A Simple Algorithm for Stable Minimum Storage Merging

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 contribute to the research on stable minimum storage merging by introducing an algorithm that is particularly simply structured compared to its competitors. The presented algorithm performs $O(m \log(\frac{n}{m}+1))$ comparisons and $O((m+n)\log m)$ assignments, where m and n are the sizes of the input sequences with $m \leq n$. Hence, according to the lower bounds of merging the algorithm is asymptotically optimal regarding the number of comparisons.
As central new idea we present a principle of symmetric splitting, where the start and end point of a rotation are computed by a repeated halving of two search spaces. This principle is structurally simpler than the principle of symmetric comparisons introduced earlier by Kim and Kutzner. It can be transparently implemented by few lines of Pseudocode.
We report concrete benchmarks that prove the practical value of our algorithm.

1 Introduction

Merging denotes the operation of rearranging the elements of two adjacent sorted sequences of size 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 requires $O(\log^2(m+n))$ bits additional space at most. It is regarded as *stable*, if it preserves the initial ordering of elements with equal value.
There are two significant lower bounds for merging. The lower bound for the number of assignments is $m+n$ because every element of the input sequences can change its position in the sorted output. As shown by Knuth in [1] the lower bound for the number of comparisons is $\Omega(m \log(\frac{n}{m}+1))$, where $m \leq n$.
The Recmerge algorithm of Dudzinski and Dydek [2] and the Symmerge algorithm of Kim and Kutzner [3] are two minimum storage merging algorithms that have been proposed in the literature so far. Both algorithms are asymptotically optimal regarding the number of comparisons and resemble structurally. They perform the merging by a binary partitioning of both input sequences

* This work was supported by the Kookmin University research grant in 2006.

which operates as the foundation of a rotation that is followed by two recursive calls. The algorithm proposed here operates similar, however the partitioning is performed by a novel technique called *symmetric splitting*. This partitioning technique is structurally simpler than the older ones, because it neither requires the detection of the shorter input sequence nor a binary search as sub operation. Further there is no static selection of any pivot element or centered subsequence. The simplicity leads to a highly transparent and well understandable algorithm that can be implemented in a few lines of Pseudocode. Despite its simplicity our algorithm is asymptotically optimal regarding the number of comparisons and requires $O((m+n)\log m)$ assignments.
Another class of merging algorithms is the class of *in place* merging algorithms, where the external space is restricted to a constant amount merely. Recent work in this area are the publications [4–7], that describe algorithms which are all asymptotically optimal regarding the number of comparisons as well as assignments. However, these algorithms are structurally quite complex and rely heavily on other concepts, as e.g. Kronrod's idea of an internal buffer [8], Mannila and Ukkonen's technique for block rearrangements [9] and Hwang and Lin's merging algorithm [10]. We included the stable in place merging algorithm proposed in [7] into our benchmarking in order to give an impression of the performance behavior of the different approaches.

We will start with a formal definition of our algorithm together with the presentation of a corresponding Pseudocode implementation. Afterwards we will prove that our algorithm is stable, minimum storage and asymptotically optimal regarding the number of comparisons. In a benchmark section we show that our algorithm performs well compared to its competitors. We will finish with a conclusion, where we give some ideas for further research.

2 Formal Definition / Pseudocode Implementation

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 $y \in v$.

The Principle of Symmetric Splitting
The algorithm presented her relies on a quite simple idea, the *principle of symmetric splitting*. Informally this principle can be described as follows:
Let u and v be our two input sequences. By a repeated halving of two search spaces we compute separations $u \equiv u'u''$ and $v \equiv v'v''$ so that we get $u'' > v'$ and $u' <= v''$. To do so we start with u and v as our initial search spaces and operate as follows:
We split our search spaces at their middle elements and compare these two elements. Depending on the result of this comparison we symmetrically shorten these spaces either to the outside or to the inside starting at the respective middle elements. In this way we repeatedly halve our search spaces until these are reduced to single points. These points then indicate the borders of a rotation.

