Acta Informatica 35, 775–793 (1998)

© Springer-Verlag 1998

A Tree-based Mergesort*

Alistair Moffat[1], **Ola Petersson**[2], **Nicholas C. Wormald**[3]

[1] Department of Computer Science, The University of Melbourne, Parkville 3052, Australia
(e-mail: alistair@cs.mu.oz.au)
[2] Department of Computer Science, Lund University, Box 118, S–221 00 Lund, Sweden; and Department of Mathematics, Statistics, and Computer Science, Växjö University, S-351 95 Växjö, Sweden
(e-mail: ola@dna.lth.se)
[3] Department of Mathematics, The University of Melbourne, Parkville 3052, Australia
(e-mail: N.Wormald@ms.unimelb.edu.au)

Received: 7 October 1993 / 18 July 1996

Abstract. We demonstrate that if standard Mergesort is implemented using finger search trees instead of arrays it optimally adapts to a set of measures of presortedness not fulfilled by any other algorithm.

1. Introduction

An *adaptive* algorithm is one which uses fewer resources to solve 'easy' problem instances than it does to solve 'hard' ones. For sorting, an adaptive algorithm should process an n-sequence in $O(n)$ time if the sequence is sorted, and preferably in $O(n \log n)$ time on any sequence, with the time for each particular sequence depending upon the 'nearness' of that sequence to being sorted.

Mannila [8] established the notion of a *measure of presortedness* to quantify the disorder in any input sequence, and introduced the concept of *optimal adaptivity*. For example, if presortedness is measured by counting the number of pairwise inversions in the input n-sequence X, denoted $Inv(X)$, then an Inv-optimal algorithm must sort X in[1] $O(n \log(Inv(X)/n))$ time [8]. The intuitive reason why this is optimal is that *any* binary comparison-tree that sorts *every* n-sequence X with $Inv(X) \leq k$ has height $\Omega(n \log(k/n))$. Several authors have proposed other measures of presortedness, some of which are defined as we go along.

Any particular sorting algorithm is optimally adaptive with respect to some subset of this set of known measures of presortedness. Of great interest then is the determination of the exact subset attained by an algorithm. The more all-inclusive the subset, the more versatile the algorithm. For example a method that

* Preliminary versions of the results reported in this paper have been presented at *14th Australian Computer Science Conf.*, 1991; *15th Australian Computer Science Conf.*, 1992; and *Third Ann. Internat. Symp. on Algorithms and Computation*, 1992.

[1] We assume that $\log x$ is defined to be $\log_2(\max\{2, x\})$.

is not only optimal with respect to *Inv* but also with respect to *Runs*, the number of ascending runs in the input sequence (that is, the algorithm sorts n-sequences composed of k runs in $O(n \log k)$ time), is in some sense more powerful than methods that are optimally adaptive to only one of the measures.

In this paper we study the adaptivity of a Mergesort, that is, in what ways it profits from existing order within its input. It is easy to see that a textbook Mergesort runs in $\Theta(n \log n)$ time not only in the worst case but also in the best case, and so is not adaptive at all. This lack of adaptivity stems from the fact that no matter how 'easy' a single merge is, it still takes linear time in the number of participating items. We show that by implementing the algorithm using finger search trees rather than arrays, merging can be accomplished adaptively, that is, some merges can take sublinear time. This *Tree-based Mergesort* is not tailored to profit from any particular kind of presortedness, and it is therefore surprising that it is optimally adaptive with respect to a wide range of known measures. The main part of the paper is indeed devoted to proving this claim. Results by the first two authors imply that it is not necessary to investigate adaptivity with respect to all measures of presortedness, and that it suffices instead to consider only a few, from which optimality (or non-optimality) with respect to many other measures comes for free [11].

In Sect. 2 we define the notion of optimal adaptivity. Section 3 then presents an adaptive merging algorithm, and shows how it can be implemented using finger search trees. In the subsequent three sections the adaptivity of the resulting Mergesort is analysed in terms of three different measures of presortedness. In Sect. 4 we prove *Inv*-optimality; in Sect. 5 *Rem*-optimality is proved, where *Rem* measures the minimum number of items that must be removed to leave a sorted sequence; and in Sect. 6 *SMS*-optimality is examined, where *SMS* is the minimum number of monotone subsequences whose shuffle forms the input sequence. In Sect. 7 it is shown that the obtained bounds imply optimality for other measures too. An example of a measure for which the algorithm is not optimal is also given. Section 8 gives experimental results that are in agreement with the analysis of Sects. 4, 5, and 6.

Finally, note that we use Θ, Ω, and O to indicate bounds on functions in the manner described by Knuth [5].

2. Optimal adaptivity

The concept of an optimal algorithm with respect to a measure of presortedness was given in a general form by Mannila [8]. In the development use the following equivalent definitions [11].

Definition 1. *Let M be a measure of presortedness. Then, for any $k \geq 0$ and $n \geq 1$,*

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

The set $below_M(n,k)$ contains all permutations that, according to M, are at least as presorted as the sequences X with $M(X) = k$.

The minimum number of comparisons needed to sort the permutations in a *below*-set is also of interest:

Definition 2. *Let M be a measure of presortedness, and T_n the set of binary comparison trees for the set S_n of all permutations of $\{1, 2, \ldots, n\}$. Then, for any $k \geq 0$ and $n \geq 1$,*

$$C_M(n,k) = \min_{T \in T_n} \max_{\pi \in below_M(n,k)} \{depth\ of\ \pi\ in\ T\}.$$

Given these two definitions, the notion of optimality of an adaptive sorting algorithm is clear:

Definition 3. *Let M be a measure of presortedness, and A 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 $T_A(X) = O(C_M(|X|, M(X)))$.*

The following theorem is helpful when proving the optimality of some algorithm [11]:

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

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

This paper considers in detail three measures of presortedness: *Inv*, the number of inversions (pairs of items out of order with respect to each other) in the input sequence; *Rem*, the minimum number of items which must be removed in order to leave a sorted sequence; and *SMS*, the minimum value k such that the sequence can be composed by shuffling together k monotone sequences. Table 2 lists the values of $C_M(n,k)$ for these three measures.

