QuickXsort: Efficient Sorting with $n \log n - 1.399n + o(n)$ Comparisons on Average

Stefan Edelkamp[1] and Armin Weiß[2]

[1] TZI, Universität Bremen, Am Fallturm 1, D-28239 Bremen, Germany
[2] FMI, Universität Stuttgart, Universitätsstr. 38, D-70569 Stuttgart, Germany

Abstract. In this paper we generalize the idea of QuickHeapsort leading to the notion of QuickXsort. Given some external sorting algorithm X, QuickXsort yields an internal sorting algorithm if X satisfies certain natural conditions. We show that up to $o(n)$ terms the average number of comparisons incurred by QuickXsort is equal to the average number of comparisons of X.

We also describe a new variant of WeakHeapsort. With QuickWeakHeapsort and QuickMergesort we present two examples for the QuickXsort construction. Both are efficient algorithms that perform approximately $n \log n - 1.26n + o(n)$ comparisons on average. Moreover, we show that this bound also holds for a slight modification which guarantees an $n \log n + \mathcal{O}(n)$ bound for the worst case number of comparisons.

Finally, we describe an implementation of MergeInsertion and analyze its average case behavior. Taking MergeInsertion as a base case for QuickMergesort, we establish an efficient internal sorting algorithm calling for at most $n \log n - 1.3999n + o(n)$ comparisons on average. QuickMergesort with constant size base cases shows the best performance on practical inputs and is competitive to STL-Introsort.

Keywords: in-place sorting, quicksort, mergesort, analysis of algorithms.

1 Introduction

Sorting a sequence of n elements remains one of the most frequent tasks carried out by computers. A lower bound for sorting by only pairwise comparisons is $\log(n!) \approx n \log n - 1.44n + \mathcal{O}(\log n)$ comparisons for the worst and average case (logarithms denoted by log are always base 2, the average case refers to a uniform distribution of all input permutations assuming all elements are different). Sorting algorithms that are optimal in the leading term are called *constant-factor-optimal*. Tab. 1 lists some milestones in the race for reducing the coefficient in the linear term. One of the most efficient (in terms of number of comparisons) constant-factor-optimal algorithms for solving the sorting problem is Ford and Johnson's MergeInsertion algorithm [9]. It requires $n \log n - 1.329n + \mathcal{O}(\log n)$ comparisons in the worst case [12]. MergeInsertion has a severe drawback that makes it uninteresting for practical issues: similar to Insertionsort the number of element moves is quadratic in n, i. e., it has quadratic running time.

E.A. Hirsch et al. (Eds.): CSR 2014, LNCS 8476, pp. 139–152, 2014.

With InsertionSort we mean the algorithm that inserts all elements successively into the already ordered sequence finding the position for each element by binary search (*not* by linear search as frequently done). However, MergeInsertion and InsertionSort can be used to sort small subarrays such that the quadratic running time for these subarrays is small in comparison to the overall running time. Reinhardt [15] used this technique to design an internal MergeSort variant that needs in the worst case $n \log n - 1.329n + \mathcal{O}(\log n)$ comparisons. Unfortunately, implementations of this InPlaceMergeSort algorithm have not been documented. Katajainen et al.'s [11,8] work inspired by Reinhardt is practical, but the number of comparisons is larger.

Throughout the text we avoid the terms *in-place* or *in-situ* and prefer the term *internal* (opposed to *external*). We call an algorithm *internal* if it needs at most $\mathcal{O}(\log n)$ space (computer words) in addition to the array to be sorted. That means we consider QuickSort as an internal algorithm whereas standard MergeSort is external because it needs a linear amount of extra space.

Based on QuickHeapSort [2], we develop the concept of QuickXSort in this paper and apply it to MergeSort and WeakHeapSort, what yields efficient internal sorting algorithms. The idea is very simple: as in QuickSort the array is partitioned into the elements greater and less than some pivot element. Then one part of the array is sorted by some algorithm X and the other part is sorted recursively. The advantage of this procedure is that, if X is an external algorithm, then in QuickXSort the part of the array which is not currently being sorted may be used as temporary space, what yields an internal variant of X. We give an elementary proof that under natural assumptions QuickXSort performs up to $o(n)$ terms on average the same number of comparisons as X. Moreover, we introduce a trick similar to IntroSort [14] which guarantees $n \log n + \mathcal{O}(n)$ comparisons in the worst case.

The concept of QuickXSort (without calling it like that) was first applied in UltimateHeapSort by Katajainen [10]. In UltimateHeapSort, first the median of the array is determined, and then the array is partitioned into subarrays of equal size. Finding the median means significant additional effort. Cantone and Cincotti [2] weakened the requirement for the pivot and designed QuickHeapSort which uses only a sample of smaller size to select the pivot for partitioning. UltimateHeapSort is inferior to QuickHeapSort in terms of average case number of comparisons, although, unlike QuickHeapSort, it allows an $n \log n + \mathcal{O}(n)$ bound for the worst case number of comparisons. Diekert and Weiß [3] analyzed QuickHeapSort more thoroughly and described some improvements requiring less than $n \log n - 0.99n + o(n)$ comparisons on average.

