QuickXsort – A Fast Sorting Scheme in Theory and Practice*

Stefan Edelkamp Armin Weiß Sebastian Wild

October 17, 2019

Abstract. QuickXsort is a highly efficient in-place sequential sorting scheme that mixes Hoare's Quicksort algorithm with X, where X can be chosen from a wider range of other known sorting algorithms, like Heapsort, Insertionsort and Mergesort. Its major advantage is that QuickXsort can be in-place even if X is not. In this work we provide general transfer theorems expressing the number of comparisons of QuickXsort in terms of the number of comparisons of X. More specifically, if pivots are chosen as medians of (not too fast) growing size samples, the average number of comparisons of QuickXsort and X differ only by $o(n)$-terms. For median-of-k pivot selection for some constant k, the difference is a linear term whose coefficient we compute precisely. For instance, median-of-three QuickMergesort uses at most $n \lg n - 0.8358n + \mathcal{O}(\log n)$ comparisons.

Furthermore, we examine the possibility of sorting base cases with some other algorithm using even less comparisons. By doing so the average-case number of comparisons can be reduced down to $n \lg n - 1.4112n + o(n)$ for a remaining gap of only $0.0315n$ comparisons to the known lower bound (while using only $\mathcal{O}(\log n)$ additional space and $\mathcal{O}(n \log n)$ time overall).

Implementations of these sorting strategies show that the algorithms challenge well-established library implementations like Musser's Introsort.

Contents

*Parts of this article have been presented (in preliminary form) at the *International Computer Science Symposium in Russia (CSR) 2014* [11] and at the *International Conference on Probabilistic, Combinatorial and Asymptotic Methods for the Analysis of Algorithms (AofA) 2018* [52].

1. Introduction

Sorting a sequence of n elements remains one of the most frequent tasks carried out by computers. In the comparisons model, the well-known lower bound for sorting n distinct elements says that using fewer than $\lg(n!) = n\lg n - \lg e \cdot n \pm \mathcal{O}(\log n) \approx n\lg n - 1.442695n \pm \mathcal{O}(\log n)$[1] comparisons is not possible, both in the worst case and in the average case. The average case refers to a uniform distribution of all input permutations (random-permutation model).

In many practical applications of sorting, element comparisons have a similar running-time cost as other operations (e.g., element moves or control-flow logic). Then, a method has to balance costs to be overall efficient. This explains why QUICKSORT is generally considered the fastest general purpose sorting method, despite the fact that its number of comparisons is slightly higher than for other methods.

There are many other situations, however, where comparisons do have significant costs, in particular, when complex objects are sorted w.r.t. a order relation defined by a custom procedure. We are therefore interested in algorithms whose comparison count is *optimal up to lower order terms*, i.e., sorting methods that use $n\lg n + o(n\log n)$ or better $n\lg n + \mathcal{O}(n)$ comparisons; moreover, we are interested in bringing the coefficient of the linear term as close to the optimal -1.4427 as possible (since the linear term is not negligible for realistic input sizes). Our focus lies on *practical* methods whose running time is competitive to standard sorting methods even when comparisons are cheap. As a consequence, expected (rather than worst case) performance is our main concern.

[1] We write lg for $\log_2$ and log for the logarithm with an unspecified constant base in the $\mathcal{O}$ notation. A term $\pm\mathcal{O}(f(n))$ indicates an error term with unspecified sign; formal details are given in Section 3.

We propose QuickXsort as a general template for practical, comparison-efficient internal[2] sorting methods. QuickXsort uses the recursive scheme of ordinary Quicksort, but instead of doing two recursive calls after partitioning, first one of the segments is sorted by some other sorting method "X". Only the second segment is recursively sorted by QuickXsort. The key insight is that X can use the second segment as a temporary buffer area; so X can be an *external* method, but the resulting QuickXsort is still an *internal* method. QuickXsort only requires $\mathcal{O}(1)$ words of extra space, even when X itself requires a linear-size buffer.

We discuss a few concrete candidates for X to illustrate the versatility of QuickXsort. We provide a precise analysis of QuickXsort in the form of "transfer theorems": we express the costs of QuickXsort in terms of the costs of X, where generally the use of QuickXsort adds a certain overhead to the lower order terms of the comparison counts. Unlike previous analyses for special cases, our results give tight bounds.

A particularly promising (and arguably the most natural) candidate for X is Mergesort. Mergesort is both fast in practice and comparison-optimal up to lower order terms; but the linear-extra space requirement can make its usage impossible. With QuickMergesort we describe an internal sorting algorithm that is competitive in terms of comparisons count and running time.

Outline. The remainder of this section surveys previous work and summarizes the contributions of this article. We describe QuickXsort in Section 2 and fix our mathematical notation in in Section 3. Section 4 contains three "transfer theorems" that express the cost of QuickXsort in terms of the costs of X; they consider, respectively, the average case for growing size samples (Section 4.2), for constant size samples (Section 4.3), and the variance for constant size samples (Section 4.4). We only sketch the proofs in this section for readability; readers interested in the full proofs find them in Section 9, at the end of this paper. In Section 5, we apply our transfer theorems to QuickMergesort and QuickHeapsort and discuss the results. Then, in Section 6, we discuss how QuickMergesort's comparison count can be improved by sorting small base cases with Insertionsort or MergeInsertion, and we analyze the average costs of these algorithms. After that, in Section 7 we present our experimental results including an extensive running time study. We conclude with an outlook remark and open questions in Section 8. Section 9 contains full proofs for all our analytical results.

For the reader's convenience, Appendix A lists all used notations and Appendix B collects mathematical preliminaries used in our analysis. Finally, Appendix C describes the Mergesort variants used for QuickMergesort.

1.1. Related work

We pinpoint selected relevant works from the vast literature on sorting; our overview cannot be comprehensive, though.

[2] Throughout the text, we avoid the (in our context somewhat ambiguous) terms *in-place* or *in-situ*. We instead call an algorithm *internal* if it needs at most $\mathcal{O}(\log n)$ words of space (in addition to the array to be sorted). In particular, Quicksort is an internal algorithm whereas standard Mergesort is not (hence called *external*) since it uses a linear amount of buffer space for merges.

Comparison-efficient sorting. There is a wide range of sorting algorithms achieving the bound of $n \lg n + \mathcal{O}(n)$ comparisons. The most prominent is MERGESORT, which additionally comes with a small coefficient in the linear term. Unfortunately, MERGESORT requires linear extra space. Concerning the space ULTIMATEHEAPSORT [29] does better, however, with the cost of a quite large linear term. Other algorithms, provide even smaller linear terms than MERGESORT. Table 1 lists some milestones in the race for reducing the coefficient in the linear term. Despite the fundamental nature of the problem, little improvement has been made (w.r.t. the worst-case comparisons count) over Ford and Johnson's MERGEINSERTION algorithm [18] – which was published 1959! MERGEINSERTION requires $n \lg n - 1.329n + \mathcal{O}(\log n)$ comparisons in the worst case [32].

Table 1: Milestones of comparison-efficient sorting methods. The methods use (at most) $n \lg n + bn + o(n)$ comparisons for the given b in worst (b_{wc}) and/or average case (b_{ac}). Space is given in machine words (unless indicated otherwise).

Algorithm	b_{ac}	b_{ac} empirical	b_{wc}	Space	Time
Lower bound	-1.44		-1.44	$\mathcal{O}(1)$	$\mathcal{O}(n \log n)$
MERGESORT [32]	-1.24		-0.91	$\mathcal{O}(n)$	$\mathcal{O}(n \log n)$
INSERTIONSORT [32]	$-1.38^{\#}$		-0.91	$\mathcal{O}(1)$	$\mathcal{O}(n^2)^{\diamond}$
MERGEINSERTION [32]	$-1.3999^{\#\times}$	$[-1.43, -1.41]$	-1.32	$\mathcal{O}(n)$	$\mathcal{O}(n^2)^{\diamond}$
MI+IS [28]	$-1.4106^{\star}$			$\mathcal{O}(n)$	$\mathcal{O}(n^2)^{\diamond}$
BOTTOMUPHEAPSORT [50]	?	$[0.35, 0.39]$	$\omega(1)$	$\mathcal{O}(1)$	$\mathcal{O}(n \log n)$
WEAKHEAPSORT [7, 9]	?	$[-0.46, -0.42]$	0.09	$\mathcal{O}(n)$ bits	$\mathcal{O}(n \log n)$
RELAXEDWEAKHEAPSORT [8]	-0.91	-0.91	-0.91	$\mathcal{O}(n)$	$\mathcal{O}(n \log n)$
INPLACEMERGESORT [42]	?		-1.32	$\mathcal{O}(1)$	$\mathcal{O}(n \log n)$
QUICKHEAPSORT [3]	$-0.03^{\leq}$	≈ 0.20	$\omega(1)$	$\mathcal{O}(1)$	$\mathcal{O}(n \log n)$
Improved QUICKHEAPSORT [4]	$-0.99^{\leq}$	≈ -1.24	$\omega(1)$	$\mathcal{O}(n)$ bits	$\mathcal{O}(n \log n)$
ULTIMATEHEAPSORT [29]	$\mathcal{O}(1)$	≈ 6 [4]	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(n \log n)$
QUICKMERGESORT $^{\#}$	-1.24	$[-1.29, -1.27]$	$-0.32^{\dagger}$	$\mathcal{O}(1)$	$\mathcal{O}(n \log n)$
QUICKMERGESORT (IS)$^{\#\perp}$	-1.38		$-0.32^{\dagger}$	$\mathcal{O}(\log n)$	$\mathcal{O}(n \log n)$
QUICKMERGESORT (MI)$^{\#\perp}$	$-1.3999^{\times}$	$[-1.41, -1.40]$	$-0.32^{\dagger}$	$\mathcal{O}(\log n)$	$\mathcal{O}(n \log n)$
QUICKMERGESORT (MI+IS)$^{\#\perp}$	$-1.4106^{\star}$		$-0.32^{\dagger}$	$\mathcal{O}(\log n)$	$\mathcal{O}(n \log n)$

$^{\#}$ in this paper
* bound recently improved to -1.4112 by [49]
$^{\times}$ bound recently improved to -1.4005 by [49] (based on this work)
$^{\leq}$ only upper bound proven in cited source
† assuming INPLACEMERGESORT as a worst-case stopper; with median-of-medians fallback pivot selection: $\mathcal{O}(1)$, without worst-case stopper: $\omega(1)$
$^{\perp}$ using given method for small subproblems; MI = MERGEINSERTION, IS = INSERTIONSORT.
$^{\diamond}$ using a rope data structure and allowing additional $\mathcal{O}(n)$ space in $\mathcal{O}(n \log^2 n)$.

MERGEINSERTION has a severe drawback that renders the algorithm completely impractical, though: in a direct implementation the number of element moves is quadratic in n. Its running time can be improved to $\mathcal{O}(n \log^2 n)$ by using a rope data structure [2] (or a similar data structure which allows random access and insertions in $\mathcal{O}(\log n)$ time) for insertion of elements (which, of course, induces additional constant-factor overhead); see [49]. The same is true for INSERTIONSORT, which, unless explicitly indicated otherwise, refers to the algorithm that inserts elements successively into a sorted prefix by finding the insertion position by *binary search* – as opposed to linear/sequential search in STRAIGHTINSERTIONSORT.

Note that MergeInsertion or Insertionsort can still be used as comparison-efficient subroutines to sort base cases for Mergesort (and QuickMergesort) of size $\mathcal{O}(\log n)$ without affecting the overall running-time complexity of $\mathcal{O}(n \log n)$.

Reinhardt [42] used this trick (among others) to design an internal Mergesort variant that needs $n \lg n - 1.329n \pm \mathcal{O}(\log n)$ comparisons in the worst case. Unfortunately, implementations of this InPlaceMergesort algorithm have not been documented. The work of Katajainen et al. [30, 20, 16], inspired by Reinhardt, is practical, but the number of comparisons is larger.

Improvements over MergeInsertion have been obtained for the *average* number of comparisons. A combination of MergeInsertion with a variant of Insertionsort (inserting two elements simultaneously) by Iwama and Teruyama uses $\leq n \lg n - 1.41064n$ comparisons on average [28]; as for MergeInsertion, the overall complexity remains quadratic (resp. $\Theta(n \log^2 n)$), though. Since this article was submitted, a new improved upper bound for MergeInsertion of $n \lg n - 1.4005n + \mathcal{O}(\log n)$ comparisons on average has been proven by Stober and the second author in [49]. This new upper bound also allows us to strengthen Iwama and Teruyama's bound for their combined algorithm to $n \lg n - 1.4112n$.

Previous work on QuickXsort. Cantone and Cincotti [3] were the first to explicitly give a name to the mixture of Quicksort with another sorting method; they proposed QuickHeapsort. However, the concept of QuickXsort (without calling it like that) was first used in UltimateHeapsort by Katajainen [29]. Both versions use an external Heapsort variant in which a heap containing m elements is not stored compactly in the first m cells of the array, but may be spread out over the whole array. This allows to restore the heap property with $\lceil \lg n \rceil$ comparisons after extracting some element by introducing a new gap (we can think of it as an element of infinite weight) and letting it sink down to the bottom of the heap. The extracted elements are stored in an output buffer.

In UltimateHeapsort, we first find the exact median of the array (using a linear-time algorithm) and then partition the array into subarrays of equal size; this ensures that with the above external Heapsort variant, the first half of the array (on which the heap is built) does not contain gaps (Katajainen calls this a two-level heap); the other half of the array is used as the output buffer. QuickHeapsort avoids the significant additional effort for exact median computations by choosing the pivot as median of some smaller sample. In our terminology, it applies QuickXsort where X is ExternalHeapsort. UltimateHeapsort is inferior to QuickHeapsort in terms of the average case number of comparisons, although, unlike QuickHeapsort, it allows an $n \lg n + \mathcal{O}(n)$ bound for the worst case number of comparisons. Diekert and Weiß [4] analyzed QuickHeapsort more thoroughly and described some improvements requiring less than $n \lg n - 0.99n + o(n)$ comparisons on average (choosing the pivot as median of $\sqrt{n}$ elements). However, both the original analysis of Cantone and Cincotti and the improved analysis could not give tight bounds for the average case of median-of-k QuickHeapsort.

In [16] Elmasry, Katajainen and Stenmark proposed InSituMergesort, following the same principle as UltimateHeapsort, but with Mergesort replacing ExternalHeapsort. Also InSituMergesort only uses an expected-case linear-time algorithm for the median computation.[3]

[3] The first two authors elaborate on how to make this approach worst-case efficient with little additional overhead in a recent article [14].

In the conference paper [11], the first and second author introduced the name QuickXsort and first considered QuickMergesort as an application (including weaker forms of the results in Section 4.2 and Section 6 without proofs). In [52], the third author analyzed QuickMergesort with constant-size pivot sampling (see Section 4.3). A weaker upper bound for the median-of-3 case was also given by the first two authors in the preprint [13]. The present work is an extended version of [11] and [52]; it unifies and strengthens these results, includes detailed proofs, and it complements the theoretical findings with extensive running-time experiments.

1.2. Contributions

In this work, we describe QuickXsort as a general template for transforming an external algorithm into an internal algorithm. As examples we consider QuickHeapsort and QuickMergesort. For the reader's convenience, we briefly list our results here (with references to the corresponding sections).

1. If X is some sorting algorithm using $x(n) = n\lg n + bn \pm o(n)$ comparisons on expectation, then, median-of-$k(n)$ QuickXsort with $k(n) \in \omega(1) \cap o(n)$ needs $x(n) \pm o(n)$ comparisons in the average case (Theorem 4.1).

2. Under reasonable assumptions, sample sizes of $\Theta(\sqrt{n})$ are optimal among all polynomial size sample sizes.

3. The probability that median-of-$\sqrt{n}$ QuickXsort needs more than $x_{\text{wc}}(n)+6n$ comparisons decreases exponentially in $\sqrt[4]{n}$ (Proposition 4.5). Here, $x_{\text{wc}}(n)$ is the worst-case cost of X.

4. We introduce *median-of-medians fallback pivot selection* (a trick similar to Introsort [40]) which guarantees $n\lg n + \mathcal{O}(n)$ comparisons in the worst case while altering the average case only by $o(n)$-terms (Theorem 4.7).

5. Let k be a fixed constant and let X be a sorting method that needs a buffer of $\lfloor \alpha n \rfloor$ elements for some constant $\alpha \in [0,1]$ to sort n elements and requires on average $x(n) = n\lg n + bn \pm o(n)$ comparisons to do so. Then median-of-k QuickXsort needs

$$c(n) \;=\; n\lg n + (P(k,\alpha) + b)\cdot n \;\pm\; o(n),$$

   comparisons on average where $P(k,\alpha)$ is some constant depending on k and α (Theorem 4.8). We have $P(1,1) = 0.5070$ (for median-of-3 QuickHeapsort or QuickMergesort) and $P(1,1/2) = 0.4050$ (for median-of-3 QuickMergesort).

6. We approximate the standard deviation of the number of comparisons of median-of-k QuickMergesort for some small values of k. For $k = 3$ and $\alpha = \frac{1}{2}$, the standard deviation is close to $0.3268n$ (Section 4.4).

7. When sorting small subarrays of size $\mathcal{O}(\log n)$ in QuickMergesort with some sorting algorithm Z using $z(n) = n\lg n + (b \pm \varepsilon)n + o(n)$ comparisons on average and other operations taking at most $\mathcal{O}(n^2)$ time, then QuickMergesort needs $z(n) + o(n)$ comparisons on average (Corollary 6.2). In order to apply this result, we prove that

- (Binary) InsertionSort needs $n \lg n - (1.3863 \pm 0.005)n + o(n)$ comparisons on average (Proposition 6.3).
- (A simplified version of) MergeInsertion [19] needs at most $n \lg n - 1.3999n + o(n)$ on average (Theorem 6.5).

Moreover, with Iwama and Teruyama's algorithm [28] this can be sightly improved to $n \lg n - 1.4112n + o(n)$ comparisons (Corollary 6.10).

8. We run experiments confirming our theoretical estimates for the average number of comparisons of QuickMergeSort and our approximation for its standard deviation, verifying that the sublinear terms are indeed negligible (Section 7).
9. From running-time studies comparing QuickMergeSort with various other sorting methods, we conclude that our QuickMergeSort implementation is among the fastest internal general-purpose sorting methods for both the regime of cheap and expensive comparisons (Section 7).

To simplify the arguments, in all our analyses we assume that all elements in the input are distinct. This is no severe restriction since duplicate elements can be handled well using fat-pivot partitioning (which excludes elements equal to the pivot from recursive calls and calls to X).

2. QuickXsort

In this section, we precisely describe QuickXSort. Let X be a sorting method that requires buffer space for storing at most $\lfloor \alpha n \rfloor$ elements (for $\alpha \in [0, 1]$) to sort n elements. The buffer may only be accessed by swaps so that once X has finished its work, the buffer contains the same elements as before, albeit (in general) in a different order than before.

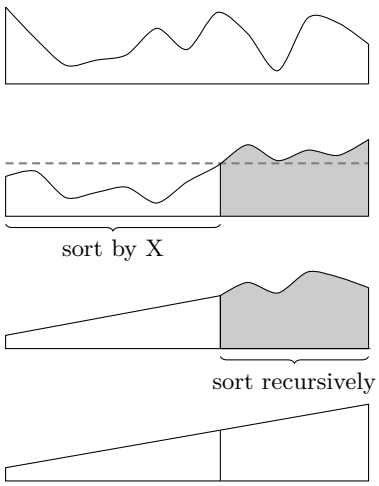


Figure 1: Schematic steps of QuickXSort. The pictures show a sequence, where the vertical height corresponds to key values. We start with an unsorted sequence (top), and partition it around a pivot value (second from top). Then one part is sorted by X (second from bottom) using the other segment as buffer area (grey shaded area). Note that this in general permutes the elements there. Sorting is completed by applying the same procedure recursively to the buffer (bottom).

QuickXSort now works as follows: First, we choose a pivot element; typically we use the median of a random sample of the input. Next, we partition the array according to this pivot element, i.e., we rearrange the array so 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. This results in two contiguous segments of J_1 resp. J_2 elements; we exclude the pivot here (since it will

have reached its final position), so $J_1 + J_2 = n - 1$. Note that the (one-based) rank R of the pivot is random, and so are the segment sizes J_1 and J_2; we have $R = J_1 + 1$.

We then sort one segment by X *using the other segment as a buffer.* To guarantee a sufficiently large buffer for X when it sorts J_r ($r = 1$ or 2), we must make sure that J_{3-r} (the other segment size) satisfies $J_{3-r} \geq \alpha J_r$. In case both segments could be sorted by X, we use the larger of the two. After one part of the array has been sorted with X, we move the pivot element to its correct position (right after/before the already sorted part) and recurse on the other segment of the array. The process is illustrated in Figure 1.

The main advantage of this procedure is that the part of the array that is not currently being sorted can be used as temporary buffer area for algorithm X. This yields fast *internal* variants for various *external* sorting algorithms such as MERGESORT. We have to make sure, however, that the contents of the buffer is not lost. A simple sufficient condition is to require that X maintains a permutation of the elements in the input and buffer: whenever a data element should be moved to the external storage, it is *swapped* with the data element occupying that respective position in the buffer area. For MERGESORT, using swaps in the merge (see Section 2.1) is sufficient. For other methods, we need further modifications.

Avoiding unnecessary copying. For some X, it is convenient to have the sorted sequence reside in the buffer area instead of the input area. We can avoid unnecessary swaps for such X by partitioning "in reverse order", i.e., so that large elements are left of the pivot and small elements right of the pivot.

Stable sorting. We point out that standard, efficient in-place partitioning is not a *stable* method, i.e., elements that compare equal might not appear in the same relative order after partitioning as in the original input. Moreover, elements in the part of the input used as buffer for X can be reordered arbitrarily while executing X. As a consequence, and much like standard QUICKSORT, QUICKXSORT is *not* a stable sorting algorithm, irrespective of whether X alone is stable.

Pivot sampling. It is a standard strategy for QUICKSORT to choose pivots as the median of some sample. This optimization is also effective for QUICKXSORT and we will study its effect in detail. We assume that in each recursive call, we choose a sample of k elements, where $k = 2t + 1$, $t \in \mathbb{N}_0$ is an odd number. The sample can either be selected deterministically (e.g., selecting fixed positions) or at random. Usually for the analysis we do not need random selection; only if the algorithm X does not preserve randomness of the buffer element, we have to assume randomness (see Section 9.2). However, notice that in any case random selection might be beneficial as it protects against a potential adversary who provides a worst-case input permutation.

Unlike for QUICKSORT, in QUICKXSORT pivot selection contributes only a minor term to the overall running time (at least in the usual case that $k \ll n$). The reason is that QUICKXSORT only makes an expected *logarithmic* number of partitioning rounds since in expectation, after each partitioning round, a constant fraction of the input is excluded from further consideration (after sorting it with X). This is in contrast to plain QUICKSORT, which always uses a linear number of partitioning rounds; it makes randomizing pivot selection and larger sample sizes practical in QUICKXSORT. For our analysis, we need not fix the precise

details of how pivots are selected; we only assume that selecting the median of k elements needs $s(k) = \Theta(k)$ comparisons on average (e.g., using QUICKSELECT [25]).

We consider both the case where k is a fixed constant and where $k = k(n)$ is an increasing function of the (sub)problem size. Previous analytical results for QUICKSORT and QUICKSELECT [36] and empirical results for QUICKHEAPSORT [4] clearly suggest that sample sizes $k(n) = \Theta(\sqrt{n})$ are optimal asymptotically, but most of the relative savings for the expected case are already realized for $k \leq 10$. It is quite natural to expect similar behavior in QUICKXSORT, and it will be one goal of this article to precisely quantify these statements.

2.1. QuickMergesort

A natural candidate for X is MERGESORT: it is comparison-optimal up to the linear term (and quite close to optimal in the linear term), and needs a $\Theta(n)$-element-size buffer for practical implementations of merging.[4]

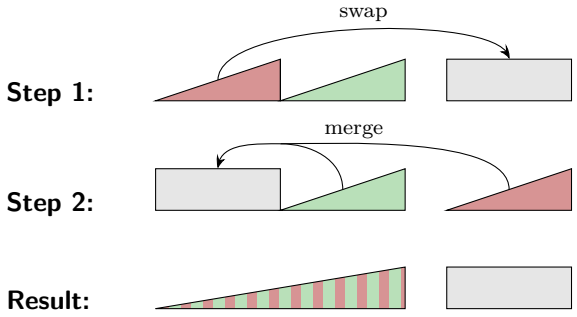


Figure 2: Simple merging procedure where one of the two runs fits into the buffer. (Buffer and input need not be adjacent in memory.)

Simple swap-based merge. To be usable in QUICKXSORT, we use a swap-based merge procedure: for merging two sorted runs, we first move the *smaller* of the two runs to a buffer using pairwise swaps (thereby bringing the buffer elements to the run's range); see Figure 2. Then, we merge the two runs back into the vacated slot. Pseudocode for this merge method is given in Algorithm C.1. When the second run is shorter, we use a symmetric version of Algorithm C.1. Using classical top-down or bottom-up MERGESORT as described in any algorithms textbook (e.g. [47]), we thus get along with $\alpha = \frac{1}{2}$.

The code in Algorithm C.1 illustrates that very simple adaptations suffice for QUICKMERGESORT. This merge procedure leaves the merged result in the range previously occupied by the two input runs. This "in-place"-style interface comes at the price of copying one run.

"Ping-pong" merge. Copying one run can be avoided if we instead write the merge result into an output buffer (and leaving it there). This saves element moves, but uses buffer space for all n elements, so we have $\alpha = 1$ here. The MERGESORT scaffold has to take care to

[4] Merging can be done in place using more advanced tricks (see, e.g., [20, 35]), but those tend not to be competitive in terms of running time with other sorting methods. By changing the global structure, a "pure" internal MERGESORT variant [30, 42] can be achieved using part of the input as a buffer (as in QUICKMERGESORT) at the expense of occasionally having to merge runs of very different lengths.

correctly orchestrate the merges, using the two arrays alternatingly; this alternating pattern resembles the ping-pong game, hence the name.

"Ping-pong" merge with smaller buffer. It is also possible to implement the "ping-pong" merge with $\alpha = \frac{1}{2}$. Indeed, the copying in Algorithm C.1 can be avoided by sorting the first subproblem recursively with "ping-pong" sort *into* the desired position in the buffer. Then, merging can proceed as in Algorithm C.1. Figure 3 illustrates this idea, which is easily realized with a recursive procedure; full code is given in Algorithm C.2. Our implementation of QuickMergesort (both for $\alpha = 1$ and $\alpha = 1/2$) uses this variant.

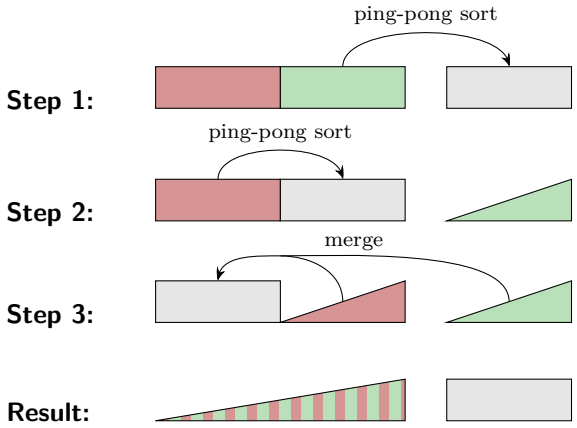


Figure 3: Mergesort with $\alpha = 1/2$ using ping-pong merges. Note that the recursive calls have a buffer area large enough to hold their entire subproblem at their disposal, so they can use input and output area alternatingly, following the standard ping-pong strategy.

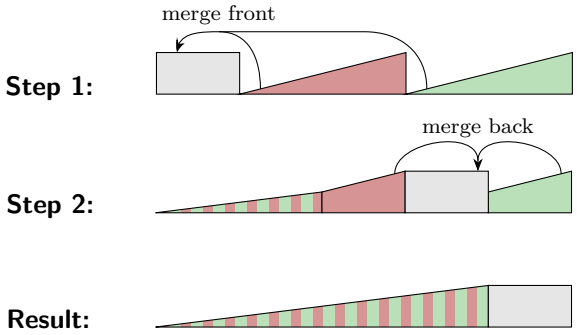


Figure 4: Reinhardt's merging procedure that needs only buffer space for *half of the smaller run*. In the first step, the two sequences are merged starting with the smallest elements until the empty space is filled. Then there is enough empty space to merge the sequences from the right into the final position.
Note that – unlike for the other merge procedures – array and buffer area in Reinhardt's merge are assumed to be adjacent and a symmetric version of the above is required when the buffer is initially located to the right of the input.

Reinhardt's merge. A third, less obvious alternative was proposed by Reinhardt [42], which allows to use an even smaller α for merges where input and buffer area form a contiguous region; see Figure 4. Assume we are given an array A with positions $A[1,\dots,t]$ being empty or containing dummy elements (to simplify the description, we assume the first case), $A[t+1,\dots,t+\ell]$ and $A[t+\ell+1,\dots,t+\ell+r]$ containing two sorted sequences. We wish to merge the two sequences into the space $A[1,\dots,\ell+r]$ (so that $A[\ell+r+1,\dots,t+\ell+r]$ becomes empty). We require that $r/2 \le t < r$. First we start from the left merging the two sequences into the empty space until there is no space left between the last element of the already merged part and the first element of the left sequence (first step in Figure 4). At this point, we know that at least t elements of the right sequence have been introduced into the merged part; so, the positions $t+\ell+1$ through $\ell+2t$ are empty now. Since $\ell+t+1 \le \ell+r \le \ell+2t$, in particular, $A[\ell+r]$ is empty now and we can start merging the two sequences right-to-left into the now empty space (where the right-most element is moved to position $A[\ell+r]$ – see the second step in Figure 4).

In order to have a balanced merge, we need $\ell = r$ and so $t \ge (\ell+r)/4$. Therefore, when applying this method in QuickMergesort, we have $\alpha = \frac{1}{4}$. We give full code of the merging procedure in Algorithm C.3. We also test an implementation of this variant in some of our experiments.

Remark 2.1 (Even less buffer space?): *Reinhardt goes even further: even with εn space, we can merge in linear time when ε is fixed by moving one run whenever we run out of space. Even though not more comparisons are needed, this method is quickly dominated by the additional data movements when $\varepsilon < \frac{1}{4}$, so we do not discuss it in this article.*

Another approach for dealing with less buffer space is to allow imbalanced merges: for both Reinhardt's merge and the simple swap-based merge, we need only additional space for (half) the size of the smaller *run. Hence, we can merge a short run into a long run with a relatively small buffer. The price of this method is that the number of comparisons increases, while the number of additional moves is better than with the previous method. We shed some more light on this approach in [14].*

Avoiding Stack Space. The standard version of Mergesort uses a top-down recursive formulation. It requires a stack of logarithmic height, which is usually deemed acceptable since it is dwarfed by the buffer space for merging. Since QuickMergesort removes the need for the latter, one might prefer to also avoid the logarithmic stack space.

An elementary solution is bottom-up Mergesort, where we form pairs of runs and merge them, except for, potentially, a lonely rightmost run. This variant occasionally merges two runs of very different sizes, which affects the overall performance (see Section 3.2).

A simple (but less well-known) modification of bottom-up Mergesort allows us to get the best of both worlds [21]: instead of always leaving a lonely rightmost run unmerged, we decide based on the number of runs whether we merge it with the penultimate run before proceeding with the next stage. The logic for handling these cases correctly makes this method more involved than top-down Mergesort, but constant extra space can be achieved without a loss in the number of comparisons, and with negligible running-time overhead.

2.2. QuickHeapsort

Another good option – and indeed the historically first one – for X is Heapsort.

Why Heapsort? In light of the fact that HEAPSORT is the only textbook method with reasonable overall performance that already sorts with *constant extra space,* this suggestion might be surprising. HEAPSORT rather appears to be the candidate *least* likely to profit from QUICKXSORT. Indeed, it is a refined variant of HEAPSORT that is an interesting candidate for X.

To work in place, standard HEAPSORT has to maintain the heap in a very rigid shape to store it in a contiguous region of the array. And this rigid structure comes at the price of extra comparisons. Standard HEAPSORT requires up to $2(h-1)$ comparisons to extract the maximum from a heap of height h, for an overall $2n \lg n \pm \mathcal{O}(n)$ comparisons in the worst case.

Comparisons can be saved by first finding the cascade of promotions (a.k.a. the *special path*), i.e., the path from the root to a leaf, always choosing to the larger of the two children. Then, in a second step, we find the correct insertion position along this line of the element currently occupying the last position of the heap area. The standard procedure corresponds to sequential search from the root. Floyd's optimization (a.k.a. bottom-up HEAPSORT [50]) instead uses sequential search from the leaf. It has a substantially higher chance to succeed early (in the second phase), and is probably optimal in that respect for the average case. If a better worst case is desired, one can use binary search on the special path, or even more sophisticated methods [22].