Table 1. Values of $C_M(n,k)$

M	$C_M(n,k)$	Reference
Inv	$\Theta(n \log(k/n))$	[8]
Rem	$\Theta(n + k \log k)$	[8]
SMS	$\Theta(n \log k)$	[6]

3. Adaptive merging and Tree-based Mergesort

The standard linear time algorithm for merging two sorted sequences can be thought of as first finding the maximum of the two smallest items and then performing a linear search for its successor in the opposite sequence. Items smaller than the successor are transferred to the output as they are passed by the search. This process is then repeated with the roles of the two sequences reversed, continuing until one or the other of the two sequences is empty.

Due to the linear search the algorithm spends a linear number of comparisons even in the best case. The reason why the search chosen *is* linear is that the sequences are usually implemented using arrays, in which case it takes linear

time to move the output, and so, why worry about saving some comparisons? For now, let us only count comparisons, and see what can be achieved. This simplification will be justified shortly.

The next search that comes to mind after linear search is, of course, binary search. In its normal form binary search is not, however, an attractive alternative, since it might result in an algorithm requiring logarithmic time per item rather than constant. A better choice is exponential and binary search [9]. Recall that this algorithm finds the successor of an item in a sorted sequence by starting at one of the endpoints and probing further and further towards the opposite endpoint, each time doubling the distance, until the target position has been enclosed, after which it switches to an ordinary binary search. For example, a search from location 1 for an item in location 20 of a sorted array probes locations 1, 3, 7, 15, and 31, at which point the search changes from being exponential to binary; and then locations 23, 19, 21, and, finally, 20. More generally, if the sought position is at distance d from the starting point of the search, it is found in time $O(\log d)$. Using this search, chunks of consecutive items that do not interleave with the other sequence are thus traversed in logarithmic time rather than linear.

The merge process can be thought of as cutting the two sequences into the minimum number of consecutive subsequences or *blocks* necessary to achieve the merge, and then interleaving the two sets of blocks to form the sorted output. In what follows, suppose that the merged output sequence

$$Z = X_1 Y_1 X_2 Y_2 \cdots X_{b-1} Y_{b-1} X_b Y_b$$

consists of b such blocks from each of the input n-sequence X and the input m-sequence Y, $m \leq n$. Note that either or both of X_1 and Y_b may be empty. Given the blocks, computing the sequence Z requires no comparisons at all, and so, using exponential and binary search, the total number of comparisons becomes

$$O\left(\sum_{i=1}^{b} \log |X_i| + \sum_{i=1}^{b} \log |Y_i|\right) = O\left(b \log \frac{n}{b}\right) = O\left(m \log \frac{n}{m}\right).$$

Note that this bound matches the worst-case lower bound for merging, that is, $\lceil \log_2 \binom{n+m}{m} \rceil = \Omega(m \log(n/m))$. We now show that the above bound can also be achieved when operations other than comparisons are also counted.

A *finger search tree* [9] is a data structure for representing a sorted sequence. Among the operations supported is searching from a 'finger' that points into the sequence in time logarithmic in the 'distance' from the finger, that is, exponential and binary search. There are several different ways of implementing finger search trees, but for definiteness, our discussion assumes level linked 2–3 trees [2]; however, the results extend to other implementations too. Indeed, our investigation was prompted by the merging algorithm given by Pugh [12] for Skip Lists, a probabilistic finger search tree.

Suppose then that X and Y are sorted sequences of n and m items respectively, stored as level linked 2–3 trees. Further, suppose that they are to be merged, generating an output tree Z of $n + m$ items. In what follows, if the

largest item in X is no greater than the smallest item in Y the trees are said to be *disjoint*, denoted $X \leq Y$. We first consider the cost of splitting the trees X and Y into disjoint components that can be joined to form their merged output, and then consider the complexity of joining a list of b disjoint 2–3 trees.

Theorem 2. *Let X be a 2–3 tree, which is to be split into b disjoint 2–3 trees X_i, $1 \leq i \leq b$, determined by the sequence $\langle -\infty, y_1, y_2, \ldots, y_{b-1}, +\infty \rangle$. That is, the items in X_i are to have values that are greater than y_{i-1} and less than or equal to y_i. Then the splitting can be performed in $O(\sum_{i=1}^{b} \log |X_i|)$ time.*

Proof. The sequence of operations considered is:

```
for i := 1 to b do
    X_i := Split(X, y_i)
endfor
```

where it is assumed that $Split(X, y_i)$ prunes from X all items less than or equal to y_i and returns them as a separate 2–3 tree, leaving X as a depleted 2–3 tree.

To locate the last node to be pruned a finger search for y_i is performed from the leftmost node of (what remains of) X, which takes $O(\log |X_i|)$ time. Then, to actually perform the *Split*, start at the parent node of the rightmost item to be included in X_i and work up the tree, duplicating every ancestor until either the leftmost or the rightmost branch of X is reached. One of each of these pairs of nodes, together with all of the items to the left of the duplicated node, eventually forms the tree X_i. The other node of the pair, and items to the right of it, remains in X. At each level the children of each duplicated node are partitioned between the original node and the duplicate node according to their range of key values, and the level links adjusted accordingly.

This results in two trees, X_i and X, which are 2–3 trees, *except* there may be violations down the right branch of X_i and down the left branch of X. For each of the at most $2 \log_2 |X_i|$ nodes thus affected a distribution step is performed, starting from the parent nodes of the leaves of the tree:

- if the node (after the children have been partitioned) has zero children, or if it is the root node of either X_i or X and has only one child, it is deleted;
- if the node has one child, and the node immediately to the left (if it is in X_i) or right (if it is in X) has three children, a child is adopted from the sibling;
- if the node has one child, and the node immediately to the left (if it is in X_i) or right (if it is in X) has only two children, the child is moved to the sibling, and the node is deleted;
- if the node has two or three children, no action is required.

This sequence of steps takes $O(1)$ time at each node along the cutting path from the leaf level up to either the left or right branch of X. The length of that cutting path is $O(\log |X_i|)$, and so the unzipping phase costs $O(\log |X_i|)$ time.