Because of this principle of splitting our algorithm is called Splitmerge.

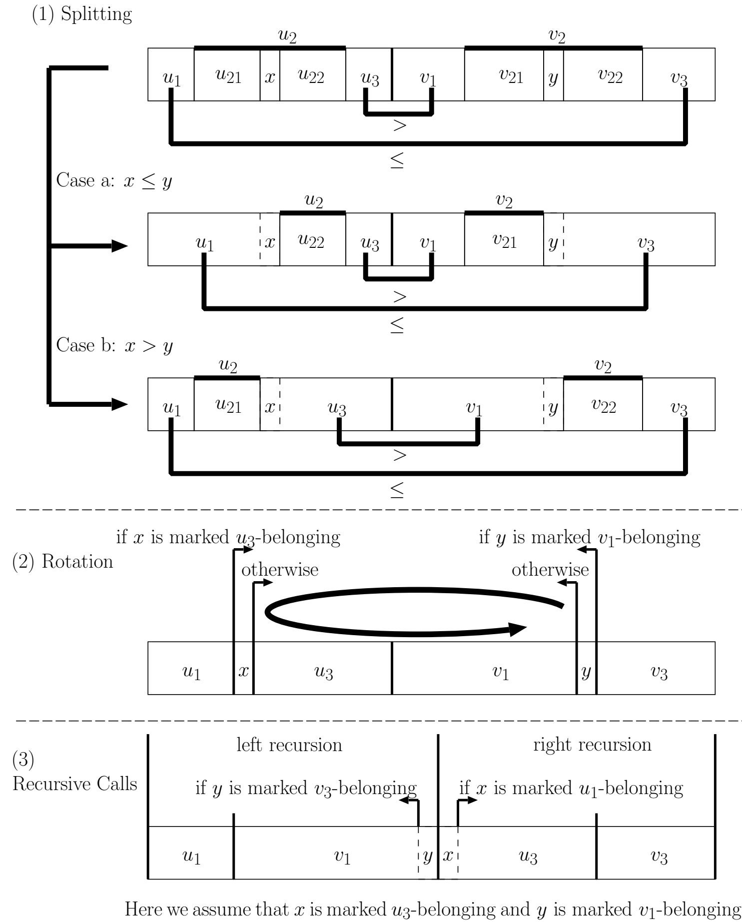

Fig. 1. Graphical description of the Splitmerge algorithm

Definition of the Splitmerge Algorithm

We now give a formal definition of our merging process, where Fig. 1 contains an additional graphical description.

Initially we decompose u into $u_1 u_2 u_3$ with $u_2 \equiv u$ and u_1 as well as u_3 is empty. Just the same way we decompose v into $v_1 v_2 v_3$ with $v_2 \equiv v$ and v_1 as well as v_3 is empty.

(1) We split u_2 into $u_{21} x u_{22}$, so that $|u_{21}| = |u_{22}|$ or $|u_{21}| = |u_{22}| + 1$ and we split v_2 into $v_{21} y v_{22}$, so that $|v_{21}| = |v_{22}|$ or $|v_{21}| = |v_{22}| + 1$.

Algorithm 1 SplitMerge Algorithm

SplitMerge($A, first1, first2, last$)
    ▷ u is in $A[first1 : first2 - 1]$, v is in $A[first2 : last - 1]$
1 **if** $first1 \geq first2$ or $first2 \geq last$
2   **then return**
3
4 $l \leftarrow first1;\ r \leftarrow first2;\ l' \leftarrow first2;\ r' \leftarrow last$
5 **repeat** **if** $l < r$
6   **then** $m \leftarrow (l + r)/2$
7   **if** $l' < r'$
8   **then** $m' \leftarrow (l' + r')/2$
9
10   **if** $A[m] \leq A[m']$
11   **then** $l \leftarrow m + 1;\ r' \leftarrow m'$
12   **else** $l' \leftarrow m' + 1;\ r \leftarrow m$
13   **until** $l \geq r$ and $l' \geq r'$
14
15 Rotate(A, r, $first2$, l')
16 SplitMerge(A, $first1$, r, $r + r' - first2$)
17 SplitMerge(A, $l + l' - first2$, l', $last$)