External Heapsort. In EXTERNALHEAPSORT, we avoid any such extra comparisons by relaxing the heap's shape. Extracted elements go to an output buffer and we only promote the elements along the special path into the gap left by the maximum. This leaves a gap at the leaf level, that we fill with a sentinel value smaller than any element's value (in the case of a max-heap). EXTERNALHEAPSORT uses $n \lg n \pm \mathcal{O}(n)$ comparisons in the worst case, but requires a buffer to hold n elements. By using it as our X in QUICKXSORT, we can avoid the extra space requirement.

When using EXTERNALHEAPSORT as X, we cannot simply overwrite gaps with sentinel values, though: we have to keep the buffer elements intact! Fortunately, the buffer elements themselves happen to work as sentinel values. If we sort the segment of large elements with EXTERNALHEAPSORT, we swap the max from the heap with a buffer element, which automatically is smaller than any remaining heap element and will thus never be promoted as long as any actual elements remain in the heap. We know when to stop since we know the segment sizes; after that many extractions, the right segment is sorted and the heap area contains only buffer elements.

We use a symmetric variant (with a min-oriented heap) if the left segment shall be sorted by X. For detailed code for the above procedure, we refer to [3] or [4].

Trading space for comparisons. Many options to further reduce the number of comparisons have been explored. Since these options demand extra space beyond an output buffer and cannot restore the original contents of that extra space, using them in QUICKXSORT does not yield an internal sorting method, but we briefly mention these variants here.

One option is to remember outcomes of sibling comparisons to avoid redundant comparisons in following steps [38]. In [4, Thm. 4], this is applied to QUICKHEAPSORT together with some further improvements using extra space.

Another option is to modify the heap property itself. In a weak heap, the root of a subtree is only larger than one of the subtrees, and we use an extra bit to store (and modify) which one it is. The more liberal structure makes construction of weak heaps more efficient: indeed, they can be constructed using $n-1$ comparisons. WeakHeapsort has been introduced by Dutton [7] and applied to QuickWeakHeapsort in [8]. We introduced a refined version of ExternalWeakHeapsort in [11] that works by the same principle as ExternalHeapsort; more details on this algorithm, its application in QuickWeakHeapsort, and the relation to Mergesort can be found in our preprint [10].

Due to the additional bit-array, which is not only space-consuming, but also costs time to access, WeakHeapsort and QuickWeakHeapsort are considerably slower than ordinary Heapsort, Mergesort, or Quicksort; see the experiments in [8, 11]. Therefore, we do not consider these variants here in more detail.

3. Preliminaries

In this section, we introduce some important notation and collect known results for reference. A comprehensive list of notation is given in Appendix A. Appendix B recapitulates a few mathematical preliminaries used in the analysis.

3.1. Notation

We use *Iverson's bracket*, $[\![stmt]\!]$, to mean 1 if *stmt* is true and 0 otherwise, as popularized by [23]. We also follow their notation $a^{\underline{b}}$ (resp. $a^{\overline{b}}$) for the falling (resp. rising) factorial power $a(a-1)\cdots(a-b+1)$ (resp. $a(a+1)\cdots(a+b-1)$). $\mathbb{P}[E]$ denotes the probability of event E, $\mathbb{E}[X]$ the expectation of random variable X. We write $X \stackrel{\mathcal{D}}{=} Y$ to denote equality in distribution.

With $f(n) = g(n) \pm h(n)$ we mean that $|f(n)-g(n)| \le h(n)$ for all n, and we use similar notation $f(n) = g(n) \pm \mathcal{O}(h(n))$ to state asymptotic bounds on the difference $|f(n)-g(n)| = \mathcal{O}(h(n))$. We remark that both use cases are examples of "one-way equalities" that are in common use for notational convenience, even though $\subseteq$ instead of $=$ would be formally more appropriate. Moreover, $f(n) \sim g(n)$ means $f(n) = g(n) \pm o(g(n))$. We use lg for the logarithm base 2, and ln for the natural logarithm. We reserve log for an unspecified base in $\mathcal{O}$-notation.

In our analysis, we make frequent use of the *beta distribution:* For $\lambda, \rho \in \mathbb{R}_{>0}$, $X \stackrel{\mathcal{D}}{=} \mathrm{Beta}(\lambda,\rho)$ if X admits the density $f_X(z) = z^{\lambda-1}(1-z)^{\rho-1}/\mathrm{B}(\lambda,\rho)$ where $\mathrm{B}(\lambda,\rho) = \int_0^1 z^{\lambda-1}(1-z)^{\rho-1}\,dz$ is the beta function. Moreover, for $\lambda,\rho \in \mathbb{R}_{>0}$ and $n \in \mathbb{N}_0$, we say X has a *beta-binomial distribution*, $X \stackrel{\mathcal{D}}{=} \mathrm{BetaBin}(n,\lambda,\rho)$, when $\mathbb{P}[X=i] = \binom{n}{i} \cdot \mathrm{B}(\lambda+i,\rho+(n-i))/\mathrm{B}(\lambda,\rho)$. We will also use the *regularized incomplete beta function*

$$I_{x,y}(\lambda,\rho) \;=\; \int_x^y \frac{z^{\lambda-1}(1-z)^{\rho-1}}{\mathrm{B}(\lambda,\rho)}\,dz, \qquad (\lambda,\rho \in \mathbb{R}_+, 0 \le x \le y \le 1). \tag{1}$$

Clearly $I_{0,1}(\lambda,\rho) = 1$.

3.2. Average costs of Mergesort

We recapitulate some known facts about standard mergesort. The average number of comparisons for Mergesort has the same – optimal – leading term $n \lg n$ in the worst and

best case; this is true for both the top-down and bottom-up variants. The coefficient of the *linear* term of the asymptotic expansion, though, is not a constant, but a bounded periodic function with period $\lg n$, and the functions differ for best, worst, and average case and the variants of MERGESORT [46, 17, 41, 26, 27].

For this paper, we confine ourselves to upper and lower *bounds* for the average case of the form $x(n) = an \lg n + bn \pm \mathcal{O}(n^{1-\varepsilon})$ with *constant* b valid for all n. Setting b to the infimum resp. supremum of the periodic function, we obtain the following lower resp. upper bounds for t̲op-d̲own [27] and b̲ottom-u̲p [41] MERGESORT

$$\begin{aligned} x_{\mathrm{td}}(n) &= n\lg n - \begin{cases} 1.2645n \\ 1.2408n \end{cases} + 2 \pm \mathcal{O}(n^{-1}) \qquad (2) \\ &= n\lg n - (1.25265 \pm 0.01185)n + 2 \pm \mathcal{O}(n^{-1}) \qquad \text{and} \\ x_{\mathrm{bu}}(n) &= n\lg n - \begin{cases} 1.2645n \\ 0.2645n \end{cases} \pm \mathcal{O}(1) \\ &= n\lg n - (0.7645 \pm 0.5)n \pm \mathcal{O}(1). \end{aligned}$$

3.3. Variance in Mergesort

First note that since MERGESORT's costs differ by $\mathcal{O}(n)$ for the best[5] and worst case, the variance is obviously in $\mathcal{O}(n^2)$. A closer look reveals that MERGESORT's costs are indeed much more concentrated and the variance is of order $\Theta(n)$: For a given size n, the overall costs are the sum of independent contributions from the individual merges, each of which has constant variance. Indeed, the only source of variability in the merge costs is that we do not need further comparisons once one of the two runs is exhausted.

More precisely, for standard top-down mergesort, X_n can be characterized by (see [17])

$$\begin{aligned} X_n &\overset{\mathcal{D}}{=} X_{\lceil n/2 \rceil} + X_{\lfloor n/2 \rfloor} + n - L_{\lceil n/2 \rceil, \lfloor n/2 \rfloor} \\ \mathbb{P}[L_{m,n} \le \ell] &= \frac{\binom{n+m-\ell}{m} + \binom{n+m-\ell}{n}}{\binom{n+m}{m}}. \end{aligned}$$

Following Mahmoud [34, eq. (10.3), eq. (10.1)], we find that the variance of the costs for a single merge is constant:

$$\begin{aligned} \mathbb{E}[L_{m,n}] &= \frac{m}{n+1} + \frac{n}{m+1} = m^{\underline{1}} n^{\underline{-1}} + n^{\underline{1}} m^{\underline{-1}} \\ \mathbb{E}[L_{m,n}^2] &= 2m^{\underline{2}} n^{\underline{-2}} + 2n^{\underline{2}} m^{\underline{-2}} \\ \mathrm{Var}[L_{m,n}] &= \mathbb{E}[L_{m,n}^2] + \mathbb{E}[L_{m,n}] - \mathbb{E}[L_{m,n}]^2 \\ &= 2m^{\underline{2}} n^{\underline{-2}} + 2n^{\underline{2}} m^{\underline{-2}} + m^{\underline{1}} n^{\underline{-1}} + n^{\underline{1}} m^{\underline{-1}} - \left(m^{\underline{1}} n^{\underline{-1}} + n^{\underline{1}} m^{\underline{-1}} \right)^2 \\ &\le 2, \qquad \text{for } |m-n| \le 1 \end{aligned}$$

[5] We assume here an unmodified standard MERGESORT variant that executes all merges in any case. In particular we assume the following folklore trick is *not* used: One can check (with one comparison) whether the two runs are already sorted prior to calling the merge routine and skip merging entirely if they are. This optimization leads to a linear best case and will increase the variance.

which gives an upper bound of $2n$ for the variance. Precise asymptotic expansions have been computed by Hwang [27]:

$$\mathrm{Var}[X_n] \;=\; n\phi(\lg(n)) - 2 + o(1)$$

for a periodic function $\phi(x) \in [0.30, 0.37]$.

4. Transfer theorems for QuickXsort

In this section, we set up a recurrence equation for the costs of QuickXsort, and we derive asymptotic transfer theorems for it, i.e., theorems that express the expected costs of QuickXsort in terms of the costs of X (as if used in isolation). We can then directly use known results about X from the literature; instantiations for actual choices for X are deferred to Section 5.

We mainly focus on the number of key comparisons as our cost model; the transfer theorems are, however, oblivious to this, and could easily be adapted for other cost measures.

As in plain Quicksort, the performance of QuickXsort is heavily influenced by the method for choosing pivots (though the influence is only on the linear term of the number of comparisons). We distinguish two regimes here. The first regime considers the case that the median of a large sample is used; more precisely, the sample size is chosen as a growing but sublinear function in the subproblem size. This method yields optimal asymptotic results and allows a rather clean analysis; it is covered in Section 4.2.

The other regime is a fixed constant k, treating it as a design parameter of the algorithm we choose once and for all. It is known for Quicksort that increasing the sample size yields rapidly diminishing marginal returns [45], and it is natural to assume that QuickXsort behaves similarly. Asymptotically, a growing sample size will eventually be better, but the evidence in Section 7 shows that a small, fixed sample size gives the best practical performance on realistic input sizes, so these variants deserve further study; unfortunately, the analysis becomes a bit more involved. Our transfer theorem for fixed k is given in Section 4.3.

In order to have a more concise presentation, we postpone the full proofs to Section 9 and give only some short sketches in this section. We start with some prerequisites and assumptions about X.

4.1. Prerequisites

For simplicity, we assume that inputs of size below a (constant) threshold w are sorted with X (using a constant amount of extra space). We require $w \geq k$ in the case of constant size-k samples for pivot selection. We could use another algorithm like StraightInsertionsort instead, but since QuickXsort recurses on only one subproblem, the base case only influences the *constant term* of costs (unlike for standard Quicksort).

We further assume that selecting the pivot from a sample of size k costs $s(k)$ comparisons, where we usually assume $s(k) = \Theta(k)$, i.e., a (expected-case) linear selection method is used.

Now, let $c(n)$ be the expected number of comparisons in QuickXsort on arrays of size n, where the expectation is over the random choices for selecting the pivots for partitioning. Our goal is to set up a recurrence equation for $c(n)$. We will justify first that such a recursive relation exists.

Preservation of randomness? For the QUICKSORT part of QUICKXSORT, only the ranks of the chosen pivot elements has an influence on the costs; partitioning itself always needs precisely one comparison per element.[6] When we choose the pivot elements randomly (from a random sample), the order of the input does not influence the costs of the QUICKSORT part of QUICKXSORT.

For general X, the sorting costs do depend on the order of the input, and we would like to use the average-case bounds for X, when it is applied on a random permutation. We may assume that our *initial* input is indeed a random permutation of the elements,[7] but this is not sufficient! We also have to guarantee that the inputs for recursive calls are again random permutations of their elements.

A simple sufficient condition for this "randomness-preserving" property is that X may not compare buffer contents. This is a natural requirement, e.g., for our MERGESORT variants. If no buffer elements are compared to each other and the original input is a random permutation of its elements, so are the segments after partitioning, and so will be the buffer after X has terminated. Then we can set up a recurrence equation for $c(n)$ using the average-case cost for X. We may also replace the random sampling of pivots by choosing any *fixed* positions without affecting the average costs $c(n)$.

However, not all candidates for X meet this requirement. (Basic) QUICKHEAPSORT does compare buffer elements to each other (see Section 2.2) and, indeed, the buffer elements are *not* in random order when the HEAPSORT part has finished. For such X, we assume that genuinely random samples for pivot selection are used. Moreover, we will have to use conservative bounds for the number of comparisons incurred by X, e.g., worst or best case results, as the input of X is not random anymore. This only allows to derive upper or lower bounds for $c(n)$, whereas for randomness preserving methods, the expected costs can be characterized precisely by the recurrence.

In both cases, we use $x(n)$ as (a bound for) the number of comparisons needed by X to sort n elements, and we will assume that

$$x(n) \;=\; an\lg n + bn \;\pm\; \mathcal{O}(n^{1-\varepsilon}), \qquad (n\to\infty),$$

for constants a, b and $\varepsilon \in (0,1]$.

4.1.1. The QuickXsort Recurrence

We can now describe the expected costs $c(n)$ of QUICKXSORT by a recurrence relation; it directly follows the recursive nature of the algorithm:

$$c(n) \;=\; \sum_{r=1}^{2} \mathbb{E}[A_r\, c(J_r)] \;+\; t(n) \tag{3}$$

[6] We remark that this is no longer true for multiway partitioning methods where the number of comparisons per element is not necessarily the same for all possible outcomes. Similarly, the number of swaps in the standard partitioning method depends not only on the rank of the pivot, but also on how "displaced" the elements in the input are.

[7] It is, indeed, a reasonable option to *enforce* this assumption in an implementation by an explicit *random shuffle* of the input before we start sorting. Sedgewick and Wayne, for example, do this for the implementation of QUICKSORT in their textbook [47]. In the context of QUICKXSORT, a full random shuffle is overkill, though; see Remark 5.2 for more discussion.

Here, A_1 and A_2 are indicator random variables with $A_1 = [\![\text{recurse on left subproblem}]\!]$ $A_2 = [\![\text{recurse on right subproblem}]\!]$, and $t(n)$ is the "toll function" of the recurrence, i.e., the non-recursive costs, given by

$$t(n) \;=\; \underbrace{n - k(n)}_{\text{partitioning}} \;+\; \underbrace{s\big(k(n)\big)}_{\text{pivot sampling}} \;+\; \underbrace{\textstyle\sum_{r=1}^{2} \mathbb{E}[A_r\, x(J_{3-r})]}_{\text{calls to X}}. \tag{4}$$

A rigorous derivation and precise description of A_r is given in Section 9.2.

4.2. Growing sample sizes

In this section, we assume that the pivot is selected as the median of $k = k(n)$ elements where $k(n)$ grows when n grows. We show that under natural assumptions the average number of comparisons of X and of median-of-$k(n)$ QuickXsort differ only by an $o(n)$-term. After that, we give a bound on the probability for running into a worst case and discuss some ideas to obtain a worst-case $\mathcal{O}(n \log n)$-algorithm.

4.2.1. Expected costs

The following theorem allows to transfer an asymptotic approximation for the costs of X to an asymptotic approximation of the costs of QuickXsort. We will apply this theorem to concrete methods X in Section 5.

Theorem 4.1 (Transfer theorem (expected costs, growing k)): *Let $c(n)$ be defined by Equation (3) (the recurrence for the expected costs of QuickXsort) and assume $x(n)$ (the costs of X) and $k = k(n)$ (the sample size) fulfill $x(n) = an \lg n + bn \pm o(n)$ for constants $a \geq 1$ and b, and $k = k(n) \in \omega(1) \cap o(n)$ as $n \to \infty$ with $1 \leq k(n) \leq n$ for all n.*

Then, $c(n) \leq x(n) + o(n)$. For $a = 1$, the above holds with equality, i.e., $c(n) = x(n) + c'(n)$ with $c'(n) = o(n)$. Moreover, in the typical case with $k(n) = \Theta(n^\kappa)$ for $\kappa \in (0,1)$ and $x(n) = an \lg n + bn \pm \mathcal{O}(n^\delta)$ with $\delta \in [0,1)$, we have for any fixed $\varepsilon > 0$ that

$$c'(n) \;=\; \Theta(n^{\max\{\kappa, 1-\kappa\}}) \;\pm\; \mathcal{O}(n^{\max\{\delta, 1/2+\varepsilon\}}).$$

A fully detailed proof is given in Section 9.3; we given an overview of the main arguments here. To prove Theorem 4.1, we consider the difference $c'(n) = c(n) - x(n)$ and observe that it fulfills a very similar recurrence as $c(n)$ itself:

$$\begin{aligned} c'(n) \;&=\; n - k(n) \;+\; s(k(n)) \;+\; \mathbb{E}\Big[A_1 \cdot \big(c'(J_1) + x(J_1) + x(J_2)\big)\Big] \\ &\qquad +\; \mathbb{E}\Big[A_2 \cdot \big(c'(J_2) + x(J_2) + x(J_1)\big)\Big] \;-\; x(n) \\ &=\; \mathbb{E}[A_1\, c'(J_1)] + \mathbb{E}[A_2\, c'(J_2)] \;+\; \underbrace{n - k(n) + s\big(k(n)\big) + \mathbb{E}[x(J_1)] + \mathbb{E}[x(J_2)] - x(n)}_{t'(n)}. \end{aligned} \tag{5}$$

Note how taking the difference here "magically" turns the complicated terms $\mathbb{E}[A_r x(J_{3-r})]$ from $t(n)$ into the simpler $\mathbb{E}[x(J_r)]$ terms in $t'(n)$.

The next step in the proof is to find a precise asymptotic approximation for $t'(n)$, given in the following lemma.

Lemma 4.2 (Approximating $t'(n)$): *Let $t'(n)$ as in Equation (5). Then for $\varepsilon > 0$, we have*

$$t'(n) \;=\; (1-a)n \;+\; \Theta\Big(k(n) + \tfrac{n}{k(n)}\Big) \;\pm\; \mathcal{O}\Big(\xi(n) + n^{1/2+\varepsilon}\Big).$$

Moreover, if $a = 1$, $k(n) = \Theta(n^\kappa)$ for $\kappa \in (0,1)$ and $\xi(n) = \mathcal{O}(n^\delta)$ for $\delta \in [0,1)$, we have

$$t'(n) \;=\; \Theta\Big(n^{\max\{\kappa,1-\kappa\}}\Big) \;\pm\; \mathcal{O}\Big(n^{\max\{\delta,1/2+\varepsilon\}}\Big).$$

While it is obvious that the $n \lg n$ terms cancel, a much closer look and heavier mathematical machinery is required to obtain the precise bound as stated; we defer the proof to Section 9.3.2.

It remains to bound $c'(n)$; since $c'(n)$ is only the error term, we can confine ourselves with a simple upper bound. Since our sample size grows with n, the subproblem sizes J_1 and J_2 are concentrated around $\frac{n}{2}$; we can use this fact to treat Equation (5) (almost) like a standard divide-and-conquer recurrence using the (textbook) recursion-tree method to obtain the following result.

Lemma 4.3: *Let $\hat{t} : \mathbb{R}_{\geq 0} \to \mathbb{R}_{\geq 0}$ be monotonically increasing and consider the recurrence*

$$\hat{c}(n) \;=\; \mathbb{E}\big[A_1\,\hat{c}(J_1)\big] + \mathbb{E}\big[A_2\,\hat{c}(J_2)\big] + \hat{t}(n)$$

with $\hat{c}(n) = c_0$ for $n \leq 1$. Then for any constant $\beta \in (\frac{1}{2}, 1)$ there is a constant $C = C(\beta) > 0$ such that

$$\hat{c}(n) \;\leq\; C \sum_{i=0}^{\lceil \log_{1/\beta}(n) \rceil} \hat{t}(n\beta^i)$$

In the application of Lemma 4.3 for our $c'(n)$, additional care is needed since our $t'(n)$ can, in general, be positive or negative. Treating different regimes of the claim separately finally yields Theorem 4.1.

* * *

We note that our Theorem 4.1 considerably strengthens the error term from $o(n)$ in versions of this theorem in earlier work to $\mathcal{O}(n^{1/2+\varepsilon})$ (for $k(n) = \sqrt{n}$). Since this error term is the only difference between the costs of QuickXsort and X (for $a = 1$), we feel that this improved bound is not merely a technical contribution, but significantly strengthens our confidence in the utility and practicality of QuickXsort as an algorithmic template.

Remark 4.4 (Optimal sample sizes): *The experiments in [4] and the results for QuickSort in [36] suggest that sample sizes $k(n) = \Theta(\sqrt{n})$ are likely to be optimal w.r.t. balancing costs of pivot selection and benefit of better-quality pivots within the lower order terms also for QuickXsort. Theorem 4.1 gives a proof for this in a special situation: assume that $a = 1$, the error term $\xi(n) \in \mathcal{O}(n^\delta)$ for some $\delta \in [0, \frac{1}{2}]$ and that we are restricted to sample sizes $k(n) = \Theta(n^\kappa)$, for $\kappa \in (0,1)$. For this case, Theorem 4.1 shows that $\kappa = \frac{1}{2}$ is the optimal choice, i.e., $k(n) = \sqrt{n}$ has the "best polynomial growth" among all feasible polynomial sample sizes.*

Theorem 4.1 shows that for methods X that have optimal costs up to linear terms ($a = 1$), also median-of-$k(n)$ QuickXsort with $k(n) = \Theta(n^\kappa)$ and $\kappa \in (0,1)$ as $n \to \infty$ is optimal

up to linear terms. We obtain the best lower-order terms with median-of-$\sqrt{n}$ QuickXsort, namely $c(n) = x(n) \pm \mathcal{O}(n^{1/2+\varepsilon})$, and we will in the following focus on this case.

Note that our proof actually gives slightly more information than stated in the theorem for the case that the cost of X are not optimal in the leading-term coefficient ($a > 1$). Then QuickXsort uses asymptotically *fewer* comparisons than X, whereas for X with optimal leading-term costs, QuickXsort uses slightly more comparisons.

4.2.2. Large-deviation bounds

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, a running time of $\Theta(n^{3/2})$ is obtained. Nevertheless, we can prove that such a worst case is very unlikely.

Proposition 4.5: *Let $x_{\mathrm{wc}}(n)$ be the worst case number of comparisons of the algorithm X and let $\varepsilon > 0$. The probability that median-of-$\sqrt{n}$ QuickXsort needs more than $x_{\mathrm{wc}}(n)+6n$ comparisons is less than $(3/4+\varepsilon)^{\sqrt[4]{n}}$ for n large enough.*

The proof relies on a simple concentration inequality (Lemma 9.4) for the rank of the pivots; full details of the computation are given in Section 9.4. Note that we do not aim for a tight bound here.

4.2.3. Worst-case guarantees

In order to obtain a provable bound for the worst case complexity we apply a simple trick similar to the one used in Introsort [40]. We choose some $\delta \in (0, 1/2)$. Now, whenever the pivot is more than δn off from the median (i.e., if $J_1 \le (1/2-\delta)n$ or $J_2 \le (1/2-\delta)n$), we choose the next pivot as median of the whole array using the median-of-medians algorithm [1] (or some other selection algorithm with a linear worst case). Afterwards we continue with the usual sampling strategy. We call this strategy *median-of-medians fallback pivot selection.*

Remark 4.6 (Alternative: Fallback to different algorithm): *Instead of choosing the next pivot as median, we can also switch to an entirely different sorting algorithm – as done in Introsort, and as the first two authors originally proposed in [11]. The advantage is that better worst-case bounds can be achieved: we showed that the worst-case is only $n+o(n)$ comparisons above the worst case of the fallback algorithm. Thus, using Reinhardt's Mergesort [42], we obtain a worst case of $n \log n - 0.25n + o(n)$.*

However, here we follow a different approach for two reasons: first, we prefer the self-contained description of the algorithm; second, we are not aware of a fallback algorithm which in practice performs better than our approach: Heapsort and most internal Mergesort variants are considerably slower. Furthermore, at the time of this writing, we were not aware of any implementation of Reinhardt's Mergesort.[8]

Theorem 4.7 (QuickXsort Worst-Case): *Let X be a sorting algorithm with at most $x(n) = n\lg n + bn + o(n)$ comparisons in the average case and $x_{\mathrm{wc}}(n) = n\lg n + \mathcal{O}(n)$ comparisons in the worst case and let $k(n) \in \omega(1) \cap o(n)$ with $1 \le k(n) \le n$ for all n. If*

[8] Meanwhile, an implementation has been created and made available at `https://github.com/rbroesamle/Implementierung-von-in-place-Mergesort-Algorithmen`. Although it is surprisingly fast, the experiments by its creators suggest that it does not beat our approach.

$k(n) = \omega(\sqrt{n})$, we additionally require that always some worst-case linear time algorithm is used for pivot selection, (e.g., using IntroSelect [40] or the median-of-medians algorithm); otherwise, the worst-case is allowed to be at most quadratic, (e.g., using QuickSelect).

Then, median-of-$k(n)$ QuickXsort with median-of-medians fallback pivot selection is a sorting algorithm that performs $x(n)+o(n)$ comparisons in the average case and $n \lg n + \mathcal{O}(n)$ comparisons in the worst case.

Again, the full proof is deferred to Section 9.4, and we sketch the main arguments here. With our fallback, there can be at most $\max\{2 \lg n, \log_{1/2+\delta} n\}$ rounds of partitioning, so the worst case is $n \lg n + \mathcal{O}(n)$ comparisons (using the additional requirement that pivot selection takes linear time in the worst case).

For the average case, we distinguish the cases where a pivot choice is "bad" (next pivot selected as true median of the whole array) resp. "good" (otherwise). By Lemma 9.4 the "bad" pivot choice occurs with a probability in $o(1)$. This allows us to reduce the recurrence to the one we already solved in the proof Theorem 4.1.

* * *

By applying median-of-medians fallback pivot selection, the average case changes only in the $o(n)$-terms, and we get a strong worst-case guarantee. Note, however, that the $\mathcal{O}(n)$-term for the worst case of QuickXsort is rather large because of the median-of-medians algorithm. In [14], the first two authors further elaborate on the technique of median-of-medians pivot selection and show how to bring down the $\mathcal{O}(n)$-term for the worst case to $3.58n$ for QuickMergesort.

4.3. Fixed sample sizes

We now consider the practically relevant version of QuickXsort, where we choose pivots as the median of a sample of fixed size k. Setting $k = 1$ corresponds to selecting pivots uniformly at random; good practical performance is often achieved for moderate values, say, $k = 3, \dots, 9$.

For very small subproblems, when $n \le w$ for a constant $w \ge k$, we directly sort with X (using constant extra space). Clearly this only influences the *constant term* of costs in QuickXsort. Moreover, the costs of sampling pivots is $\mathcal{O}(\log n)$ in expectation (for constant k and w); we will see that the costs $s(k)$ of finding the median of the k sample elements is negligible here.

We now state our transfer theorem for median-of-k QuickXsort when k is fixed. Recall that $I_{x,y}(\lambda, \rho)$ denotes the regularized incomplete beta function (Equation (1) on page 13).

Theorem 4.8 (Transfer theorem (expected costs, fixed k)): *Let $c(n)$ be defined by Equation (3), the recurrence for the expected costs of QuickXsort, and assume $x(n)$, the expected cost of X, fulfills $x(n) = an \lg n + bn \pm \mathcal{O}(n^{1-\varepsilon})$ for constants $\varepsilon \in (0, 1]$, $a \ge 1$ and $b \in \mathbb{R}$. Assume further that the sample size k is a fixed odd constant $k = 2t + 1$, $t \in \mathbb{N}_0$. Then it holds that*

$$c(n) \;=\; x(n) \;+\; q \cdot n \;\pm\; \mathcal{O}(n^{1-\varepsilon} + \log n),$$

where

$$\begin{aligned} q &= \frac{1}{H} - a \cdot \frac{H_{k+1} - H_{t+1}}{H \ln 2} \\ H &= I_{0,\frac{\alpha}{1+\alpha}}(t+2, t+1) \;+\; I_{\frac{1}{2},\frac{1}{1+\alpha}}(t+2, t+1). \end{aligned}$$

We start with Equation (5) on page 17, the recurrence for $c'(n) = c(n) - x(n)$. By the same arguments as in the proof of Theorem 4.1, we have

$$t'(n) \;=\; \left(1 + 2a\mathbb{E}\Big[\tfrac{J}{n}\lg(\tfrac{J}{n})\Big]\right)n \;\pm\; \mathcal{O}(n^{1-\varepsilon}). \tag{6}$$

The main complication for fixed k is that – unlike for the median-of-$\sqrt{n}$ case, where pivots are close to the overall median with high probability – $\frac{J}{n}$ here has significant variance. We will thus have to compute $\mathbb{E}[\frac{J}{n}\lg(\frac{J}{n})]$ more precisely and also solve the recurrence for $c'(n)$ precisely. For that, we need additional techniques over what we used in the previous section: beta-integral approximations and Roura's continuous master theorem [44]. The latter is a tool to obtain precise asymptotic approximations (including constants) for certain full-history recurrences. We restate the theorem in Appendix B for the reader's convenience. The detailed computations required for its application on Equation (5) are given in Section 9.5.

To obtain an asymptotic approximation of $t'(n)$, we approximate $\frac{J}{n}$ by a beta distributed variable, relying on a local limit law for beta-binomial variables by the last author (Lemma 9.2 in this article). Carefully tracing the error of this approximation yields the following result; its proof and the full details of the proof of Theorem 4.8 are given in Section 9.5.

Lemma 4.9 (Beta-integral approximation): *Let $J \overset{\mathcal{D}}{=} \mathrm{BetaBin}(n - c_1, \lambda, \rho) + c_2$ be a random variable that differs by fixed constants c_1 and c_2 from a beta-binomial variable with parameters $n \in \mathbb{N}$ and $\lambda, \rho \in \mathbb{N}_{\geq 1}$. Then for any $\eta \in (0,1)$ we have*

$$\mathbb{E}\Big[\tfrac{J}{n}\ln\tfrac{J}{n}\Big] \;=\; \frac{\lambda}{\lambda+\rho}(H_\lambda - H_{\lambda+\rho}) \;\pm\; \mathcal{O}(n^{-\eta}), \qquad (n \to \infty).$$

The QuickXsort penalty. Since all our choices for X are optimal up to linear terms, so will QuickXsort be. We thus always have $a = 1$ in Theorem 4.8, whereas b (and the allowable α) depend on X. Going from X to QuickXsort then adds a "penalty" q in the linear term that depends on the sampling size (and α), *but not on X itself*. Table 2 shows that this penalty is roughly n without sampling, but can be reduced drastically when choosing pivots from a sample of 3 or 5 elements.

	$k=1$	$k=3$	$k=5$	$k=7$	$k=21$	$t \to \infty$
$\alpha = 1$	1.1146	0.5070	0.3210	0.2328	0.07705	0
$\alpha = 1/2$	0.9120	0.4050	0.2526	0.1815	0.05956	0
$\alpha = 1/4$	0.6480	0.2967	0.1921	0.1431	0.05498	0

Table 2: QuickXsort penalty. QuickXsort with $x(n) = n\lg n + bn$ yields $c(n) = n\lg n + (q+b)n$, where q, the QuickXsort penalty, is given in the table.

As we increase the sample size, we converge to the situation for growing sample sizes where no linear-term penalty is left (recall Theorem 4.1). That q is less than 0.08 already

for a sample of 21 elements clearly indicates that most of the benefits of pivot sampling are achieved for moderate sample sizes. It is noteworthy that the improvement from no sampling to median-of-3 yields a reduction of q by more than 50 %. By contrast, its effect on QUICKSORT itself only reduces the leading term of costs by 15 %, from $2n \ln n$ to $\frac{12}{7} n \ln n$.

4.4. Variance

If an algorithm's cost regularly exceeds its expectation by far, good expected performance is not enough. The purpose of this section is to explore what influence the distribution of the costs of X have on QUICKXSORT. To that end, we study the *variance* of the number of comparisons in QUICKXSORT; similar to the expected costs, we prove a transfer theorem for a recurrence for the variance.