Edelkamp and Stiegeler [5] applied the idea of QuickXSort to WeakHeapSort (which was first described by Dutton [4]) introducing QuickWeakHeapSort. The worst case number of comparisons of WeakHeapSort is $n\lceil \log n \rceil - 2^{\lceil \log n \rceil} + n - 1 \leq n \log n + 0.09n$, and, following Edelkamp and Wegener [6], this bound is tight. In [5] an improved variant with $n \log n - 0.91n$ comparisons in the worst case and requiring extra space is presented. With ExternalWeakHeapSort we propose a further refinement with the same worst case bound, but on

Table 1. Constant-factor-optimal sorting with $n \log n + \kappa n + o(n)$ comparisons

	Mem.	Other	κ Worst	κ Avg.	κ Exper.
Lower bound	$\mathcal{O}(1)$	$\mathcal{O}(n \log n)$	-1.44	-1.44	
BottomUpHeapsort [16]	$\mathcal{O}(1)$	$\mathcal{O}(n \log n)$	$\omega(1)$	–	[0.35,0.39]
WeakHeapsort [4,6]	$\mathcal{O}(n/w)$	$\mathcal{O}(n \log n)$	0.09	–	[-0.46,-0.42]
RelaxedWeakHeapsort [5]	$\mathcal{O}(n)$	$\mathcal{O}(n \log n)$	-0.91	-0.91	-0.91
Mergesort [12]	$\mathcal{O}(n)$	$\mathcal{O}(n \log n)$	-0.91	-1.26	–
ExternalWeakHeapsort #	$\mathcal{O}(n)$	$\mathcal{O}(n \log n)$	-0.91	-1.26*	–
Insertionsort [12]	$\mathcal{O}(1)$	$\mathcal{O}(n^2)$	-0.91	-1.38 #	–
MergeInsertion [12]	$\mathcal{O}(n)$	$\mathcal{O}(n^2)$	-1.32	-1.3999 #	[-1.43,-1.41]
InPlaceMergesort [15]	$\mathcal{O}(1)$	$\mathcal{O}(n \log n)$	-1.32	–	–
QuickHeapsort [2,3]	$\mathcal{O}(1)$	$\mathcal{O}(n \log n)$	$\omega(1)$	-0.03	$\approx$ 0.20
	$\mathcal{O}(n/w)$	$\mathcal{O}(n \log n)$	$\omega(1)$	-0.99	$\approx$ -1.24
QuickMergesort (IS) #	$\mathcal{O}(\log n)$	$\mathcal{O}(n \log n)$	-0.32	-1.38	–
QuickMergesort #	$\mathcal{O}(1)$	$\mathcal{O}(n \log n)$	-0.32	-1.26	[-1.29,-1.27]
QuickMergesort (MI) #	$\mathcal{O}(\log n)$	$\mathcal{O}(n \log n)$	-0.32	-1.3999	[-1.41,-1.40]

Abbreviations: # established in this paper, MI MergeInsertion, – not analyzed, * for $n = 2^k$, w: computer word width in bits; we assume $\log n \in \mathcal{O}(n/w)$.
For QuickXsort we assume InPlaceMergesort as a worst-case stopper (without $\kappa_{\text{worst}} \in \omega(1)$). The column "Mem." exhibits the amount of computer words of memory needed additionally to the data. "Other" gives the amount of other operations than comparisons performed during sorting.

average requiring approximately $n \log n - 1.26n$ comparisons. Using ExternalWeakHeapsort as X in QuickXsort we obtain an improvement over QuickWeakHeapsort of [5].

Mergesort is another good candidate for applying the QuickXsort construction. With QuickMergesort we describe an internal variant of Mergesort which not only in terms of number of comparisons competes with standard Mergesort, but also in terms of running time. As mentioned before, MergeInsertion can be used to sort small subarrays. We study MergeInsertion and provide an implementation based on weak heaps. Furthermore, we give an average case analysis. When sorting small subarrays with MergeInsertion, we can show that the average number of comparisons performed by Mergesort is bounded by $n \log n - 1.3999n + o(n)$, and, therefore, QuickMergesort uses at most $n \log n - 1.3999n + o(n)$ comparisons in the average case. To our best knowledge this is better than any previously known bound.

The paper is organized as follows: in Sect. 2 the concept of QuickXsort is described and our main theorems about the average and worst case number of comparisons are stated. The following sections are devoted to present examples for X in QuickXsort: In Sect. 3 we develop ExternalWeakHeapsort, analyze it, and show how it can be used for QuickWeakHeapsort. The next section treats QuickMergesort and the modification that small base cases are sorted with some other algorithm, e. g. MergeInsertion, which is then described in Sect. 5. Finally, we present our experimental results in Sect. 6.

Due to space limitations most proofs can be found in the arXiv version [7].

2 QuickXsort

In this section we give a more precise description of QuickXsort and derive some results concerning the number of comparisons performed in the average and worst case. Let X be some sorting algorithm. QuickXsort works as follows: First, choose some pivot element as median of some random sample. Next, partition the array according to this pivot element, i. e., rearrange the array such that all elements left of the pivot are less or equal and all elements on the right are greater or equal than the pivot element. (If the algorithms X outputs the sorted sequence in the extra memory, the partitioning is performed such that the all elements left of the pivot are greater or equal and all elements on the right are less or equal than the pivot element.) Then, choose one part of the array and sort it with algorithm X. (The preferred choice depends on the sorting algorithm X.) After one part of the array has been sorted with X, move the pivot element to its correct position (right after/before the already sorted part) and sort the other part of the array recursively with QuickXsort.

