Sorting *In-Place* with a *Worst Case* Complexity of $n \log n - 1.3n + O(\log n)$ Comparisons and $\varepsilon\, n \log n + O(1)$ Transports *

Klaus Reinhardt
Institut für Informatik, Universität Stuttgart
Breitwiesenstr.22, D-7000 Stuttgart-80, Germany
e-mail: reinhard@informatik.uni-stuttgart.de

Abstract

First we present a new variant of Merge-sort, which needs only $1.25n$ space, because it uses space again, which becomes available within the current stage. It does not need more comparisons than classical Merge-sort.

The main result is an easy to implement method of iterating the procedure in-place starting to sort 4/5 of the elements. Hereby we can keep the additional transport costs linear and only very few comparisons get lost, so that $n \log n - 0.8n$ comparisons are needed.

We show that we can improve the number of comparisons if we sort blocks of constant length with Merge-Insertion, before starting the algorithm. Another improvement is to start the iteration with a better version, which needs only $(1+\varepsilon)n$ space and again additional $O(n)$ transports. The result is, that we can improve this theoretically up to $n \log n - 1.3289n$ comparisons in the worst case. This is close to the theoretical lower bound of $n \log n - 1.443n$.

The total number of transports in all these versions can be reduced to $\varepsilon\, n \log n + O(1)$ for any $\varepsilon > 0$.

1 Introduction

In regard to well known sorting algorithms, there appears to be a kind of trade-off between space and time complexity: Methods for sorting like Merge-sort, Merge-Insertion or Insertion-sort, which have a small number of comparisons, either need $O(n^2)$ transports or work with a data structure, which needs $2n$ places (see [Kn72] [Me84]). Although the price for storage is decreasing, it is a desirable property for an algorithm to be *in-place*, that means to use only the storage needed for the input (except for a constant amount).

In [HL88] Huang and Langston gave an upper bound of $3n \log n$ for the number of comparisons of their *in-place* variant of merge-sort. Heap-sort needs $2n \log n$ comparisons and the upper bound for the comparisons in Bottom-up-Heapsort of $1.5n \log n$ [We90] is tight [Fl91]. Carlsson's variant of heap-sort [Ca87] needs $n \log n + \Theta(n \log \log n)$ comparisons. The first algorithm, which is nearly in-place and has $n \log n + O(n)$ as the number of comparisons, is Mc Diarmid and Reed's variant of Bottom-up-Heap-sort. Wegener showed

*this research has been partially supported by the EBRA working group No. 3166 ASMICS.

in [We91], that it needs $n \log n + 1.1n$ comparisons, but the algorithm is not in-place, since n additional bits are used.

For in-place sorting algorithms, we can find a trade-off between the number of comparisons and the number of transports: In [MR91] Munro and Raman describe an algorithm, which sorts in-place with only $O(n)$ transports but needs $O(n^{1+\varepsilon})$ comparisons.

The way the time complexity is composed of transports and essential[1] comparisons depends on the data structure of the elements. If an element is a block of fixed size and the key field is a small part of it, then a transport can be more expensive than a comparison. On the other hand, if an element is a small set of pointers to an (eventually external) database, then a transport is much cheaper than a comparison. But in any way an $O(n \log n)$ time algorithm is asymptotically faster than the algorithm in [MR91].

In this paper we describe the first sorting algorithm, which fulfills the following properties:

- It is **general**. (Any kind of elements of an ordered set can be sorted.)
- It is **in-place**. (The used storage except the input is constant.)
- It has the **worst-case** time complexity $O(n \log n)$.
- It has $n \log n + O(n)$ as the number of (essential) comparisons in the worst case (the constant factor 1 for the term $n \log n$).

The negative linear constant in the number of comparisons and the possibility of reducing the number of transports to $\varepsilon n \log n + O(1)$ for every fixed $\varepsilon > 0$ are further advantages of our algorithm.

