ELSEVIER

Information Processing Letters 48 (1993) 253–259

Information Processing Letters

Queue-mergesort

Mordecai J. Golin *,[a], Robert Sedgewick **,[b]

[a] *Hong Kong University of Science and Technology, Clear Water Bay, Kowloon, Hong Kong*
[b] *Department of Computer Science, Princeton University, Princeton, NJ 08544, USA*

Communicated by L. Kott
Received 20 May 1992
Revised 3 May 1993

Abstract

Mergesort is one of the oldest and most venerable sorting algorithms and exists in many different flavors. In this short note we present yet another mergesort variant, *queue-mergesort*. We show that, like top–down mergesort but unlike bottom–up mergesort, this new variant performs an optimal number of comparisons in the worst case.

Key words: Algorithms; Mergesort; Huffman encoding; Binary trees

1. Introduction

Mergesort is one of the oldest and most venerable sorting algorithms known to computer science. The idea of sorting a set by partitioning it into two subsets, sorting each of the subsets separately and then merging the two now sorted subsets back together again is a very natural one. In fact it is thought that Mergesort was the first sorting algorithm to have been programmed into a computer [4]. Since those early days many variations on the original theme have sprung up, differing mainly in the method of partitioning the set. The best known variant is probably top–down mergesort in which the sizes of the two sets differ by at most one element. This variant, usually implemented recursively, runs particularly well on arrays. There is a bottom–up mergesort which is well suited to sorting elements in a singly linked list. There is another variant which runs slightly better on linked lists but requires that the lists be doubly and not singly linked. This variant has been picturesquely described [6] as "burning the candle at both ends". Knuth [6] and Sedgewick [7] provide descriptions and implementations of these variants.

In this note we will describe yet another variant of Mergesort and analyze how well it compares with some of the existing versions. We name this new variant *queue-mergesort* for reasons soon to become obvious. It is particularly well suited for sorting elements in a linked list. In

* Corresponding author. This research was performed while the author was at Princeton University. It was supported by National Science Foundation grant DCR-8605962 and Office of Naval Research grant N00014-87-K-0460.

** This research was supported by National Science Foundation grant DCR-8605962 and Office of Naval Research grant N00014-87-K-0460.

Elsevier Science Publishers B.V.
SSDI 0020-0190(93)E0171-F

```
queue-mergesort(Q):

    while (Q.size! = 1) do
        Q.put(Merge(Q.get, Q.get))
```

Fig. 1. Pseudo-code for *queue-mergesort*.

Section 2 we present queue-mergesort. In Section 3 we prove that, as mergesorts go, this new variant performs an optimal number of comparisons. This will follow directly from some well-known facts about trees and Huffman encoding. We conclude in Section 4 by comparing the performance of this new variant with those of some of the older variants.

