_{Rem}(n, k)$. From [5],

LEMMA 5. $C_{Rem}(n, k) = \Omega(n + k \log k)$.

Combining Lemmas 3–5 gives

LEMMA 6. $C_{Block}(n, k) = \Omega(n + k \log k)$.

3. Adaptive Merging

3.1. Measuring Instance Easiness. We use both *Max* and *Block* for measuring easiness in the merging problem. The modifications needed are due to that the measures should not take the relative order of input sequences into account.

Maximum Distance. Given two sorted sequences X and Y, *Max* tells the maximum distance that an element in either sequence has to be moved.

DEFINITION 5. For two sorted sequences X and Y of length n and m, respectively, $Max(X, Y) = \min\{Max(XY), Max(YX)\}$.

The following lemma can easily be proved [8].

LEMMA 7. *For any two sorted sequences X and Y of length n and m, $m \leq n$, respectively,*

(a) $Max(XY) = \max\{rank(x_n, Y), n - rank(y_1, X)\}$.
(b) *If $x_1 < y_1$ and $x_n < y_m$, then*

$$Max(X, Y) = \max\{rank(x_n, Y), n - rank(y_1, X)\}.$$

(c) *If $x_1 < y_1$ and $x_n > y_m$, then*

$$Max(X, Y) = \max\{m, \min\{n - rank(y_1, X), rank(y_m, X)\}\}.$$

Number of Blocks. The *Block* measure for the merging problem can also be defined in terms of that for sorting, however, the following definition is easier to grasp.

DEFINITION 6. Let X and Y be two sorted sequences of length n and m, respectively, and let $Z = \langle z_1, \ldots, z_{n+m} \rangle$ be the resulting merged sequence. Then

$$Block(X, Y) = \|\{i \mid 1 \leq i < n + m \text{ and } z_i \in X \text{ and } z_{i+1} \in Y \text{ or } z_i \in Y \text{ and } z_{i+1} \in X\}\| + 1.$$

As for sorting, *Block* tells the number of blocks in the input sequences that appear in the merged sequence. The intuition then is that if $Block(X, Y)$ is low, X and Y need just be split into a small number of blocks for the merging to take

place. Consequently, we should not spend constant time on every single element, but just find the positions where we have to split the sequences.

3.2. Optimality. The following definitions of optimality of adaptive merging algorithms follow the ones for sorting algorithms in Section 2.2.

DEFINITION 7. Let M be a measure of easiness for the merging problem. Furthermore, let A be any subsequence of $\langle 1, 2, \ldots, n+m\rangle$ and let B be the sequence remaining after removing A. Then

$$below_M(n, m, k) = \{(A, B) \mid |A| = n, |B| = m, \text{ and } M(A, B) \leq k\},$$

for any $k \geq 0$, $m \geq 1$, and $n \geq m$.

DEFINITION 8. Let M be a measure of easiness for the merging problem, and let T_{n+m} be the set of binary comparison trees that merges any pair of sorted sequences of total length $n+m$. Then, for any $k \geq 0$, $m \geq 1$, and $n \geq m$,

$$C_M(n, m, k) = \min_{T \in T_{n+m}} \max_{(A,B) \in below_M(n,m,k)} \{\text{number of comparisons spent by } T \text{ to merge } A \text{ and } B\}.$$

For a comparison-based merging algorithm, let the number of steps used on an input be the time taken to compute a function that can be used to bring the input sequences into a sorted sequence, without performing any comparisons, and not the time required to report the output consecutively in an array. The latter task will always take linear time.

DEFINITION 9. Let M be a measure of easiness for the merging problem, and let A be a comparison-based merging algorithm that uses $T_A(X, Y)$ steps on inputs X and Y. We say that A is *M-optimal*, or *optimal with respect to M*, if $T_A(X, Y) = O(C_M(|X|, |Y|, M(X, Y)))$.

The following theorem follows from a straightforward information-theoretic argument.

THEOREM 8. *Let M be a measure of easiness for the merging problem. Then*

$$C_M(n, m, k) = \Omega(\log \| below_M(n, m, k) \|).$$

3.3. Lower Bounds

LEMMA 9.

$$C_{Max}(n, m, k) = \begin{cases} \Omega(m \log((m+k)/m)) & \text{if } m \leq k, \\ \Omega(k) & \text{otherwise.} \end{cases}$$

PROOF. We estimate the cardinality of $below_{Max}(n, m, k)$ from below. Partition $C = \langle 1, 2, \ldots, n + m\rangle$ into two subsequences A and B of length n and m, respectively. We distinguish two cases depending on the relation between m and k.