In our description some parameters of the algorithm are left open. One of them depends on the given ε. Other parameters influence the linear constant in the amount of comparisons. Regarding these parameters, we can again find a trade-off between the linear component in the number of comparisons and a linear amount of transports. Although a linear amount of transports does not influence the asymptotic transport cost, it is surely important for a practical application. We will see that by choosing appropriate parameters, we obtain a negative linear component in the number of comparisons as close as we want to 1.328966, but for a practical application, we should be satisfied with at most 1. The main issue of the paper is theoretical, but we expect that a good choice of the parameters leads to an efficient algorithm for practical application.

2 A Variant of Merge-sort in $1.25n$ Places

The usual way of merging is to move the elements from one array of length n to another one. We use only one array, but we add a gap of length $\frac{n}{4}$ and move the elements from one side of the gap to the other. In each step pairs of lists are merged together by repeatedly moving the bigger head element to the tail of the new list. Hereby the space of former pairs of lists in the same stage can be used for the resulting lists. As long as the lists are short, there is enough space to do this (see Figure 1). The horizontal direction in a figure shows the indices of the array and the vertical direction expresses the size of the contained elements.

[1] Comparisons of pointers are not counted

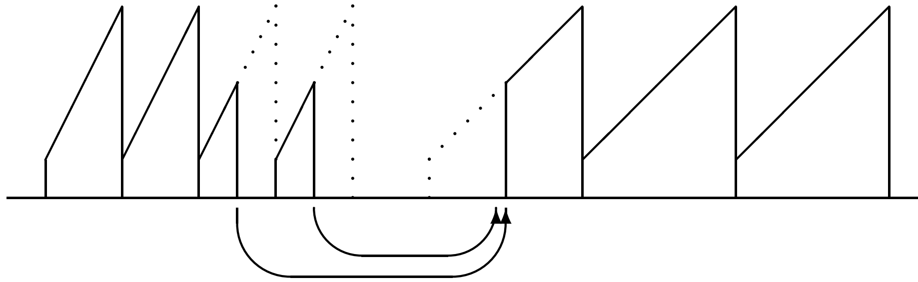


Figure 1: Merging two short lists on the left side to one list on the right side

In the last step it can happen, that in the worst case (for the place) there are two lists of length $\frac{n}{2}$. At some time during the last step the tail of the new list will hit the head of the second list. In this case half of the first list (of length $\frac{n}{4}$) is already merged and the other half is not yet moved. This means, that the gap begins at the point, where the left end of the resulting list will be. We can then start to merge from the other side (see Figure 2). This run ends exactly when all elements of the first list are moved into the new list.

