Stable Minimum Storage Merging by Symmetric Comparisons

Pok-Son Kim[1]* and Arne Kutzner[2]

[1] Kookmin University, Department of Mathematics, Seoul 136-702, Rep. of Korea
pskim@kookmin.ac.kr
[2] Seokyeong University, Department of E-Business, Seoul 136-704, Rep. of Korea
kutzner@skuniv.ac.kr

Abstract. We introduce a new stable minimum storage algorithm for merging that needs $O(m \log(\frac{n}{m} + 1))$ element comparisons, where m and n are the sizes of the input sequences with $m \leq n$. According to the lower bound for merging, our algorithm is asymptotically optimal regarding the number of comparisons.
The presented algorithm rearranges the elements to be merged by rotations, where the areas to be rotated are determined by a simple principle of symmetric comparisons. This style of minimum storage merging is novel and looks promising.
Our algorithm has a short and transparent definition. Experimental work has shown that it is very efficient and so might be of high practical interest.

1 Introduction

Merging denotes the operation of rearranging the elements of two adjacent sorted sequences of sizes m and n, so that the result forms one sorted sequence of $m+n$ elements. An algorithm merges two adjacent sequences with *minimum storage* [1] when it needs $O(\log^2(m+n))$ bits additional space at most. This form of merging represents a weakened form of *in-place* merging and allows the usage of a stack that is logarithmically bounded in $m+n$. Minimum storage merging is sometimes also referred to as *in situ* merging. A merging algorithm is regarded as *stable*, if it preserves the initial ordering of elements with equal value.
Some lower bounds for merging have been proven so far. The lower bound for the number of assignments is $m+n$, because every element may change its position in the sorted result. The lower bound for the number of comparisons is $\Omega(m \log \frac{n}{m})$ for $m \leq n$. This can be proven by a combinatorial inspection combined with an argumentation using decision trees. An accurate presentation of these bounds is given by Knuth [1].

* This work was supported by the Kookmin University research grant in 2004.

The simple standard merge algorithm is rather inefficient, because it uses linear extra space and always needs a linear number of comparisons. In particular the need of extra space motivated the search for efficient in-place merging algorithms. The first publication presenting an in-place merging algorithm was due to Kronrod [2] in 1969. Kronrod's unstable merge algorithm is based on a partition merge strategy and uses an internal buffer as central component. This strategy has been refined and improved in numerous subsequent publications. A selection of these publications is [3–10]; an accurate description of the history and evolution of Kronrod-related algorithms can be found in [6]. The more recent Kronrod-related publications, like [3] and [6], present quite complex algorithms. The correctness of these algorithms is by no means immediately clear.
A minimum storage merging algorithm that isn't Kronrod-related was proposed by Dudzinski and Dydek [11] in 1981. They presented a divide and conquer algorithm that is asymptotically optimal regarding the number of comparisons but nonlinear regarding the number of assignments. This algorithm was chosen as basis of the implementation of the merge_without_buffer-function in the C++ Standard Template Libraries[12], possibly due to its transparent nature and short definition.
A further method was proposed by Ellis and Markov in [13], where they introduce a shuffle-based in situ merging strategy. But their algorithm needs $((n+m)\log(n+m))$ assignments and $((n+m)\log(n+m))$ comparisons. So, despite some practical value, the algorithm isn't optimal from the theoretical point of view.

We present a new stable minimum storage merging algorithm performing $O(m \log \frac{n}{m})$ comparisons and $O((m+n)\log m)$ assignments for two sequences of size m and n $(m \leq n)$. Our algorithm is based on a simple strategy of symmetric comparisons, which will be explained in detail by an example. We report about some benchmarking showing that the proposed algorithm is fast and efficient compared to other minimum storage merging algorithms as well as the standard algorithm. We will finish with a conclusion, where we give a proposal for further research.