If $m \leq k$, let B be any subsequence of length m of the last $m + k$ elements in C, and A is what remains in C after removing B. Any choice of B guarantees that $Max(A, B) \leq Max(AB) \leq k$, and (A, B) is thus in $below_{Max}(n, m, k)$. Since there are $\binom{m+k}{m}$ different choices of B, we therefore have

$$\|below_{Max}(n, m, k)\| \geq \binom{m+k}{m}.$$

Taking the logarithm and applying Theorem 8 proves the claim.

If $m > k$, let B be any subsequence of C constructed as follows. B starts with a subsequence of length k of

$$\langle n - k + 1, n - k + 2, \ldots, n + k - 1, n + k\rangle,$$

followed by the last $m - k$ elements of C. A is the subsequence left in C after removing B. Again, we have that $Max(A, B) \leq Max(AB) \leq k$, and thus $(A, B) \in below_{Max}(n, m, k)$. In this case there are $\binom{2k}{k}$ different ways to choose B, implying

$$\|below_{Max}(n, m, k)\| \geq \binom{2k}{k},$$

which, after taking the logarithm, completes the proof by Theorem 8. □

LEMMA 10. $C_{Block}(n, m, k) = \Omega(k \log((n + k)/k))$.

PROOF. Again, let $C = \langle 1, 2, \ldots, n + m\rangle$. We partition C into two subsequences A and B of length n and m, respectively, as follows. B consists of the first $m - \lfloor k/2\rfloor + 1$ elements of C followed by any subsequence of length $\lfloor k/2\rfloor - 1$ of the remaining elements. A is the sequence left in C after removing B. The number of elements from A and B contributing to $Block(A, B)$ is at most $\lfloor k/2\rfloor$ each. Hence, $(A, B) \in below_{Block}(n, m, k)$. Counting the number of ways to choose B gives

$$\begin{aligned}\log\|below_{Block}(n, m, k)\| &\geq \log\left(\binom{n + \lfloor k/2\rfloor - 1}{\lfloor k/2\rfloor - 1}\right)\\ &= \Omega\left(k \log\left(\frac{n+k}{k}\right)\right),\end{aligned}$$

which completes the proof. □

We note that if the measures are maximized, that is, $Max(X, Y) = n$ and $Block(X, Y) = 2m + 1$, the lemmas give the known worst-case lower bound, that is, $\Omega(m \log((m + n)/m))$.

4. Adaptmerge

4.1. The Algorithm. At first it may seem that even if we have a merging algorithm that performs a sublinear number of comparisons on some inputs, a linear number of assignments is always necessary. This is true if the resulting sequence is required to be reported consecutively in an array; otherwise, one can do better. In Adaptmerge no elements are moved and the number of assignments and comparisons are asymptotically equal.

The input to the algorithm consists of two sorted arrays X and Y of length n and m, respectively. The output is reported in another array Z of length $Block(X, Y)$. Each entry in Z corresponds to a *Block* of X or Y. By reporting the elements in these blocks in the order given by Z the merged sequence is obtained. (See Figure 1.) Hence, Adaptmerge outputs a data structure from which the resulting sequence can be computed in $m + n$ assignments and no comparisons. A similar output format turns out to be crucial when a variant of Adaptmerge is applied in Natural Mergesort in the next section.

Consider the example in Figure 1. To compute the array Z we find the elements in X and Y that will receive new successors in the resulting sequence, together with their successors. Now, the intuition behind the algorithm is that if $Block(X, Y)$ is low, there are large blocks in X and Y, which do not need to be examined entirely. We therefore apply exponential and binary search [1] to pass these blocks as fast as possible in our search for the next element that will receive a new successor.

When merging two sequences X and Y, denote by $X_1, X_2, \ldots, X_p$ and $Y_1, Y_2, \ldots, Y_q$ the blocks of X and Y, respectively. Then we have that $p + q = Block(X, Y)$ and $|p - q| \leq 1$. For simplicity, the following description of Adaptmerge assumes that $x_1 \leq y_1$. (If $x_1 > y_1$, just swap X and Y.) Then Z starts with a block from X, followed by a block from Y, followed by a block from X, and so on. The first block in X, that is, X_1, is $X[1, rank(y_1, X)]$. This is computed by concurrently performing *two* exponential and binary searches for y_1 in X; one backward and one forward. Both searches stop once the sought

