Optimal Partitioning for Dual-Pivot Quicksort

MARTIN AUMÜLLER and MARTIN DIETZFELBINGER, Technische Universität Ilmenau

Dual-pivot quicksort refers to variants of classical quicksort where in the partitioning step two pivots are used to split the input into three segments. This can be done in different ways, giving rise to different algorithms. Recently, a dual-pivot algorithm due to Yaroslavskiy received much attention, because it replaced the well-engineered quicksort algorithm in Oracle's Java 7 runtime library. Nebel and Wild (ESA 2012) analyzed this algorithm and showed that on average it uses $1.9n \ln n + O(n)$ comparisons to sort an input of size n, beating standard quicksort, which uses $2n \ln n + O(n)$ comparisons. We introduce a model that captures all dual-pivot algorithms, give a unified analysis, and identify new dual-pivot algorithms that minimize the average number of key comparisons among all possible algorithms up to a linear term. This minimum is $1.8n \ln n + O(n)$. For the case that the pivots are chosen from a small sample, we include a comparison of dual-pivot quicksort and classical quicksort. Specifically, we show that dual-pivot quicksort benefits from a skewed choice of pivots. We experimentally evaluate our algorithms and compare them to Yaroslavskiy's algorithm and the recently described 3-pivot quicksort algorithm of Kushagra et al. (ALENEX 2014).

Categories and Subject Descriptors: F.2.2 [**Nonnumerical Algorithms and Problems**]: Sorting and searching

General Terms: Algorithms

Additional Key Words and Phrases: Sorting, quicksort, dual-pivot

ACM Reference Format:
Martin Aumüller and Martin Dietzfelbinger. 2015. Optimal partitioning for dual-pivot quicksort. ACM Trans. Algorithms 12, 2, Article 18 (November 2015), 36 pages.
DOI: http://dx.doi.org/10.1145/2743020

1. INTRODUCTION

Quicksort [Hoare 1962] is a thoroughly analyzed classical sorting algorithm, described in standard textbooks [Cormen et al. 2009; Knuth 1973; Sedgewick and Flajolet 1996] and with implementations in practically all algorithm libraries. Following the divide-and-conquer paradigm, on an input consisting of n elements quicksort uses a pivot element to partition its input elements into two parts, with the elements in one part being smaller than or equal to the pivot and the elements in the other part being larger than or equal to the pivot; it then uses recursion to sort these parts. It is well known that if the input consists of n elements with distinct keys in random order and the pivot is picked by just choosing an element, then, on average, quicksort uses

A preliminary version of this work appeared in the *Proceedings of the 40th International Colloquium on Automata, Languages, and Programming (ICALP)*, Part 1, 33–44, 2013.
Authors' addresses: M. Aumüller and M. Dietzfelbinger, Fakultät für Informatik und Automatisierung, Technische Universität Ilmenau, 98683 Ilmenau, Germany; emails: {martin.aumueller, martin.dietzfelbinger}@tu-ilmenau.de.

DOI: http://dx.doi.org/10.1145/2743020

$\ldots \leq p$	p	$p \leq \ldots \leq q$	q	$\ldots \geq q$

Fig. 1. Result of the partition step in dual-pivot quicksort schemes using two pivots p, q with $p \leq q$. Elements left of p are smaller than or equal to p, elements right of q are larger than or equal to q. The elements between p and q are at least as large as p and at most as large as q.

$2n \ln n + O(n)$ comparisons.[1] In 2009, Yaroslavskiy announced[2] that he had found an improved quicksort implementation, the claim being backed by experiments. After extensive empirical studies, in 2009, Yaroslavskiy's algorithm became the new standard quicksort algorithm in Oracle's Java 7 runtime library. This algorithm employs two pivots to split the elements. If two pivots p and q with $p \leq q$ are used, the partitioning step partitions the remaining $n - 2$ elements into three parts: elements smaller than or equal to p, elements between p and q, and elements larger than or equal to q; see Figure 1.[3] Recursion is then applied to the three parts. As remarked by Wild and Nebel [2012], it came as a surprise that two pivots should help, since in his thesis [Sedgewick 1975] Sedgewick had proposed and analyzed a dual-pivot approach that was inferior to classical quicksort. Later, Hennequin in his thesis [Hennequin 1991] studied the general approach of using $k \geq 1$ pivot elements. According to Wild and Nebel [2012], he found only slight improvements that would not compensate for the more involved partitioning procedure. (See Wild and Nebel [2012] for a short discussion.)

In Wild and Nebel [2012] (see also the full version [Wild et al. 2015]), Nebel and Wild formulated and analyzed a simplified version of Yaroslavskiy's algorithm. (For completeness, this algorithm is given as Algorithm 2 in Appendix B.2.) They showed that it makes $1.9n \ln n + O(n)$ key comparisons on average, in contrast to the $2n \ln n + O(n)$ of standard quicksort and the $\frac{32}{15} n \ln n + O(n)$ of Sedgewick's dual-pivot algorithm. On the other hand, they showed that the number of swap operations in Yaroslavskiy's algorithm is $0.6n \ln n + O(n)$ on average, which is much higher than the $0.33n \ln n + O(n)$ swap operations in classical quicksort. In this article, we concentrate on the comparison count as cost measure and on asymptotic results. We leave the study of other cost measures to further investigations (which already have taken place [Martínez et al. 2015]). We consider other measures in experimental evaluations.

Wild and Nebel [2012] state that the reason for Yaroslavskiy's algorithm being superior was that his "partitioning method is able to take advantage of certain asymmetries in the outcomes of key comparisons." They also state that "[Sedgewick's dual-pivot method] fails to utilize them, even though being based on the same abstract algorithmic idea." So, the abstract algorithmic idea of using two pivots can lead to different algorithms with different behaviors. In this article, we describe the design space from which all these algorithms originate. We fully explain which simple property makes some dual-pivot algorithms perform better and some perform worse with regard to the average comparison count, and we identify optimal members (up to a linear term) of this design space. The best ones use $1.8n \ln n + O(n)$ comparisons on average—even less than Yaroslavskiy's method.

The first observation is that everything depends on the cost (i.e., the comparison count) of the partitioning step. This is not new at all. Actually, in Hennequin's thesis [Hennequin 1991], the connection between partitioning cost and overall cost for quicksort variants with more than one pivot is analyzed in detail. For dual-pivot

[1]In this article, ln denotes the natural logarithm and log denotes the logarithm to base 2.

[2]An archived version of the relevant discussion in a Java newsgroup can be found at `http://permalink.gmane.org/gmane.comp.java.openjdk.core-libs.devel/2628`. Also see Wild and Nebel [2012].

[3]In accordance with tradition, we assume in this theoretical study that all elements have different keys. Of course, in implementations equal keys are an important issue that requires a lot of care [Sedgewick 1977].

quicksort, Hennequin proved that if the (average) partitioning cost for n elements is $a \cdot n + O(1)$, for a constant a, then the average cost for sorting n elements is

$$\frac{6}{5}a \cdot n \ln n + O(n). \tag{1}$$

The partitioning cost of some algorithms presented in this article will have a non-constant lower order term. Utilizing the Continuous Master Theorem of Roura [2001], we prove that Equation (1) describes the average cost even if the partitioning cost is $a \cdot n + O(n^{1-\varepsilon})$ for some $\varepsilon > 0$. Throughout this article, all that interests us is the constant factor with the leading term. (The reader should be warned that for real-life n, the linear term, which can even be negative, can have a big influence on the average number of comparisons.)

The second observation is that the partitioning cost depends on certain details of the partitioning procedure. This is in contrast to standard quicksort with one pivot where partitioning always takes $n-1$ comparisons. In Wild and Nebel [2012], it is shown that Yaroslavskiy's partitioning procedure uses $\frac{19}{12}n + O(1)$ comparisons on average, while Sedgewick's uses $\frac{16}{9}n + O(1)$ many. The analysis of these two algorithms is based on the study of how certain pointers move through the array, at which positions elements are compared to the pivots, which of the two pivots is used for the first comparison, and how swap operations exchange two elements in the array. For understanding what is going on, however, it is helpful to forget about concrete implementations with loops in which pointers sweep across arrays and entries are swapped and instead look at partitioning with two pivots in a more abstract way. For simplicity, we shall always assume that the input is a permutation of $\{1, \ldots, n\}$. Now pivots p and q with $p < q$ are chosen. The task is to *classify* the remaining $n-2$ elements into classes "small" ($s = p-1$ many), "medium" ($m = q-p-1$ many), and "large" ($\ell = n-p$ many) by comparing these elements one after the other with the smaller pivot or the larger pivot, or both of them if necessary. Note that for symmetry reasons it is inessential in which order the elements are treated. The only choice the algorithm can make is whether to compare the current element with the smaller pivot or the larger pivot first. Let the random variable S_2 denote the number of small elements compared with the larger pivot first, and let L_2 denote the number of large elements compared with the smaller pivot first. Then the total number of comparisons is $n-2+m+S_2+L_2$.

Averaging over all inputs and all possible choices of the pivots the term $n-2+m$ will lead to $\frac{4}{3}n + O(1)$ key comparisons on average, independently of the algorithm. Let $W = S_2 + L_2$, the number of elements that are compared with the "wrong" pivot first. Then $\mathrm{E}(W)$ is the only quantity influenced by a particular partitioning procedure.

In this article, we first devise an easy method to calculate $\mathrm{E}(W)$. The result of this analysis will lead to an asymptotically optimal strategy. The basic approach is the following. Assume a partitioning procedure is given, and assume p, q and hence $s = p-1$ and $\ell = n-q$ are fixed, and let $w_{s,\ell} = \mathrm{E}(W \mid s, \ell)$. Denote the average number of elements compared to the smaller [larger] pivot first by $f^{\mathrm{p}}_{s,\ell}$ [$f^{\mathrm{q}}_{s,\ell}$]. If the elements to be classified were chosen to be small, medium, and large independently with probabilities $s/(n-2)$, $m/(n-2)$, and $\ell/(n-2)$, respectively, then the average number of small elements compared with the large pivot first would be $f^{\mathrm{q}}_{s,\ell} \cdot s/(n-2)$; similarly for the large elements. Of course, the actual input is a sequence with exactly s [m, ℓ] small [medium, large] elements, and there is no independence. Still, we will show that the randomness in the order is sufficient to guarantee that, for some $\varepsilon > 0$,

$$w_{s,\ell} = f^{\mathrm{q}}_{s,\ell} \cdot s/n + f^{\mathrm{p}}_{s,\ell} \cdot \ell/n + O(n^{1-\varepsilon}). \tag{2}$$

The details of the partitioning procedure will determine $f^{\text{p}}_{s,\ell}$ and $f^{\text{q}}_{s,\ell}$, and hence $w_{s,\ell}$ up to $O(n^{1-\varepsilon})$. This seemingly simple insight has two consequences, one for the analysis and one for the design of dual-pivot algorithms:

(i) In order to *analyze* the average comparison count of a dual-pivot algorithm (given by its partitioning procedure) up to a linear term, determine $f^{\text{p}}_{s,\ell}$ and $f^{\text{q}}_{s,\ell}$ for this partitioning procedure. This will give $w_{s,\ell}$ up to $O(n^{1-\varepsilon})$, which must then be averaged over all s, ℓ to find the average number of comparisons in partitioning. Then apply (1).

(ii) In order to *design* a good partitioning procedure with regard to the average comparison count, try to make $f^{\text{q}}_{s,\ell} \cdot s/n + f^{\text{p}}_{s,\ell} \cdot \ell/n$ small.

We demonstrate approach (i) in Section 4. An example: As explained in Wild and Nebel [2012], if s and ℓ are fixed, in Yaroslavskiy's algorithm we have $f^{\text{q}}_{s,\ell} \approx \ell$ and $f^{\text{p}}_{s,\ell} \approx s+m$. By Equation (2) we get $w_{s,\ell} = (\ell s + (s+m)\ell)/n + O(n^{1-\varepsilon})$. This must be averaged over all possible values of s and ℓ. The result is $\frac{1}{4}n + O(n^{1-\varepsilon})$, which together with $\frac{4}{3}n + O(1)$ gives $\frac{19}{12}n + O(n^{1-\varepsilon})$, close to the result from Wild and Nebel [2012].

Principle (ii) will be used to identify an asymptotically optimal partitioning procedure that makes $1.8n \ln n + O(n)$ key comparisons on average. In brief, such a strategy should achieve the following: If $s > \ell$, compare (almost) all entries with the smaller pivot first ($f^{\text{p}}_{s,\ell} \approx n$ and $f^{\text{q}}_{s,\ell} \approx 0$); otherwise, compare (almost) all entries with the larger pivot first ($f^{\text{p}}_{s,\ell} \approx 0$ and $f^{\text{q}}_{s,\ell} \approx n$). Of course, some details have to be worked out: How can the algorithm decide which case applies? In which technical sense is this strategy optimal? We shall see in Section 5 how a sampling technique resolves these issues.

In Section 6, we consider the following simple and intuitive strategy: *If more elements have been classified as being large instead of being small so far, compare the next element to the larger pivot first; otherwise, compare it to the smaller pivot first.* We will show that this strategy is optimal with regard to minimizing the average comparison count.

In implementations of quicksort, the pivot is usually chosen as the median from a small sample of $2k+1$ elements. Intuitively, this yields more balanced subproblems, which are smaller on average. This idea already appeared in Hoare's original publication [Hoare 1962] without an analysis. This analysis was later supplied by van Emden [van Emden 1970]. The complete analysis of this variant was given by Maríez and Roura [2001]. They showed that the optimal sample size is $\sqrt{n}$. For this sample size, the average comparison count of quicksort matches the lower-order bound of $n \log n + O(n)$ comparisons. In practice, one usually uses a sample of size 3. Theoretically, this decreases the average comparison count from $2n \ln n + O(n)$ to $1.714..n \ln n + O(n)$. This strategy has been generalized in the obvious way to Yaroslavskiy's algorithm as well. The implementation of Yaroslavskiy's algorithm in Oracle's Java 7 uses the two tertiles in a sample of size 5 as pivots (i.e., the elements of rank 2 and 4). In Section 7, we analyze the comparison count of dual-pivot quicksort algorithms with this sampling strategy. Yaroslavskiy's algorithm has an average comparison count of $1.704..n \ln n + O(n)$ in this case, whereas the optimal average cost is $1.623..n \ln n + O(n)$. In that section, we also consider a question raised by Wild, Nebel, and Martínez [Wild et al. 2014, Section 8] for Yaroslavskiy's algorithm, namely of which rank the pivots should be to achieve minimum sorting cost. Although using the tertiles of the input seems the obvious choice for balancing reasons, in Wild et al. [2014] it is shown that for Yaroslavskiy's algorithm this minimum is attained for ranks $\approx 0.429n$ and $\approx 0.698n$ and is $\approx 1.4931n \ln n$. We will show that the simple strategy *"Always compare with the larger* *pivot first"* achieves sorting cost $\approx 1.4427 n \ln n$ (i.e., the lower bound for comparison-based sorting) when choosing the elements of rank $n/4$ and $n/2$ as the two pivots.

As noted in Wild et al. [2013], considering only key comparisons and swap operations does not suffice for evaluating the practicability of sorting algorithms. In Section 8, we present experimental results that indicate the following: When sorting integers, the comparison-optimal algorithms of Section 5 and Section 6 are slower than Yaroslavskiy's algorithm. However, an implementation of the simple strategy *"Always compare with the larger pivot first"* performs very well both in C++ and in Java in our experimental setup. We will also compare our algorithms to the fast three-pivot quicksort algorithm described in Kushagra et al. [2014]. While comparing these algorithms, we will provide evidence that the theoretical cost measure "cache misses" described in Kushagra et al. [2014] nicely predicts empirical cache behavior and comes closest to correctly predicting running time.

We emphasize that the purpose of this article is not to arrive at better and better quicksort algorithms by using all kinds of variations, but rather to thoroughly analyze the situation with two pivots, showing the potential and the limitations of this approach.