2 The Symmerge Algorithm

We start with a brief introduction of our approach to merging. Let us assume that we have to merge the two sequences $u = (0, 2, 5, 9)$ and $v = (1, 4, 7, 8)$. When we compare the input with the sorted result, we can see that in the result the last two elements of u occur on positions belonging to v, and the first two elements of v appear on positions belonging to u (see Fig. 1 a)). So, 2 elements were exchanged between u and v. The kernel of our algorithm is to compute this number of side-changing elements efficiently and then to exchange such a number of elements. More accurately, if we have to exchange n $(n \geq 0)$ elements between sequences u and v, we move the n greatest elements from u to v and the n smallest elements from v to u, where the exchange of elements is realized by a rotation. Then by recursive application of this technique to the arising

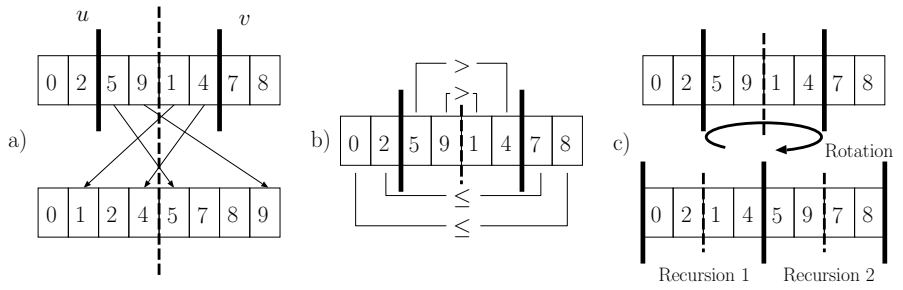

Fig. 1. SYMMERGE example

subsequences we get a sorted result. Fig. 1 illustrates this approach to merging for our above example.

We will now focus on the process of determining the number of elements to be exchanged. This number can be determined by a process of symmetrical comparisons of elements that happens according to the following principle:
We start at the leftmost element in u and at the rightmost element in v and compare the elements at these positions. We continue doing so by symmetrically comparing element-pairs from the outsides to the middle. Fig. 1 b) shows the resulting pattern of mutual comparisons for our example. There can occur at most one position, where the relation between the compared elements alters from 'not greater' to 'greater'. In Figure 1 b) two thick lines mark this position. These thick lines determine the number of side-changing elements as well as the bounds for the rotation mentioned above.
Due to this technique of symmetric comparisons we will call our algorithm SYMMERGE. Please note, if the bounds are on the leftmost and rightmost position, this means all elements of u are greater than all elements of v, we exchange u and v and get immediately a sorted result. Conversely, if both bounds meet in the middle we terminate immediately, because uv is then already sorted. So our algorithm can take advantage of the sortedness of the input sequences.
So far we introduced the computation of the number of side-changing elements as linear process of symmetric comparisons. But this computation may also happen in the style of a binary search. Then only $\lfloor \log(\min(|u|, |v|)) \rfloor + 1$ comparisons are necessary to compute the number of side-changing elements.

2.1 Formal definition

Let u and v be two adjacent ascending sorted sequences. We define $u \leq v$ ($u < v$) iff. $x \leq y$ ($x < y$) for all elements $x \in u$ and for all elements $y \in v$.
We merge u and v as follows:

If $|u| \leq |v|$, then

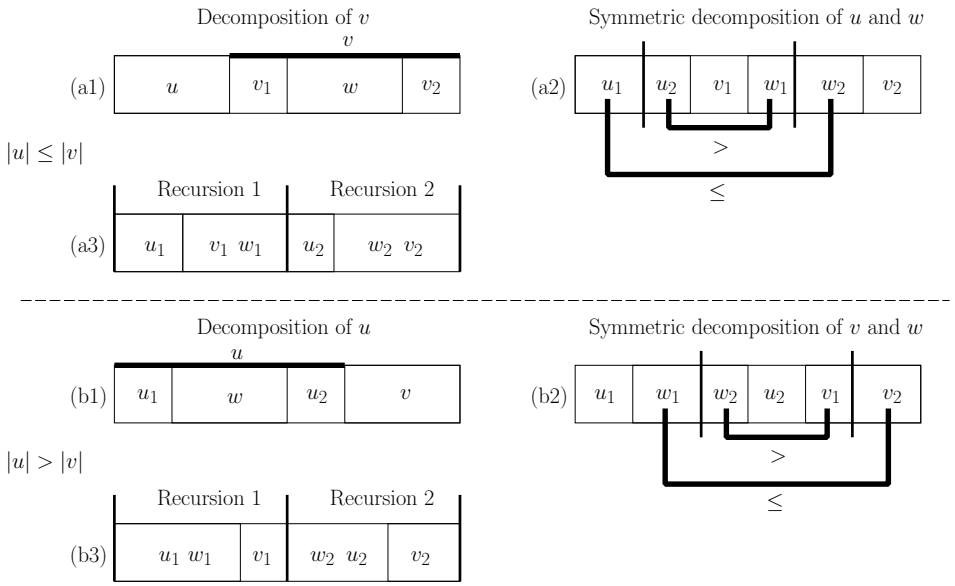

Fig. 2. Illustration of SYMMERGE

(a1) we decompose v into $v_1 w v_2$ such that $|w| = |u|$ and either $|v_2| = |v_1|$ or $|v_2| = |v_1| + 1$.

(a2) we decompose u into $u_1 u_2$ ($|u_1| \geq 0$, $|u_2| \geq 0$) and w into $w_1 w_2$ ($|w_1| \geq 0$, $|w_2| \geq 0$) such that $|u_1| = |w_2|$, $|u_2| = |w_1|$ and $u_1 \leq w_2$, $u_2 > w_1$.

(a3) we recursively merge u_1 with $v_1 w_1$ as well as u_2 with $w_2 v_2$. Let u' and v' be the resulting sequences, respectively.

else

(b1) we decompose u into $u_1 w u_2$ such that $|w| = |v|$ and either $|u_2| = |u_1|$ or $|u_2| = |u_1| + 1$.

(b2) we decompose v into $v_1 v_2$ ($|v_1| \geq 0$, $|v_2| \geq 0$) and w into $w_1 w_2$ ($|w_1| \geq 0$, $|w_2| \geq 0$) such that $|v_1| = |w_2|$, $|v_2| = |w_1|$ and $w_1 \leq v_2$, $w_2 > v_1$.

(b3) we recursively merge $u_1 w_1$ with v_1 as well as $w_2 u_2$ with v_2. Let u' and v' be the resulting sequences, respectively.

$u'v'$ then contains all elements of u and v in sorted order.

Fig. 2 contains an accompanying graphical description of the process described above. The steps (a1) and (b1) manage the situation of input sequences of different length by cutting a subsection w in the middle of the longer sequence

Algorithm 1 SYMMERGE algorithm

```
SYMMERGE (A, first1, first2, last)
  if first1 < first2 and first2 < last then
    m ← (first1 + last)/2
    n ← m + first2
    if first2 > m then
      start ← BSEARCH (A, n - last, m, n - 1)
    else
      start ← BSEARCH (A, first1, first2, n - 1)
    end ← n - start
    ROTATE (A, start, first2, end)
    SYMMERGE (A, first1, start, m)
    SYMMERGE (A, m, end, last)

BSEARCH (A, l, r, p)
  while l < r
    m ← (l + r) / 2
    if A[m] ≤A[p - m]
      then l ←  m + 1;
      else r ←  m;
  return l
```

as "active area". This active area has the same size as the shorter of either input sequences. The decomposition formulated by the steps (a2) and (b2) can be achieved efficiently by applying the principle of the symmetric comparisons between the shorter sequence u (or v) and the active area w. After the decomposition step (a2) (or (b2)), the subsequence $u_2v_1w_1$(or $w_2u_2v_1$) is rotated so that we get the subsequences $u_1v_1w_1$ and $u_2w_2v_2$($u_1w_1v_1$ and $w_2u_2v_2$). The treatment of pairs of equal elements as part of the "outer blocks" (u_1, w_2 in (a2) and w_1, v_2 in (b2)) avoids the exchange of equal elements and so any reordering of these.