Despite the fact that our transfer theorem is not applicable for periodic linear terms in $x(n)$, it yields excellent approximations of the actual variances in QUICKMERGESORT (see Section 5.2), and shows that costs are indeed concentrated around their expectation.

We assume a constant sample size k in this section. Formally, our result is the following.

Theorem 4.10 (Variance of QuickXsort): *Assume X is a sorting method whose comparison cost has expectation $x(n) = an \lg n + bn \pm \mathcal{O}(n^{1-\varepsilon})$ and variance $v_X(n) = a_v n^2 + \mathcal{O}(n^{2-\varepsilon})$ for a constant a_v and $\varepsilon > 0$; the case $a_v = 0$ is allowed. Moreover, assume QUICKXSORT preserves randomness.*

Under the technical conjecture $t_v(n) = \mathcal{O}(n^2)$ (as defined in the proof), median-of-k QUICKXSORT is a sorting method whose comparison cost has variance $v(n) \sim cn^2$ for an explicitly computable constant c that depends only on k, α and a_v.

This transfer theorem can be proven with similar techniques as for the expected value, but the computations become more involved. We give the full details in Section 9.6, where we first state a distributional recurrence for the random number of comparisons, from which a recurrence for the second moment can be derived. An asymptotic solution of the recurrence is found using beta-integral approximations and Roura's continuous master theorem.

Remark 4.11: *We could confirm the conjecture $t_v(n) = \mathcal{O}(n^2)$ for all tried combinations of values for α and k, (in particular those in Table 3), but were not able to prove it in the general setting, so we have to formally keep it as a prerequisite. We have no reason to believe it is not always fulfilled.*

	$k = 1$	$k = 3$	$k = 9$
$\alpha = 1$	$0.4344 + 0.2000a_v$	$0.1119 + 0.2195a_v$	$0.01763 + 0.2477a_v$
$\alpha = 1/2$	$0.4281 + 0.2941a_v$	$0.1068 + 0.3234a_v$	$0.01572 + 0.3632a_v$
$\alpha = 1/4$	$0.3134 + 0.4413a_v$	$0.0728 + 0.4550a_v$	$0.00988 + 0.4483a_v$

Table 3: Leading term coefficients of the variance of QUICKXSORT.

Variance for methods with optimal leading term. In Table 3, we give the leading-term coefficient for the variance (i.e., c in the terminology of Theorem 4.10) for several values of α and k. We fix $a = 1$, i.e., we consider methods X with optimal leading term; the constant b

of the linear term in $x(n)$ does not influence the leading term of the variance. In the results, we keep a_v as a variable, although for the methods X of most interest, namely MERGESORT and EXTERNALHEAPSORT, we actually have $a_v = 0$.

5. Analysis of QuickMergesort and QuickHeapsort

In the previous section, we derived several general results about QUICKXSORT; we now apply those to the concrete choices for X introduced in Section 2. Besides describing how to overcome technical complications in the analysis, we also discuss our results. Comparing with analyses and measured comparison counts from previous work, we find that our exact solutions for the QUICKXSORT recurrence yield more accurate predictions for the overall number of comparisons.

5.1. Expected costs of QuickMergesort

We use QUICKMERGESORT here to mean the "ping-pong" variant with smaller buffer ($\alpha = \frac{1}{2}$) as illustrated in Figure 3 (page 10). Among the variations of MERGESORT (that are all usable in QUICKXSORT) we discussed in Section 2.1, this is the most promising option in terms of practical performance. The analysis of the other variants is very similar.

We further assume a variant of MERGESORT that generates optimally balanced merges. While top-down mergesort is the typical choice for that, recall that there are variations of bottom-up mergesort that achieve the same result without logarithmic extra space for a recursion stack.

Corollary 5.1 (Average Case QuickMergesort): *The following results hold when sorting a random permutation of n elements.*

(a) *Median-of-$\sqrt{n}$ QUICKMERGESORT is an internal sorting algorithm that performs $n \lg n - (1.25265 \pm 0.01185)n \pm \mathcal{O}(n^{1/2+\varepsilon})$ comparisons on average for any constant $\varepsilon > 0$.*

(b) *Median-of-3 QUICKMERGESORT (with $\alpha = 1/2$) is an internal sorting algorithm that performs $n \lg n - (0.84765 \pm 0.01185)n \pm \mathcal{O}(\log n)$ comparisons on average.*

Proof: We first note that MERGESORT never compares buffer elements to each other; buffer contents are only accessed in swap operations. Therefore, QUICKMERGESORT preserves randomness: if the original input is a random permutation, both the calls to MERGESORT and the recursive call operate on a random permutation of the respective elements. The recurrence for $c(n)$ thus gives the exact expected costs of QUICKMERGESORT when we insert for $x(n)$ the expected number of comparisons used by MERGESORT on a random permutation of n elements. The latter is given in Equation (2) on page 14.

Note that these asymptotic approximations in Equation (2) are *not* of the form required for our transfer theorems; we need a *constant* coefficient in the linear term. But since $c(n)$ is a monotonically increasing function in $x(n)$, we can use upper and lower bounds on $x(n)$ to derive upper and lower bounds on $c(n)$. We thus apply Theorem 4.1 and Theorem 4.8 separately with $x(n)$ replaced by

$$\begin{aligned} \underline{x}(n) &= n \lg n - 1.2645n - \mathcal{O}(1) \qquad \text{resp.} \\ \overline{x}(n) &= n \lg n - 1.2408n + \mathcal{O}(1). \end{aligned}$$

For part (a), we find $\underline{x}(n) \pm \mathcal{O}(n^{1/2+\varepsilon}) \le c(n) \le \overline{x}(n) \pm \mathcal{O}(n^{1/2+\varepsilon})$ for any fixed $\varepsilon > 0$. Comparing upper and lower bound yields the claim.

For part (b) we obtain with $q = 0.4050$ the bounds $\underline{x}(n) + qn \pm \mathcal{O}(\log n) \le c(n) \le \overline{x}(n) + qn \pm \mathcal{O}(\log n)$. □

Remark 5.2 (Randomization vs average case): *We can also prove a bound for the expected performance on any input, where the expectation is taken over the random choices for pivot sampling. By using an upper bound for the worst case of* MERGESORT, $\overline{x}(n) = n \lg n - 0.91392n + 1$, *we find that the expected number of comparisons is at most* $n \lg n - 0.91392n \pm \mathcal{O}(n^{1/2+\varepsilon})$ *for median-of-*$\sqrt{n}$ QUICKMERGESORT *and at most* $n \lg n - 0.50892n + \mathcal{O}(\log n)$ *for median-of-3* QUICKMERGESORT.

Of course, we could also obtain the bounds of Corollary 5.1 for any input by randomly shuffling the array before sorting it. Note, however, that unlike for standard QUICKSORT, *using random samples in* QUICKMERGESORT *involves much fewer calls to a random number generator than a full shuffle since we expect only* $\mathcal{O}(\log n)$ *sampling rounds.*

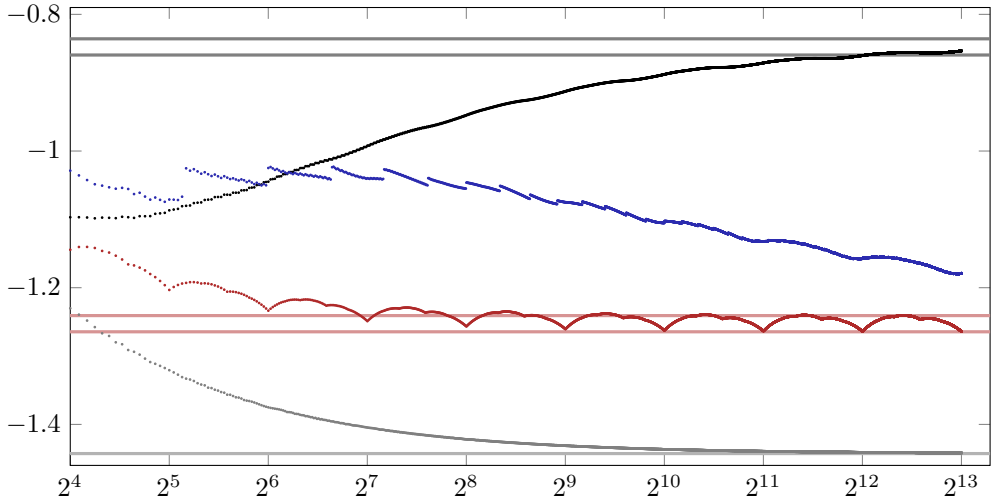


Figure 5: Exact comparison count of MERGESORT (red), median-of-3 QUICKMERGESORT (black) and median-of-$\sqrt{n}$ QUICKMERGESORT (blue) for small input sizes, computed from the recurrence. The information-theoretic lower bound (for the average case) is also shown (gray). The x-axis shows n (logarithmic), the y-axis shows $\big(c(n) - n \lg n\big)/n$. The horizontal lines are the supremum and infimum of the asymptotic periodic terms. (For median-of-$\sqrt{n}$ QUICKMERGESORT, these are the same as for MERGESORT.)

Given that the error term of our approximation for fixed k is only of logarithmic growth, we can expect very good predictive quality for our asymptotic approximation. This is confirmed by numbers reported in Section 7.1 below. The relative error between the exact value of $c(n)$ and the approximation $n \lg n - 0.84765n$ is below 1 % for $n \ge 400$.

Figure 5 gives a closer look for small n. The numbers are computed from the exact recurrences for MERGESORT (see Section 3.3) and QUICKMERGESORT (Equation (3)) by recursively tabulating $c(n)$ for all $n \le 2^{13} = 8192$. For the pivot sampling costs $s(k)$, we use the average cost of finding the median with Quickselect, which are known precisely [33,

p. 14]. For the numbers for median-of-$\sqrt{n}$ QuickMergesort, we use $k(n) = 2\lfloor\sqrt{n}/2\rfloor + 1$. The computations were done using Mathematica.

For standard Mergesort, the linear coefficient reaches its asymptotic regime rather quickly; this is due to the absence of a logarithmic term. For median-of-3 QuickMergesort, considerably larger inputs are needed, but for $n \geq 2000$ we are again close to the asymptotic regime. Median-of-$\sqrt{n}$ QuickMergesort needs substantially larger inputs than considered here to come close to Mergesort. It is interesting to note that for roughly $n \leq 100$, the median-of-3 variant is better, but from then onwards, the median-of-$\sqrt{n}$ version uses fewer comparisons.

Figure 5 shows the well-known periodic behavior for Mergesort. Oscillations are clearly visible also for QuickMergesort, but compared to the rather sharp "bumps" in Mergesort's cost, QuickMergesort's costs are smoothed out. Figure 5 also confirms that the amplitude of the periodic term is very small in QuickMergesort.

Skewed Pivots for QuickMergesort? For Mergesort with $\alpha = \frac{1}{2}$ the largest fraction of elements we can sort by Mergesort in one step is $\frac{2}{3}$; could it hence be beneficial to use a slightly skewed pivot, so as to increase the subproblem size for Mergesort and decrease the size for recursive calls?

The answer is "no". Even for $\alpha = \frac{1}{2}$, the best choice is to use the *median* of the sample. For QuickMergesort, skewed pivots turn out to be a pessimization, despite the fact that we sort a larger part by Mergesort. Our analysis for fixed-size samples can be extended to skewed sampling schemes, but to illustrate this point we confine ourselves to a short visit to "wishful-thinking land": Let us assume that we can find exact quantiles in the input for free. We can then show (e.g., with Roura's discrete master theorem [44]) that if we always pick the exact ρ-quantile of the input, for $\rho \in (0,1)$, the overall costs are

$$c_\rho(n) = \begin{cases} n\lg n + \left(\frac{1-h(\rho)}{1-\rho} + b\right)n \pm \mathcal{O}(n^{1-\varepsilon}) & \text{if } \rho \in (\frac{1}{3},\frac{1}{2}) \cup (\frac{2}{3},1) \\ n\lg n + \left(\frac{1-h(\rho)}{\rho} + b\right)n \pm \mathcal{O}(n^{1-\varepsilon}) & \text{otherwise} \end{cases}$$

for $h(x) = -x\lg x - (1-x)\lg(1-x)$. The coefficient of the linear term has a *strict minimum* at $\rho = \frac{1}{2}$; any choice other than the median makes QuickMergesort (asymptotically) worse.

5.2. Variance in QuickMergesort

Since the variance of Mergesort is subquadratic, Theorem 4.10 would be applied with $a_v = 0$, and we obtain, e.g., a variance of $0.4281n^2$ for $k = 1$ and $0.1068n^2$ for $k = 3$. Interestingly, these results do not depend on our choice for the constant b of the linear term of $x(n)$.

They match empirical numbers quite well. There is still a noticeable difference in Figure 6, which compares the above approximations with exact values for small n computed from the recurrence. For larger n, though, the accuracy is stunningly good, see Figure 12 and Table 6 in the experiments section.

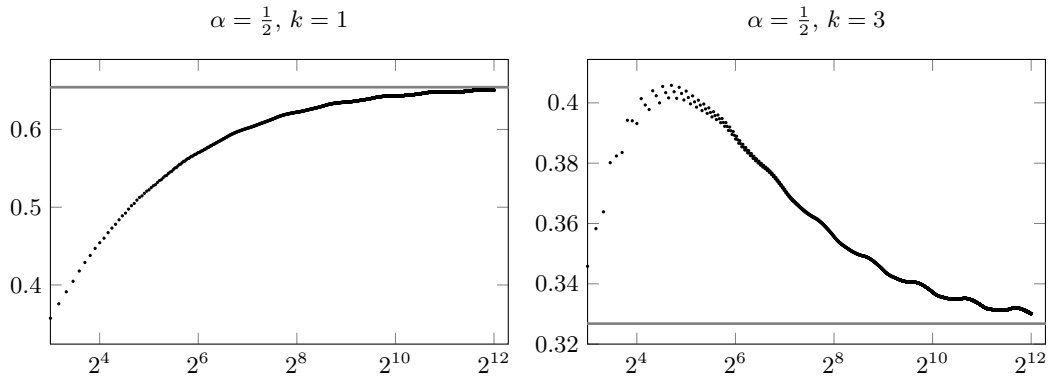


Figure 6: Exact values for the normalized standard deviation in QuickMergesort (computed from the exact recurrence for the second moment) and the asymptotic approximation from Table 3 (gray line). The x-axis shows the inputs size n (logarithmic) and the y-axis is the standard deviation of the number of comparisons divided by n. The plots show different sample sizes.

Fine print. Although our transfer theorem is perfectly valid and fits Monte Carlo simulations very well, it is *not* actually applicable to QuickMergesort. The reason for this are the tiny periodic fluctuations (w.r.t. n) in the expected cost of Mergesort.

For the expectation, upper and lower bounds for $x(n)$ were sufficient to derive upper and lower bounds for the costs of QuickXsort. Determining the precise influence of fluctuations in QuickXsort's expected cost is an interesting topic for future research, but since the bounds are so close, our approach taken in this paper is certainly sufficient on practical grounds. For the variance, the situation is different. The variance of QuickMergesort is influenced by the periodic terms of the *expected* costs of Mergesort, and simple arguments do not yield rigorous bounds (in either direction).

Intuitively QuickMergesort acts as a smoothing on the costs of Mergesort since subproblem sizes are random. It is therefore quite expected to find very smooth periodic influences of small amplitude. The fact that our estimate does not depend on b or the precise variance of Mergesort at all, gives hope that it is a very good approximation, but it remains heuristic approximation. A rigorous analysis of the variance of QuickMergesort remains the subject of future work.

5.3. QuickHeapsort

By QuickHeapsort we refer to QuickXsort using the basic ExternalHeapsort version (as described in Section 2.2) as X. We have the following result.

Corollary 5.3 (Expected Case QuickHeapsort): *The following results hold for the expected number of comparisons where the expectation is taken over the random choices of the pivots.*

(a) *Median-of-$\sqrt{n}$ QuickHeapsort is an internal sorting algorithm that performs $n \lg n + (0.54305 \pm 0.54305)n \pm \mathcal{O}(n^{1/2+\varepsilon})$ comparisons for any constant $\varepsilon > 0$.*

(b) *Median-of-3 QuickHeapsort is an internal sorting algorithm that performs $n \lg n + (1.05005 \pm 0.54305)n \pm \mathcal{O}(n^{\varepsilon})$ comparisons for any constant $\varepsilon > 0$.*

Proof: ExternalHeapsort always traverses one path in the heap from root to bottom and does one comparison for each edge followed, i.e., $\lfloor \lg n \rfloor$ or $\lfloor \lg n \rfloor - 1$ many per delete-max operation. By counting how many leaves we have on each level one can show that we need

$$n(\lfloor \lg n \rfloor - 1) + 2(n - 2^{\lfloor \lg n \rfloor}) \pm \mathcal{O}(\log n) \quad \leq \quad n \lg n - 0.913929n \pm \mathcal{O}(\log n)$$

comparisons for the sort-down phase (both in the best and worst case) [4, Eq. 1]. The constant of the given linear term is $1 - \frac{1}{\ln 2} - \lg(2 \ln 2)$, the supremum of the periodic function at the linear term. Using the classical heap construction method adds between $n-1$ and $2n$ comparisons and $1.8813726n$ comparisons on average [6]. We therefore find the following upper bounds for the average and worst case resp. lower bound for the best case of ExternalHeapsort:

$$\begin{aligned} x_{\mathrm{ac}}(n) &= n \lg n + 0.967444n \pm \mathcal{O}(n^{\varepsilon}) \\ x_{\mathrm{wc}}(n) &= n \lg n + 1.086071n \pm \mathcal{O}(n^{\varepsilon}) \\ x_{\mathrm{bc}}(n) &= n \lg n \pm \mathcal{O}(n^{\varepsilon}) \end{aligned}$$

for any $\varepsilon > 0$.

Notice that every deleteMax operation performs comparisons until the element inserted at the top of the heap (replacing the maximum) reaches the bottom of the heap. That means when the heap is already quite empty, some of those comparisons are between two buffer elements and these buffer elements are exchanged according to the outcome of the comparison. Therefore, ExternalHeapsort does *not* preserve the randomness of the buffer elements. Our recurrence, Equation (3), is thus not valid for QuickHeapsort directly.

We can, however, study a hypothetical method X that always uses $x(n) = x_{\mathrm{wc}}(n)$ comparisons on an input of size n, and consider the costs $c(n)$ of QuickXsort for this method. This is clearly an upper bound for the cost of QuickHeapsort since $c(n)$ is a monotonically increasing function in $x(n)$. Similarly, using $x(n) = x_{\mathrm{bc}}(n)$ yields a lower bound. The results then follow by applying Theorem 4.1 and Theorem 4.8. □

We note that our transfer theorems are only applicable to worst resp. best case bounds for ExternalHeapsort, but nevertheless, using the average case $x_{\mathrm{ac}}(n)$ still might give us a better (heuristic) approximation of the actual numbers.

Comparison with previously reported comparison counts. Both [3] and [4] report averaged comparison counts from running time experiments. We compare them in Table 4 against the estimates from our results and previous analyses. We compare both proven upper bound from above and the heuristic estimate using ExternalHeapsort's average case.

While the approximation is not very accurate for $n = 100$ (for all analyses), for larger n, our estimate is correct up to the first three digits, whereas previous upper bounds have almost one order of magnitude larger errors. Our provable upper bound is somewhere in between. Note that we expect even our estimate to be still on the conservative side because we used the supremum of the periodic linear term for ExternalHeapsort.

6. QuickMergesort with base cases

In QuickMergesort, we can improve the number of comparisons even further by sorting small subarrays with yet another algorithm Z. The idea is to use Z only for tiny subproblems,

Instance	observed	estimate	upper bound	CC	DW
Fig. 4 [3], $n = 10^2$, $k = 1$	806	+67	+79	+158	+156
Fig. 4 [3], $n = 10^2$, $k = 3$	714	+98	+110	—	+168
Fig. 4 [3], $n = 10^5$, $k = 1$	1 869 769	−600	+11 263	+90 795	+88 795
Fig. 4 [3], $n = 10^5$, $k = 3$	1 799 240	+9 165	+21 028	—	+79 324
Fig. 4 [3], $n = 10^6$, $k = 1$	21 891 874	+121 748	+240 375	+1 035 695	+1 015 695
Fig. 4 [3], $n = 10^6$, $k = 3$	21 355 988	+49 994	+168 621	—	+751 581
Tab. 2 [4], $n = 10^4$, $k = 1$	152 573	+1 125	+2 311	+10 264	+10 064
Tab. 2 [4], $n = 10^4$, $k = 3$	146 485	+1 136	+2 322	—	+8 152
Tab. 2 [4], $n = 10^6$, $k = 1$	21 975 912	+37 710	+156 337	+951 657	+931 657
Tab. 2 [4], $n = 10^6$, $k = 3$	21 327 478	+78 504	+197 131	—	+780 091

Table 4: Comparison of estimates from this paper where we use the average for ExternalHeapsort (estimate) resp. the worst case for ExternalHeapsort (upper bound), Theorem 6 of [3] (CC) and Theorem 1 of [4] (DW). The numbers give the difference between the estimate and the observed average.

so that it is viable to use methods that require extra space and have otherwise prohibitive cost for other operations like moves. Obvious candidates for Z are Insertionsort and 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 an overall $\mathcal{O}(n^2)$ running time for Z and still obtain only $\mathcal{O}((n/\log n) \cdot \log^2 n) = \mathcal{O}(n \log n)$ overhead in QuickMergesort. We note that for the following result, we only need that the size of the base cases grows with n, but not faster than logarithmic.

We start by bounding the costs of Mergesort with base-case sorter Z. Reinhardt [42] proposes this idea using MergeInsertion for base cases of constant size and essentially states the following result, but does not provide a proof for it. In Section 9.7 we give the proof, which is an easy induction.

Theorem 6.1 (Mergesort with Base Case): *Let Z be a sorting algorithm using $z(n) = n \lg n + (b \pm \varepsilon)n + o(n)$ comparisons on average and other operations taking at most $\mathcal{O}(n^2)$ time. If base cases of size $\omega(1) \cap \mathcal{O}(\log n)$ are sorted with Z, Mergesort uses at most $n \lg n + (b \pm \varepsilon)n + o(n)$ comparisons and $\mathcal{O}(n \log n)$ other instructions on average.*

Mergesort with base cases can thus be very efficient w.r.t. comparisons, but is an external algorithm. By combining it with QuickMergesort, we obtain an internal method with essentially the same comparison cost. Using the same route as in the proof of Corollary 5.1, we obtain the following result.

Corollary 6.2 (QuickMergesort with Base Case): *Let Z be some sorting algorithm with $z(n) = n \lg n + (b \pm \varepsilon)n + o(n)$ comparisons on average and other operations taking at most $\mathcal{O}(n^2)$ time. If base cases of size $\Theta(\log n)$ are sorted with Z, QuickMergesort uses at most $n \lg n + (b \pm \varepsilon)n + o(n)$ comparisons and $\mathcal{O}(n \log n)$ other instructions on average.*

Base cases of size $\Theta(\log n)$ always lead to a constant factor overhead in running time if an algorithm Z with a quadratic number of total operations is used. 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.[9] A modification of our proof above

[9] We could also sort base cases of some slower growing size with Z, e.g., $\Theta(\log \log n)$. This avoids a constant factor overhead, but still gives a non-negligible additional term in $\omega(n) \cap o(n \log n)$.

allows to bound the impact on the number of comparisons, but we are facing a trade-off between comparisons and other operations, so the best threshold for Z depends on the type of data to be sorted and the system on which the algorithms run.

6.1. Insertionsort

We know study the average cost of the natural candidates for Z. We start with InsertionSort, since it is an elementary method and its analysis is used as part of our average-case analysis of MergeInsertion later. Recall that InsertionSort inserts the elements one by one into the already sorted sequence by binary search. For the average number of comparisons we obtain the following result.

Proposition 6.3 (Average Case of Insertionsort): *The sorting algorithm InsertionSort needs* $n \lg n - 2 \ln 2 \cdot n + c(n) \cdot n + \mathcal{O}(\log n)$ *comparisons on average where* $c(n) \in [-0.005, 0.005]$.

The proof of Proposition 6.3 is based on the observation that inserting one element by binary insertion into $k-1$ already sorted elements requires

$$x_{\text{Ins}}(k) = \lceil \lg k \rceil + 1 - \frac{2^{\lceil \lg k \rceil}}{k} \tag{7}$$

comparisons on average (assuming a uniform distribution). Summing this up, we obtain

$$x_{\text{InsSort}}(n) \;=\; n \cdot \lceil \lg n \rceil - 2^{\lceil \lg n \rceil} + n - \ln 2 \cdot (2 + \lg n - \lceil \lg n \rceil) \cdot 2^{\lceil \lg n \rceil} \;+\; \mathcal{O}(\log n)$$

comparisons on average for sorting n elements with InsertionSort. For the full proof, see Section 9.7. In order to obtain an explicit bound on the linear term of $x_{\text{InsSort}}(n)$, we compute $(x_{\text{InsSort}}(n) - n \lg n)/n$ and then replace $\lceil \lg n \rceil - \lg n$ by x. This yields a function

$$x \;\mapsto\; x - 2^x + 1 - \ln 2 \cdot (2 - x) \cdot 2^x,$$

which oscillates between -1.381 and -1.389 for $x \in [0, 1)$; see Figure 7. For $x = 0$, its value is $2 \ln 2 \approx 1.386$.

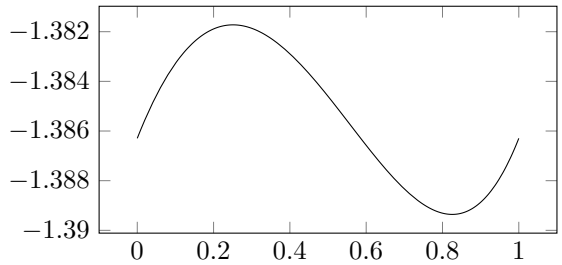


Figure 7: The periodic function in InsertionSort $x \mapsto x - 2^x + 1 - \ln 2 \cdot (2 - x) \cdot 2^x$ for $x = \lg n - \lfloor \lg n \rfloor \in [0, 1)$.

Sorting base cases of logarithmic size in QuickMergesort with InsertionSort, we obtain the next result by Corollary 6.2:

Corollary 6.4 (QuickMergesort with Base Case Insertionsort):
Median-of-$\sqrt{n}$ *QuickMergesort with InsertionSort base cases uses at most* $n \lg n - 1.381n + o(n)$ *comparisons and* $\mathcal{O}(n \log n)$ *other instructions on average.*

6.2. MergeInsertion

MergeInsertion by Ford and Johnson [18] is one of the best sorting algorithms in terms of number of comparisons. Applying it for sorting base cases of QuickMergesort yields even better results than Insertionsort. We give a brief description of the algorithm and analyze its average case for a simplified version. Algorithmically, MergeInsertion $(s_0, \ldots, s_{n-1})$ can be described as follows (an intuitive example for $n = 21$ can be found in [32]):

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}, $b_9, \ldots, b_5, \ldots$; $b_{t_{k-1}-1}, \ldots b_{t_{k-2}}$; $b_{t_k-1}, \ldots$ into the main chain, where $t_k = (2^{k+1} + (-1)^k)/3$ using (at most) k comparisons for the elements $b_{t_k-1}, \ldots, b_{t_{k-1}}$.

While the description is simple, MergeInsertion is not easy to implement efficiently because of the different renamings, the recursion, and the insertion in the sorted list. Our proposed implementation of MergeInsertion is based on a tournament tree representation with weak heaps as in [7, 9]. It uses quadratic time and requires $n \lg n + n$ extra bits (note that in [49], an implementation with running time $n \log^2 n$ was presented).

When inserting some of the b_i with $t_{k-1} \leq i \leq t_k - 1$ in the already sorted chain, we know that at most k comparisons are needed. During an actual execution of the algorithm, it might happen, that only $k - 1$ comparisons are needed (if the insertion tree is balanced at least $k - 1$ comparisons are needed). This decreases the average number of comparisons. Since the analysis is involved, we analyze a simplified variant, where all elements of one insertion block (i. e. elements $b_{t_k-1}, b_{t_{k-1}-1}, \ldots b_{t_{k-1}}$) are always inserted into the same number of elements. Thus, for the elements of the k-th block always k comparisons are used – except for the last block $b_{\lceil n/2 \rceil - 1}, \ldots b_{t_k}$. In our experiments we evaluate the simplified and the original variant.

Theorem 6.5 (Average Case of MergeInsertion): *Simplified MergeInsertion needs* $n \lg n - c(n) \cdot n + \mathcal{O}(\log n)$ *comparisons on average, where* $c(n) > 1.3999$.

The proof of Theorem 6.5 can be found in Section 9.7. The key observation is that, if $3n$ is not (approximately) a power of two, then the elements of the last block $b_{\lceil n/2 \rceil - 1}, \ldots, t_k$ are inserted into less than 2^{k+1} elements, so on average less than $k + 1$ comparisons are needed. These savings can be estimated using Equation (7). Here it is important to notice that, although the different positions for insertion are *not* uniformly distributed, we can use (7) as an upper bound because we know that the more likely positions are further to the left and the less likely positions are further to the right (and we assume the algorithm uses a decision tree with longer paths for the positions on the right).

The proof gives an upper bound of $n \lg n - c(n) \cdot n + \mathcal{O}(\log n)$ comparisons on average for MergeInsertion where

$$c(n) \;\geq\; (3 - \lg 3) - (1 - x + 1 - 2^{1-x}) + (1 - 2^{-x}) \cdot \left(\frac{3}{2^x + 1} - 1 \right) \;>\; 1.3999 \qquad (8)$$

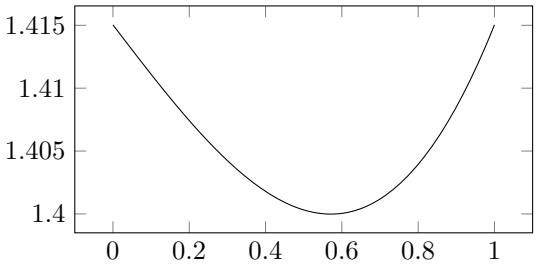


Figure 8: The periodic function in MergeInsertion $x \mapsto (3 - \lg 3) - (2 - x - 2^{1-x}) + (1 - 2^{-x}) \cdot \left(\frac{3}{2^x+1} - 1\right)$ for $x = \lg 3n - \lfloor \lg 3n \rfloor \in [0, 1)$. If n is a power of two, we have $x = \lg 3 - 1 \approx 0.5850$

for $x = \lg 3n - \lfloor \lg 3n \rfloor \in [0, 1)$. This function reaches its minimum in $[0, 1)$ for

$$x \;=\; \lg\left(\ln 8 - 1 + \sqrt{(1 - \ln 8)^2 - 1}\right) \;\approx\; 0.5713.$$

When applying MergeInsertion to sort base cases of size $\mathcal{O}(\log n)$ in QuickMergesort, we obtain the next corollary from Corollary 6.2 and Theorem 6.5.

Corollary 6.6 (QuickMergesort with Base Case MergeInsertion):
Median-of-$\sqrt{n}$ QuickMergesort with MergeInsertion for base cases needs at most $n \lg n - 1.3999n + o(n)$ comparisons and $\mathcal{O}(n \log n)$ other instructions on average.

Instead of growing-size base cases, we can also sort constant-size base cases with MergeInsertion. When the size of the base cases is reasonably small, we can hard-code the MergeInsertion algorithm to get a good practical performance combined with a lower number of comparisons than just QuickMergesort. In our experiments we also test one variant where subarrays up to nine elements are sorted with MergeInsertion.

Remark 6.7 (Best/Worst n for MergeInsertion): *For standard Mergesort, the optimal input sizes are powers of two. Is the same true for MergeInsertion? We know that for the worst case, the best n are (close to) $\frac{1}{3} \cdot 2^k$ for an integer k, which also gives the largest value of $c(\frac{1}{3} \cdot 2^k) \approx 1.4150$ in our bound on the average case from Equation (8), which should give a reasonable approximation.*

If on the other hand n is a power of two, we have $c(2^k) = 1.4$, which is close to – but not exactly! – the minimal value for $c(n)$. Thus, for powers of two the proof of Theorem 6.5 gives almost the worst bounds, so presumably these are among the worst *input sizes for MergeInsertion. This is in line with the empirical results shown in Figure 8.*

Remark 6.8 (Better bounds?): *Can one push the coefficient -1.3999 even further? For the simplified version of MergeInsertion studied here, the empirical numbers from Section 7 seem to suggest that our bound is approximately tight. Moreover, there is only one step in our analysis which is not tight (all others leading to an error term of $\mathcal{O}(\log n)$): to estimate the costs of the binary search, we approximate the probability distribution of where elements are inserted by the uniform distribution, where it actually (slightly) favors smallest indices over larger ones. We conjecture that the difference between the approximation and the real values is a very small linear term meaning that the actual coefficient of the linear term can be still just above or below -1.4.*

As we can see in our experiments in Figure 9, the linear-term coefficient of the non-simplified version of MergeInsertion *seems to be noticeably below* -1.4*. Very recently, a formal proof for this has been given [49], showing an upper bound of* $n\lg n - 1.4005n + \mathcal{O}(\log n)$ *comparisons. This is still quite far from the experimental results, but it breaks the barrier of* -1.4*; (admittedly, this has mostly psychological relevance). The analysis in [49] is based on our Theorem 6.5 and its proof.*

We finally point out that the exact number of comparisons of the algorithm depends on a (rather) small implementation detail: in the binary search, it is not completely specified which is the first element to compare with.