2. BASIC APPROACH TO ANALYZING DUAL-PIVOT QUICKSORT

We assume the input sequence $(a_1, \ldots, a_n)$ to be a random permutation of $\{1, \ldots, n\}$, with each permutation occurring with probability $(1/n!)$. If $n \leq 1$, there is nothing to do; if $n = 2$, sort by one comparison. Otherwise, choose the first element a_1 and the last element a_n as the set of pivots, and set $p = \min(a_1, a_n)$ and $q = \max(a_1, a_n)$. Partition the remaining elements into elements smaller than p ("small" elements), elements between p and q ("medium" elements), and elements larger than q ("large" elements); see Figure 1. Then apply the procedure recursively to these three groups. Clearly, each pair p, q with $1 \leq p < q \leq n$ appears as set of pivots with probability $1/\binom{n}{2}$. Our cost measure is the number of key comparisons needed to sort the given input. Let C_n be the random variable counting this number. Let P_n denote the partitioning cost to partition the $n-2$ non-pivot elements into the three groups. As explained by Wild and Nebel [Wild and Nebel 2012, Appendix A], the average number of key comparisons obeys the following recurrence:

$$\mathrm{E}(C_n) = \mathrm{E}(P_n) + \frac{2}{n(n-1)} \cdot 3 \sum_{k=0}^{n-2} (n-k-1) \cdot \mathrm{E}(C_k). \tag{3}$$

If $\mathrm{E}(P_n) = a \cdot n + O(1)$, for a constant a, this can be solved (*cf.* Hennequin [1991] and Wild and Nebel [2012]) to give

$$\mathrm{E}(C_n) = \frac{6}{5} a \cdot n \ln n + O(n). \tag{4}$$

In Section 3, we show that Equation (4) also holds if $\mathrm{E}(P_n) = a \cdot n + O(n^{1-\varepsilon})$ for some $\varepsilon > 0$. For the proof, we utilize the Continuous Master Theorem from Roura [2001].

In view of this simple relation, it is sufficient to study the cost of partitioning. Abstracting from moving elements around in arrays, we arrive at the following "classification problem": Given a random permutation $(a_1, \ldots, a_n)$ of $\{1, \ldots, n\}$ as the input sequence and a_1 and a_n as the two pivots p and q, with $p < q$, classify each of the remaining $n-2$ elements as being small, medium, or large. Note that there are exactly $s := p-1$ small elements, $m := q-p-1$ medium elements, and $\ell := n-q$ large elements. Although this classification does not yield an actual partition of the input sequence, a classification algorithm can be turned into a partitioning algorithm using only swap operations but no additional key comparisons. Since elements are only compared with the two pivots, the randomness of subarrays is preserved. Thus, in the recursion, we may always assume that the input is arranged randomly.

We make the following observations (and fix notation) for all classification algorithms. One key comparison is needed to decide which of the elements a_1 and a_n is the smaller pivot p and which is the larger pivot q. For classification, each of the remaining $n-2$ elements has to be compared against p or q or both. Each *medium* element has to be compared to p and q. On average, there are $(n-2)/3$ medium elements. Let S_2 denote the number of small elements that are compared to the larger pivot first (i.e., the number of small elements that need two comparisons for classification). Analogously, let L_2 denote the number of large elements compared to the smaller pivot first. Conditioning on the pivot choices, and hence the values of s and ℓ, we may calculate $\mathrm{E}(P_n)$ as follows:[4]

$$\mathrm{E}(P_n) = (n-1) + (n-2)/3 + \frac{1}{\binom{n}{2}} \sum_{s+\ell \le n-2} \mathrm{E}(S_2 + L_2 \mid s, \ell). \tag{5}$$

We call the third summand in Equation (5) the *Additional Cost Term (ACT)* because it is the only value that depends on the actual classification algorithm.

3. ANALYZING THE ADDITIONAL COST TERM

We use the following formalization of a partitioning procedure: A *classification strategy* is given as a three-way decision tree T with a root and $n-2$ levels numbered $0, 1, \ldots, n-3$ of inner nodes as well as one leaf level. The root is on level 0. Each node v is labeled with an index $i(v) \in \{2, \ldots, n-1\}$ and an element $l(v) \in \{\mathrm{p}, \mathrm{q}\}$. If $l(v)$ is p, then at node v element $a_{i(v)}$ is compared with the smaller pivot first; otherwise (i.e., $l(v) = \mathrm{q}$), it is compared with the larger pivot first. The three edges out of a node are labeled σ, μ, λ, respectively, representing the outcome of the classification as small, medium, large, respectively. The label of edge e is called $c(e)$. On each of the 3^{n-2} paths, each index occurs exactly once. Each input determines exactly one path w from the root to a leaf in the obvious way; the classification of the elements can then be read off from the node and edge labels along this path. We call such a tree a *classification tree*.

Identifying a path π from the root to a leaf lf by the sequence of nodes and edges $(v_1, e_1, v_2, e_2, \ldots, v_{n-2}, e_{n-2}, \mathrm{lf})$ on it, we define the cost c_π as

$$c_\pi = |\{j \in \{1, \ldots, n-2\} \mid l(v_j) = \mathrm{q} \wedge c(e_j) = \sigma \text{ or } l(v_j) = \mathrm{p} \wedge c(e_j) = \lambda\}|.$$

For a given input, the cost of the path associated with this input exactly describes the number of additional comparisons on this input. An example for such a classification tree is given in Figure 2.

We let S_2^T $[L_2^T]$ denote the random variable that, for a random input, counts the number of small [large] elements classified in nodes with label q [p]. We now describe how we can calculate the ACT of a classification tree T. First consider fixed s and ℓ and let the input excepting the pivots be arranged randomly. For a node v in T, we let s_v, m_v, and ℓ_v denote the number of edges labeled σ, μ, and λ, respectively, from the root to v. By the randomness of the input, the probability that the element classified at v is "small" (i.e., that the edge labeled σ is used) is exactly $(s - s_v)/(n - 2 - \mathrm{level}(v))$. The probability that it is "medium" is $(m - m_v)/(n - 2 - \mathrm{level}(v))$, and that it is "large" is $(\ell - \ell_v)/(n - 2 - \mathrm{level}(v))$. The probability $p^v_{s,\ell}$ that node v in the tree is reached is then

[4]We use the following notation throughout this article: To indicate that sums run over all $\binom{n}{2}$ combinations (s, ℓ) with $s, \ell \ge 0$ and $s + \ell \le n-2$ we simply write $\sum_{s+\ell \le n-2}$.

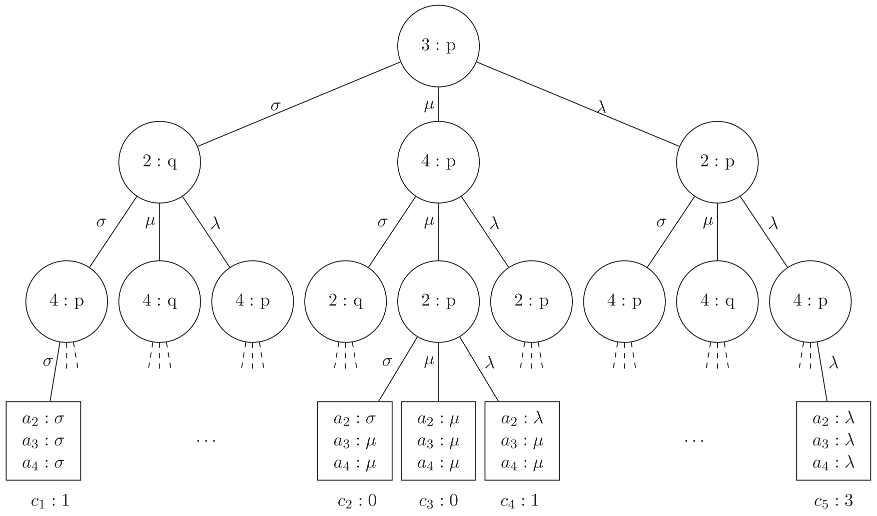


Fig. 2. An example for a decision tree to classify three elements a_2, a_3, and a_4 according to the pivots a_1 and a_5. Five out of the 27 leaves are explicitly drawn, showing the classification of the elements and the costs c_i of the specific paths.

just the product of all these edge probabilities on the unique path from the root to v. The probability that the edge labeled σ out of a node v is used can then be calculated as $p^v_{s,\ell} \cdot (s - s_v)/(n - 2 - \text{level}(v))$. Similarly, the probability that the edge labeled λ is used is $p^v_{s,\ell} \cdot (\ell - \ell_v)/(n - 2 - \text{level}(v))$. Note that all this is independent of the actual ordering in which the classification tree inspects the elements. We can thus always assume some fixed ordering and forget about the label $i(v)$ of node v.

By linearity of expectation, we can sum up the contribution to the additional comparison count for each node separately. Thus, we may calculate

$$\mathrm{E}\big(S_2^T + L_2^T \mid s, \ell\big) = \sum_{\substack{v \in T \\ l(v)=\mathrm{q}}} p^v_{s,\ell} \cdot \frac{s - s_v}{n - 2 - \text{level}(v)} + \sum_{\substack{v \in T \\ l(v)=\mathrm{p}}} p^v_{s,\ell} \cdot \frac{\ell - \ell_v}{n - 2 - \text{level}(v)}. \tag{6}$$

The setup developed so far makes it possible to describe the connection between a classification tree T and its average comparison count in general. Let F^T_{p} and F^T_{q} be two random variables that denote the number of elements that are compared with the smaller and larger pivot first, respectively, when using T. Given s and ℓ, let $f^{\mathrm{q}}_{s,\ell} = E(F^T_{\mathrm{q}} \mid s, \ell)$ $[f^{\mathrm{p}}_{s,\ell} = E(F^T_{\mathrm{p}} \mid s, \ell)]$ denote the average number of comparisons with the larger [smaller] pivot first. Now, if it was decided in each step by independent random experiments with the correct expectations $s/(n-2)$, $m/(n-2)$, and $\ell/(n-2)$, respectively, whether an element is small, medium, or large, it would be clear that, for example, $f^{\mathrm{q}}_{s,\ell} \cdot s/(n-2)$ is the average number of small elements that are compared with the larger pivot first. Lemma 3.1 shows that one can indeed use this intuition in the calculation of the average comparison count, except that one gets an additional $O(n^{1-\varepsilon})$ term due to the elements tested not being independent.

Lemma 3.1. *Let T be a classification tree. Let $\mathrm{E}(P_n^T)$ be the average number of key comparisons for classifying an input of n elements using T. Then there exists a constant $\varepsilon > 0$ such that*

$$\mathrm{E}(P_n^T) = \frac{4}{3}n + \frac{1}{\binom{n}{2}\cdot(n-2)} \sum_{s+\ell\leq n-2} \left(f^{\mathrm{q}}_{s,\ell}\cdot s + f^{\mathrm{p}}_{s,\ell}\cdot \ell\right) + O(n^{1-\varepsilon}).$$

Proof. Fix p and q (and thus s, m, and ℓ). We will show that

$$\mathrm{E}\left(S_2^T + L_2^T \,\middle|\, s,\ell\right) = \frac{f^{\mathrm{q}}_{s,\ell}\cdot s + f^{\mathrm{p}}_{s,\ell}\cdot \ell}{n-2} + O(n^{1-\varepsilon}). \tag{7}$$

(The lemma then follows by substituting this into Equation (5).)

We call a node v in T *on-track* (to the expected values) if

$$\begin{aligned} l(v) = \mathrm{q} \text{ and } \left|\frac{s}{n-2} - \frac{s-s_v}{n-\mathrm{level}(v)-2}\right| &\leq \frac{1}{n^{1/12}} \qquad \text{or} \\ l(v) = \mathrm{p} \text{ and } \left|\frac{\ell}{n-2} - \frac{\ell-\ell_v}{n-\mathrm{level}(v)-2}\right| &\leq \frac{1}{n^{1/12}}. \end{aligned} \tag{8}$$

Otherwise, we call v *off-track*.

We first obtain an upper bound. Starting from Equation (6), we calculate:

$$\begin{aligned} \mathrm{E}\left(S_2^T + L_2^T \mid s,\ell\right) &= \sum_{v\in T, l(v)=\mathrm{q}} p^v_{s,\ell}\cdot\frac{s-s_v}{n-2-\mathrm{level}(v)} + \sum_{v\in T, l(v)=\mathrm{p}} p^v_{s,\ell}\cdot\frac{\ell-\ell_v}{n-2-\mathrm{level}(v)} \\ &= \sum_{v\in T, l(v)=\mathrm{q}} p^v_{s,\ell}\cdot\frac{s}{n-2} + \sum_{v\in T, l(v)=\mathrm{p}} p^v_{s,\ell}\cdot\frac{\ell}{n-2} \\ &\quad + \sum_{v\in T, l(v)=\mathrm{q}} p^v_{s,\ell}\left(\frac{s-s_v}{n-2-\mathrm{level}(v)} - \frac{s}{n-2}\right) \\ &\quad + \sum_{v\in T, l(v)=\mathrm{p}} p^v_{s,\ell}\left(\frac{\ell-\ell_v}{n-2-\mathrm{level}(v)} - \frac{\ell}{n-2}\right) \\ &\leq \sum_{v\in T, l(v)=\mathrm{q}} p^v_{s,\ell}\cdot\frac{s}{n-2} + \sum_{v\in T, l(v)=\mathrm{p}} p^v_{s,\ell}\cdot\frac{\ell}{n-2} \\ &\quad + \sum_{\substack{v\in T, l(v)=\mathrm{q}\\ v \text{ on-track}}} \frac{p^v_{s,\ell}}{n^{1/12}} + \sum_{\substack{v\in T, l(v)=\mathrm{q}\\ v \text{ off-track}}} p^v_{s,\ell} + \sum_{\substack{v\in T, l(v)=\mathrm{p}\\ v \text{ on-track}}} \frac{p^v_{s,\ell}}{n^{1/12}} + \sum_{\substack{v\in T, l(v)=\mathrm{p}\\ v \text{ off-track}}} p^v_{s,\ell}, \end{aligned} \tag{9}$$

where the last step follows by separating on-track and off-track nodes and using Equation (8). (For off-track nodes we use that the left-hand side of the inequalities in Equation (8) is at most 1.) For the sums in the last line of Equation (9), consider each level of the classification tree separately. Since the probabilities $p^v_{s,\ell}$ for nodes v on the same level sum up to 1, the contribution of the $1/n^{1/12}$ terms is bounded by $O(n^{11/12})$. Using the definition of $f^{\mathrm{q}}_{s,\ell}$ and $f^{\mathrm{p}}_{s,\ell}$, we continue as follows:

$$\begin{aligned}\mathrm{E}\big(S_2^T + L_2^T \mid s,\ell\big) &\le \sum_{v\in T, l(v)=\mathrm{q}} p^v_{s,\ell}\cdot\frac{s}{n-2} + \sum_{v\in T, l(v)=\mathrm{p}} p^v_{s,\ell}\cdot\frac{\ell}{n-2} + \sum_{\substack{v\in T\\ v \text{ off-track}}} p^v_{s,\ell} + O(n^{11/12})\\ &= \frac{f^{\mathrm{q}}_{s,\ell}\cdot s + f^{\mathrm{p}}_{s,\ell}\cdot\ell}{n-2} + \sum_{v\in T, v\text{ off-track}} p^v_{s,\ell} + O(n^{11/12})\\ &= \frac{f^{\mathrm{q}}_{s,\ell}\cdot s + f^{\mathrm{p}}_{s,\ell}\cdot\ell}{n-2}\\ &\quad + \sum_{k=0}^{n-3}\Pr(\text{the node reached on level } k \text{ is off-track}) + O(n^{11/12}), \quad (10)\end{aligned}$$

where in the last step we just rewrote the sum to consider each level in the classification tree separately. So, to show Equation (7), it remains to bound the sum in Equation (10) by $O(n^{1-\varepsilon})$.

To do this, consider a random input that is classified using T. Using an appropriate tail bound (viz. *the method of average bounded differences*), we will show that with very high probability we do not reach an off-track node in the classification tree in the first $n - n^{3/4}$ levels. Intuitively, this means that it is highly improbable that, while under way, the observed fraction of small elements deviates very far from the average $s/(n-2)$.

Let X_j be the 0-1 random variable that is 1 if the j-th classified element is small; let Y_j be the 0-1 random variable that is 1 if the j-th classified element is large. Let $s_i = X_1 + \cdots + X_i$ and $\ell_i = Y_1 + \cdots + Y_i$ for $1 \le i \le n-2$.