The main advantage of this procedure is that the part of the array that is not being sorted currently can be used as temporary memory for the algorithm X. This yields fast *internal* variants for various *external* sorting algorithms such as Mergesort. The idea is that whenever a data element should be moved to the external storage, instead it is swapped with the data element occupying the respective position in part of the array which is used as temporary memory. Of course, this works only if the algorithm needs additional storage only for data elements. Furthermore, the algorithm has to be able to keep track of the positions of elements which have been swapped. As the specific method depends on the algorithm X, we give some more details when we describe the examples for QuickXsort.

For the number of comparisons we can derive some general results which hold for a wide class of algorithms X. Under natural assumptions the average number of comparisons of X and of QuickXsort differ only by an $o(n)$-term. For the rest of the paper, we assume that the pivot is selected as the median of approximately $\sqrt{n}$ randomly chosen elements. Sample sizes of approximately $\sqrt{n}$ are likely to be optimal as the results in [3,13] suggest.

The following theorem is one of our main results. It can be proved using Chernoff bounds and then solving the linear recurrence.

Theorem 1 (QuickXsort Average-Case). *Let X be some sorting algorithm requiring at most $n \log n + cn + o(n)$ comparisons in the average case. Then,* QuickXsort *implemented with $\Theta(\sqrt{n})$ elements as sample for pivot selection is a sorting algorithm that also needs at most $n \log n + cn + o(n)$ comparisons in the average case.*

Does QuickXsort provide a good bound for the worst case? The obvious answer is "no". If always the $\sqrt{n}$ smallest elements are chosen for pivot selection, $\Theta(n^{3/2})$ comparisons are performed. However, we can prove that such a worst case is very unlikely. Let $R(n)$ be the worst case number of comparisons of the algorithm X.

Proposition 1. *Let $\epsilon > 0$. The probability that* QuickXsort *needs more than $R(n) + 6n$ comparisons is less than $(3/4 + \epsilon)^{\sqrt[4]{n}}$ for n large enough.*

In order to obtain a provable bound for the worst case complexity we apply a simple trick similar to the one used in Introsort [14]. We fix some worst case efficient sorting algorithm Y. This might be, e. g., InPlaceMergesort. (In order to obtain an efficient internal sorting algorithm, Y has to be internal.) Worst case efficient means that we have a $n \log n + \mathcal{O}(n)$ bound for the worst case number of comparisons. We choose some slowly decreasing function $\delta(n) \in o(1) \cap \Omega(n^{-\frac{1}{4}+\epsilon})$, e. g., $\delta(n) = 1/\log n$. Now, whenever the pivot is more than $n \cdot \delta(n)$ off the median, we stop with QuickXsort and continue by sorting both parts of the partitioned array with the algorithm Y. We call this QuickXYsort. To achieve a good worst case bound, of course, we also need a good bound for algorithm X. W. l. o. g. we assume the same worst case bounds for X as for Y. Note that QuickXYsort only makes sense if one needs a provably good worst case bound. Since QuickXsort is always expected to make at most as many comparisons as QuickXYsort (under the reasonable assumption that X on average is faster than Y – otherwise one would use simply Y), in every step of the recursion QuickXsort is the better choice for the average case.

Theorem 2 (QuickXYsort **Worst-Case).** *Let X be a sorting algorithm with at most $n \log n + cn + o(n)$ comparisons in the average case and $R(n) = n \log n + dn + o(n)$ comparisons in the worst case ($d \geq c$). Let Y be a sorting algorithm with at most $R(n)$ comparisons in the worst case. Then,* QuickXYsort *is a sorting algorithm that performs at most $n \log n + cn + o(n)$ comparisons in the average case and $n \log n + (d + 1)n + o(n)$ comparisons in the worst case.*

In order to keep the the implementation of QuickXYsort simple, we propose the following algorithm Y: Find the median with some linear time algorithm (see e.g. [1]), then apply QuickXYsort with this median as first pivot element. Note that this algorithm is well defined because by induction the algorithm Y is already defined for all smaller instances. The proof of Thm. 2 shows that Y, and thus QuickXYsort, has a worst case number of comparisons in $n \log n + \mathcal{O}(n)$.

3 QuickWeakHeapsort

In this section consider QuickWeakHeapsort as a first example of QuickXsort. We start by introducing weak heaps and then continue by describing WeakHeapsort and a novel external version of it. This external version is a good candidate for QuickXsort and yields an efficient sorting algorithm that uses approximately $n \log n - 1.2n$ comparisons (this value is only a rough estimate and neither a bound from below nor above). A drawback of WeakHeapsort and its variants is that they require one extra bit per element. The exposition also serves as an intermediate step towards our implementation of MergeInsertion, where the weak-heap data structure will be used as a building block. Conceptually, a *weak heap* (see Fig. 1) is a binary tree satisfying the following conditions:

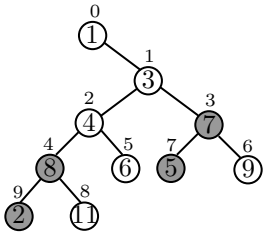

Fig. 1. A weak heap (reverse bits are set for grey nodes, above the nodes are array indices.)

(1) The root of the entire tree has no left child.
(2) Except for the root, the nodes that have at most one child are in the last two levels only. Leaves at the last level can be scattered, i. e., the last level is not necessarily filled from left to right.
(3) Each node stores an element that is smaller than or equal to every element stored in its right subtree.