6.3. Combination of (1,2)-Insertion and MergeInsertion

Iwama and Teruyama [28] propose an improvement of Insertionsort, which inserts a (sorted) *pair* of elements in one shot. Their main observation is that standard binary searches are good only if n is close to a power of two, but become more wasteful for other n. Inserting two elements together helps in such cases.

On the other hand, MergeInsertion is much better than $n\lg n - 1.3999n + o(n)$ when n is close to $\frac{4}{3}\cdot 2^k$ for an integer k (see Figure 8). By first sorting an optimal-size subarray with MergeInsertion and then using their new (1,2)-Insertionsort for inserting the remaining elements, Iwama and Teruyama obtain a portfolio algorithm "Combination", which needs $n\lg n - c(n)\cdot n + \mathcal{O}(\log n)$ comparisons on average, where $c(n) \geq 1.4106$.

Remark 6.9 (A slightly improved bound): *Using the analysis in [49], we can improve this bound slightly: [28] uses the worst-case bound for* MergeInsertion *for* n *close to* $\frac{4}{3}\cdot 2^k$ *for an integer* k *(which in this case equals the average-case bound in Theorem 6.5). By [49],* MergeInsertion *needs at most* $n\log n + (3 - \lg 3 - \frac{1}{764})n + \mathcal{O}(\log^2 n)$ *on average if* n *is just above* $\frac{4}{3}\cdot 2^k$ *for an integer* k*. This bound is by* $\frac{1}{764}n$ *better than the one used in [28]. Since in [28] an array of size between* $n/2$ *and* n *is sorted with* MergeInsertion*, the total improvement compared to their analysis is (a staggering)* $\frac{1}{1528}n \approx 0.00065n$*.*

The running time of the portfolio algorithm is at most $\mathcal{O}(n^2)$ (in a naive implementation), so that we can also use this algorithm as a base case sorter Z.

Corollary 6.10 (QuickMergesort with Base Case Combination):
Median-of-$\sqrt{n}$ QuickMergesort *with Iwama and Teruyama's* MergeInsertion*/(1,2)-*Insertionsort *method for base cases needs at most* $n\lg n - 1.4112n + o(n)$ *comparisons and* $\mathcal{O}(n\log n)$ *other instructions on average.*

In contrast to the original method of Iwama and Teruyama, QuickMergesort with their method for base cases is an internal sorting method with $\mathcal{O}(n\log n)$ running time.

With this present champion in terms of the average-case number of comparisons, we close our investigation of asymptotically optimal sorting methods. In the following, we will take a look at their actual running times on realistic input sizes.

7. Experiments

In this section, we report on studies with efficient implementations of our sorting methods. We conducted two sets of experiments: First, we compare our asymptotic approximations

with experimental averages for finite n to assess the influence of lower order terms for realistic input sizes. Second, we conduct an extensive running-time study to compare QuickMergesort with other sorting methods from the literature.

Experimental setup. We ran thorough experiments with implementations in C++ with different kinds of input permutations. The experiments are run on an Intel Core i5-2500K CPU (3.30GHz, 4 cores, 32KB L1 instruction and data cache, 256KB L2 cache per core and 6MB L3 shared cache) with 16GB RAM and operating system Ubuntu Linux 64bit version 14.04.4. We used GNU's `g++` (4.8.4); optimized with flags `-O3 -march=native`. For time measurements, we used `std::chrono::high_resolution_clock`, for generating random inputs, the Mersenne Twister pseudo-random generator `std::mt19937`. All experiments, except those in Figure 16, were conducted with random permutations of 32-bit integers.

Implementation details. The code of our implementation of QuickMergesort as well as the other algorithms and our running time experiments is available at `https://github.com/weissan/QuickXsort`. In our implementation of QuickMergesort, we use the merging procedure from [16], which avoids avoids branches based on comparisons altogether. We use the partitioner from the GCC implementation of `std::sort`. For all running time experiments in QuickMergesort we sort base cases up to 42 elements with StraightInsertionsort. When counting the number of comparisons StraightInsertionsort is deactivated and Mergesort is used down to arrays of size two. We also test one variant where base cases up to nine elements are sorted by a hard-coded MergeInsertion variant. The median-of-$\sqrt{n}$ variants are always implemented with $\alpha = 1/2$ (notice that different values for α make very little difference as the pivot is almost always very close to the median). Moreover, they switch to pseudomedian-of-25 (resp. pseudomedian-of-9, resp. median-of-3) pivot selection for n below 20 000 (resp. 800, resp. 100).

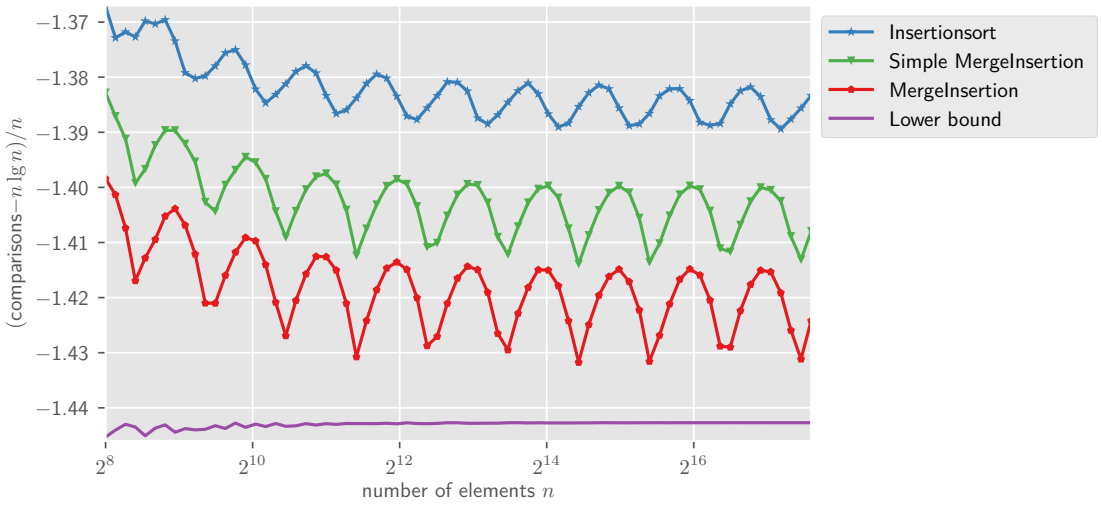


Figure 9: Coefficient of the linear term of the number of comparisons of MergeInsertion, its simplified variant and Insertionsort (for the number of comparisons $n \lg n + bn$ the value of b is displayed).

7.1. Comparison counts

The first set of experiments uses our efficient implementations to obtain empirical estimates for the number of comparisons used.

Base case sorters. First, we compare the different algorithms we use as base cases: MergeInsertion, its simplified variant, and InsertionSort[10]. The results can be seen in Figure 9. It shows that both InsertionSort and the simplified version of MergeInsertion match the theoretical estimates very well. Moreover, MergeInsertion achieves results for the coefficient of the linear term in the range of $[-1.43, -1.41]$ (for some values of n are even smaller than -1.43). We can see very well the oscillating linear term of InsertionSort (as predicted in Proposition 6.3) and MergeInsertion ((8) for the simple variant).

Number of comparisons of QuickXsort variants. We counted the number of comparisons of different QuickMergesort variants. We also include an implementation of top-down Mergesort which agrees in all relevant details with the Mergesort part of our QuickMergesort implementation. The results can be seen in Figure 10, Figure 11, and Table 5. Here each data point is the average of 400 measurements (with deterministically chosen seeds for the random generator) and for each measurement at least 128MB of data were sorted – so the values for $n \le 2^{24}$ are actually averages of more than 400 runs. From the actual number of comparisons we subtract $n \lg n$ and then divide by n. Thus, we get an approximation of the linear term b in the number of comparisons $n \lg n + bn + o(n)$.

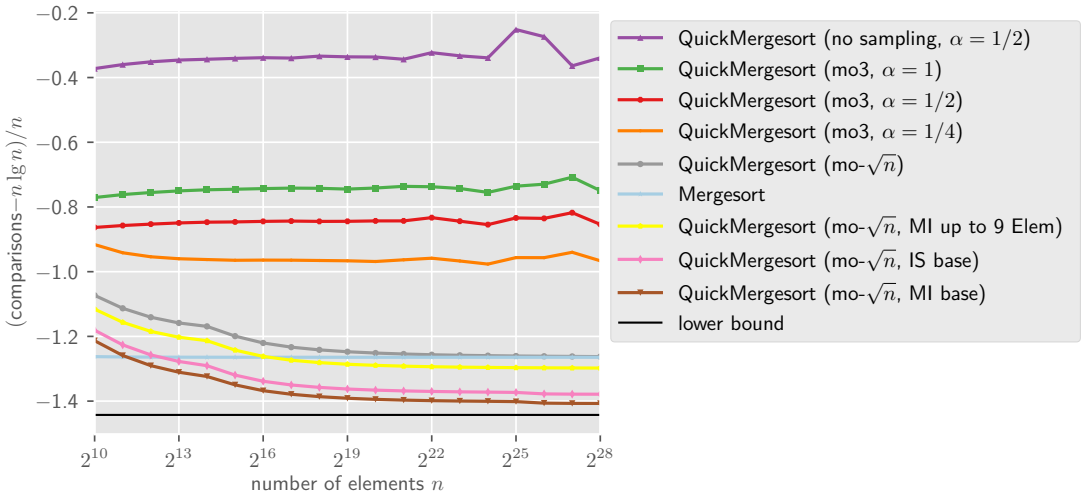


Figure 10: Coefficient of the linear term of the number of comparisons ((comparisons − $n \lg n)/n$). Median-of-$\sqrt{n}$ QuickMergesort is always with $\alpha = 1/2$.

In Table 5, we also show the theoretical values for b. We can see that the actual number of comparisons matches the theoretical estimate very well. In particular, we experimentally confirm that the sublinear terms in our estimates are negligible for the total number of

[10] For these experiments we use a different experimental setup: depending on the size of the arrays the displayed numbers are averages over 10 – 10 000 runs.

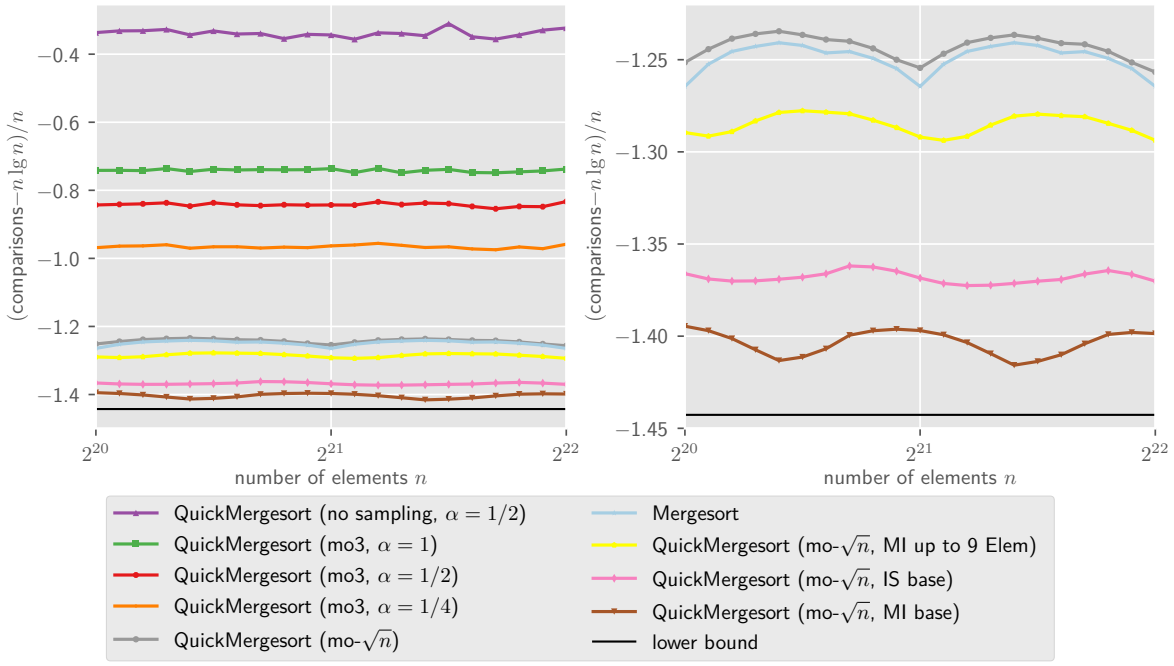


Figure 11: Detailed view of the coefficient of the linear term of the number of comparisons ((comparisons $- n \lg n)/n$) for $n \in [2^{20}..2^{22}]$. Enlarged view of bottom part of the plot.

Algorithm	absolute $n = 2^{22}$	absolute $n = 2^{28}$	empirical b $n = 2^{22}$	empirical b $n = 2^{28}$	theoretical b $(n \to \infty)$
$k = 1$, $\alpha = 1/2$	90 919 646	7 425 155 999	-0.323	-0.339 ± 0.037	-0.3407 ± 0.0119
mo3, $\alpha = 1$	89 181 407	7 314 997 953	-0.737	-0.750 ± 0.017	-0.7456 ± 0.0119
mo3, $\alpha = 1/2$	88 780 825	7 287 011 306	-0.833	-0.854 ± 0.016	-0.8476 ± 0.0119
mo3, $\alpha = 1/4$	88 254 970	7 256 806 284	-0.958	-0.966 ± 0.013	-0.9560 ± 0.0119
mo-$\sqrt{n}$	87 003 696	7 177 302 635	-1.257	$-1.262 \pm 4.1 \cdot 10^{-5}$	-1.2526 ± 0.0119
mo-$\sqrt{n}$, IS	86 527 879	7 146 103 511	-1.370	$-1.379 \pm 5.3 \cdot 10^{-6}$	-1.3863 ± 0.005
mo-$\sqrt{n}$, MI	86 408 550	7 138 442 729	-1.399	$-1.407 \pm 4.6 \cdot 10^{-6}$	≤ -1.3999

Table 5: Absolute numbers of comparisons and linear term ($b = (\text{comparisons} - n \lg n)/n$) of QuickMergesort variants for $n = 2^{22}$ and $n = 2^{28}$. We also show the asymptotic regime for b due to Table 2, Corollary 5.1, Corollary 6.4 and Corollary 6.6. The $\pm$-terms for the theoretical b represent our lower and upper bound. For the experimental b, the $\pm$-terms are the standard error of the mean (standard deviation of the measurements divided by the square-root of the number of measurements).

comparisons (at least for larger values of n). The experimental number of comparisons of QuickMergesort with MergeInsertion base cases is better than the theoretical estimate because we analyzed only the simplified variant of MergeInsertion.

For constant-size samples we see that even with 400 measurements the plots still look a bit bumpy, particularly for the largest inputs. Also the difference to the theoretical values is larger for $n = 2^{28}$ than for $n = 2^{22}$ in Table 5 – presumably because the average is taken over more measurements (see setup above). We note however that the deviations are still within the range we could expect from the values of the standard deviation (both established theoretically and experimentally – Table 6): for 400 runs, we obtain a standard deviation of approximately $0.65n/\sqrt{400} = 0.0325$. Even the largest "bump" is thus only slightly over two standard deviations.

In Figure 10, we see that median-of-$\sqrt{n}$ QuickMergesort uses almost the same number of comparisons as Mergesort for larger values of n. This shows that the error terms in Theorem 4.1 are indeed negligible for practical issues. The difference between experimental and theoretical values for median-of-$\sqrt{n}$ QuickMergesort is due to the fact that the bound holds for arbitrary n, but the average costs of Mergesort are actually minimal for powers of two.

In Figure 11 we see experimental results for problem sizes which are not powers of two. The periodic coefficients of the linear terms of Mergesort, Insertionsort and MergeInsertion can be observed – even though these algorithms are only applied in QuickXsort (and for the latter two even only as base cases in QuickMergesort). The version with constant size 9 base cases seems to combine periodic terms of Mergesort and MergeInsertion. For the median-of-three version, no significant periodic patterns are visible. We conjecture that the higher variability of subproblem sizes makes the periodic behavior disappear in the noise.

Standard deviation. Since not only the average running time (or number of comparisons) is of interest, but also how far an algorithm deviates from the mean running time, we also measure the standard deviation of the running time and number of comparisons of QuickMergesort. For comparison we also measured two variants of Quicksort (which has a standard deviation similar to QuickMergesort): the GCC implementation of the C++ standard sorting function `std::sort` (GCC version 4.8.4) and a modified version where the pivot is excluded from recursive calls and otherwise agreeing with `std::sort`. We call the latter variant simply Quicksort as it is the more natural way to implement Quicksort. Moreover, from both variants we remove the final StraightInsertionsort and instead use Quicksort down to size three arrays.

In order to get a meaningful estimate of the standard deviation we need many more measurements than for the mean values. Therefore, we ran each algorithm 40 000 times (for every input size) and compute the standard deviation of these. Moreover, for every measurement only one array of the respective size is sorted. For each measurement we use a pseudo-random seed (generated with `std::random_device`). The results can be seen in Table 6 and Figure 12.

In Table 6 we also compare the experiments to the theoretical values from Table 3. Although these theoretical values are only approximate values (because Theorem 4.10 is not applicable to QuickMergesort), they match the experimental values very well. This

shows that increase in variance due to the periodic functions in the linear term of the average number of comparisons is negligible.

Furthermore, we see that choosing the pivot as median-of-3 halves the standard deviation compared to no sampling. This gives another good reason to always use at least the median-of-3 version. While the difference between $\alpha = 1$ and $\alpha = 1/2$ is rather small, $\alpha = 1/4$ gives a considerably smaller standard deviation. Moreover, selecting the pivot as median-of-$\sqrt{n}$ is far better than median-of-3 (for $N = 2^{20}$ the standard deviation is only around one hundredth).

All algorithms have a rather large standard deviation of running times for small inputs (which is no surprise because measurement imprecisions etc. play a bigger role here). Therefore, we only show the results for $n \geq 2^{18}$. Also, while QuickMergesort with $\alpha = 1/4$ has the smallest standard deviation for the number of comparisons (except median-of-$\sqrt{n}$) it has the *largest* standard deviation for the running time for large n. This is probably due to the fact that (our implementation of) Reinhardt's merging method is not as efficient as the standard merging method. Although median-of-$\sqrt{n}$ QuickMergesort has the smallest standard deviation of running times, the difference is by far not as large as for the number of comparisons. This indicates that other factors than the number of comparisons are more relevant for standard deviation of running times.

We also see that including the pivot into recursive calls in Quicksort should be avoided. It increases the standard deviation of both the number of comparisons and the running time, and also for the average number of comparisons (which we do not show here).

Algorithm	empirical $n = 2^{16}$	empirical $n = 2^{20}$	theoretical
Quicksort (mo3)	0.3385	0.3389	0.3390
Quicksort (`std::sort`, no SIS)	0.3662	0.3642	–
QuickMergesort (no sampling, $\alpha = 1/2$)	0.6543	0.6540	0.6543
QuickMergesort (mo3, $\alpha = 1$)	0.3353	0.3355	0.3345
QuickMergesort (mo3, $\alpha = 1/2$)	0.3285	0.3257	0.3268
QuickMergesort (mo3, $\alpha = 1/4$)	0.2643	0.2656	0.2698
QuickMergesort (mo-$\sqrt{n}$)	0.0172	0.00365	–

Table 6: Experimental and theoretical values for the standard deviation divided by n of QuickMergesort and Quicksort (theoretical value for Quicksort by [24, p. 331] and for QuickMergesort by Table 3). Recall that for QuickMergesort, the theoretical value is only a heuristic approximation as Theorem 4.10 is not formally applicable with periodic linear terms. In light of this, the high precision of all these predictions is remarkable.

7.2. Running time experiments

We compare QuickMergesort and QuickHeapsort with Mergesort (our own implementations; Mergesort code is identical to the version used in QuickMergesort, but with using an external buffer of length $n/2$), Wikisort [39] (in-place stable Mergesort based on [31]), `std::stable_sort` (a bottom-up Mergesort, from GCC version 4.8.4), InSituMergesort [16] (which is essentially QuickMergesort where always the median is used as pivot), and `std::sort` (median-of-three Introsort, from GCC version 4.8.4).

All time measurements were repeated with the same 100 deterministically chosen seeds – the displayed numbers are the averages of these 100 runs. Moreover, for each time measure-

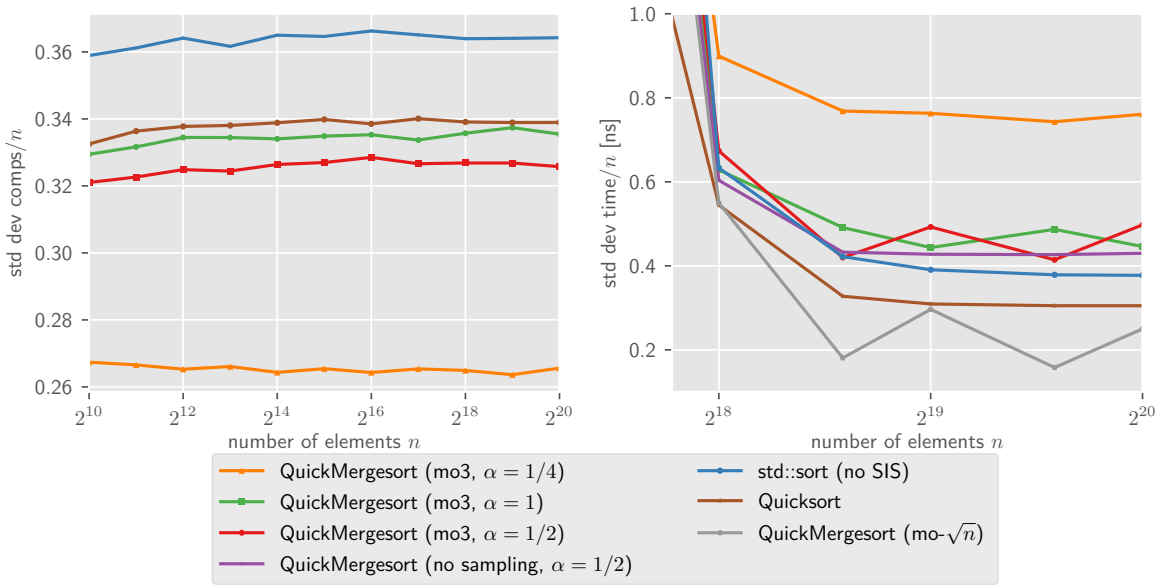


Figure 12: Standard deviation of the number of comparisons (left) and the running times (right). For the number of comparisons, median-of-$\sqrt{n}$ QuickMergesort and QuickMergesort without pivot sampling are out of range.

ment, at least 128MB of data were sorted – if the array size is smaller, then for this time measurement several arrays have been sorted and the total elapsed time measured. The results for sorting 32-bit integers are displayed in Figure 14, Figure 13, and Figure 15, which all contain the results of the same set of experiments – we use three different figures because of the large number of algorithms and different scales on the y-axes.

Figure 13 compares different QuickMergesort variants to Mergesort and `std::sort`. In particular, we compare median-of-3 QuickMergesort with different values of α. While for the number of comparisons a smaller α was beneficial, it turns out that for the running time the opposite is the case: the variant with $\alpha = 1$ is the fastest. Notice, however, that the difference is smaller than 1 %. The reason is presumably that partitioning is faster than merging: for large α the problem sizes sorted by Mergesort are reduced and more "sorting work" is done by the partitioning. As we could expect our Mergesort implementation is faster than all QuickMergesort variants – because it can do simply moves instead of swaps. Except for small n, `std::sort` beats QuickMergesort. However, notice that for $n = 2^{28}$ the difference between `std::sort` and QuickMergesort without sampling is only approximately 5 %, thus, can most likely be bridged with additional tuning efforts, (e.g., block partitioning [12]).

In Figure 14 we compare the QuickMergesort variants with base cases with Quick-Heapsort and `std::sort`. While QuickHeapsort has still an acceptable speed for small n, it becomes very slow when n grows. This is presumably due to the poor locality of memory accesses in Heapsort. The variants of QuickMergesort with growing size base cases are always quite slow. This could be improved by sorting smaller base cases with the respective algorithm – but this opposes our other aim to minimize the number of comparisons. Only the version with constant size MergeInsertion base cases reaches a speed comparable to `std::sort` (as it can be seen also in Figure 13).

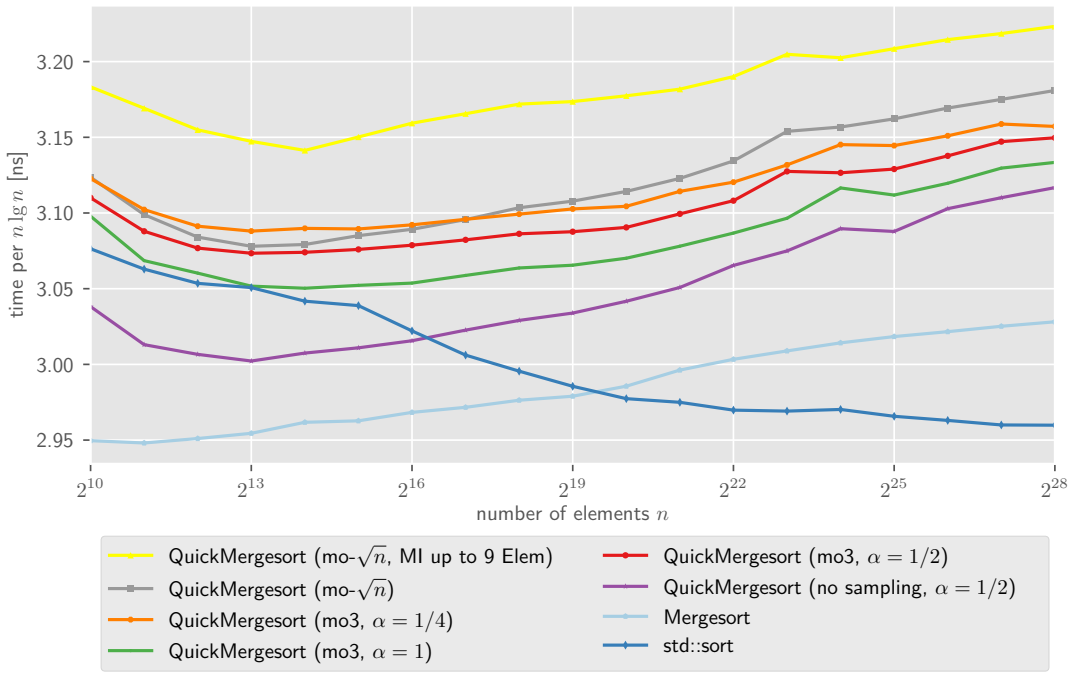


Figure 13: Running times of QUICKMERGESORT variants, MERGESORT, and `std::sort` when sorting random permutations of integers.

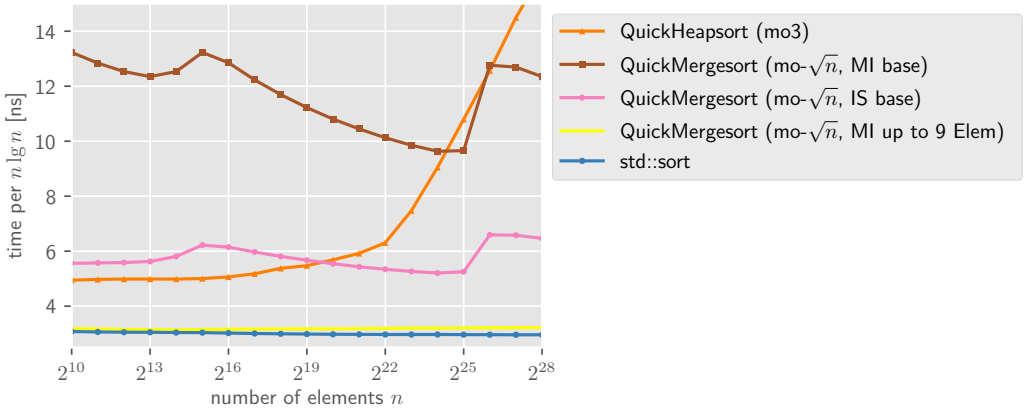


Figure 14: Running times of QUICKMERGESORT variants with base cases and QUICKHEAPSORT when sorting random permutations of integers.

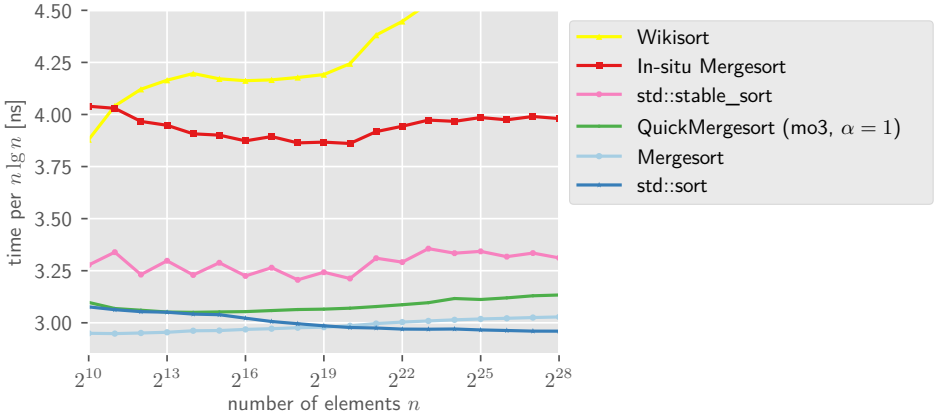


Figure 15: Running times when sorting random permutations of integers.

Figure 15 shows median-of-3 QuickMergesort together with the other algorithms listed above. As we see, QuickMergesort beats the other in-place Mergesort variants InSituMergesort and Wikisort by a fair margin. However, be aware that QuickMergesort (as well as InSituMergesort) neither provides a guarantee for the worst case nor is it a stable sorting algorithm.

Other data types. While all the previous running time measurements were for sorting 32-bit integers, in Figure 16 we also tested two other data types: (1) 32-bit integers with a special comparison function which before every comparison computes the logarithm of the operands, and (2) pointers to records of 40 bytes which are compared by the first 4 bytes. Thus in both cases, comparisons are considerably more expensive than for standard integers. Each record is allocated on the heap with `new` – since we do this in increasing order and only shuffle the pointers, we expect them to reside memory in close-to-sorted order.

For both data types, QuickMergesort with constant size MergeInsertion base cases is the fastest (except when sorting pointers for very large n). This is plausible since it combines the best of two worlds: on one hand, it has an almost minimal number of comparisons, on the other hand, it does not induce the additional overhead for growing size base cases. Moreover, the bad behavior of the other QuickMergesort variants ("without" base cases) is probably because we sort base cases up to 42 elements with StraightInsertionsort – incurring many more comparisons (which we did not count in Section 7.1).

8. Conclusion

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 to convert an external sorting algorithm into an internal one introducing only a lower order term of additional comparisons on average.

We examined QuickHeapsort 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 efficient in terms of running time, even when the comparisons are fast.