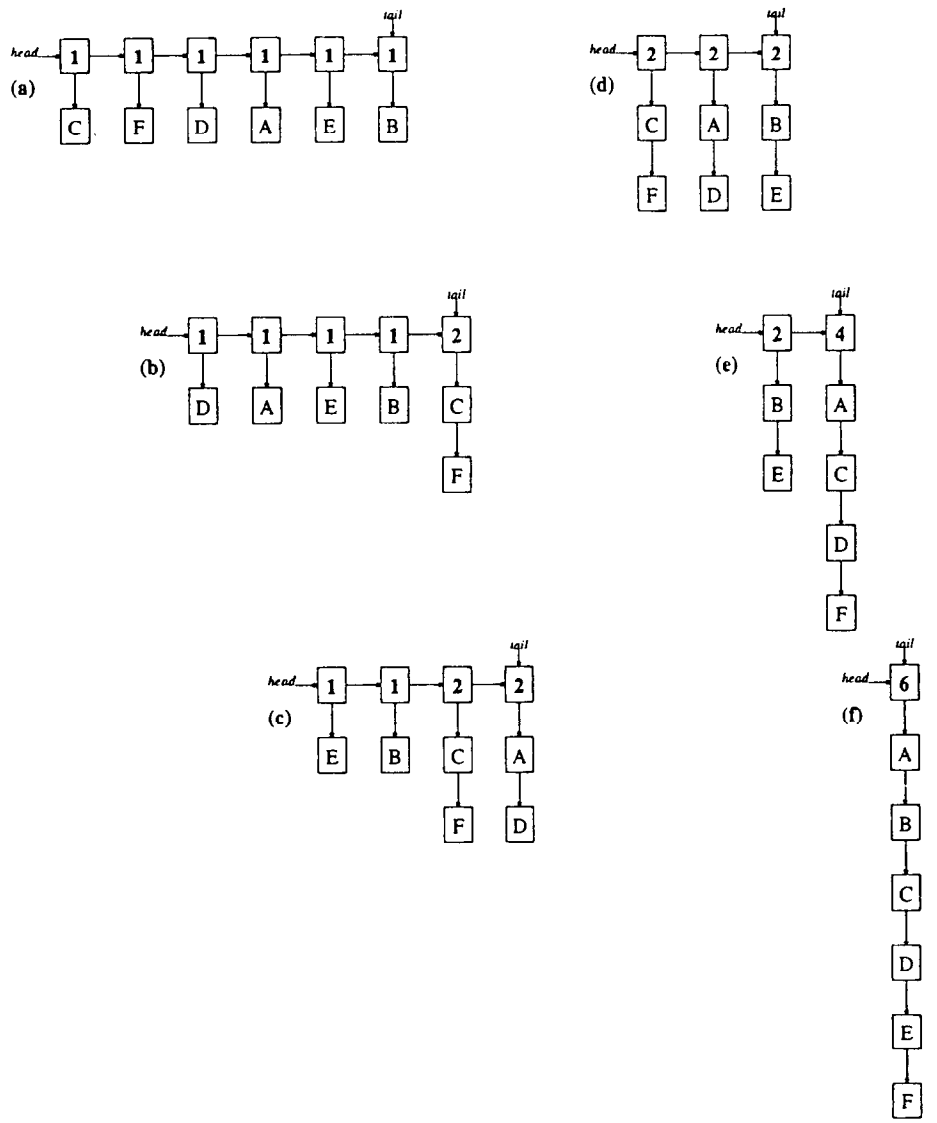

Fig. 2. A worked example of queue-mergesort that sorts the keys C, F, D, A, E, B. Figs. (a)–(e) show the state of the queue at each step of the algorithm. The number at the top of each list is the number of elements contained in that list. Compare to Fig. 3(a).

2. The algorithm

It is very easy to merge two lists L_1 and L_2 which are already sorted by increasing value. This is done by repeating the following operation until one of the lists is empty: compare the minimal element in L_1 to the minimal element in L_2 and remove the smaller of these two elements from its corresponding list. Conclude by taking, in order, all of the elements remaining in the non-empty list. If the sizes of L_1 and L_2 are respectively n_1 and n_2 then merging the two lists will require, in the worst case, $n_1 + n_2 - 1$ comparisons. *Merge*(L_1, L_2) will be a function that takes pointers to two sorted lists and returns a pointer to the sorted merged list.

To merge n items our algorithm will start with each item in its own list, these n lists linked together to form a queue. A queue, Q, is a linked list with *head* and *tail* pointers pointing to its first and last elements. In our case the elements in the queue will be sorted lists. The queue Q is accessed through the operations Q.get which returns a pointer to the list which was at the head of Q while removing the list from Q and $Q.put(p)$ which puts the list pointed to by p at the tail of Q. The function $Q.size$ will return the number of lists in Q

We now present *queue-mergesort.* Start with each of the items to be sorted in its own unique list and store the lists in the queue Q. Repeat the following until only one list remains in Q: *get* the first two lists from the head of the queue, merge them, and *put* the merged list at the tail of the queue. When the algorithm terminates the single list remaining in the queue will contain all of the items in sorted order. We present pseudo-code in Fig. 1 and a worked example in Fig. 2.

It is quite simple to prove correctness of the algorithm. Notice that at each step the number of lists in the queue decreases by one so the algorithm must terminate. Notice too that the following invariant is true after each step; each list in the queue is internally sorted and together all lists contain the original n items. Consequentially, when the algorithm terminates the one list remaining in the queue will contain all of the items in sorted order.