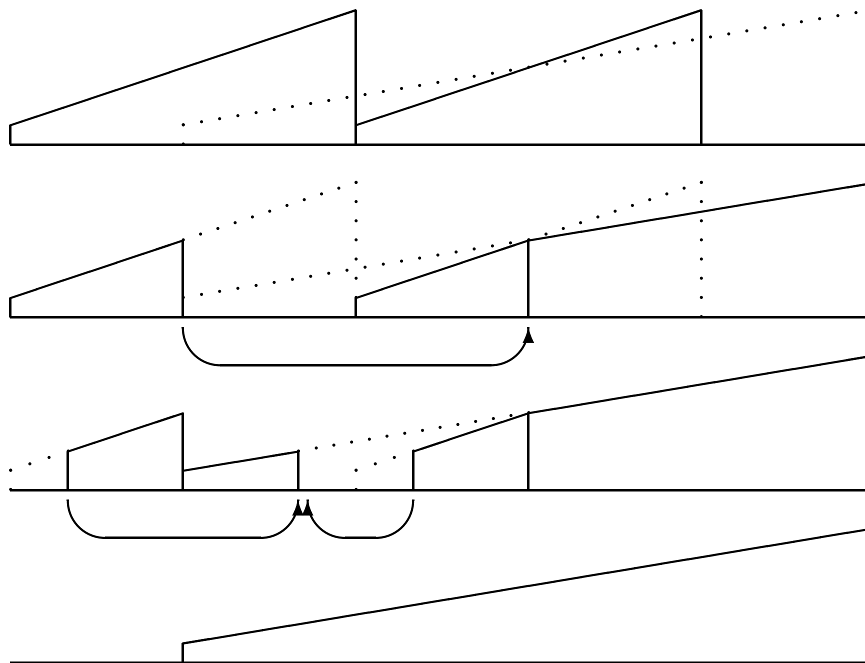


Figure 2: The last step, merging two lists of length $\frac{n}{2}$ using a gap of length $\frac{n}{4}$

Remark: In general, it suffices to have a gap of half of the size of the shorter list (If the number of elements is not a power of 2, it may happen that the other list has up to twice this length.) or $\frac{1}{2+r}$, if we don't mind the additional transports for shifting the second list r times. The longer list has to be between the shorter list and the gap.

Remark: This can easily be performed in a stable way by regarding the element in the left list as smaller, if two elements are equal.

3 In-Place sorting by Iteration

Using the procedure described in Section 2 we can also sort $0.8n$ elements of the input in-place by treating the $0.2n$ elements like the gap. Whenever an element is moved by the procedure, it is swapped with an unsorted element from the gap similar to the method in [HL88].

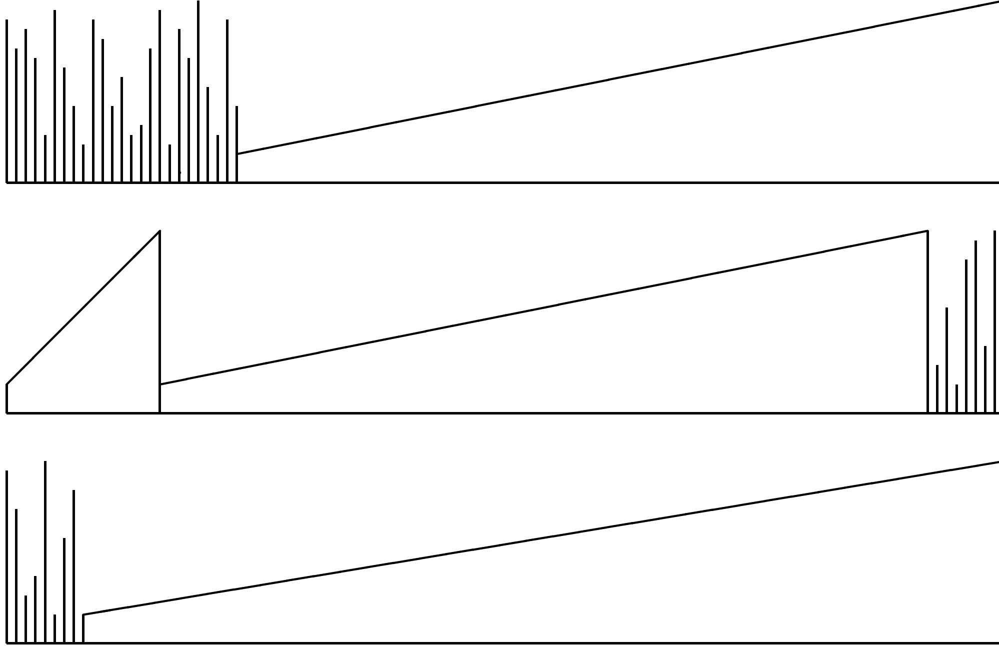


Figure 3: Reducing the number of unsorted element to $\frac{1}{3}$

In one iteration we sort $\frac{2}{3}$ of the unsorted elements with the procedure and merge them into the sorted list, where we again use $\frac{1}{3}$ of the unsorted part of the array, which still remains unsorted, as gap (see Figure 3). This means that we merge $\frac{2}{15}n$ elements to $\frac{4}{5}n$ elements, $\frac{2}{45}n$ to $\frac{14}{15}n$, $\frac{2}{135}n$ to $\frac{44}{45}n$ and so on.