From the first two properties we deduce that the height of a weak heap that has n elements is $\lceil \log n \rceil + 1$. The third property is called the *weak-heap ordering* or half-tree ordering. In particular, this property enforces no relation between an element in a node and those stored its left subtree. On the other hand, it implies that any node together with its right subtree forms a weak heap on its own. In an array-based implementation, besides the element array s, an array r of *reverse bits* is used, i. e., $r_i \in \{0, 1\}$ for $i \in \{0, \ldots, n-1\}$. The root has index 0. The array index of the left child of s_i is $2i + r_i$, the array index of the right child is $2i + 1 - r_i$, and the array index of the parent is $\lfloor i/2 \rfloor$ (assuming that $i \neq 0$). Using the fact that the indices of the left and right children of s_i are exchanged when flipping r_i, subtrees can be reversed in constant time by setting $r_i \leftarrow 1 - r_i$. The *distinguished ancestor* (*d-ancestor*(j)) of s_j for $j \neq 0$, is recursively defined as the parent of s_j if s_j is a right child, and the distinguished ancestor of the parent of s_j if s_j is a left child. The distinguished ancestor of s_j is the first element on the path from s_j to the root which is known to be smaller or equal than s_j by (3). Moreover, any subtree rooted by s_j, together with the distinguished ancestor s_i of s_j, forms again a weak heap with root s_i by considering s_j as right child of s_i.

The basic operation for creating a weak heap is the *join* operation which combines two weak heaps into one. Let $i < j$ be two nodes in a weak heap such that s_i is smaller than or equal to every element in the left subtree of s_j. Conceptually, s_j and its right subtree form a weak heap, while s_i and the left subtree of s_j form another weak heap. (Note that s_i is not part of the subtree with root s_j.) The result of *join* is a weak heap with root at position i. If $s_j < s_i$, the two elements are swapped and r_j is flipped. As a result, the new element s_j will be smaller than or equal to every element in its right subtree, and the new element s_i will be smaller than or equal to every element in the subtree rooted at

s_j. To sum up, *join* requires constant time and involves one element comparison and a possible element swap in order to combine two weak heaps to a new one.

The construction of a weak heap consisting of n elements requires $n-1$ comparisons. In the standard bottom-up construction of a weak heap the nodes are visited one by one. Starting with the last node in the array and moving to the front, the two weak heaps rooted at a node and its distinguished ancestor are joined. The amortized cost to get from a node to its distinguished ancestor is $\mathcal{O}(1)$ [6].

When using weak heaps for sorting, the minimum is removed and the weak heap condition restored until the weak heap becomes empty. After extracting an element from the root, first the *special path* from the root is traversed top-down, and then, in a bottom-up process the weak-heap property is restored using at most $\lceil \log n \rceil$ join operations. (The special path is established by going once to the right and then to the left as far as it is possible.) Hence, extracting the minimum requires at most $\lceil \log n \rceil$ comparisons.

Now, we introduce a modification to the standard procedure described by Dutton [4], which has a slightly improved performance, but requires extra space. We call this modified algorithm ExternalWeakHeapsort. This is because it needs an extra output array, where the elements which are extracted from the weak heap are moved to. On average ExternalWeakHeapsort requires less comparisons than RelaxedWeakHeapsort [5]. Integrated in QuickXsort we can implement it without extra space other than the extra bits r and some other extra bits. We introduce an additional array *active* and weaken the requirements of a weak heap: we also allow nodes on other than the last two levels to have less than two children. Nodes where the *active* bit is set to false are considered to have been removed. ExternalWeakHeapsort works as follows: First, a usual weak heap is constructed using $n-1$ comparisons. Then, until the weak heap becomes empty, the root – which is the minimal element – is moved to the output array and the resulting hole has to be filled with the minimum of the remaining elements (so far the only difference to normal WeakHeapsort is that there is a separate output area).

The hole is filled by searching the special path from the root to a node x which has no left child. Note that the nodes on the special path are exactly the nodes having the root as distinguished ancestor. Finding the special path does not need any comparisons since one only has to follow the reverse bits. Next, the element of the node x is moved to the root leaving a hole. If x has a right subtree (i. e., if x is the root of a weak heap with more than one element), this hole is filled by applying the hole-filling algorithm recursively to the weak heap with root x. Otherwise, the *active* bit of x is set to false. Now, the root of the whole weak heap together with the subtree rooted by x forms a weak heap. However, it remains to restore the weak heap condition for the whole weak heap. Except for the root and x, all nodes on the special path together with their right subtrees form weak heaps. Following the special path upwards these weak heaps are joined with their distinguished ancestor as during the weak heap construction (i. e., successively they are joined with the weak heap consisting of the root and the already treated

nodes on the special path together with their subtrees). Once, all the weak heaps on the special path are joined, the whole array forms a weak heap again.

Theorem 3. *For* $n = 2^k$ ExternalWeakHeapsort *performs exactly the same comparisons as* Mergesort *applied on a fixed permutation of the same input array.*

By [12, 5.2.4–13] we obtain the following corollary.

Corollary 1 (Average Case ExternalWeakHeapsort**).** *For* $n = 2^k$ *the algorithm* ExternalWeakHeapsort *uses approximately* $n \log n - 1.26n$ *comparisons in the average case.*

If n is not a power of two, the sizes of left and right parts of WeakHeapsort are less balanced than the left and right parts of ordinary Mergesort and one can expect a slightly higher number of comparisons. For QuickWeakHeapsort, the half of the array which is not sorted by ExternalWeakHeapsort is used as output area. Whenever the root is moved to the output area, the element that occupied that place before is inserted as a dummy element at the position where the *active* bit is set to false. Applying Thm. 1, we obtain the rough estimate of $n \log n - 1.2n$ comparisons for the average case of QuickWeakHeapsort.