Then, at the node on the left or right branch that is at the top of the cutting path, a child must be removed to form the root of the new tree X_i (when on the

left branch) or the root of the remainder of the tree X (when on the right branch). We call this operation a *branch deletion*.

The removal of this child during a branch deletion might cause a violation of the 'at least two children' rule, and a sequence of cascading node fusings that reaches right up to the root of the larger tree. The total number of node fusings over all *Split* operations is, however, bounded by the sum of the number of branch deletions plus the total length of the right and left branches of the trees in the resulting forest, and so the result follows. □

In the case where $b = 2$ this yields the standard result that it requires at most $O(\log n + \log m) = O(\log n)$ time to split a tree into two parts of size n and m.

Joining b components is no more expensive than splitting:

Theorem 3. *A list $Z_1, Z_2, \ldots, Z_b$ of disjoint level-linked 2–3 trees, where $Z_i \leq Z_{i+1}$, for $1 \leq i < b$, can be concatenated to make a single level-linked 2–3 tree in $O(\sum_{i=1}^{b} \log |Z_i|)$ time.*

Proof. The sequence of operations considered is given by

$Z := \emptyset$
for $i := 1$ **to** b **do**
    $Z := Join(Z, Z_i)$
endfor

where to *Join* two trees Z and Z_i we begin at the rightmost node of Z and the leftmost node of Z_i and *zip* the two trees together, setting the level links, until either the root of Z or the root of Z_i is reached. In either case the time required during the zipping stage is at most proportional to the depth of Z_i, which is $O(\log |Z_i|)$.

The root of Z_i is then inserted as an additional rightmost child of the corresponding node on the right branch of Z, or, if it was the root of Z that was reached first, Z is inserted as a leftmost child of the appropriate node on the left branch of Z_i—that is, a *branch insertion* takes place. If this causes that node to have more than three children, it is split and a new child added to the parent of that node, and so on up either the right branch of Z or the left branch of Z_i. If the root is split a new root is added, and Z grows by one level.

The cost of all of the zipping operations is $O(\sum_{i=1}^{b} \log |Z_i|)$. The node splittings that take place during the b branch insertions must also be accounted for. The splitting of a node that became a 3-node as a result of a previous branch insertion can be charged as an $O(1)$ overhead to that operation, since each such insertion creates at most one 3-node. Pre-existing 3-nodes that cannot be charged in this way might also be split, but any such nodes must lie on either a left branch or a right branch of one of the original input trees, and since the total number of such nodes is $O(\sum_{i=1}^{b} \log |Z_i|)$, we have, summed over the sequence of operations, that the cost of the node splittings does not dominate the cost of zipping the trees together. □

Again, in the case when $b = 2$ this gives the familiar result that a *Join* of two 2–3 trees can be performed in $O(\log n + \log m) = O(\log n)$ time [9].

Using trees the merging algorithm can be described as an interleaved sequence of *Split*s and *Join*s:

```
function Merge(X:tree; Y:tree):tree
    Z := ∅; i := 0
    while X ≠ ∅ or Y ≠ ∅ do
        i := i + 1
        X_i := Split(X, y_i)
        Z := Join(Z, X_i)
        Y_i := Split(Y, x_i)
        Z := Join(Z, Y_i)
    endwhile
    return Z
end
```

Here x_i and y_i, the sequences of splitting items, are always the smallest remaining items in the (shrinking) trees X and Y; and, for descriptive purposes, it is assumed that reference to the first item of an empty tree returns the value $+\infty$.

Note that each of the trees involved is subject to either *Join* or *Split* operations, but not both. For 2–3 trees, the *Join*s and *Split*s on the same tree cannot be interleaved without affecting the given analysis, since the amortized bounds on the numbers of node splittings and fusings rely on there being no other intervening operations [9]. Theorems 2 and 3, and the observation that each tree is only subject to one type of operation now provide the following bound on the cost of merging two level-linked 2–3 trees:

Theorem 4. *Let X and Y be two sorted sequences of n and m items, respectively, $m \leq n$, which are represented as level-linked 2–3 trees. Further let the tree Z be the result of merging X and Y:*

$$Z = X_1 Y_1 X_2 Y_2 \ldots X_{b-1} Y_{b-1} X_b Y_b,$$

where $X = X_1 X_2 \ldots X_b$, $Y = Y_1 Y_2 \ldots Y_b$, and either or both of X_1 and Y_b might be empty. Then Z can be computed in

$$O\left(b + \sum_{i=1}^{b} \log |X_i| + \sum_{i=1}^{b} \log |Y_i|\right) = O\left(b \log \frac{n}{b}\right) = O\left(m \log \frac{n}{m}\right)$$

time in the worst case.

Carlsson, Levcopoulos, and Petersson [3] have also described a merging algorithm similar to this. In their solution, however, the output sequence is not in the same format as the input sequences.

The Tree-based Mergesort is now obvious. The items in the input sequence X are inserted into n singleton 2–3 trees in $\Theta(n)$ time. Next, the trees are pairwise

merged in the order they appeared in the input, building trees of size 2, then 4, then 8, and so on. After $\lceil \log_2 n \rceil$ passes a single tree remains. Finally, it takes $\Theta(n)$ time to traverse that tree and copy the items to the desired output location.

4. Inversions

We now turn our attention to the adaptivity of the new Mergesort with respect to the number of inversions, which is formally defined as

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

Over the years several sorting algorithms that adapt to *Inv* have been devised; for example, Mehlhorn's A-Sort [9] and Mannila's Local Insertion Sort [8] are both *Inv*-optimal, requiring $O(n \log(k/n))$ time to sort an n-sequence X with $Inv(X) = k$ inversions. Here it is shown that the same bound applies to the Tree-based Mergesort.

We first need the following result describing the cost of a single merge.

Lemma 5. *Let X and Y be sorted sequences of n and m items, respectively, and let I be the number of inversions in the concatenation XY:*

$$I = \|\{(i,j) \mid 1 \le i \le n,\ 1 \le j \le m, \text{ and } x_i > y_j\}\|.$$

Then X and Y can be merged in time $O(\log n + \log m + \sqrt{I})$.