Remark: In one iteration step we must either shift the long list once, which costs additional transports, or address the storage in a cyclic way in the entire algorithm.

Remark: This method mixes the unsorted elements completely, which makes it impossible to perform the algorithm in a stable way.

3.1 Asymmetric Merging

If we want to merge a long of length l and a short list of length h together, we can compare the first element of the short list with the 2^k'th element of the long list and then we can either move 2^k elements of the long list or we need another k comparisons to insert and move the element of the short list. Hence, in order to merge the two lists one needs in the worst case $\lfloor \frac{l}{2^k} \rfloor + (k+1)h - 1$ comparisons. This was first described by [HL72].

3.2 The Number of Additional Comparisons

Let us assume, that we can sort n elements with $n\log n - dn + O(1)$ comparisons, then we get the following: Sorting $\frac{4}{5}n$ elements needs $\frac{4}{5}n(\log\frac{4}{5}n - d) + O(1)$ comparisons. Sorting $\frac{2}{5\cdot 3^k}n$ elements for all k needs

$$
\begin{aligned}
& \sum_{k=1}^{\log n}\left(\frac{2}{5\cdot 3^k}n(\log(\frac{2}{5\cdot 3^k}n) - d) + O(1)\right) \\
= \; & \frac{2}{5}n\left(\sum_{k=1}^{\log n}\frac{1}{3^k}(\log\frac{2}{5}n - d) - \log 3\sum_{k=1}^{\log n}\frac{k}{3^k}\right) + O(\log n) \\
= \; & \frac{2}{5}n\left(\frac{1}{2}(\log\frac{2}{5}n - d) - \log 3\frac{3}{4}\right) + O(\log n) \\
= \; & \frac{1}{5}n\left(\log n - d + \log\frac{2}{5} - \log 3\frac{3}{2}\right) + O(\log n)
\end{aligned}
$$

comparisons. Together this yields $n(\log n - d + \frac{4}{5}\log\frac{4}{5} + \frac{1}{5}\log\frac{2}{5} - \frac{3}{10}\log 3) + O(\log n) = n(\log n - d - 0.9974) + O(\log n)$ comparisons. Merging $\frac{2}{15}n$ elements to $\frac{4}{5}n$ elements, $\frac{2}{5\cdot 3^{2m}}n$ to $(1 - \frac{1}{5\cdot 3^{2m-1}})n$ with $k = 3m+1$ and $\frac{2}{5\cdot 3^{2m+1}}n$ to $(1 - \frac{1}{5\cdot 3^{2m}})n$ with $k = 3m+3$ needs

$$
\begin{aligned}
& n\frac{1}{5} + n\frac{3\cdot 2}{15} + \\
& n\sum_{m=1}^{\log n}\left(\frac{1 - \frac{1}{5\cdot 3^{2m-1}}}{2^{3m+1}} + \frac{(3m+2)2}{5\cdot 3^{2m}} + \frac{1 - \frac{1}{5\cdot 3^{2m}}}{2^{3m+3}} + \frac{(3m+4)2}{5\cdot 3^{2m+1}}\right) + O(\log n) \\
= \; & n\left(\frac{3}{5} + \frac{5}{8}\sum_{m=1}^{\log n}\frac{1}{8^m} - \frac{13}{40}\sum_{m=1}^{\log n}\frac{1}{72^m} + \frac{8}{5}\sum_{m=1}^{\log n}\frac{m}{9^m} + \frac{4}{3}\sum_{m=1}^{\log n}\frac{1}{9^m}\right) + O(\log n) \\
= \; & n\left(\frac{3}{5} + \frac{5}{8}\cdot\frac{1}{7} - \frac{13}{40}\cdot\frac{1}{71} + \frac{8}{5}\cdot\frac{9}{64} + \frac{4}{3}\cdot\frac{1}{8}\right) + O(\log n) \\
= \; & 1.076n + O(\log n)
\end{aligned}
$$

comparisons. Thus we need only $n\log n - d_{ip}n + O(\log n) = n\log n - (d - 0.0785)n + O(\log n)$ comparisons for the in-place algorithm. For the algorithm described so far $d_{ip} = 0.8349$; the following section will explain this and show how this can be improved.