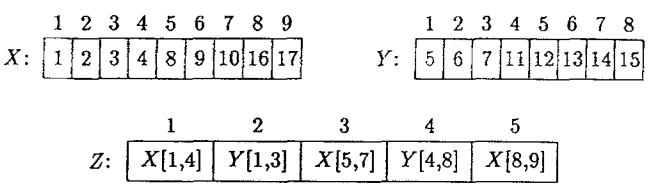

Fig. 1. Example of input and output of Adaptmerge. Here $m = 8$, $n = 9$, $Max(X, Y) = 8$, and $Block(X, Y) = 5$.

position has been reached by either of them. The first block in Y, that is, $Y_1 = Y[1, rank(x_{|X_1|+1}, Y)]$, is computed by an exponential and binary search forward in Y. The second block in X, $X[|X_1| + 1, rank(y_{|Y_1|+1}, X)]$, is computed by an exponential and binary search forward in X. In this way we continue to perform exponential and binary searches, alternating between X and Y. We always start the search from where the last element found to receive a new successor is located. When one of the sequences is finished, the remaining block of the nonempty sequence is returned.

For a slightly less informal description of Adaptmerge, let $B \circ L$ denote concatenating a block B to a sequence of blocks L.

```
function Adaptmerge(X, Y: sequence): sequence of blocks;
  if |Y| = 0 then Adaptmerge: = X
    else begin
      r: = rank(y_1, X);
      Adaptmerge: = X[1, r] ∘ Adaptmerge(Y, X[r + 1, |X|]);
    endelse;
end;
```

Here *rank* is computed by forward exponential and binary searches, except for the first one, which is two-way.

4.2. Upper Bounds

THEOREM 11. *Adaptmerge is optimal with respect to Block.*

PROOF. Let $Block(X, Y) = k$ and recall that an exponential and binary search finds an element located ℓ positions away in a sorted array in $\Theta(\log \ell)$ time. The time consumed by Adaptmerge is given by

$$T(X, Y) = O\left(\sum_{i=1}^{p} \log|X_i| + \sum_{i=1}^{q} \log|Y_i|\right),$$

where the first term corresponds to all searches performed in X and the second corresponds to all searches performed in Y. Here,

$$\sum_{i=1}^{p} \log|X_i| = \log\left(\prod_{i=1}^{p} |X_i|\right) \leq \log\left(\prod_{i=1}^{p} \frac{n}{p}\right)$$
$$= O\left(p \log\left(\frac{n}{p}\right)\right) = O\left(k \log\left(\frac{n}{k}\right)\right),$$

because $p \leq \lfloor k/2 \rfloor$. Similarly,

$$\sum_{i=1}^{q} \log|Y_i| = O\left(k \log\left(\frac{m}{k}\right)\right).$$

Hence,

$$T(X, Y) = O\left(k \log\left(\frac{n}{k}\right) + k \log\left(\frac{m}{k}\right)\right) = O\left(k \log\left(\frac{m+n}{k}\right)\right),$$

which completes the proof, by Lemma 10. □

THEOREM 12. *Adaptmerge is optimal with respect to Max.*

PROOF. Let $|X| = n$, $|Y| = m$, where $m \leq n$, and let $Max(X, Y) = k$. We divide the analysis into three different cases:

(a) X and Y do not overlap at all;
(b) X and Y overlap partially;
(c) one sequence completely overlaps with the other.

(a) If X and Y do not overlap, Adaptmerge completes after the first search, which takes constant time.

(b) There is a partial overlap between X and Y. Assume that $x_1 < y_1$ and $x_n < y_m$. (The case when $y_1 < x_1$ and $y_m < x_n$ is analogous.) Recall that the first search is done by a two-way exponential and binary search with y_1 in X. This takes time

$$O(\log(\min\{n - rank(y_1, X), rank(y_1, X)\})),$$

which is $O(\log k)$ by Lemma 7(b).

Since X will be finished before Y, the last block in Y, that is, Y_q, only causes constant time. The time used by Adaptmerge is therefore

$$T(X, Y) = O\left(\log k + \sum_{i=2}^{p} \log|X_i| + \sum_{i=1}^{q-1} \log|Y_i|\right). \tag{2}$$

The same calculations as in the proof of Theorem 11 give that the second and third terms in (2) sum to

$$O\left((p + q - 2) \cdot \log\left(\frac{n - |X_1| + m - |Y_q|}{p + q - 2}\right)\right).$$

Here $|X_1| = rank(y_1, X)$ and $m - |Y_q| = rank(x_n, Y)$, and thus

$$n - |X_1| + m - |Y_q| \leq n - rank(y_1, X) + rank(x_n, Y) \leq 2k,$$

by Lemma 7(b). Therefore, (2) can be expressed as

$$T(X, Y) = O\left(\log k + (p + q - 2) \cdot \log\left(\frac{k}{p + q - 2}\right)\right). \tag{3}$$

Now, as each block has length at least one,

$$p + q - 2 \leq n - |X_1| + m - |Y_q| \leq 2k,$$

which if inserted into (3) gives that $T(X, Y) = O(k)$. On the other hand, $p + q - 2 = Block(X, Y) - 2$, which is at most $2m - 1$, since the number of blocks is at most $2m + 1$. In this case (3) becomes $O(\log k + m \log(k/m))$, which is $O(m \log((m + k)/m))$. Hence,

$$T(X, Y) = O\left(\min\left\{k, m \log\left(\frac{m + k}{m}\right)\right\}\right),$$

which is *Max*-optimal, by Lemma 9.