CLAIM 3.2. *Let* $1 \le i \le n-2$. *Then*

$$\Pr(|s_i - E(s_i)| > n^{2/3}) \le 2\exp(-n^{1/3}/2), \text{ and}$$
$$\Pr(|\ell_i - E(\ell_i)| > n^{2/3}) \le 2\exp(-n^{1/3}/2).$$

PROOF. We prove the first inequality. First, we bound the difference c_j between the expectation of s_i conditioned on $X_1,\ldots,X_j$, respectively, $X_1,\ldots,X_{j-1}$ for $1 \le j \le i$. Using linearity of expectation we calculate

$$\begin{aligned} c_j &= \big|\mathrm{E}(s_i \mid X_1,\ldots,X_j) - \mathrm{E}(s_i \mid X_1,\ldots,X_{j-1})\big|\\ &= \left|\sum_{k=1}^{j} X_k + \sum_{k=j+1}^{i}\frac{s-s_j}{n-j-2} - \sum_{k=1}^{j-1}X_k - \sum_{k=j}^{i}\frac{s-s_{j-1}}{n-j-1}\right|\\ &= \left|X_j + \sum_{k=j+1}^{i}\frac{s-s_{j-1}-X_j}{n-j-2} - \sum_{k=j}^{i}\frac{s-s_{j-1}}{n-j-1}\right|\\ &= \left|X_j - X_j\cdot\frac{i-j}{n-j-2} + (s-s_{j-1})\left(\frac{i-j}{n-j-2} - \frac{i-j+1}{n-j-1}\right)\right|\\ &= \left|X_j\left(1-\frac{i-j}{n-j-2}\right) - (s-s_{j-1})\left(\frac{n-i-2}{(n-j-1)(n-j-2)}\right)\right|\\ &\le \max\left\{\left|X_j\left(1-\frac{i-j}{n-j-2}\right)\right|, \left|\frac{s-s_{j-1}}{n-j-1}\right|\right\} \le 1.\end{aligned}$$

In this situation, we may apply the bound known as the method of averaged bounded differences [Dubhashi and Panconesi 2009, Theorem 5.3], which reads

$$\Pr(|s_i - \mathrm{E}(s_i)| > t) \le 2\exp\left(-\frac{t^2}{2\sum_{j\le i} c_j^2}\right),$$

and get

$$\Pr(|s_i - \mathrm{E}(s_i)| > n^{2/3}) \le 2\exp\left(\frac{-n^{4/3}}{2i}\right),$$

which is not larger than $2\exp(-n^{1/3}/2)$.

Assume that $|s_i - \mathrm{E}(s_i)| = |s_i - i \cdot s/(n-2)| \le n^{2/3}$. We have

$$\left|\frac{s}{n-2} - \frac{s-s_i}{n-2-i}\right| \le \left|\frac{s}{n-2} - \frac{s(1-i/(n-2))}{n-2-i}\right| + \left|\frac{n^{2/3}}{n-2-i}\right| = \frac{n^{2/3}}{n-2-i}.$$

That means that for each of the first $i \le n - n^{3/4}$ levels, with very high probability we are in an *on-track node* on level i, because the deviation from the ideal case that we see a small element with probability $s/(n-2)$ is $n^{2/3}/(n-2-i) \le n^{2/3}/n^{3/4} = 1/n^{1/12}$. Thus, for the first $n - n^{3/4}$ levels, the contribution of the sums of the probabilities of off-track nodes in Equation (10) is at most $n^{11/12}$. For the last $n^{3/4}$ levels of the tree, we use that the contribution of the probabilities that we reach an off-track node on level i is at most 1 for a fixed level.

This shows that the contribution of the sum in Equation (10) is $O(n^{11/12})$. This finishes the proof of the upper bound on $\mathrm{E}(S_2^T + L_2^T \mid s, \ell)$ given in Equation (10). The calculations for the lower bound are similar and are omitted here. □

There is the following technical complication when using this lemma in analyzing a strategy that is turned into a dual-pivot quicksort algorithm: The cost bound is $a \cdot n + O(n^{1-\varepsilon})$, and Hennequin's result (Equation (4)) cannot be applied directly to such partitioning costs. However, Theorem 3.3 says that the leading term of Equation (4) applies to this situation as well, and the additional $O(n^{1-\varepsilon})$ term in the partitioning cost is completely covered in the $O(n)$ error term of (4).

THEOREM 3.3. *Let $\mathcal{A}$ be a dual-pivot quicksort algorithm that gives rise to a classification tree T_n for each subarray of length n. Assume $\mathrm{E}(P_n^{T_n}) = a \cdot n + O(n^{1-\varepsilon})$ for all n, for some constants a and $\varepsilon > 0$. Then $\mathrm{E}(C_n^{\mathcal{A}}) = \frac{6}{5} a n \ln n + O(n)$.*

PROOF. By linearity of expectation, we may split the partitioning cost into two terms $t_1(n) = a \cdot n$ and $t_2(n) = K \cdot n^{1-\varepsilon}$, solve recursion Equation (3) independently for these two cost terms, and add the results. Applying Equation (4) for average partitioning cost $t_1(n)$ yields an average comparison count of $\frac{6}{5} a n \ln n + O(n)$. Obtaining the bound of $O(n)$ for the term $t_2(n)$ is a standard application of the Continuous Master Theorem of Roura [Roura 2001] and has been derived for the dual-pivot quicksort recurrence by Wild, Nebel, and Martínez in a recent technical report [Wild et al. 2014, Appendix D]. For completeness, the calculation is given in Appendix A. □

Lemma 3.1 and Theorem 3.3 tell us that, for the analysis of the average comparison count of a dual-pivot quicksort algorithm, we just have to find out what $f^{\mathrm{p}}_{s,\ell}$ and $f^{\mathrm{q}}_{s,\ell}$ are for this algorithm. Moreover, to design a good algorithm (with regard to the average comparison count), we should try to make $f^{\text{q}}_{s,\ell} \cdot s + f^{\text{p}}_{s,\ell} \cdot \ell$ small for each pair s, ℓ.

4. ANALYSIS OF SOME KNOWN CLASSIFICATION STRATEGIES

In this section, we study different classification strategies in the light of the formulas from Section 3.

Oblivious Strategies. We first consider strategies that do not use information of previous classifications for future classifications. To this end, we call a classification tree *oblivious* if, for each level, all nodes v on this level share the same label $l(v) \in \{\text{p}, \text{q}\}$. This means that these algorithms do not react to the outcome of previous classifications, but use a fixed sequence of pivot choices. Examples for such strategies are, e.g.,

—always compare to the smaller pivot first,
—always compare to the larger pivot first,
—alternate the pivots in each step.

Let f^{q}_n denote the average number of comparisons to the larger pivot first. By assumption, this value is independent of s and ℓ. Hence, these strategies make sure that $f^{\text{q}}_{s,\ell} = f^{\text{q}}_n$ and $f^{\text{p}}_{s,\ell} = n - 2 - f^{\text{q}}_n$ for all pairs of values s, ℓ.

Applying Lemma 3.1 gives us

$$\begin{aligned}
\mathrm{E}(P_n) &= \frac{4}{3}n + \frac{1}{\binom{n}{2} \cdot (n-2)} \cdot \sum_{s+\ell \le n-2} \left(f^{\text{q}}_n \cdot s + (n-2-f^{\text{q}}_n) \cdot \ell\right) + O(n^{1-\varepsilon}) \\
&= \frac{4}{3}n + \frac{f^{\text{q}}_n}{\binom{n}{2} \cdot (n-2)} \cdot \left(\sum_{s+\ell \le n-2} s\right) + \frac{n-2-f^{\text{q}}_n}{\binom{n}{2} \cdot (n-2)} \cdot \left(\sum_{s+\ell \le n-2} \ell\right) + O(n^{1-\varepsilon}) \\
&= \frac{4}{3}n + \frac{1}{\binom{n}{2}} \cdot \left(\sum_{s+\ell \le n-2} s\right) + O(n^{1-\varepsilon}) = \frac{5}{3}n + O(n^{1-\varepsilon}).
\end{aligned}$$

Using Theorem 3.3, we get $\mathrm{E}(C_n) = 2n \ln n + O(n)$—the leading term being the same as in standard quicksort. So, for each strategy that does not adapt to the outcome of previous classifications, there is no difference to the average comparison count of classical quicksort. Note that this also holds for *randomized strategies* such as "flip a coin to choose the pivot used in the first comparison" since such a strategy can be seen as a probability distribution on oblivious strategies.

Yaroslavskiy's Algorithm. Following Wild and Nebel [2012, Section 3.2], Yaroslavskiy's algorithm is an implementation of the following Strategy $\mathcal{Y}$: *Compare ℓ elements to q first, and compare the other elements to p first.* We get that $f^{\text{q}}_{s,\ell} = \ell$ and $f^{\text{p}}_{s,\ell} = s + m$. Applying Lemma 3.1, we get

$$\mathrm{E}\left(P^{\mathcal{Y}}_n\right) = \frac{4}{3}n + \frac{1}{\binom{n}{2}} \sum_{s+\ell \le n-2} \left(\frac{s\ell}{n-2} + \frac{(s+m)\ell}{n-2}\right) + O(n^{1-\varepsilon}).$$

Of course, it is possible to evaluate this sum by hand. We used Maple® to obtain $\mathrm{E}(P^{\mathcal{Y}}_n) = \frac{19}{12}n + O(n^{1-\varepsilon})$. Using Theorem 3.3 gives $\mathrm{E}(C_n) = 1.9n \ln n + O(n)$, as in Wild and Nebel [2012].

Sedgewick's Algorithm. Following Wild and Nebel [2012, Section 3.2], Sedgewick's algorithm amounts to an implementation of the following Strategy $\mathcal{S}$: *Compare (on* *average) a fraction of $s/(s+\ell)$ of the keys with q first, and compare the other keys with p first.* We get $f^{\mathrm{q}}_{s,\ell} = (n-2)\cdot s/(s+\ell)$ and $f^{\mathrm{p}}_{s,\ell} = (n-2)\cdot \ell/(s+\ell)$.

Plugging these values into Lemma 3.1, we calculate

$$\mathrm{E}\big(P^{\mathcal{S}}_n\big) = \frac{4}{3}n + \frac{1}{\binom{n}{2}} \sum_{s+\ell\le n-2} \left(\frac{s^2}{s+\ell} + \frac{\ell^2}{s+\ell}\right) + O(n^{1-\varepsilon}) = \frac{16}{9}n + O(n^{1-\varepsilon}).$$

Applying Theorem 3.3 gives $\mathrm{E}(C_n) = 2.133... \cdot n \ln n + O(n)$, as known from Wild and Nebel [2012].

Obviously, this is worse than the oblivious strategies considered earlier.[5] This is easily explained intuitively: If the fraction of small elements is large, it will compare many elements with q first. But this costs two comparisons for each small element. Conversely, if the fraction of large elements is large, it will compare many elements to p first, which is again the wrong decision.

Since Sedgewick's strategy seems to do exactly the opposite of what one should do to lower the comparison count, we consider the following modified Strategy $\mathcal{S}'$: *For given p and q, compare (on average) a fraction of $s/(s+\ell)$ of the keys with p first, and compare the other keys with q first.* ($\mathcal{S}'$ simply uses p first when $\mathcal{S}$ would use q first and vice versa.)

Using the same analysis as previously, we get $\mathrm{E}(P_n) = \frac{14}{9}n + O(n^{1-\varepsilon})$, which yields $\mathrm{E}(C_n) = 1.866... \cdot n \ln n + O(n)$—improving on the standard algorithm and even on Yaroslavskiy's algorithm! Note that this has been observed by Wild in his Master's Thesis as well [Wild 2013].

Remark. Swapping the first comparison with p and q as in the strategy just described is a general technique. In fact, if the leading coefficient of the average number of comparisons for a fixed rule for choosing p or q first is α, for example, $\alpha = 2.133...$ for Strategy $\mathcal{S}$, then the leading coefficient of the strategy that does the opposite is $4-\alpha$ (e.g., $4 - 2.133... = 1.866...$) as in Strategy $\mathcal{S}'$.

5. AN ASYMPTOTICALLY OPTIMAL CLASSIFICATION STRATEGY

Looking at the previous sections, all strategies used the idea that we should compare a certain fraction of elements to p first and the other elements to q first. In this section, we study the following Strategy $\mathcal{I}$: *If $s > \ell$ then always compare with p first, otherwise always compare with q first.*

Of course, for an implementation of this strategy, we have to deal with the problem of finding out which case applies before the comparisons have been made. We shall analyze a guessing strategy to resolve this.

5.1. Analysis of the Idealized Classification Strategy

Assume for a moment that for a given random input with pivots p, q the strategy "magically" knows whether $s > \ell$ or not and correctly determines the pivot that should be used for all comparisons. For fixed s and ℓ, this means that for $s > \ell$ the classification strategy makes exactly ℓ additional comparisons, and for $s \le \ell$ it makes s additional comparisons.

[5]We remark that in his thesis Sedgewick [1975] focused on the average number of swaps, not on the comparison count.

When we start from Equation (5), a standard calculation shows that for this strategy

$$\mathrm{E}(P_n) = \frac{4}{3}n + \frac{1}{\binom{n}{2}} \sum_{s+\ell \le n-2} \min(s, \ell) = \frac{3}{2}n + O(1). \tag{11}$$

Applying Equation (4), we get $\mathrm{E}(C_n) = 1.8n \ln n + O(n)$, which is by $0.1n \ln n$ smaller than the average number of key comparisons in Yaroslavskiy's algorithm.

To see that this method is asymptotically optimal, recall that according to Lemma 3.1 the average comparison count is determined up to a linear term by the parameters $f^{\mathrm{q}}_{s,\ell}$ and $f^{\mathrm{p}}_{s,\ell} = n - 2 - f^{\mathrm{q}}_{s,\ell}$. Strategy $\mathcal{I}$ chooses these values such that $f^{\mathrm{q}}_{s,\ell}$ is either 0 or $n - 2$, minimizing each term of the sum in Lemma 3.1—and thus minimizing the sum.

5.2. Guessing Whether $s < \ell$ or Not

We explain how the idealized classification strategy just described can be approximated by an implementation. The idea is to make a few comparisons and use the outcome as a basis for a guess.

After p and q are chosen, classify the first $n_{\mathrm{s}} = O(n^{1-\varepsilon})$ many elements (the *sample*) and calculate s' and ℓ', the number of small and large elements in the sample. If $s' < \ell'$, compare the remaining $n - 2 - n_{\mathrm{s}}$ elements with q first; otherwise, compare them with p first. We say that the guess was *correct* if $s' < \ell'$ and $s < \ell$ or $s' \ge \ell'$ and $s \ge \ell$. In order not to clutter up formulas, we will always assume that $n^{1-\varepsilon}$ is an integer. One would otherwise work with $\lceil n^{1-\varepsilon} \rceil$.

We incorporate guessing errors and sampling cost into Equation (11) as follows:

$$\begin{aligned}
\mathrm{E}(P_n) &= \frac{4}{3}n + \frac{1}{\binom{n}{2}} \sum_{s+\ell \le n-2} (\Pr(\text{guess correct} \mid s, \ell) \cdot \min(s, \ell) \\
&\quad + \Pr(\text{guess wrong} \mid s, \ell) \cdot \max(s, \ell)) + O(n^{1-\varepsilon}) \\
&= \frac{4}{3}n + \frac{2}{\binom{n}{2}} \sum_{s=0}^{n/2} \sum_{\ell=s+1}^{n-s} (\Pr(\text{guess correct} \mid s, \ell) \cdot s \\
&\quad + \Pr(\text{guess wrong} \mid s, \ell) \cdot \ell) + O(n^{1-\varepsilon}),
\end{aligned} \tag{12}$$

where the cost for comparing the elements in the sample is covered by the $O(n^{1-\varepsilon})$ term. Lemma 5.1 says that, for a wide range of values s and ℓ, the probability of a guessing error is exponentially small.

LEMMA 5.1. *Let s and ℓ with $s \le \ell - n^{3/4}$ and $\ell \ge n^{3/4}$ for $n \in \mathbb{N}$ be given. Let $n_s = n^{2/3}$. Then* $\Pr(\text{guess wrong} \mid s, \ell) \le \exp(-n^{1/6}/18)$.

