JournalName:ASComputerScienecs
ArticleType:Research

Merge Sort Revisited

Merge sort is a sorting technique based on the divide-and-conquer technique. With its worst-case time complexity being O(n·log n), it is one of the most respected algorithms. However, in practice, Quick sort is almost three times faster than it although the worst-case time complexity of Quick sort is bounded by O(n^2), much worse than O(n·log n). In this paper, we discuss a new algorithm, which improves the merge sort in two ways: (i) cutting down data movements conducted in the merging processes; and (ii) replacing the recursive calls with a series of improved merging operations. Our experiments show that for the randomly generated input sequences, the performance of our algorithm is comparable to the quick sort. But for the sorted or almost sorted input sequences, or reverse sorted input sequences, our algorithm is nearly 5000 times better than it.

CCS Concepts: • **Theory of computation → Algorithm design and analysis**.

Additional Key Words and Phrases: sequences, merge sorting, quick sorting.

ACM Reference Format:
Yangjun Chen and Ruilin Su. 2021. Merge Sort Revisited. *Acta Scientific Computer Sciences*, Article xxx (August 2021), 6 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

Merge sort (sometimes spelled mergesort) is an efficient sorting algorithm that uses a divide-and-conquer strategy to order elements in a sequence. Its worst-case time complexity is bounded by O(n·log n), where n is the number of elements in the sequence. This running time is better than Quick sort's, O(n^2). However, in practice, the quick sort is normally faster. One reason for this is that Quick sort is an in-place algorithm (by which only quite small extra space is used) and its average running time is bounded by O(n·log n). But the most important reason for this is due to the huge amount of data movements carried out by Merge sort itself when merging subsequences.

In this paper, we address this issue and propose a method which is able to cut down the number of data movements of the merge sort by half. Another observation is that the conquer step of Merge sort can further be greatly improved by replacing recursive calls directly with a series of merging operations.

As our experiments demonstrate, the running time of our algorithm for randomly generated input sequences is comparable to Quick sort. However, for the sorted or almost sorted input sequences, or reversely sorted input sequences, our algorithm can achieve more than 5000-fold improvements over Quick sort.

Since the sorting is almost the most frequently performed operation in the software engineering, we think that these improvements are highly significant.

This work is supported by NSERC, Canada, 239074-01 (242523).
Authors' address: Yangjun Chen, y.chen@uwinnipeg.ca; Ruilin Su, su-r@webmail.uwinnipeg.ca, Department of Applied Computer Science, University of Winnpeg, Winnipeg, Manitoba, Canada, R3B 2E9 .

The rest of this paper is organized as follows. In Section 2, we restate Merge sort as a discussion background. Then, in Section 3, we discuss our algorithm. Next, we show the test results in Section 4. Finally, a short conclusion is set forth in Section 5.

2 DECRIPTION OF MERGE SORTING

Merge sort is typically a divide-and-conquer strategy. Given a sequence with n elements, the merge sort involves the following three steps:

(1) *Divide* the sequence into two subsequences such that one is with $\lceil n/2 \rceil$ elements, and the other is with $\lfloor n/2 \rfloor$.

(2) *Conquer* each subsequence by sorting it. Unless the sequence is sufficiently small, use recursion to do this.

(3) *Combine* the solutions to the subsequences by merging them into a single sorted sequence.

The following algorithm implements the above idea. For simplicity, the input of this algorithm is just an array A of numbers to be sorted.

Algorithm 1: *mergeSort*(A)

Input: A - a sequence of elements stored as an array
Output: sorted A

1 **if** $|A| = 1$ **then** return A;
2 $p := 1$; $r := |A|$; $q := (p + r)/2$;
3 *mergeSort*($A[p .. q]$);
4 *mergeSort*($A[q + 1 .. r]$);
5 *merge*(A, p, q, r);

In line 1 of the above algorithm, *mergeSort*( ), we first check whether $|A| = 1$. If it is the case, return A. Otherwise, the divide step simply computes an index q (see line 2) that partitions A into two subarrays: $A[p .. q]$ containing $\lceil n/2 \rceil$ elements, and $A[q + 1 .. r]$ containing $\lfloor n/2 \rfloor$ elements.

By the first recursive call, we will sort $A[p .. q]$ (see line 3). By the second recursive call, we will sort $A[q + 1 .. r]$ (see line 4). Then, we will call the merge procedure to create an entirely sorted array A (see line 5).