4 Methods for Reducing the Number of Comparisons

The idea is, that we can sort blocks of chosen length b with c comparisons by Merge-Insertion. If the time costs for one block are $O(b^2)$, then the time we need is $O(bn) = O(n)$ since b is fixed.

4.1 Good Applications of Merge-Insertion

Merge-Insertion needs $\sum_{k=1}^{b}\lceil\log\frac{3}{4}k\rceil$ comparisons to sort b elements [Kn72]. So it works well for $b_i := \frac{4^i - 1}{3}$, where it needs $c_i := \frac{2i4^i + i}{3} - 4^i + 1$ comparisons. We prove this in the following by induction: Clearly $b_1 = 1$ element is sorted with $c_1 = 0$ comparisons. To sort

$b_{i+1} = 4b_i + 1$ we need

$$\begin{aligned}
c_{i+1} &= c_i + b_i\lceil\log\frac{3}{2}b_i\rceil + (2b_i+1)\lceil\log\frac{3}{4}b_{i+1}\rceil \\
&= c_i + b_i\lceil\log\frac{4^i-1}{2}\rceil + (2b_i+1)\lceil\log\frac{4^{i+1}-1}{4}\rceil \\
&= \frac{2i4^i+i}{3} - 4^i + 1 + \frac{4^i-1}{3}(2i-1) + \frac{2\cdot 4^i+1}{3}2i \\
&= \frac{8i4^i+i-4\cdot 4^i+4}{3} \\
&= \frac{2(i+1)4^{i+1}+i+1}{3} - 4^{i+1} + 1.
\end{aligned}$$

Table 1 shows some instances for b_i and c_i.

i	b_i	c_i	$d_{best,i}$	d_i	$d_{ip,i}$
1	1	0	1	0.9139	0.8349
2	5	7	1.1219	1.0358	0.9768
3	21	66	1.2971	1.2110	1.1320
4	85	429	1.3741	1.2880	1.2090
5	341	2392	1.4019	1.3158	1.2368
6	1365	12290	1.4118	1.3257	1.2467
⋮	⋮	⋮	⋮	⋮	⋮
∞	∞	∞	1.415037	1.328966	1.250008

Table 1: The negative linear constant depending on the block size.

4.2 The Complete Number of Comparisons

In order to merge two lists of length l and h one needs $l+h-1$ comparisons. In the best case where $n = b2^k$ we would need 2^kc comparisons to sort the blocks first and then the comparisons to merge 2^{k-i} pairs of lists of the length $2^{i-1}b$ in the i'th of the k steps. These are

$$\begin{aligned}
2^kc + \sum_{i=1}^{k} 2^{k-i}(2^{i-1}b + 2^{i-1}b - 1) &= 2^kc + kn - 2^k + 1 \\
&= n\log n - n\underbrace{(\log b + \frac{1-c}{b})}_{d_{best}:=} + 1 \\
&= n\log n - nd_{best} + 1
\end{aligned}$$

comparisons. In the general case the negative linear factor d is not so good as d_{best}. Let the number of elements be $n = (2^k+m)b - j$ for $j < b$ and $m < 2^k$, then we need $(2^k+m)c$ comparisons to sort the blocks, $m(2b-1)-j$ comparisons to merge m pairs of blocks together and $\sum_{i=1}^{k}(n-2^{k-i}) = kn - 2^k + 1$ comparisons to merge everything together in k