PROOF. Let X_i be a random variable that is -1 if the i-th classified element of the sample is large, 0 if it is medium, and 1 if it is small. Let $d = \sum_{i=1}^{n_{\mathrm{s}}} X_i$.

As in the proof of Lemma 3.1, we want to apply the method of averaged bounded differences. Using the assumptions on the values of s and ℓ, straightforward calculations show that $\mathrm{E}(d) \le -n_{\mathrm{s}}/n^{1/4} = -n^{5/12}$. Furthermore, we have that

$$c_i = \left|\mathrm{E}(d \mid X_1, \ldots, X_i) - \mathrm{E}(d \mid X_1, \ldots, X_{i-1})\right| \le 3, \text{ for } 1 \le i \le n_{\mathrm{s}}.$$

To see this, we let s_i and ℓ_i denote the number of small and large elements, respectively, that are still present after the first i elements have been classified (i.e., $X_1, \ldots, X_i$ have been determined). Let Y_i be the 0-1 random variable that is 1 if and only if X_i is 1, and let Z_i be the 0-1 random variable that is 1 if and only if X_i is -1. We calculate:

$$\begin{aligned}
&|\mathrm{E}(d \mid X_1, \ldots, X_i) - \mathrm{E}(d \mid X_i, \ldots, X_{i-1})| \\
&= \Bigg| \sum_{j=1}^{i} X_j + \sum_{j=i+1}^{n_s} [\Pr(X_j = 1 \mid X_1, \ldots, X_i) - \Pr(X_j = -1 \mid X_1, \ldots, X_i)] \\
&\quad - \sum_{j=1}^{i-1} X_j - \sum_{j=i}^{n_s} [\Pr(X_j = 1 \mid X_1, \ldots, X_{i-1}) - \Pr(X_j = -1 \mid X_1, \ldots, X_{i-1})] \Bigg| \\
&= \left| X_i + \sum_{j=i+1}^{n_s} \left[\frac{s_i}{n-i} - \frac{\ell_i}{n-i} \right] - \sum_{j=i}^{n_s} \left[\frac{s_{i-1}}{n-i+1} - \frac{\ell_{i-1}}{n-i+1} \right] \right| \\
&= \left| X_i + (n_s - i) \cdot \left[\frac{s_{i-1} - Y_i}{n-i} - \frac{\ell_{i-1} - Z_i}{n-i} \right] - (n_s - i + 1) \cdot \left[\frac{s_{i-1}}{n-i+1} - \frac{\ell_{i-1}}{n-i+1} \right] \right| \\
&= \left| X_i + \frac{(n_s - i) \cdot (Z_i - Y_i)}{n-i} + s_{i-1} \left[\frac{n_s - i}{n-i} - \frac{n_s - i + 1}{n-i+1} \right] + \ell_{i-1} \left[\frac{n_s - i + 1}{n-i+1} - \frac{n_s - i}{n-i} \right] \right| \\
&= \left| X_i + \frac{(n_s - i) \cdot (Z_i - Y_i)}{n-i} + (\ell_{i-1} - s_{i-1}) \cdot \frac{n - n_s}{(n-i)(n-i+1)} \right| \\
&\le |X_i| + |Z_i - Y_i| + \left| \frac{\ell_{i-1} - s_{i-1}}{n-i+1} \right| \le 3.
\end{aligned}$$

The method of averaged bounded differences (see, for example, [Dubhashi and Panconesi 2009, Theorem 5.3]) now implies that

$$\Pr(d > \mathrm{E}(d) + t) \le \exp\left(-\frac{t^2}{2 \sum_{i \le n_s} c_i^2} \right), \text{ for } t > 0,$$

which with $t = n^{5/12} \le -\mathrm{E}(d)$ yields

$$\Pr(d > 0) \le \exp\left(-\frac{n^{1/6}}{18} \right). \quad \square$$

Of course, we get an analogous result for $s \ge n^{3/4}$ and $\ell \le s - n^{3/4}$.

Classification Strategy $\mathcal{SP}$ works as follows: Classify the first $n_s = n^{2/3}$ elements. Let s' $[\ell']$ be the number of elements classified as being small [large]. If $s' > \ell'$, then use p for the first comparison for the remaining elements; otherwise, use q.

We can now analyze the average number of key comparisons of this strategy turned into a dual-pivot quicksort algorithm.

THEOREM 5.2. *The average comparison count of Strategy $\mathcal{SP}$ turned into a dual-pivot quicksort algorithm is* $1.8n \ln n + O(n)$.

PROOF. We only have to analyze the expected classification cost. First, we classify n_s many elements. The number of key comparisons for these classifications is at most $2n^{2/3} = O(n^{1-\varepsilon})$. By symmetry, we may focus on the case that $s \le \ell$. We distinguish the following three cases:

(1) $\ell \leq n^{3/4}$: The contribution of terms in Equation (12) satisfying this case is at most

$$\frac{2}{\binom{n}{2}} \sum_{\ell=0}^{n^{3/4}} \sum_{s=0}^{\ell} \ell = O(n^{1/4}).$$

(2) $\ell - n^{3/4} \leq s \leq \ell$: Let $\mathrm{m}_1(s,n) = \min(s + n^{3/4}, n - s)$. The contribution of terms in Equation (12) satisfying this case is at most

$$\frac{2}{\binom{n}{2}} \sum_{s=0}^{n/2} \sum_{\ell=s}^{\mathrm{m}_1(s,n)} \ell = O(n^{3/4}).$$

(3) $\ell \geq n^{3/4}$ and $s \leq \ell - n^{3/4}$. Let $\mathrm{m}_2(\ell,n) = \min(n-\ell, \ell - n^{3/4})$. Following Lemma 5.1, the probability of guessing wrong is at most $\exp(-n^{1/6}/18)$. The contribution of this case in Equation (12) is hence at most

$$\frac{2}{\binom{n}{2}} \sum_{\ell=n^{3/4}}^{n} \sum_{s=0}^{\mathrm{m}_2(\ell,n)} \left(s + \exp(-n^{1/6}/18)\ell\right) = \left(\frac{2}{\binom{n}{2}} \sum_{\ell=n^{3/4}}^{n} \sum_{s=0}^{\mathrm{m}_2(\ell,n)} s\right) + o(1) = \frac{n}{6} + O(1).$$

Using these contributions in Equation (12), we expect a partitioning step to make $\frac{3}{2}n + O(n^{1-\varepsilon})$ key comparisons. Applying Theorem 3.3, we get $\mathrm{E}(C_n) = 1.8n \ln n + O(n)$. □

In this section, we have seen an asymptotically optimal strategy. In the next section, we present the optimal classification strategy. Unfortunately, it is even more unrealistic than the idealized Strategy $\mathcal{I}$ just presented. However, we will give an implementation that comes very close to the optimal strategy in terms of the number of comparisons.

6. THE OPTIMAL CLASSIFICATION STRATEGY AND ITS IMPLEMENTATION

We will consider two more strategies, an optimal (but not algorithmic) strategy and an algorithmic strategy that is optimal up to a very small error term.

We first study the (unrealistic!) setting where s and ℓ (i.e., the number of small and large elements) are known to the algorithm after the pivots are chosen, and the classification tree can have different node labels for each such pair of values. Recall that s_v and ℓ_v respectively denote the number of elements that have been classified as small and large when at node v in the classification tree. We consider the following Strategy $\mathcal{O}$: *Given s and ℓ, the comparison at node v is with the smaller pivot first if $s - s_v > \ell - \ell_v$; otherwise, it is with the larger pivot first.*[6]

THEOREM 6.1. *Strategy $\mathcal{O}$ is optimal; that is, its ACT is at most as large as ACT_T for all classification trees T. When using $\mathcal{O}$ in a dual-pivot quicksort algorithm, we have $\mathrm{E}(C_n^{\mathcal{O}}) = 1.8n \ln n + O(n)$.*

PROOF. The proof of the first statement (optimality) is surprisingly simple. Fix the two pivots, and consider Equation (6). For each node v in the classification tree, Strategy $\mathcal{O}$ chooses the label that minimizes the contribution of this node to Equation (6). So, it minimizes each term of the sum and thus minimizes the additional cost term in Equation (5).

We now prove the second statement. For this, we first derive an upper bound of $1.8n \ln n + O(n)$ for the average number of comparisons and then show that this is tight.

For the first part, let an input with n entries and two pivots be given so that there are s small and ℓ large elements. Assume $s \geq \ell$. Omit all medium elements to obtain a

[6]This strategy was suggested to us by Thomas Hotz (personal communication).

reduced input $(a_1, \ldots, a_{n'})$ with $n' = s + \ell$. For $0 \leq i \leq n'$, let s_i and ℓ_i denote the number of small and large elements remaining in $(a_{i+1}, \ldots, a_{n'})$. Let $D_i = s_i - \ell_i$. Of course we have $D_0 = s - \ell$ and $D_{n'} = 0$. Let $i_1 < i_2 < \cdots < i_k$ be the list of indices i with $D_i = 0$. (In particular, $i_k = n'$.) Rounds i with $D_i = 0$ are called *zero-crossings*. Consider some j with $D_{i_j} = D_{i_{j+1}} = 0$. The numbers $D_{i_j+1}, \ldots, D_{i_{j+1}-1}$ are nonzero and have the same positive [or negative] sign. The algorithm compares $a_{i_j+2}, \ldots, a_{i_{j+1}}$ with the smaller [larger] pivot first and a_{i_j+1} with the larger pivot first. Since $\{a_{i_j+1}, \ldots, a_{i_{j+1}}\}$ contains the same number of small and large elements, the contribution of this segment to the additional comparison count is $\frac{1}{2}(i_{j+1} - i_j) - 1$ [or $\frac{1}{2}(i_{j+1} - i_j)$].

If $D_0 > 0$ (i.e., $s > \ell$), all elements in $\{a_1, \ldots, a_{i_1}\}$ are compared with the smaller pivot first, and this set contains $\frac{1}{2}(i_1 - (s - \ell))$ large elements (and $\frac{1}{2}(i_1 + (s - \ell))$ small elements), giving a contribution of $\frac{1}{2}(i_1 - (s - \ell))$ to the additional comparison count. Overall, the additional comparison count $S_2 + L_2$ (see the end of Section 2, in particular Equation (5)) of Strategy $\mathcal{O}$ on the considered input is

$$\frac{i_1 - (s - \ell)}{2} + \sum_{j=1}^{k-1} \frac{i_{j+1} - i_j}{2} - k^* = \frac{n' - (s - \ell)}{2} - k^* = \ell - k^*, \tag{13}$$

for some correction term $k^* \in \{0, \ldots, k - 1\}$.

Averaging the upper bound ℓ over all pivot choices, we obtain the following bound for the additional cost term of Strategy $\mathcal{O}$:

$$\mathrm{E}\,(S_2 + L_2) \leq \frac{1}{\binom{n}{2}} \cdot \left(2 \cdot \sum_{\substack{s+\ell \leq n \\ \ell < s}} \ell + \sum_{\ell \leq n/2} \ell \right). \tag{14}$$

This gives an average number of at most $1.5n + O(1)$ comparisons. For such a partitioning cost, we can use Equation (4) and obtain an average comparison count for sorting via Strategy $\mathcal{O}$ of at most $1.8n \ln n + O(n)$.

It remains to show that this is tight. We shall see that the essential step in this analysis is to show that the average (over all inputs) of the number of *zero-crossings* (the number k from earlier, excepting the zero-crossing at position n) is $O(\log n)$. Again, we temporarily omit medium elements to simplify calculations (i.e., we assume that the number of small and large elements together is n'). Fix a position $n' - 2i$, for $1 \leq i \leq n'/2$. If $D_{n'-2i} = 0$, then there are as many small elements as there are large elements in the last $2i$ positions of the input. Consequently, the input has to contain between i and $n' - i$ small elements; otherwise, no zero-crossing is possible. The probability that a random input (excepting the two pivots) of n' elements has exactly s small elements is $1/(n' + 1)$, for $0 \leq s \leq n'$. Let $Z_{n'}$ be the random variable that denotes the number of zero-crossings for an input of n' elements excepting the two pivots. We calculate:

$$\begin{aligned}
\mathrm{E}(Z_{n'}) &= \sum_{1 \leq i \leq n'/2} \Pr(\text{there is a zero-crossing at position } n' - 2i) \\
&= \frac{1}{n' + 1} \sum_{i=1}^{n'/2} \sum_{s=i}^{n'-i} \Pr(D_{n'-2i} = 0 \mid s \text{ small elements}) \\
&= \frac{1}{n' + 1} \sum_{i=1}^{n'/2} \sum_{s=i}^{n'-i} \frac{\binom{2i}{i} \cdot \binom{n'-2i}{s-i}}{\binom{n'}{s}} \leq \frac{2}{n' + 1} \sum_{i=1}^{n'/2} \sum_{s=i}^{n'/2} \frac{\binom{2i}{i} \cdot \binom{n'-2i}{s-i}}{\binom{n'}{s}},
\end{aligned}$$

where the last step follows by symmetry: Replace $s > n'/2$ by $n' - s$.

By using the well-known estimate $\binom{2i}{i} = \Theta(2^{2i}/\sqrt{i})$ (which follows directly from Stirling's approximation), we continue by

$$\begin{aligned}
\mathrm{E}(Z_{n'}) &= \Theta\left(\frac{1}{n'}\right)\sum_{i=1}^{n'/2}\frac{2^{2i}}{\sqrt{i}}\sum_{s=i}^{n'/2}\frac{\binom{n'-2i}{s-i}}{\binom{n'}{s}}\\
&= \Theta\left(\frac{1}{n'}\right)\sum_{i=1}^{n'/2}\frac{2^{2i}}{\sqrt{i}}\sum_{s=i}^{n'/2}\frac{(n'-s)\cdot\ldots\cdot(n'-s-i+1)\cdot s\cdot\ldots\cdot(s-i+1)}{n'\cdot\ldots\cdot(n'-2i+1)}\\
&= \Theta\left(\frac{1}{n'}\right)\sum_{i=1}^{n'/2}\frac{n'+1}{\sqrt{i}(n'-2i+1)}\sum_{j=0}^{n'/2-i}\prod_{k=0}^{i-1}\frac{(n'+2j-2k)(n'-2j-2k)}{(n'-2k+1)(n'-2k)},
\end{aligned} \tag{15}$$

where the last step follows by an index transformation using $j = n'/2 - s$ and multiplying 2^{2i} into the terms of the rightmost fraction. We now obtain an upper bound for the rightmost product:

$$\prod_{k=0}^{i-1}\frac{(n'+2j-2k)(n'-2j-2k)}{(n'-2k+1)(n'-2k)} \le \prod_{k=0}^{i-1}\left(1-\left(\frac{2j}{n'-2k}\right)^2\right) \le \left(1-\left(\frac{2j}{n'}\right)^2\right)^i.$$

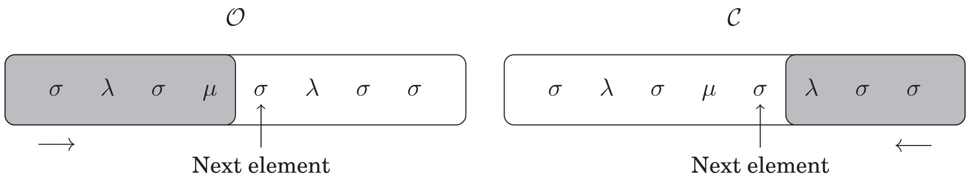
We substitute this bound into Equation (15). Bounding the rightmost sum of Equation (15) by an integral and using Maple®, we obtain

$$\begin{aligned}
\mathrm{E}(Z_{n'}) &= O\left(\frac{1}{n'}\right)\sum_{i=1}^{n'/2}\frac{n'+1}{\sqrt{i}(n'-2i+1)}\left(\int_0^{n'/2}\left(1-\left(\frac{2t}{n'}\right)^2\right)^i dt+1\right)\\
&= O\left(\frac{1}{n'}\right)\sum_{1\le i\le n'/2}\frac{n'+1}{\sqrt{i}(n'-2i+1)}\cdot\left(n'\cdot\frac{\Gamma(i+1)}{\Gamma(i+3/2)}+1\right),
\end{aligned}$$

involving the Gamma function $\Gamma(x) = \int_0^\infty t^{x-1}e^{-t}\,dt$. Since $\Gamma(i+1)/\Gamma(i+3/2) = \Theta(1/\sqrt{i})$, we may continue by calculating

$$\mathrm{E}(Z_{n'}) = O\left(\sum_{i=1}^{n'/2}\frac{n'+1}{i(n'-2i+1)}\right) = O\left(\sum_{i=1}^{n'/4}\frac{1}{i}+\sum_{i=n'/4+1}^{n'/2}\frac{1}{n'-2i+1}\right) = O(\log n').$$