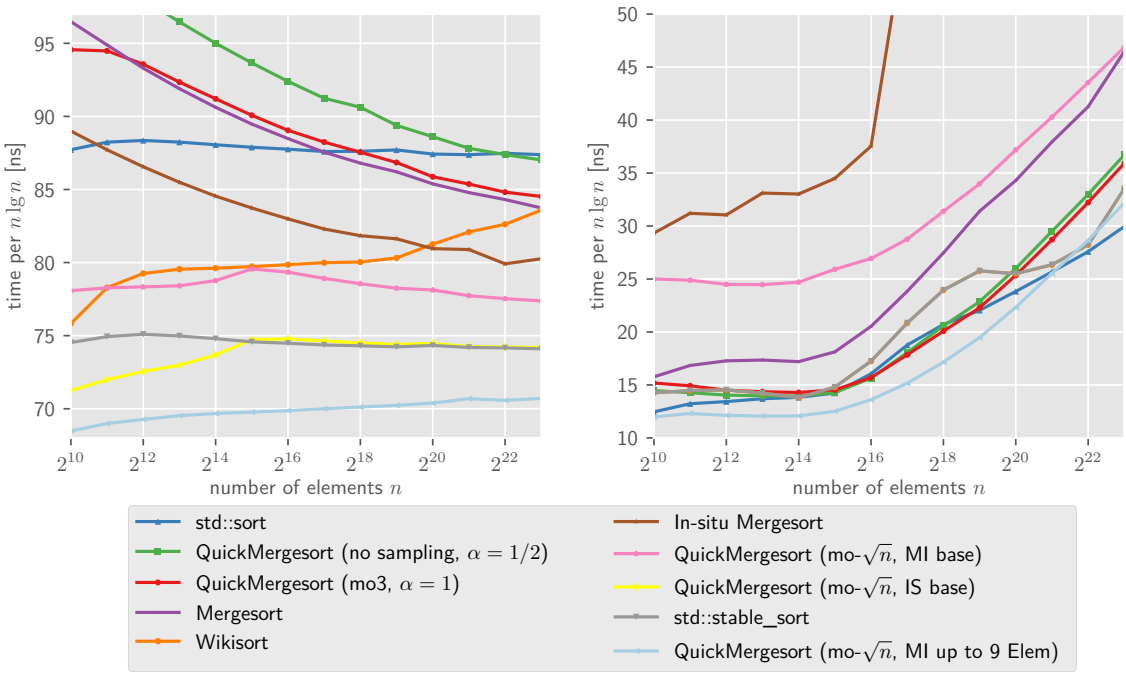


Figure 16: Running times when sorting random permutations of ints with special comparison function (computing the log in every comparison – left) and pointers to Records (right). WIKISORT did not run for sorting pointers and QUICKMERGESORT with INSERTIONSORT base cases is out of range.

Using MERGEINSERTION to sort base cases of growing size for QUICKMERGESORT, we derive an upper bound of $n \lg n - 1.3999n + o(n)$ comparisons for the average case. Using the recent algorithm by Iwama and Teruyama [28], this can be improved even further to $n \lg n - 1.4112n + o(n)$, without causing the overall operations to become more than $\mathcal{O}(n \log n)$. Thus, the average of our best implementation has a proven gap of at most $0.0321n + o(n)$ comparisons to the lower bound. Of course, there is still room in closing the gap to the lower bound of $n \lg n - 1.44n + \mathcal{O}(\log n)$ comparisons.

This illustrates one underlying strength of the framework architecture of QUICKXSORT: by applying the transfer results as shown in this paper QUICKXSORT directly participates in advances to the performance of algorithm X. Moreover, our experimental results suggest that the bound of $n \lg n - 1.43n + \mathcal{O}(\log n)$ element comparisons may be beaten at least for some values of n. This very close gap between the lower and upper bound manifested in the second order (linear) term makes the sorting problem a fascinating topic and mainstay for the analysis of algorithms in general.

We were also interested in the practical performance of QUICKXSORT and study variants with smaller sampling sizes for the pivot in great detail. Besides average-cases analyses, variances were analyzed. The established close mapping of the theoretical results with the empirical findings should be taken as a convincing arguments for the preciseness of the mathematical derivations.

Open questions. Below, we list some possibilities for extensions of this work.

- By Theorem 4.1 for the average number of comparisons sample sizes of $\Theta(\sqrt{n})$ are optimal among all polynomial size samples. However, it remains open whether $\Theta(\sqrt{n})$ sample sizes are also optimal among *all* (also non-polynomial) sample sizes.
- In all theorems, we only use Θ (or $\mathcal{O}$) notation for sublinear terms and only give upper and lower bounds for the periodic linear terms. Exact formulas for the average number of comparisons of QuickXsort are still open and also would be a tool to find the exact optimal sample sizes.
- In this work, the focus was on expected behavior. Nevertheless, in practice often also guarantees for the worst case are desired. In Theorem 4.7, we did a first step towards such guarantees. Moreover, in [14], we examined the same approach in more detail. Still there are many possibilities for good worst-case guarantees to investigate.
- In Theorem 4.10, we needed the technical conjecture that the variance is in $\mathcal{O}(n^2)$ since we only could show it for special values of k and α. Hence, it remains to find a general proof (or disproof) that the variance is always in $\mathcal{O}(n^2)$ for constant size samples. This issue becomes even more interesting when fluctuations in the expected costs of X are taken into account.
- What is the order of growth of the variance of QuickXsort for growing size samples for pivot selection?
- We only analyzed the simplified variant of MergeInsertion. The average number of comparisons of the original variant still is an open problem and seems rather difficult to attack. Nevertheless, better bounds than just the simplified version should be within reach.
- Further future research avenues are to improve the empirical behavior for large-scale inputs and to study options for parallelization.

9. Full Proofs

In this section, we give the full details of the proofs skipped in the main text for readability.

9.1. Preliminaries

We begin by collecting preliminary results about the involved probability distributions.

9.1.1. Beta distribution

For $\lambda, \rho \in \mathbb{R}_{>0}$, a random variable X has the beta distribution $X \stackrel{\mathcal{D}}{=} \mathrm{Beta}(\lambda, \rho)$ if X admits the density $f_X(z) = z^{\lambda-1}(1-z)^{\rho-1}/\mathrm{B}(\lambda,\rho)$ where $\mathrm{B}(\lambda,\rho) = \int_0^1 z^{\lambda-1}(1-z)^{\rho-1}\,dz$ is the beta function. It is a standard fact that for $\lambda, \rho \in \mathbb{N}_{\geq 1}$ we have

$$\mathrm{B}(\lambda,\rho) \;=\; \frac{(\lambda-1)!(\rho-1)!}{(\lambda+\rho-1)!}; \tag{9}$$

a generalization of this identity using the gamma function holds for any $\lambda, \rho > 0$ [5, Eq. (5.12.1)].

Let us denote by h the function $h : [0,1] \to \mathbb{R}_{\geq}0$ with $h(x) = -x \lg x$. We have for a beta-distributed random variable $X \stackrel{\mathcal{D}}{=} \mathrm{Beta}(\lambda, \rho)$ for $\lambda, \rho \in \mathbb{N}_{\geq 1}$ that

$$\mathbb{E}[h(X)] \;=\; \mathrm{B}(\lambda,\rho)(H_{\lambda+\rho} - H_\lambda). \tag{10}$$

This follows directly from a well-known closed form a "logarithmic beta integral" (see, e.g., [51, Eq. (2.30)]).

$$\int_0^1 \ln(z) \cdot z^{\lambda-1}(1-z)^{\rho-1}\,dz \;=\; \mathrm{B}(\lambda,\rho)(H_{\lambda-1} - H_{\lambda+\rho-1})$$

We will make use of the following elementary properties of h later (towards applying Lemma 9.3).

Lemma 9.1 (Elementary Properties of h): *Let $h : [0,1] \to \mathbb{R}_{\geq 0}$ with $h(x) = -x \lg(x)$.*

(a) *h is bounded by $0 \leq h(x) \leq \frac{\lg e}{e} \leq 0.54$ for $x \in [0,1]$.*

(b) *$g(x) := -x \ln x = \ln(2) h(x)$ is Hölder-continuous in $[0,1]$ for any exponent $\eta \in (0,1)$, i.e., there is a constant $C = C_\eta$ such that $|g(y) - g(x)| \leq C_\eta |y-x|^\eta$ for all $x, y \in [0,1]$. A possible choice for C_η is given by*

$$C_\eta \;=\; \left(\int_0^1 \left|\ln(t) + 1\right|^{\frac{1}{1-\eta}}\right)^{1-\eta} \tag{11}$$

For example, $\eta = 0.99$ yields $C_\eta \approx 37.61$. □

A detailed proof for the second claim appears in [51, Lemma 2.13]. Hence, h is sufficiently smooth to be used in Lemma 9.3.

9.1.2. Beta-binomial distribution

Moreover, we use the *beta-binomial distribution,* which is a conditional binomial distribution with the success probability being a beta-distributed random variable. If $X \stackrel{\mathcal{D}}{=} \mathrm{BetaBin}(n, \lambda, \rho)$ then

$$\mathbb{P}[X = i] \;=\; \binom{n}{i} \frac{\mathrm{B}(\lambda + i, \rho + (n-i))}{\mathrm{B}(\lambda, \rho)}.$$

Beta-binomial distributions are precisely the distribution of subproblem sizes after partitioning in Quicksort. We detail this in Section 9.2.1.

A property that we repeatedly use here is a local limit law showing that the normalized beta-binomial distribution converges to the beta distribution. Using Chernoff bounds after conditioning on the beta distributed success probability shows that $\mathrm{BetaBin}(n, \lambda, \rho)/n$ converges to $\mathrm{Beta}(\lambda, \rho)$ (in a specific sense); but we obtain stronger error bounds for fixed λ and ρ by directly comparing the probability density functions (PDFs). This yields the following result; (a detailed proof appears in [51, Lemma 2.38]).

Lemma 9.2 (Local Limit Law for Beta-Binomial, [51]): *Let $(I^{(n)})_{n\in\mathbb{N}_{\geq 1}}$ be a family of random variables with beta-binomial distribution, $I^{(n)} \stackrel{\mathcal{D}}{=} \mathrm{BetaBin}(n,\lambda,\rho)$ where $\lambda,\rho \in \{1\}\cup\mathbb{R}_{\geq 2}$, and let $f_B(z) = z^{\lambda-1}(1-z)^{\rho-1}/\mathrm{B}(\lambda,\rho)$ be the density of the $\mathrm{Beta}(\lambda,\rho)$ distribution. Then we have uniformly in $z\in(0,1)$ that*

$$n\cdot\mathbb{P}\big[I = \lfloor z(n+1)\rfloor\big] \;=\; f_B(z) \,\pm\, \mathcal{O}(n^{-1}), \qquad (n\to\infty).$$

That is, $I^{(n)}/n$ converges to $\mathrm{Beta}(\lambda,\rho)$ in distribution, and the probability weights converge uniformly to the limiting density at rate $\mathcal{O}(n^{-1})$.

9.1.3. Transformed Chernoff bound

The standard Chernoff bound for binomial random variables is given in Appendix B. Based on it, we can bound expectations of the form $\mathbb{E}[f(\frac{X}{n})]$, by $f(p)$ plus a small error term if f is "sufficiently smooth". Hölder-continuous (recapitulated in Appendix B) is an example for such a criterion.

Lemma 9.3 (Expectation via Chernoff): *Let $p\in(0,1)$ and $X \stackrel{\mathcal{D}}{=} \mathrm{Bin}(n,p)$, and let $f:[0,1]\to\mathbb{R}$ be a function that is bounded by $|f(x)|\leq A$ and Hölder-continuous with exponent $\eta\in(0,1]$ and constant C. Then it holds that*

$$\mathbb{E}\left[f\left(\frac{X}{n}\right)\right] \;=\; f(p) \,\pm\, \rho,$$

where we have for any $\delta\geq 0$ that

$$\rho \;\leq\; \frac{C}{\ln 2}\cdot\delta^{\eta}\big(1-2e^{-2\delta^2 n}\big) \,+\, 4Ae^{-2\delta^2 n}$$

For any fixed $\varepsilon > \frac{1-\eta}{2}$, we obtain $\rho = o(n^{-1/2+\varepsilon})$ as $n\to\infty$ for a suitable choice of δ.

A similar result appears in [51, Lemma 2.36] and [53, Lemma 2.7], but our application in this paper requires a slight generalization.

Proof: By the Chernoff bound we have

$$\mathbb{P}\left[\left|\frac{X}{n} - p\right| \geq \delta\right] \;\leq\; 2u\exp(-2\delta^2 n). \tag{12}$$

To use this on $\mathbb{E}[|f(\frac{X}{n}) - f(p)|]$, we divide the domain $[0,1]$ of $\frac{X}{n}$ into the region of values with distance at most δ from p, and all others. This yields

$$\begin{aligned}\mathbb{E}\left[\left|f\left(\frac{X}{n}\right) - f(p)\right|\right] &\underset{(12)}{\leq} \sup_{\xi:|\xi|<\delta}|f(p+\xi)-f(p)|\cdot\left(1-2e^{-2\delta^2 n}\right)\\ &\qquad + \sup_x|f(x)-f(p)|\cdot 2e^{-2\delta^2 n}\\ &\underset{\text{Lemma 9.1}}{\leq} C\cdot\delta^{\eta}\cdot\left(1-2e^{-2\delta^2 n}\right) \,+\, 2A\cdot 2e^{-2\delta^2 n}.\end{aligned}$$

This proves the first part of the claim.

For the second part, we assume $\varepsilon > \frac{1-\eta}{2}$ is given, so we can write $\eta = 1 - 2\varepsilon + 4\beta$ for a constant $\beta > 0$, and $\eta = (1-2\varepsilon)/(1-2\beta')$ for another constant $\beta' > 0$. We may further assume $\varepsilon < \frac{1}{2}$; for larger values the claim is vacuous. We then choose $\delta = n^c$ with $c = (-\frac{1}{2} - \frac{1/2-\varepsilon}{\eta})/2 = -\frac{1}{4} - \frac{1-2\varepsilon}{4\eta}$. For large n we thus have

$$\begin{aligned}
\rho \cdot n^{1/2-\varepsilon} &\leq C\delta^\eta n^{1/2-\varepsilon}(1 - 2\exp(-2\delta^2 n)) \;+\; 4An^{1/2-\varepsilon}\exp(-2\delta^2 n) \\
&= \underbrace{Cn^{-\beta}}_{\to 0} \cdot (1 - \underbrace{2\exp(-2n^{\beta'})}_{\to 0}) \;+\; 4A \underbrace{\exp\Big(-2n^{\beta'} + (\tfrac{1}{2}-\varepsilon)\ln(n)\Big)}_{\to 0} \\
&\to 0
\end{aligned}$$

for $n \to \infty$, which implies the claim. □

9.2. The QuickXsort recurrence

We here give the recursive description of the expected costs $c(n)$ of QuickXsort. Recall that QuickXsort tries to sort the largest segment with X for which the other segment gives sufficient buffer space. We first consider the case $\alpha = 1$, in which this largest segment is always the smaller of the two segments created.

Case $\alpha = 1$. Let us consider the recurrence for $c(n)$ (which holds for both constant and growing size $k = k(n)$). We distinguish two cases: first, let $\alpha = 1$. We obtain the recurrence

$$\begin{aligned}
c(n) &= x(n) \;\geq\; 0, \qquad (\text{for } n \leq w) \\
c(n) &= \underbrace{n - k(n)}_{\text{partitioning}} + \underbrace{s\big(k(n)\big)}_{\text{pivot sampling}} + \underbrace{\mathbb{E}\big[[\![J_1 > J_2]\!](x(J_1) + c(J_2))\big] + \mathbb{E}\big[[\![J_1 \leq J_2]\!](x(J_2) + c(J_1))\big]}_{\text{calls to X and QuickXsort (recurse)}} \qquad (\text{for } n > w) \\
&= \sum_{r=1}^{2} \mathbb{E}[A_r(J_r)c(J_r)] + t(n)
\end{aligned}$$

where A_1 (resp. A_2) is the indicator random variable for the event "left (resp. right) segment sorted recursively",

$$A_1(J_1, J_2) \;=\; [\![J_1 \leq J_2]\!], \qquad A_2(J_1, J_2) \;=\; [\![J_2 < J_1]\!],$$

and $t(n)$ is the "toll function"

$$t(n) \;=\; n - k + s(k) \;+\; \mathbb{E}[A_2(J_1, J_2)x(J_1)] \;+\; \mathbb{E}[A_1(J_1, J_2)x(J_2)].$$

The expectation here is taken over the choice for the random pivot, i.e., over the segment sizes J_1 resp. J_2. Note that we use both J_1 and J_2 to express the conditions in a convenient form, but actually either one is fully determined by the other via $J_1 + J_2 = n - 1$. Note how A_1 and A_2 change roles in recursive calls and toll functions, since we always sort one segment recursively and the other segment by X.

General α. For $\alpha < 1$, we obtain two cases: When the split induced by the pivot is "uneven" – namely when $\min\{J_1, J_2\} < \alpha \max\{J_1, J_2\}$, i.e., $\max\{J_1, J_2\} > \frac{n-1}{1+\alpha}$ – the smaller segment is not large enough to be used as buffer. Then we can only assign the large segment

as a buffer and run X on the *smaller* segment. If however the split is "about even", i.e., both segments are $\leq \frac{n-1}{1+\alpha}$ we can sort the *larger* of the two segments by X. These cases also show up in the recurrence of costs.

$$\begin{aligned}
c(n) &= x(n) \geq 0, \qquad (\text{for } n \leq w)\\
c(n) &= (n-k) + s(k) \;+\; \mathbb{E}\Big[[\![J_1, J_2 \leq \tfrac{1}{1+\alpha}(n-1)]\!] \cdot [\![J_1 > J_2]\!] \cdot \big(x(J_1) + c(J_2)\big)\Big]\\
&\qquad\qquad + \;\mathbb{E}\Big[[\![J_1, J_2 \leq \tfrac{1}{1+\alpha}(n-1)]\!] \cdot [\![J_1 \leq J_2]\!] \cdot \big(x(J_2) + c(J_1)\big)\Big]\\
&\qquad\qquad + \;\mathbb{E}\Big[[\![J_2 > \tfrac{1}{1+\alpha}(n-1)]\!] \cdot \big(x(J_1) + c(J_2)\big)\Big]\\
&\qquad\qquad + \;\mathbb{E}\Big[[\![J_1 > \tfrac{1}{1+\alpha}(n-1)]\!] \cdot \big(x(J_2) + c(J_1)\big)\Big] \qquad (\text{for } n > w)\\
&= \sum_{r=1}^{2} \mathbb{E}[A_r(J_1, J_2)c(J_r)] + t(n)
\end{aligned}$$

where

$$\begin{aligned}
A_1(J_1, J_2) &= \Big[\!\Big[J_1, J_2 \leq \tfrac{1}{1+\alpha}(n-1)\Big]\!\Big] \cdot [\![J_1 \leq J_2]\!] \;+\; \Big[\!\Big[J_1 > \tfrac{1}{1+\alpha}(n-1)\Big]\!\Big]\\
A_2(J_1, J_2) &= \Big[\!\Big[J_1, J_2 \leq \tfrac{1}{1+\alpha}(n-1)\Big]\!\Big] \cdot [\![J_2 < J_1]\!] \;+\; \Big[\!\Big[J_2 > \tfrac{1}{1+\alpha}(n-1)\Big]\!\Big]\\
t(n) &= n - k + s(k) \;+\; \mathbb{E}[A_2(J_1, J_2)x(J_1)] \;+\; \mathbb{E}[A_1(J_1, J_2)x(J_2)]
\end{aligned}$$

The above formulation actually covers $\alpha = 1$ as a special case, so abbreviating $A_r(J_1, J_2)$ by A_r, we have in both cases

$$c(n) = \sum_{r=1}^{2} \mathbb{E}[A_r\, c(J_r)] + t(n) \tag{13}$$

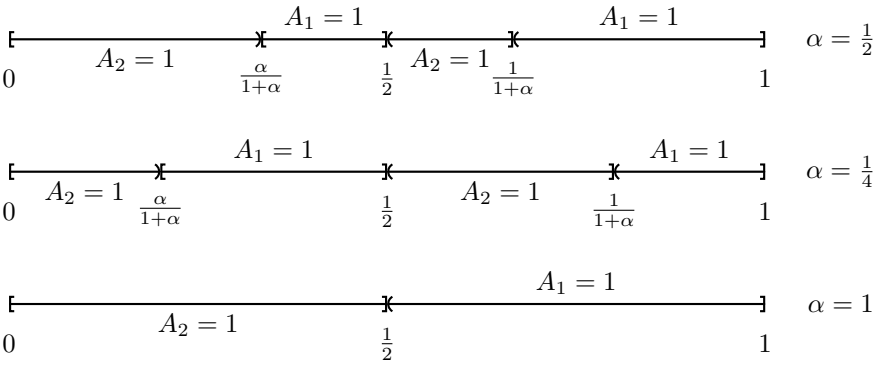
$$t(n) = n - k + s(k) \;+\; \sum_{r=1}^{2} \mathbb{E}[A_r\, x(J_{3-r})]. \tag{14}$$

We note that the expected number of partitioning rounds is only $\Theta(\log n)$ and hence also the expected overall number of comparisons used in all pivot sampling rounds combined is only $\mathcal{O}(k \log n)$.

Recursion indicator variables. It will be convenient to rewrite A_1 and A_2 in terms of the *relative subproblem size*:

$$\begin{aligned}
A_1 &= \left[\!\!\left[\frac{J_1}{n-1} \in \Big[\frac{\alpha}{1+\alpha}, \frac{1}{2}\Big] \cup \Big(\frac{1}{1+\alpha}, 1\Big]\right]\!\!\right],\\
A_2 &= \left[\!\!\left[\frac{J_2}{n-1} \in \Big[\frac{\alpha}{1+\alpha}, \frac{1}{2}\Big) \cup \Big(\frac{1}{1+\alpha}, 1\Big]\right]\!\!\right].
\end{aligned}$$

Graphically, if we view $J_1/(n-1)$ as a point in the unit interval, the following picture shows which subproblem is sorted recursively for typical values of α; (the other subproblem is sorted by X).



Obviously, we have $A_1 + A_2 = 1$ for any choice of J_1, which corresponds to having exactly one recursive call in QuickXsort.

9.2.1. Distribution of subproblem sizes

A vital ingredient to our analyses below is to characterize the distribution of the subproblem sizes J_1 and J_2.

Without pivot sampling, we have $J_1 \stackrel{\mathcal{D}}{=} \mathcal{U}[0..n-1]$, a discrete uniform distribution. In this paper, though, we assume throughout that pivots are chosen as the me the *median* of a random sample of $k = 2t + 1$, elements, where $t \in \mathbb{N}_0$. k may or may not depend on n; we write $k = k(n)$ to emphasize a potential dependency.

By symmetry, the two subproblem sizes always have the same distribution, $J_1 \stackrel{\mathcal{D}}{=} J_2$. We will therefore in the following simply write J instead of J_1 when the distinction between left and right subproblem is not important.

Combinatorial model. What is the probability $\mathbb{P}[J = j]$ to obtain a certain subproblem size j? An elementary counting argument yields the result: For the pivot to rank $(j+1)$-st in the input, the sample has to contain exactly t elements smaller than the pivot and t elements larger than the pivot. There are $\binom{n}{k}$ possible choices for the k elements in the sample, among which $\binom{j}{t} \cdot \binom{n-1-j}{t}$ choices make the $(j+1)$-st smallest element the pivot. Thus,

$$\mathbb{P}[J = j] \quad = \quad \frac{\binom{j}{t}\binom{n-1-j}{t}}{\binom{n}{k}}$$

Note that this is 0 for $j < t$ or $j > n - 1 - t$, so we can always write $J = I + t$ for a random variable $I \in [0..n-k]$ with $\mathbb{P}[I = i] = \mathbb{P}[J = i + t]$.

The following lemma can be derived by direct elementary calculations, showing that J is concentrated around its expected value $\frac{n-1}{2}$.

Lemma 9.4 ([4, Lemma 2]): *Let $0 < \delta < \frac{1}{2}$. If we choose the pivot as median of a random sample of $k = 2t + 1$ elements where $k \le \frac{n}{2}$, then the rank of the pivot $R = J_1 + 1$ satisfies*

$$\mathbb{P}\big[R \le \tfrac{n}{2} - \delta n\big] \quad < \quad k\rho^t \qquad \text{and} \qquad \mathbb{P}\big[R \ge \tfrac{n}{2} + \delta n\big] \quad < \quad k\rho^t$$

where $\rho = 1 - 4\delta^2 < 1$. □

Uniform model. There is a second view on the distribution of J that will turn out convenient for our analysis. Here, our input consists of n real numbers drawn i.i.d. uniformly from $(0,1)$. Since our algorithms are comparison based and the ranks of these numbers form a random permutation almost surely, this assumption is without loss of generality for expected-case considerations.

The vital aspect of this uniform model is that we can separate the *value* $P \in (0,1)$ of the (first) pivot from its *rank* $R \in [1..n]$. In particular, P only depends on the values in the random sample, whereas R necessarily depends on the values of all elements in the input. It is a well-known result that the median of a sample of $\mathcal{U}(0,1)$ random variates has a *beta distribution:* $P \stackrel{\mathcal{D}}{=} \mathrm{Beta}(t+1, t+1)$. Indeed, the density of the beta distribution is proportional to $x^t(1-x)^t$, which is the probability to have t of the $\mathcal{U}(0,1)$ elements $\le x$ and t elements $\ge x$ (for a given value x of the sample median).

Now suppose the pivot value P is fixed. Then, conditional on P, all further (non-sample) elements fall into the categories "smaller than P" resp. "larger than P" *independently* and with probability P resp. $1-P$ (almost surely there are no duplicates). Apart from the t small elements from the sample, J_1 precisely counts how many elements are less than P, so we can write $J_1 = I_1 + t$ where I_1 is the number of elements that turned out to be smaller than the pivot during partitioning.

Since each of the $n-k$ non-sample elements is smaller than P with probability P independent of all other elements, we have conditional on P that $I_1 \stackrel{\mathcal{D}}{=} \mathrm{Bin}(n-k, P)$. If we drop the conditioning on P, we obtain the so-called *beta-binomial distribution*: $I_1 \stackrel{\mathcal{D}}{=} \mathrm{BetaBin}(n-k, t+1, t+1)$. By "integrating P out", we obtain the probabilities as given in Section 9.1.2; (for the explicit calculation, see [15]).

The uniform model is convenient since it allows to compute expectations involving J by first conditioning on P, and then in a second step also taking expectations w.r.t. P, formally using the law of total expectation. In the first step, we can make use of the simple Chernoff bounds for the binomial distribution (Lemma B.3) instead of Lemma 9.4. The second step is often much easier than the original problem and can use known formulas for integrals, such as the ones given in Section 9.1.1. An easy calculation shows that, indeed, both models yield the same probability for $\mathbb{P}[J = j]$.

9.3. Transfer theorem for growing sample sizes

In this section, we prove our asymptotic transfer theorem for the recurrence (13), thereby expressing the expected costs of median-of-$k(n)$ QuickXsort in terms of the costs of X.

Proof of Theorem 4.1: Let $c(n)$ denote the average number of comparisons performed by QuickXsort on an input array of length n and let $x(n) = an\lg n + bn \pm \xi(n)$ with $\xi(n) \in o(n)$ be (upper and lower) bounds for the average number of comparisons performed by the algorithm X on an input array of length n. Without loss of generality we may assume that $\xi(n)$ is monotone.

Let A_1 be the indicator random variable for the event "left segment sorted recursively" and $A_2 = 1 - A_1$ similarly for the right segment. Recall that $c(n)$ fulfills the recurrence

$$\begin{aligned} c(n) &= \sum_{r=1}^{2} \mathbb{E}[A_r\, c(J_r)] \;+\; t(n), \qquad \text{where} \\ t(n) &= n - k(n) \;+\; s\big(k(n)\big) \;+\; \sum_{r=1}^{2} \mathbb{E}[A_r\, x(J_{3-r})] \end{aligned}$$

and J_1 and J_2 are the sizes for the left resp. right segment created in the first partitioning step and $s(k) \in \Theta(k)$ is the expected number of comparisons to find the median of the sample of k elements.

9.3.1. Recurrence for the difference

To prove our claim, we will bound the difference $c'(n) = c(n) - x(n)$; it satisfies a recurrence very similar to the one for $c(n)$: Recall (Equation (5) on page 17) that

$$c'(n) \;=\; \mathbb{E}[A_1\, c'(J_1)] + \mathbb{E}[A_2\, c'(J_2)] \;+\; \underbrace{n - k(n) + s(k(n)) + \mathbb{E}[x(J_1)] + \mathbb{E}[x(J_2)] - x(n)}_{t'(n)}.$$

9.3.2. Approximating the toll function

The first step to bound $c'(n)$ is a precise statement about the (asymptotic) behavior of the residual toll function $t'(n)$ given in Lemma 4.2 (page 18), which we now prove.

Proof of Lemma 4.2: We start with the simple observation that

$$J \lg J \;=\; J(\lg(\tfrac{J}{n}) + \lg n) \;=\; n \cdot \left(\tfrac{J}{n} \lg \tfrac{J}{n} + \tfrac{J}{n} \lg n\right) \;=\; \tfrac{J}{n}\, n \lg n \;+\; \tfrac{J}{n} \lg(\tfrac{J}{n})\, n. \qquad (15)$$

With that, we can simplify $t'(n)$ to (recall $s(k) \in \Theta(k)$)

$$\begin{aligned}
t'(n) \;&=\; n - k(n) + s(k(n)) \;+\; \sum_{r=1}^{2} \mathbb{E}\big[aJ_r \lg J_r + bJ_r \pm \xi(J_r)\big] - x(n) \\
&=\; n \;+\; \sum_{r=1}^{2} \left(a\mathbb{E}[J_r] \lg n + a\mathbb{E}[\tfrac{J_r}{n} \lg(\tfrac{J_r}{n})]n + b\mathbb{E}[J_r] \pm \xi(n) \right) - x(n) \;+\; \Theta(k(n)) \\
&=\; n \;+\; (an \lg n \pm \mathcal{O}(\log n)) + 2a\mathbb{E}\Big[\tfrac{J_1}{n} \lg(\tfrac{J_1}{n})\Big] n + (bn \pm \mathcal{O}(1)) \pm 2\xi(n) \\
&\qquad - (an \lg n + bn \pm \xi(n)) \;+\; \Theta(k(n)) \\
&=\; \left(1 + 2a\mathbb{E}\Big[\tfrac{J_1}{n} \lg(\tfrac{J_1}{n})\Big]\right) n \;+\; \Theta(k(n)) \pm \mathcal{O}(\xi(n)) \qquad (16)
\end{aligned}$$

The expectation $\mathbb{E}\Big[\frac{J_1}{n} \lg(\frac{J_1}{n})\Big] = -\mathbb{E}[h(J_1/n)]$ is almost of the form addressed in Lemma 9.3 when we write the beta-binomial distribution of J_1 as the mixed distribution $J_1 = t(n) + I_1$, where $I_1 \stackrel{\mathcal{D}}{=} \mathrm{BetaBin}(n-k, t+1, t+1)$: we only have to change the argument from $-\mathbb{E}[h(J_1/n)]$ to $-\mathbb{E}[h(I_1/(n-k))]$. The first step is to show that this can be done with a sufficiently small error. For brevity we write J (resp. I) instead of J_1 (resp. I_1).

Let $\delta = \delta(n) = 1/\sqrt[4]{k(n)}$. Thus, by Lemma 9.4 and $1 + x \le \exp(x)$, we obtain

$$\begin{aligned}
\mathbb{P}[J \le (1/2 - \delta)n] \;&\le\; k(n) \cdot \left(1 - 4 \cdot \frac{1}{\sqrt{k(n)}}\right)^{(k(n)-1)/2} \\
&\le\; k(n) \cdot \exp\left(-\frac{2(k(n)-1)}{\sqrt{k(n)}}\right) \\
&\le\; k(n) \cdot \exp(-\sqrt{k(n)}\,) \\
&=\; \mathcal{O}(k(n)^{-2}). \qquad (17)
\end{aligned}$$

Notice that better bounds are easily possible, but do not affect the result. We need to change the argument in the expectation from J/n to $I/(n-k)$ where $J = I + t$. The idea is that we split the expectation into two ranges: one for $J \in \{\lceil(\frac{1}{2}-\delta)n\rceil, \ldots, \lfloor(\frac{1}{2}+\delta)n\rfloor\}$ and one outside. By Equation (17), the outer part has negligible contribution. For the inner part, we will now show that the difference between J/n and $I/(n-k)$ is very small. So let $j \in \{\lceil(\frac{1}{2}-\delta)n\rceil, \ldots, \lfloor(\frac{1}{2}+\delta)n\rfloor\}$ and write $j = i + t$. Then it holds that

$$\begin{aligned}
\frac{j}{n} - \frac{i}{n-k} &= \frac{j(n-k)-(j-t)n}{n(n-k)} = \frac{tn-jk}{n(n-k)} \\
&= \frac{t-k\cdot(\frac{1}{2}\pm(\delta+\frac{1}{n}))}{n-k} && \text{(because } j = n/2 \pm (\delta n+1)\text{)} \\
&= \frac{-\frac{1}{2}\pm k(\delta+\frac{1}{n})}{n-k} = \mathcal{O}\Big(\frac{k^{3/4}}{n}\Big) && \text{(because } k = 2t+1\text{)}
\end{aligned}$$

(Note that this difference is $\Omega(k/n)$ for unrestricted values of j; only for the region close to $n/2$, the above bound holds.)