(c) If Y overlaps X completely, that is, if $y_1 < x_1$ and $y_m > x_n$, then $Max(X, Y) = n$. By Lemma 9, we may spend $O(m \log((m + n)/m))$ time in this case. However, since Adaptmerge is *Block*-optimal, it is also worst-case optimal, that is, it merges in $O(m \log((m + n)/m))$ time.

If X overlaps Y completely, that is, if $x_1 < y_1$ and $x_n > y_m$, we have to be a little bit careful. As in (b), the first search done by Adaptmerge takes time

$$O(\log(\min\{n - rank(y_1, X), rank(y_1, X)\})).$$

Since $rank(y_1, X) \leq rank(y_m, X)$, this is bounded by $O(\log Max(X, Y))$ by Lemma 7(c).

In this case Y is finished before X so the last block in X, that is, X_p, takes only constant time to handle. The total time spent by Adaptmerge is therefore

$$\begin{aligned} T(X, Y) &= O\left(\log k + \sum_{i=2}^{p-1} \log|X_i| + \sum_{i=1}^{q} \log|Y_i|\right) \\ &= O\left(\log k + (p + q - 2) \log\left(\frac{n - |X_1| - |X_p| + m}{p + q - 2}\right)\right). \end{aligned}$$

As $n - |X_1| = n - rank(y_1, X)$ and $n - |X_p| = rank(y_m, X)$, we have

$$n - |X_1| - |X_p| \leq \min\{n - rank(y_1, X), rank(y_m, X)\},$$

which is at most $Max(X, Y) = k$, by Lemma 7(c). It follows that

$$T(X, Y) = O\left(\log k + (p + q - 2) \cdot \log\left(\frac{k + m}{p + q - 2}\right)\right).$$

As in (b), $p + q - 2 \leq 2m - 1$, and thus $T(X, Y) = O(m \log((m + k)/m))$, which is *Max*-optimal by Lemma 9, since $k \geq m$, by Lemma 7(c). □

5. Adaptive Mergesort

5.1. The Algorithm. Natural Mergesort [4] is a sorting algorithm that optimally adapts to the measure *Runs*, formally defined as

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

Presented with a sequence X, Natural Mergesort starts by finding the runs of the input sequence X in linear time. Then the first run is merged with the second, the third with the fourth, and so on, resulting in $\lceil Runs(X)/2 \rceil$ longer runs. These longer runs are then repeatedly merged pairwise until there is just one run left, which is the sorted sequence. It is easy to see that Natural Mergesort sorts a sequence X of length n with $Runs(X) = k$ in $O(n \log k)$ time. Moreover, Mannila [5] proved that this is optimal with respect to *Runs*.

What differs Adaptive Mergesort from Natural Mergesort is how the merges are implemented. In Natural Mergesort any linear-time merging algorithm can be used, while in Adaptive Mergesort we adopt the ideas from the previous section, and apply an adaptive merging algorithm, similar to Adaptmerge.

Observe that if we want to sort some sequences in $o(n \log Runs(X))$ time, while sticking to the idea of repeated pairwise merging, we cannot afford to spend constant time per participating element in the merges. Hence, the merges cannot return two sorted sequences stored consecutively in order, since that would take a linear number of assignments in each merging pass.

In Adaptive Mergesort a run is implemented by a doubly linked list Z with pointers to the first and last elements in the list. Initially, there is one linked list per run, consisting of one (list) element each, pointing out the run in the input sequence X. After some merging passes, a (long) run may consist of several list elements, each corresponding to a consecutive portion of X. The list elements are linked together in such a way that by concatenating the portions of X in the order given by Z the actual (long) run is obtained. For example, see Figure 2, which shows the situation after one merging pass. Note that after $\lceil \log Runs(X) \rceil$ passes we are finished and there is only one long run consisting of $Block(X)$ list elements, which correspond to the blocks of X.