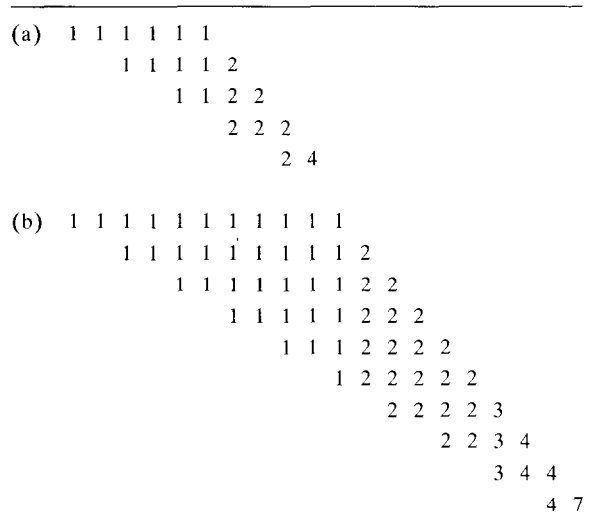

Fig. 3. The merge patterns for (a) 6 and (b) 11 items. The ith row of each pattern contains the number of items in each the lists stored in the queue immediately before the ith merge occurs; the lists appear in the order that they appear in the queues. Notice that the lists appear in the queue sorted by non-decreasing size.

We point out that the number of items in a list at any given time depends only upon the *number of items* to be sorted and not upon the items themselves. In Fig. 3 we show the sizes of the lists in the queue at each step for the cases of 6 and 11 inputs. Notice that at any given time the lists appear in the queue in non-decreasing order of size.

3. Analysis

In this section we will prove that queue-mergesort is an optimal mergesort. By this we mean that for every n the worst-case number of comparisons performed by queue-mergesort is no more than that performed by any other mergesort. In order for this last statement to have any meaning we must first define what we mean by a mergesort.

A *mergesort* is an algorithm which sorts by merging. Such an algorithm maintains a collection of lists, each of which is internally sorted.

The algorithm starts with each item to be sorted in a separate list. At each stage of the algorithm two lists are removed from the collection and merged together to form a new list. This new list is added back to the collection. The algorithm terminates when the collection contains only one list. This list will in turn contain all of the input items in sorted order. Variants of mergesort differ in how they choose the lists to be merged. A convenient way of describing a mergesort is by drawing its merge tree [5, 2.3.4.5]. The leaves (external nodes) of the tree represent the original items to be sorted. The internal nodes represent the lists formed by the merges. The two children of the internal node corresponding to a list L are the nodes corresponding to the two lists that were merged to form L.

When drawing the tree we associate a weight $w(i)$ with each node i. This weight will be the number of items in the list that that particular node corresponds to, e.g. all leaves have weight 1 and the root, representing the fully sorted list, has weight n.

Fig. 4(a) is the merge tree for queue-mergesort run on 11 elements. Fig. 4(b) is the merge tree for top-down mergesort run on 11 elements. Top-down (recursive) mergesort always splits a set of n elements into two sets of size $\lfloor n/2 \rfloor$ and $\lceil n/2 \rceil$. Fig. 4(c) is the merge tree for bottom-up mergesort run on 11 elements. Bottom-up mergesort starts with every item in its own list. It merges the lists in passes. In each pass it pairs up the current lists, ignoring the last list if there are an odd number of them, and merges the paired lists. It continues doing this until only one list remains.

It is now easy to see how many comparisons a mergesort will perform in the worst case. Let T be the tree associated with a mergesort on n items, i.e. T has n leaves. As mentioned at the beginning of Section 2 merging two lists of size n_1 and n_2 requires $n_1 + n_2 - 1$ comparisons in the worst case. Put another way, merging two lists which together contain m items requires $m - 1$ comparisons in the worst case. A binary tree with n leaves has $n - 1$ internal nodes. Thus the total number of comparisons performed by the mergesort in the worst case will be

$$\sum_{\substack{v \in T \\ v \text{ internal}}} [w(v) - 1] = \left[\sum_{\substack{v \in T \\ v \text{ internal}}} w(v) \right] - (n - 1). \tag{1}$$

We define the weight of T to be $w(T) = \sum_{v \in T,\, v \text{ internal}} w(v)$, where the sum is over all internal nodes of the tree. The tree in Fig. 4(a) has $w(T) = 39$, the tree in Fig. 4(b) also has $w(T) = 39$ while the tree in Fig. 4(c) has $w(T) = 40$.