Now, recall from Lemma 9.1 that h is Hölder-continuous for any exponent $\eta \in (0,1)$ with Hölder constant $C_\eta/\ln 2$. Thus, $|h(y)-h(z)| = \mathcal{O}\Big(\Big(\frac{k(n)^{3/4}}{n}\Big)^\eta\Big)$ for $y, z \in [0,1]$ with $|y-z| = \mathcal{O}(\frac{k(n)^{3/4}}{n})$. We use this observation to show:

$$\begin{aligned}
\mathbb{E}[-h(J/n)] &= -\sum_{j=0}^{n}\mathbb{P}[J=j]h(j/n) \\
&\underset{\text{Lemma 9.1–(a)}}{=} -\sum_{j=\lfloor(1/2-\delta)n\rfloor}^{\lceil(1/2+\delta)n\rceil}\mathbb{P}[J=j]\cdot h\Big(\frac{j}{n}\Big) \pm 2\mathbb{P}\big[J \le (1/2-\delta)n\big]\cdot\frac{\lg e}{e} \\
&\underset{\text{Hölder-cont.}}{=} -\sum_{j=\lfloor(1/2-\delta)n\rfloor}^{\lceil(1/2+\delta)n\rceil}\mathbb{P}[J=j]\cdot h\Big(\frac{j-t}{n-k}\Big) \pm 2\mathbb{P}\big[J \le (1/2-\delta)n\big]\cdot\frac{\lg e}{e} \pm \mathcal{O}\Big(\Big(\frac{k(n)^{3/4}}{n}\Big)^\eta\Big) \\
&\underset{\text{Lemma 9.1–(a)}}{=} -\sum_{j=0}^{n}\mathbb{P}[J=j]\cdot h\Big(\frac{j-t}{n-k}\Big) \pm 4\mathbb{P}\big[J \le (1/2-\delta)n\big]\cdot\frac{\lg e}{e} \pm \mathcal{O}\Big(\Big(\frac{k(n)^{3/4}}{n}\Big)^\eta\Big) \\
&\underset{(17)}{=} \mathbb{E}\Big[\tfrac{I}{n-k}\lg(\tfrac{I}{n-k})\Big] \pm \mathcal{O}\Big(\frac{1}{k(n)^2} + \Big(\frac{k(n)^{3/4}}{n}\Big)^\eta\Big). \qquad (18)
\end{aligned}$$

Thus, it remains to examine $\mathbb{E}[-h(I/(n-k))]$ further. By the definition of the beta binomial distribution, we have $I \overset{\mathcal{D}}{=} \mathrm{Bin}(n-k(n), P)$ conditional on the *value* of the pivot $P \overset{\mathcal{D}}{=} \mathrm{Beta}(t(n)+1, t(n)+1)$ (see Section 9.2.1). So we apply Lemma 9.3 on the *conditional* expectation to get for any $\zeta \ge 0$:

$$\mathbb{E}\Big[h(\tfrac{I}{n-k}) \,\Big|\, P\Big] = h(P) \pm \rho$$

where

$$\rho = \frac{C_\eta}{\ln 2}\cdot\zeta^\eta\Big(1-2e^{-2\zeta^2(n-k(n))}\Big) + 4\frac{\lg e}{e}e^{-2\zeta^2(n-k(n))}.$$

By Equation (10) and the asymptotic expansion of the harmonic numbers (see, e.g., [23, Eq. (9.89)]), we find

$$\begin{aligned}\mathbb{E}\Big[-h\Big(\tfrac{I}{n-k}\Big)\Big] &= -\mathbb{E}[h(P)] \pm \rho \\ &\underset{(10)}{=} -\frac{\frac{1}{2}(H_{k(n)+1} - H_{(k(n)+1)/2})}{\ln 2} \pm \rho \\ &= -\frac{1}{2}\cdot\frac{\ln 2 - \Theta(1/k(n))}{\ln 2} \pm \rho\end{aligned}$$

and using the choice for ζ from Lemma 9.3

$$= -\frac{1}{2} + \Theta(k(n)^{-1}) \pm \mathcal{O}(n^{-1/2+\varepsilon})$$

for any fixed $\varepsilon \in (0, \frac{1}{2})$ with $\varepsilon > \frac{1-\eta}{2}$ (recall that still ε can be an arbitrarily small constant). Together with (16) and (18) this allows us to estimate $t'(n)$. Here, we set $\varepsilon' = 1 - \eta$:

$$t'(n) = (1-a)n + \Theta\Big(k(n) + \tfrac{n}{k(n)}\Big) \pm \mathcal{O}\Big(\xi(n) + \sqrt{n}\cdot n^{\varepsilon} + \tfrac{n}{k(n)^2} + k(n)^{3/4}\cdot\big(\tfrac{n}{k(n)^{3/4}}\big)^{\varepsilon'}\Big)$$

replacing ε and ε' by their maximum, we obtain for any small enough $\varepsilon > 0$, that

$$\begin{aligned}t'(n) &= (1-a)n + \Theta\Big(k(n) + \tfrac{n}{k(n)}\Big) \pm \mathcal{O}\Big(\xi(n) + n^{\varepsilon}\cdot\Big(\sqrt{n} + k(n)^{3/4}\Big) + \tfrac{n}{k(n)^2}\Big) \\ &= (1-a)n + \Theta\Big(k(n) + \tfrac{n}{k(n)}\Big) \pm \mathcal{O}\Big(\xi(n) + n^{1/2+\varepsilon}\Big).\end{aligned}$$

To see the last step, let us verify that $n^{\varepsilon}k(n)^{3/4} = \mathcal{O}(n^{1/2+\varepsilon}) + o(k(n))$: we write $\mathbb{N} = N_1 \cup N_2$ with $N_1 = \{\, n \in \mathbb{N} \mid k(n) \le \sqrt{n} \,\}$ and $N_1 = \{\, n \in \mathbb{N} \mid k(n) \ge \sqrt{n} \,\}$. For $n \in N_1$ clearly we have $n^{\varepsilon}k(n)^{3/4} \le n^{1/2+\varepsilon}$. For $n \in N_2$, we have $\sqrt[4]{k(n)} \ge \sqrt[8]{n} \ge n^{\varepsilon+\varepsilon''}$ for some small $\varepsilon'' > 0$ (here we need that ε is small); thus, $n^{\varepsilon}k(n)^{3/4} \le k(n)n^{-\varepsilon''}$. Altogether, we obtain $n^{\varepsilon}k(n)^{3/4} = \mathcal{O}(n^{1/2+\varepsilon}) + o(k(n))$.

In the case that $a = 1$, $k(n) = \Theta(n^{\kappa})$ for $\kappa \in (0,1)$ and $\xi(n) = \mathcal{O}(n^{\delta})$ for $\delta \in [0,1)$, we have

$$t'(n) = \Theta\Big(n^{\max\{\kappa, 1-\kappa\}}\Big) \pm \mathcal{O}\Big(n^{\max\{\delta, \frac{1}{2}+\varepsilon\}}\Big)$$

That concludes the proof of Lemma 4.2. □

Note that $t'(n)$ can be positive or negative (depending on $x(n)$), but the Θ-bound is definitively a positive term, and it will be minimal for $k(n) \sim \sqrt{n}$. Now that we know the order of growth of $t'(n)$, we can proceed to our recurrence for the difference $c'(n)$.

9.3.3. Bounding the difference

The final step is to bound $c'(n)$ from above. Recall that by (5), we have $c'(n) = \mathbb{E}[A_1\, c(J_1)] + \mathbb{E}[A_2\, c'(J_2)] + t'(n)$. For the case $a > 1$, Lemma 4.2 tells us that $t'(n)$ is eventually negative and asymptotic to $(1-a)n$. Thus $c'(n)$ is eventually negative, as well, i.e., $c(n) \le x(n)$ for large enough n. The claim follows.

We therefore are left with the case $a = 1$. Lemma 4.2 only gives us a bound in that case and certainly $t'(n) = o(n)$. The fact that $t'(n)$ can in general be positive or negative and need not be monotonic, makes solving the recurrence for $c'(n)$ a formidable problem. Since we are only interested in an upper bound, we can use the recursion-tree-style result stated in Lemma 4.3 (page 18), which we now prove.

Proof of Lemma 4.3: Since $\hat{t}(n)$ is non-negative and monotonically increasing, so is $\hat{c}(n)$ and we can bound

$$\hat{c}(n) \;\leq\; \mathbb{E}\big[\hat{c}(\max\{J_1, J_2\})\big] \;+\; \hat{t}(n).$$

Let us abbreviate $\hat{J} = \max\{J_1, J_2\}$. For given constant $\beta \in (\frac{1}{2}, 1)$, we have by the law of total expectation and monotonicity of $\hat{c}$ that

$$\begin{aligned}\hat{c}(n) &\leq \mathbb{E}\big[\hat{c}(\hat{J})\big] \;+\; \hat{t}(n)\\ &= \mathbb{E}\big[\hat{c}(\hat{J}) \mid \hat{J} \leq \beta n\big] \cdot \mathbb{P}\big[\hat{J} \leq \beta n\big] \;+\; \mathbb{E}\big[\hat{c}(\hat{J}) \mid \hat{J} > \beta n\big] \cdot \mathbb{P}\big[\hat{J} > \beta n\big] \;+\; \hat{t}(n)\\ &\leq \mathbb{P}\big[\hat{J} \leq \beta n\big] \cdot \hat{c}(\beta n) + \mathbb{P}\big[\hat{J} > \beta n\big] \cdot \hat{c}(n) \;+\; \hat{t}(n).\end{aligned}$$

For fixed $\beta > \frac{1}{2}$, we can bound $\mathbb{P}\big[\hat{J} \leq \beta n\big] \geq C > 0$ for a constant C and all large enough n. Hence for n large enough,

$$\begin{aligned}\hat{c}(n) &\leq \frac{\mathbb{P}\big[\hat{J} \leq \beta n\big]}{1 - \mathbb{P}\big[\hat{J} > \beta n\big]} \cdot \hat{c}(\beta n) \;+\; \frac{1}{1 - \mathbb{P}\big[\hat{J} > \beta n\big]} \cdot \hat{t}(n)\\ &\leq \hat{c}(\beta n) \;+\; \frac{1}{C} \cdot \hat{t}(n).\end{aligned}$$

Iterating the last inequality $\lceil \log_{1/\beta}(n) \rceil$ times, we find $\hat{c}(n) \leq \frac{1}{C} \sum_{i=0}^{\lceil \log_{1/\beta}(n) \rceil} \hat{t}(n\beta^i)$. □

We can apply this lemma if we replace $t'(n)$ by $\hat{t}(n) := \max_{m \leq n} |t'(m)|$, which is both non-negative and monotone. We clearly have $t'(n) \leq \hat{t}(n)$ by definition. Moreover, if $t'(n) = \mathcal{O}(g(n))$ for a monotonically increasing function g, then also $\hat{t}(n) = \mathcal{O}(g(n))$, and the same statement holds with $\mathcal{O}$ replaced by o.

Now let $\hat{c}(n)$ be defined by the recurrence $\hat{c}(n) = \mathbb{E}\big[A_1\, \hat{c}(J_1)\big] + \mathbb{E}\big[A_2\, \hat{c}(J_2)\big] + \hat{t}(n)$. Then, we have $|c'(n)| \leq \hat{c}(n)$. We will now bound $\hat{c}(n)$.

9.3.4. $o(n)$ bound

We first show that $\hat{c}(n) = o(n)$. By Lemma 4.2 we have $t'(n) = o(n)$, and by the above argument also $\hat{t}(n) = o(n)$. Since $\hat{t}(n) \in o(n)$, we know that for every $\varepsilon > 0$, there is some $N_\varepsilon \in \mathbb{N}$ such that for $n \geq N_\varepsilon$, we have $\hat{t}(n) \leq n\varepsilon$. Let $D_\varepsilon = \sum_{i=0}^{N_\varepsilon} \hat{t}(i)$. Then, for any $\beta \in (\frac{1}{2}, 1)$ by Lemma 4.3 there is some constant C such that for all n we have

$$|c'(n)| \;\leq\; \hat{c}(n) \;\leq\; C \sum_{i=0}^{\lceil \log_{1/\beta}(n) \rceil} \hat{t}(n\beta^i) \;\leq\; D_\varepsilon + C \sum_{i=0}^{\lceil \log_{1/\beta}(n) \rceil} \varepsilon \beta^i n \;\leq\; C D_\varepsilon + \varepsilon' n$$

for $\varepsilon' := \frac{C}{1-\beta} \cdot \varepsilon \geq C\varepsilon \sum_{i=0}^{\lceil \log_{1/\beta}(n) \rceil} \beta^i$. Since we can hence find a suitable $\varepsilon = \varepsilon(\varepsilon') > 0$ for any given $\varepsilon' > 0$, the above inequality holds for all $\varepsilon' > 0$, and therefore $\hat{c}(n) = o(n)$ holds. This proves the first part of Theorem 4.1.

9.3.5. Refined bound

Now, consider the case that $k(n) = \Theta(n^\kappa)$ for $\kappa \in (0, 1)$ and $\xi(n) \in \mathcal{O}(n^\delta)$ with $\delta \in [0, 1)$. Then, by Lemma 4.2, we have $t'(n) = \mathcal{O}(n^\gamma)$ for some $\gamma \in (0, 1)$, i.e., there is some constant

C_γ such that $t'(n) \leq C_\gamma n^\gamma + \mathcal{O}(1)$. By Lemma 4.3, we obtain

$$c'(n) \;\leq\; \sum_{i=0}^{\lceil \log_{1/\beta}(n) \rceil} C_\gamma (n\beta^i)^\gamma + \mathcal{O}(1) \;\leq\; C_\gamma n^\gamma \sum_{i\geq 0} (\beta^\gamma)^i + \mathcal{O}(1) \;=\; \mathcal{O}(n^\gamma).$$

Moreover, if $t'(n) \in \Theta(n^\gamma)$ (the case that $\max\{\kappa, 1-\kappa\} > \max\{\delta, \frac{1}{2}+\varepsilon\}$ in Lemma 4.2), then also $c'(n) \in \Theta(n^\gamma)$ as $c'(n) \geq t'(n)$. This concludes the proof of the last part of Theorem 4.1. □

9.4. Large deviation and worst-case bounds

In this section, we prove our results on the likelihood of large deviations in the cost of median-of-$\sqrt{n}$ QuickXsort, and for the influence of median-of-medians pivot selection to obtain strong worst-case guarantees.

Proof of Proposition 4.5: Let n be the size of the input. We say that we are in a *good* case if an array of size m is partitioned in the interval $[m/4..3m/4]$, i.e., if the pivot rank is chosen in that interval. We can obtain a bound for the desired probability by estimating the probability that we are always in such a good case until the array contains only $\sqrt{n}$ elements. For smaller arrays, we can assume an upper bound of $\sqrt{n}^2 = n$ comparisons for the worst case.

If we are always in a good case, all partitioning steps sums up to less than $n \cdot \sum_{i\geq 0}(3/4)^i = 4n$ comparisons. However, we also have to consider the number of comparisons required to find the pivot element. At any stage the pivot is chosen as median of at most $\sqrt{n}$ elements. Since the median can be determined in linear time, for all stages together this sums up to less than n comparisons if we are always in a good case and n is large enough. Finally, for all the sorting phases with X, we need at most $x_{\mathrm{wc}}(n)$ comparisons in total (that is only a rough upper bound which could be improved). Hence, we need at most $x_{\mathrm{wc}}(n) + 6n$ comparisons if always a good case occurs.

Now, we only have to estimate the probability that always a good case occurs. By Lemma 9.4, the probability for a good case in the first partitioning step is at least $1 - d \cdot \sqrt{n} \cdot (3/4)^{\sqrt{n}}$ for some constant d. We have to choose $\log_{3/4}(\sqrt{n}/n) < 1.21 \lg n$ times a pivot in the interval $[m/4..3m/4]$, then the array has size less than $\sqrt{n}$. We only have to consider partitioning steps where the array has size greater than $\sqrt{n}$ (if the size of the array is already less than $\sqrt{n}$ we define the probability of a good case as 1). Hence, for each of these partitioning steps we obtain that the probability for a good case is greater than $1 - d \cdot \sqrt[4]{n} \cdot (3/4)^{\sqrt[4]{n}}$. Therefore, we obtain

$$\begin{aligned} \mathbb{P}[\text{always good case}] \;&\geq\; \left(1 - d \cdot \sqrt[4]{n} \cdot (3/4)^{\sqrt[4]{n}}\right)^{1.21 \lg(n)} \\ &\geq\; 1 - 1.21 \lg(n) \cdot d \cdot \sqrt[4]{n} \cdot (3/4)^{\sqrt[4]{n}} \end{aligned}$$

by Bernoulli's inequality. For n large enough we have $1.21 \lg(n) \cdot d \cdot \sqrt[4]{n} \cdot (3/4)^{\sqrt[4]{n}} \leq (3/4 + \varepsilon)^{\sqrt[4]{n}}$. □

Proof of Theorem 4.7: It is clear that the worst case is $n \lg n + \mathcal{O}(n)$ comparisons since there can be at most $\max\{2 \lg n, \log_{1/2+\delta} n\}$ rounds of partitioning (by the additional

requirement, pivot selection takes at most linear time). Thus, it remains to consider the average case – for which we follow the proof of Theorem 4.1. We say a pivot choice is "bad" if the next pivot is selected as median of the whole array (i.e., if $J_1 \leq (1/2-\delta)n$ or $J_2 \leq (1/2-\delta)n$), otherwise we call the pivot "good".

The difference to the situation in Theorem 4.1 is that now we have four segments to distinguish instead of two: let A'_1 be the indicator random variable for the event "left segment sorted recursively" and A'_2 similarly for the right segment – both for the case that the pivot was good. Likewise, let A''_1 be the indicator random variable for the event "left segment sorted recursively " and $A''_2 = 1 - A'_1 - A'_2 - A''_1$ "right segment sorted recursively" in the case that the pivot was bad. Then, $A_1 = A'_1 + A''_1$ is the indicator random variable for the event "left segment sorted recursively" and $A_2 = A'_2 + A''_2$ the same for the right segment. Let $c(n)$ denote the average number of comparisons of median-of-$k(n)$ QuickXsort with median-of-medians fallback pivot selection and $\tilde{c}(n)$ the same but in the case that the first pivot is selected with the median-of-medians algorithms. We obtain the following recurrence

$$\begin{aligned} c(n) &= \underbrace{n-k(n)}_{\text{partitioning}} + \underbrace{s\big(k(n)\big)}_{\text{pivot sampling}} + \mathbb{E}\Big[A'_1\cdot\big(c(J_1)+x(J_2)\big) + A'_2\cdot\big(c(J_2)+x(J_1)\big)\Big] \\ &\quad + \mathbb{E}\Big[A''_1\cdot\big(\tilde{c}(J_1)+x(J_2)\big) + A''_2\cdot\big(\tilde{c}(J_2)+x(J_1)\big)\Big] \\ &= \sum_{r=1}^{2}\mathbb{E}[A_r\,c(J_r)] + t(n), \qquad \text{where} \\ t(n) &= n-k(n) + s\big(k(n)\big) + \sum_{r=1}^{2}\mathbb{E}[A_r\,x(J_r)] + \sum_{r=1}^{2}\mathbb{E}[A''_{r+2}\,(\tilde{c}(J_r)-c(J_r))]. \end{aligned}$$

As before $s(k)$ is the number of comparisons to select the median from the k sample elements and J_1 and J_2 are the sizes for the left resp. right segment created in the first partitioning step. Since $n\log n - \mathcal{O}(n) \leq \tilde{c}(n) \leq c_{\text{wc}}(n)$ and $c_{\text{wc}}(n) = n\log n + \mathcal{O}(n)$, it follows that $\tilde{c}(n) - c(n) \in \mathcal{O}(n)$. By Lemma 9.4 we have $\mathbb{P}[A''_1], \mathbb{P}[A''_2] \in o(1)$. Thus,

$$\zeta(n) := \sum_{r=1}^{2}\mathbb{E}[A''_r\,(\tilde{c}(J_r)-c(J_r))] \in o(n).$$

As for Theorem 4.1 we now consider $c'(n) = c(n) - x(n)$ yielding

$$\begin{aligned} c'(n) &= n-k(n) + s\big(k(n)\big) + \mathbb{E}\Big[A_1\cdot\big(c'(J_1)+x(J_1)+x(J_2)\big)\Big] \\ &\quad + \mathbb{E}\Big[A_2\cdot\big(c'(J_2)+x(J_2)+x(J_1)\big)\Big] + \zeta(n) - x(n) \\ &= \mathbb{E}[A_1\,c'(J_1)] + \mathbb{E}[A_2\,c'(J_2)] + t'(n) \end{aligned}$$

for $t'(n) = n-k(n)+s\big(k(n)\big)+\mathbb{E}[x(J_1)]+\mathbb{E}[x(J_2)]+\zeta(n)-x(n)$. Now the proof proceeds exactly as for Theorem 4.1. □

9.5. Transfer theorem for fixed sample sizes

In this section, we prove our transfer theorem for median-of-k QuickXsort for k a fixed constant.[11]

[11] Although the statement of our theorem is the same as for [52, Theorem 5.1], our proof here is significantly shorter than the one given there. By first taking the difference $c(n) - x(n)$, we turn the much more

Proof of Theorem 4.8: Recall that $c(n)$ denotes the expected number of comparisons performed by QuickXsort. With $x(n) = an\lg n + bn \pm \xi(n)$ for a monotonic function $\xi(n) = \mathcal{O}(n^{1-\varepsilon})$, the same arguments as in the proof of Theorem 4.1 lead to

$$\begin{aligned} t'(n) &= \Big(1 + 2a\mathbb{E}\Big[\tfrac{J_1}{n}\lg(\tfrac{J_1}{n})\Big]\Big)n \;+\; \Theta(s(k(n))) \pm \mathcal{O}(\xi(n)) && (16)\text{ revisited} \\ &= \Big(1 + 2a\mathbb{E}\Big[\tfrac{J}{n}\lg(\tfrac{J}{n})\Big]\Big)n \;\pm\; \mathcal{O}(n^{1-\varepsilon}). && (6)\text{ revisited} \end{aligned}$$

We will first derive an asymptotic approximation for $t'(n)$, before we apply Roura's continuous master theorem (CMT, Theorem B.4) to solve the recurrence for $c'(n)$.

9.5.1. Approximation by beta integrals

We begin by proving our beta-integral approximation.

Proof of Lemma 4.9: By the local limit law for beta binomials (Lemma 9.2), it is plausible to expect a reasonably small error when we replace $\mathbb{E}[J\lg J]$ by $\mathbb{E}[(Pn)\lg(Pn)]$ where $P \overset{\mathcal{D}}{=} \mathrm{Beta}(\lambda,\rho)$ is beta distributed. We bound the error in the following.

We first replace J by $I \overset{\mathcal{D}}{=} \mathrm{BetaBin}(n,\lambda,\rho)$ and argue later that this results in a sufficiently small error.

$$\begin{aligned} \mathbb{E}[\tfrac{I}{n}\ln(\tfrac{I}{n})] &= \sum_{i=0}^{n} \tfrac{i}{n}\ln(\tfrac{i}{n})\cdot\mathbb{P}[I=i] \\ &= \frac{1}{n}\sum_{i=0}^{n} \tfrac{i}{n}\ln(\tfrac{i}{n})\cdot n\mathbb{P}[I=i] \\ &\overset{\text{Lemma 9.2}}{=} \frac{1}{n}\sum_{i=0}^{n} \tfrac{i}{n}\ln\tfrac{i}{n}\cdot\left(\frac{(i/n)^{\lambda-1}(1-(i/n))^{\rho-1}}{\mathrm{B}(\lambda,\rho)} \;\pm\; \mathcal{O}(n^{-1})\right) \\ &= -\frac{1}{\mathrm{B}(\lambda,\rho)}\cdot\frac{1}{n}\sum_{i=0}^{n} f(i/n) \;\pm\; \mathcal{O}(n^{-1}), \end{aligned}$$

where $f(z) = \ln(1/z)\cdot z^{\lambda}(1-z)^{\rho-1}$. Since the derivative is ∞ for $z=0$, f cannot be Lipschitz-continuous, but it is Hölder-continuous on $[0,1]$ for any exponent $\eta \in (0,1)$. This is because $z \mapsto z\ln(1/z)$ is Hölder-continuous (Lemma 9.1–(b)), products of Hölder-continuous function remain so on bounded intervals and the remaining factor of f is a polynomial in z, which is Lipschitz- and hence Hölder-continuous. By Lemma B.1 we then have

$$\frac{1}{n}\sum_{i=0}^{n} f(i/n) \;=\; \int_0^1 f(z)\,dz \;\pm\; \mathcal{O}(n^{-\eta}).$$

Note that we can choose η as close to 1 as we wish; this will only affect the constant inside $\mathcal{O}(n^{-\eta})$.

Changing from I back to J has no influence on the given approximation: To compensate for the difference in the number of trials ($n - c_1$ instead of n), we use the above formulas for $n - c_1$ instead of n; since we let n go to infinity anyway, this does not change the result.

complicated terms $\mathbb{E}[A_r x(J_{3-r})]$ from $t(n)$ into the simpler $\mathbb{E}[x(J_r)]$ in $t'(n)$, which allows us to entirely omit [52, Lemma E.1].

Moreover, replacing I by $I + c_2$ changes the value of the argument $z = I/n$ of f by $\mathcal{O}(n^{-1})$; since $z \mapsto z\ln(1/z)$ is smooth, namely Hölder-continuous, this also changes $z\ln(1/z)$ by at most $\mathcal{O}(n^{-\eta})$.

It remains to evaluate the beta integral; it is given in Equation (10). Inserting, we find

$$\begin{aligned}\mathbb{E}[\tfrac{J}{n}\ln\tfrac{J}{n}] &= \mathbb{E}[\tfrac{I}{n}\ln\tfrac{I}{n}] \pm \mathcal{O}(n^{-\eta}) \\ &= \frac{\lambda}{\lambda+\rho}(H_\lambda - H_{\lambda+\rho}) \pm \mathcal{O}(n^{-\eta})\end{aligned}$$

for any $\eta \in (0,1)$. □

Remark 9.5 (General beta-integral approximation): *The above technique directly extends to $\mathbb{E}[g(\frac{J}{n})]$ for any Hölder-continuous function g. For computing the variance in Section 4.4, we will have to deal with more complicated functions including the indicator variables $A_1(J_1, J_2)$ resp. $A_2(J_1, J_2)$. As long as g is piecewise Hölder-continuous, the same arguments and error bounds apply: We can break the sums resp. integrals into several parts and apply the above approximation to each part individually. The indicator variables simply translate into restricted bounds of the integral. For example, we obtain for constants $0 \le x \le y \le 1$ that*

$$\mathbb{E}\big[[\![xn \le J \le yn]\!] \cdot J\lg J\big] = \frac{\lambda}{\lambda+\rho} I_{x,y}(\lambda+1,\rho)\cdot n\lg n \pm \mathcal{O}(n), \qquad (n\to\infty).$$

9.5.2. The toll function

Building on the preparatory work from Lemma 4.9, we can easily determine an asymptotic approximation for the toll function. We find

$$\begin{aligned}t'(n) &= \Big(1 + 2a\mathbb{E}\Big[\tfrac{J}{n}\lg(\tfrac{J}{n})\Big]\Big)n \pm \mathcal{O}(n^{1-\varepsilon}) \\ &= \Big(1 + 2a\frac{\mathbb{E}\Big[\frac{J}{n}\ln(\frac{J}{n})\Big]}{\ln 2}\Big)n \pm \mathcal{O}(n^{1-\varepsilon}) \\ &\overset{\text{Lemma 4.9}}{=} \left(1 + \frac{2a}{\ln 2}\left(\frac{t+1}{2(t+1)}(H_{t+1} - H_{2t+2}) \pm \mathcal{O}(n^{-\eta})\right)\right)n \pm \mathcal{O}(n^{1-\varepsilon}) \\ &= \underbrace{\left(1 - \frac{a(H_{k+1} - H_{t+1})}{\ln 2}\right)}_{\hat{q}} n \pm \mathcal{O}(n^{1-\varepsilon} + n^{1-\eta}). \qquad (19)\end{aligned}$$

9.5.3. The shape function

Towards applying Roura's CMT, we first rewrite our recurrence in the form required by the theorem.

The expectations $\mathbb{E}[A_r\, c'(J_r)]$ in Equation (5) (and in the same way for the original costs in Equation (13)) are finite sums over the values $0, \dots, n-1$ that $J := J_1$ can attain. Recall

that $J_2 = n - 1 - J_1$ and $A_1 + A_2 = 1$ for any value of J. With $J = J_1 \overset{\mathcal{D}}{=} J_2$, we find

$$\begin{aligned}
\sum_{r=1}^{2} \mathbb{E}[A_r(J_r)c(J_r)] &= \mathbb{E}\left[\left[\!\left[\frac{J}{n-1} \in \left[\frac{\alpha}{1+\alpha}, \frac{1}{2}\right] \cup \left(\frac{1}{1+\alpha}, 1\right]\right]\!\right] \cdot c(J)\right] \\
&\quad + \mathbb{E}\left[\left[\!\left[\frac{J}{n-1} \in \left[\frac{\alpha}{1+\alpha}, \frac{1}{2}\right) \cup \left(\frac{1}{1+\alpha}, 1\right]\right]\!\right] \cdot c(J)\right] \\
&= \sum_{j=0}^{n-1} w_{n,j} \cdot c(j), \qquad \text{where} \\
w_{n,j} &= \mathbb{P}[J = j] \cdot \left[\!\left[\tfrac{j}{n-1} \in [\tfrac{\alpha}{1+\alpha}, \tfrac{1}{2}] \cup (\tfrac{1}{1+\alpha}, 1]\right]\!\right] \\
&\quad + \mathbb{P}[J = j] \cdot \left[\!\left[\tfrac{j}{n-1} \in [\tfrac{\alpha}{1+\alpha}, \tfrac{1}{2}) \cup (\tfrac{1}{1+\alpha}, 1]\right]\!\right] \\
&= \begin{cases} 2 \cdot \mathbb{P}[J = j] & \text{if } \frac{j}{n-1} \in [\frac{\alpha}{1+\alpha}, \frac{1}{2}) \cup (\frac{1}{1+\alpha}, 1] \\ 1 \cdot \mathbb{P}[J = j] & \text{if } \frac{j}{n-1} = \frac{1}{2} \\ 0 & \text{otherwise.} \end{cases}
\end{aligned}$$

We thus have a recurrence of the form required by the CMT with the weights $w_{n,j}$ from above. Figure 17 shows a specific example for how these weights look like.

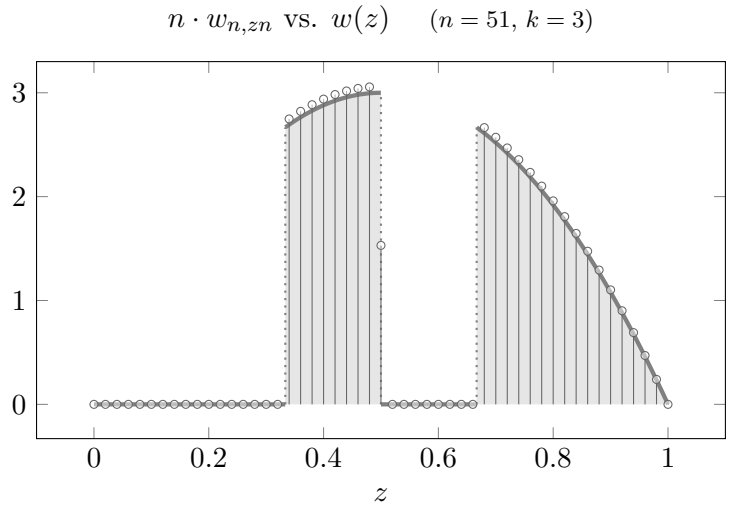


Figure 17: The weights $w_{n,j}$ (circles) for $n = 51$, $t = 1$ and $\alpha = \frac{1}{2}$ and the corresponding shape function $w(z)$ (fat gray line); note the singular point at $j = 25$.

It remains to determine $\mathbb{P}[J = j]$. Recall that we choose the pivot as the median of $k = 2t + 1$ elements for a fixed constant $t \in \mathbb{N}_0$, and the subproblem size J fulfills $J = t + I$ with $I \overset{\mathcal{D}}{=} \text{BetaBin}(n - k, t + 1, t + 1)$. So we have for $i \in [0, n - 1 - t]$ by definition

$$\begin{aligned}
\mathbb{P}[I = i] &= \binom{n-k}{i} \frac{\mathrm{B}(i + t + 1, (n - k - i) + t + 1)}{\mathrm{B}(t + 1, t + 1)} \\
&= \binom{n-k}{i} \frac{(t+1)^{\overline{i}}(t+1)^{\overline{n-k-i}}}{(k+1)^{\overline{n-k}}}
\end{aligned}$$

The next step towards applying the CMT is to identify a shape function $w(z)$ that approximates the relative subproblem size probabilities $w(z) \approx n w_{n,\lfloor zn \rfloor}$ for large n. Now the local limit law for beta binomials (Lemma 9.2) says that the normalized beta binomial I/n converges to a beta variable "in density", and the convergence is uniform. With the beta density $f_P(z) = z^t(1-z)^t/\mathrm{B}(t+1,t+1)$, we thus find by Lemma 9.2 that

$$\mathbb{P}[J=j] \;=\; \mathbb{P}[I=j-t] \;=\; \frac{1}{n} f_P(j/n) \;\pm\; \mathcal{O}(n^{-2}), \qquad (n\to\infty).$$

The shift by the small constant t from $(j-t)/n$ to j/n only changes the function value by $\mathcal{O}(n^{-1})$ since f_P is Lipschitz continuous on $[0,1]$ (see Section B.1).