In the merge procedure *merge*(A, p, q, r) shown below, line 1 computes the length n_1 and n_2 of the subarrays $A[p .. q]$ and $A[q + 1 .. r]$, respectively; and initializes index variable k to p, which is used to scan A from left to right. The **for**-loop of lines 3-4 copies the subarray $A[p .. q]$ into $L[1 .. n_1]$ while the **for**-loop of lines 5-6 copies the subarray $A[q + 1 .. r]$ into $R[1 .. n_2]$. In the **while**-loop of lines 7-12, two index variables i, j are used to scan L and R, respectively. Depending on whether $L[i] \le R[j]$, $L[i]$ or $R[j]$ will be sent to $A[k]$. When we go out of the **while**-loop, lines 13-16 will be executed, by which the remaining elements in L or in R will be copied back into A, depending on whether $i > n_1$ or $j > n_2$.

3 IMPROVEMENTS

In this section, we discuss how to improve the algorithm described in the previous section. First, we discuss a method to reduce data movements conducted in *merge*( ), which enables us to decrease the running time by more than a half. Then, we change the recursive algorithm to a non-recursive procedure by which the performance can be further improved.

Algorithm 2: $merge(A, p, q, r)$

Input: Both $A[p \,..\, q]$ and $A[q+1 \,..\, r]$ are sorted; but A as a whole is not sorted
Output: sorted A

$n_1 := q - p + 1$; $n_2 := r - q$; $k := p$;
let $L[1 \,..\, n_1]$ and $R[1 \,..\, n_2]$ be new arrays;
for $i = 1$ to n_1 **do**
  $L[i] := A[p + i - 1]$;
for $j = 1$ to n_2 **do**
  $R[j] := A[q + j]$;
while $i \le n_1$ and $j \le n_2$ **do**
  if $L[i] \le R[j]$ **then**
    $A[k] := L[i]$; $i := i + 1$;
  else
    $A[k] := R[j]$; $j := j + 1$;
  $k := k + 1$;
if $j > n_2$ **then**
  copy the remaining elements in L into $A[p \,..\, r]$
else
  copy the remaining elements in R into $A[p \,..\, r]$;

- *Deduction of data movements*

We notice that in the procedure $merge(\ )$ of Merge sort the copying of $A[q + 1 \,..\, r]$ into R is not necessary, since we can directly merge L and $A[q + 1 \,..\, r]$ and store the merged, but sorted sequence back into A.

Denote by A' the sorted version of A. Denote by $A'(i, j)$ a prefix of A' which contains the first i elements from L and first j elements from $A[q + 1 \,..\, r]$. Obviously, we can store $A'(i, j)$ in A itself since after the jth element has been inserted into A', the first $q - p + j + 1$ entries in A are empty and $q - p + 1 \ge i$ (thus, $q - p + j + 1 \ge i + j$). In terms of this simple analysis, we give the following algorithm for merging two sorted subarrays: L and $A[q + 1 \,..\, r]$ (see Algorithm 3).

The difference of this algorithm from $merge(\ )$ (Algorithm 2, described in the previous section) mainly consists in:

(1) Array R is not created.
(2) The copying of the remaining part of Array R into A is not needed in the case that $j > n_2$ since R itself is replaced by $A[q + 1 \,..\, r]$, and the remaining elements of R are now already in A.

These two differences enable us to save more than half of the running time of Merge sort.

In addition, less space is needed since R is not created at all.

- *Non-recursive Algorithm*

Merge Sort can be further improved by replacing its recursive calls with a series of merging operations, by which the recursive execution of the algorithm is simulated. The whole working process can be divided into $\lceil \log_2 (r - p + 1) \rceil$ phases. In the first phase, we will make $\lceil n/2 \rceil$ merging

Algorithm 3: *mergeImpr* (A, p, q, r)

Input: Both $A[p\ ..\ q]$ and $A[q+1\ ..\ r]$ are sorted; but A as a whole is not sorted
Output: sorted A

$n_1 := q - p + 1$; $n_2 := r - p + 1$; $k := p$;
let $L[1\ ..\ n_1]$ be a new array;
for $i = 1$ to n_1 **do**
    $L[i] := A[p + i - 1]$;
$i := 1$; $j := q + 1$;
while $i \le n_1$ and $j \le n_2$ **do**
    if $L[i] \le A[j]$ **then**
        $A[k] := L[i]$; $i := i + 1$;
    else
        $A[k] := A[j]$; $j := j + 1$;
    $k := k + 1$;
if $j > n_2$ **then**
    copy the remaining elements in L into $A[k\,..r]$

operations, where $n = r - p + 1$, with each merging two single-element sequences together. In the second phase, we will make $\lceil n/4 \rceil$ merging operations with each merging two two-element sequences together, and so on. Finally, we will make only one operation to merge two sorted subsequences to form a globally sorted sequence. Between the sorted subsequences, one contains $\lceil n/2 \rceil$ elements while the other contains $\lfloor n/2 \rfloor$ elements.