The result of merging two runs Z_1 and Z_2 is that some list of elements are split into smaller ones and these appear in their correct positions in the list Z together with those that are not split. For example, the list element $X[9, 14]$ in

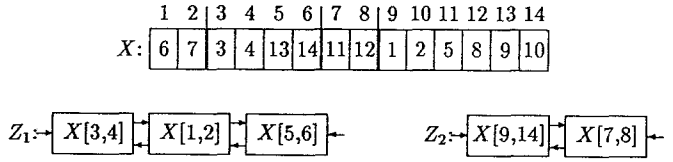

Fig. 2. Example of how the runs are implemented in Adaptive Mergesort. Z_1 corresponds to the (long) run obtained when the run in $X[1, 2]$ has been merged with the run in $X[3, 6]$.

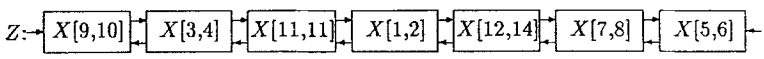

Fig. 3. The result of a merge of Z_1 and Z_2 in Figure 2.

Figure 2 is split into the three new ones $X[9, 10]$, $X[11, 11]$, and $X[12, 14]$, as shown in Figure 3, while the list element $X[3, 4]$ remains unchanged.

A merge of two runs Z_1 and Z_2, where Z_1 precedes Z_2 in X, resulting in the longer run Z, is essentially done list element by list element. To start the merge we find where in Z_1 the first list element of Z_2 fits by a backward sequential search in Z_1. We then merge the linked lists based on the first and last elements in the portions of X which correspond to the list elements, in the obvious way, and search *inside* some portion only when it has to be split. Whenever splitting a list element, the position in which to split is sought by a two-way exponential and binary search, in the same way as the first step of Adaptmerge in Section 4.1. When one of the runs is finished, we let the output run Z end with the remaining sublist of the nonfinished run, without visiting the list elements.

The following slightly less informal description of a merge of two long runs Z_1 and Z_2 in Adaptive Mergesort is similar to that of Adaptmerge given in Section 4.1. Let $Z_1 \circ Z_2$ denote concatenating the linked lists Z_1 and Z_2. Further, let *Split*$(x, Z, firstZ, lastZ)$ be a procedure that first performs a sequential search forward in Z to find the list element into which x fits. Second, it splits that list element by a two-way exponential and binary search, and returns *firstZ* and *lastZ*, the sublists of Z preceding and following the splitting position, respectively.

```
function Merge(Z1, Z2: list of portions): list of portions;
  if |Z1| = 0 then Merge: = Z2
    else begin
      Split(min(Z2), Z1, firstZ1, lastZ1);
      Merge: = firstZ1 ∘ Merge(Z2, lastZ1);
    endelse;
end;
```

Recall that the first time *Split* is invoked, the search for the correct list element is done backwards and not forwards.

5.2. *Upper Bounds*

Max-*optimality.* In the following it is helpful to think of Adaptive Mergesort as performing the merges in $\lceil \log Runs(X) \rceil$ passes.

We divide the analysis into two parts; the first $\lceil \log Max(X) \rceil$ passes, and the remaining passes. The first $\lceil \log Max(X) \rceil$ passes take $O(n \log Max(X))$ time, since at most linear time is spent in each pass. We show that the remaining passes take linear time altogether. To prove this we need two lemmas.

LEMMA 13. *Sorting a consecutive subsequence of a sequence does not increase the value of Max.*

PROOF. Let X be a sequence with $Max(X) = k$. Suppose we sort a consecutive subsequence of a sequence X. Any such sort can be viewed as a sequence of swaps of pairs of adjacent elements that are unordered (cf. Bubblesort). If we show that no swap increases the value of Max, the lemma follows. Assume that we swap the elements x_j and x_{j+1}, that is, $x_j > x_{j+1}$, and let X' be the resulting sequence. Since the swap does not affect other elements than x_j and x_{j+1}, Definition 1 gives

$$Max(X') \leq \max\{k, |j - \pi(j+1)|, |j+1 - \pi(j)|\}.$$

The only way for $Max(X')$ to exceed k is if one of the last two terms is greater than k. Without loss of generality, assume that $|j+1-\pi(j)| > k$. Since $|j - \pi(j)| \leq k$ and $|j+1-\pi(j)| > k$, $|j+1-\pi(j)| = j+1-\pi(j)$, but as $x_j > x_{j+1}$, we have that $\pi(j) > \pi(i+1)$, and thus

$$j+1-\pi(j+1) > j+1-\pi(j) > k.$$

This contradicts that $Max(X) = k$, because $Max(X) \geq j+1-\pi(j+1)$. □

LEMMA 14. *Any merge in Adaptive Mergesort runs in $O(Max(X))$ time.*

PROOF. Let Z_1 and Z_2 be two (long) runs that are merged in Adaptive Mergesort. Let X_1 and X_2 be the sorted consecutive subsequences of X corresponding to Z_1 and Z_2, respectively. Then $Max(X_1X_2) \leq Max(X)$, by Lemma 13. The lemma follows if we prove that Z_1 and Z_2 are merged in $O(Max(X_1X_2))$ time.