Eq. (1) tells us that a mergesort with associated merge tree T will be an optimal mergesort if $w(T) \leqslant w(T')$ for all merge trees T' with n leaves. We use this fact to prove the following:

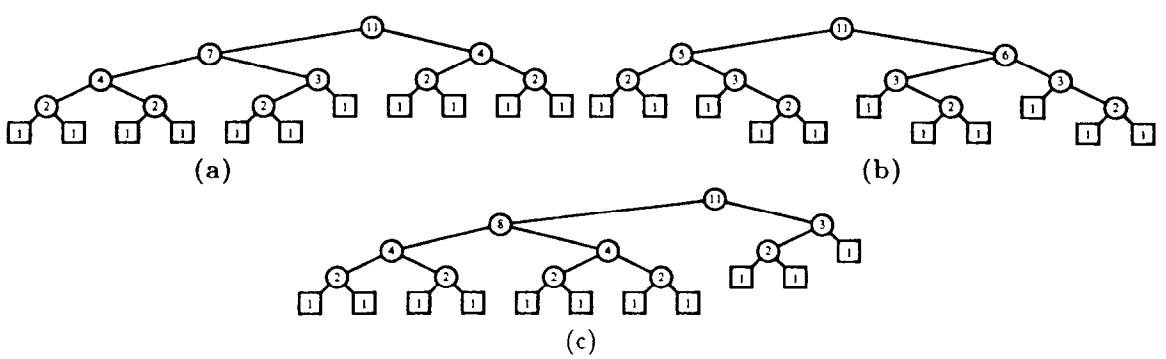

Fig. 4. These trees represent three different methods for mergesorting 11 elements. The number in each node is its weight. (a) is queue-mergesort, (b) is top-down mergesort and (c) is bottom-up mergesort. Notice that even though (a) and (b) are very different trees they have the same external path length $5*3 + 6*4 = 39$. (c) has external path length $1*2 + 2*3 + 8*4 = 40$.

Theorem 1. *Queue-mergesort is an optimal mergesort.*

Proof. Let n be an integer and let T be the merge tree associated with queue-mergesorting n items. From the comments preceeding the proof it is enough to show that $w(T) \leqslant w(T')$ for all merge trees T' with n leaves.

We use the fact that the weight of a merge tree is equal to its external path length. The height $h(l)$ of a node l in a tree is the distance of a path from l to the root. The *external path length* of a tree T' is the sum $E(T') = \sum_{l \text{ a leaf of } T'} h(l)$.

It is known [5, 2.3.4.5–9] that $w(T') = E(T')$ for any merge tree T'. Therefore to prove the theorem it suffices to prove that $E(T) \leqslant E(T')$ for all T' with n leaves. This will follow directly from certain properties of Huffman encoding.

Let $w_1, w_2, \ldots, w_n$ be n non-negative weights. Huffman encoding builds a binary tree with n leaves $l_1, l_2, \ldots, l_n$ such that the *weighted external path length* $\sum_{i \leqslant n} h(l_i) w_i$ is minimal [1] [1]. The Huffman encoding algorithm works as follows: Start with a set of n nodes having weights $w_1, w_2, \ldots, w_n$. These nodes will be the leaves of the tree. Repeat the following until the set contains only one node: remove two nodes a and b of smallest weight from the set. Create a new internal node c whose children will be a and b and let $w_c = w_a + w_b$. Insert c into the set.

Queue-mergesort bears a resemblance to Huffman encoding. Think of queue-mergesort as building its associated merge tree from the leaves up. It starts with the n leaves (with weight 1) in a set. Each step in queue-mergesort can be thought of as removing two nodes (i.e. lists) from the set and making them the children of a third node (i.e. merging the lists) whose weight, exactly as in Huffman encoding, will be the sum of the weights of its children. This third node will be inserted into the set. When there is only one node left in the set, this sole survivor will be the root of the merge tree.

[1] We have not been able to find any explicit references to queue-mergesort in the existing literature. Even's [1] queue-based implementation of Huffman encoding does have a similar flavor though.