Now we generalize the analysis to the case that the input contains medium elements. Let $(a_1,\ldots,a_n)$ be a random input. Omit all medium elements to obtain a reduced input $(a_1,\ldots,a_{n'})$ with $n' = s+\ell$. The additional cost term of Strategy $\mathcal{O}$ on the reduced input is the same as on the original input since medium elements influence neither the decisions of the strategy on elements of the reduced input nor the additional cost term. Starting from Equation (13), we may bound the difference between the additional cost term $\mathrm{E}(S_2+L_2)$ and the bound given in Equation (14) by the average over all s, ℓ of the



$\mathcal{O}$
σ λ σ μ σ λ σ σ
Next element
$\mathcal{C}$
σ λ σ μ σ λ σ σ
Next element

Fig. 3. Visualization of the decision process when inspecting an element using Strategy $\mathcal{O}$ (left) and $\mathcal{C}$ (right). Applying Strategy $\mathcal{O}$ from left to right uses the information that, of the remaining elements, three are small and one is large, so it decides that the element should be compared with p first. Applying Strategy $\mathcal{C}$ from right to left uses the information that, of the inspected elements, two were small and only one was large, so it decides to compare the element with p first, too. Note that the strategies would differ if, for example, the rightmost element were a medium element.

values k^*. This is bounded by the average over all s, ℓ of the values $Z_{n'}$; hence, by

$$\frac{1}{\binom{n}{2}}\left(2\sum_{\substack{s+\ell\leq n\\ \ell<s}}\ell+\sum_{\ell\leq n/2}\ell\right)-\mathrm{E}(S_2+L_2)\leq\frac{1}{\binom{n}{2}}\sum_{s+\ell\leq n-2}\mathrm{E}(Z_{s+\ell}\mid s,\ell)$$
$$=\frac{1}{\binom{n}{2}}\sum_{s+\ell\leq n-2}O(\log(s+\ell))=O(\log n).$$

Since $O(\log n)$ is in $O(n^{1-\varepsilon})$, the influence of these $O(\log n)$ terms to the total average sorting cost is bounded by $O(n)$; see Theorem 3.3 and Appendix A. □

Although making an algorithm with minimum ACT possible, the assumption that the exact number of small and large elements is known is of course not true for a real algorithm or for a fixed tree. We can, however, identify a real, algorithmic partitioning strategy whose ACT differs from the optimal one only by a logarithmic term. We study the following Strategy $\mathcal{C}$: *The comparison at node v is with the smaller pivot first if $s_v > l_v$; otherwise, it is with the larger pivot first.*

Strategy $\mathcal{O}$ looks into the future ("Are there more small elements or more large elements left?"), whereas Strategy $\mathcal{C}$ looks into the past ("Have I seen more small or more large elements so far?"). It is not hard to see that for some inputs the number of additional comparisons of Strategies $\mathcal{O}$ and $\mathcal{C}$ can differ significantly. Theorem 6.2 shows that, averaged over all possible inputs, however, there is only a small difference.

THEOREM 6.2. *Let $\mathrm{ACT}_{\mathcal{O}}$ (respectively, $\mathrm{ACT}_{\mathcal{C}}$) be the ACT for classifying n elements using Strategy $\mathcal{O}$ (respectively, $\mathcal{C}$). Then $\mathrm{ACT}_{\mathcal{C}} = \mathrm{ACT}_{\mathcal{O}} + O(\log n)$. When using $\mathcal{C}$ in a dual-pivot quicksort algorithm, we get $\mathrm{E}(C_n^{\mathcal{C}}) = 1.8n\ln n + O(n)$.*

PROOF. Assume that Strategy $\mathcal{O}$ inspects the elements in the order $a_{n-1},\ldots,a_2$, whereas $\mathcal{C}$ uses the order $a_2,\ldots,a_{n-1}$. If the strategies compare the element a_i to different pivots, then there are exactly as many small elements as there are large elements in $\{a_2,\ldots,a_{i-1}\}$ or $\{a_2,\ldots,a_i\}$, depending on whether i is even or odd; see Figure 3.

The same calculation as the proof of the previous theorem shows that $\mathrm{ACT}_{\mathcal{C}} - \mathrm{ACT}_{\mathcal{O}}$ is $O(\log n)$, which—as mentioned in the proof of the previous theorem—sums up to a total additive contribution of $O(n)$ when using Strategy $\mathcal{C}$ in a dual-pivot quicksort algorithm. □

Thus, dual-pivot quicksort with Strategy $\mathcal{C}$ has average cost at most $O(n)$ larger than dual-pivot quicksort using the (unrealistic) optimal Strategy $\mathcal{O}$.

We have identified two asymptotically optimal strategies ($\mathcal{SP}$ and $\mathcal{C}$) that can be used in an actual algorithm. Note that both strategies have an additional summand of $O(n)$. Unfortunately, the solution of the dual-pivot quicksort recurrence (cf. Theorem 3.3) does not give any information about the constant hidden in the $O(n)$ term. However, the average partitioning cost of Strategy $\mathcal{C}$ differs by only $O(\log n)$ from the partitioning cost of the optimal Strategy $\mathcal{O}$, whereas $\mathcal{SP}$ differs by $O(n^{1-\varepsilon})$. So one may be led to believe that $\mathcal{C}$ has a (noticeably) lower average comparison count for real-world input lengths. In Section 8, we will see that differences are clearly visible in the average comparison count measured in experiments. However, we will also see that the necessity for bookkeeping renders implementations of Strategy $\mathcal{C}$ impractical.

7. CHOOSING PIVOTS FROM A SAMPLE

In this section, we consider the variation of quicksort where the pivots are chosen from a small sample. Intuitively, this guarantees better pivots in the sense that the partition sizes are more balanced. For classical quicksort, the Median-of-k strategy is optimal with regard to minimizing the average comparison count [Martínez and Roura 2001], which means that the median in a sample of k elements is chosen as the pivot. The standard implementation of Yaroslavskiy's algorithm in Oracle's Java 7 uses an intuitive generalization of this strategy: It chooses the two tertiles in a sample of five elements as pivots.[7]

We will compare dual-pivot quicksort algorithms that use the two tertiles of the first five elements of the input as the two pivots with classical quicksort. Moreover, we will see that the optimal pivot choices for dual-pivot quicksort are not the two tertiles of a sample, but rather the elements of rank $k/4$ and $k/2$.

We remark that Wild, Nebel, and Martínez [Wild et al. 2014] provide a much more detailed study of pivot sampling in Yaroslavskiy's algorithm.

7.1. Choosing the Two Tertiles in a Sample of Size 5 as Pivots

We sort the first five elements and take the second and fourth elements as pivots. The probability that p and q, $p < q$, are chosen as pivots is exactly $(s \cdot m \cdot \ell)/\binom{n}{5}$. Following Hennequin [Hennequin 1991, pp. 52–53], for average partitioning cost $\mathrm{E}(P_n) = a \cdot n + O(1)$ we get

$$\mathrm{E}(C_n) = \frac{1}{H_6 - H_2} \cdot a \cdot n \ln n + O(n) = \frac{20}{19} \cdot a \cdot n \ln n + O(n), \tag{16}$$

where H_n denotes the n-th harmonic number.

When applying Lemma 3.1, we have average partitioning cost $a \cdot n + O(n^{1-\varepsilon})$. Using the same argument as in the proof of Theorem 3.3, the average comparison count becomes $20/19 \cdot a \cdot n \ln n + O(n)$ in this case. (This a special case of the much more general pivot sampling strategy that is described and analyzed in Wild et al. [2014, Theorem 6.2].)

We now investigate the effect on the average number of key comparisons in Yaroslavskiy's algorithm and the asymptotically optimal Strategy $\mathcal{SP}$ from Section 5. The average number of medium elements remains $(n-2)/3$. For Strategy $\mathcal{Y}$, we calculate (again using Maple®)

$$\mathrm{E}\big(P_n^{\mathcal{Y}}\big) = \frac{4}{3}n + \frac{1}{\binom{n}{5}} \sum_{s+\ell \le n-5} \frac{\ell \cdot (2s+m) \cdot s \cdot m \cdot \ell}{n-5} + O(n^{1-\varepsilon}) = \frac{34}{21}n + O(n^{1-\varepsilon}).$$

Applying Equation (16), we get $\mathrm{E}(C_n^{\mathcal{Y}}) = 1.704..n \ln n + O(n)$ key comparisons on average. (Note that Wild et al. [2013] calculated this leading coefficient as well.) This

[7]In an ordered set $S = \{x_1, \ldots, x_k\}$, the two tertiles are the elements of rank $\lceil k/3 \rceil$ and $\lceil 2k/3 \rceil$.

is slightly better than "clever quicksort", which uses the median of a sample of three elements as a single pivot element and achieves $1.714..n \ln n + O(n)$ key comparisons on average [van Emden 1970]. For Strategy $\mathcal{SP}$, we get (similarly as in Section 5)

$$\mathrm{E}\left(P_n^{\mathcal{SP}}\right) = \frac{4}{3}n + \frac{2}{\binom{n}{5}} \sum_{\substack{s+\ell \le n-5 \\ s \le \ell}} s \cdot s \cdot m \cdot \ell + O(n^{1-\varepsilon}) = \frac{37}{24}n + O(n^{1-\varepsilon}).$$

Again using Equation (16), we obtain $\mathrm{E}(C_n^{\mathcal{SP}}) = 1.623..n \ln n + O(n)$, improving further on the leading coefficient compared to clever quicksort and Yaroslavskiy's algorithm.

7.2. Pivot Sampling in Classical Quicksort and Dual-Pivot Quicksort

In the previous subsection, we showed that optimal dual-pivot quicksort using a sample of size 5 clearly beats clever quicksort, which uses the median of three elements. We now investigate how these two variants compare when the sample size grows.

The following proposition, which is a special case of Hennequin [1991, Proposition III.9 and Proposition III.10], will help in this discussion.

PROPOSITION 7.1. *Let $a \cdot n + O(1)$ be the average partitioning cost of a quicksort algorithm $\mathcal{A}$ that chooses the pivot(s) from a sample of size k, for constants a and k. Then the following holds:*

(1) *If $k+1$ is even and $\mathcal{A}$ is a classical quicksort variant that chooses the median of these k samples, then the average sorting cost is*

$$\frac{1}{H_{k+1} - H_{(k+1)/2}} \cdot a \cdot n \ln n + O(n).$$

(2) *If $k+1$ is divisible by 3 and $\mathcal{A}$ is a dual-pivot quicksort variant that chooses the two tertiles of these k samples as pivots, then the average sorting cost is*

$$\frac{1}{H_{k+1} - H_{(k+1)/3}} \cdot a \cdot n \ln n + O(n).$$

Note that for classical quicksort we have partitioning cost $n-1$. Thus, the average sorting cost becomes $\frac{1}{H_{k+1} - H_{(k+1)/2}} n \ln n + O(n)$.

For dual-pivot partitioning algorithms, the probability that p and q, $p < q$, are chosen as pivots in a sample of size k where $k+1$ is divisible by 3 is exactly

$$\frac{\binom{p-1}{(k-2)/3}\binom{q-p-1}{(k-2)/3}\binom{n-q}{(k-2)/3}}{\binom{n}{k}}.$$

Thus, the average partitioning cost $\mathrm{E}(P_{n,k}^{\mathcal{SP}})$ of Strategy $\mathcal{SP}$ using a sample of size k can be calculated as follows:

$$\mathrm{E}\left(P_{n,k}^{\mathcal{SP}}\right) = \frac{4}{3}n + \frac{2}{\binom{n}{k}} \sum_{s \le \ell} \binom{s}{(k-2)/3}\binom{m}{(k-2)/3}\binom{\ell}{(k-2)/3} \cdot s + O(n^{1-\varepsilon}). \tag{17}$$

Unfortunately, we could not find a closed form of $\mathrm{E}(P_{n,k}^{\mathcal{SP}})$. Some calculated values in which classical and dual-pivot quicksort with Strategy $\mathcal{SP}$ use the same sample size can be found in Table I. These values clearly indicate that starting from a sample of size 5 classical quicksort has a smaller average comparison count than dual-pivot

Table I. Comparison of the Leading Term of the Average Cost of Classical Quicksort and Dual-Pivot Quicksort for Specific Sample Sizes

Sample Size	5	11	17	41
Median (QS)	$1.622..n\ln n$	$1.531..n\ln n$	$1.501..n\ln n$	$1.468..n\ln n$
Tertiles (DP QS)	$1.623..n\ln n$	$1.545..n\ln n$	$1.523..n\ln n$	$1.504..n\ln n$

(Note that for real-world input sizes the linear term can make a big difference.)

quicksort.[8] This raises the question whether dual-pivot quicksort is inferior to classical quicksort using the Median-of-k strategy.

7.3. Optimal Segment Sizes for Dual-Pivot Quicksort

It is known from, for example, Martínez and Roura [2001] that for classical quicksort in which the pivot is chosen as the median of a fixed-sized sample, the leading term of the average comparison count converges with increasing sample size to the lower bound of $(1/\ln 2)\cdot n\ln n = 1.4426..n\ln n$. Wild, Nebel, and Martínez observed in Wild et al. [2014] that this is not the case for Yaroslavskiy's algorithm, which makes at least $1.4931..n\ln n - O(n)$ comparisons on average no matter how well the pivots are chosen. In this section, we show how to match the lower bound for comparison-based sorting algorithms with a dual-pivot approach.

We study the following setting, which was considered before in Martíez and Roura [2001] and Wild et al. [2014]. We assume that for a random input of n elements[9] we can choose (for free) two pivots with regard to a vector $\vec{\tau} = (\tau_1, \tau_2, \tau_3)$ such that the input contains exactly $\tau_1 n$ small elements, $\tau_2 n$ medium elements, and $\tau_3 n$ large elements. Furthermore, we consider the (simple) classification Strategy $\mathcal{L}$: *"Always compare with the larger pivot first."*

Lemma 7.2 says that this strategy achieves the minimum possible average comparison count for comparison-based sorting algorithms, $1.4426..n\ln n$, when setting $\tau = (\frac{1}{4}, \frac{1}{4}, \frac{1}{2})$.

Lemma 7.2. *Let $\vec{\tau} = (\tau_1, \tau_2, \tau_3)$ with $0 < \tau_i < 1$ and $\sum_i \tau_i = 1$, for $i \in \{1, 2, 3\}$. Assume that for each input size n we can choose two pivots such that there are exactly $\tau_1 \cdot n$ small, $\tau_2 \cdot n$ medium, and $\tau_3 \cdot n$ large elements. Then the comparison count of Strategy $\mathcal{L}$ is*[10]

$$p^{\vec{\tau}}(n) \sim \frac{1 + \tau_1 + \tau_2}{-\sum_{1\le i\le 3} \tau_i \ln \tau_i} n \ln n.$$

This value is minimized for $\vec{\tau}^* = (1/4, 1/4, 1/2)$ giving

$$p^{\vec{\tau}^*}(n) \sim \left(\frac{1}{\ln 2}\right) n \ln n = 1.4426..n\ln n.$$

Proof. On an input consisting of n elements, Strategy $\mathcal{L}$ makes $n + (\tau_1 + \tau_2)n$ comparisons. Thus, the comparison count of Strategy $\mathcal{L}$ follows the recurrence

$$p^{\vec{\tau}}(n) = n + (\tau_1 + \tau_2)n + p^{\vec{\tau}}(\tau_1 \cdot n) + p^{\vec{\tau}}(\tau_2 \cdot n) + p^{\vec{\tau}}(\tau_3 \cdot n).$$

[8]Note that this statement does not include that these two variants have different linear terms for the same sample size.
[9]We disregard the two pivots in the following discussion.
[10]Here, $f(n) \sim g(n)$ means that $\lim_{n\to\infty} f(n)/g(n) = 1$.