steps. The total number of comparisons is

$$\begin{aligned} c_b(n) &:= (2^k + m)c + m(2b-1) - j + kn - 2^k + 1 \\ &= n\log n - n\log\frac{n}{2^k} + (2^k+m)(c+2b-1) - 2b2^k - j + 1 \\ &= n\log n - n\left(\log\frac{n}{2^k} + 2b\frac{2^k}{n} - \frac{c+2b-1}{b}\right) + \frac{c+2b-1}{b}j - j + 1 \\ &\le n\log n - n\underbrace{\left(\log(2b\ln 2) + \frac{1}{\ln 2} - \frac{c+2b-1}{b}\right)}_{d:=} + \frac{c+b-1}{b}j + 1 \\ &= n\log n - nd + \frac{c+b-1}{b}j + 1. \end{aligned}$$

The inequality follows from the fact that the expression $\log x + \frac{2b}{x}$ has its minimum for $x = 2b\ln 2$, since $(\log x + \frac{2b}{x})' = \frac{1}{x\ln 2} - \frac{2b}{x^2} = 0$. We have used $x := \frac{n}{2^k}$. Hence we loose at most $(\log(2\ln 2) - 2 + \frac{1}{\ln 2})n = -0.086071n = (d - d_{best})n$ comparisons in contrast to the case of an ideal value of n.

Table 1 shows the influence of the block size b_i and the number of comparisons c_i to sort it by Merge-Insertion on the negative linear constant d_i for the comparisons of the algorithm in Section 2 and $d_{ip,i}$ for the algorithm in Section 3. It shows that d_{ip} can be improved as close as we want to 1.250008. As one can see looking at Table 1, most of the possible improvement is already reached with relatively small blocks.

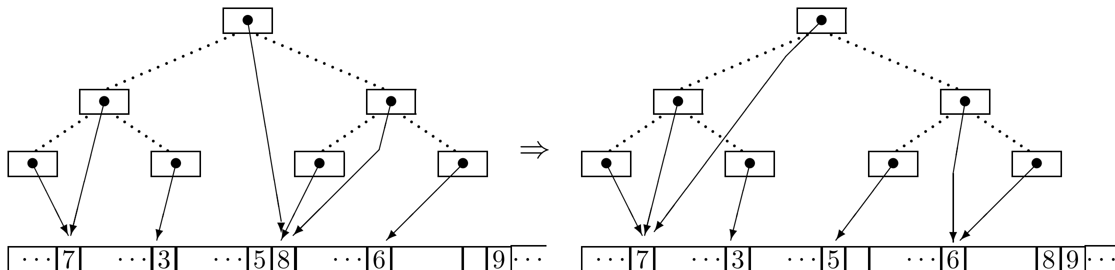
Another idea is that we can reduce the additional comparisons in Section 3 to a very small amount, if we start using the algorithm in Section 4.3. This allows us to improve the linear constant for the in-place algorithm as close as we want to d_i and in combination with the block size we can improve it as close as we want to 1.328966.

4.3 A variant of Merge-sort in $(1+\varepsilon)n$ places

We can change the algorithm of Section 2 in the following way: For a given $\varepsilon > 0$ we have to choose appropriate r's for the last $\lceil \log \frac{1}{\varepsilon} \rceil - 2$ steps according to the first remark in that section. Because the number of the last steps and the r's are constants determined by ε, the additional transports are $O(n)$ (of course the constant becomes large for small ε's).

5 The Reduction of the Number of Transports

The algorithms in Section 2 and Section 4.3 perform $n \log n + O(n)$ transports. We can improve this to $\varepsilon\, n \log n + O(1)$ for all constants $\varepsilon > 0$, if we combine $\lceil \frac{1}{\varepsilon} + 1 \rceil$ steps to 1 by merging $2^{\lceil \frac{1}{\varepsilon}+1 \rceil}$ (constantly many) lists in each step, as long as the lists are short enough. Hereby we keep the number of comparisons exactly the same using for example the following technique: We use a binary tree of that (constant) size, which contains on every leaf node the pointer to the head of a list (*'nil'* if it is empty, and additionally a pointer to the tail) and on every other node that pointer of a son node, which points to the bigger element. After moving one element we need $\lceil \frac{1}{\varepsilon} + 1 \rceil$ comparisons to repair the tree as shown in this example:



Note that if we want to reduce the number of transports to $o(n \log n)$, then the size of the tree must increase depending on n, but then the algorithm would not be in-place.