We claim that the merge tree T corresponding to queue-mergesorting n items is exactly the tree constructed by the Huffman algorithm when $w_1 = w_2 = \cdots = -w_n = 1$. To prove this claim it suffices to show that at each step of queue-mergesort the two lists at the front of the queue, i.e. the two lists to be merged, are the two smallest lists on the queue (see Fig. 3). We will actually prove something more, namely that after each step of the algorithm the lists in the queue are all sorted by non-decreasing size.

Fix n. The proof is by induction on the number of merges performed by the algorithm so far. We assume that $n \geqslant 4$; the cases $n < 4$ can be examined separately. When the algorithm begins the lists are certainly sorted by size. After the first merge step all of the lists still contain only one item except for the last list which contains two items and the lists are still sorted by size. Now assume that we know that the lists are sorted by size after each of the first s steps of the algorithm, $s \geqslant 1$. To show that they are sorted after the $(s+1)$st step it is enough to show that the list L_{s+1} inserted at the tail of the queue after step $s+1$ contains at least as many elements as list L_s inserted after step s. But L_s is the union of the first two lists in the queue after step $s-1$ and so by the induction hypothesis it is no bigger than L_{s+1} which is the union of the third and fourth lists on the queue after step $s-1$.

Every step of queue-mergesort therefore merges two lists of smallest size; by reduction to Huffman encoding the external path length of the merge tree associated with the algorithm is minimal over all weighted binary trees with n leaves each having weight 1. Since all merge trees of n items are weighted binary trees with exactly n leaves each having weight 1 this proves that queue-mergesort is an optimal mergesort. □

4. Optimal mergesorts

In the previous section we proved that queue-mergesort is an optimal mergesort. That is, queue-mergesort performs an optimal number of comparisons in the worst case. Our proof did not

tell us *how many* comparisons that is. In this section we remedy that lack by quickly reviewing some facts about minimal external path lengths from [6, 5.3.1].

Let T be a merge tree describing an optimal mergesort on n items. The worst-case number of comparisons that can be performed while executing these merges is

$$w(T) - (n-1) = E(T) - (n-1) = \left(\sum_{\substack{l \in T \\ l \text{ a leaf}}} h(l) \right) - (n-1). \quad (2)$$

Thus a merge tree T describes an optimal mergesort on n items if and only if T has minimum external path length $\sum_{l \text{ a leaf}} h(l)$. It is known that this occurs if and only if the following condition is satisfied: all of T's leaves are located on its bottom two levels. For example the trees in Figs. 4(a) and 4(b) have minimal external path lengths for 11 leaves while that in Fig. 4(c) does not.

Let T be a binary tree with minimal external path length for n leaves where $2^k \leqslant n < 2^{k+1}$. Then all of T's leaves are on its bottom two levels, the kth level must be full and the $(k+1)$st must contain only leaves. Let s be the number of leaves on the kth level of T and r the number of internal (non-leaf) nodes on the kth level: $s + r = 2^k$. The two children of any of the r internal nodes on the kth level are on the $(k+1)$st level so these children must be leaves. Thus $s + 2r = n$ and the external path length of T is

$$\begin{aligned} &\sum_{\substack{l \in T \\ l \text{ a leaf}}} h(l) \\ &= sk + 2r(k+1) = nk + 2(n - 2^k) \\ &= n\lfloor \log_2 n \rfloor + 2n - 2^{\lfloor \log_2 n \rfloor + 1} \\ &= n \log_2 n + n\left[2 - \{\log_2 n\} - 2^{1-\{\log_2 n\}}\right], \end{aligned} \quad (3)$$

where we define $\{x\} = x - \lfloor x \rfloor$ to be the fractional part of x. Plugging this back into (2) we find that the worst-case number of comparisons performed by an optimal mergesort is

$$f(n) = n \log_2 n + nh(\log_2 n) - (n-1), \quad (4)$$

where $h(x) = 2 - \{x\} - 2^{1-\{x\}}$ is a continuous periodic function with period 1, that is, for all x, $h(x+1) = h(x)$.

In the previous two sections we have shown that queue-mergesort is an optimal mergesort and calculated the worst-case number of comparisons it performs. Before concluding we would like to make a few general comments about optimal mergesorts.