Proof. As before, it is supposed that the sorted output sequence is

$$Z = X_1 Y_1 X_2 Y_2 \cdots X_{b-1} Y_{b-1} X_b Y_b,$$

where either or both of X_1 and Y_b might be empty. Also, let $n_i = |X_i|$ and $m_i = |Y_i|$, $1 \le i \le b$. Then, by Theorem 4, Z can be computed in time

$$O\left(b + \sum_{i=1}^{b} \log n_i + \sum_{i=1}^{b} \log m_i\right) = O\left(b + \log n + \log m + \sum_{i=1}^{b-1} \log(m_i \cdot n_{i+1})\right)$$

The number of inversions equals

$$I = \sum_{i=1}^{b-1} \left(m_i \cdot \sum_{j=i+1}^{b} n_j\right).$$

To prove the lemma it is sufficient to show that $T^2 = O(I)$, where

$$T = b + \sum_{i=1}^{b-1} \log(m_i \cdot n_{i+1}).$$

Without loss of generality (we simply exchange the roles of X and Y if it is not the case) it may be assumed that

$$T \leq b + 2\sum_{i=1}^{b-1} \log m_i .$$

Making the substitution $m_i = k_i b/(b-i)$ yields

$$\begin{aligned} T &\leq b + 2\sum_{i=1}^{b-1} \log k_i + 2\sum_{i=1}^{b-1} \log(b/(b-i)) \\ &= O(b) + 2\sum_{i=1}^{b-1} \log k_i . \end{aligned}$$

In order to bound T^2 from above use is made of the simple fact that

$$\left(\sum_{i=1}^{p} a_i\right)^2 \leq p\sum_{i=1}^{p} a_i^2,$$

for any positive integer p, which follows from the Cauchy-Schwarz inequality. Setting $p = 2$ gives

$$T^2 \leq O(b^2) + 4 \cdot \left(\sum_{i=1}^{b-1} \log k_i\right)^2 .$$

The summation on the right hand side can similarly be manipulated by setting $p = b - 1$ to yield

$$\begin{aligned} T^2 &\leq O(b^2) + 4 \cdot (b-1) \cdot \sum_{i=1}^{b-1} (\log k_i)^2 \\ &= O\left(b^2 + \sum_{i=1}^{b-1} b \cdot k_i\right) \\ &= O\left(b^2 + \sum_{i=1}^{b-1} m_i \cdot (b-i)\right) \end{aligned}$$

The sum is bounded by I, because $m_i \geq 1$ for $1 \leq i \leq b-1$ implies that $I = \Omega(b^2)$, and because

$$I \geq \sum_{i=1}^{b-1} \left(m_i \cdot \sum_{j=i+1}^{b} 1\right) = \sum_{i=1}^{b-1} m_i \cdot (b-i).$$

Hence, both terms are $O(I)$, and the lemma follows. □

It is now possible to show that Tree-based Mergesort is *Inv*-optimal:

Theorem 6. *Tree-based Mergesort sorts any n-sequence X with $Inv(X) = k$ in time $O(n \log(k/n))$, which is optimal with respect to Inv.*

Proof. Denote the time consumed by $T(n,k)$. First, note that Tree-based Mergesort is worst-case optimal, that is, it runs in $O(n \log n)$ on any input. Second, let $M(n/2, I)$ be the time required to merge two sorted sequences of $n/2$ items each with I inversions by Lemma 5. Then there is a constant $c > 0$ such that

$$\begin{aligned} T(n,k) &\leq c \cdot n \log n \\ M\left(\frac{n}{2}, I\right) &\leq c \cdot (\log n + \sqrt{I}) \end{aligned}$$

We show by induction on n that there exists a function $f(n) > 0$ such that

$$T(n,k) \leq 3cn\left(1 + \log(1 + \frac{k}{n})\right) - f(n).$$

For convenience, $f(n)$ is not specified until later. Two cases are considered separately: $I \leq n^{3/2}$ and $I > n^{3/2}$.

Suppose first that $I \leq n^{3/2}$. Recall that in effect Tree-based Mergesort sorts X by recursively sorting the two halves X_1 and X_2 of X and then merging them. To account for these three tasks the k inversions are partitioned into three categories: those that are removed by the recursive sort of X_1; those removed by the recursive sort of X_2; and those that are removed during the final merge. Denote the numbers of inversions in these three categories by i_1, i_2, and i_3, respectively. That is, i_1 is the number of inversions on items in X_1 induced by items in X_1; i_2 is the number of inversions on items in X_2 induced by items in X_2; and i_3 is the number of inversions on items in X_2 induced by items in X_1. Then $k = i_1 + i_2 + i_3$, and, by the inductive hypothesis, the sorting time is bounded by

$$\begin{aligned} & T(n,k) \leq T\left(\frac{n}{2}, i_1\right) + T\left(\frac{n}{2}, i_2\right) + M\left(\frac{n}{2}, i_3\right) \\ \leq\ & 3c\frac{n}{2}\left(2 + \log(1 + \frac{2i_1}{n}) + \log(1 + \frac{2i_2}{n})\right) - 2f(\frac{n}{2}) + c\log n + c\sqrt{i_3}. \end{aligned}$$

By the concavity of the log-function, the first term is maximised when $i_1 = i_2 = (k - i_3)/2$, which yields

$$\begin{aligned} & T(n,k) \leq 3cn\left(1 + \log(1 + \frac{k - i_3}{n})\right) - 2f(\frac{n}{2}) + c\log n + c\sqrt{i_3} \\ =\ & 3cn\left(1 + \log(1 + \frac{k}{n})\right) + 3cn\log\frac{1 + \frac{k - i_3}{n}}{1 + \frac{k}{n}} - 2f(\frac{n}{2}) + c\log n + c\sqrt{i_3} \\ =\ & 3cn\left(1 + \log(1 + \frac{k}{n})\right) + 3cn\log\frac{n + k - i_3}{n + k} - 2f(\frac{n}{2}) + c\log n + c\sqrt{i_3}. \end{aligned}$$

In order to eliminate i_3 the terms involving i_3 are bounded above by expanding the second term:

$$\begin{aligned} 3n\log\frac{n + k - i_3}{n + k} + \sqrt{i_3} &= 3n\left(-\frac{i_3}{n + k} - \frac{i_3^2}{2(n + k)^2} - \cdots\right) + \sqrt{i_3} \\ &\leq -\frac{3ni_3}{n + k} + \sqrt{i_3}. \end{aligned}$$