Remark: This method can also be used to reduce the total number of I/Os (for transports and comparisons) to a slow storage to $\varepsilon\, n \log n + O(1)$, if we can keep the $2^{\lceil \frac{1}{\varepsilon}+1 \rceil}$ elements in a faster storage.

5.1 Transport Costs in the In-Place Procedure

Choosing ε of the half size eliminates the factor 2 for swapping instead of moving. But there are still the additional transports in the iteration during the asymmetric merging in Section 3:

Since $\approx \log_3 n$ iterations have to be performed, we would get $O(n \log n)$ additional transports, if we perform only iterations of the described kind, which need $O(n)$ transports each time. But we can reduce these additional transports to $O(n)$, if we perform a second kind of iteration after each i iterations for any chosen i. Each time it reduces the number of elements to the half, which have to be moved in later iterations. This second kind of iteration works as follows (see Figure 4):

One Quick-Sort iteration step is performed on the unsorted elements, where the middle element of the sorted list is chosen as reference. Then one of the lists is sorted in the usual way (the other unsorted list is used as a gap) and merged with the corresponding half of the sorted list. These elements will never have to be moved again.

Remark: It is easy to see that we can reduce the number of additional comparisons to εn, if we choose i big enough. Although a formal estimation is quite hard, we conjecture, that we need only a few comparisons more, if we mainly apply iterations of the second kind, even in the worst case, where all unsorted elements are on one side. Clearly this iteration is good in the average.

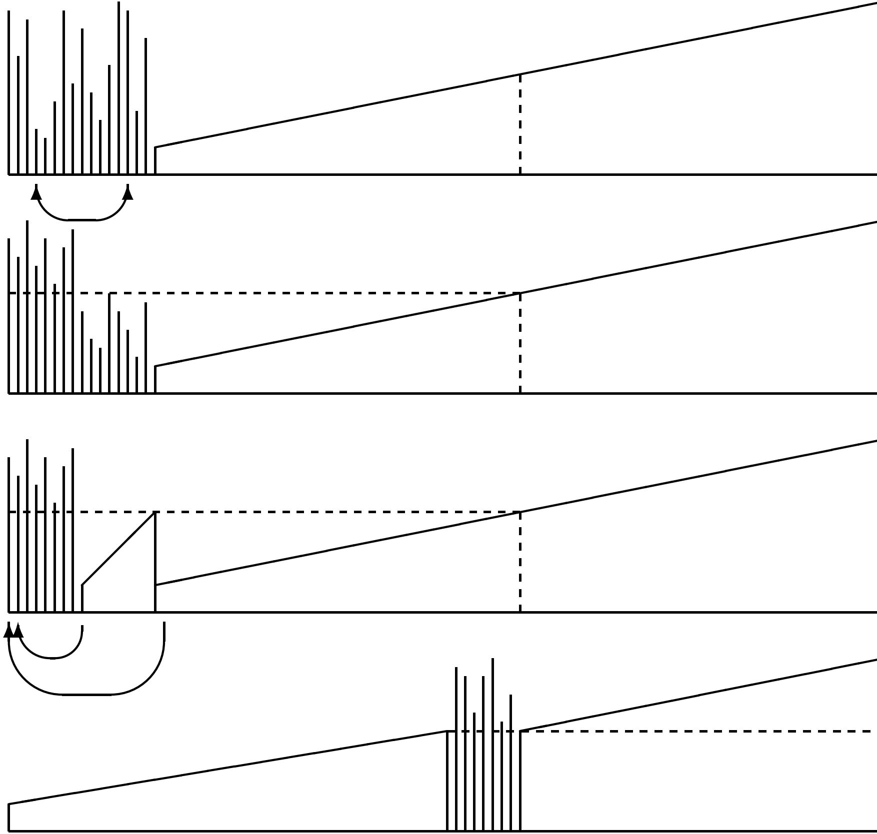


Figure 4: Moving half of the elements to their final position.