Table II. Overview of the Algorithms Considered in the Experiments

Abbreviation	Full Name	Strategy	Pseudocode
$\mathcal{QS}$	Classical Quicksort	—	e.g., [Wild and Nebel 2012, Algo. 1]
$\mathcal{Y}$	Yaroslavskiy's Algorithm	Section 4	Algorithm 2 (Page 29)
$\mathcal{L}$	Larger Pivot First	Section 4	Algorithm 3 (Page 30)
$\mathcal{S}$	Sedgewick's Algorithm (modified)	Section 4	Algorithm 5 (Page 32)
$\mathcal{SP}$	Sample Algorithm	Section 5	Algorithm 6 (Page 32)
$\mathcal{C}$	Counting Algorithm	Section 6	Algorithm 7 (Page 33)
$\mathcal{K}$	3-Pivot-Algorithm	—	[Kushagra et al. 2014, Algo. A.1.1]

Using the Discrete Master Theorem [Roura 2001, Theorem 2.3, Case (2.1)], we obtain the following solution for this recurrence:

$$p^{\vec{\tau}}(n) \sim \frac{1 + \tau_1 + \tau_2}{-\sum_{i=1}^{3} \tau_i \ln \tau_i} n \ln n.$$

Using Maple®, one finds that $p^{\vec{\tau}}$ is minimized for $\vec{\tau}^* = (\frac{1}{4}, \frac{1}{4}, \frac{1}{2})$, giving $p^{\vec{\tau}^*}(n) \sim 1.4426..n \ln n$. $\square$

The reason that Strategy $\mathcal{L}$ with this particular choice of pivots achieves the lower bound is simple: It makes (almost) the same comparisons as does classical quicksort using the median of the input as pivot. On an input of length n, Strategy $\mathcal{L}$ makes $3/2n$ key comparisons and then makes three recursive calls to inputs of length $n/4$, $n/4$, $n/2$. On an input of length n, classical quicksort using the median of the input as the pivot makes n comparisons to split the input into two subarrays of length $n/2$. Now consider only the recursive call on the left subarray. After $n/2$ comparisons, the input is split into two subarrays of size $n/4$ each. Now there remain two recursive calls on two subarrays of size $n/4$ and one recursive call on a subarray of size $n/2$ (the right subarray of the original input), as in Strategy $\mathcal{L}$. Since classical quicksort using the median of the input clearly makes $n \log n$ key comparisons, this bound must also hold for Strategy $\mathcal{L}$.

8. EXPERIMENTS

We implemented the methods presented in this article in C++ and Java. Our experiments were carried out on an Intel i7-2600 at 3.4GHz with 16GB RAM running Ubuntu 13.10 with kernel version 3.11.0. For compiling C++ code, we used *gcc* in version 4.8. For compiling Java code, we used Oracle's *Java 8 SDK*. All Java runtime tests were preceeded by a warmup phase for the just-in-time compiler (JIT), in which we let each algorithm sort 10,000 inputs of length 100,000.

For better orientation, the algorithms considered in this section are presented in Table II. Pseudocode for the dual-pivot methods is provided in Appendix B. In the following, we use a calligraphic letter both for the classification strategy and the actual dual-pivot quicksort algorithm.

In Section 8.1, we experimentally evaluate the average comparison count of the algorithms considered here. In Section 8.2, we focus on the actual running time needed to sort a given input. The charts of our experiments can be found at the end of this article.

8.1. The Average Comparison Count

We first have a look at the number of comparisons needed to sort a random input of up to 2^{29} integers. We did not switch to a different sorting algorithm (e.g., insertion sort) to sort short subarrays.

Figure 5 shows the results of our experiments for algorithms that choose the pivots directly from the input. We see that for practical values of n the lower order terms in the average comparison count have a big influence on the number of comparisons for all algorithms. (E.g., for Yaroslavskiy's algorithm this lower order term is dominated by the linear term $-2.46n$, as known from Wild and Nebel [2012].) Nevertheless, we may conclude that the theoretical studies on the average comparison count correspond to what can be observed in practice. Note that there is a difference between the optimal strategies $\mathcal{SP}$ and $\mathcal{C}$. We also see that the modified version of Sedgewick's algorithm beats Yaroslavskiy's algorithm, as calculated in Section 4.

Figure 6 shows the same experiment for algorithms that choose the pivots from a small sample. For dual-pivot quicksort variants, we used two different versions. Algorithms $\mathcal{Y}, \mathcal{SP}, \mathcal{C}, \mathcal{L}$ choose the tertiles of a sample of size 5 as pivots. $\mathcal{QS}$ is classical quicksort using the median of three strategy. $\mathcal{Y}'$ is Yaroslavskiy's algorithm with the tertiles of a sample of size 11 as the pivots; $\mathcal{L}'$ is algorithm $\mathcal{L}$ using the third- and sixth-largest element in a sample of size 11 as the pivots. This plot confirms the theoretical results from Section 7. Especially, the simple Strategy $\mathcal{L}$ beats Yaroslavskiy's algorithm for a sample of size 11 for the pivot choices introduced in Section 7.3.

8.2. Running Times

We now consider the running times of the algorithms for sorting a given input. We restrict our experiments to sorting random permutations of the integers $\{1, \ldots, n\}$. It remains for future work to compare the algorithms in more detail.

As a basis for comparison with other methods, we also include the three pivot quicksort algorithm described in Kushagra et al. [2014] (about 8% faster than Yaroslavskiy's algorithm in their setup), which we call $\mathcal{K}$. Similarly to classical quicksort, this algorithm uses two pointers to scan the input from left-to-right and right-to-left until these pointers cross. Classifications are done in a symmetrical way: First compare to the middle pivot; then (appropriately) compare either to the largest or smallest pivot. (In this way, each element is compared to exactly two pivots.) Elements are moved to a suitable position by the help of two auxiliary pointers. We note that the pseudocode from Kushagra et al. [2014, Algorithm A.1.1] uses multiple swaps when fewer assignments are sufficient to move elements. We show pseudocode of our implementation in Appendix C.

In our experiments, subarrays of size at most 16, 20, and 23 were sorted by insertion sort for classical quicksort, dual-pivot quicksort, and the three-pivot quicksort algorithm, respectively. Furthermore, Strategy $\mathcal{SP}$ uses Strategy $\mathcal{L}$ to sort inputs that contain no more than 1024 elements.

C++ Experiments. We first discuss our results on C++ code compiled with *gcc*-4.8. Since the used compiler flags might influence the observed running times, we considered four different compiler settings. In setting 1, we compiled the source code with *-O2*, in setting 2 with *-O2 -funroll-loops*, in setting 3 with *-O3*, and in setting 4 with *-O3 -funroll-loops*. The option *-funroll-loops* tells the compiler to unroll (the first few iterations of) a loop. In all settings, we used *-march=native*, which means that the compiler tries to optimize for the specific CPU architecture we use during compilation.

The experiments showed that there is no significant difference between the settings using *-O2* and those using *-O3*. So, we just focus on the first two compiler settings that use *-O2*. The running time results we obtained in these two settings are shown in Figures 7 and 8 at the end of this article. We first note that *loop unrolling* makes all algorithms behave slightly worse with respect to running times. Since a single innocuous-looking compiler flag may have such an impact on running

Table III. Comparison of the Actual Running Times of the Algorithms on 1,000 Different Inputs of Size 2^{27}

	$\mathcal{Y}$	$\mathcal{L}$	$\mathcal{K}$	$\mathcal{SP}$	$\mathcal{QS}$	$\mathcal{C}$
$\mathcal{Y}$	—	–/–/0.9%	–/1.0%/2.1%	6.4%/7.2%/8.0%	7.2%/8.3%/9.4%	13.9%/14.8%/15.8%
$\mathcal{L}$	–/–/0.8%	—	–/0.9%/2.1%	5.8%/7.0%/8.5%	7.0%/8.1%/9.5%	13.7%/14.6%/15.8%
$\mathcal{K}$	—	–/–/0.3%	—	4.7%/6.0%/7.6%	5.9%/7.2%/8.5%	12.3%/13.6%/14.9%
$\mathcal{SP}$	—	—	—	—	–/1.0%/2.4%	5.8%/7.1%/8.4%
$\mathcal{QS}$	—	—	—	—	—	4.7%/5.9%/7.2%

A table cell in row labelled "A" and column labelled "B" contains a string "$x\%/y\%/z\%$" and is read as follows: "In about 95%, 50%, and 5% of the cases, Algorithm A was more than x, y, and z percent faster than Algorithm B, respectively."

time, we stress that our results do not allow final statements about the running time behavior of quicksort variants. In the following, we restrict our evaluation to setting 1.

With respect to average running time, we get the following results (see Figure 7; the discussion uses the measurements obtained for inputs of size 2^{27}): Yaroslavskiy's algorithm is the fastest algorithm, but the difference to the dual-pivot algorithm $\mathcal{L}$ and the three-pivot algorithm $\mathcal{K}$ is negligible. The asymptotically comparison-optimal sampling algorithm $\mathcal{SP}$ cannot compete with these three algorithms with regard to running time. On average, it is about 7.2% slower than algorithm $\mathcal{Y}$. Classical quicksort is about 8.3% slower than $\mathcal{Y}$. The slowest algorithm is the counting algorithm $\mathcal{C}$; on average, it is about 14.8% slower than $\mathcal{Y}$.

Now we consider the significance of running time differences. For that, we let each algorithm sort the same 1,000 inputs containing 2^{27} items. In Table III, we consider the number of cases that support the hypothesis that an algorithm is a given percentage faster than another algorithm. The table shows that the difference in running time is about 1% smaller than the average suggested if we consider only significant running time differences (i.e., differences that were observed in at least 95% of the inputs). We conclude that there is no significant difference between the running times of the three fastest quicksort variants.

The good performance of the simple strategy "always compare to the larger pivot first" is especially interesting, because it is bad from a theoretical point of view: It makes $2n \ln n$ comparisons and $0.6n \ln n$ swaps on average. So, although it does not improve in both of these cost measures compared to classical quicksort, it is still faster. We may try to explain these differences in measured running times by looking at the average instruction count and the cache behavior of these algorithms. (The latter is motivated by the cost measure "cache misses" considered in Kushagra et al. [2014].) Table IV shows measurements of the *average total instruction count* and the *average number of L1/L2 cache misses*,[11] both in total and in relation to algorithm $\mathcal{K}$.

With respect to the average number of total instructions, we see that Strategy $\mathcal{K}$ needs the fewest instructions on average. It is about 1.2% better than algorithm $\mathcal{L}$. Furthermore, it is 7.1% better than the sampling algorithm $\mathcal{SP}$, about 11% better than Yaroslavskiy's algorithm, and about 12.7% better than classical quicksort. As expected, the theoretically optimal Strategy $\mathcal{C}$ needs by far the most instructions because of its necessity for bookkeeping. However, focusing exclusively on the total instruction count does not predict the observed running time behavior accurately. (In particular, $\mathcal{Y}$ should be slower than classical quicksort, which has also been observed in Wild et al. [2013] with respect to Oracle's Java 7.)

The authors of Kushagra et al. [2014] conjectured that another cost measure—the *average number of cache misses*—explains observed running time behavior. From

[11]Such statistics can be collected, e.g., by using the *performance application programming interface* from http://icl.cs.utk.edu/papi/.

Table IV. Average Number of Total Instructions and Cache Misses Scaled by n ln n for the Algorithms Considered in This Section

Algorithm	total #instructions	L1 misses	L2 misses
$\mathcal{QS}$	$10.58n\ln n$ (+12.7%)	$0.142n\ln n$ (+47.9%)	$0.030n\ln n$ (+178.1%)
$\mathcal{Y}$	$10.42n\ln n$ (+11.0%)	$0.111n\ln n$ (+15.6%)	$0.015n\ln n$ (+ 39.9%)
$\mathcal{SP}$	$10.05n\ln n$ (+ 7.1%)	$0.111n\ln n$ (+15.6%)	$0.014n\ln n$ (+ 31.7%)
$\mathcal{C}$	$14.08n\ln n$ (+50.1%)	$0.111n\ln n$ (+15.6%)	$0.012n\ln n$ (+ 11.8%)
$\mathcal{L}$	$9.50n\ln n$ (+ 1.2%)	$0.111n\ln n$ (+15.6%)	$0.016n\ln n$ (+ 49.0%)
$\mathcal{K}$	$9.38n\ln n$ (—)	$0.096n\ln n$ (—)	$0.011n\ln n$ (—)

In parentheses, these figures are set into relation to the values measured for Algorithm $\mathcal{K}$. (The relative difference has been calculated directly from the experimental data.) The figures were obtained by sorting 1,000 inputs of length 2^{27} separately by each algorithm using the compiler flag *-O2*.

Table IV, we see that Strategy $\mathcal{K}$ shows the best performance with respect to cache misses, confirming the theoretical results of Kushagra et al. [2014]. In our experiments, we observed that the relative difference of L1 cache misses among the 1-, 2-, and 3-pivot algorithms match the theoretical results from Kushagra et al. [2014]. (According to their calculations, dual-pivot variants should incur about 16% more cache misses, whereas classical quicksort should incur about 44% more cache misses.) This might explain why classical quicksort is the slowest algorithm, but it cannot explain why the dual-pivot algorithms $\mathcal{Y}$ and $\mathcal{L}$ can compete with the three-pivot algorithm.

With respect to L2 cache misses, we see that the cache misses are much higher than predicted. Compared to the three-pivot algorithm, classical quicksort incurs 178% more cache misses, and the worst dual-pivot algorithm with regard to L2 misses has 49% more cache misses. Since L2 misses have a much higher impact on the CPU cycles spent waiting for memory, this might amplify differences between these algorithms. In general, we note that only a fraction of about 1% of the instructions lead to cache misses. Consequently, one has to accurately measure the cycles spent waiting for memory in these situations. Investigating these issues is a possible direction for future work. With respect to other cost measures, Martínez, Nebel, and Wild [Martínez et al. 2015] analyzed branch misses in classical quicksort and Yaroslavskiy's algorithm. Their result is that the difference in the average number of branch mispredictions between these two algorithms is too small to explain differences in empirical running time. Based on our experimental measurements, we believe that this cost measure cannot explain running time differences between dual-pivot quicksort algorithms either.

We conclude that, in the light of our experiments, neither the average number of instructions nor the average number of cache misses can fully explain empirical running time.

Java Experiments. Figure 9 shows the running time measurements we got using Oracle's *Java 8*. Again, there is no significant difference among algorithms $\mathcal{K}$, $\mathcal{L}$, and $\mathcal{Y}$. The three-pivot quicksort algorithm $\mathcal{K}$ is the fastest algorithm using Java 8 on our setup. It is about 1.3% faster than Yaroslavskiy's algorithm on average. The optimal Strategy $\mathcal{C}$ is about 37% slower on average than algorithm $\mathcal{K}$. Classical quicksort and the sampling Strategy $\mathcal{SP}$ are about 7.7% slower on average.

9. CONCLUSION AND OPEN QUESTIONS

We have studied dual-pivot quicksort algorithms in a unified way and found optimal partitioning methods that minimize the average number of key comparisons up to $O(n)$. This minimum is $1.8n\ln n + O(n)$. We showed an optimal pivot choice for a simple dual-pivot quicksort algorithm and conducted an experimental study of the practicability of dual-pivot quicksort algorithms.

Several open questions remain. From a theoretical point of view, one should generalize our algorithms to the case that three or more pivots are used. At the moment, the optimal average comparison count for k-pivot quicksort is unknown for $k \geq 3$. This is the subject of ongoing work. From both a theoretical and an engineering point of view, the most important question is to find a cost measure that accurately predicts empirical running time behavior of variants of quicksort algorithms. Here, we could only provide evidence that neither standard measures like the average comparison count or the average swap count, nor empirical measures like the average instruction count or the cache behavior predict running time correctly when considered in isolation.

APPENDIXES

A. MISSING DETAILS OF THE PROOF OF THEOREM 3.3

Here we solve the recurrence given in Equation (3) for partitioning cost $\mathrm{E}(P_n)$ at most $K \cdot n^{1-\varepsilon}$. We use the Continuous Master Theorem of Roura [2001], whose statement we review first.