This is maximised when $\sqrt{i_3} = (n+k)/(6n)$, and, since $k \leq n^{3/2}$, has a maximum value of $\sqrt{n}/12$. Hence,

$$T(n,k) \leq 3cn\left(1+\log(1+\frac{k}{n})\right) + \frac{c\sqrt{n}}{12} - 2f(\frac{n}{2}) + c\log n.$$

Now set $f(n) = c\cdot(\sqrt{n}+\log n+2)$ (which is positive for $n>0$). Then

$$\begin{aligned} T(n,k) &\leq 3cn\left(1+\log(1+\frac{k}{n})\right) + \frac{c\sqrt{n}}{12} - 2c\left(\sqrt{\frac{n}{2}}+\log\frac{n}{2}+2\right) + c\log n \\ &= 3cn\left(1+\log(1+\frac{k}{n})\right) - c\left((\sqrt{2}-\frac{1}{12})\sqrt{n}+\log n+2\right) \\ &< 3cn\left(1+\log(1+\frac{k}{n})\right) - f(n), \end{aligned}$$

as required.

The same bound must also be shown for the case when $k > n^{3/2}$. In this case the worst-case optimality of the algorithm can be directly applied to yield

$$\begin{aligned} T(n,k) &\leq cn\log n \\ &\leq 3cn\left(1+\log(1+\frac{k}{n})\right) - f(n) \end{aligned}$$

when $n \geq 1$ and $k > n^{3/2}$.

To complete the inductive proof it remains to establish a base case; this is easily done by considering $n = 1$, for which $3cn(1+\log(1+k/n)) - f(n)$ allows at least zero comparisons and the algorithm never consumes any. □

5. Out of place items

Another intuitively attractive measure of presortedness is *Rem*. For an n-sequence X, $Rem(X)$ is the minimum number of items that must be removed to leave a sorted sequence. This is equivalent to n minus the length of a longest ascending (not necessarily consecutive) subsequence in X. Cook and Kim [4] described an adaptive variant of Quicksort that uses $O(n + k\log k)$ time on average to sort a sequence with $Rem(X) = k$; and more generally Mannila showed both that Local Insertion Sort attains the same bound in the worst case and that this performance is sufficient for *Rem*-optimality [8].

Tree-based Mergesort is also *Rem*-optimal:

Theorem 7. *Tree-based Mergesort sorts any n-sequence X with $Rem(X) = k$ in time $O(n + k\log k)$, which is optimal with respect to Rem.*

Proof. By the definition of *Rem*, X has an ascending subsequence of length $n - Rem(X)$, and for the purposes of the analysis one such subsequence is assumed to be fixed. At each stage of the sorting process each item is categorized as

being either *inplace*, *stable*, or *unstable*. The items that are part of the ascending subsequence are always inplace; the remaining $Rem(X)$ items are initially unstable; and at the beginning of the sorting process there are no stable items.

Recall that sorting takes place by the building of larger and larger trees, with all initial trees consisting of a single item. An unstable item x becomes stable if there are inplace items x_p and x_q in the same tree as x, such that $x_p < x < x_q$. That is, x becomes stable when there are smaller and larger items in the same tree that were originally part of the longest ascending subsequence. Note that once an unstable item becomes stable it cannot revert to unstable.

At each intermediate stage of the sorting process each tree consists of a number (possibly zero) of unstable items, followed by a mixture of stable items and inplace items, followed by another series of unstable items. Figure 1 shows a possible pair of trees that are about to be merged. Unstable items are denoted by `u` in the first tree and `U` in the second; stable items by `s` and `S`; and inplace items by `i` and `I`. One possible outcome of the merge is also shown, and, in this example, one unstable item (the rightmost `u`) would, for the next merge, be considered to be stable. Note that during the merging there can be no interaction between the two middle sections, containing inplace and stable items, since all the inplace items form an ascending subsequence, and all stable items are surrounded by inplace items.

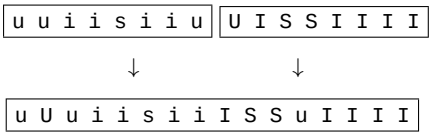

Fig. 1. Stable and unstable items

Let us now charge the cost of the various *Split* and *Join* operations involved in a single merge to the unstable items that caused them, as follows. Unstable items that remain unstable after the merge are charged $O(1)$ time, since they are simply merged in linear fashion at the beginning and end of the output tree, and the worst that can happen is that two comparisons are consumed per unstable item. Unstable items that become stable are more expensive, and are charged $O(\log n)$, since they might appear anywhere in the output tree. Finally, a single charge of $O(\log n)$ suffices to cover all of the stable and inplace items, since the cost of any *Split*s within these two blocks has already been charged to the unstable item that caused the *Split*. This final fee is paid for out of 'petty cash' rather than charged to any particular item.

Over the course of the whole sort each unstable item is charged $O(1)$ at most $\log n$ times, since it participates in $\log n$ merges; and $O(\log n)$ at most once, since it can only convert from unstable to stable once. In total, over all k initially unstable items, the total charge is $O(k \log n)$.

Some expenses have also been charged to petty cash, described by the recurrence

$$P(n) = \begin{cases} O(1) & n = 1 \\ 2 \cdot P(n/2) + O(\log n) & n > 1 \end{cases}$$

which is $O(n)$.

Thus the running time for Tree-based Mergesort on an n-sequence X with $Rem(X) = k$ is $O(n + k \log n)$. This then gives the theorem, since $n + k \log n = \Theta(n + k \log k)$. □

6. Shuffled monotone sequences

Now consider the adaptivity of Tree-based Mergesort with respect to *SMS*:

$$SMS(X) = \min\{k \mid X \text{ can be partitioned into } k \text{ monotone sequences}\}.$$

This is quite a general measure of sortedness. If we take k subsets of a random permutation; sort each into either ascending or descending order; and then interleave them in any order subject only to the constraint that the elements out of each shuffle must appear in that sorted order (either ascending or descending); we will have a sequence X for which $SMS(X) \leq k$.