With this observation, a natural candidate for the shape function of the recurrence is

$$w(z) \;=\; 2\left[\!\left[\tfrac{\alpha}{1+\alpha} < z < \tfrac{1}{2} \;\vee\; z > \tfrac{1}{1+\alpha}\right]\!\right] \frac{z^t(1-z)^t}{\mathrm{B}(t+1,t+1)}. \tag{20}$$

It remains to show that this is indeed a suitable shape function, i.e., that $w(z)$ fulfills Equation (28), the approximation-rate condition of the CMT.

We consider the following ranges for $\frac{\lfloor zn \rfloor}{n-1} = \frac{j}{n-1}$ separately:

- $\frac{\lfloor zn \rfloor}{n-1} < \frac{\alpha}{1+\alpha}$ and $\frac{1}{2} < \frac{\lfloor zn \rfloor}{n-1} < \frac{1}{1+\alpha}$.
  Here $w_{n,\lfloor zn \rfloor} = 0$ and so is $w(z)$. So actual value and approximation are exactly the same.
- $\frac{\alpha}{1+\alpha} < \frac{\lfloor zn \rfloor}{n-1} < \frac{1}{2}$ and $\frac{\lfloor zn \rfloor}{n-1} > \frac{1}{1+\alpha}$.
  Here $w_{n,j} = 2\mathbb{P}[J=j]$ and $w(z) = 2f_P(z)$ where $f_P(z) = z^t(1-z)^t/\mathrm{B}(t+1,t+1)$ is twice the density of the beta distribution $\mathrm{Beta}(t+1,t+1)$. Since f_P is Lipschitz-continuous on the bounded interval $[0,1]$ (it is a polynomial) the uniform pointwise convergence from above is enough to bound the sum of $\left|w_{n,j} - \int_{j/n}^{(j+1)/n} w(z)\,dz\right|$ over all j in the range by $\mathcal{O}(n^{-1})$.
- $\frac{\lfloor zn \rfloor}{n-1} \in \{\frac{\alpha}{1+\alpha}, \frac{1}{2}, \frac{1}{1+\alpha}\}$.
  At these boundary points, the difference between $w_{n,\lfloor zn \rfloor}$ and $w(z)$ does not vanish (in particularly $\frac{1}{2}$ is a singular point for $w_{n,\lfloor zn \rfloor}$), but the absolute difference is bounded. Since this case only concerns 3 out of n summands, the overall contribution to the error is $\mathcal{O}(n^{-1})$.

Together, we find that Equation (28) is fulfilled as claimed:

$$\sum_{j=0}^{n-1} \left| w_{n,j} \;-\; \int_{j/n}^{(j+1)/n} w(z)\,dz \right| \;=\; \mathcal{O}(n^{-1}) \qquad (n\to\infty). \tag{21}$$

Remark 9.6 (Relative subproblem sizes): *The integral $\int_0^1 z w(z)\,dz$ is precisely the expected relative subproblem size for the recursive call. This is of independent interest; while it is intuitively clear that for $t\to\infty$, i.e., the case of exact medians as pivots, we must have a relative subproblem size of exactly $\frac{1}{2}$, this convergence is not obvious from the behavior for finite t: the mass of the integral $\int_0^1 z w(z)\,dz$ concentrates at $z=\frac{1}{2}$, a point of discontinuity in $w(z)$. It is also worthy of note that for, e.g., $\alpha=\frac{1}{2}$, the expected subproblem size is initially larger than $\frac{1}{2}$ ($0.69\overline{4}$ for $t=0$), then decreases to ≈ 0.449124 around $t=20$ and then starts to slowly increase again (see Figure 18). This effect is even more pronounced for $\alpha=\frac{1}{4}$.*

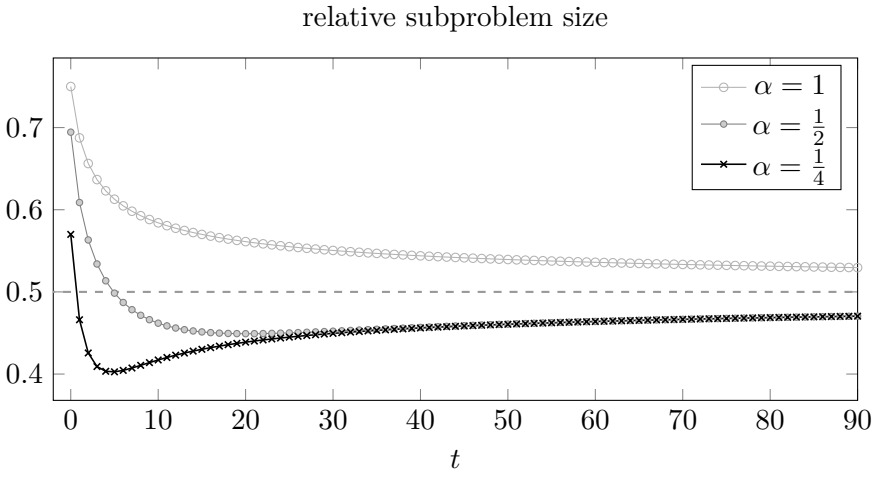


Figure 18: $\int_0^1 zw(z)\,dz$, the relative recursive subproblem size, as a function of t.

9.5.4. Which case of the CMT?

We are now ready to apply the CMT (Theorem B.4). Assume that $a \neq \ln 2/(H_{k+1} - H_{t+1})$; the other (special) case will be addressed later. Then by Equation (19) our toll function fulfills $t'(n) \sim \hat{q}n$ for $\hat{q} = (1 - a(H_{k+1} - H_{t+1})/\ln 2)$. Thus, we have $\sigma = 1$, $\tau = 0$ and $K = \hat{q} \neq 0$ and we compute

$$\begin{aligned}
H &= 1 - \int_0^1 z\, w(z)\, dz \\
&= 1 - \int_0^1 2 \left[\!\left[\tfrac{\alpha}{1+\alpha} < z < \tfrac{1}{2} \ \vee\ z > \tfrac{1}{1+\alpha} \right]\!\right] \frac{z^{t+1}(1-z)^t}{\mathrm{B}(t+1, t+1)}\, dz \\
&= 1 - 2\frac{t+1}{k+1} \int_0^1 \left[\!\left[\tfrac{\alpha}{1+\alpha} < z < \tfrac{1}{2} \ \vee\ z > \tfrac{1}{1+\alpha} \right]\!\right] \frac{z^{t+1}(1-z)^t}{\mathrm{B}(t+2, t+1)}\, dz \\
&= 1 - \left(I_{\frac{\alpha}{1+\alpha}, \frac{1}{2}}(t+2, t+1) + I_{\frac{1}{1+\alpha}, 1}(t+2, t+1) \right) \\
&= I_{0, \frac{\alpha}{1+\alpha}}(t+2, t+1) + I_{\frac{1}{2}, \frac{1}{1+\alpha}}(t+2, t+1) \qquad (22)
\end{aligned}$$

For any sampling parameters, we have $H > 0$, so by Case 1 of Theorem B.4, we have that

$$c'(n) \ \sim\ \frac{t'(n)}{H} \ \sim\ \frac{\hat{q}n}{H} \ =\ qn, \qquad (n \to \infty).$$

Special case for a. If $a = \ln 2/(H_{k+1} - H_{t+1})$, i.e., $\hat{q} = 0$, then $t'(n) = \mathcal{O}(n^{1-\varepsilon})$. Then the claim follows from a coarser bound for $c'(n) = \mathcal{O}(n^{1-\varepsilon} + \log n)$ which can be established by the same arguments as in the proof of Theorem 4.1.

9.5.5. Error bound

Since our toll function is not given precisely, but only up to an error term $\mathcal{O}(n^{1-\varepsilon})$ for a given fixed $\varepsilon \in (0, 1]$, we also have to estimate the overall influence of this term. For that we consider the recurrence for $c(n)$ again, but replace $t(n)$ (entirely) by $C \cdot n^{1-\varepsilon}$. If $\varepsilon > 0$,

$\int_0^1 z^{1-\varepsilon} w(z)\,dz < \int_0^1 w(z)\,dz = 1$, so we still find $H > 0$ and apply case 1 of the CMT. The overall contribution of the error term is then $\mathcal{O}(n^{1-\varepsilon})$. For $\varepsilon = 1$, we have $H = 0$ and case 2 applies, giving an overall error term of $\mathcal{O}(\log n)$.

This completes the proof of Theorem 4.8. □

9.6. Transfer theorem for the variance

In this section, we give the detailed computations for the transfer theorem for the variance of median-of-k QuickXsort.

Proof of Theorem 4.10: We begin by deriving a recurrence for the variance, which we then solve using the CMT. The overall plan is similar as for Theorem 4.8, but the computations become more involved.

9.6.1. Distributional recurrence

We can precisely characterize the distribution of the random number of comparisons, C_n, that we need to sort an input of size n. We will generally denote the random variables by capital letter C_n and their expectations by lowercase letters $c(n)$. We describe the distribution of C_n in the form of a *distributional recurrence,* i.e., a recursive description of the distribution of the family of random variables $(C_n)_{n\in\mathbb{N}}$. From these, we can mechanically derive recurrence equations for the moments of the distribution and in particular for the variance. We have

$$C_n \overset{\mathcal{D}}{=} \underbrace{n - k + s(k) \;+\; A_1 \cdot X_{J_2} + A_2 \cdot \tilde{X}_{J_1}}_{T_n} + A_1 \cdot C_{J_1} + A_2 \cdot \tilde{C}_{J_2}, \qquad (n > w) \qquad (23)$$

for $(X_n)_{n\in\mathbb{N}}$ the family of random variables given by the number of comparisons to sort a random permutation of n elements with X. $(\tilde{C}_n)_{n\in\mathbb{N}}$ and $(\tilde{X}_n)_{n\in\mathbb{N}}$ are independent copies of $(C_n)_{n\in\mathbb{N}}$ and $(X_n)_{n\in\mathbb{N}}$, respectively, and these are also independent of (J_1, J_2); we will in the following omit the tildes for legibility; we implicitly define all terms in an equation from the same family as each coming from its own independent copy. Base cases for small n are given by the recursion-stopper method and are immaterial for the asymptotic regime (for constant w).

9.6.2. Recurrence for the second moment

We start with the elementary equation $\mathrm{Var}[C_n] = \mathbb{E}[C_n^2] - \mathbb{E}[C_n]^2$. Of course, $\mathbb{E}[C_n]^2 = c^2(n)$, which we already know by Theorem 4.8. From the distributional recurrence, we can compute the second moment $m_2(n) = \mathbb{E}[C_n^2]$ as follows: Square both sides in Equation (23) and take expectations; that leaves $m_2(n)$ on the left-hand side. To simplify the right-hand side, we use the law of total expectation to first take expectations conditional on J_1 (which also fixes

$J_2 = n - 1 - J_1$) and then take expectations over J_1. We find

$$\begin{aligned}\mathbb{E}[C_n^2 \mid J_1] &= \mathbb{E}\Big[\Big(T_n + \textstyle\sum_{r=1}^{2} A_r C_{J_r}\Big)^2 \;\Big|\; J_1\Big] \\ &= \mathbb{E}[T_n^2 \mid J_1] + \sum_{r=1}^{2} \mathbb{E}\Big[\underbrace{A_r^2}_{=A_r} C_{J_r}^2 \;\Big|\; J_1\Big] \\ &\quad + \mathbb{E}\Big[2\underbrace{A_1 A_2}_{=0} C_{J_1} C_{J_2} \;\Big|\; J_1\Big] + \sum_{r=1}^{2} \mathbb{E}[2T_n \cdot A_r C_{J_r} \mid J_1]\end{aligned}$$

since A_1 and A_2 are fully determined by J_1; as T_n and C_{J_r} are *conditionally independent* given J_1, this is

$$= \mathbb{E}[T_n^2 \mid J_1] + \sum_{r=1}^{2} A_r\, m_2(J_r) + 2\mathbb{E}[T_n \mid J_1] \sum_{r=1}^{2} A_r\, c(J_r).$$

We now take expected values also w.r.t. J_1 and exploit symmetries $J_1 \stackrel{\mathcal{D}}{=} J_2$. We will write $A := A_1$ and $J := J_1$; we find

$$m_2(n) = 2\mathbb{E}[A\, m_2(J)] + \underbrace{\mathbb{E}[T_n^2] + 2\sum_{r=1}^{2} \mathbb{E}_J\Big[A_r\, \mathbb{E}[T_n \mid J_1]\, c(J_r)\Big]}_{t_{m_2}(n)}.$$

To continue, we have to unfold $t_{m_2}(n)$ a bit more. We start with the simplest one, the conditional expectation of T_n. For constant k, we find

$$\begin{aligned}\mathbb{E}[T_n \mid J] &= \mathbb{E}\Big[n \pm \mathcal{O}(1) + \textstyle\sum_{r=1}^{2}(1 - A_r) X_{J_r} \;\Big|\; J\Big] \\ &= n + \sum_{r=1}^{2}(1 - A_r)\mathbb{E}[X_{J_r} \mid J] \;\pm\; \mathcal{O}(1) \\ &= n + \sum_{r=1}^{2}(1 - A_r) x(J_r) \;\pm\; \mathcal{O}(1).\end{aligned}$$

So we find for the last term in the equation for $m_2(n)$

$$\begin{aligned}&2\sum_{r=1}^{2} \mathbb{E}_J\Big[A_r\, \mathbb{E}[T_n \mid J_1]\, c(J_r)\Big] \\ &= 2\sum_{r=1}^{2} \mathbb{E}_J\Big[A_r\Big(n + \textstyle\sum_{\ell=1}^{2}(1 - A_\ell) x(J_\ell) \pm \mathcal{O}(1)\Big)\, c(J_r)\Big] \\ &= 2n\sum_{r=1}^{2} \mathbb{E}[A_r\, c(J_r)] + 2\sum_{r=1}^{2} \mathbb{E}[A_r\, x(J_{3-r})\, c(J_r)] \;\pm\; \mathcal{O}(n \log n) \\ &= 4n\mathbb{E}[A\, c(J)] + 4\mathbb{E}[A\, c(J)\, x(n - 1 - J)] \;\pm\; \mathcal{O}(n \log n).\end{aligned}$$

It remains to compute the second moment of T_n:

$$\begin{aligned}\mathbb{E}[T_n^2] &= \mathbb{E}\Big[\Big(n(1\pm\mathcal{O}(n^{-1})) + \textstyle\sum_{r=1}^{2}(1-A_r)X_{J_r}\Big)^2\Big] \\ &= \sum_{r=1}^{2}\mathbb{E}_{J_r}\Big[(1-A_r)\mathbb{E}[X_{J_r}^2 \mid J_r]\Big] + n^2(1\pm\mathcal{O}(n^{-1})) \\ &\quad + 2n(1\pm\mathcal{O}(n^{-1}))2\mathbb{E}[(1-A)x(J)],\end{aligned}$$

denoting $\mathrm{Var}[X_n]$ by $v_X(n) = \Theta(n)$ and using $\mathbb{E}[X^2] = \mathbb{E}[X]^2 + \mathrm{Var}[X]$

$$\begin{aligned}&= 2\mathbb{E}\Big[(1-A)(x^2(J)+v_X(J))\Big] + n^2 \;\pm\; \mathcal{O}(n\log n) \\ &= 2\mathbb{E}\Big[(1-A)x^2(J)\Big] + 2a_v\mathbb{E}[(1-A)J^2] + 4n\mathbb{E}[(1-A)x(J)] + n^2 \;\pm\; \mathcal{O}(n^{2-\varepsilon}).\end{aligned}$$

We can see here that the variance of X only influences lower order terms of the variance of QuickXsort when $v_X(n) = o(n^2)$.

9.6.3. Recurrence for the variance

We now have all ingredients together to compute an asymptotic solution of the recurrence for $m_2(n)$, the second moment of costs for QuickMergesort. However, it is more economical to first subtract $c^2(n)$ on the level of recurrences, since many terms will cancel. We thus derive a direct recurrence for $v(n) = \mathrm{Var}[C_n]$.

$$\begin{aligned}v(n) &= m_2(n) - c^2(n) \\ &= 2\mathbb{E}[A\,v(J)] + \underbrace{2\mathbb{E}[A\,c^2(J)] - c^2(n) + t_{m_2}(n)}_{t_v(n)}. \qquad (24)\end{aligned}$$

For brevity, we write $\overline{J} = n-1-J$. Inserting $c(n) = x(n) + qn \pm \mathcal{O}(n^\delta)$ for a $\delta < 1$, we find

$$\begin{aligned}t_v(n) &= 2\mathbb{E}\big[x^2(J)\big] + 4\mathbb{E}\Big[Ax(J)x(\overline{J})\Big] - x^2(n) \\ &\quad + 4n\mathbb{E}[x(J)] + 4q\mathbb{E}\big[AJ\big(x(J)+x(\overline{J})\big)\big] - 2qx(n)n \\ &\quad + n^2 + 2q^2\mathbb{E}\big[AJ^2\big] + 2a_v\mathbb{E}[(1-A)J^2] + 4qn\mathbb{E}[AJ] - q^2n^2 \;\pm\; \mathcal{O}(n^{2-\varepsilon}\log n).\end{aligned}$$

At this point, the only route to make progress seems to be to expand all occurrences of x into $x(n) = an\lg n + bn + \mathcal{O}(n^{1-\varepsilon})$ and compute the expectations. For that, we use the approximation by incomplete beta integrals that we introduced in Section 9.5.1 to compute the expectations of the form $\mathbb{E}[g(J)]$, where g only depends on J. All arising functions g are Hölder-continuous in $z \in [0,1]$, and the same arguments as in Lemma 4.9 apply; see also Remark 9.5. The expression for $t_v(n)$ is too big to state in full, but it can easily be computed for given values of t by computer algebra. We provide a Mathematica notebook for this step as supplementary material [54], and list numeric approximations for small sample sizes in Table 3 (page 22).

9.6.4. Solving the recurrence

Although the expression for $t_v(n)$ contains terms of order $n^2\lg^2 n$ and $n^2\lg n$, in all examined cases, these higher-order terms canceled and left $t_v(n) \sim cn^2$ for an explicitly computable

constant $c > 0$. We conjecture that this is always the case, but we did not find a simple proof. We therefore need the technical assumption that indeed $t_v(n) = \Theta(n^2)$. Under that assumption, we obtain an asymptotic approximation for $v(n)$ from Equation (24) using the CMT (Theorem B.4) with $\sigma = 2$ and $\tau = 0$. Note that the shape function $w(z)$ of the recurrence is the same as for the expected costs (see Section 9.5.3). We thus compute

$$\begin{aligned} H &= 1 - \int_0^1 z^2\, w(z)\, dz \\ &= 1 - \int_0^1 2 \left[\!\left[\tfrac{\alpha}{1+\alpha} < z < \tfrac{1}{2} \ \vee\ z > \tfrac{1}{1+\alpha} \right]\!\right] \frac{z^{t+2}(1-z)^t}{\mathrm{B}(t+1,t+1)}\, dz \\ &= 1 - 2\frac{(t+1)^{\overline{2}}}{(k+1)^{\overline{2}}} \int_0^1 \left[\!\left[\tfrac{\alpha}{1+\alpha} < z < \tfrac{1}{2} \ \vee\ z > \tfrac{1}{1+\alpha} \right]\!\right] \frac{z^{t+2}(1-z)^t}{\mathrm{B}(t+3,t+1)}\, dz \\ &= 1 - \frac{t+2}{k+2}\Big(I_{\frac{\alpha}{1+\alpha},\frac{1}{2}}(t+3,t+1) + I_{\frac{1}{1+\alpha},1}(t+3,t+1) \Big). \end{aligned} \tag{25}$$

Since $\frac{t+2}{k+2} \le \frac{2}{3}$ and the integral over the entire unit interval would be exactly 1, we have $H > 0$ for all α and t. So by Case 1 of the CMT, the variance of QuickXsort is

$$v(n) \ \sim\ \frac{t_v(n)}{H}.$$

In particular, it is quadratic in n with a computable coefficient. That concludes the proof of Theorem 4.10. □

9.7. Base cases for QuickMergesort

This section contains the proofs of Section 6: we prove the general transfer theorem for base cases of Mergesort and analyze the average cases of Insertionsort and MergeInsertion.

Proof of Theorem 6.1: Since Z uses $z(n) = n\lg n + (b \pm \varepsilon)n + o(n)$ comparisons on average, for every $\delta > 0$ we have $|z(n) - (n\lg n + bn)| \le (\varepsilon + \delta)\cdot n$ for n large enough. Let $k \ge 6$ be large enough such that this bound is satisfied for all $k/2 \le n \le k$ and let $x_k(m)$ denote the average case number of comparisons of Mergesort with base cases of size k sorted with Z, i.e., $x_k(n) = z(n)$ for $n \le k$.

We will show by induction that

$$|x_k(n) - (n\lg n + bn)| \ \le\ \left(\varepsilon + \delta + \frac{8}{k}\right)\cdot n - 4 \ =:\ e_k(n)$$

for $n \ge k/2$.

For $k/2 \le n \le k$ this holds by hypothesis, so assume that $n > k$. We have

$$x_k(n) = x_k(\lceil n/2 \rceil) + x_k(\lfloor n/2 \rfloor) + n - \eta(n)$$

for some η with $1 \leq \eta(n) \leq 2$ for all n (see, e.g., [17, p. 676]). It follows that

$$\begin{aligned}
|x_k(n) - (n\lg n + bn)| &= \Big|x_k(\lceil n/2\rceil) + x_k(\lfloor n/2\rfloor) + n - \eta(n) - (n\lg n + bn)\Big| \\
&\underset{\text{[inductive hypothesis]}}{\leq} e_k(\lceil n/2\rceil) + e_k(\lfloor n/2\rfloor) + \Big|\lceil n/2\rceil(\lg\lceil n/2\rceil + b) \\
&\qquad + \lfloor n/2\rfloor(\lg\lfloor n/2\rfloor + b) + n - \eta(n) - (n\lg n + bn)\Big| \\
&\leq e_k(n) - 4 + \Big|\lceil n/2\rceil(\lg(n/2) + b) \\
&\qquad + \lfloor n/2\rfloor(\lg(n/2) + b) + n - \eta(n) - (n\lg n + bn)\Big| + 2 \\
&\leq e_k(n) - 2 + \eta(n) \;\leq\; e_k(n)
\end{aligned}$$

Notice here that $\lg\lceil n/2\rceil - \lg(n/2) \leq \frac{1}{\ln(2)\cdot(n+1)} \leq \frac{2}{n}$. This can be easily seen by the series expansion of the logarithm. By choosing k, the base case size for sorting n elements (e.g. $k = \lg n$), the theorem follows. □

Proof of Proposition 6.3: First, we take a look at the average number of comparisons $x_{\text{Ins}}(k)$ to insert one element into a sorted array of $k-1$ elements by binary insertion. To insert a new element into $k-1$ elements either needs $\lceil\lg k\rceil - 1$ or $\lceil\lg k\rceil$ comparisons. There are k positions where the element to be inserted can end up, each of which is equally likely. For $2^{\lceil\lg k\rceil} - k$ of these positions $\lceil\lg k\rceil - 1$ comparisons are needed. For the other $k - (2^{\lceil\lg k\rceil} - k) = 2k - 2^{\lceil\lg k\rceil}$ positions $\lceil\lg k\rceil$ comparisons are needed. This means

$$\begin{aligned}
x_{\text{Ins}}(k) &= \frac{(2^{\lceil\lg k\rceil} - k)\cdot(\lceil\lg k\rceil - 1) + (2k - 2^{\lceil\lg k\rceil})\cdot\lceil\lg k\rceil}{k} \\
&= \lceil\lg k\rceil + 1 - \frac{2^{\lceil\lg k\rceil}}{k}
\end{aligned}$$

comparisons are needed on average. We obtain for the average case for sorting n elements:

$$\begin{aligned}
x_{\text{InsSort}}(n) &= \sum_{k=1}^{n} x_{\text{Ins}}(k) = \sum_{k=1}^{n}\left(\lceil\lg k\rceil + 1 - \frac{2^{\lceil\lg k\rceil}}{k}\right) \\
&\underset{\text{[32, 5.3.1–(3)]}}{=} n\cdot\lceil\lg n\rceil - 2^{\lceil\lg n\rceil} + 1 + n - \sum_{k=1}^{n}\frac{2^{\lceil\lg k\rceil}}{k}.
\end{aligned}$$

We examine the last sum separately. As before we write $H_n = \sum_{k=1}^{n}\frac{1}{k} = \ln n + \gamma \pm \mathcal{O}(\frac{1}{n})$ for the harmonic numbers where $\gamma \in \mathbb{R}$ is Euler's constant.

$$\begin{aligned}
\sum_{k=1}^{n}\frac{2^{\lceil\lg k\rceil}}{k} &= 1 + \sum_{i=0}^{\lceil\lg n\rceil-2}\sum_{\ell=1}^{2^i}\frac{2^{i+1}}{2^i+\ell} \;+ \sum_{\ell=2^{\lceil\lg n\rceil-1}+1}^{n}\frac{2^{\lceil\lg n\rceil}}{\ell} \\
&= 1 + \left(\sum_{i=0}^{\lceil\lg n\rceil-2} 2^{i+1}\cdot\Big(H_{2^{i+1}} - H_{2^i}\Big)\right) + 2^{\lceil\lg n\rceil}\cdot(H_n - H_{2^{\lceil\lg n\rceil-1}}) \\
&= \sum_{i=0}^{\lceil\lg n\rceil-2} 2^{i+1}\cdot\Big(\ln\Big(2^{i+1}\Big) + \gamma - \ln\Big(2^i\Big) - \gamma\Big) \\
&\qquad + \Big(\ln(n) + \gamma - \ln\big(2^{\lceil\lg n\rceil-1}\big) - \gamma\Big)\cdot 2^{\lceil\lg n\rceil} \;\pm\; \mathcal{O}(\log n)
\end{aligned}$$

$$\begin{aligned} &= \ln 2 \cdot \sum_{i=0}^{\lceil \lg n \rceil - 2} 2^{i+1} + \Big(\lg(n) \cdot \ln 2 - (\lceil \lg n \rceil - 1) \cdot \ln 2\Big) \cdot 2^{\lceil \lg n \rceil} \pm \mathcal{O}(\log n) \\ &= \ln 2 \cdot \Big(2 \cdot (2^{\lceil \lg n \rceil - 1} - 1) + (\lg n - \lceil \lg n \rceil + 1) \cdot 2^{\lceil \lg n \rceil}\Big) \pm \mathcal{O}(\log n) \\ &= \ln 2 \cdot (2 + \lg n - \lceil \lg n \rceil) \cdot 2^{\lceil \lg n \rceil} \pm \mathcal{O}(\log n). \end{aligned}$$

The error term of $\mathcal{O}(\log n)$ is due to the fact that for any C we have $\sum_{i=0}^{\lceil \lg n \rceil - 2} 2^{i+1} \cdot \frac{C}{2^i} = 2C(\lceil \lg n \rceil - 2)$. Hence, we have

$$x_{\text{InsSort}}(n) = n \cdot \lceil \lg n \rceil - 2^{\lceil \lg n \rceil} + n - \ln 2 \cdot (2 + \lg n - \lceil \lg n \rceil) \cdot 2^{\lceil \lg n \rceil} + \mathcal{O}(\log n). \quad \square$$

Proof of Theorem 6.5: According to Knuth [32], MergeInsertion requires at most $W(n) = n \lg n - (3 - \lg 3)n + n(y + 1 - 2^y) + \mathcal{O}(\log n)$ comparisons in the worst case, where $y = y(n) = \lceil \lg(3n/4) \rceil - \lg(3n/4) \in [0, 1)$. In the following we want to analyze the average savings relative to the worst case. We use the simplified version meaning that the average differs from the worst case only for the insertion of the elements of the last block (in every level of recursion). Therefore, let $F(n)$ denote the average number of comparisons of the insertion steps of MergeInsertion, i. e., all comparisons minus the number of comparisons $P(n)$ for forming pairs (during all recursion steps). It is easy to see that $P(n) = n - \mathcal{O}(\log n)$ (indeed, $P(n) = n - 1$ if n is a power of two); moreover, it is independent of the actual input permutation. We obtain the recurrence relation

$$\begin{aligned} F(n) &= F(\lfloor n/2 \rfloor) + G(\lceil n/2 \rceil), \qquad \text{with} \\ G(m) &= (k_m - \alpha_m) \cdot (m - t_{k_m - 1}) + \sum_{j=1}^{k_m - 1} j \cdot (t_j - t_{j-1}), \end{aligned}$$

with k_m such that $t_{k_m - 1} \le m < t_{k_m}$ and some $\alpha_m \in [0, 1]$ (recall that $t_k = (2^{k+1} + (-1)^k)/3$). As we do not analyze the improved version of the algorithm, the insertion of elements with index less or equal $t_{k_m - 1}$ requires always the same number of comparisons. Thus, the term $\sum_{j=1}^{k_m - 1} j \cdot (t_j - t_{j-1})$ is independent of the data. However, inserting an element after $t_{k_m - 1}$ may either need k_m or $k_m - 1$ comparisons. This is where α_m comes from. Note that α_m only depends on m. We split $F(n)$ into $F'(n) + F''(n)$ with

$$\begin{aligned} F'(n) &= F'(\lfloor n/2 \rfloor) + G'(\lceil n/2 \rceil) && \text{and} \\ G'(m) &= (k_m - \alpha_m) \cdot (m - t_{k_m - 1}) && \text{with } k_m \text{ such that } t_{k_m - 1} \le m < t_{k_m}, \end{aligned}$$

and

$$\begin{aligned} F''(n) &= F''(\lfloor n/2 \rfloor) + G''(\lceil n/2 \rceil) && \text{and} \\ G''(m) &= \sum_{j=1}^{k_m - 1} j \cdot (t_j - t_{j-1}) && \text{with } k_m \text{ such that } t_{k_m - 1} \le m < t_{k_m}. \end{aligned}$$

For the average case analysis, we have that $F''(n)$ is independent of the data. For $n \approx (4/3) \cdot 2^k$ we have $G'(n) \approx 0$, and hence, $F'(n) \approx 0$. Since otherwise $G'(n)$ is positive, this shows that approximately for $n \approx (4/3) \cdot 2^k$ the average case matches the worst case and otherwise it is better.

Now, we have to estimate $F'(n)$ for arbitrary n. We have to consider the calls to binary insertion more closely. To insert a new element into an array of $m-1$ elements either needs $\lceil \lg m \rceil - 1$ or $\lceil \lg m \rceil$ comparisons. For a moment assume that the element is inserted at every position with the same probability. Under this assumption the analysis in the proof of Proposition 6.3 is valid, which states that

$$x_{\mathrm{Ins}}(m) \;=\; \lceil \lg m \rceil + 1 - \frac{2^{\lceil \lg m \rceil}}{m}$$

comparisons are needed on average.

The problem is that in our case, the probability at which position an element is inserted is not uniformly distributed. However, it is monotonically decreasing with the index in the array (indices as in the description in Section 6.2). Informally speaking, this is because if an element is inserted further to the left, then for the following elements there are more possibilities to be inserted than if the element is inserted on the right.

Now, *binary-insert* can be implemented such that for an odd number of positions the next comparison is made such that the larger half of the array is the one containing the positions with lower probabilities. (In our case, this is the part with the higher indices.) That means the less likely positions lie on longer paths in the search tree, and hence, the average path length is better than in the uniform case. Therefore, we may assume a uniform distribution as an upper bound in the following.