Corollary 1. SYMMERGE *is stable.*

Algorithm 1 gives an implementation of the SYMMERGE algorithm in Pseudocode. The Pseudocode conventions are taken from [14].

3 Worst Case Complexity

We will now investigate the worst case complexity of SYMMERGE regarding the number of comparisons and assignments.
Unless stated otherwise, let us denote $m = |u|$, $n = |v|$, $m \leq n$, $k = \lfloor \log m \rfloor$ and let m_j^i and n_j^i denote the minimum and maximum of lengths of sequences

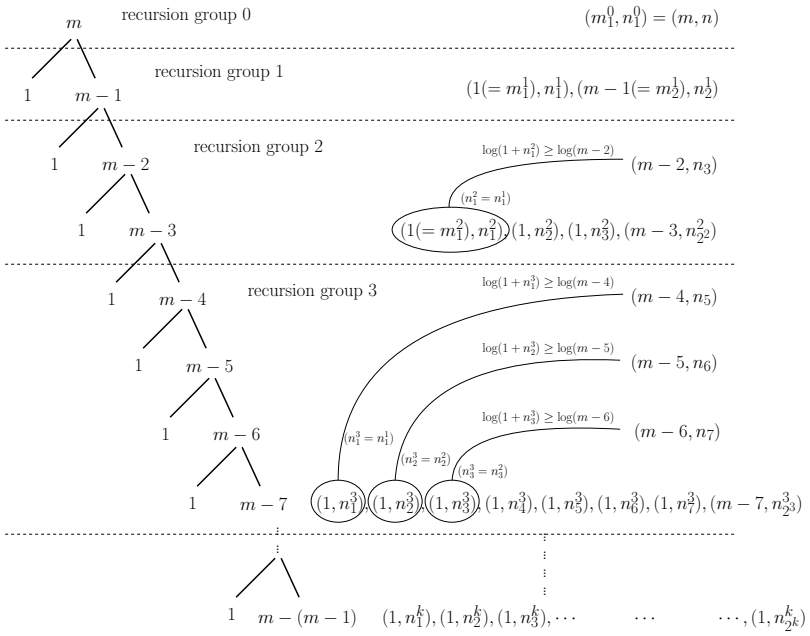

Fig. 3. Maximum spanning case

merging on the ith recursion group for $i = 0, 1, \ldots, k$ and $j = 1, 2, 3, \ldots, 2^i$ (initially $m_1^0 = m$ and $n_1^0 = n$). A recursion group consists of one or several recursion levels and comprises $2^i (i = 0, 1, \ldots, k)$ subsequence mergings at most (see figure 3). In the special case where each subsequence merging always triggers two nonempty recursive calls - in this case the recursion depth becomes exactly $k = \lfloor \log m \rfloor$ -, recursion groups and recursion levels are identical, but in general for the recursion depth dp it holds $k = \lfloor \log m \rfloor \leq dp \leq m$. Further, for each recursion group $i = 0, 1, \ldots, k$, it holds $\sum_{j=1}^{2^i} (m_j^i + n_j^i) = m + n$.

Lemma 1. *([11] Lemma 3.1) If $k = \sum_{j=1}^{2^i} k_j$ for any $k_j > 0$ and integer $i \geq 0$, then $\sum_{j=1}^{2^i} \log k_j \leq 2^i \log(k/2^i)$.*

Theorem 1. *The* SYMMERGE *algorithm needs $O(m \log(n/m+1))$ comparisons.*

Proof. The number of comparisons for the binary search for the recursion group 0 is equal to $\lfloor \log m \rfloor + 1 \leq \lfloor \log m + n \rfloor + 1$. For the recursion group 1 we need at most $\log(m_1^1 + n_1^1) + 1 + \log(m_2^1 + n_2^1) + 1$ comparisons, and so on. For the recursion group i we need at most $\sum_{j=1}^{2^i} \log(m_j^i + n_j^i) + 2^i$ comparisons. Since $\sum_{j=1}^{2^i} (m_j^i + n_j^i) = m+n$, it holds $\sum_{j=1}^{2^i} \log(m_j^i + n_j^i) + 2^i \leq 2^i \log((m+n)/2^i) + 2^i$ by Lemma 1. So the