In this section we prove that Tree-based Mergesort is *SMS*-optimal, running in $O(n \log k)$ time on a sequence that is the shuffle of k monotone sequences. The only other algorithm which is known to be *SMS*-optimal is Slabsort of Levcopoulos and Petersson [6], and that algorithm is not optimal with respect to *Inv*.

The analysis is in an amortized sense, making use of a potential function Φ to measure the amount of 'indebtedness' currently in the data structure. Following Tarjan [13], we define the amortized time a_i of the i'th operation to be $a_i = t_i + \Phi_i - \Phi_{i-1}$, where t_i is the actual time of the i'th operation, Φ_i is the value of the potential function after the i'th operation has taken place, and Φ_0 is the potential before any of the operations have taken place.

The worst-case time T required by a sequence of n operations is then given by $T = \sum_{i=1}^{n} t_i = \left(\sum_{i=1}^{n} a_i\right) + \Phi_0 - \Phi_n$, that is, the sum of the amortized cost of the operations plus the net decrease in potential over the whole sequence.

Suppose that $SMS(X) = k$, that is, it is possible to decompose the input sequence X into k monotone shuffles. Let $(S_1, \ldots, S_k)$ be any fixed decomposition of size k, such that item x_i is a member of shuffle S_{s_i}, $1 \leq s_i \leq k$.

We will say that x_i is *guarded* by shuffle S_j if there are items x_l and x_g such that $s_l = s_g = j$; $x_l \leq x_i \leq x_g$; and x_l, x_i, and x_g are all currently contained in the same tree. That is, x_i is guarded by a shuffle when there are both smaller and larger items from that shuffle in the same tree as x_i. Initially, when x_i is the only item in its tree, the only shuffle that it is guarded by is its own, S_{s_i}.

Define the *guardedness* g_i of any item x_i to be the number of shuffles that guard x_i. At the beginning of the sorting $g_i = 1$, for all i. During the mergings affecting x_i, g_i is non-decreasing, and at the end of the sorting $1 \leq g_i \leq k$, for all i. Finally, letting ln denote $\log_e$, take as the potential function

$$\Phi = \sum_i -c \ln g_i ,$$

where c is a constant that will be fixed below. Note that in our analysis the potential function is always negative. This in no way diminishes the validity of the amortized argument, since we are only interested in tracking relative movements in value. Suppose that Φ_i is the value of the function Φ after the ith merge takes place.

Consider one merge, in which two trees each of $n/2$ items are merged to make a single tree of n items. For the purposes of the analysis it is supposed that the least and greatest items within each shuffle are identified for each input tree. That is, the (at most) $4k$ shuffle extrema are noted. Loosely, a *slab* is defined to be the items in the input trees whose values are between an adjacent pair of these shuffle extrema, including either or both of the extremal items if they belong to shuffles that guard the slab. More precisely, if x_a and x_b are shuffle extrema and there is no shuffle extremum x_c such that $x_a < x_c < x_b$, then the slab defined by (x_a, x_b) contains all items x_i from the two inputs trees such that $x_a < x_i < x_b$, plus x_a if it is in a shuffle guarding x_b, plus x_b if it is in a shuffle guarding x_a. Thus, if any item is the only item from its shuffle in the merge, it forms a slab all by itself.

Without explicitly identifying the slabs, the main merge can be thought of as first merging the two 'sides' in the first slab; then merging the two sides in the second slab; and so on until the left and right sides of the final slab have been merged.

Let us now consider one of these 'slab merges'. Figure 2 sketches one possible configuration, with items represented by circles, shuffles by looped lines, and a shuffle that continues from the left side to the right side shown by a dotted line. Both the left sequence and the right sequence have already been ordered, and we seek at this stage to merge those two ordered lists.

Suppose that there are n' items in total within the slab, that r of them are on the right side, and, without loss of generality, that $r \leq n' - r$. By the definition of a slab, all items on the right side have the same guardedness, j say. Similarly, all items on the left side have the same guardedness, which is $k' - j$, where k' is the total number of shuffles guarding items in this slab merge. After the merge all n' items in the slab have $g_i = k'$, since all items will be guarded by all shuffles guarding any items in the slab. No items can escape this remorseless increase in guardedness.

By Theorem 4, there is a constant c such that the actual cost t_i of the slab merge is

$$t_i \leq c + c(r+1) \ln \frac{n'}{r+1},$$

since each of the two sequences is broken into at most $r + 1$ sections. The amortized cost of a slab merge is then given by

$$\begin{aligned} a_i &= t_i + \Phi_i - \Phi_{i-1} \\ &\leq c + c(r+1) \ln \frac{n'}{r+1} - cn' \ln k' + c(n'-r) \ln(k'-j) + cr \ln j . \end{aligned}$$

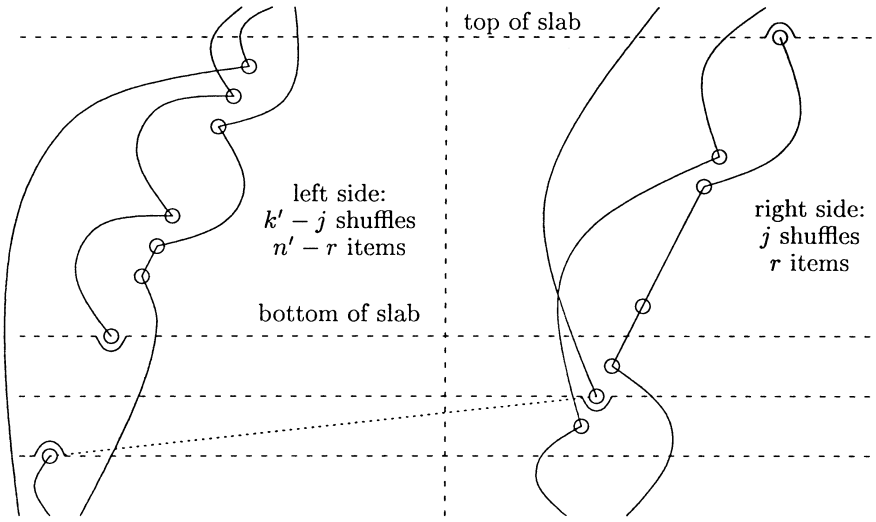

Fig. 2. Decomposition into slabs

Considered as a function of j, this is maximized when $j = rk'/n'$. Substituting for j and dividing by c gives