If $x <= y$ then

(a) If u_{22} is nonempty, then we extend u_1 to $u_1u_{21}x$ and we reduce u_2 to u_{22}, otherwise we extend u_1 to u_1u_{21}, set $u_2 \equiv x$ and mark x as u_1-belonging.
If v_{21} is nonempty, then we extend v_3 to $yv_{22}v_3$ and we reduce v_2 to v_{21}, otherwise we extend v_3 to $v_{22}v_3$, set $v_2 \equiv y$ and mark y as v_3-belonging.

otherwise

(b) If u_{21} is nonempty, then we extend u_3 to $xu_{22}u_3$ and we reduce u_2 to u_{21}, otherwise we extend u_3 to $u_{22}u_3$, set $u_2 \equiv x$ and mark x as u_3-belonging.
If v_{22} is nonempty, then we extend v_1 to $v_1v_{21}y$ and we reduce v_2 to v_{22}, otherwise we extend v_1 to v_1v_{21}, set $v_2 \equiv y$ and mark y as v_1-belonging.

We continue the above splitting until x and y are marked.

(2) If x is marked u_3-belonging, then let $u'_1 \equiv u_1$ and $u'_3 \equiv xu_3$ else let $u'_1 \equiv u_1x$ and $u'_3 \equiv u_3$. If y is marked v_1-belonging then let $v'_1 \equiv v_1y$ and $v'_3 \equiv v_3$ else let $v'_1 \equiv v_1$ and $v'_3 \equiv yv_3$. We rotate $u'_1u'_3v'_1v'_3$ to $u'_1v'_1u'_3v'_3$.
(3) We recursively merge u'_1 with v'_1 and u'_3 with v'_3, where we shorten v'_1 to v_1 in the case that y is marked v_3-belonging and we shorten u'_3 to u_3 in the case that x is marked u_1-belonging.

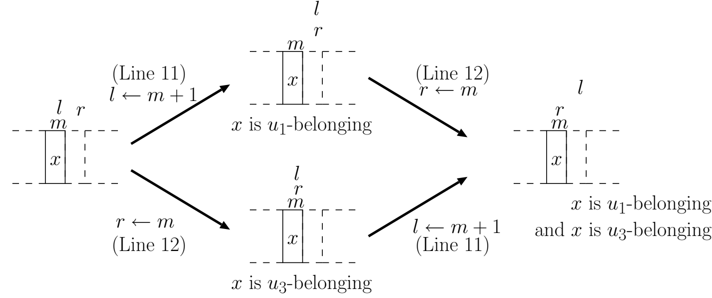

Fig. 2. The meaning of the variables l and r in the case $l \geq r$

Pseudocode Implementation of Splitmerge
A Pseudocode implementation for the above algorithm is given in Alg. 1, where the code notation is taken from [11]. The following remarks shall ease the understanding of the correspondence between Pseudocode and formal definition: The inputs u and v are passed in $A[first1 : first2-1]$ and $A[first2 : last-1]$, respectively. The symmetric splitting happens in the code section from line 4 to line 13. In the case of $l < r$ the subsequence u_1 is in $A[first1 : l-1]$, u_2 is in $A[l : r-1]$ and u_3 is in $A[r : first2-1]$. m holds the index of the element x, i.e. $A[m] = x$. In the case of $l \geq r$ the search space u_2 has been reduced to x merely and we have u_1 in $A[first1 : m-1]$, u_3 in $A[m+1 : first2-1]$. l together with r are now used to store markings as shown in Fig. 2. In the same way l' and r' store the splitting into v_1, v_2, v_3 or marking information. m' is the index of y. The lines 15 to 17 contain the rotation and the recursive calls.