THEOREM A.1 ([ROURA 2001, THEOREM 18]). *Let F_n be recursively defined by*

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

where the toll function t_n satisfies $t_n \sim K n^{\alpha} \log^{\beta}(n)$ as $n \to \infty$ for constants $K \neq 0, \alpha \geq 0, \beta > -1$. Assume there exists a function $w : [0,1] \to \mathbb{R}$ such that

$$\sum_{j=0}^{n-1} \left| w_{n,j} - \int_{j/n}^{(j+1)/n} w(z)\, dz \right| = O(n^{-d}), \tag{18}$$

for a constant $d > 0$. Let $H := 1 - \int_0^1 z^{\alpha} w(z)\, dz$. Then we have the following cases:

(1) *If $H > 0$, then $F_n \sim t_n / H$.*
(2) *If $H = 0$, then $F_n \sim (t_n \ln n)/\hat{H}$, where*

$$\hat{H} := -(\beta + 1) \int_0^1 z^{\alpha} \ln(z) w(z)\, dz.$$

(3) *If $H < 0$, then $F_n \sim \Theta(n^c)$ for the unique $c \in \mathbb{R}$ with*

$$\int_0^1 z^c w(z)\, dz = 1.$$

We now solve recurrence (3) for $t_n = K \cdot n^{1-\varepsilon}$, for some $\varepsilon > 0$. First, observe that recurrence (3) has weights

$$w_{n,j} = \frac{6(n-j-1)}{n(n-1)}.$$

We define the shape function $w(z)$ as suggested in Roura [2001] by

$$w(z) = \lim_{n \to \infty} n \cdot w_{n,zn} = 6(1-z).$$

Now we have to check Equation (18) to see whether the shape function is suitable. We calculate:

$$\begin{aligned}\sum_{j=0}^{n-1}\left|w_{n,j}-\int_{j/n}^{(j+1)/n} w(z)\,dz\right|\\ &= 6\sum_{j=0}^{n-1}\left|\frac{n-j-1}{n(n-1)}-\int_{j/n}^{(j+1)/n}(1-z)\,dz\right|\\ &= 6\sum_{j=0}^{n-1}\left|\frac{n-j-1}{n(n-1)}+\frac{2j+1}{2n^2}-\frac{1}{n}\right|\\ &< 6\sum_{j=0}^{n-1}\left|\frac{1}{2n(n-1)}\right| = O(1/n).\end{aligned}$$

Thus, w is a suitable shape function. By observing that

$$H := 1-6\int_0^1 z^{1-\varepsilon}(1-z)\,dz < 0,$$

we conclude that the third case of Theorem A.1 applies for our recurrence. Consequently, we have to find the unique $c \in \mathbb{R}$ such that

$$6\int_0^1 z^{c}(1-z)\,dz = 1,$$

which is true for $c = 1$. Thus, an average partitioning cost of at most $K \cdot n^{1-\varepsilon}$ yields average sorting cost of $O(n)$.

B. DUAL-PIVOT QUICKSORT: ALGORITHMS IN DETAIL

B.1. General Setup

The general outline of a dual-pivot quicksort algorithm is presented as Algorithm 1.

ALGORITHM 1: Dual-Pivot-Quicksort (outline)

```
procedure Dual-Pivot-Quicksort(A, left, right)
 1: if $right - left \le THRESHOLD$ then
 2:     InsertionSort(A, left, right);
 3:     return ;
 4: if $A[right] > A[left]$ then
 5:     swap A[left] and A[right];
 6: p ← A[left];
 7: q ← A[right];
 8: partition(A, p, q, left, right, $pos_p$, $pos_q$);
 9: Dual-Pivot-Quicksort(A, left, $pos_p$ - 1);
10: Dual-Pivot-Quicksort(A, $pos_p$ + 1, $pos_q$ - 1);
11: Dual-Pivot-Quicksort(A, $pos_q$ + 1, right);
```

To get an actual algorithm, we have to implement a *partition* function that partitions the input as depicted in Figure 1. A partition procedure in this article has two output variables, pos_p and pos_q, that are used to return the positions of the two pivots in the partitioned array.

For moving elements around, we make use of the following two operations:

```
procedure rotate3($a, b, c$)
1: tmp $\leftarrow a$;
2: $a \leftarrow b$;
3: $b \leftarrow c$;
4: $c \leftarrow$ tmp;
```

```
procedure rotate4($a, b, c, d$)
1: tmp $\leftarrow a$;
2: $a \leftarrow b$;
3: $b \leftarrow c$;
4: $c \leftarrow d$;
5: $d \leftarrow$ tmp;
```

B.2. Yaroslavskiy's Partitioning Method

As mentioned in Section 4, Yaroslavskiy's algorithm makes sure that for ℓ large elements in the input, ℓ or $\ell - 1$ elements will be compared to the larger pivot first. How does it accomplish this? By default, it compares to the smaller pivot first, but for each large elements that it sees, it will compare the next element to the larger pivot first.

Algorithm 2 shows the partition step of (a slightly modified version of) Yaroslavskiy's algorithm. In contrast to the algorithm studied in Wild and Nebel [2012], it saves an index check at Line 8 and uses a `rotate3` operation to save assignments. (In our experiments, this makes Yaroslavskiy's algorithm about 4% faster.)

ALGORITHM 2: Yaroslavskiy's Partitioning Method

```
procedure Y-Partition($A, p, q, left, right, pos_p, pos_q$)
1: l $\leftarrow left + 1$; g $\leftarrow right - 1$; k $\leftarrow$ l;
2: while k $\leq$ g do
3:     if $A$[k] $< p$ then
4:         swap $A$[k] and $A$[l];
5:         l $\leftarrow$ l $+ 1$;
6:     else
7:         if $A$[k] $> q$ then
8:             while $A$[g] $> q$ do
9:                 g $\leftarrow$ g $- 1$;
10:            if k $<$ g then
11:                if $A$[g] $< p$ then
12:                    rotate3($A$[g], $A$[k], $A$[l]);
13:                    l $\leftarrow$ l $+ 1$;
14:                else
15:                    swap $A$[k] and $A$[g];
16:                g $\leftarrow$ g $- 1$;
17:    k $\leftarrow$ k $+ 1$;
18: swap $A[left]$ and $A$[l $- 1$];
19: swap $A[right]$ and $A$[g $+ 1$];
20: $pos_p \leftarrow$ l $- 1$; $pos_q \leftarrow$ g $+ 1$;
```

B.3. Algorithm Using "Always Compare to the Larger Pivot First"

Algorithm 3 presents an implementation of Strategy $\mathcal{L}$ (*"Always compare to the larger pivot first"*). Like Yaroslavskiy's algorithm, it uses three pointers into the array. One pointer is used to scan the array from left to right until a large element has been found (moving small elements to a correct position using the second pointer on the way). Subsequently, it scans the array from right to left using the third pointer until a non-large element has been found. These two elements are then placed into a correct position. This is repeated until the two pointers have crossed. The design goal is to check as rarely as possible if these two pointers have met since this event occurs infrequently.

ALGORITHM 3: Always Compare To Larger Pivot First Partitioning

procedure *L-Partition*$(A, p, q, left, right, pos_p, pos_q)$

```
 1: $i \leftarrow left + 1; k \leftarrow right - 1; j \leftarrow i;$
 2: while $j \le k$ do
 3:     while $q < A[k]$ do
 4:         $k \leftarrow k - 1;$
 5:     while $A[j] < q$ do
 6:         if $A[j] < p$ then
 7:             swap $A[i]$ and $A[j]$;
 8:             $i \leftarrow i + 1;$
 9:         $j \leftarrow j + 1;$
10:     if $j < k$ then
11:         if $A[k] > p$ then
12:             $rotate3(A[k], A[j], A[i]);$
13:             $i \leftarrow i + 1;$
14:         else
15:             swap $A[j]$ and $A[k]$;
16:         $k \leftarrow k - 1;$
17:     $j \leftarrow j + 1;$
18: swap $A[left]$ and $A[i - 1]$;
19: swap $A[right]$ and $A[k + 1]$;
20: $pos_p \leftarrow i - 1; pos_q \leftarrow k + 1;$
```

(In contrast, Yaroslavskiy's algorithm checks this for each element scanned by index k in Algorithm 2.)

This strategy makes $2n \ln n$ comparisons and $1.6n \ln n$ assignments on average.

B.4. Partitioning Methods Based on Sedgewick's Algorithm

Algorithm 4 shows Sedgewick's partitioning method as studied in Sedgewick [1975].

ALGORITHM 4: Sedgewick's Partitioning Method

procedure *S-Partition*$(A, p, q, left, right, pos_p, pos_q)$

```
 1: $i \leftarrow i_1 \leftarrow left; j \leftarrow j_1 := right;$
 2: while true do
 3:     $i \leftarrow i + 1;$
 4:     while $A[i] \le q$ do
 5:         if $i \ge j$ then
 6:             break outer while
 7:         if $A[i] < p$ then
 8:             $A[i_1] \leftarrow A[i]; i_1 \leftarrow i_1 + 1; A[i] \leftarrow A[i_1];$
 9:         $i \leftarrow i + 1;$
10:     $j \leftarrow j - 1;$
11:     while $A[j] \ge p$ do
12:         if $A[j] > q$ then
13:             $A[j_1] \leftarrow A[j]; j_1 \leftarrow j_1 - 1; A[j] \leftarrow A[j_1];$
14:         if $i \ge j$ then
15:             break outer while
16:         $j \leftarrow j - 1;$
17:     $A[i_1] \leftarrow A[j]; A[j_1] \leftarrow A[i];$
18:     $i_1 \leftarrow i_1 + 1; j_1 \leftarrow j_1 - 1;$
19:     $A[i] \leftarrow A[i_1]; A[j] \leftarrow A[j_1];$
20: $A[i_1] \leftarrow p; A[j_1] \leftarrow q;$
21: $pos_p \leftarrow i_1; pos_q \leftarrow j_1;$
```

Sedgewick's partitioning method uses two pointers, i and j, to scan through the input. It does not swap entries in the strict sense but instead has two "holes" at positions

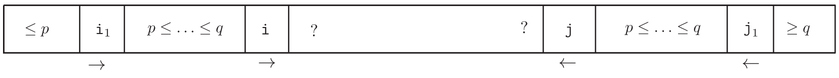


Fig. 4. An intermediate partitioning step in Sedgewick's algorithm.

$\mathtt{i}_1$ (respectively, $\mathtt{j}_1$) that can be filled with small (respectively, large) elements. "Moving a hole" is not a swap operation in the strict sense (three elements are involved), but requires the same amount of work as a swap operation (in which we have to save the content of a variable into a temporary variable [Sedgewick 1975]). An intermediate step in the partitioning algorithm is depicted in Figure 4.

The algorithm works as follows: Using i, it scans the input from left to right until it has found a large element, always comparing to the larger pivot first. Small elements found in this way are moved to a correct final position using the hole at array position $\mathtt{i}_1$. Subsequently, using j, it scans the input from right to left until it has found a small element, always comparing to the smaller pivot first. Large elements found in this way are moved to a correct final position using the hole at array position $\mathtt{j}_1$. Now it exchanges the two elements at positions i (respectively, j) and continues until i and j have met.

Algorithm 5 shows an implementation of the modified partitioning strategy from Section 4. In the same way as Algorithm 4, it scans the input from left to right until it has found a large element. However, it uses the smaller pivot for the first comparison in this part. Subsequently, it scans the input from right to left until it has found a small element. Here, it uses the larger pivot for the first comparison.

ALGORITHM 5: Sedgewick's Partitioning Method, modified

procedure *S2-Partition*(A, p, q, *left*, *right*, pos_p, pos_q)

```
1:  $i \leftarrow i_1 \leftarrow left; j \leftarrow j_1 := right;$
2:  while true do
3:      $i \leftarrow i + 1;$
4:      while true do
5:          if $i \geq j$ then
6:              break outer while
7:          if $A[i] < p$ then
8:              $A[i_1] \leftarrow A[i]; i_1 \leftarrow i_1 + 1; A[i] \leftarrow A[i_1];$
9:          else if $A[i] > q$ then
10:             break inner while
11:         $i \leftarrow i + 1;$
12:     $j \leftarrow j - 1;$
13:     while true do
14:         if $A[j] > q$ then
15:             $A[j_1] \leftarrow A[j]; j_1 \leftarrow j_1 - 1; A[j] \leftarrow A[j_1];$
16:         else if $A[j] < p$ then
17:             break inner while
18:         if $i \geq j$ then
19:             break outer while
20:         $j \leftarrow j - 1;$
21:     $A[i_1] \leftarrow A[j]; A[j_1] \leftarrow A[i];$
22:     $i_1 \leftarrow i_1 + 1; j_1 \leftarrow j_1 - 1;$
23:     $A[i] \leftarrow A[i_1]; A[j] \leftarrow A[j_1];$
24: $A[i_1] \leftarrow p; A[j_1] \leftarrow q;$
25: $pos_p \leftarrow i_1; pos_q \leftarrow j_1;$
```

B.5. Algorithms for the Sampling Partitioning Method

The sampling method $\mathcal{SP}$ from Section 5 uses a mix of two classification algorithms: *"Always compare to the smaller pivot first"* and *"Always compare to the larger pivot* *first"*. The actual partitioning method uses Algorithm 3 for the first $n_s = n/1024$ classifications and then decides which pivot should be used for the first comparison in the remaining input. (This is done by comparing the two variables `i` and `k` in Algorithm 3.) If there are more large elements than small elements in the sample, it continues using Algorithm 3; otherwise, it uses Algorithm 6. If the input contains fewer than 1,024 elements, Algorithm 3 is used directly.

ALGORITHM 6: Simple Partitioning Method (smaller pivot first)

procedure *SimplePartitionSmall*$(A, p, q, \mathit{left}, \mathit{right}, \mathit{pos}_p, \mathit{pos}_q)$

```
1:  $l \leftarrow \mathit{left} + 1$; $g \leftarrow \mathit{right} - 1$; $k \leftarrow l$;
2:  while $k \leq g$ do
3:      if $A[k] < p$ then
4:          swap $A[k]$ and $A[l]$;
5:          $l \leftarrow l + 1$;
6:          $k \leftarrow k + 1$;
7:      else
8:          if $A[k] < q$ then
9:              $k \leftarrow k + 1$;
10:         else
11:             swap $A[k]$ and $A[g]$
12:             $g \leftarrow g - 1$;
13: swap $A[\mathit{left}]$ and $A[l - 1]$;
14: swap $A[\mathit{right}]$ and $A[g + 1]$;
15: $\mathit{pos}_p \leftarrow l - 1$; $\mathit{pos}_q \leftarrow g + 1$;
```

ALGORITHM 7: Counting Strategy $\mathcal{C}$

procedure *C-Partition*$(A, p, q, \mathit{left}, \mathit{right}, \mathit{pos}_p, \mathit{pos}_q)$

```
1:  $i \leftarrow \mathit{left} + 1$; $k \leftarrow \mathit{right} - 1$; $j \leftarrow i$;
2:  $d \leftarrow 0$;        ▷ d holds the difference of the number of small and large elements.
3:  while $j \leq k$ do
4:      if $d > 0$ then
5:          if $A[j] < p$ then
6:              swap $A[i]$ and $A[j]$;
7:              $i \leftarrow i + 1$; $j \leftarrow j + 1$; $d \leftarrow d + 1$;
8:          else
9:              if $A[j] < q$ then
10:                 $j \leftarrow j + 1$;
11:             else
12:                 swap $A[j]$ and $A[k]$;
13:                 $k \leftarrow k - 1$; $d \leftarrow d - 1$;
14:     else
15:         while $A[k] > q$ do
16:             $k \leftarrow k - 1$; $d \leftarrow d - 1$;
17:         if $j \leq k$ then
18:             if $A[k] < p$ then
19:                 rotate3$(A[k], A[j], A[i])$;
20:                 $i \leftarrow i + 1$; $d \leftarrow d + 1$;
21:             else
22:                 swap $A[j]$ and $A[k]$;
23:             $j \leftarrow j + 1$;
24: swap $A[\mathit{left}]$ and $A[i - 1]$;
25: swap $A[\mathit{right}]$ and $A[k + 1]$;
26: $\mathit{pos}_p \leftarrow i - 1$; $\mathit{pos}_q \leftarrow k + 1$;
```

B.6. Algorithm for the Counting Strategy

Algorithm 7 is an implementation of the counting strategy from Section 6. It uses a variable `d` that stores the difference of the number of small elements and the number of large elements that have been classified so far. On average, this algorithm makes $1.8n \ln n + O(n)$ comparisons and $1.6n \ln n$ assignments.