$$\begin{aligned}
\frac{a_i}{c} &\leq 1 + (r+1)\ln\frac{n'}{r+1} - n'\ln k' + (n'-r)\ln\frac{(n'-r)k'}{n'} + r\ln\frac{rk'}{n'} \\
&= 1 + r\ln n' + \ln n' - (r+1)\ln(r+1) - n'\ln k' \\
&\qquad +(n'-r)\ln\frac{n'-r}{n'} + (n'-r)\ln k' + r\ln r + r\ln k' - r\ln n' \\
&= 1 + \ln n' - (r+1)\ln(r+1) + (n'-r)\ln\frac{n'-r}{n'} + r\ln r \\
&\leq 1 + \ln n' + (n'-r)\ln\frac{n'-r}{n'} \\
&\leq 1 + \ln n',
\end{aligned}$$

with the last inequality following from the assumption that $(n'-r)/n' \leq 1$. That is, the amortized cost of each slab merge is logarithmic in the number of items contained in the slab.

Denote by n_i the number of items involved in the i'th slab merge when merging two trees of $n/2$ items each. The amortized cost of the merge can now be obtained by summing the amortized costs of the at most $4k$ slab merges:

$$\sum_{i=1}^{4k} c(1+\ln n_i) \leq c \cdot \min\{n, 4k(1+\ln(n/4k))\},$$

where the first alternative in the right hand side corresponds to the case $n \leq 4k$.

The amortized time for the entire *sort* is then bounded by the recurrence

$$A(n) = \begin{cases} O(1) & n \le 1 \\ 2 \cdot A(n/2) + O(n) & 1 < n \le 4k \\ 2 \cdot A(n/2) + O(k(1 + \log(n/4k))) & n > 4k \end{cases}$$

Expanding $A(n)$ in terms of $A(n/2)$, then $A(n/4)$, and so on, gives

$$\begin{aligned} A(n) &= O\left(\sum_{i=0}^{\log \frac{n}{4k}-1} k2^i \cdot \left(1 + \log\left(\frac{n}{2^i \cdot 4k}\right)\right) + \sum_{i=\log \frac{n}{4k}}^{\log n} n \right) \\ &= O\left(\left(k + k \log \frac{n}{4k}\right) \cdot \sum_{i=0}^{\log \frac{n}{4k}-1} 2^i - k \cdot \sum_{i=0}^{\log \frac{n}{4k}-1} 2^i i + n \log 4k \right) \\ &= O\left(\left(k + k \log \frac{n}{4k}\right) \cdot \frac{n}{4k} - k \cdot \frac{n}{4k} \log \frac{n}{4k} + n \log 4k \right) \\ &= O(n \log k). \end{aligned}$$

The actual time required by the sort (after $n - 1$ merges) is now

$$T(n) = A(n) + \Phi_0 - \Phi_{n-1}.$$

Since $A(n) = O(n \log k)$, $\Phi_0 = 0$, and $\Phi_{n-1} \ge -cn \ln k$, it has been proved that the algorithm runs in $O(n \log k)$ time, that is

Theorem 8. *Tree-based Mergesort sorts any n-sequence X with $SMS(X) = k$ in time $O(n \log k)$, which is optimal with respect to SMS.*

7. Other measures

To fully appreciate the adaptivity of Tree-based Mergesort it should be evaluated against the framework for adaptive sorting developed by Petersson and Moffat [11]. This framework is illustrated in the Hasse diagram of Figure 3, which illustrates a partial order on measures of presortedness.

Broadly speaking, each edge in the diagram is a containment relation on optimality, with all optimal algorithms for the higher measure automatically inheriting optimality for any connected lower measures. For example, consider the measures *Rem* and *Block*; *Block* is the number of items in a sequence that receive a new successor when the sequence is sorted [3]. The edge from *Rem* to *Block* reflects the fact that *every Block*-optimal algorithm is also *Rem*-optimal. Conversely, the presence of upward paths from *Block* to *Loc*, *Hist*, and *Reg* means that an algorithm that is not *Block*-optimal cannot be optimal with respect to any of the higher measures. For details of these measures and the exact relationship that is captured by the edges of the diagram the interested reader is referred to [11].

Each adaptive sorting algorithm corresponds to a *descriptor* line across the diagram, and the goal of the algorithm designer is to develop an algorithm that is optimally adaptive with respect to the highest possible combination of measures.

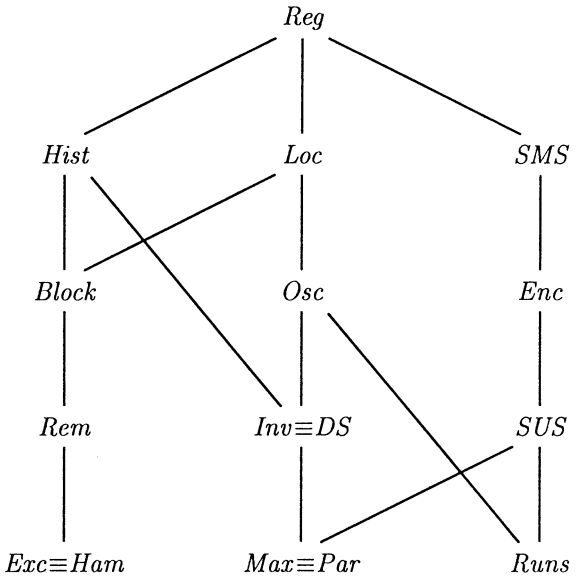

Fig. 3. Partial order on measures of presortedness

To establish lower bounds on the location of the descriptor corresponding to some algorithm proofs of optimality are needed, such as those of Theorems 6, 7, and 8. Those three theorems show respectively that the descriptor for Tree-based Mergesort crosses above *Inv*, above *Rem*, and above *SMS*.