3 Stability, Minimum Storage Property, Complexity

Lemma 1. *During the splitting process* $u_1 \leq v_3$ *and* $u_3 > v_1$ *always hold.*

Proof. At the beginning of the splitting u_1, u_3, v_1, v_3 are all empty. So the properties trivially hold. In the case of $x \leq y$ we have $u_{21}x \leq yv_{22}$, so after the extension step we preserve $u_1 \leq v_3$ (u_3 and v_1 stay untouched). In the other case we have $xu_{22} > v_{21}y$, so we preserve $u_3 > v_1$ (u_1 and v_3 stay untouched). □

Additionally the following holds: If y is marked v_1-belonging, we have $u_3 > v_1y$. If x is marked u_3-belonging, we have $xu_3 > v_1$ or even $xu_3 > v_1y$ if y is additionally marked v_1-belonging.

Corollary 1. Splitmerge *is stable.*

Lemma 2. *The recursion depth is limited by* $\min\{\lfloor \log m \rfloor + \lfloor \log n \rfloor, m-1\}$

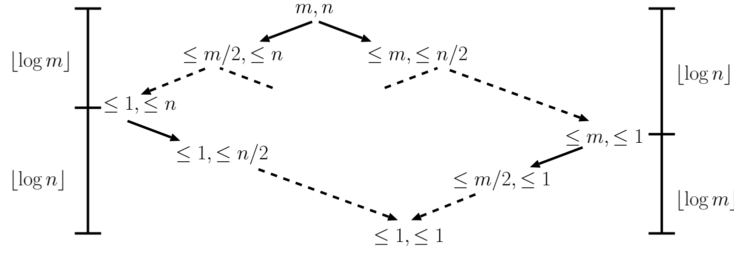

Fig. 3. Recursion Depth

Proof. We prove both upper bounds separately.
(1) After the splitting (step (2) in the formal definition) we get either $|u'_3| \leq m/2$ and $|v'_1| \leq n/2$ or $|u'_1| \leq m/2$ and $|v'_3| \leq n/2$. This in turn implies $(\leq m/2, \leq n)$ and $(\leq m, \leq n/2)$ as sizes of the two recursive calls. Hence according to figure 3 the recursion depth is limited by $\lfloor \log m \rfloor + \lfloor \log n \rfloor$.
(2) $u'_1 \equiv u$ implies that we did not touch the (b)-alternative during the splitting. This in turn implies that v'_1 is empty. In the opposite case ($u'_3 \equiv u$) we have to distinguish two alternatives: Either we did not touch the (a)-alternative and so v'_3 is empty or we touched the (a)-alternative with empty u_{22} as well as empty u_{21} and marked x as u_1-belonging. In the latter case we get a recursion where u'_3 is shorten by one element. So the shorter side loses at least one element with every successful recursive invocation and the overall recursion depth is limited by $m - 1$. □

Since $m \leq n$, the following corollary holds:

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

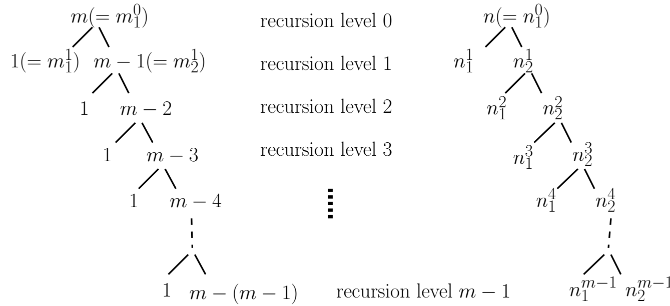

Fig. 4. Maximum spanning case

Complexity
Unless stated otherwise, let us denote $m = |u|$, $n = |v|$ with $m \leq n$ and let

$k = \lfloor \log m \rfloor + 1$ if $2^{k-1} < m < 2^k$ or $k = \log m$ if $m = 2^k$. Further m_j^i and n_j^i denote sizes of sequences merged on the ith recursion level (initially $m_1^0 = m$ and $n_1^0 = n$).