In each of the recursion steps we have $\lceil n/2 \rceil - t_{k_{\lceil n/2 \rceil}-1}$ calls to binary insertion into sets of size $\lceil n/2 \rceil + t_{k_{\lceil n/2 \rceil}-1} - 1$ elements each where as before $t_{k_{\lceil n/2 \rceil}-1} \le \lceil n/2 \rceil < t_{k_{\lceil n/2 \rceil}}$. We write $u_{\lceil n/2 \rceil} = t_{k_{\lceil n/2 \rceil}-1}$. Hence, for inserting one element, the difference between the average and the worst case is

$$\frac{2^{\lceil \lg(\lceil n/2 \rceil + u_{\lceil n/2 \rceil}) \rceil}}{\lceil n/2 \rceil + u_{\lceil n/2 \rceil}} - 1.$$

Summing up, we obtain the following recurrence for the average savings $S(n) = W(n) - (F(n) + P(n)))$ over the worst case number $W(n)$ (recall that $P(n)$ is the number of comparisons for forming pairs)

$$S(n) \;\ge\; S(\lfloor n/2 \rfloor) + (\lceil n/2 \rceil - u_{\lceil n/2 \rceil}) \cdot \left(\frac{2^{\lceil \lg(\lceil n/2 \rceil + u_{\lceil n/2 \rceil}) \rceil}}{\lceil n/2 \rceil + u_{\lceil n/2 \rceil}} - 1 \right).$$

For $m \in \mathbb{R}_{>0}$ we write $m = 2^{\ell_m - \lg 3 + x}$ with $\ell_m \in \mathbb{Z}$ and $x \in [0,1)$ and we set

$$f(m) \;=\; (m - 2^{\ell_m - \lg 3}) \cdot \left(\frac{2^{\ell_m}}{m + 2^{\ell_m - \lg 3}} - 1 \right).$$

Recall that we have $t_k = (2^{k+1} + (-1)^k)/3$ meaning that $k_m - 1$ is the largest exponent such that $2^{k_m - \log 3} + (-1)^k/3 \le m$. Therefore, $u_m = 2^{\ell_m - \lg 3}$ and $k_m - 1 = \ell_m$ except for the case $m = t_k$ for some odd $k \in \mathbb{Z}$. Assume $m \ne t_k$ for any odd $k \in \mathbb{Z}$; then we have

$$\lceil \lg(m + u_m) \rceil \;=\; \left\lceil \lg(2^{\ell_m - \lg 3 + x} + 2^{\ell_m - \lg 3}) \right\rceil \;=\; \ell_m + \lceil \lg((2^x + 1)/3)) \rceil \;=\; \ell_m$$

and, hence, $f(m) = (m - u_m) \cdot \left(\frac{2^{\lceil \lg(m + u_m) \rceil}}{m + u_m} - 1 \right)$. On the other hand, if $m = t_k$ for some odd $k \in \mathbb{Z}$, we have $k_m = \ell_m$ and

$$f(t_k) \;\le\; t_k \cdot \left(\frac{2^k}{t_k + 2^k/3} - 1 \right) \;=\; t_k \cdot \left(\frac{3 \cdot 2^k}{2^{k+1} - 1 + 2^k} - 1 \right) \;=\; \frac{t_k}{3 \cdot 2^k - 1} \;\le\; 1.$$

Altogether this implies that $f(m)$ and $(m-u_m)\cdot\left(\frac{2^{\lceil \lg(m+u_m)\rceil}}{m+u_m}-1\right)$ differ by at most some constant (as before $u_m = t_{k_m-1}$). Furthermore, $f(m)$ and $f(m+1/2)$ differ by at most a constant. Hence, we have:

$$S(n) \;\geq\; S(n/2) + f(n/2) \pm \mathcal{O}(1).$$

Since we have $f(n/2) = f(n)/2$, this resolves to

$$S(n) \;\geq\; \sum_{i>0} f(n/2^i) \pm \mathcal{O}(\log n) \;=\; \sum_{i>0} f(n)/2^i \pm \mathcal{O}(\log n) \;=\; f(n) \pm \mathcal{O}(\log n).$$

With $n = 2^{k-\lg 3+x}$ this means

$$\begin{aligned}\frac{S(n)}{n} &= \frac{2^{k-\lg 3+x} - 2^{k-\lg 3}}{2^{k-\lg 3+x}} \cdot \left(\frac{2^k}{2^{k-\lg 3+x}+2^{k-\lg 3}} - 1\right) \pm \mathcal{O}(\log n/n)\\ &= (1-2^{-x})\cdot\left(\frac{3}{2^x+1}-1\right) \pm \mathcal{O}(\log n/n).\end{aligned}$$

Recall that we wish to compute $F(n)+P(n) \leq W(n) - S(n)$. Writing $F(n)+P(n) = n\lg n - c(n)\cdot n$ with $c(n)\in\mathcal{O}(1)$, we obtain with [32, 5.3.1 Ex. 15]

$$c(n) \;\geq\; -(F(n)-n\lg n)/n \;=\; (3-\lg 3) - (y+1-2^y) + S(n)/n,$$

where $y = \lceil \lg(3n/4)\rceil - \lg(3n/4) \in [0,1)$, i. e., $n = 2^{\ell-\lg 3-y}$ for some $\ell\in\mathbb{Z}$. With $y = 1-x$ it follows

$$c(n) \;\geq\; (3-\lg 3) - (1-x+1-2^{1-x}) + (1-2^{-x})\cdot\left(\frac{3}{2^x+1}-1\right) \;>\; 1.3999. \qquad \square$$

Acknowledgments

We thank our anonymous reviewers for their thoughtful comments which significantly helped improving the presentation.

References

[1] Manuel Blum, Robert W. Floyd, Vaughan R. Pratt, Ronald L. Rivest, and Robert E. Tarjan. Time bounds for selection. *Journal of Computer and System Sciences*, 7(4):448–461, 1973.

[2] Hans-Juergen Boehm, Russell R. Atkinson, and Michael F. Plass. Ropes: An alternative to strings. *Softw., Pract. Exper.*, 25(12):1315–1330, 1995. doi:10.1002/spe.4380251203.

[3] D. Cantone and G. Cincotti. Quickheapsort, an efficient mix of classical sorting algorithms. *Theoretical Computer Science*, 285(1):25–42, August 2002. doi:10.1016/S0304-3975(01)00288-2.

[4] Volker Diekert and Armin Weiß. QuickHeapsort: Modifications and improved analysis. *Theory of Computing Systems*, 59(2):209–230, August 2016. doi:10.1007/s00224-015-9656-y.

[5] NIST Digital Library of Mathematical Functions. Release 1.0.10; Release date 2015-08-07. URL: http://dlmf.nist.gov.

[6] Ernst E. Doberkat. An average case analysis of Floyd's algorithm to construct heaps. *Information and Control*, 61(2):114–131, May 1984. doi:10.1016/S0019-9958(84)80053-4.

[7] Ronald D. Dutton. Weak-heap sort. *BIT*, 33(3):372–381, 1993.

[8] S. Edelkamp and P. Stiegeler. 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. doi:10.1145/944618.944623.

[9] Stefan Edelkamp and Ingo Wegener. On the performance of Weak-Heapsort. In *Symposium on Theoretical Aspects of Computer Science (STACS) 2000*, volume 1770, pages 254–266. Springer-Verlag, 2000. doi:10.1007/3-540-46541-3_21.

[10] Stefan Edelkamp and Armin Weiß. QuickXsort: Efficient Sorting with $n \log n - 1.399n + o(n)$ Comparisons on Average, 2013. arXiv:1307.3033.

[11] Stefan Edelkamp and Armin Weiß. QuickXsort: Efficient sorting with $n \log n - 1.399n + o(n)$ comparisons on average. In *International Computer Science Symposium in Russia*, pages 139–152. Springer, 2014. doi:10.1007/978-3-319-06686-8_11.

[12] Stefan Edelkamp and Armin Weiß. BlockQuicksort: Avoiding branch mispredictions in Quicksort. In P. Sankowski and C. D. Zaroliagis, editors, *European Symposium on Algorithms (ESA) 2016*, volume 57 of *LIPIcs*, pages 38:1–38:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016. doi:10.4230/LIPIcs.ESA.2016.38.

[13] Stefan Edelkamp and Armin Weiß. QuickMergesort: Practically efficient constant-factor optimal sorting, 2018. arXiv:1804.10062.

[14] Stefan Edelkamp and Armin Weiß. Worst-case efficient sorting with quickmergesort. In *Proceedings of the Twenty-First Workshop on Algorithm Engineering and Experiments, ALENEX 2019, San Diego, CA, USA, January 7-8, 2019.*, pages 1–14, 2019. doi:10.1137/1.9781611975499.1.

[15] Stefan Edelkamp, Armin Weiß, and Sebastian Wild. Quickxsort - A fast sorting scheme in theory and practice. *CoRR*, abs/1811.01259, 2018. URL: http://arxiv.org/abs/1811.01259, arXiv:1811.01259.

[16] Amr Elmasry, Jyrki Katajainen, and Max Stenmark. Branch mispredictions don't affect mergesort. In *International Symposium on Experimental Algorithms (SEA) 2012*, pages 160–171, 2012. doi:10.1007/978-3-642-30850-5_15.

[17] Philippe Flajolet and Mordecai Golin. Mellin transforms and asymptotics. *Acta Informatica*, 31(7):673–696, July 1994. doi:10.1007/BF01177551.

[18] Jr. Ford, Lester R. and Selmer M. Johnson. A tournament problem. *The American Mathematical Monthly*, 66(5):pp. 387–389, 1959. URL: http://www.jstor.org/stable/2308750.

[19] Lester R. Ford and Selmer M. Johnson. A tournament problem. *The American Mathematical Monthly*, 66(5):387, May 1959. doi:10.2307/2308750.

[20] Viliam Geffert, Jyrki Katajainen, and Tomi Pasanen. Asymptotically efficient in-place merging. *Theor. Comput. Sci.*, 237(1-2):159–181, 2000. URL: https://doi.org/10.1016/S0304-3975(98)00162-5, doi:10.1016/S0304-3975(98)00162-5.

[21] Mordecai J. Golin and Robert Sedgewick. Queue-mergesort. *Information Processing Letters*, 48(5):253–259, December 1993. doi:10.1016/0020-0190(93)90088-q.

[22] Gaston H. Gonnet and J. Ian Munro. Heaps on heaps. *SIAM Journal on Computing*, 15(4):964–971, nov 1986. doi:10.1137/0215068.

[23] Ronald L. Graham, Donald E. Knuth, and Oren Patashnik. *Concrete Mathematics: A Foundation For Computer Science*. Addison-Wesley, 1994.

[24] P. Hennequin. Combinatorial analysis of quicksort algorithm. *RAIRO - Theoretical Informatics and Applications - Informatique Théorique et Applications*, 23(3):317–333, 1989. URL: http://eudml.org/doc/92337.

[25] C. A. R. Hoare. Algorithm 65: Find. *Commun. ACM*, 4(7):321–322, July 1961. doi:10.1145/366622.366647.

[26] Hsien-Kuei Hwang. Limit theorems for mergesort. *Random Structures and Algorithms*, 8(4):319–336, July 1996. doi:10.1002/(sici)1098-2418(199607)8:4<319::aid-rsa3>3.0.co;2-0.

[27] Hsien-Kuei Hwang. Asymptotic expansions of the mergesort recurrences. *Acta Informatica*, 35(11):911–919, November 1998. doi:10.1007/s002360050147.

[28] Kazuo Iwama and Junichi Teruyama. Improved average complexity for comparison-based sorting. In Faith Ellen, Antonina Kolokolova, and Jörg-Rüdiger Sack, editors, *Workshop on Algorithms and Data Structures (WADS), Proceedings*, volume 10389 of *Lecture Notes in Computer Science*, pages 485–496. Springer, 2017. doi:10.1007/978-3-319-62127-2_41.

[29] Jyrki Katajainen. The ultimate heapsort. In *Proceedings of the Computing: The 4th Australasian Theory Symposium*, Australian Computer Science Communications, pages 87–96. Springer-Verlag Singapore Pte. Ltd., 1998. URL: http://www.diku.dk/~jyrki/Myris/Kat1998C.html.

[30] Jyrki Katajainen, Tomi Pasanen, and Jukka Teuhola. Practical in-place mergesort. *Nordic Journal of Computing*, 3(1):27–40, 1996. URL: http://www.diku.dk/~jyrki/Myris/KPT1996J.html.

[31] Pok-Son Kim and Arne Kutzner. Ratio based stable in-place merging. In Manindra Agrawal, Ding-Zhu Du, Zhenhua Duan, and Angsheng Li, editors, *Theory and Applications of Models of Computation, 5th International Conference, TAMC 2008, Xi'an, China, April 25-29, 2008. Proceedings*, volume 4978 of *Lecture Notes in Computer Science*, pages 246–257. Springer, 2008. URL: https://doi.org/10.1007/978-3-540-79228-4, doi:10.1007/978-3-540-79228-4_22.

[32] Donald E. Knuth. *The Art Of Computer Programming: Searching and Sorting*. Addison Wesley, 2nd edition, 1998.

[33] Donald E. Knuth. *Selected Papers on Analysis of Algorithms*, volume 102 of *CSLI Lecture Notes*. Center for the Study of Language and Information Publications, 2000.

[34] Hosam M. Mahmoud. *Sorting: A distribution theory*. John Wiley & Sons, 2000.

[35] Heikki Mannila and Esko Ukkonen. A simple linear-time algorithm for in situ merging. *Information Processing Letters*, 18(4):203–208, May 1984. doi:10.1016/0020-0190(84)90112-1.

[36] Conrado Martínez and Salvador Roura. Optimal sampling strategies in Quicksort and Quickselect. *SIAM Journal on Computing*, 31(3):683–705, 2001. doi:10.1137/S0097539700382108.

[37] C. J. H. McDiarmid. Concentration. In M. Habib, C. McDiarmid, J. Ramirez-Alfonsin, and B. Reed, editors, *Probabilistic Methods for Algorithmic Discrete Mathematics*, pages 195–248. Springer, Berlin, 1998.

[38] Colin J. H. McDiarmid and Bruce A. Reed. Building heaps fast. *Journal of Algorithms*, pages 352–365, 1989.

[39] Mike McFadden. WikiSort. Github repository at https://github.com/BonzaiThePenguin/WikiSort. URL: https://github.com/BonzaiThePenguin/WikiSort.

[40] David R. Musser. Introspective sorting and selection algorithms. *Software—Practice and Experience*, 27(8):983–993, 1997.

[41] Wolfgang Panny and Helmut Prodinger. Bottom-up mergesort—a detailed analysis. *Algorithmica*, 14(4):340–354, October 1995. doi:10.1007/BF01294131.

[42] Klaus Reinhardt. 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. In *International Symposium on Algorithms and Computation (ISAAC)*, pages 489–498, 1992. doi:10.1007/3-540-56279-6_101.

[43] Salvador Roura. *Divide-and-Conquer Algorithms and Data Structures*. Tesi doctoral (Ph. D. thesis, Universitat Politècnica de Catalunya, 1997.

[44] Salvador Roura. Improved master theorems for divide-and-conquer recurrences. *Journal of the ACM*, 48(2):170–205, 2001. doi:10.1145/375827.375837.

[45] Robert Sedgewick. The analysis of Quicksort programs. *Acta Informatica*, 7(4):327–355, 1977. doi:10.1007/BF00289467.

[46] Robert Sedgewick and Philippe Flajolet. *An Introduction to the Analysis of Algorithms*. Addison-Wesley-Longman, 2nd edition, 2013.

[47] Robert Sedgewick and Kevin Wayne. *Algorithms*. Addison-Wesley, 4th edition, 2011.

[48] Houshang H. Sohrab. *Basic Real Analysis*. Springer Birkhäuser, 2nd edition, 2014.

[49] Florian Stober and Armin Weiß. On the average case of MergeInsertion. In *International Workshop on Combinatorial Algorithms (IWOCA) 2019*, 2019. arXiv:1905.09656.

[50] Ingo Wegener. 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.

[51] Sebastian Wild. *Dual-Pivot Quicksort and Beyond: Analysis of Multiway Partitioning and Its Practical Potential*. Doktorarbeit (Ph.D. thesis), Technische Universität Kaiserslautern, 2016. ISBN 978-3-00-054669-3. URL: http://nbn-resolving.de/urn/resolver.pl?urn:nbn:de:hbz:386-kluedo-44682.

[52] Sebastian Wild. Average cost of QuickXsort with pivot sampling. In James Allen Fill and Mark Daniel Ward, editors, *International Conference on Probabilistic, Combinatorial and Asymptotic Methods for the Analysis of Algorithms (AofA 2018)*, LIPIcs, 2018. doi:10.4230/LIPIcs.AofA.2018.36.

[53] Sebastian Wild. Quicksort is optimal for many equal keys. In *Workshop on Analytic Algorithmics and Combinatorics (ANALCO) 2018*, pages 8–22. SIAM, January 2018. arXiv:1608.04906, doi:10.1137/1.9781611975062.2.

[54] Sebastian Wild. Supplementary mathematica notebook for variance computation. October 2018. doi:10.5281/zenodo.1463020.

Appendix

A. Notation

A.1. Generic mathematics

$\mathbb{N}$, $\mathbb{N}_0$, $\mathbb{Z}$, $\mathbb{R}$ . . . . . natural numbers $\mathbb{N} = \{1, 2, 3, \ldots\}$, $\mathbb{N}_0 = \mathbb{N} \cup \{0\}$, integers $\mathbb{Z} = \{\ldots, -2, -1, 0, 1, 2, \ldots\}$, real numbers $\mathbb{R}$.

$\mathbb{R}_{>1}$, $\mathbb{N}_{\geq 3}$ etc. . . . . restricted sets $X_{\text{pred}} = \{x \in X : x \text{ fulfills pred}\}$.

$\ln(n)$, $\lg(n)$, $\log n$ . . . . natural and binary logarithm; $\ln(n) = \log_e(n)$, $\lg(n) = \log_2(n)$. We use log for an unspecified (constant) base in $\mathcal{O}$-terms

X . . . . . to emphasize that X is a random variable it is Capitalized.

$[a, b)$ . . . . . real intervals, the end points with round parentheses are excluded, those with square brackets are included.

$[m..n]$, $[n]$ . . . . . integer intervals, $[m..n] = \{m, m+1, \ldots, n\}$; $[n] = [1..n]$.

$[\![\text{stmt}]\!]$, $[\![x = y]\!]$ . . . . . Iverson bracket, $[\![\text{stmt}]\!] = 1$ if stmt is true, $[\![\text{stmt}]\!] = 0$ otherwise.

H_n . . . . . nth harmonic number; $H_n = \sum_{i=1}^{n} 1/i$.

$x \pm y$ . . . . . x with absolute error $|y|$; formally the interval $x \pm y = [x - |y|, x + |y|]$; as with $\mathcal{O}$-terms, we use one-way equalities $z = x \pm y$ instead of $z \in x \pm y$.

$a^{\underline{b}}$, $a^{\overline{b}}$ . . . . . factorial powers; "a to the b falling resp. rising"; e.g., $x^{\underline{3}} = x(x-1)(x-2)$, $x^{\underline{-3}} = 1/((x+1)(x+2)(x+3))$.

$\binom{n}{k}$ . . . . . binomial coefficients; $\binom{n}{k} = n^{\underline{k}}/k!$.

$\mathrm{B}(\lambda, \rho)$ . . . . . for $\lambda, \rho \in \mathbb{R}_+$; the beta function, $\mathrm{B}(\lambda, \rho) = \int_0^1 z^{\lambda-1}(1-z)^{\rho-1}\,dz$; see also Equation (9) on page 42.

$I_{x,y}(\lambda, \rho)$ . . . . . the regularized incomplete beta function; $I_{x,y}(\lambda, \rho) = \int_x^y \frac{z^{\lambda-1}(1-z)^{\rho-1}}{\mathrm{B}(\lambda,\rho)}\,dz$ for $\lambda, \rho \in \mathbb{R}_+$, $0 \leq x \leq y \leq 1$.

A.2. Stochastics-related notation

$\mathbb{P}[E]$, $\mathbb{P}[X = x]$ . . . . . probability of an event E resp. probability for random variable X to attain value x.

$\mathbb{E}[X]$ . . . . . expected value of X; we write $\mathbb{E}[X \mid Y]$ for the conditional expectation of X given Y, and $\mathbb{E}_X[f(X)]$ to emphasize that expectation is taken w.r.t. random variable X.

$X \overset{\mathcal{D}}{=} Y$ . . . . . equality in distribution; X and Y have the same distribution.

$\mathcal{U}(a, b)$ . . . . . uniformly in $(a, b) \subset \mathbb{R}$ distributed random variable.

$\mathrm{Beta}(\lambda, \rho)$ . . . . . Beta distributed random variable with shape parameters $\lambda \in \mathbb{R}_{>0}$ and $\rho \in \mathbb{R}_{>0}$.

$\mathrm{Bin}(n, p)$ . . . . . binomial distributed random variable with $n \in \mathbb{N}_0$ trials and success probability $p \in [0, 1]$.

$\mathrm{BetaBin}(n, \lambda, \rho)$ . . . . . beta-binomial distributed random variable; $n \in \mathbb{N}_0$, $\lambda, \rho \in \mathbb{R}_{>0}$;

A.3. Specific notation for algorithms and analysis

n . . . . . length of the input array, i.e., the input size.

k, t . . . . . sample size $k \in \mathbb{N}_{\geq 1}$, odd; $k = 2t + 1$, $t \in \mathbb{N}_0$; we write $k(n)$ to emphasize that k might depend on n.

w . . . . . threshold for recursion, for $n \leq w$, we sort inputs by X; we require $w \geq k - 1$.

α . . . . . $\alpha \in [0, 1]$; method X may use buffer space for $\lfloor \alpha n \rfloor$ elements.

$c(n)$ . . . . . expected costs of QuickXsort; see Section 9.2.

$x(n)$, a, b . . . . . expected costs of X, $x(n) = an \lg n + bn \pm o(n)$; see Section 9.2.

J_1, J_2 . . . . . (random) subproblem sizes; $J_1 + J_2 = n - 1$; $J_1 = t + I_1$;

I_1, I_2 . . . . . (random) segment sizes in partitioning; $I_1 \stackrel{\mathcal{D}}{=} \mathrm{BetaBin}(n - k, t + 1, t + 1)$; $I_2 = n - k - I_1$; $J_1 = t + I_1$

R . . . . . (one-based) rank of the pivot; $R = J_1 + 1$.

$s(k)$ . . . . . (expected) cost for pivot sampling, i.e., cost for choosing median of k elements.

A_1, A_2, A . . . . . indicator random variables; $A_1 = [\![\text{left subproblem sorted recursively}]\!]$; see Section 9.2.

B. Mathematical Preliminaries

In this appendix, we restate some known results for the reader's convenience.

B.1. Hölder continuity

A function $f : I \to R$ defined on a bounded interval I is *Hölder-continuous* with exponent $\eta \in (0, 1]$ if

$$\exists C\, \forall x, y \in I \;:\; \big|f(x) - f(y)\big| \;\leq\; C|x - y|^\eta.$$

Hölder-continuity is a notion of smoothness that is stricter than (uniform) continuity but slightly more liberal than Lipschitz-continuity (which corresponds to $\eta = 1$). $f : [0, 1] \to \mathbb{R}$ with $f(z) = z \ln(1/z)$ is a stereotypical function that is Hölder-continuous (for any $\eta \in (0, 1)$) but not Lipschitz (see Lemma 9.1 below).

One useful consequence of Hölder-continuity is given by the following lemma: an error bound on the difference between an integral and the Riemann sum ([51, Proposition 2.12–(b)]).

Lemma B.1 (Hölder integral bound): *Let $f : [0, 1] \to \mathbb{R}$ be Hölder-continuous with exponent η. Then*

$$\int_0^1 f(x)\,dx \;=\; \frac{1}{n}\sum_{i=0}^{n-1} f(i/n) \;\pm\; \mathcal{O}(n^{-\eta}), \qquad (n \to \infty). \qquad \square$$

Remark B.2 (Properties of Hölder-continuity): *We considered only the unit interval as the domain of functions, but this is no restriction: Hölder-continuity (on bounded domains) is preserved by addition, subtraction, multiplication and composition (see, e.g., [48, Section 4.6] for details). Since any linear function is Lipschitz, the result above holds for Hölder-continuous functions $f : [a, b] \to \mathbb{R}$.*

If our functions are defined on a bounded domain, Lipschitz-continuity implies Hölder-continuity and Hölder-continuity with exponent η implies Hölder-continuity with exponent $\eta' < \eta$. A real-valued, differentiable function is Lipschitz if its derivative is bounded.

B.2. Chernoff bound

We write $X \stackrel{\mathcal{D}}{=} \mathrm{Bin}(n, p)$ if X is has a binomial distribution with $n \in \mathbb{N}_0$ trials and success probability $p \in [0, 1]$. Since X is a sum of independent random variables with bounded influence on the result, Chernoff bounds imply strong concentration results for X. We will only need a very basic variant given in the following lemma.

Lemma B.3 (Chernoff Bound, Theorem 2.1 of [37]): *Let $X \stackrel{\mathcal{D}}{=} \mathrm{Bin}(n, p)$ and $\delta \geq 0$. Then*

$$\mathbb{P}\left[\left|\frac{X}{n} - p\right| \geq \delta\right] \;\leq\; 2\exp(-2\delta^2 n). \tag{26}$$

□

B.3. Continuous Master Theorem

For solving recurrences, we build upon Roura's master theorems [44]. The relevant *continuous master theorem* is restated here for convenience:

Theorem B.4 (Roura's Continuous Master Theorem (CMT)):
Let F_n be recursively defined by

$$F_n \;=\; \begin{cases} b_n\,, & \text{for } 0 \leq n < N; \\ t_n \;+\; \displaystyle\sum_{j=0}^{n-1} w_{n,j}\, F_j, & \text{for } n \geq N\,, \end{cases} \tag{27}$$

where t_n, the toll function, satisfies $t_n \sim K n^\sigma \log^\tau(n)$ as $n \to \infty$ for constants $K \neq 0$, $\sigma \geq 0$ and $\tau > -1$. Assume there exists a function $w : [0, 1] \to \mathbb{R}_{\geq 0}$, the shape function, with $\int_0^1 w(z)dz \geq 1$ and

$$\sum_{j=0}^{n-1} \left| w_{n,j} \;-\; \int_{j/n}^{(j+1)/n} w(z)\, dz \right| \;=\; \mathcal{O}(n^{-d}), \qquad (n \to \infty), \tag{28}$$

for a constant $d > 0$. With $H := 1 - \displaystyle\int_0^1 z^\sigma w(z)\, dz$, we have the following cases:

1. *If $H > 0$, then $F_n \sim \dfrac{t_n}{H}$.*
2. *If $H = 0$, then $F_n \sim \dfrac{t_n \ln n}{\widetilde{H}}$ with $\widetilde{H} = -(\tau + 1) \displaystyle\int_0^1 z^\sigma \ln(z)\, w(z)\, dz$.*

3. If $H < 0$, then $F_n = \mathcal{O}(n^c)$ for the unique $c \in \mathbb{R}$ with $\int_0^1 z^c w(z)\,dz = 1$. □

Theorem B.4 is the "reduced form" of the CMT, which appears as Theorem 1.3.2 in Roura's doctoral thesis [43], and as Theorem 18 of [36]. The full version (Theorem 3.3 in [44]) allows us to handle sublogarithmic factors in the toll function, as well, which we do not need here.

C. Pseudocode

In this appendix, we list explicit pseudocode for the most important merging procedures.

C.1. Simple merge by swaps

Algorithm C.1 is a merging procedure with an in-place interface: upon termination, the merge result is found in the same area previously occupied by the merged runs. The method works by moving the left run into a buffer area first. More specifically, we move $A[\ell..m-1]$ into the buffer area $B[b..b+n_1-1]$ and then merge it with the second run $A[m..r]$ – still residing in the original array – into the empty slot left by the first run. By the time this first half is filled, we either have consumed enough of the second run to have space to grow the merged result, or the merging was trivial, i.e., all elements in the first run were smaller.

Algorithm C.1 Simple merging procedure that uses the buffer only by swaps.

```
SimpleMergeBySwaps(A[ℓ..r], m, B[b..e])
    // Merges runs A[ℓ..m−1] and A[m..r] in-place into A[ℓ..r] using scratch space B[b..e].
 1  n1 := m − ℓ;   n2 := r − m + 1
    // Requires A[ℓ..m−1] and A[m..r] to be sorted, n1 ≤ n2 and n1 ≤ e − b + 1.
 2  for i = 0, ..., n1 − 1
 3      Swap(A[ℓ + i], B[b + i])
 4  end for
 5  i1 := b;   i2 := m;   o := ℓ
 6  while i1 < b + n1 and i2 ≤ r
 7      if B[i1] ≤ A[i2]
 8          Swap(A[o], B[i2]);   o := o + 1;   i1 := i1 + 1
 9      else
10          Swap(A[o], A[i1]);   o := o + 1;   i2 := i2 + 1
11      end if
12  end while
13  while i1 < b + n1
14      Swap(A[o], B[i2]);   o := o + 1;   i1 := i1 + 1
15  end while
```

This method is mostly for demonstration purposes, how little changes are required to use Mergesort in QuickXsort. It is not the best solution, though.

C.2. Ping-pong Mergesort

In this section, we give detailed code for the "ping-pong" MERGESORT variant that we use in our QUICKMERGESORT implementation. It also uses $\alpha = \frac{1}{2}$, i.e., buffer space to hold half of the input. By using a special procedure, MERGESORTOUTER, for the outermost call following the strategy illustrated in Figure 3, we reduce the task to the case of $\alpha = 1$ for two subproblems. These are solved by moving elements *from* the input area `to` the output area (while sorting them), which is easily achieved by a recursive procedure (MERGESORTINNER).

Algorithm C.2 Recursive Ping-pong Mergesort with $\alpha = \frac{1}{2}$.

```
MergesortOuter(A[ℓ..r], B[b..e])
    // Sort A[ℓ..r] using B[b..e] as temporary space. Requires that e − b + 1 ≥ ⌈(r − ℓ + 1)/2⌉
 1  if ℓ < r
 2      n1 := ⌊(r − ℓ + 1)/2⌋;   n2 := (r − ℓ + 1) − n1
 3      MergesortInner(A[r − n2 + 1..r], B[b..b + n2 − 1])
 4      MergesortInner(A[ℓ..ℓ + n1 − 1], A[r − n1 + 1..r])
 5      MergeBySwapsBtoA(A[ℓ..r], r − n1 + 1, B[b..b + n2 − 1],
 6  end if

MergesortInner(A[ℓ..r], B[b..e])
    // Move the elements A[ℓ..r] to B[b..e] in sorted order; assumes r − ℓ = e − b.
 1  if ℓ = r
 2      Swap(A[ℓ], B[b])
 3  else
 4      n1 := ⌊(r − ℓ + 1)/2⌋;   n2 := (r − ℓ + 1) − n1
 5      MergesortInner(A[r − n2 + 1..r], B[e − n2 + 1..e])
 6      MergesortInner(A[ℓ..ℓ + n1 − 1], A[r − n1 + 1..r])
 7      MergeBySwapsBtoA(B[b..e], e − n2 + 1, A[r − n1 + 1..r])
 8  end if

MergeBySwapsBtoA(A[ℓ..r], m, B[b..e])
    // Merges sorted runs B[b..e] and A[m..r] in-place into A[ℓ..r]; requires e − b + 1 = m − ℓ.
 1  i1 := b;   i2 := m;   o := ℓ
 2  while i1 ≤ e and i2 ≤ r
 3      if B[i1] ≤ A[i2]
 4          Swap(A[o], B[i2]);   o := o + 1;   i1 := i1 + 1
 5      else
 6          Swap(A[o], A[i1]);   o := o + 1;   i2 := i2 + 1
 7      end if
 8  end while
 9  while i1 ≤ e
10      Swap(A[o], B[i2]);   o := o + 1;   i1 := i1 + 1
11  end while
```

C.3. Reinhardt's merge

In this section, we give the code for Reinhardt's merge [42], which allows a smaller buffer $\alpha = \frac{1}{4}$. We do not present Reinhardt's whole MERGESORT algorithm based on it but merely the actual merging routine as illustrated in Figure 4. The organization of the different calls to merging is easy but requires many tedious index calculations, which do not add much insight for the reader.

Algorithm C.3 Reinhardt's merging method allowing Mergesort with $\alpha = \frac{1}{4}$.

MERGEBYSWAPSREINHARD$(A[\ell..r], k, m)$

// Merges sorted runs $A[k..m-1]$ and $A[m..r]$ into $A[\ell..\ell+r-k]$;
// requires $(r-m+1)/2 \le k-\ell < (r-m+1)$.

```
1  i1 := k;   i2 := m;   o := ℓ
2  while i1 < m and o < i1
3      if A[i1] ≤ A[i2]
4          Swap(A[o], A[i2]);   o := o + 1;   i1 := i1 + 1
5      else
6          Swap(A[o], A[i1]);   o := o + 1;   i2 := i2 + 1
7      end if
8  end while
9  i'1 := m − 1;   i'2 := r;   o := ℓ + r − k
10 while i'1 ≥ i1 and i'2 ≥ i2
11     if A[i'1] ≥ A[i'2]
12         Swap(A[o], A[i'2]);   o := o − 1;   i'1 := i'1 − 1
13     else
14         Swap(A[o], A[i'1]);   o := o − 1;   i'2 := i'2 − 1
15     end if
16 end while
17 while i'2 ≥ i2
18     Swap(A[o], A[i'2]);   o := o − 1;   i'2 := i'2 − 1
19 end while
```