To establish upper bounds on the location of the descriptor for an algorithm sequences must be described that are nearly sorted according to the measure, but for which the algorithm is not optimal. For example, consider the n-sequence constructed as follows. First, let

$$X_i = \langle i \log n, i \log n + 1, \ldots, (i+1)\log n - 1 \rangle,$$

for $0 \leq i < n/\log n$. Second, let X be the concatenation of a random permutation of the subsequences X_i. Then $Block(X) \leq n/\log n$, and a *Block*-optimal algorithm must sort it in $O(n + Block(X)\log Block(X)) = O(n)$ time [3]. However, on this sequence Tree-based Mergesort uses $\Theta(n \log\log n)$ time, and so cannot be *Block*-optimal. The adaptive mergesort of Carlsson, Levcopolous, and Petersson [3] is *Block*-optimal; on the other hand it is neither *Inv*-optimal nor *SMS*-optimal.

Using the transitivity of the diagram, the descriptor for Tree-based Mergesort thus crosses between *Rem* and *Block*; crosses above *Inv* and below *Loc*; and above *SMS* and below *Reg*. Hence, to fully understand its adaptivity it remains to analyse its behaviour with respect to *Osc*, the amount of oscillation in a sequence [7]. This is left as an open problem. Our knowledge of the adaptivity of Tree-based Mergesort is summarized as follows:

Corollary 9. *Tree-based Mergesort is optimal with respect to the measures Exc, Rem, Max, Inv, Runs, SUS, Enc, and SMS.*

No other known sorting algorithm is optimal with respect to the same combination of measures as Tree-based Mergesort. In fact, the only other sorting algorithm that can compete in terms of proven adaptivity is Mannila's Local Insertion Sort, the descriptor for which is known to cross immediately above *Block*, *Loc*, and *Runs* [11].

8. Experimental behaviour

Figure 4 shows the adaptivity of the Tree-based Mergesort in an experimental setting. Each point in the graph indicates the number of comparisons per item expended when sorting a sequence of n = 65,536 items with a pseudo-randomly controlled amount of presortedness according to one of the three measures discussed above [10]. That is, the three curves demonstrate the adaptivity of the algorithm with respect to *Inv*, *Rem*, and *SMS*. As the value of the measure increases (note that *Inv* is normalized by n, and that the graph is plotted as a function of the average number of inversions per item) so too does the number of comparisons consumed.

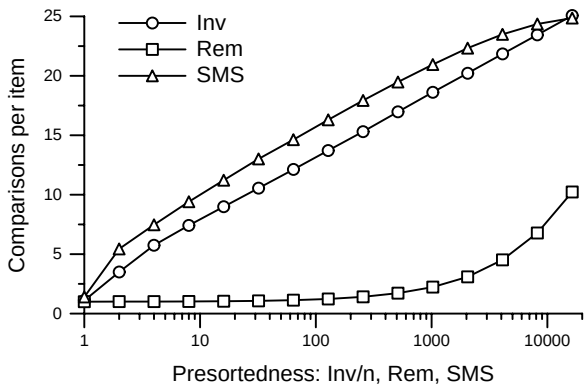

Fig. 4. Comparisons required to sort 65,536 items

It is also interesting to compare the behaviour of the Tree-based Mergesort with other methods for sorting. Table 2 shows the time taken by the Tree-based Mergesort and two other sorting algorithms when sorting n = 10,000 and n = 100,000 integers on a Sun workstation. The Bentley-McIlroy [1] Quicksort implementation is probably the best general-purpose sorting programme devised to date; while the Splaysort implementation [10] is a good example of an adaptive sorting algorithm. In the first section of the table the integers are fully presorted; in the second they are random.

Table 2. Time to sort n items (seconds)

Method	Sorted		Random	
	10,000	100,000	10,000	100,000
Quicksort [1]	0.13	1.55	0.16	1.97
Splaysort [10]	0.05	0.50	0.24	3.99
Tree-based Mergesort	0.21	2.16	2.21	29.09

Although the number of comparisons expended by the Tree-based Mergesort is small when the list is sorted, the constant factor on the running time is large, and it cannot compete with either the Bentley-McIlroy Quicksort or the Splaysort if speed is the primary selection criteria, even on fully sorted lists. This is unsurprising given the complexity of the required tree manipulations.

Acknowledgements. We thank Gary Eddy, who undertook the implementation of the Tree-based Mergesort reported in Sect. 8. We also thank the referees for their helpful comments. This work was in part supported by the Australian Research Council.

References

1. J.L. Bentley, M.D. McIlroy: Engineering a sorting function. Softw. Pract. Exper. **23**, 1249–1265 (1993)
2. M.R. Brown, R.E. Tarjan: Design and analysis of a data structure for representing sorted lists. SIAM J. Comput. **9**, 594–614 (1980)
3. S. Carlsson, C. Levcopoulos, O. Petersson: Sublinear merging and Natural Mergesort. Algorithmica **9**, 629–648 (1993)
4. C.R. Cook, D.J. Kim: Best sorting algorithms for nearly sorted lists. Comm. ACM **23**, 620–624 (1980)
5. D.E. Knuth: Big omega and big omicron and big theta. SIGACT News **8**, 18–24 (1976)
6. C. Levcopoulos, O. Petersson: Sorting shuffled monotone sequences. Inf. Comput. **112**, 37–50 (1994)
7. C. Levcopoulos, O. Petersson: Adaptive Heapsort. J. Algorithms **14**, 395–413 (1993)
8. H. Mannila: Measures of presortedness and optimal sorting algorithms. IEEE Trans. Comput. **C-34**, 318–325 (1985)
9. K. Mehlhorn: Data Structures and Algorithms, Vol. 1: Sorting and Searching. Berlin: Springer 1984.
10. A. Moffat, G. Eddy, O. Petersson: Splaysort: fast, versatile, practical. Softw. Pract. Exper. **26**, 781–797 (1996)
11. O. Petersson, A. Moffat: A framework for adaptive sorting. Discrete Appl Math. **59**, 153–179 (1995)
12. W. Pugh: Skip lists: a probabilistic alternative to balanced trees. Comm. ACM **33**, 668–676 (1990)
13. R.E. Tarjan: Amortized computational complexity. SIAM J. Algebraic Discrete Met **6**, 306–318 (1985)