overall number of comparisons for all $k+1$ recursion groups is not greater than $\sum_{i=0}^{k}(2^i+2^i\log((m+n)/2^i)) = 2^{k+1}-1+(2^{k+1}-1)\log(m+n)-\sum_{i=0}^{k} i2^i$. Since $\sum_{i=0}^{k} i2^i = (k-1)2^{k+1}+2$, algorithm SYMMERGE needs at most $2^{k+1}-1+(2^{k+1}-1)\log(m+n)-(k-1)2^{k+1}-2 \leq 2^{k+1}\log(m+n)-k2^{k+1}+2^{k+2}-\log(m+n)-3 \leq 2m(\log\frac{m+n}{m}+2)-\log(m+n)-3 = O(m\log(\frac{n}{m}+1))$ comparisons. □

Theorem 2. *The recursion-depth of* SYMMERGE *is bounded by* $\lceil\log(m+n)\rceil$.

Proof. The decomposition steps (1a) and (1b) satisfy the property $\max\{|u_1|, |u_2|, |v_1|, |v_2|, |w_1|, |w_2|\} \leq (|u|+|v|)/2$. So, on recursion level $\lceil\log(m+n)\rceil$ all remaining unmerged subsequences are of length 1. □

Corollary 2. SYMMERGE *is a minimum storage algorithm.*

In [11] Dudzinski and Dydek presented an optimal in-place rotation (sequence exchange) algorithm, that needs $m+n+\gcd(m,n)$ element assignments for exchanging two sequences of lengths m and n.

Theorem 3. *If we take the rotation algorithm given in [11], then* SYMMERGE *requires* $O((m+n)\log m)$ *element assignments.*

Proof. Inside each recursion group $i = 0, 1, \ldots, k$ disjoint parts of u are merged with disjoint parts of v. Hence each recursion group i comprises at most $\sum_{j=1}^{i}((m_j^i+n_j^i)+\gcd(m_j^i,n_j^i)) \leq m+n+\sum_{j=1}^{2^i} m_j^i = 2m+n$ assignments resulted from rotations. So the overall number of assignments for all k recursion groups is less than $(2m+n)(k+1) = (2m+n)\log m+2m+n = O((m+n)\log m)$. □

4 Practical Results

We did some experimental work with the unfolded version of the SYMMERGE algorithm (see Sect. 4.1) and compared it with the implementations of three other merging algorithms. As first competitor we chose the merge_without_buffer-function contained in the C++ Standard Template Libraries (STL) [12]. This function implements a modified version of the RECMERGE algorithm devised by Dudzinski and Dydek [11]. The STL-algorithm operates in a mirrored style compared to RECMERGE and doesn't release one element in each recursion step. Fig. 4 gives a graphical description of the differences. We preferred the STL-variant because of its significance in practice.
The second competitor was taken from [15], where a simplified implementation of the in-place algorithm from Mannila & Ukkonen [7] is given. Unlike the original algorithm the simplified version relies on a small external buffer whose length is restricted by the square root of the input size.