Lemma 2 shows that the recursion depth is limited by $m - 1$. We will now consider the relationship between m and n for the maximum spanning case, the case where the recursion depth is $m - 1$. Here (m, n) can be partitioned to either $(1\,(= m_1^1),\, n_1^1))$ and $(m-1\,(= m_2^1),\, n_2^1)$ or $(m-1\,(= m_1^1),\, n_1^1)$ and $(1\,(= m_2^1),\, n_2^1))$ merely. If there are other partitions with $1 < m_1^1, m_2^1 < m - 1$, then the algorithm may reach at most the recursion depth $m - 2\ (= m - 2 - 1 + 1)$. Without loss of generality we suppose that (m, n) is partitioned to $(1\,(= m_1^1),\, n_1^1))$ and $(m - 1\,(= m_2^1),\, n_2^1)$ on recursion level 1. Since the SPLITMERGE algorithm applies the symmetric splitting principle, it must be satisfied that $n_1^1 \geq n - \frac{n}{2^{\lfloor \log m \rfloor}}$ and $n_2^1 < \frac{n}{2^{\lfloor \log m \rfloor}}$ (if $m = 2^k$, then $n_2^1 < \frac{n}{2^{\lfloor \log m \rfloor}} = \frac{n}{m}$). Further if $m - 1 > n_2^1$, the recursion depth would be smaller than $m - 1$. Thus $m - 1 \leq n_2^1$. Here $m - 1 \leq n_2^1$ and $n_2^1 < \frac{n}{2^{\lfloor \log m \rfloor}}$ implies $2^{\lfloor \log m \rfloor} \cdot (m - 1) < n$. Suppose that, just as on the first recursion level, $(m - 1\,(= m_2^1),\, n_2^1)$ is again partitioned to $(1,\, n_1^2))$ and $(m - 2,\, n_2^2)$ on the second recursion level. Then $n_1^2 \geq \frac{n}{2^{\lfloor \log m \rfloor}} - \frac{n}{2^{\lfloor \log m \rfloor}.2^{\lfloor \log(m-1) \rfloor}}$, $n_2^2 < \frac{\frac{n}{2^{\lfloor \log m \rfloor}}}{2^{\lfloor \log(m-1) \rfloor}} = \frac{n}{2^{\lfloor \log m \rfloor}.2^{\lfloor \log(m-1) \rfloor}}$ and $m - 2 \leq n_2^2$. Thus from $m - 2 \leq n_2^2$ and $n_2^2 < \frac{n}{2^{\lfloor \log m \rfloor}.2^{\lfloor \log(m-1) \rfloor}}$ it holds $2^{\lfloor \log m \rfloor} \cdot 2^{\lfloor \log(m-1) \rfloor} \cdot (m - 2) < n$. On the i^{th}recursion level, suppose $(m - (i - 1)\,(= m_2^{i-1}),\, n_2^{i-1})$ is partitioned to $(1,\, n_1^i)$ and $(m - i,\, n_2^i)$. Then $n_1^i \geq \frac{n}{2^{\lfloor \log m \rfloor}.2^{\lfloor \log(m-1) \rfloor}\ldots 2^{\lfloor \log(m-i+2) \rfloor}} - \frac{n}{2^{\lfloor \log m \rfloor}.2^{\lfloor \log(m-1) \rfloor}\ldots 2^{\lfloor \log(m-i+2) \rfloor}.2^{\lfloor \log(m-i+1) \rfloor}}$, $n_2^i < \frac{\frac{n}{2^{\lfloor \log m \rfloor}.2^{\lfloor \log(m-1) \rfloor}\ldots 2^{\lfloor \log(m-i+2) \rfloor}}}{2^{\lfloor \log(m-i+1) \rfloor}} = \frac{n}{2^{\lfloor \log m \rfloor}.2^{\lfloor \log(m-1) \rfloor}\ldots 2^{\lfloor \log(m-i+2) \rfloor}.2^{\lfloor \log(m-i+1) \rfloor}}$ and $m - i \leq \frac{n}{2^{\lfloor \log m \rfloor}.2^{\lfloor \log(m-1) \rfloor}\ldots 2^{\lfloor \log(m-i+2) \rfloor}.2^{\lfloor \log(m-i+1) \rfloor}}$ i. e. $2^{\lfloor \log m \rfloor} \cdot 2^{\lfloor \log(m-1) \rfloor} \ldots 2^{\lfloor \log(m-i+1) \rfloor} \cdot (m - i) < n$, and so on. Hence, to reach the recursion depth $m-1$, we need the assumption $2^{\lfloor \log m \rfloor} \cdot 2^{\lfloor \log(m-1) \rfloor} \cdot 2^{\lfloor \log(m-2) \rfloor} \ldots 2^{\lfloor \log 1 \rfloor} < n$ and can state the following theorem:

Theorem 1. *If the* SPLITMERGE *algorithm reaches the recursion level* $m - 1$ *for two input sequences of sizes* m, n $(m \leq n)$*, then* $n > 2^{\lfloor \log m \rfloor} \cdot 2^{\lfloor \log(m-1) \rfloor} \cdot 2^{\lfloor \log(m-2) \rfloor} \ldots 2^{\lfloor \log 1 \rfloor}$.

We will now investigate the worst case complexity of the SPLITMERGE algorithm regarding the number of comparisons and assignments. Fig. 4 shows the partitions in the maximum spanning case. Note that on the recursion level i, a sequence of length $m_1^i = 1$ $(m_2^i = m - i)$ is merged with a sequence of length n_1^i (n_2^i).

Lemma 3. *([2] 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 2. *The* SPLITMERGE *algorithm needs* $O(m \log(n/m + 1))$ *comparisons.*

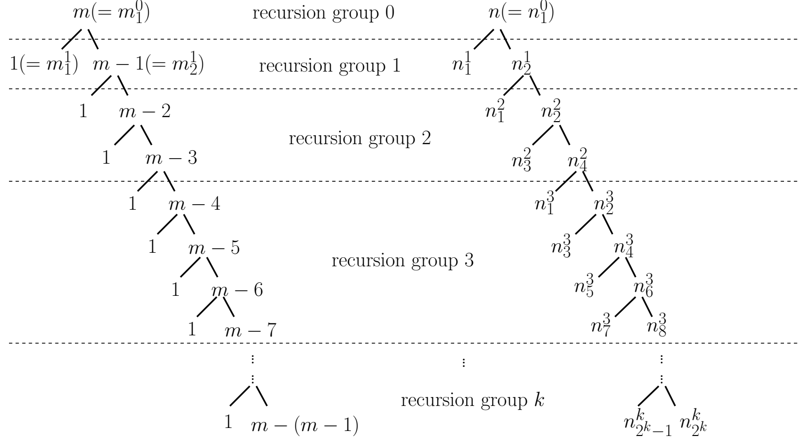

Fig. 5. Construction of recursion groups

Proof. Lemma 2 shows that the recursion depth is limited by $m-1$ (Note that if $m = 2^k$then $m-1 = 2^0 + 2^1 + 2^2 + \cdots + 2^{k-1} = 2^0 + 2^1 + 2^2 + \cdots + 2^{\log m - 1}$). We group the recursion levels into $k+1$ recursion groups, say recursion group 0, recursion group 1, $\cdots$, recursion group k, so that each recursion group $i\,(i = 1, 2, \cdots, k)$ holds at most 2^{i-1} recursion levels (see Fig. 5). Till now m^i_j and n^i_j denoted the lengths of sequences merged on the ith recursion level. From now on we change the meaning of indexes so that m^i_j and n^i_j denote the lengths of sequences merged on the ith recursion group. Then there are at most 2^ipartitions in each recursion group $i\,(i = 1, 2, \cdots, k)$, say $(m^i_1,\, n^i_1)$, $(m^i_2,\, n^i_2)$, $\cdots$ $(m^i_{2^i},\, n^i_{2^i})$. Thus the number of comparisons for symmetric splitting with the recursion group 0 is equal to $\lfloor \log n \rfloor + 1 \leq \lfloor \log(m+n) \rfloor + 1$. For the recursion group 1 we need $\lfloor \max(\log m^1_1, \log n^1_1) \rfloor + 1 + \lfloor \max(\log m^1_2, \log n^1_2) \rfloor + 1 \leq \log(m^1_1 + n^1_1) + 1 + \log(m^1_2 + n^1_2) + 1$ comparisons, and so on. For the recursion group i we need at most $\sum_{j=1}^{2^i} \log(m^i_j + n^i_j) + 2^i$ comparisons. Since for each recursion group $i\,(i = 1, 2, \cdots, k)$ $\sum_{j=1}^{2^i}(m^i_j + n^i_j) \leq m+n$, it holds $\sum_{j=1}^{2^i} \log(m^i_j + n^i_j) + 2^i \leq 2^i \log((m+n)/2^i) + 2^i$ by Lemma 3. Note the following special case: if each merging of subsequences triggers two nonempty recursive calls, the recursion level becomes exactly k and recursion groups and recursion levels are identical. In this case each ith recursion level comprises $2^i (i = 0, 1, \ldots, k)$ subsequence mergings and for each recursion group (level) $i = 0, 1, \ldots, k$, it holds $\sum_{j=1}^{2^i}(m^i_j + n^i_j) = m+n$. Therefore we need at most $\sum_{j=1}^{2^i} \log(m^i_j + n^i_j) + 2^i \leq 2^i \log((m+n)/2^i) + 2^i$ comparisons as well. 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 $\sum_{i=0}^{k} i2^i = (k-1)2^{k+1} + 2$, the SPLITMERGE algorithm needs at most $2^{k+1} - 1 + (2^{k+1} - 1)\log(m+n) - (k-1)2^{k+1} - 2 = 2^{k+1}\log(m+n) - k2^{k+1} +$

n, m	i	St.-In-Pl.-Merge		Recmerge		Symmerge		Splitmerge	
		$\#comp$	t_e	$\#comp$	t_e	$\#comp$	t_e	$\#comp$	t_e
2^{23}	2^{24}	25193982	6774	18642127	12864	21285269	11841	21986651	11587
2^{21}	2^{22}	6307320	1652	4660230	2457	5320495	2093	5496000	2128
2^{19}	2^{20}	1582913	395	1165009	402	1329813	359	1373814	349
2^{19}	2^{16}	1854321	406	962181	311	863284	241	837843	216
2^{19}	2^{12}	2045316	307	263410	289	196390	196	119072	187
2^{19}	2^{8}	1225279	97	38401	283	27917	164	11478	159
2^{19}	2^{4}	1146326	107	4409	276	1477	83	927	60
2^{19}	2^{1}	786492	34	687	556	55	16	91	14

n, m : Lengths of input sequences ($m = n$) i : Number of different input elements
t_e : Execution time in ms, $\#comp$: Number of comparisons

Table 1. Runtimes of different merging algorithms

$2^{k+2} - \log(m+n) - 3 = 2m(\log \frac{m+n}{m} + 2) - \log(m+n) - 3 = O(m \log(\frac{n}{m} + 1))$ comparisons. □

Corollary 3. *The* Splitmerge *algorithm is asymptotically optimal regarding the number of comparisons.*

Regarding the sizes of merged sequences theorem 2 states $\sum_{j=1}^{2^i}(m_j^i + n_j^i) \leq m+n$ for all recursion groups i ($i = 0, 1, \cdots, k$). Hence, if we take the optimal rotation algorithm proposed in [2], we perform $O(m+n)$ assignments on every recursion group. Because we have at most $k+1$ recursion groups the following theorem holds:

Theorem 3. *The* Splitmerge *algorithm needs* $O((m+n) \log m)$ *assignments.*

4 Experimental Work / Benchmarking

We did some benchmarking for the Splitmerge algorithm, in order to get an impression of its practical value. We compared our algorithm with Dudzinsky and Dydek's Recmerge [2] algorithm, Kim and Kutzner's Symmerge [3] algorithm and the asymptotically optimal in place merging algorithm proposed in [7]. For rotations we generally used the rotation algorithm proposed in [2] that is optimal with respect to the number of assignments. Table 1 contains a summary of our results. Each entry shows a mean value of 30 runs with different random data. We took a state of the art hardware platform with 2 Ghz processor speed and 512MB main memory, all coding was done in the C programming language, all compiler optimizations had been switched of.
The benchmarks show that Splitmerge can fully compete with Recmerge and Symmerge. Please note, that despite a slightly higher number of comparisons our algorithm performs a bit better than its two sisters. This seems to be due

to SPLITMERGE's simpler structure. The second column of Table 1 shows the number of different elements in both input sequences. Regarding their runtime all algorithms can take more or less profit of a decreasing number of different elements in the input sequences. However, the effect is particular well visible with SPLITMERGE.

5 Conclusion

We presented a simply structured minimum storage merging algorithm called SPLITMERGE. Our algorithm relies on a novel binary partition technique called symmetric splitting and has a short implementation in Pseudcode. It requires $O(m \log(\frac{n}{m}+1))$ comparisons and $O((m+n)\log m)$ assignments, so it is asymptotically optimal regarding the number of comparisons. Our benchmarking proved that it is of practical interest.
During our benchmarking we observed that none of the investigated algorithms could claim any general superiority. We could always find input sequences so that a specific algorithm performed particularly well or bad. Nevertheless, we could recognize criteria that indicated the superiority of a specific algorithm for specific inputs. For example SPLITMERGE performs well if we have only few different elements in our input sequences. We plan more research on this topic in order to develop guidelines for a clever algorithm selection in the case of merging.

References

1. Knuth, D.E.: The Art of Computer Programming. Volume Vol. 3: Sorting and Searching. Addison-Wesley (1973)
2. Dudzinski, K., Dydek, A.: On a stable storage merging algorithm. Information Processing Letters **12** (1981) 5–8
3. Kim, P.S., Kutzner, A.: Stable minimum storage merging by symmetric comparisons. In Albers, S., Radzik, T., eds.: Algorithms - ESA 2004. Volume 3221 of Lecture Notes in Computer Science., Springer (2004) 714–723
4. Symvonis, A.: Optimal stable merging. Computer Journal **38** (1995) 681–690
5. Geffert, V., Katajainen, J., Pasanen, T.: Asymptotically efficient in-place merging. Theoretical Computer Science **237** (2000) 159–181
6. Chen, J.: Optimizing stable in-place merging. Theoretical Computer Science **302** (2003) 191–210
7. Kim, P.S., Kutzner, A.: On optimal and efficient in place merging. In Wiedermann, J., Tel, G., Pokorný, J., Bieliková, M., Stuller, J., eds.: SOFSEM 2006. Volume 3831 of Lecture Notes in Computer Science., Springer (2006) 350–359
8. Kronrod, M.A.: An optimal ordering algorithm without a field operation. Dokladi Akad. Nauk SSSR **186** (1969) 1256–1258
9. Mannila, H., Ukkonen, E.: A simple linear-time algorithm for in situ merging. Information Processing Letters **18** (1984) 203–208
10. Hwang, F., S.Lin: A simple algorithm for merging two disjoint linearly ordered sets. SIAM J. Comput. **1** (1972) 31–39
11. Cormen, T., Leiserson, C., Rivest, R., Stein, C.: Introduction to Algorithms. 2nd edn. MIT Press (2001)