Let $|X_1| = n_1$ and $|X_2| = n_2$. Further, let $X_1(n_1)$ be the greatest element in X_1 and let $X_2(1)$ be the smallest element in X_2. A merge of Z_1 and Z_2 starts by finding the correct position for $X_2(1)$ in X_1 by a backward sequential search in Z_1. Once the correct list element in Z_1 is found, the exact position is found by a two-way exponential and binary search. In total, this takes $O(n_1 - rank(X_2(1), X_1))$ time. After that, only the last $n_1 - rank(X_2(1), X_1)$ elements in X_1 take part in the merge. Each following search splits off a portion of either Z_1 or Z_2 in time at most linear in the number of X-elements split off. Furthermore, the merge stops after having considered the first $rank(X_1(n_1), X_2)$ elements in X_2, because at that point Z_1 is finished. Hence, the total time spent is

$$O(n_1 - rank(X_2(1), X_1) + rank(X_1(n_1), X_2)),$$

which is $O(Max(X_1X_2))$, by Lemma 7(a). □

Now it is easy to prove that all passes in Adaptive Mergesort except the first $\lceil \log Max(X) \rceil$ take linear time altogether. If there are no remaining passes, we are done. Otherwise, since there are at most $n/2^j$ merges performed in the jth pass,

$j \geq \lceil \log Max(X) \rceil + 1$, and as each merge takes $O(Max(X))$ time, by Lemma 14, the remaining passes run in time

$$O\left(\sum_{j=\lceil \log Max(X)\rceil+1}^{\lceil \log Runs(X)\rceil} \frac{n}{2^j} \cdot Max(X)\right) = O\left(\sum_{j=1}^{\infty} \frac{n}{Max(X) \cdot 2^j} \cdot Max(X)\right)$$
$$= O\left(\sum_{j=1}^{\infty} \frac{n}{2^j}\right) = O(n).$$

By the above discussion and Lemma 2, we can conclude

THEOREM 15. *Adaptive Mergesort sorts a sequence X of length n for which $Max(X) = k$ in time $O(n \log k)$, which is optimal with respect to Max.*

5.2.1. Block-*optimality.* In order to prove that the time consumed by Adaptive Mergesort matches Lemma 6 we again divide the analysis into two parts. This time, however, not by passes, but by different kinds of operations. First, we estimate the time spent on traversing the linked lists. Second, we examine the time spent on searches *in* X, when splitting list elements.

Initially, there are $Runs(X)$ linked lists containing one element each. As the algorithm proceeds the total length of the linked lists is increasing and reaches $Block(X)$ when finished. In each merging pass no link in a linked list is traversed more than twice (once in the initial search backward and once in the following merge). Hence, even if the total length of the linked lists is $Block(X)$ in the first pass, the cost for all linked list traversals is bounded by $O(Block(X) \log Runs(X))$.

When analyzing the time spent on searches for splitting list elements, let $Block(X) = k$. The number of searches performed is $k - Runs(X)$, since each initial run, except one, defines a splitting position which we do not have to search for. As $Runs(X) \geq 1$, this number is at most $k - 1$. Note that what the searches do is splitting an array of length n into k portions. Since the searches are exponential and binary and both forward and backward, the position sought is found in $O(\log \ell)$ time if it is within distance ℓ from the left or right boundary of the portion considered. Since we want an upper bound on the time spent by all searches, we might as well assume that each search takes exactly $c \cdot \log \ell$ time, for some constant $c > 0$, although some searches may be faster.