One can prove by induction on $\lceil \log_2 n \rceil$ that the merge tree T that describes top–down mergesort on n items has all of its leaves on levels $\lceil \log_2 n \rceil - 1$ and $\lceil \log_2 n \rceil$. Thus, by the comments at the beginning of this section, T has minimum external path length for a tree with n leaves and top–down mergesort is also an optimal mergesort. Note that even though queue-mergesort and top–down mergesort are both optimal and therefore perform the same worst-case number of comparisons they are two very different algorithms since they have different merge trees. Compare for example Figs. 4(a) and 4(b).

Another small curiosity: The worst-case number of comparisons performed by top–down mergesort satisfies the recurrence

$$f(n) = f(\lfloor n/2 \rfloor) + f(\lceil n/2 \rceil) + n - 1, \quad n > 1,$$
$$f(1) = 0. \quad (5)$$

This recurrence is usually analyzed [3, p. 9] by differencing and re-summing to find $f(n) = \sum_{i \leqslant n} \lceil \log_2 i \rceil$. This sum is then massaged to derive a closed form similar to (3) or (4). It is interesting to note that if one understands the underlying combinatorics, then the solution to (5) can be found "for free". Eq. (4) describes the worst-case number of comparisons used by an optimal mergesort, top–down mergesort is an optimal mergesort, ergo (4) is the solution to (5).

We should point out that there is a close relationship between binary representations and the number of comparisons required for mergesort. In fact, writing $f(n)$ as $\sum_{i \leqslant n} \lceil \log_2 i \rceil$ tells us that $f(n)$ is the number of bits needed to write down the binary representations of all the integers less than n. This relationship between mergesort and binary representations will be further developed in an upcoming paper [2] which will show that the cost of bottom–up mergesort is

also expressible as a function (albeit a more complicated one) of the binary representations of the integers less than n.

We conclude with a few words an optimal versus non-optimal mergesorts. Not all mergesorts are optimal but, sometimes, minor changes can transform a non-optimal one into an optimal one.

Bottom–up mergesort, for example, is not optimal. Fig. 4(c) shows that the merge tree that describes bottom–up mergesorting 11 items does not have minimum external path length since its external path length is greater than those in Figs. 4(a) and 4(b).

Going further it is not difficult to show that, as n increases, bottom–up mergesort will require arbitrarily more comparisons in the worst case than an optimal mergesort. Let $n = 2^k + 1$. Working through the details of (4) we find that for this n an optimal mergesort uses $(2^k + 1)k + 2 - 2^k$ comparisons in the worst case. In contrast bottom–up mergesort will mergesort the first 2^k items together and then merge this sorted list with the one leftover item. In total this will require $k2^k + 1$ comparisons in the worst case [6, 5.2.4–14]. Thus, for $n = 2^k + 1$, bottom–up mergesort will need $2^k - k - 1 \approx n - \log_2 n$ more comparisons than an optimal mergesort such as top–down or queue-mergesort.

On the other hand a simple modification makes bottom–up mergesort optimal. As in the standard bottom–up algorithm we make repeated passes over the lists, pairing them up, two by two. The modification is that, at the end of a pass – in the case that there are an odd number of lists – we do not leave the last list by itself but merge it with the new list that was just created when the second and third to last lists on that pass were merged. It is not difficult to show that all of the leaves of the merge-tree created are on its bottom two levels so this new mergesort is an optimal one.

We end by pointing out that the "burning the candle at both ends" mergesort described in the first paragraph of this paper is easily shown to be non-optimal.

Acknowledgement

The authors would like to thank the unknown referee who suggested the modification that makes bottom–up mergesort optimal.

References

[1] S. Even, *Graph Algorithms* (Computer Science Press, Rockville, MD, 1979).

[2] M. Golin and R. Sedgewick, Mergesort and digital sums, In preparation.

[3] R. Graham, D. Knuth and O. Patashnik, *Concrete Mathematics: A Foundation For Computer Science* (Addison-Wesley, Reading, MA, 1988).

[4] D.E. Knuth, Von Neumann's first computer program, *Computing Surveys* **2** (4) (1970) 247–260.

[5] D.E. Knuth, *The Art of Computer Programming: Vol. I. Fundamental Algorithms*, (Addison-Wesley, Reading, MA, 2nd ed., 1973).

[6] D.E. Knuth, *The Art of Computer Programming: Vol. III. Sorting and Searching* (Addison-Wesley, Reading, MA, 1973).