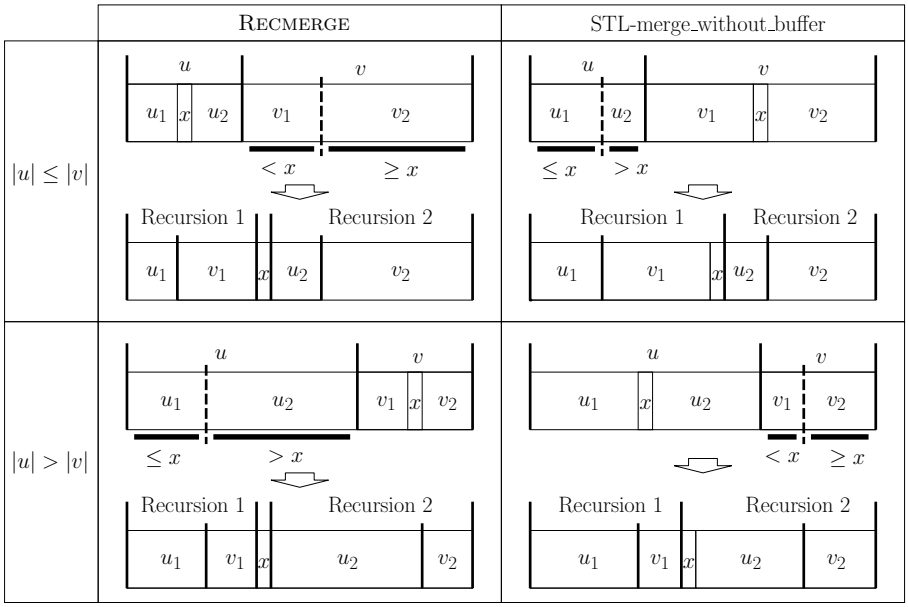

Fig. 4. RECMERGE versus STL-merge_without_buffer

As third competitor we took the classical standard algorithm. We chose randomly generated sequences of integers as input. The results of our evaluation are contained in Table 1, each entry shows a mean value of 30 runs with different data. We took a state of the art hardware platform with 2.4 Ghz processor speed and 512MB main memory; all coding was done in the C-programming language. Each comparison was accompanied by a slight artificial delay in order to strengthen the impact of comparisons on the execution time.
It can be read from the table that the STL-algorithm seems to be less efficient than SYMMERGE. However both algorithm show an accurate "$O(m \log(n/m))$-behavior".
SYMMERGE and RECMERGE rely on rotations as encapsulated operations for element reordering. There are several algorithms that achieve rotations, three of them are presented and evaluated by Bentley in [16]. The evaluation by Bentley showed, that a rotation algorithm proposed by Dudzinski and Dydek in [11] is rather slow compared to its alternatives, although it is optimal with respect to the number of assignments. Nevertheless this algorithm was chosen in the STL for the implementation of rotations. We exchanged the rotation algorithm of the merge_without_buffer-function (STL) by one of its faster alternatives and compared the modified function with its original. The result of this comparisons is given in the columns $\text{STL}_{modified}$ and STL_{orig} of Table 1. From this comparison it can be read that the chosen rotation algorithm is one of the driving factors regarding the performance of RECMERGE, and so of SYMMERGE.

n	m	SYMMERGE		$\text{STL}_{modified}$		STL_{orig}	Mannila & Ukkonen		Standard	
		$\#comp$	t_e	$\#comp$	t_e	t_e	$\#comp$	t_e	$\#comp$	t_e
2^{21}	2^{21}	5217738	*1831*	6020121	*2263*	*3408*	7464568	*1483*	4194177	*743*
2^{21}	2^{18}	1348902	*635*	1541524	*751*	*1398*	5615225	*1116*	2359159	*415*
2^{21}	2^{15}	264279	*298*	295063	*334*	*815*	5301311	*1049*	2129765	*372*
2^{21}	2^{12}	45227	*211*	49272	*231*	*607*	4660337	*942*	2100744	*373*
2^{23}	2^{9}	8198	*680*	8711	*777*	*2735*	14909001	*3101*	8374604	*1503*
2^{23}	2^{6}	1212	*579*	1279	*673*	*1254*	14297482	*2996*	8292870	*1496*
2^{23}	2^{3}	170	*465*	179	*548*	*196*	14202829	*2967*	7544491	*1402*
2^{23}	2^{0}	23	*191*	24	*222*	*31*	14188252	*2959*	4040739	*832*

t_e : Execution time in ms, $\#comp$: Number of comp., m, n : Lengths of inp. seq.