4 QuickMergesort

As another example for QuickXsort we consider QuickMergesort. For the Mergesort part we use standard (top-down) Mergesort which can be implemented using m extra spaces to merge two arrays of length m. After the partitioning, one part of the array – we assume the first part – has to be sorted with Mergesort. In order to do so, the second half of this first part is sorted recursively with Mergesort while moving the elements to the back of the whole array. The elements from the back of the array are inserted as dummy elements into the first part. Then, the first half the first part is sorted recursively with Mergesort while being moved to the position of the former second part. Now, at the front of the array, there is enough space (filled with dummy elements) such that the two halves can be merged. The procedure is depicted in Fig. 2. As long as there is at least one third of the whole array as temporary memory left, the larger part of the partitioned array is sorted with Mergesort, otherwise the smaller part is sorted with Mergesort. Hence, the part which is not sorted by Mergesort always provides enough temporary space. Whenever a data element is moved to or from the temporary space, it is swapped with the dummy element occupying the respective position. Since Mergesort moves through the data from left to right, it is always clear which elements are the dummy elements. Depending on the implementation the extra space needed is $\mathcal{O}(\log n)$ words for the recursion stack of Mergesort. By avoiding recursion this can be reduced to $\mathcal{O}(1)$. Thm. 1 together with [12, 5.2.4–13] yields the next result.

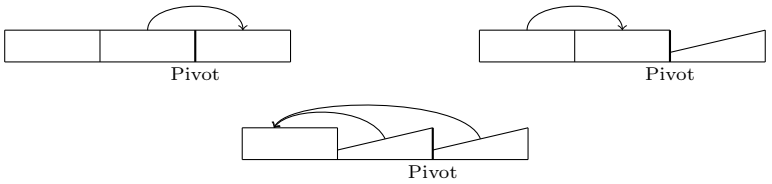

Fig. 2. First the two halves of the left part are sorted moving them from one place to another. Then, they are merged to the original place.

Theorem 4 (Average Case QuickMergesort**).** QuickMergesort *is an internal sorting algorithm that performs at most* $n \log n - 1.26n + o(n)$ *comparisons on average.*

We can do even better if we sort small subarrays with another algorithm Z requiring less comparisons but extra space and more moves, e. g., Insertionsort or MergeInsertion. If we use $\mathcal{O}(\log n)$ elements for the base case of Mergesort, we have to call Z at most $\mathcal{O}(n/\log n)$ times. In this case we can allow additional operations of Z like moves in the order of $\mathcal{O}(n^2)$ given that $\mathcal{O}((n/\log n) \cdot \log^2 n) = \mathcal{O}(n \log n)$. Note that for the next result we only need that the size of the base cases grows as n grows. Nevertheless, when applying an algorithm which uses $\Theta(n^2)$ moves, the size of the base cases has to be in $\mathcal{O}(\log n)$ in order to achieve an $\mathcal{O}(n \log n)$ overall running time.

Theorem 5 (QuickMergesort **with Base Case).** *Let Z be some sorting algorithm with* $n \log n + en + o(n)$ *comparisons on average and other operations taking at most* $\mathcal{O}(n^2)$ *time. If base cases of size* $\mathcal{O}(\log n)$ *are sorted with Z,* QuickMergesort *uses at most* $n \log n + en + o(n)$ *comparisons and* $\mathcal{O}(n \log n)$ *other instructions on average.*

Proof. By Thm. 1 and the preceding remark, the only thing we have to prove is that Mergesort with base case Z requires on average at most $\leq n \log n + en + o(n)$ comparisons, given that Z needs $\leq U(n) = n \log n + en + o(n)$ comparisons on average. The latter means that for every $\epsilon > 0$ we have $U(n) \leq n \log n + (e + \epsilon) \cdot n$ for n large enough.

Let $S_k(m)$ denote the average case number of comparisons of Mergesort with base cases of size k sorted with Z and let $\epsilon > 0$. Since $\log n$ grows as n grows, we have that $S_{\log n}(m) = U(m) \leq m \log m + (e + \epsilon) \cdot m$ for n large enough and $(\log n)/2 < m \leq \log n$. For $m > \log n$ we have $S_{\log n}(m) \leq 2 \cdot S_{\log n}(m/2) + m$ and by induction we see that $S_{\log n}(m) \leq m \log m + (e + \epsilon) \cdot m$. Hence, also $S_{\log n}(n) \leq n \log n + (e + \epsilon) \cdot n$ for n large enough. □

Recall that Insertionsort inserts the elements one by one into the already sorted sequence by binary search. Using Insertionsort we obtain the following result. Here, ln denotes the natural logarithm.

Proposition 2 (Average Case of Insertionsort**).** *The sorting algorithm* Insertionsort *needs* $n \log n - 2 \ln 2 \cdot n + c(n) \cdot n + \mathcal{O}(\log n)$ *comparisons on average where* $c(n) \in [-0.005, 0.005]$.