6 Possible Further Improvements

The algorithm in Section 2 works optimal for the case that $n = b2^m$ and has its worst behavior (loosing -0.086n) in the case that $n = 2 \ln 2 b 2^m = 1.386 b 2^m$. But we could avoid this for the start of the iteration, if we are not fixed to sort 0.8 of the elements in the first step but exactly $b2^m$ elements for $\frac{2}{5} < b2^m \leq \frac{4}{5}$. A simpler method would be to sort always the biggest possible $b2^m$ in each iteration step. In both cases the k's for the asymmetric merging in the further iterations have to be found dynamically, which makes a proper formal estimation difficult.

For practical applications it is worth to note, that the average number of comparisons for asymmetric merging is better than in the worst case, because for each element of the smaller list, which is inserted in $2^k - 1$ elements, on the average some elements of the longer list also can be moved to the new list, so that the average number of comparisons is less than $\lfloor \frac{l}{2^k} \rfloor + (k+1)h - 1$. So we conjecture, that an optimized in-place algorithm can have $n \log n - 1.4n + O(\log n)$ comparisons in the average and $n \log n - 1.3n + O(\log n)$ comparisons in the worst case avoiding large constants for the linear amount of transports.

If there exists a constant $c > 0$ and an in-place algorithm, which sorts cn of the input with $O(n \log n)$ comparisons and $O(n)$ transports, then the iteration method could be used to solve the open problem in [MR91].

6.0.1 Open Problems:

- Does there exists an in-place sorting algorithm with $O(n \log n)$ comparisons and $O(n)$ transports [MR91]?
- Does there exists an in-place sorting algorithm, which needs only $n \log n + O(n)$ comparisons and only $o(n \log n)$ transports?
- Does there exists a *stable* in-place sorting algorithm, which needs only $n \log n + O(n)$ comparisons and only $O(n \log n)$ transports?

References

[Ca87] S. Carlsson: A variant of HEAPSORT with almost optimal number of comparisons. Information Processing Letters 24:247-250,1987.

[Fl91] R. Fleischer: A tight lower bound for the worst case of bottom-up-heapsort. Technical Report MPI-I-91-104, Max-Planck-Institut für Informatik, D-W-6600 Saarbrücken, Germany April 1991.

[HL88] B-C. Huang and M.A. Langston: Practical in-place merging. CACM, 31:248-352,1988.

[HL72] F. K. Hwang and S. Lin: A simple algorithm for merging two disjoint linearly ordered sets. SIAM J. Computing 1:31-39, 1972.

[Kn72] D. E. Knuth: The Art of Computer Programming Volume 3 / Sorting and Searching. Addison-Wesley 1972.

[Me84] K. Mehlhorn: Data Structures and Algorithms, Vol 1: Sorting and Searching. Springer-Verlag, Berlin/Heidelberg, 1984.

[MR91] J. I. Munro and V. Raman: Fast Stable In-Place Sorting with $O(N)$ Data Moves. Proceedings of the FST&TCS, LNCS 560:266-277, 1991.

[We90] I. Wegener: BOTTOM-UP-HEAPSORT, a new variant of HEAPSORT beating on average QUICKSORT (if n is not very small). Proceedings of the MFCS90, LNCS 452, 516-522, 1990.

[We91] I. Wegener: The worst case complexity of Mc Diarmid and Reed's variant of BOTTOM-UP-HEAP SORT is less than $n \log n + 1.1n$. Proceedings of the STACS91, LNCS 480:137-147, 1991.