Table 1. Practical comparison of various merging algorithms

4.1 Optimisations on Pseudocode-Level

We would like to remark two optimisations of SYMMERGE on Pseudocode-Level that can be applied for decreasing the execution time without changing the number of comparisons and element assignments carried out. First of all, "needless recursive calls", i.e. recursive calls with $m = 0$ or $n = 0$, can be avoided by unfolding the algorithm. For the unfolded version the caller has to ensure that SYMMERGE is called with $m \geq 1$ and $n \geq 1$. A second optimisation is the treatment of the case $m = 1$ and $n \geq 1$ as loop-driven direct binary insertion. This avoids $\lfloor \log n + 1 \rfloor$ recursive calls of the algorithm in this special case.
The impact of both optimizations on the execution time depends on the sequences to be merged. In our environment we could observe a time-reduction up to 25%.

5 Conclusion

We presented an efficient minimum storage merging algorithm called SYMMERGE. Our algorithm uses a novel technique for merging that relies on symmetric comparisons as central operation. We could prove that our algorithm is asymptotically optimal regarding the number of necessary comparisons. Practical evaluation could show that it is fast and efficient. So, SYMMERGE is not only of theoretical but also of practical interest.
Finally let us note that our algorithm, unlike the standard algorithm, can take advantage of the sortedness of the input sequences. If the overall input sequence is in sorted order, i.e. $u \leq v$ for two sequences u and v of sizes m and n, SYMMERGE needs only $O(\log(m + n))$ comparisons and hence becomes sub-linear. This in turn reflects to a SYMMERGE based Merge-sort, which speeds up for presorted sequences as well. Mehlhorn showed in [17] that sequences of size n with $O(n)$ inversions, i.e. with $O(n)$ pairs of elements that are not in sorted order,

can be sorted in time $O(n)$. How many comparisons does a SYMMERGE based Merge-sort need depending on the number of inversions? We would like to leave this question to the further research.

Acknowledgment

We are grateful to a referee whose careful reading and valuable remarks helped to improve the contents and presentation of the paper.

References

1. Knuth, D.E.: The Art of Computer Programming. Volume Vol. 3: Sorting and Searching. Addison-Wesley (1973)
2. Kronrod, M.A.: An optimal ordering algorithm without a field operation. Dokladi Akad. Nauk SSSR **186** (1969) 1256–1258
3. Geffert, V., Katajainen, J., Pasanen, T.: Asymptotically efficient in-place merging. Theoretical Computer Science **237** (2000) 159–181
4. Huang, B.C., Langston, M.: Practical in-place merging. Communications of the ACM **31** (1988) 348–352
5. Huang, B.C., Langston, M.: Fast stable merging and sorting in constant extra space. The Computer Journal **35** (1992) 643–650
6. Chen, J.: Optimizing stable in-place merging. Theoretical Computer Science **302** (2003) 191–210
7. Mannila, H., Ukkonen, E.: A simple linear-time algorithm for in situ merging. Information Processing Letters **18** (1984) 203–208
8. Pardo, L.T.: Stable sorting and merging with optimal space and time bounds. SIAM Journal on Computing **6** (1977) 351–372
9. Symvonis, A.: Optimal stable merging. Computer Journal **38** (1995) 681–690
10. Salowe, J., Steiger, W.: Simplified stable merging tasks. Journal of Algorithms **8** (1987) 557–571
11. Dudzinski, K., Dydek, A.: On a stable storage merging algorithm. Information Processing Letters **12** (1981) 5–8
12. C++ Standard Template Library: (http://www.sgi.com/tech/stl)
13. Ellis, J., Markov, M.: In situ, stable merging by way of the perfect shuffle. The Computer Journal **43** (2000) 40–53
14. Cormen, T., Leiserson, C., Rivest, R., Stein, C.: Introduction to Algorithms. 2nd edn. MIT Press (2001)
15. Møllerhøj, K., Søttrup, C.: Undersøgelse og implementation af effektiv inplace merge. Technical report, CPH STL Reports 2002-06, Department of Computer Science, University of Copenhagen, Denmark (2002)
16. Bentley, J.: Programming Pearls. 2nd edn. Addison-Wesley, Inc (2000)
17. Mehlhorn, K.: Sorting presorted files. In: Proc. 4th GI Conf. on Theoretical Comp. Science (Aachen) - Lect. Notes in Comp. Science 67. Springer (1979) 199–212