LEMMA 16. *For $i \geq 1$, let k_i denote the number of searches which take at least $c \cdot \log(n/2^i)$ time and strictly less than $c \cdot \log(n/2^{i-1})$ time. Then $k_i \leq 2^i - 1$.*

PROOF. A search which takes at least $c \cdot \log(n/2^i)$ time and strictly less than $c \cdot \log(n/2^{i-1})$ time cuts off a portion of X of length at least $n/2^i$ and less than $n/2^{i-1}$. Any future search in this portion runs in time less than $c \cdot \log(n/2^i)$, and can thus not contribute to k_i. We therefore maximize k_i if each search contributing to it cuts off as small a portion of X as possible, that is, a portion of length $n/2^i$. As this can be done at most $2^i - 1$ times, the lemma follows. □

The time consumed by all searches is thus bounded from above by

$$c \sum_{i=1}^{r} (2^i - 1) \cdot \log\left(\frac{n}{2^{i-1}}\right),$$

where the upper limit r is chosen so that $\sum_{i=1}^{r}(2^i - 1) \geq k - 1$, which is satisfied for $r = \log k$. Now,

$$\begin{aligned} c \sum_{i=1}^{\log k} (2^i - 1) \cdot \log\left(\frac{n}{2^{i-1}}\right) &\leq c \sum_{i=1}^{\log k} 2^i \cdot \log\left(\frac{n}{2^{i-1}}\right) \\ &= \sum_{j=0}^{\log k - 1} \frac{k}{2^j} \cdot \log\left(\frac{n \cdot 2^{j+1}}{k}\right) \\ &= c \sum_{j=0}^{\log k - 1} \frac{k}{2^j} \cdot \log\left(\frac{n}{k}\right) + c \cdot k \sum_{j=0}^{\log k - 1} \frac{j+1}{2^j} \\ &= c \cdot k \log\left(\frac{n}{k}\right) \sum_{j=0}^{\log k - 1} \frac{1}{2^j} + c \cdot k \sum_{j=0}^{\log k - 1} \frac{j+1}{2^j} \\ &\leq 2c \cdot k \log\left(\frac{n}{k}\right) + 4c \cdot k, \end{aligned}$$

which is $O(k \log(n/k))$.

Hence, the time spent on splitting list elements in Adaptive Mergesort is $O(Block(X) \log(n/Block(X)))$. If we add the time required for finding the runs and the time spent on linked list traversals, we have

THEOREM 17. *Adaptive Mergesort sorts a sequence X of length n for which $Block(X) = B$ and $Runs(X) = R$ in time $O(n + B \log R + B \log (n/B))$, which is optimal with respect to Block and Runs.*

PROOF. To see that the upper bound matches the lower bound in Lemma 6, observe that the second term is at most $B \log B$, because $Runs(X) \leq Block(X)$, and the third term is at most linear. Furthermore, as $Block(X) \leq n$, it is bounded by $O(n \log R)$, which is *Runs*-optimal, as proved by Mannila [5]. □

6. A Comparison with other Measures. We study the relation between the new measure *Block* and other measures of presortedness proposed in the literature, using the framework introduced by Petersson and Moffat [6]. They compared measures based on the relation between the values of $C_M(n, k)$, as follows:

LEMMA 18. *Let M_1 and M_2 be measures of presortedness. If*

$$C_{M_1}(|X|, M_1(X)) = O(C_{M_2}(|X|, M_2(X))),$$

then every M_1-optimal sorting algorithm is M_2-optimal.

PROOF. Immediate from Definition 4. □

To use this lemma, we need a method to derive upper bounds on $C_M(n, k)$, which can be obtained from upper bounds on sorting algorithms:

LEMMA 19. *Let M be a measure of presortedness, for which $C_M(n, k) = \Omega(f(n, k))$, where f is nondecreasing on k. If there exists a comparison-based sorting algorithm S that sorts a sequence X in $C_S(X) = O(f(|X|, M(X)))$ comparisons, then $C_M(n, k) = \Theta(f(n, k))$.*

PROOF. As S defines a comparison tree in T_n, the definition of $C_M(n, k)$ gives

$$\begin{aligned} C_M(n, k) &\le \max_X \{C_s(X) \mid |X| = n \text{ and } M(X) \le k\} \\ &= O\left(\max_{0 \le j \le k} \{f(n, j)\}\right) \\ &= O(f(n, k)), \end{aligned}$$

since f is nondecreasing on j. □

In Section 5.2 we showed that Adaptive Mergesort runs in time $O(n + Block(X) \log Block(X))$. Furthermore, as $C_{Block}(n, k) = \Omega(n + k \log k)$ by Lemma 6, we conclude that $C_{Block}(n, k) = \Theta(n + k \log k)$ by Lemma 19. Applying Lemmas 4 and 5 now gives

$$\begin{aligned} C_{Block}(|X|, Block(X)) &= \Theta(|X| + Block(X) \log Block(X)) \\ &= O(|X| + Rem(X) \log Rem(X)) \\ &= O(C_{Rem}(|X|, Rem(X))), \end{aligned}$$

and hence, by Lemma 18,

THEOREM 20. *Every Block-optimal sorting algorithm is Rem-optimal.*

Consider the sequence

$$X = \left\langle \frac{n}{2} + 1, \frac{n}{2} + 2, \ldots, n, 1, 2, \ldots, \frac{n}{2} \right\rangle,$$

for which $Rem(X) = n/2$ and $Block(X) = 2$. To sort X, any *Rem*-optimal sorting algorithm may use $C_{Rem}(n, Rem(X)) = \Omega(n \log n)$ time, while any *Block*-optimal one completes in linear time. Hence, a *Rem*-optimal sorting algorithm is not necessarily *Block*-optimal.