Corollary 2 (QuickMergesort with Base Case Insertionsort). *If we use as base case* Insertionsort*,* QuickMergesort *uses at most* $n \log n - 1.38n + o(n)$ *comparisons and* $\mathcal{O}(n \log n)$ *other instructions on average.*

Bases cases of growing size always lead to a constant factor overhead in running time if an algorithm with a quadratic number of total operations is applied. Therefore, in the experiments we also consider constant size base cases, which offer a slightly worse bound for the number of comparisons, but are faster in practice. We do not analyze them separately since the preferred choice for the size depends on the type of data to be sorted and the system on which the algorithms run.

5 MergeInsertion

MergeInsertion by Ford and Johnson [9] is one of the best sorting algorithms in terms of number of comparisons. Hence, it can be applied for sorting base cases of QuickMergesort what yields even better results than Insertionsort. Therefore, we want to give a brief description of the algorithm and our implementation. Algorithmically, MergeInsertion$(s_0, \ldots, s_{n-1})$ can be described as follows (an intuitive example for $n = 21$ can be found in [12]):

1. Arrange the input such that $s_i \geq s_{i+\lfloor n/2 \rfloor}$ for $0 \leq i < \lfloor n/2 \rfloor$ with one comparison per pair. Let $a_i = s_i$ and $b_i = s_{i+\lfloor n/2 \rfloor}$ for $0 \leq i < \lfloor n/2 \rfloor$, and $b_{\lfloor n/2 \rfloor} = s_{n-1}$ if n is odd.
2. Sort the values a_0,...,$a_{\lfloor n/2 \rfloor - 1}$ recursively with MergeInsertion.
3. Rename the solution as follows: $b_0 \leq a_0 \leq a_1 \leq \cdots \leq a_{\lfloor n/2 \rfloor - 1}$ and insert the elements $b_1, \ldots, b_{\lceil n/2 \rceil - 1}$ via binary insertion, following the ordering b_2, b_1, b_4, b_3, b_{10}, $b_9, \ldots, b_5, \ldots, b_{t_{k-1}}, b_{t_{k-1}-1}, \ldots b_{t_{k-2}+1}, b_{t_k}, \ldots$ into the main chain, where $t_k = (2^{k+1} + (-1)^k)/3$.

While the description is simple, MergeInsertion is not easy to implement efficiently because of the different renamings, the recursion, and the change of link structure. Our proposed implementation of MergeInsertion is based on a tournament tree representation with weak heaps as in Sect. 3. It uses $n \log n + n$ extra bits and works as follows: First, step 1 is performed for all recursion levels by constructing a weak heap. (Pseudo-code implementations for all the operations to construct a tournament tree with a weak heap and to access the partners in each round can be found in [7] – note that for simplicity in the above formulation the indices and the order are reversed compared to our implementation.) Then, in a second phase step 3 is executed for all recursion levels, see Fig. 3. One main subroutine of MergeInsertion is binary insertion. The call *binary-insert*(x, y, z) inserts the element at position z between position $x - 1$ and $x + y$ by binary insertion. In this routine we do not move the data elements themselves, but we use an additional index array $\phi_0, \ldots, \phi_{n-1}$ to point to the elements contained in the weak heap tournament tree and move these indirect

```
procedure: merge(m: integer)
global: φ array of n integers imposed by weak-heap
for l ← 0 to ⌊m/2⌋ − 1
    φ_{m−odd(m)−l−1} ← d-child(φ_l, m − odd(m));
k ← 1; e ← 2^k; c ← f ← 0;
while e < m
    k ← k + 1; e ← 2e;
    l ← ⌈m/2⌉ + f; f ← f + (t_k − t_{k−1});
    for i ← 0 to (t_k − t_{k−1}) − 1
        c ← c + 1;
        if c = ⌈m/2⌉ then
            return;
        if t_k > ⌈m/2⌉ − 1 then
            binary-insert(i + 1 − odd(m), l, m − 1);
        else
            binary-insert(⌊m/2⌋ − f + i, e − 1, ⌊m/2⌋ + f);
```

Fig. 3. Merging step in MergeInsertion with $t_k = (2^{k+1}+(-1)^k)/3$, $odd(m) = m \mod 2$, and $d\text{-}child(\phi_i, n)$ returns the highest index less than n of a grandchild of ϕ_i in the weak heap (i. e, $d\text{-}child(\phi_i, n)$ = index of the bottommost element in the weak heap which has $d\text{-}ancestor = \phi_i$ and index $< n$)

addresses. This approach has the advantage that the relations stored in the tournament tree are preserved. The most important procedure for MergeInsertion is the organization of the calls for *binary-insert*. After adapting the addresses for the elements b_i (w. r. t. the above description) in the second part of the array, the algorithm calls the binary insertion routine with appropriate indices. Note that we always use k comparisons for all elements of the k-th block (i. e., the elements $b_{t_k}, \ldots, b_{t_{k-1}+1}$) even if there might be the chance to save one comparison. By introducing an additional array, which for each b_i contains the current index of a_i, we can exploit the observation that not always k comparisons are needed to insert an element of the k-th block. In the following we call this the *improved* variant. The pseudo-code of the basic variant is shown in Fig. 3. The last sequence is not complete and is thus tackled in a special case.

Theorem 6 (Average Case of MergeInsertion). *The sorting algorithm* MergeInsertion *needs* $n \log n - c(n) \cdot n + \mathcal{O}(\log n)$ *comparisons on average, where* $c(n) \geq 1.3999$.