C. A FAST THREE-PIVOT ALGORITHM

We give here the complete pseudocode for the three-pivot algorithm described in Kushagra et al. [2014]. In contrast to the pseudocode given there [Kushagra et al. 2014, Algorithm A.1.1], we removed two unnecessary bound checks (Line 5 and Line 10 in our code), and we move misplaced elements in Lines 15–29 using less assignments. This is used in the implementation of Kushagra et al. [2014], as well.[12] On average, this algorithm makes $1.846..n \ln n + O(n)$ comparisons and $1.57..n \ln n + O(n)$ assignments.

ALGORITHM 8: Symmetric Three-Pivot Sorting Algorithm

```
procedure 3-Pivot(A, left, right)
Require: $right - left \geq 2, A[left] \leq A[left+1] \leq A[right]$
 1: $p_1 \leftarrow A[left]; p_2 \leftarrow A[left+1]; p_3 \leftarrow A[right]$;
 2: $i \leftarrow left + 2; j \leftarrow i; k \leftarrow right - 1; l \leftarrow k$;
 3: while $j \leq k$ do
 4:     while $A[j] < p_2$ do
 5:         if $A[j] < p_1$ then
 6:             swap $A[i]$ and $A[j]$;
 7:             $i \leftarrow i + 1$;
 8:         $j \leftarrow j + 1$;
 9:     while $A[k] > p_2$ do
10:         if $A[k] > p_3$ then
11:             swap $A[k]$ and $A[l]$;
12:             $l \leftarrow l - 1$;
13:         $k \leftarrow k - 1$;
14:     if $j \leq k$ then
15:         if $A[j] > p_3$ then
16:             if $A[k] < p_1$ then
17:                 rotate4($A[j], A[i], A[k], A[l]$);
18:                 $i \leftarrow i + 1$;
19:             else
20:                 rotate3($A[j], A[k], A[l]$);
21:             $l \leftarrow l - 1$;
22:         else
23:             if $A[k] < p_1$ then
24:                 rotate3($A[j], A[i], A[k]$);
25:                 $i \leftarrow i + 1$;
26:             else
27:                 swap $A[j]$ and $A[k]$;
28:         $j \leftarrow j + 1; k \leftarrow k - 1$;
29: rotate3($A[left+1], A[i-1], A[j-1]$);
30: swap $A[left]$ and $A[i-2]$;
31: swap $A[right]$ and $A[l-1]$;
32: 3-Pivot($A$, $left$, $i - 3$);
33: 3-Pivot($A$, $i - 1$, $j - 2$);
34: 3-Pivot($A$, $j$, $l$);
35: 3-Pivot($A$, $l + 2$, $right$);
```

[12]Code made available by Alejandro López-Ortiz.

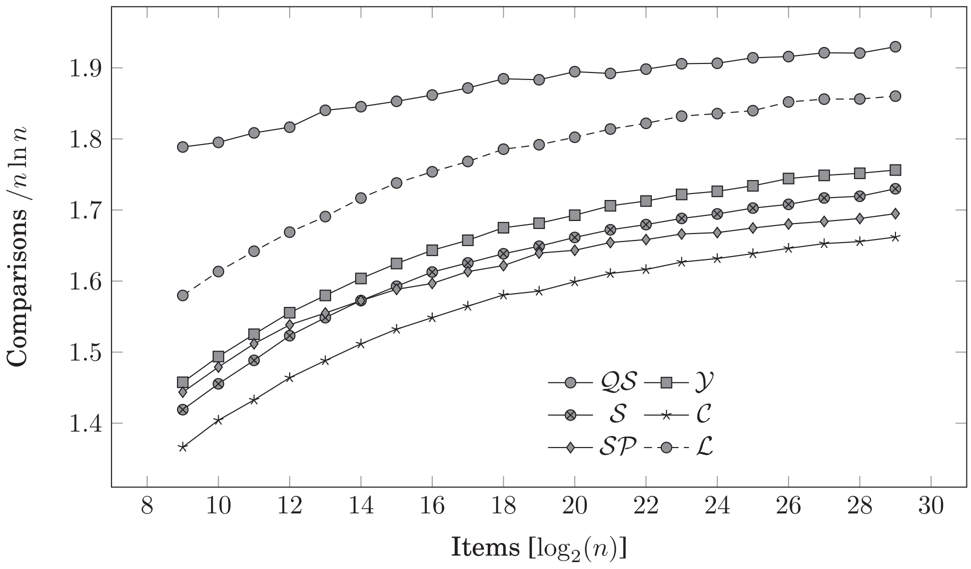


Fig. 5. Average comparison count (scaled by $n \ln n$) needed to sort a random input of up to $n = 2^{29}$ integers. We compare classical quicksort ($\mathcal{QS}$), Yaroslavskiy's algorithm ($\mathcal{Y}$), the sampling algorithm ($\mathcal{SP}$), the counting algorithm ($\mathcal{C}$), the modified version of Sedgewick's algorithm ($\mathcal{S}$), and algorithm $\mathcal{L}$. Each data point is the average over 400 trials.

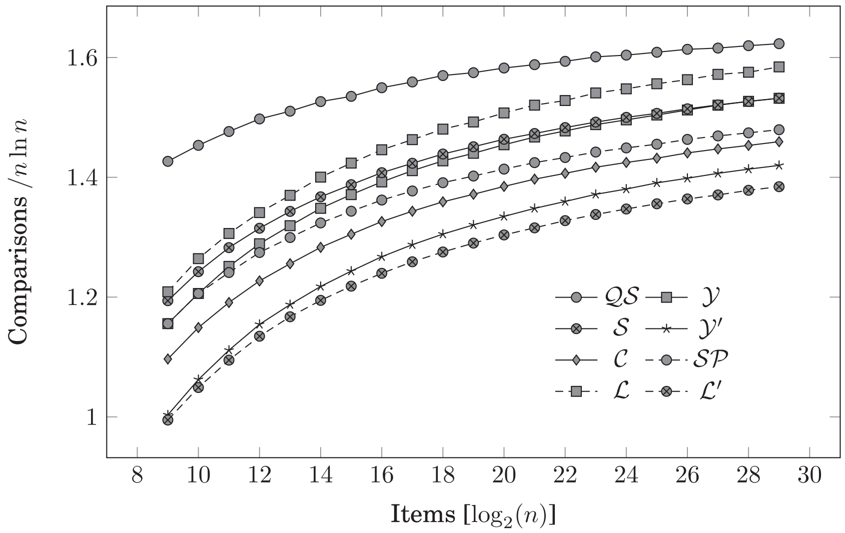


Fig. 6. Average comparison count (scaled by $n \ln n$) needed to sort a random input of up to $n = 2^{29}$ integers. We compare classical quicksort ($\mathcal{QS}$) with the Median-of-3 strategy, Yaroslavskiy's algorithm ($\mathcal{Y}$), the sampling algorithm ($\mathcal{SP}$), the counting algorithm ($\mathcal{C}$), the modified version of Sedgewick's algorithm from Section 4 ($\mathcal{S}$), and algorithm $\mathcal{L}$. Each of these dual-pivot algorithms uses the tertiles in a sample of size 5 as the two pivots. Moreover, $\mathcal{L}'$ is an implementation of Strategy $\mathcal{L}$ that uses the third- and sixth-largest element from a sample of size 11. $\mathcal{Y}'$ is Yaroslavskiy's algorithm choosing the tertiles of a sample of size 11 as the two pivots. Each data point is the average over 400 trials.

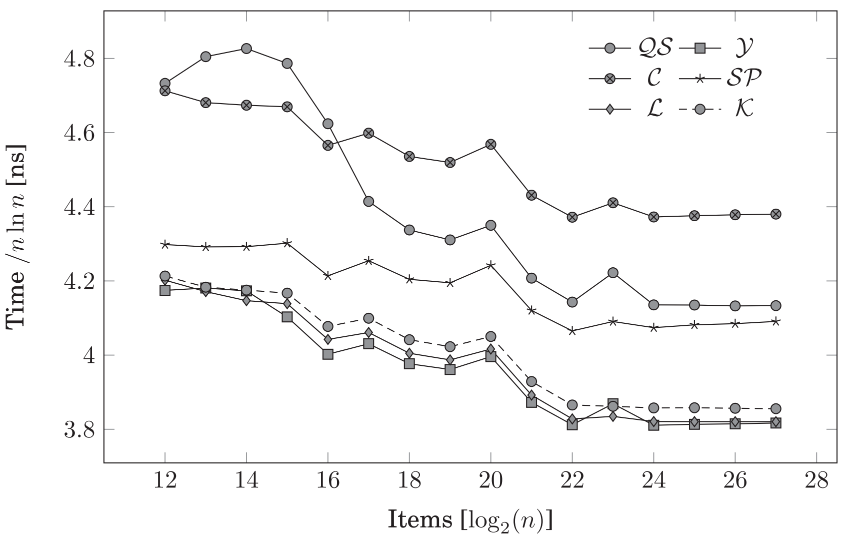


Fig. 7. Running time experiments in C++, setting 1 with compiler flags: *-O2*. Each data point is the average over 1,000 trials. Times are scaled by $n \ln n$.

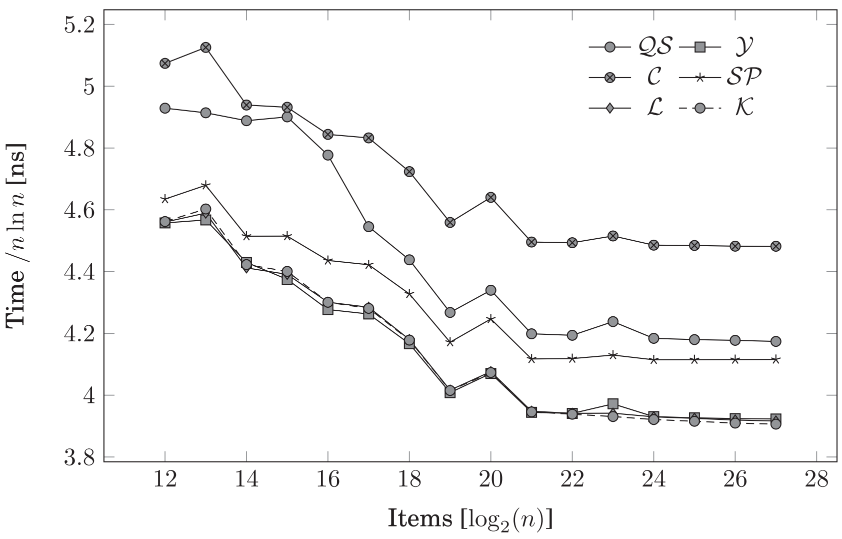


Fig. 8. Running time experiments in C++, setting 2 with compiler flags: *-O2 -funroll-loops*. Each data point is the average over 1,000 trials. Times are scaled by $n \ln n$.

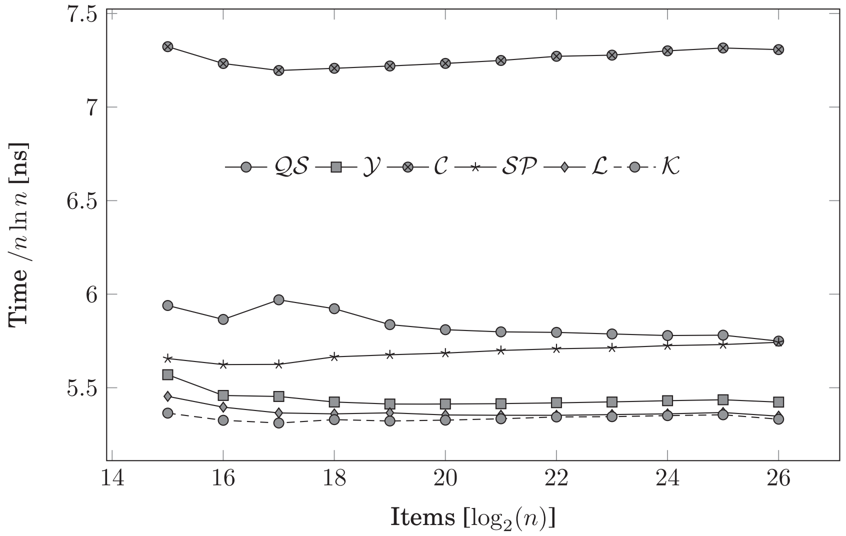


Fig. 9. Running time experiments in Java 8. For warming up the JIT, we let each algorithm sort 10,000 inputs consisting of 100,000 elements. Each data point is the average over 500 trials. Times are scaled by $n \ln n$.

ACKNOWLEDGMENTS

The authors thank Thomas Hotz for a useful suggestion regarding the optimal strategy of Section 6 and Pascal Klaue for implementing the algorithms and carrying out initial experiments. The first author thanks Timo Bingmann, Michael Rink, and Sebastian Wild for interesting and useful discussions. We thank the referees of the conference and journal submissions for their insightful comments, which helped greatly in improving the presentation. We especially thank the referee who pointed out a way of showing that the lower order term of most algorithms considered in this article is $O(n)$, not only $o(n \ln n)$.

REFERENCES

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms (3rd ed.)*. MIT Press.

Devdatt P. Dubhashi and Alessandro Panconesi. 2009. *Concentration of Measure for the Analysis of Randomized Algorithms*. Cambridge University Press.

Pascal Hennequin. 1991. *Analyse en moyenne d'algorithmes: tri rapide et arbres de recherche*. Ph.D. Dissertation. Ecole Politechnique, Palaiseau. http://www-lor.int-evry.fr/~pascal/.

C. A. R. Hoare. 1962. Quicksort. *Computer Journal* 5, 1 (1962), 10–15.

Donald E. Knuth. 1973. *The Art of Computer Programming, Volume III: Sorting and Searching*. Addison-Wesley.

Shrinu Kushagra, Alejandro López-Ortiz, Aurick Qiao, and J. Ian Munro. 2014. Multi-pivot quicksort: Theory and experiments. In *Proceedings of the 16th Meeting on Algorithms Engineering and Experiments (ALENEX'14)*. SIAM, 47–60.

Conrado Martínez, Markus E. Nebel, and Sebastian Wild. 2015. Analysis of branch misses in quicksort. In *Proceedings of the 12th Meeting on Analytic Algorithmics and Combinatorics (ANALCO'15)*. 114–128.

Conrado Martínez and Salvador Roura. 2001. Optimal sampling strategies in quicksort and quickselect. *SIAM Journal on Computing* 31, 3 (2001), 683–705.

Salvador Roura. 2001. Improved master theorems for divide-and-conquer recurrences. *Journal of the ACM* 48, 2 (2001), 170–205.

Robert Sedgewick. 1975. *Quicksort*. Ph.D. Dissertation. Standford University.

Robert Sedgewick. 1977. Quicksort with equal keys. *SIAM J. Comput.* 6, 2 (1977), 240–268.

Robert Sedgewick and Philippe Flajolet. 1996. *An Introduction to the Analysis of Algorithms*. Addison-Wesley-Longman.

M. H. van Emden. 1970. Increasing the efficiency of quicksort. *Communications of the ACM* 13, 9 (1970), 563–567.

Sebastian Wild. 2013. *Java 7's Dual Pivot Quicksort*. Master's thesis. University of Kaiserslautern.

Sebastian Wild and Markus E. Nebel. 2012. Average case analysis of Java 7's dual pivot quicksort. In *Proceedings of the 20th European Symposium on Algorithms (ESA'12)*. 825–836.

Sebastian Wild, Markus E. Nebel, and Conrado Martínez. 2014. Analysis of pivot sampling in dual-pivot quicksort. *CoRR* abs/1412.0193 (2014).

Sebastian Wild, Markus E. Nebel, and Ralph Neininger. 2015. Average case and distributional analysis of dual-pivot quicksort. *ACM Transactions on Algorithms* 11, 3 (2015), 22.

Sebastian Wild, Markus E. Nebel, Raphael Reitzig, and Ulrich Laube. 2013. Engineering Java 7's dual pivot quicksort using MaLiJAn. In *Proceedings of the 15th Meeting on Algorithms Engineering and Experiments (ALENEX'13)*. 55–69.

Received July 2014; revised February 2015; accepted March 2015