Theorem 20 should be interpreted as follows. From a sorting algorithmic point of view, *Block* is superior to *Rem*. That is, instances which are "easy" to sort

according to *Rem* are easy to sort according to *Block* as well. Therefore, in this sense *Block* generalizes *Rem*.

Next, we take the opportunity to show the relation between *Rem* and a measure of presortedness for which until this paper no algorithm was known to be optimal with respect to, namely, *Exc*, defined as the minimum number of exchanges of arbitrary elements needed to bring a sequence into sorted order [5].

LEMMA 21. *For any sequence* X, $Rem(X) \leq 2 \cdot Exc(X)$.

PROOF. Let X be a sequence of length n, and consider a sequence of $Exc(X)$ exchanges which brings the sorted permutation of X into X. Each exchange performed removes at most two elements from a longest ascending subsequence in the current sequence. Since the length of a longest ascending subsequence is initially n, its length in X is at least $n - 2 \cdot Exc(X)$. The definition of *Rem* now gives that $Rem(X) \leq 2 \cdot Exc(X)$. □

LEMMA 22. $C_{Exc}(n, k) = \Omega(n + k \log k)$.

PROOF. As usual, we estimate the size of $below_{Exc}(n, k)$ from below. Consider any sequence of at most k exchanges performed on the first $k + 1$ elements of the identity permutation in S_n. Each such sequence of exchanges defines a permutation π in $below_{Exc}(n, k)$. Furthermore, any permutation of the $k + 1$ elements can be produced by performing at most k exchanges, and so $\|below_{Exc}(n, k)\| \geq (k + 1)!$. Theorem 1 now gives

$$C_{Exc}(n, k) = \Omega(n + \log\|below_{Ecx}(n, k)\|) = \Omega(n + k \log k),$$

which completes the proof. □

An argument analogous to that in the proof of Theorem 20 gives

THEOREM 23. *Every Rem-optimal sorting algorithm is Exc-optimal.*

For the sequence $X = \langle n, 1, 2, \ldots, n - 1\rangle$, $Exc(X) = n - 1$ and $Rem(X) = 1$, showing that Theorem 23 is not true the other way around.

As Mannila's Local Insertion Sort [5] is *Rem*-optimal, Theorem 23 affirmatively answers the question of whether Local Insertion Sort is *Exc*-optimal or not (see discussion in [5]).

Combining Theorems 15, 17, 20 and 23 gives

COROLLARY 24. *Adaptive Mergesort is optimal with respect to the measures Max, Block, Rem, Exc, and Runs.*

7. Concluding Remarks. We would like to mention that the observation that some instances of the merging problem are easier than others was also made by

Mannila and Ukkonen [9]. They evaluated the easiness by the number of inversions in the sequence obtained by concatenating the input sequences.

Results related to those reported in this paper have independently been achieved by Moffat [10]. However, his algorithm is not *Block*-optimal.

References

[1] K. Mehlhorn. *Data Structures and Algorithms*, Vol. 1. Springer-Verlag, Berlin, 1984.

[2] W. H. Burge. Sorting, trees, and measures of order. *Information and Control*, 1(3):181–197, 1958.

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

[4] D. E. Knuth. *The Art of Computer Programming*, Vol. 3. Addison-Wesley, Reading, Mass., 1973.

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

[6] O. Petersson and A. Moffat. A framework for adaptive sorting. In *Proceedings of the Third Scandinavian Workshop on Algorithm Theory*, pp. 422–433. Lecture Notes in Computer Science, Vol. 621, Springer-Verlag, Berlin, 1992.

[7] M. L. Fredman. How good is the information theory bound in sorting? *Theoretical Computer Science*, 1:355–361, 1976.

[8] O. Petersson. Adaptive Sorting. Ph.D. thesis, Department of Computer Science, Lund University, Lund, 1990.

[9] H. Mannila and E. Ukkonen. A simple linear-time algorithm for in situ merging. *Information Processing Letters*, 18(4):203–208, 1984.

[10] A. Moffat. Adaptive merging and a naturally Natural Merge Sort. In *Proceedings of the 14th Australian Computer Science Conference*, pp. 08.1–08.8, 1991.