Corollary 3 (QuickMergesort with Base Case MergeInsertion). *When using* MergeInsertion *as base case,* QuickMergesort *needs at most* $n \log n - 1.3999n + o(n)$ *comparisons and* $\mathcal{O}(n \log n)$ *other instructions on average.*

6 Experiments

Our experiments consist of two parts. First, we compare the different algorithms we use as base cases, i. e., MergeInsertion, its improved variant, and Insertionsort. The results can be seen in Fig. 4. Depending on the size of the arrays

the displayed numbers are averages over 10-10000 runs[1]. The data elements we sorted were randomly chosen 32-bit integers. The number of comparisons was measured by increasing a counter in every comparison[2].

The outcome in Fig. 4 shows that our improved MergeInsertion implementation achieves results for the constant κ of the linear term in the range of $[-1.43, -1.41]$ (for some values of n are even smaller than -1.43). Moreover, the standard implementation with slightly more comparisons is faster than Insertionsort. By the $\mathcal{O}(n^2)$ work, the resulting runtimes for all three implementations raise quickly, so that only moderate values of n can be handled.

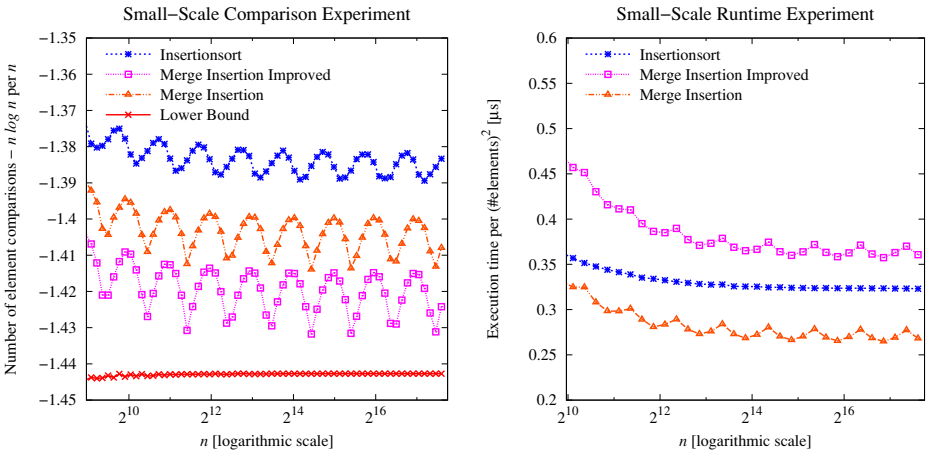

Fig. 4. Comparison of MergeInsertion, its improved variant and Insertionsort. For the number of comparisons $n \log n + \kappa n$ the value of κ is displayed.

The second part of our experiments (shown in Fig. 5) consists of the comparison of QuickMergesort (with base cases of constant and growing size) and QuickWeakHeapsort with state-of-the-art algorithms as STL-Introsort (i. e., Quicksort), STL-stable-sort (BottomUpMergesort) and Quicksort with median of $\sqrt{n}$ elements for pivot selection. For QuickMergesort with base cases, the improved variant of MergeInsertion is used to sort subarrays of size up to $40 \log_{10} n$. For the normal QuickMergesort we used base cases of size ≤ 9. We also implemented QuickMergesort with median of three for pivot selection, which turns out to be practically efficient, although it needs slightly more comparisons than QuickMergesort with median of $\sqrt{n}$. However,

[1] Our experiments were run on one core of an Intel Core i7-3770 CPU (3.40GHz, 8MB Cache) with 32GB RAM; Operating system: Ubuntu Linux 64bit; Compiler: GNU's `g++` (version 4.6.3) optimized with flag `-O3`.

[2] To rely on objects being handled we avoided the flattening of the array structure by the compiler. Hence, for the running time experiments, and in each comparison taken, we left the counter increase operation intact.

since also the larger half of the partitioned array can be sorted with MERGESORT, the difference to the median of $\sqrt{n}$ version is not as big as in QUICKHEAPSORT [3]. As suggested by the theory, we see that our improved QUICKMERGESORT implementation with growing size base cases MERGEINSERTION yields a result for the constant in the linear term that is in the range of $[-1.41, -1.40]$ – close to the lower bound. However, for the running time, normal QUICKMERGESORT as well as the STL-variants INTROSORT (`std::sort`) and BOTTOMUPMERGESORT (`std::stable_sort`) are slightly better. With about 15% the time gap, however, is not overly big, and may be bridged with additional optimizations. Also, when comparisons are more expensive, QUICKMERGESORT performs faster than INTROSORT and BOTTOMUPMERGESORT, see the arXiv version [7].