Algorithm 4: *mSort* (A)

Input : A - a sequence of elements stored as an array;
Output: sorted A

if $|A| \le 1$ **then** return A;
$r := |A|$;
$l := \lceil \log_2 r \rceil$;
$j := 2$;
    for $i = 1$ to l **do**
        for $k = 1$ to $\lceil r/j \rceil$) **do**
            $s := \lfloor (k-1)j \rfloor$;
            mergeImpr $(A, s+1, s+\lceil j/2 \rceil, s+j)$;
        $j := 2j$;

More importantly, by our method, there is no system stack frame overhead, and therefore no stack overflow problem (caused by huge numbers of recursive calls) as by the recursive merge sort and also by the recursive quick sort. Thus, given a certain size of main memory, much longer input sequences can be sorted by our method, as demonstrated by our experiments.

4 EXPERIMENTS

In our experiments, we have tested altogether 5 different methods:

TABle 1. Time on sorted input sequences (ms)

input size	*ms*	*ims-1*	*ims-2*	*qs*	*r-qs*
6,5536	6	1	1	751	518
131,072	13	2	2	3,012	1,044
262,144	28	6	4	12,789	2,088
524,288	54	11	9	49,625	4,176
1,048,576	106	24	19	-	9,728
2,097,152	214	49	41	-	21,489
4,194,304	430	110	97	-	54,763

TABle 2. Time on random input sequences (ms)

input size	*ms*	*ims-1*	*ims-2*	*qs*	*r-qs*
6,5536	10	4	3	3	5
131,072	19	9	8	7	9
262,144	39	19	17	14	19
524,288	80	38	35	28	31
1,048,576	160	78	72	55	69
2,097,152	320	158	136	110	153
4,194,304	670	324	295	210	312

– Merge sort (*ms* for short, [2]),
– Improved merge sort 1 (*ims-1* for short, discussed in this paper),
– Improved merge sort 2 (*ims-2* for short, discussed in this paper),
– Quick sort *(qs* for short, [3]), and
– Random pivot quick sort (*r-qs* for short, [1]).

Among all the above 5 methods, *ms* is the traditional merge sort described in Section 2. *qs* is the traditional quick sort, by which a fix element (e.g., the last, the first or the middle element) is chosen as the pivot for each sequence partition while *r-qs* is one of its variants, by which the pivot for each sequence partition is randomly selected. *ims-1* and *ims-2* are two of our improvements discussed in Section 3.

The code of our two improvements are produced by ourselves while all the other codes are downloaded from the Internet. They are all written in C++ and compiled by GNU g++ compiler version 5.4.0 with compiler option '-O2'. All tests run on a Windows 10 machine with a single CPU i7-11800H. The system memory is of 32 GB.

In Table 1, we show the running time of all the algorithms on sorted input sequences. From this, we can see that Quick sort is much worse than all the other methods. Especially, it interrupts due to stack overflows even for an input whose size is not so large. Its performance can be somehow improved by randomly choosing pivots for each sequence partition. But it is still orders of magnitude worse than Merge sort. In the opposite, our non-recursive algorithm essentially improves Merge sort and can achieve more than 5000-fold improvements over Quick sort.

In Table 2, we show the test results over randomly generated inputs. For large inputs, they clearly show that Merge sort is almost three time slower than Quick sort. But our algorithm is comparable to Quick sort, and even a little bit better than its variant.

5 CONCLUSIONS

In this paper, a method for sorting is discussed. It improves Merge sort in two ways. First, it cuts down data movements conducted in the merging processes of Merge sort. Second, it replaces the recursion of Merge sort with "iteration", by which the recursive calls are changed to a series of improved merging operations. Our experiments show that for the randomly generated input sequences, the performance of our algorithm is comparable to the quick sort. But for the sorted or almost sorted input sequences, or reverse sorted input sequences, our algorithm is nearly 5000 times better than Quick sort.

REFERENCES

[1] A. Latif, A. Dalhoum, T. Kobbay, A. , Sleit, M. Alfonseca and A. Ortega, *Enhancing QuickSort Algorithm using a Dynamic Pivot Selection Technique*, Wulfenia 19(10):543-552, Jan. 2012.
[2] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, Third edition. 2009.
[3] C. A.R. Hoare, *Quicksort*, The Computer Journal, 5, 10-15, 1962.