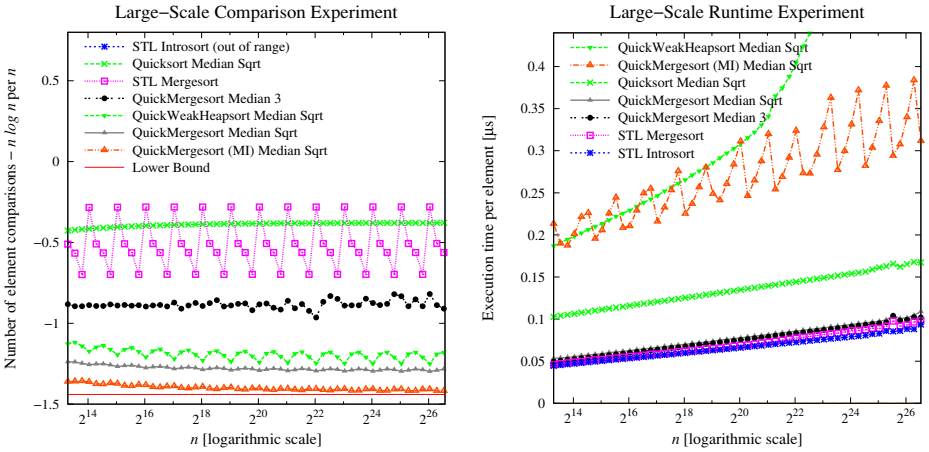

Fig. 5. Comparison of QUICKMERGESORT (with base cases of constant and growing size) and QUICKWEAKHEAPSORT with other sorting algorithms; (MI) is short for including growing size base cases derived from MERGEINSERTION. For the number of comparisons $n \log n + \kappa n$ the value of κ is displayed.

7 Concluding Remarks

Sorting n elements remains a fascinating topic for computer scientists both from a theoretical and from a practical point of view. With QUICKXSORT we have described a procedure how to convert an external sorting algorithm into an internal one introducing only $o(n)$ additional comparisons on average. We presented QUICKWEAKHEAPSORT and QUICKMERGESORT as two examples for this construction. QUICKMERGESORT is close to the lower bound for the average number of comparisons and at the same time is practically efficient, even when the comparisons are fast.

Using MERGEINSERTION to sort base cases of growing size for QUICKMERGESORT, we derive an an upper bound of $n \log n - 1.3999n + o(n)$ comparisons for

the average case. As far as we know a better result has not been published before. We emphasize that the average of our best implementation has a proven gap of at most $0.05n+o(n)$ comparisons to the lower bound. The value $n \log n - 1.4n$ for $n = 2^k$ matches one side of Reinhardt's conjecture that an optimized in-place algorithm can have $n \log n - 1.4n + \mathcal{O}(\log n)$ comparisons in the average [15]. Moreover, our experimental results validate the theoretical considerations and indicate that the factor -1.43 can be beaten. Of course, there is still room in closing the gap to the lower bound of $n \log n - 1.44n + \mathcal{O}(\log n)$ comparisons.

References

1. Blum, M., Floyd, R.W., Pratt, V., Rivest, R.L., Tarjan, R.E.: Time bounds for selection. J. Comput. Syst. Sci. 7(4), 448–461 (1973)
2. Cantone, D., Cinotti, G.: QuickHeapsort, an efficient mix of classical sorting algorithms. Theoretical Computer Science 285(1), 25–42 (2002)
3. Diekert, V., Weiß, A.: Quickheapsort: Modifications and improved analysis. In: Bulatov, A.A., Shur, A.M. (eds.) CSR 2013. LNCS, vol. 7913, pp. 24–35. Springer, Heidelberg (2013)
4. Dutton, R.D.: Weak-heap sort. BIT 33(3), 372–381 (1993)
5. Edelkamp, S., Stiegeler, P.: Implementing HEAPSORT with $n \log n - 0.9n$ and QUICKSORT with $n \log n + 0.2n$ comparisons. ACM Journal of Experimental Algorithmics 10(5) (2002)
6. Edelkamp, S., Wegener, I.: On the performance of $WEAK-HEAPSORT$. In: Reichel, H., Tison, S. (eds.) STACS 2000. LNCS, vol. 1770, pp. 254–266. Springer, Heidelberg (2000)
7. Edelkamp, S., Weiß, A.: QuickXsort: Efficient Sorting with $n \log n - 1.399n + o(n)$ Comparisons on Average. ArXiv e-prints, abs/1307.3033 (2013)
8. Elmasry, A., Katajainen, J., Stenmark, M.: Branch mispredictions don't affect mergesort. In: Klasing, R. (ed.) SEA 2012. LNCS, vol. 7276, pp. 160–171. Springer, Heidelberg (2012)
9. Ford, J., Lester, R., Johnson, S.M.: A tournament problem. The American Mathematical Monthly 66(5), 387–389 (1959)
10. Katajainen, J.: The Ultimate Heapsort. In: CATS, pp. 87–96 (1998)
11. Katajainen, J., Pasanen, T., Teuhola, J.: Practical in-place mergesort. Nord. J. Comput. 3(1), 27–40 (1996)
12. Knuth, D.E.: Sorting and Searching, 2nd edn. The Art of Computer Programming, vol. 3. Addison Wesley Longman (1998)
13. Martínez, C., Roura, S.: Optimal Sampling Strategies in Quicksort and Quickselect. SIAM J. Comput. 31(3), 683–705 (2001)
14. Musser, D.R.: Introspective sorting and selection algorithms. Software—Practice and Experience 27(8), 983–993 (1997)
15. Reinhardt, K.: Sorting *in-place* with a *worst case* complexity of $n \log n - 1.3n + o(\log n)$ comparisons and $\epsilon n \log n + o(1)$ transports. In: Ibaraki, T., Iwama, K., Yamashita, M., Inagaki, Y., Nishizeki, T. (eds.) ISAAC 1992. LNCS, vol. 650, pp. 489–498. Springer, Heidelberg (1992)
16. Wegener, I.: Bottom-up-Heapsort, a new variant of Heapsort beating, on an average, Quicksort (if n is not very small). Theoretical Computer Science 118(1), 81–98 (1993)